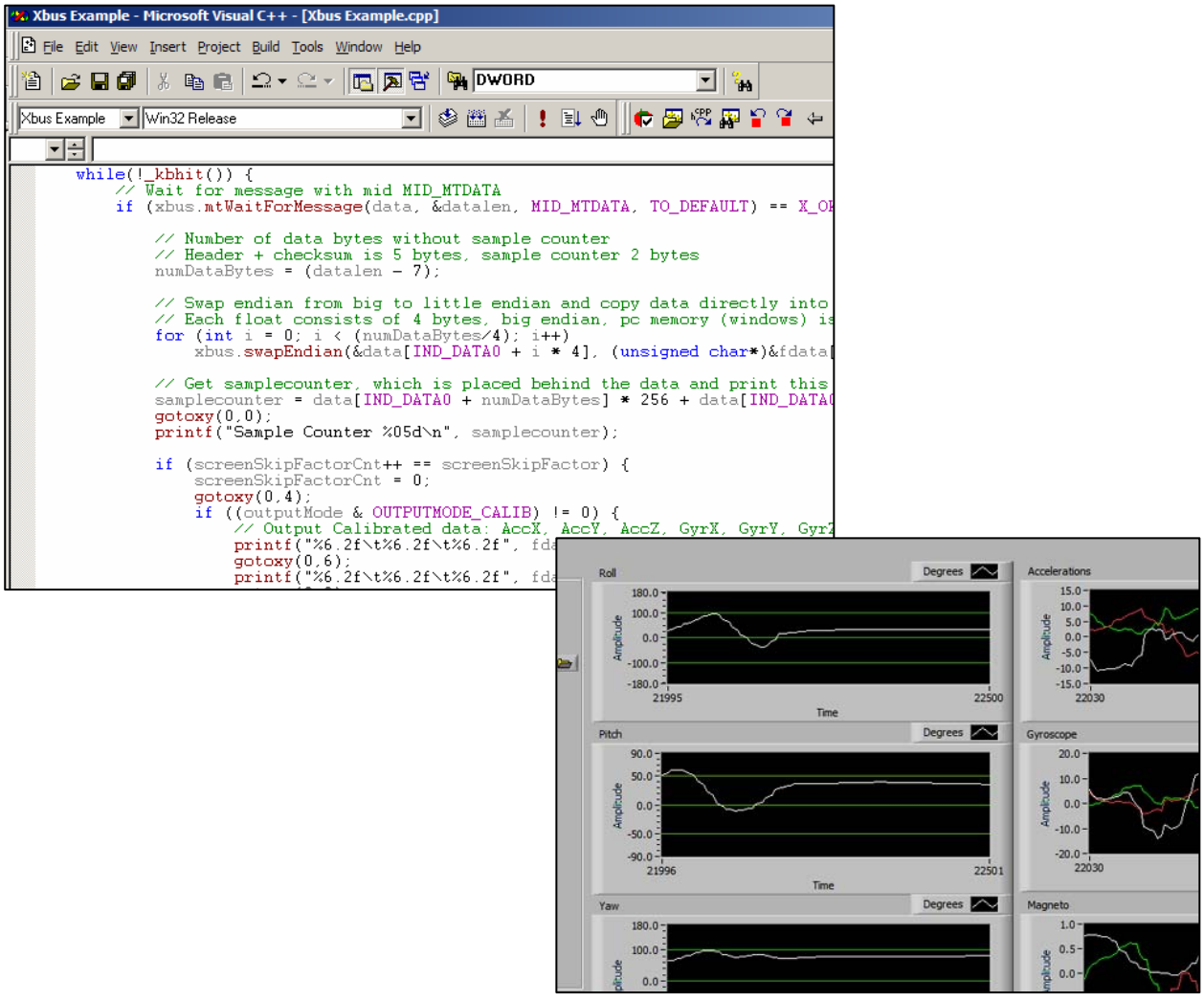
MT Software Development Kit Documentation



Document MT0200P
Revision A, June 1st 2005

Xsens Technologies B.V.
Capitool 50
P.O. Box 545
7500 AM Enschede
The Netherlands

phone +31-(0)53-4836444
fax +31-(0)53-4836445
e-mail support@xsens.com
internet www.xsens.com

Revisions

Revision	Date	By	Changes
A	June 2nd 2005	PR	First version

Table of Contents

1 Introduction

This document treats two different ways of interfacing with your MTi or MTx. First, direct interfacing on the serial COM port is discussed, utilizing the Xbus C++ communication class that is supplied with the SDK. Then the MT Object (Windows COM API) is discussed, which creates an extra abstraction level between the MTi / MTx and your software application, this is the major part of this documentation. The details of the underlying communication protocol that is implemented in the Xbus C++ class is discussed in the **MTi and MTx Low-level communication protocol documentation.**

Both approaches have their advantages and they will be discussed in the following sections.

1.1 Low-level direct serial port (RS-232/422) interfacing

Direct interfacing with the MTi or MTx is the natural choice if you are looking for full-control, maximum flexibility and/or have hard real-time performance requirements. The MTi/MTx's low power embedded DSP does all the calculations/calibration, you just retrieve the data from the serial COM-port using the MTi/MTx binary communication protocol using streaming (free-running) mode or request mode. This part is made easy for you by the inclusion of the source code (C++) of the "Xbus" communication class in the MT SDK. Example C/C++ application code should get you quickly started on your development platform of choice.

The Xbus C++ class that implements the low-level communication protocol is discussed in Chapter 2.

Applies to: *Any* (RT)OS or processor platform! (C/C++)

1.2 Interface through COM-object API (MT Object)

If you want to develop a software application that uses the MTi or MTx you can consider using the COM-object API (MTObj.DLL) which provides easy to use function calls to obtain data from the sensor or to change settings. The COM-object takes care of the hardware communication interfacing and it is an easy way to get (soft) real-time performance. Both polling and events based methods are supported.

The advantage is that all Windows software that support the Component Object Model (COM), including MATLAB, LabVIEW, Excel, etc. can easily be used, without the need for C/C++ programming.

Chapters 3 and forward describes the COM-object API.

Applies to: Windows PC platform

***NOTE*:** Using the MT Object provides backwards compatibility of your software developed for the MT9-B.

2 The Xbus communication class

2.1 Introduction

The Xbus[1] C++ class included in the Software Development Kit makes low-level communication with the MTi/MTx, MT9-B and Xbus Master in your own C++ application easy. The Xbus C++ class implements all the messages in the binary Xbus communication protocol that is used by Xsens' devices. It also implements functions to read/write to serial COM-ports. The C++ code is well commented and should get you up and running in your on C/C++ application very quickly.

The Xbus C++ class is especially useful when using the MTi or MTx. Since, the internal digital signal processor of the MTi / MTx calculates the 3D orientation data and calibrated inertial data and transmits this data over RS-232/422 in the mode you request, you can use the Xbus class to easily obtain this data directly in your application. Using the Xbus class, you do not need to deal directly with the serial port, buffering, and the specifics of the Xbus binary messages. On the other hand, the source code is available and can be modified to suit the needs of your platform, OS, and application needs.

The Xbus class is implemented in the **Xbus.h** and **Xbus.cpp** files in the Xbus class directory. The header file contains Message IDs of the Xbus protocol, headers of the most common messages and the declaration of the Xbus class.

The Xbus C++ class is tested on Windows and Linux.

For more information on the specific details on the RS-232/422 low-level communication please refer to the **MTi and MTx Low-level Communication Documentation**.

2.2 Function descriptions

The Xbus class has four sections of functions which will be shortly described below. The more detailed function descriptions can be found in the Xbus.cpp file in the comments associated with each individual function decalration.

- Low level general functions
  - **clock_t clockms** – Return the processor time in milliseconds

- Low level serial COM port functions
  - **initSerialPort** – Opens and initialize the serial port
  - **isPortOpen** – Returns protected boolean indicating if the COM port is open
  - **writeSerialPort** – Writes characters to serial port
  - **readSerialPort** – Reads characters from serial port
  - **flushSerialPort** – Remove any 'old' data in COM port buffer
  - **closeSerialPort** – Close serial port

[1] The Xsens Bus communication protocol (Xbus) is a fully documented standard message based protocol developed by Xsens tailor made for the needs of inertial sensors.

    - **setSerialPortQueueSize** – Set input and output buffer size of serial port

- Basics – These functions use the Xbus protocol to read complete Xbus messages from the serial port.
    - **mtWaitForMessage** – Wait for a certain message for a certain time
    - **mtReadMessage** – Read a message from the serial port
    - **mtReadRawString** – Read raw data from serial port for a certain time
    - **mtSendMessage** – Send a message to the serial port using the MSG_ defines
    - **mtSendMessageEx** – Send a message to the serial port using the MID_ defines
    - **mtSendRawString** – Send raw data to the serial port
    - **mtSendAndGetResponse** – Send a message to the serial port and wait for any response from the attached device
    - **mtCheckAndGetSetting** – Use this function to set a setting of the attached device. The function first checks if the setting is already at the desired value and if needed, sets the new setting.
    - **calcChecksum** – Calculate checksum of message
    - **checkChecksum** – Check if checksum of received message is correct
    - **swapEndian** – Convert 2 or 4 bytes data from little to big endian or back

- Useful functions – Use the basic messages to perform some standard actions
    - **mtGotoConfig** – Puts attached device in Config State
    - **mtGotoMeasurement** – Puts attached device in Measurement State

2.3 Example

An example of how to use the Xbus class can be found in the example directory with the Xbus Class. This program illustrates the basic usages of the Xbus class and can be found in "**Xbus Example.cpp**". It uses a screen skip factor, to update the data on the screen with a lower refresh rate than the MTi/MTx. This is pure cosmetic and makes the screen look smoother.

The file structures is as follows:

- Includes and global variables, including the Xbus class.
- Common functions – Used for console screen manipulation
    - gotoxy
    - clrscr
- getUserInputs – Queries user for MTi/MTx settings to use
- doMTSettings – Puts MTi/MTx in user selected mode using Xbus class
- writeHeaders – Writes table headers to console screen
- main
    - Gets user inputs using getUserInputs
    - Opens serial port using Xbus class
    - Puts MTi/MTx in correct mode using doMTSettings
    - Wait for a "MTData" message by using the Xbus class function mtWaitForMessage.
    - Converts read data and writes selected data to the screen in correct format
    - Closes Xbus class serial port when done

For more information, check the source code, all functionality is well documented.

3 Application Programming Interfaces of the MT Object

3.1 Introduction MT Object

The functionality of the MTi and MTx, but also MT9-B and the Xbus (all supported device together called "**MT**", for MotionTracker) are made available to software developers through a software component; COM-object for Windows platform and a shared object for Linux platform. After installing the software component you can access the MT device functionality with simple function-calls in your own code.

This chapter describes how to use the MT Object software component, which interfaces are available, and how to use the sample source-code included in the SDK.

NOTE: The sections on the Xbus system and the Linux shared object are not (yet) applicable to the MTi and MTx.

The MT Software Development Kit consists of a COM-object, which can be used on MS Windows operating systems, and a shared object, which can be used on Linux operating systems. This documentation is based on the Windows version, but the functionality between the two versions is identical. The differences between the Linux version and the Windows version are described in chapter 6. The integration with MATLAB, LabVIEW, etc. is made possible through the COM-interfaces and is therefore only supported on the Windows platform.

MS Windows 2000/XP and Debian GNU/Linux and Red Hat are officially supported. Please contact support@xsens.com for other platforms or custom developments.

3.2 Overview

The MT software component is implemented in a COM object (for a general introduction to COM-objects, see Appendix A). When using the MT Object, the user does not have to deal with subjects like data IO (serial communication). This functionality is all implemented in the COM object. All the user needs are the pointers to the functions of the MT object (see Figure 1).

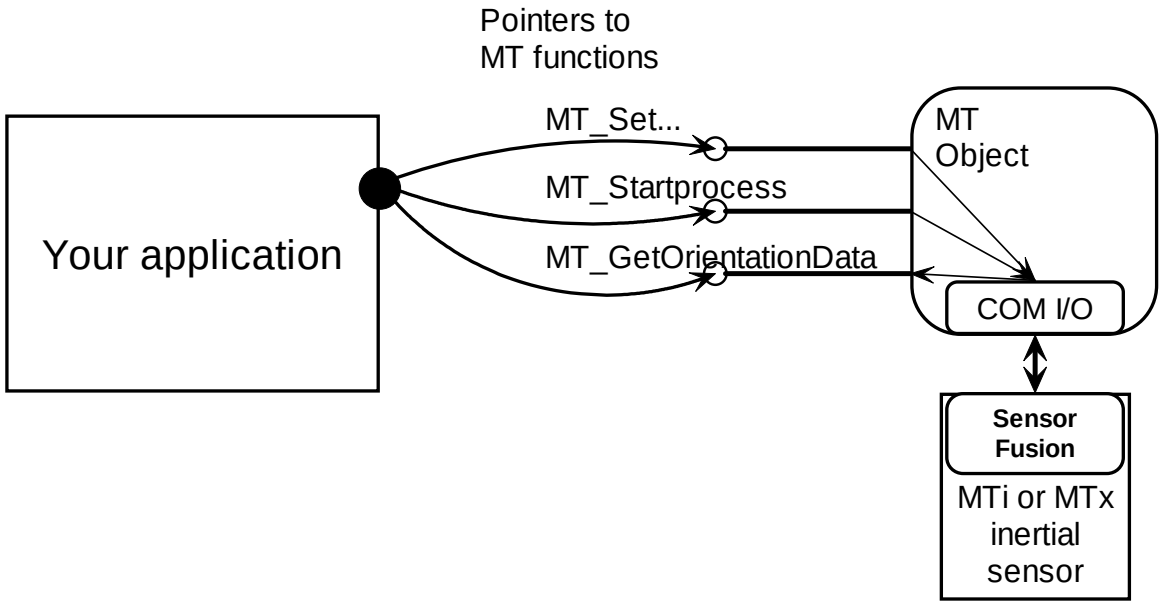


Figure 1 - Representation of data flow when using MT Object with MTi/MTx

The MT Object has the following inputs and outputs. The functions needed to manipulate them are listed in section 3.5 and further.

Inputs:

- *short*: COM port number
- *short*: output mode, format of orientation data returned by the MT Object
- *float, short, float*: gain, correction interval, weighting factor
- *short*: Do AMD. Use Adapt To Magnetic Disturbances filter
- *float*: Inclination

Outputs:

- *VARIANT*: orientation data (quaternion, Euler angles or rotation matrix)
- *VARIANT*: calibrated sensor data (3D acceleration, rate of turn, magnetic field)
- *Event*: notification when new orientation values are calculated (Call to clients sink object)
- *Float, integer, float*: gain, correction interval and weighting factor
- *Short*: Error code

3.3 Overview using Xbus system

The MT software component is implemented in a COM object (for a general introduction to COM-objects, see Appendix A). When using the MT object, the user does not have to deal with subjects like data IO (serial communication), communication protocol of the Xbus and Xbus Master and the implementation of an algorithm to calculate 3D orientation from 3D rate of turn, 3D accelerations and 3D earth magnetic field data. This functionality is all implemented in the COM object. All the user needs are the pointers to the functions of the MT Object (see Figure 1).

NOTE: The MTi/MTx is not yet supported by the Xbus system but are intended for standalone use.

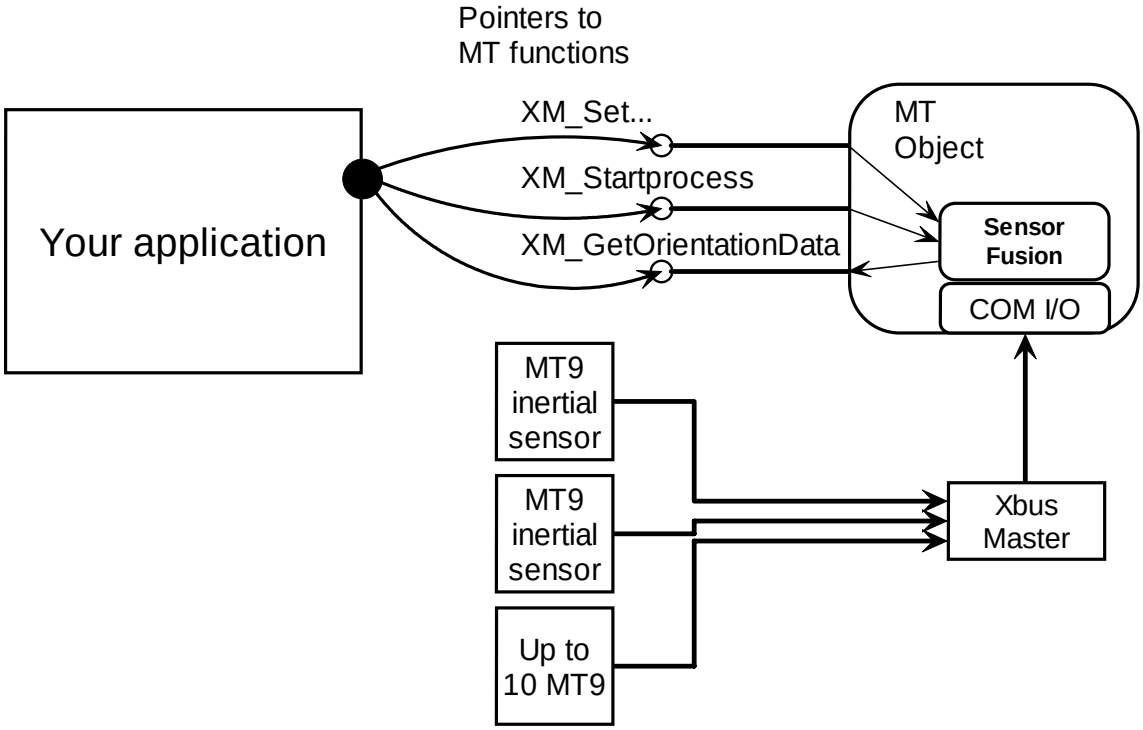
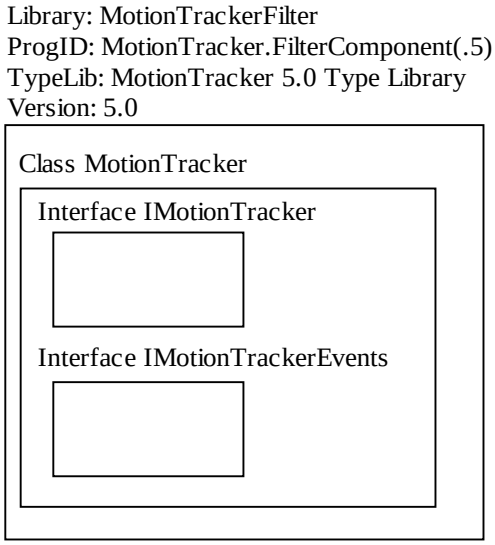


Figure 2 - Representation of data flow when using MT Object with the Xbus Master

The inputs and outputs are basically the same as without the Xbus Master. Manipulation functions are listed in section 3.5.2 and further.

3.4 IDispatch interface to applications on Windows

The MT Object supports the IDispatch interface. This enables it to be used in other languages like Visual Basic, Matlab 6.5 or Labview 6.0i and higher. The interface IProvideClassInfo is also available for use in conjunction with IDispatch. With the use of IDispatch, some naming schemes can be confusing. Officially, the following are the correct names for all parts of the object.



MATLAB and VBA use the ProgID for identification, Labview uses the TypeLib and then creates a name based on the library name and an interface name.

3.5 Input functions – Set() functions

The MT Object accepts several different inputs which enable the user to use different settings, change output modes, etc. Most settings do not need to be changed but can be left at their default value for normal operation.

Normally there is no need to call any of the MT_Setxxx functions if the MTi or MTx is connected to COM 1 (default). When using the MTi/MTx, most settings are stored in the device non-volatile memory. These settings are not stored directly, but are only stored permanently after a call to MT_SaveToMTS or during MT_StartProcess. The settings stored on the sensor are the settings set with the MT_SetFilterSettings, MT_SetDoAMD and the MT_SetMotionTracker* functions.

All the functionality of the MTi/MTx is not supported in the MT Object, but all basic functions now based on the MT9 functionality; however some functions are not yet supported: MT_SetInputPacket(B), MT_SetMTSData, MT_GetMTSData.

Inputs must be set with the following functions, which are defined in the IMotionTracker interface.

```
MT_SetOutputMode(short nMode);
```

nMode:
- 0 – Quaternion
- 1 – Euler
- 2 – Rotation Matrix

Default: 0 (Quaternion)
Functionality: Set the type of output required from the MT Object

```
MT_SetCalibratedOutput(short nEnabled);
```

nEnabled:
- 0 – Disabled
- 1 – Enabled

Default: 0 (Disabled)
Functionality: Set if the MT Object should return calibrated sensor data

```
MT_SetCOMPort(short nPort , int nBaudrate);
```

nPort: port number starting at 1 for first COM port
Default: 1
nBaudrate [optional]: Baudrate at which COM port should be opened. (And at which the sensor is running)
Default: 115200
Functionality: Set to which COM port the MotionTracker is connected, -1 when using post processing (See chapter 4.5)

```
MT_SetTimeout(short nTimeout);
```

Default: 3 (seconds).
Functionality: Set the timeout period. The time before an error is returned when the connection is lost. Can be set from 1 to 120 seconds.

```
MT_SetFilterSettings(float fGain, short nCorInterval, float fRho);
```

Default: 1.0 (gain), 1 (correction interval), 1.0 (weighting factor)
Functionality: Set the filter's gain, correction interval and weighting factor.

```
MT_SetDoAMD(short nDoAMD);
```

nDoAMD:
0 – Disabled
1 – Enabled
Default: 0 (disabled).
Functionality: Set if this adaptive filter should be used to correct for magnetic disturbance.
NOTE: This option is under development, performance cannot be guaranteed.

```
MT_SetSampleFrequency(short nSampleFreq);
```

nSampleFreq:
100 – 100 Hz Sample Frequency
250 – 250 Hz Sample Frequency
Default: 100
Functionality: Set the Sample Frequency of the MT used with the MT Object.
IMPORTANT: Only needed when using MT9-A.

```
MT_SetInputPacket(BSTR bstrInputPacket);
```

bstrInputPacket: BSTR string containing complete input packet starting with preamble (0xFAAF).
Functionality: Post processing of data with to the MT Object.
Caution: When post-processing data, make sure not to call MT_SetInputPacket before a new orientation was retrieved with MT_GetOrientationData. Otherwise data will be lost and incorrect orientations will be calculated.
IMPORTANT: Only needed when using MT9-A.

```
MT_SetxmuLocation(BSTR bstrName);
```

bstrName: wide, double-byte (Unicode) string (See for conversion below)
No Default !
Functionality: Set the location of the .XMU file containing the calibration values for the used sensor.
IMPORTANT: Only needed when using MT9-A.

3.5.1 Input Functions applicable to MT9-B and MTi / MTx

These functions can be used with the MT9-B, MTi and MTx devices (MT).

```
MT_SetTimeStampOutput(short nEnabled);
```

nEnabled:
0 – Disabled
1 – Enabled
Default: 0 (Disabled)
Functionality: Set if the MT Object should return timestamp data from the MotionTracker. Timestamp will be placed before the orientation data and calibrated data. Timestamp values are in seconds, with 3 digits resolution (1 ms).

```
MT_SetMotionTrackerBaudrate(int nBaudrate);
```

nBaudrate: MotionTracker Baudrate. Can be one of the following:
460800 – 460k8
230400 – 230k4
115200 – 115k2
57600 – 57k6
Functionality: Set the baudrate at which the MotionTracker should be sending its data.
NOTE: Call to MT_SaveToMTS function to save into device non-volatile memory is needed to activate new baudrate setting!

```
MT_SetMotionTrackerSampleFrequency(int nSampleFrequency);
```

nSampleFrequency: Sample Frequency required. Can be one of the following:
25, 50, 100, 200, 256, 320, 400, 512 Hz
Functionality: Set the frequency at which the MotionTracker should be sampling its data.
NOTE: Sample frequencies above 100 Hz are not yet supported for MTi and MTx. Contact support@xsens.com for details.
NOTE2: Call to MT_SaveToMTS function to save into device non-volatile memory is needed to activate new sample frequency!

```
MT_SetMotionTrackerHeading(float fHeading);
```

fHeading: Set the required heading. See MT Technical Documentation for more information on the heading.
Functionality: Set the heading in respect to which the orientation will be calculated.
NOTE: Call to MT_SaveToMTS function to save into device non-volatile memory is needed to activate the new heading reference setting!

```
MT_SetMotionTrackerLocation(short nLocation);
```

nLocation: Indicator for the location of the sensor. Can be a number from 0 to 255.
Functionality: The user can set a location ID, so each ID can represent a certain encoded location (e.g. 0=upper arm 1=lower arm etc.). This value is not used be the MT9 Software or SDK.
NOTE: Call to MT_SaveToMTS function to save into device non-volatile memory is needed to activate location setting!

```
MT_SaveToMTS(BSTR bstrDeviceID);
```

bstrDeviceID: wide, double-byte (Unicode) string containing the device ID of the sensor (Can be obtained with the MT_QueryMotionTrackerB function).
Functionality: Store new values, set with the above 4 functions, into MotionTracker non-volatile memory.

```
MT_SetInputPacketB(BSTR bstrInputPacket);
```

bstrInputPacket: BSTR string containing complete input packet starting with preamble (0xFA), BID and MID(0x32).
Functionality: Post processing of data with to the MT Object.

Caution: When post-processing data, make sure not to call MT_SetInputPacket before a new orientation was retrieved with MT_GetOrientationData. Otherwise data will be lost and incorrect orientations will be calculated.

```
MT_SetMTSData(BSTR bstrMTSData0, BSTR bstrMTSData1, BSTR
bstrMTSData2);
```

bstrMTSData0: When bstrMTSData1 and bstrMTSData2 are 0:
BSTR string containing complete MTSData packet as obtained from MT9 Software log or from MT_GetMTSData function.
Otherwise:
bstrMTSData0, bstrMTSData1 and bstrMTSData2 contain the MTS data in 3 subparts as obtained when requested from the MT9-B. (See MTx-B Technical Documentation or C++ Sample code)
Functionality: Pass MTS data to MT Object when using the post processing of data.
Caution: Call this function before MT_StartProcess is called.

3.5.2 Input Functions using Xbus system

When using the MT9-B there is no need to call any of the MT_Setxxx functions if the Xbus Master is connected to COM 1 (default). The electronic datasheet containing calibration values of the MT9-B (the MotionTracker Specification (MTS) data) are stored in non-volatile memory embedded in the MT9-B and are automatically fetched by the COM-object.

If you are using the MT9-A's on the Xbus, the minimum input required is to point the COM-object where to find the appropriate MT9 calibration values (*.XMU file), and the attached COM port, if not connected to COM 1.

Inputs must be set with the following functions, which are defined in the IMotionTracker interface.

```
XM_SetCOMPort(short nPort, int nBaudrate);
```

nPort: port number starting at 1 for first COM port
Default: 1
nBaudrate [optional]: Baudrate at which COM port should be opened. (And at which the sensor is running)
Default: 115200
Functionality: Set to which COM port the Xbus Master is connected

```
XM_SetTimeStampOutput(short nEnabled);
```

nEnabled:
0 – Disabled
1 – Enabled
Default: 0 (Disabled)
Functionality: Set if the MT Object should return timestamp data from the Xbus Master. Timestamp will be placed before the orientation data and calibrated data. Timestamp values are in seconds, with 3 digits resolution (1 ms).

```
XM_SetOutputMode(short nMode);
```

nMode:
0 – Quaternion
1 – Euler
2 – Rotation Matrix
Default: 0 (Quaternion)
Functionality: Set the type of output required from the MT Object

```
XM_SetXbusMasterBaudrate(short nSampleFreq);
```

nSampleFreq: Baudrate. Can be one of the following:
460800 – 460k8
230400 – 230k4
115200 – 115k2
57600 – 57k6
Functionality: Set the baudrate at which the Xbus Master should be sending its data.
NOTE: Saved immediately into Xbus Master non-volatile memory.

```
XM_SetXbusMasterSampleFrequency(short nSampleFreq);
```

nSampleFreq: Sample Frequency required. Can be one of the following:
25, 50, 100, 200, 256, 320, 400, 512 Hz
Default: 100
Functionality: Set the frequency at which the Xbus Master should be sampling its data.
NOTE: Saved immediately into Xbus Master non-volatile memory.

```
XM_SetCalibratedOutput(short nEnabled);
```

nEnabled:
0 – Disabled
1 – Enabled
Default: 0 (Disabled)
Functionality: Set if the MT Object should return calibrated sensor data

```
XM_SetxmuLocation(BSTR bstrDeviceID, BSTR bstrName);
```

bstrDeviceID: wide, double-byte (Unicode) string containing the device ID for which a corresponding XMU file location is entered in the second parameter.
bstrName: wide, double-byte (Unicode) string (See for conversion below). Must contain the full path to the XMU file corresponding to the first parameter
No Default !
Functionality: Set the location of the .XMU file containing the calibration values for the used sensor
IMPORTANT: Only needed when using MT9-A. The electronic datasheet containing calibration values of the **MT9-B** (the MotionTracker Specification (MTS) data) are stored in non-volatile memory embedded in the device and are automatically fetched by the COM-object if needed.

```
XM_SetFilterSettings(BSTR bstrSensorID, float fGain, short nCorInterval, float fRho);
```

bstrDeviceID: wide, double-byte (Unicode) string containing the device ID for which the parameters are meant.
Default: 1.0 (gain), 1 (correction interval), 1.0 (weighting factor)
Functionality: Set the filter's gain, correction interval and weighting factor.

```
XM_SetTimeout(short nTimeout);
```

Default: 3 (seconds).
Functionality: Set the timeout period. The time before an error is returned when the connection is lost. Can be set from 1 to 120 seconds.

```
XM_SetDoAMD(BSTR bstrDeviceID, short nDoAMD);
```

bstrDeviceID: wide, double-byte (Unicode) string containing the device ID for which the short in the second applies.
nDoAMD:

0 – Disabled
1 – Enabled

Default: 0 (disabled).
Functionality: Set if this adaptive filter should be used.

3.5.3 Input Functions using MT9-B with Xbus system

```
XM_SetMotionTrackerHeading(BSTR bstrDeviceID, float fHeading);
```

bstrDeviceID: wide, double-byte (Unicode) string containing the device ID of the sensor
fHeading: Set the required heading. See MT9 Technical Documentation for more information on the heading.
Functionality: Set the heading in respect to which the orientation will be calculated.
NOTE: Call to MT_SaveToMTS function to save into device non-volatile memory is needed to activate the new heading reference setting!

```
XM_SetMotionTrackerLocation(BSTR bstrDeviceID, short nLocation);
```

bstrDeviceID: wide, double-byte (Unicode) string containing the device ID of the sensor
nLocation: Indicator for the location of the sensor. Can be a number from 0 to 255.
Functionality: The user can set a location ID, so each ID can represent a certain location. This value is not used be the MT9 Software or SDK. Call the MT_SaveToMTS function to save into MotionTracker non-volatile memory.

```
XM_SaveToMTS(BSTR bstrDeviceID);
```

bstrDeviceID: wide, double-byte (Unicode) string containing the device ID of the sensor (Can be obtained with the MT_QueryMotionTrackerB function).
Functionality: Store new values, set with the above 4 functions, into MotionTracker non-volatile memory.

3.6 Output Functions – Get() functions

To retrieve data from the MT Object, the following functions are available, which are defined in the IMotionTracker interface.

```
MT_GetOrientationData(short *nRetVal, VARIANT *pfOutputArray, short nLatest)
```

nRetVal :

0 – No new data available (No Error)
1 – New data available
2 – No data on COM port
3 – No device ID received from sensor
4 – Incomplete data received (Connection lost)
5 – Checksum error in data from sensor
6 – COM port could not be opened

When using MT9-B

7 – MTS data not available in file input (Not set with MT_SetMTSData)

When using MT9-A

7 – XMU file with calibration data could not be read

pfOutputArray: contains orientation data (See chapter 5 below for retrieving the data)

nLatest: Optional parameter indicating whether to use the internal output buffer or not. See chapter 3.8.2, for more information.

0 – Use the buffer. This way all data can be retrieved from the object with consecutive calls to MT_GetOrientationData.
1 – Always return the latest calculated orientation data.

The VARIANT always contains a SAFEARRAY with the orientation data calculated by the orientation estimation filter. The SAFEARRAY is constructed with VARIANT data types VT_ARRAY | VT_R4. Depending on chosen output mode (default is quaternion) these are 4 (quaternion), 3 (Euler angles) or 9 (rotation matrix) floating point numbers.

See chapter 5 and the MT User Manual for more details on the different output options.

Functionality: Function used to query the MT Object for new orientation data. The nRetVal parameter contains the objects answer. If the answer is 1 (New Data) the second parameter contains a pointer to the new orientation data

```
MT_GetCalibratedData(short *nRetVal, VARIANT *pfOutputArray, short nLatest);
```

nRetVal :

0 – No new data available (No Error)
1 – New data available
2 – No data on COM port
3 – No device ID received from sensor
4 – Incomplete data received (Connection lost)
5 – Checksum error in data from sensor
6 – COM port could not be opened

When using MT9-B

7 – MTS data not available in file input (Not set with MT_SetMTSData)

When using MT9-A

7 – XMU file with calibration data could not be read

pfOutputArray contains calibrated sensor data

nLatest: Optional parameter indicating whether to use the internal output buffer or not. See chapter 3.8.2 about polling for more information.

0 – Use the buffer. This way all data can be retrieved from the object with consecutive calls to MT_GetOrientationData.
1 – Always return the latest calculated orientation data.

The VARIANT always contains a SAFEARRAY with the calibrated sensor data including the temperature from the internal sensor.
The SAFEARRAY is constructed with VARIANT data types VT_ARRAY | VT_R4 and contains 10 values, 9 for the calibrated data (3D acceleration, 3D rate of turn and 3D magnetic field) and 1 for the temperature.

See chapter 5 and the MT User Manual for more details on the different output options.

Functionality: This function queries the MT Object for new calibrated data. The nRetVal parameter contains the objects answer. If the answer is 1 (New Data) the second parameter contains a pointer to the new calibrated data

```
MT_GetFilterSettings(float *fGain, short *nCorInterval, float *fRho);
```

Functionality: Retrieve the current filter settings (gain, correction interval and weighting factor).

```
MT_StartProcess();
```

Functionality: Start the processing of MT data by the MT Object.
Note: Returns when all initialization internally is done. If COM port could not be opened, returns E_ACCESSDENIED (General access denied error)

```
MT_StopProcess();
```

Functionality: Stop the processing of MT data by the MT Object.

```
MT_ResetOrientation(short nResetType, short bSaveAfterStop);
```

nResetType: Type of reset to be done
- 0 – Heading reset
- 1 – Global reset (MT_SaveToMTS not available)
- 2 – Object reset
- 3 – Align (Object followed by Heading reset)

bSaveAfterStop: If 1, the heading and object reset values are stored in the MotionTracker non-volatile memory. This way, next time you use the sensor with our software, it will always be in this coordinate system. When the MT_StopProcess function has been called, the reset values of the reset which was called with this set to 1 are stored in the MotionTracker.
Default: 0, Do no save to memory
Functionality: Reset the reference orientation in the MotionTracker algorithm.

Please, refer to the MT User Manual for more details on the different co-ordinate system resets available.

3.6.1 Output functions applicable to the MT9-B and MTi / MTx

```
MT_QueryMotionTrackerB(BSTR * bstrDeviceID);
```

bstrDeviceID: wide, double-byte (Unicode) string containing the device ID of attached sensor.
Functionality: Query the attached MotionTracker for its device ID.

```
MT_GetMotionTrackerBaudrate(int *nBaudrate);
```

nBaudrate: Requested baudrate. Can be one of the following:
- 460800 – 460k8
- 230400 – 230k4
- 115200 – 115k2
- 57600 – 57k6

Functionality: Get the baudrate at which the MT is sending its data.

```
MT_GetMotionTrackerSampleFrequency(int *nSampleFrequency);
```

nSampleFrequency: Requested Sample Frequency. Can be one of the following:
- 25, 50, 100, 200, 256, 320, 400, 512 Hz

Functionality: Get the frequency at which the MT is updating its output data.

```
MT_GetMotionTrackerHeading(float *fHeading);
```

fHeading: Requested heading.
Functionality: Get the heading in respect to which the orientation will be calculated.

```
MT_GetMotionTrackerLocation(short *nLocation);
```

nLocation: Indicator for the location of the sensor. Can be a number from 0 to 255.
Functionality: The user can get a location ID, so each ID can represent a certain location. This value is not used be the MT Software or SDK.

```
MT_GetMTSData(BSTR *bstrMTSData);
```

bstrMTSData: BSTR string containing 173 bytes of MTS data
Functionality: The user can get the MTS data for use with post-processing of data.

Caution: Only works in C++ or similar environment. Not IDispatch compatible.

3.6.2 Output Functions using Xbus system

```
XM_QueryXbusMaster(short *nNumSensors, BSTR * bstrDeviceID0,
                   BSTR * bstrDeviceID1, BSTR * bstrDeviceID2,
                   BSTR * bstrDeviceID3, BSTR * bstrDeviceID4);
```

nNumSensors: Number of sensors found by the Xbus Master.
bstrDeviceID0-5: wide, double-byte (Unicode) string containing the device ID of attached sensors.
Functionality: Query the Xbus Master for attached sensors. The first return value contains the number of sensors attached. The 2nd till 6th parameter contains a device ID for each attached sensor.

```
XM_QueryXbusMasterB(short *nNumSensors, BSTR * bstrDeviceIDs);
```

nNumSensors: Number of sensors found by the Xbus Master.
bstrDeviceIDs: wide, double-byte (Unicode) string containing the device IDs of attached sensors.
Functionality: Query the Xbus Master for attached sensors. The first return value contains the number of sensors attached. The 2nd parameter contains a device IDs for each attached sensor. Each Device ID is 8 characters long (For Example 0000701A)

To retrieve data from the MT Object, the following functions are available, which are defined in the IMotionTracker interface.

```
XM_GetOrientationData(short *nRetVal, VARIANT *pfOutputArray, short nLatest);
```

nRetVal :

0 – No new data calculated (No Error)
1 – New data available
2 – No data on COM port
3 – No device ID received from sensor
4 – Incomplete data received (Connection lost)
5 – Checksum error in data from sensor

6 – COM port could not be opened
7 – XMU file with calibration data could not be read (not applicable for MT9-B)

pfOutputArray: contains orientation data (See chapter 5 below for retrieving the data)
nLatest: Optional parameter indicating whether to use the internal output buffer or not. See chapter 3.8.2 about polling for more information.

0 – Use the buffer. This way all data can be retrieved from the object with consecutive calls to MT_GetOrientationData.
1 – Always return the latest calculated orientation data.

The VARIANT always contains a SAFEARRAY with the orientation data calculated by the orientation estimation filter. The SAFEARRAY is constructed with VARIANT data types VT_ARRAY | VT_R4. Depending on chosen output mode (default is quaternion) these are 4 (quaternion), 3 (euler angles) or 9 (cosine matrix) floats.

See chapter 5 and the MT User Manual for more details on the different output options.

Functionality: Function used to query the MT Object for new orientation data. The nRetVal parameter contains the objects answer. If the answer is 1 (New Data) the second parameter contains a pointer to the new orientation data

```
XM_GetCalibratedData(short *nRetVal, VARIANT *pfOutputArray, short nLatest);
```

nRetVal :

0 – No new data calculated (No Error)
1 – New data available
2 – No data on COM port
3 – No device ID received from sensor
4 – Incomplete data received (Connection lost)
5 – Checksum error in data from sensor
6 – COM port could not be opened
7 – XMU file with calibration data could not be read (not applicable for MT9-B)

pfOutputArray contains calibrated sensor data
nLatest: Optional parameter indicating whether to use the internal output buffer or not. See chapter 3.8.2 about polling for more information.

0 – Use the buffer. This way all data can be retrieved from the object with consecutive calls to MT_GetOrientationData.
1 – Always return the latest calculated orientation data.

The VARIANT always contains a SAFEARRAY with the calibrated sensor data including the temperature from the internal sensor.
The SAFEARRAY is constructed with VARIANT data types VT_ARRAY | VT_R4 and contains 10 values, 9 for the calibrated data (3D acceleration, 3D rate of turn and 3D magnetic field) and 1 for the temperature.

See chapter 5 and the MT User Manual for more details on the different output options.

Functionality: This function queries the MT Object for new calibrated data. This function only returns 0 or 1 for this query.

```
XM_GetFilterSettings(BSTR bstrDeviceID, float *fGain, short *nCorInterval, float *fRho);
```

bstrDeviceID: wide, double-byte (Unicode) string containing the device ID of sensor for which to retrieve the filter settings.
Functionality: Retrieve the current filter settings (gain, correction interval and weighting factor).

```
MT_GetMotionTrackerBaudrate(int *nBaudrate);
```

nBaudrate: Requested baudrate. Can be one of the following:

460800 – 460k8
230400 – 230k4
115200 – 115k2
57600 – 57k6

Functionality: Get the baudrate at which the Xbus Master is sending its data.

```
MT_GetMotionTrackerSampleFrequency(int *nSampleFrequency);
```

nSampleFrequency: Requested Sample Frequency. Can be one of the following:

25, 50, 100, 200, 256, 320, 400, 512 Hz

Functionality: Get the frequency at which the Xbus Master is sampling its data.

```
XM_StartProcess();
```

Functionality: Start the processing of MT9 data by the MT Object.
Note: Returns when all initialization internally is done. If COM port could not be opened, returns E_ACCESSDENIED (General access denied error)

```
XM_StopProcess();
```

Functionality: Stop the processing of MT9 data by the MT Object.

```
XM_ResetOrientation(short nResetType, short bSaveAfterStop);
```

nResetType: Type of reset to be done

0 – Heading reset
1 – Global reset
2 – Object reset
3 – Align (Object followed by Heading reset)

bSaveAfterStop: If 1, the heading and object reset values are stored in the MotionTracker non-volatile memory. This way, next time you use the sensor with our software, it will always be in this coordinate system. When the XM_StopProcess function has been called, the reset values of the reset which was called with this set to 1 are stored in the MotionTracker.
Default: 0, Do no save to memory
Functionality: Reset the reference orientation in the MotionTracker algorithm.

Please, refer to the MT User Manual for more details on the different co-ordinate system resets available.

3.6.3 Output functions using MT9-B with Xbus System

```
XM_GetMotionTrackerHeading(BSTR bstrDeviceID, float *fHeading);
```

bstrDeviceID: wide, double-byte (Unicode) string containing the device ID of the sensor
fHeading: Requested heading. See MT9 Technical Documentation for more information on the heading.
Functionality: Get the heading in respect to which the orientation will be calculated.

```
XM_GetMotionTrackerLocation(BSTR bstrDeviceID, short *nLocation);
```

bstrDeviceID: wide, double-byte (Unicode) string containing the device ID of the sensor
nLocation: Indicator for the location of the sensor. Can be a number from 0 to 255.
Functionality: The user can get a location ID, so each ID can represent a certain location. This value is not used be the MT9 Software or SDK.

3.7 Events

The MT Object can also generate events. This is explained in section 3.8. Events are not yet functional when using the IDispatch interface.

A second interface, called IMotionTrackerEvents (_IMotionTrackerEvents in version 1.0) contains the function used to signal the sink that a new orientation has been calculated:

This function is called by the MT Object when it has calculated new orientation data and the user selected to use events. (i.e., advised the internal sink object to the MT Object)

```
MT_OrientationChanged();
```

This function must be implemented in the sink object by the user.

3.8 Polling and Events

You can decide whether to use the **polling mechanism** or the **events mechanism** to retrieve the data available in the MT Object.

3.8.1 Polling

When using the polling method, the user continuously (or at a certain interval) queries the MT Object if new orientation data (or calibrated sensor data) has been calculated. The query is carried out with the MT_GetOrientationData and/or MT_GetCalibratedData function. **When using the Xbus system the functions XM_GetOrientationData or XM_GetCalibratedData are used.**

When queried, the MT Object will immediately return the most recently calculated data. The polling method may be useful when the query function runs in a loop at a certain update rate (different from or out of synch with the sampling in the MT hardware) and each time orientation data is needed, the user just needs the latest data and not necessarily every single sample.

The sample program (source-code) includes an example implementation of the polling mechanism. The polling is implemented in a separate thread, the '*pollthread*', and in the case of the example implementation it is used to poll the MT Object at a certain frequency. This is implemented by waiting an interval period (using a timer) before the next poll is done.

Note that when using polling, the Disconnect-/ConnectMTSink functions are not required, as they are only needed when using the events method. Initialization and destruction of the local MotionTrackerSink object are also not needed.

3.8.2 Internal cyclical buffering

The COM object has an internal buffer which is used to give the user some breathing space with polling. If the user cannot poll for a short time (maximum buffer is 256 samples, 2.56 s @ 100 Hz sampling frequency) the number of samples in the buffer is increased.

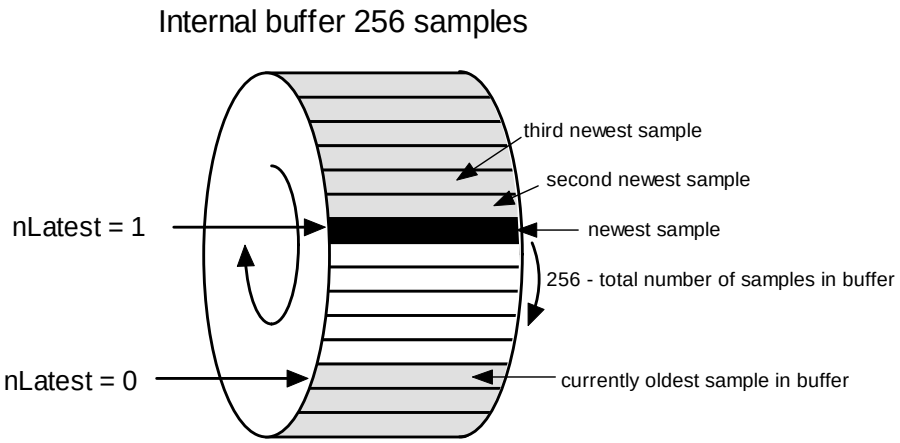


Figure 3 - Representation of COM object internal buffer

By default, the COM object returns the latest sample in the buffer (nLatest = 1). As can be seen above, by adjusting the optional parameter to the poll function (see section 2.6 and 2.7) one can decide to get the oldest sample (nLatest = 0), and by using consecutive calls with the poll function, catch up from the delay until the number of samples in the buffer is zero. If polled significantly faster than the sample frequency the buffer will soon be empty, the user then has retrieved all buffered data in the original order.

Due to the use of the internal buffer in the MT Object, it is now possible to retrieve **all data**, independent of the use of polling or events. When polling, by passing an extra optional parameter to the poll function (see section 2.6 and 2.7), the user can decide whether to use this internal buffer or not. By default the polling functions always returns the **latest** calculated sample. By setting the optional parameter to 0 the buffer is enabled and the user may, if polling was halted for a short time, by using consecutive calls to the COM object, retrieve all data which was calculated during the short halt.

3.8.3 Events

When using the events method, instead of continuously querying the MT Object, the object notifies the sink when new data has been calculated and is available for retrieval with the MT_GetOrientationData or MT_GetCalibratedData function. **When using the Xbus system the functions XM_GetOrientationData or XM_GetCalibratedData are used.**

When using events The Disconnect-/ConnectMTSink functions and the local MotionTrackerSink object are needed. How this should be done can be seen in chapter 3.

Polling vs. Events

Whether to use polling or events is mostly dependent on user programming environment. For instance, the usage of COM events is not yet supported in Matlab, so polling is necessary here. If you are using C++ with the Microsoft Foundation Classes (MFC) the use of events is the logical way to go.

Also some other considerations might be worth noting. For instance, if the update rate needed

in your software is lower than (or higher) the MT device sample frequency (usually 100 Hz) the use of polling is advised. The polling method ensures you always get the latest available orientation data when you ask for it. The polling method allows that the other process in your software (e.g. 3D rendering) to be asynchronous with the sampling rate of the MT itself, and you can synchronize the MT data (within about 10 ms precision) with your processes. Furthermore, polling is slightly more straightforward to implement.

When it is very important that every sample the MT sends is retrieved **at the same rate as the sample frequency of the** MT, the use of events is advised. When using the events method the MT software component will notify you when another sample is available and your software must react to this and retrieve the sample for further processing. So, in this method the MT is "**active**".

On a MS Windows OS, the use of events is sometimes more reliable when it comes to timing. This is because with the polling mechanism, the polling is dependent on whether it is scheduled continuously by the OS. Depending on the rest of your software code, and other activities of the PC, it could happen that the poll thread is not scheduled for 50-100 ms, which would result in data loss if the internal buffer of the COM-object is not used. With the events method, the retrieval of data after the event is always scheduled within 10 ms.

4 Implementation of the MT Object

4.1 Construction of MT Object

There are a number of ways to make the capabilities of the MT Object software component available in your software development environment:

1: IMTObj.h header file.

2: #import “IMTObj.dll” (already includes the IMTObj.tlb)

3: Generate c++ code using some tool or wizard. **Not explained further in this document refer to the documentation of your software development tool.**

In case 2, the compiler creates intermediate files (.tlh, .tli) that contain the expanded interface declarations. Further, the compiler also declares smart interface pointer classes built around the raw interfaces.

When using COM functionality in a program, the COM library must be initialized before use. This is done with CoInitialize(NULL). Un-initialization must be performed at cleanup (Section 4.5)

1) Header file method:

```
#include “IMTObj.h”          // Contains function declarations
#include “IMTObj_i.c”        // Contains GUIDs

//
// Initialize the COM library;
//
CoInitialize(NULL);

// Create instance of MTObj COM object
hRes = CoCreateInstance(CLSID_MotionTracker, NULL, CLSCTX_SERVER,
                        IID_IMotionTracker, (void**)&pMT);
if (FAILED(hRes))
{
        CString csError;
        csError.Format(_T("Error %x in CoCreateInstance for MT object!"),hRes);
        AfxMessageBox(csError);
        return FALSE;
}
```

2) Import DLL method:

```
#import "..\release\MTObj.dll"
CoInitialize(NULL);
MotionTrackerFilter::IMotionTrackerPtr pMT;
pMT.CreateInstance("MotionTracker.FilterComponent");
```

4.2 Initialize local CMotionTrackerSink object

When events are used, the sink object must be initialized. This class is explained in detail in Appendix B.

```
CMotionTrackerSink* pCob = NULL;
if (SUCCEEDED(hRes))       {
        // Create CMTObj Sink object to reveive CMTObj events
        pCob = new CMotionTrackerSink(NULL,this);
        if (NULL != pCob) {
          // Save a pointer to the CMotionTracker IUnknown interface. Addref
          // because of this saved copy.
          m_pCMotionTrackerSink = pCob;
          m_pCMotionTrackerSink->AddRef();
        }
        else
          hRes = E_OUTOFMEMORY;
}
```

4.3 Basic use

In the case of events you need to do this first (function implementation in Appendix C), other wise skip to next step:

```
ConnectMTSink ();
```

Set MT Object parameters, if default values not apply. Usually MT_SetCOMPort is needed to be set to the COM-port where you have connected your MT device:

```
pMT->MT_SetOutputMode(MT_LOGEULER);
pMT->MT_SetCOMPort(1);
pMT->MT_SetFilterSettings(1.0f,1,1.0f);     // Not necessary
pMT->MT  SetDoAMD(0);                       // Not necessary
```

Start processing data in the MT Object. This will let the MT Object receive the stream of data from the MTi, MTx or MT9 you have connected. In the case of the MTi / MTx the MT Object will only read data from the device and pass it on, the processing is done on the embedded DSP in the device itself. In the case of the MT9-B, the processing is done on the host PC:

```
pMT->MT_StartProcess();
```

Once the “process” is started, you can retrieve data using the MT_GetOrientationData and MT_GetCalibratedData.

When processing is no longer needed, call following code to stop the processing by the MT Object (this will free up some of your computing resources):

```
pMT->MT_StopProcess();
```

In case of events (function implementation in Appendix C):

```
DisconnectMTSink();
```

4.4 Basic use when using Xbus system

In the case of events you need to do this first (function implementation in Appendix C), other wise skip to next step:

```
ConnectMTSink ();
```

To set MT Object parameters:
It is necessary to call some functions in a certain order:

First the COM port needs to be set. After this the function *XM_QueryXbusMaster* will try to query the Xbus Master on the COM port just set. If the first parameter contains a value greater then zero after the function returns, the XbusMaster was found and the 2nd to 6th parameter will contain the device ID's of the attached sensors.

```
// Set COM port number where MT9 is attached
pMT->XM_SetCOMPort(nPortNumber);

short nNumSensors;
pMT->XM_QueryXbusMaster(&nNumSensors , &bstrDeviceIDs);

if (nNumSensors > 0){
        char chSensorID[MAXSENSORS][9];
        WCHAR szuText[MAXSENSORS * 8];
        wcscpy(szuText,bstrDeviceIDs);
        for (int i = 0; i < nNumSensors; i++) {
                // Use DeviceID to set filter settings
                // If DeviceID is needed copy to local char array
                WideCharToMultiByte(CP_ACP, 0, &szuText[i*8], 8,
                                chSensorID[i], _MAX_PATH, NULL, NULL);

                // Set Gain, Correction interval and Rho
                pMT->XM_SetFilterSettings(&bstrDeviceIDs[i*8],
                                        fGain,nCorInterval,fRho);
        }
        // Free allocated string, was allocated in the COM object
        ::SysFreeString(bstrDeviceIDs);

        // Optional settings
        pMT->XM_SetCalibratedOutput(m_bLogCalibratedData);

        // Required settings
        pMT->XM_SetOutputMode(m_nMode);

        // Set timestamp to be included in output data
        pMT->XM_SetTimeStampOutput(nTimeStampOutput);
}
```

Start processing data in the MT Object

```
pMT->XM_StartProcess();
```

Once the “process” is started, you can retrieve data using the XM_GetOrientationData and XM_GetCalibratedData.

When processing is no longer needed, call following code to stop the processing by the MT Object (this will free up some of your computing resources):

```
pMT->XM_StopProcess();
```

In case of events (function implementation in Appendix C):

```
DisconnectMTSink();
```

4.5 Post-Processing

The MT Object supports offline processing of data. This way, binary data logged with the MT9 Software can be used as input for the object.

NOTE: This is only supported for the MT9-B and MT9-A.

The data must be sent to the object per data packet (to simulate a 'connection' to a MT9). This also enables the user to process only a part of a file or a whole file depending on user demands.

For this purpose, the MT_SetInputPacketB function must be utilized. Data packets for this function must always be 27 bytes long, starting with the preamble (0xFA), BID and MID (0x32). This is the binary output format of the MT9 Software or data logged directly from the MT9-B.

When using the MT9-A, use the MT_SetInputPacket function. Data packets for this function must always be 22 bytes long, starting with the preamble 0xFAAF (binary output format of the MT9 Software).

To use this function, apply the following actions:

- Set all normal filter settings, except this time set the COM port to -1:
  MT_SetCOMPort(-1);
- MT9-B requires MTS data which must be set with MT_SetMTSData function.
- MT_StartProcess
- Loop with MT_SetInputPacketB (see sample below)
- MT_StopProcess.

Note: After the MT_StartProcess call the MT Object will time out after a few seconds. So MT_SetInputPacketB must be called within a few seconds of the call to MT_StartProcess.

Please also note that the MT9 sensor must not be moving (quasi-static) the few first seconds after MT_StartProcess. During this time the sensor seeks its orientation in space. Keep this in mind when analyzing portions of data only.

Remember: Do not send new packet before previous orientation has been calculated!
Following pseudo-code shows how to use the COM object to post-process data. Complete code can be found in the ClientDialog sample installed in a typical MT9 SDK install.

Post Processing is not yet possible with the Xbus System nor the MTx/MTi.

```
char* pfoo;              // Pointer to char to hold data to be sent to COM object.
pfoo = new char[33];  // Equivalent of BSTR:
                         // 0-3: Length (in bytes, not including \0)
                         // 4-26: binary data
                         // 27-28: \0

pfoo[0] = 27;
pfoo[1] = pfoo[2] = pfoo[3] = pfoo[31] = pfoo[32] = 0;

unsigned char dataBuf[27];    // Array to hold raw captured data

// Set MTS data using MT_SetMTSData function

// When using MT9-A set the location of the XMU file.

pMT->MT_SetCOMPort(-1);

// Start processing by MTObj COM object

pMT->MT_StartProcess();

while (!bDone) {

// Only send new data when new orientations have been read from the object

if (WaitForSingleObject(hDataReadEvent, 500) == WAIT_OBJECT_0){
        // Always check if data is 27 bytes long and starts with the complete preamble
          if (dwRead == 27 && dataBuf[0] == 0xFA && dataBuf[1] == 0xFF) {
                    // Copy data to char* (First 4 contain string length, see above)
                    memcpy((pfoo + 4),dataBuf,27);
                    // Feed packet to MotionTracker object
                    pMT->MT_SetInputPacketB(BSTR(pfoo + 4));
          }
}
}

// Stop processing by MTObj COM object

pMT->MT_StopProcess();

// Clean up
delete [] pfoo;
```

4.6 Destruction of COM object

When the MT Object is no longer needed (i.e. at program termination), the object must be informed it is no longer needed so it can unload itself from memory. This is done by calling the Release function of object.

In case of events:

```
// Release and clean up internal sink object
if (m_pCMotionTrackerSink != NULL){
        m_pCMotionTrackerSink->Release();
        m_pCMotionTrackerSink = NULL;
}
```

Release the MT Object and COM library:

```
// Release and clean up MotionTracker object
if (pMT != NULL) {
        pMT->Release();
        pMT = NULL;
}
CoUninitialize();
```

5 Retrieve data from COM object

The data available in the MT Object can be accessed through a *safearray*. The safearray, as its name says, is an array data type that is safe which means that it keeps track of its limitations and bounds and therefore limits access within these bounds. Safearrays are used when passing arrays encapsulated in Variants (their base type is VT_ARRAY), for example between COM objects.

To retrieve orientation data from the MT Object, create a local variable which can contain the required orientation data or calibrated sensor data.

***Remember*: When using Xbus, MT9-B and timestamp output is requested (with MT/XM_SetTimeStampOutput) each array below needs to be 1 float larger. The timestamp will be placed in from of the other data.**

```
float fQuat[4] = {0}; or
float fEuler[3] = {0}; or
float fRotationMatrix[9] = {0};
float fCalibratedData[10] = {0};
```

When using the Xbus system, these arrays need to be the number of sensors times larger:

```
float fQuat[4 * NUMSENSORS] = {0}; or
float fEuler[3 * NUMSENSORS] = {0}; or
float fRotationMatrix[9 * NUMSENSORS] = {0};
float fCalibratedData[10 * NUMSENSORS] = {0};
```

To retrieve the data:

```
// Variable of VARIANT type to retrieve data from output of COM object
VARIANT buffer;

// Pointer to array data
void* pDest;

m_pMT->MT_GetOrientationData(&nNew, &buffer); // Or MT_GetCalibratedData
if (nNew == MT_NEWDATA) {
        // Check if array is not empty
        if (buffer.vt != VT_EMPTY){
          // Retrieve pointer to array data
          SafeArrayAccessData(buffer.parray, &pDest);

          // Copy data from the VARIANT array to the local fData array
          memcpy(fData, pDest, buffer.parray->rgsabound->cElements);

          // Invalidate pointer
          SafeArrayUnaccessData(buffer.parray);

          Data is now available in the fData array.
        }
}
else if (nNew != 0)
          //Check error code (nNew) when no new data
```

5.1 Orientation Data

5.1.1 Quaternion

A quaternion consists of 4 numbers, q_0, q_1, q_2, and q_3. They are returned by the *MT_GetOrientationData* function as *floats,* e.g. in variable *fQuat* in exactly this order:

fQuat[0] = q_0
fQuat[1] = q_1
fQuat[2] = q_2
fQuat[3] = q_3

Please, refer to the **MT User Manual** for co-ordinate system definitions and other definitions concerning the output formats.

When using the Xbus system, data of all Xbus sensors is concatenated in the output array returned by the XM_GetOrientationData function. Data is sorted starting with the sensor with the highest device ID (See Xbus Documentation).

Assume sensor 00002073, 00002071 and 00002070, data will be received as follows:

fData[0]: $q_{0(00002073)}$	fData[4]: $q_{0(00002071)}$	fData[8]: $q_{0(00002070)}$
fData[1]: $q_{1(00002073)}$	fData[5]: $q_{1(00002071)}$	fData[9]: $q_{1(00002070)}$
fData[2]: $q_{2(00002073)}$	fData[6]: $q_{2(00002071)}$	fData[10]: $q_{2(00002070)}$
fData[3]: $q_{3(00002073)}$	fData[7]: $q_{3(00002071)}$	fData[11]: $q_{3(00002070)}$

5.1.2 Euler Angles

Euler angles returned are respectively the roll, pitch and yaw at position 1, 2 and 3. All 3 are in degrees and not radians. They are returned by the *MT_GetOrientationData* function as *floats*, e.g. in variable *fEuler* in exactly this order:

fEuler[0] = roll [deg]
fEuler[1] = pitch [deg]
fEuler[2] = yaw [deg]

Please, refer to the **MT User Manual** for co-ordinate system definitions and other definitions concerning the output formats.

When using the Xbus system, data from all attached sensors is concatenated in the output array. Data is sorted starting with the sensor with the highest device ID. (See Xbus Documentation). The method is the same as desribed in section 5.1.1.

5.1.3 Rotation Matrix

A rotation matrix consists of 9 numbers. They are returned by the *MT_GetOrientationData* function as *floats* in e.g. the variable *fRotationMatrix*, in exactly this order:

$$\text{Matrix: } \begin{bmatrix} fRotationMatrix[0] & fRotationMatrix[1] & fRotationMatrix[2] \\ fRotationMatrix[3] & fRotationMatrix[4] & fRotationMatrix[5] \\ fRotationMatrix[6] & fRotationMatrix[7] & fRotationMatrix[8] \end{bmatrix}$$

Please, refer to the **MT User Manual** for co-ordinate system definitions and other definitions concerning the output formats.

When using the Xbus system, data from all attached sensors is concatenated in the output array. Data is sorted starting with the sensor with the highest device ID. (See Xbus Documentation). The method is the same as desribed in section 5.1.1.

5.2 Calibrated Data

The calibrated MT data consists of 10 values. When using timestamp output, 11 values are returned. They are returned by the *MT_GetCalibratedData* function as *floats* in e.g. the variable *fCalibratedData*, in exactly this order:

fCalibratedData[0] = accX [m/s2]
fCalibratedData[1] = accY [m/s2]
fCalibratedData[2] = accZ [m/s2]
fCalibratedData[3] = gyrX [rad/s]
fCalibratedData[4] = gyrY [rad/s]
fCalibratedData[5] = gyrZ [rad/s]
fCalibratedData[6] = magX [a.u.]
fCalibratedData[7] = magY [a.u.]
fCalibratedData[8] = magZ [a.u.]
fCalibratedData[9] = temp [ºC]

Please, refer to the **MT User Manual** for co-ordinate system definitions and other definitions concerning the output formats.

When using the Xbus system, this array continues with calibrated data for all attached sensors. Data from all attached sensors is concatenated in the array. Data is sorted starting with the sensor with the highest device ID. (See Xbus Documentation). The method is the same as described in section 5.1.1.

6 MT shared object for Linux

The shared object is for use on the Linux operating system. It was compiled using gcc-2.95 and glibc version 5 for i386. A version using gcc-3.2 and glibc version 6 is also available in the gcc-3.0 directory.

The shared object has the same basic functions as the COM object. Events, however, are not present in the shared object. For further reference, the README, INSTALL and the sample code should be consulted.

The file MotionTracker.h contains all function definitions available in the shared object.

6.1 Input Functions – Set() functions

Most input functions have the same declaration as the functions in the COM object for Windows. **Only the functions with different declaration are listed here.**

```
MT_SetxmuLocation(char *pchName);
```

pchName: Pointer to array of characters containing the full path to the XMU file.
No Default
Functionality: Set the location of the .XMU file containing the calibration values for the used sensor
IMPORTANT: Only needed when using MT9-A. The electronic datasheet containing calibration values of the **MT9-B** (the MotionTracker Specification (MTS) data) are stored in non-volatile memory embedded in the MT9-B and are automatically fetched by the COM-object.

```
MT_SetInputPacket(unsigned char *puchInputPacket);
```

puchInputPacket: Array of characters containing complete input packet starting with preamble (0xFAAF).
Functionality: Post processing of data with to the MT Object.

Caution: When processing data, make sure not to call MT_SetInputPacket before a new orientation was retrieved with MT_GetOrientationData. Otherwise data will be lost and incorrect orientations will be calculated.

```
MT_SetCOMPort_DeviceName(char *pchDeviceName);
```

pchDeviceName: Pointer to array of characters containing the fully qualified device name to which the MT is attached. (Starting with /dev/tty)
No Default
Functionality: Set the port used when not connected to standard serial device /dev/ttyS*)

6.1.1 Input Functions for MT9-B only

```
MT_SaveToMTS(char chDeviceID[]);
```

chDeviceID: Array of characters containing the device ID of the sensor (Can be obtained with the MT_QueryMotionTrackerB function).
Functionality: Store new values, set with MT_SetMotionTrackerBaudrate, -Heading, -Location, -SampleFrequency, into MotionTracker non-volatile memory.

```
MT_SetInputPacketB(unsigned char *puchInputPacket);
```

puchInputPacket: Array of characters containing complete input packet starting with preamble (0xFA), BID and MID(0x32).
Functionality: Post processing of data with to the MT Object.

Caution: When post-processing data, make sure not to call MT_SetInputPacketB before a new orientation was retrieved with MT_GetOrientationData. Otherwise data will be lost and incorrect orientations will be calculated.

```
MT_SetMTSData(unsigned char *puchMTSData0, unsigned char *puchMTSData1,
unsigned char *puchMTSData2);
```

puchMTSData0: When puchInputPacket1 and bstrInputPacket2 are 0:
Pointer to array of characters containing complete MTSData packet as obtained from MT9-B Software log or from MT_GetMTSData function.
Otherwise:
puchMTSData0, puchMTSData1 and puchMTSData1 contain the MTS data in 3 subparts as obtained when requested from the MT9-B. (See MTx-B Technical Documentation or C++ Sample code)
Functionality: Pass MTS data to MT Object when using the post processing of data.

Caution: Call this function before MT_StartProcess is called.

6.1.2 Input Functions using Xbus system

```
XM_SetxmuLocation(unsigned char chDeviceID[], unsigned char *pchName);
```

chDeviceID[]: Array of characters containing the device ID for which a corresponding XMU file location is entered in the second parameter.
pchName: Array of characters Must contain the full path to the XMU file corresponding to the first parameter
No Default !
Functionality: Set the location of the .XMU file containing the calibration values for the used sensor
IMPORTANT: Only needed when using MT9-A.

```
XM_SetFilterSettings(unsigned char chDeviceID[], float fGain, short nCorInterval,
float fRho);
```

chDeviceID[]: Array of characters containing the device ID for which the parameters are meant.
Default: 1.0 (gain), 1 (correction interval), 1.0 (weighting factor)
Functionality: Set the filter's gain, correction interval and weighting factor.

```
XM_SetDoAMD(unsigned char chDeviceID[], short nDoAMD);
```

chDeviceID[]: Array of characters containing the device ID for which the short in the second applies.
nDoAMD:
0 – Disabled
1 – Enabled
Default: 0 (disabled).
Functionality: Set if this adaptive filter should be used.

```
XM_SetCOMPort_DeviceName(char *pchDeviceName);
```

pchDeviceName: Pointer to array of characters containing the fully qualified device name to which the Xbus Master is attached. (Starting with /dev/tty)
No Default
Functionality: Set the port used when not connected to standard serial device /dev/ttyS*)

6.1.3 Input Functions using MT9-B with Xbus system

```
XM_SetMotionTrackerHeading(char chDeviceID[], float fHeading);
```

chDeviceID: Array of characters containing the device ID of the sensor fHeading: Set the required heading. See MT9 Technical Documentation for more information on the heading.
Functionality: Set the heading in respect to which the orientation will be calculated.
NOTE: Call to MT_SaveToMTS function to save into MotionTracker non-volatile memory is needed to activate the new heading reference setting!

```
XM_SetMotionTrackerLocation(char chDeviceID[], short nLocation);
```

chDeviceID: Array of characters containing the device ID of the sensor nLocation: Indicator for the location of the sensor. Can be a number from 0 to 255.
Functionality: The user can set a location ID, so each ID can represent a certain location. This value is not used be the MT9 Software or SDK. Call the MT_SaveToMTS function to save into MotionTracker non-volatile memory.

```
XM_SaveToMTS(char chDeviceID[], bstrDeviceID);
```

chDeviceID: Array of characters containing the device ID of the sensor (Can be obtained with the XM_QueryXbusMaster function).
Functionality: Store new values, set with the above 4 functions, into MotionTracker non-volatile memory.

6.2 Output Functions – Get() functions

```
MT_GetOrientationData(short *pnRetVal, float pfOutputArray[], short nLatest);
```

pnRetVal:

0 – No new data calculated (No Error)
1 – New data available
2 – No data on COM port
3 – No device ID received from sensor
4 – Incomplete data received (Connection lost)
5 – Checksum error in data from sensor
6 – COM port could not be opened

When using MT-B

7 – MTS data not available in file input (Not set with MT_SetMTSData)

When using MT9-A

7 – XMU file with calibration data could not be read

pfOutputArray contains orientation data (See chapter 6.4 below for retrieving the data)
nLatest: Optional parameter indicating whether to use the internal output buffer or not. See chapter 3.8.2 about polling for more information.

0 – Use the buffer. This way all data can be retrieved from the object with consecutive calls to MT_GetOrientationData.
1 – Always return the latest calculated orientation data.

See chapter 5 and the MT User Manual for more details on the different output options.

Functionality: Function used to query the MT Object for new orientation data. The pnRetVal parameter contains the objects answer. If the answer is 1 (New Data) the second parameter contains a pointer to the new orientation data.

```
MT_GetCalibratedData(short *pnRetVal, float pfOutputArray[], short nLatest);
```

pnRetVal:

0 – No new data calculated (No Error)
1 – New data available

pfOutputArray contains calibrated sensor data
nLatest: Optional parameter indicating whether to use the internal output buffer or not. See chapter 3.8.2 about polling for more information.

0 – Use the buffer. This way all data can be retrieved from the object with consecutive calls to MT_GetOrientationData.
1 – Always return the latest calculated orientation data.

See chapter 5 and the MT User Manual for more details on the different output options.

Functionality: This function queries the MT Object for new calibrated data. This function only returns 0 or 1 for this query.

6.2.1 Output functions for MT9-B only

```
MT_QueryMotionTrackerB(char *pchDeviceID);
```

pchDeviceID: Pointer to array of characters containing the device ID of attached sensor.
Functionality: Query the attached MT for its device ID.

```
MT_GetMTSData(unsigned char *puhMTSData);
```

pchMTSData: Pointer to array of characters containing 173 bytes of MTS data
Functionality: The user can get the MTS data for use with post-processing of data.

6.2.2 Output Functions using Xbus system

```
XM_QueryXbusMaster(short *nNumSensors, char * pchDeviceID0,
                   char * pchDeviceID1, char * pchDeviceID2,
                   char * pchDeviceID3, char * pchDeviceID4);
```

nNumSensors: Number of sensors found by the Xbus Master.
pchDeviceID0-5: Pointer to array of characters containing the device ID of attached sensors.
Functionality: Query the Xbus Master for attached sensors. The first return value contains the number of sensors attached. The 2nd till 6th parameter contains a device ID for each attached sensor.

To retrieve data from the MT Object, the following functions are available, which are defined in the IMotionTracker interface.

```
XM_GetOrientationData(short *nRetVal, float fOutputArray[], short nLatest);
```

nRetVal :

- 0 – No new data calculated (No Error)
- 1 – New data available
- 2 – No data on COM port
- 3 – No device ID received from sensor
- 4 – Incomplete data received (Connection lost)
- 5 – Checksum error in data from sensor
- 6 – COM port could not be opened
- 7 – XMU file with calibration data could not be read

fOutputArray: contains orientation data (See chapter 5 below for retrieving the data)
nLatest: Optional parameter indicating whether to use the internal output buffer or not. See chapter 3.8.2 about polling for more information.

- 0 – Use the buffer. This way all data can be retrieved from the object with consecutive calls to XM_GetOrientationData.
- 1 – Always return the latest calculated orientation data.

See chapter 5 and the MT User Manual for more details on the different output options.

Functionality: Function used to query the MT Object for new orientation data. The nRetVal parameter contains the objects answer. If the answer is 1 (New Data) the second parameter contains a pointer to the new orientation data

```
XM_GetCalibratedData(short *nRetVal, float fOutputArray[], short nLatest);
```

nRetVal :

0 – No new data calculated (No Error)
1 – New data available
2 – No data on COM port
3 – No device ID received from sensor
4 – Incomplete data received (Connection lost)
5 – Checksum error in data from sensor
6 – COM port could not be opened
7 – XMU file with calibration data could not be read

fOutputArray contains calibrated sensor data
nLatest: Optional parameter indicating whether to use the internal output buffer or not. See chapter 3.8.2 about polling for more information.

0 – Use the buffer. This way all data can be retrieved from the object with consecutive calls to XM_GetOrientationData.
1 – Always return the latest calculated orientation data.

See chapter 5 and the MT User Manual for more details on the different output options.

Functionality: This function queries the MT Object for new calibrated data. This function only returns 0 or 1 for this query.

```
XM_GetFilterSettings(char  pchDeviceID[], float *fGain, short *nCorInterval, float *fRho);
```

pchDeviceID[]:Array of characters containing the device ID of sensor for which to retrieve the filter settings.
Functionality: Retrieve the current filter settings (gain, correction interval and weighting factor).

6.2.3 Output functions using MT9-B with Xbus System

```
XM_GetMotionTrackerHeading(char  pchDeviceID[], float *fHeading);
```

pchDeviceID[]: Array of characters containing the device ID of the sensor
fHeading: Requested heading. See MT9 Technical Documentation for more information on the heading.
Functionality: Get the heading in respect to which the orientation will be calculated.

```
XM_GetMotionTrackerLocation(char  pchDeviceID[], short *nLocation);
```

pchDeviceID[]: Array of characters containing the device ID of the sensor
nLocation: Indicator for the location of the sensor. Can be a number from 0 to 255.
Functionality: The user can get a location ID, so each ID can represent a certain location. This value is not used be the MT Software or SDK.

6.3 Implementation of the MT shared object for Linux

6.3.1 Construction of the MT shared object

```
// Needed for MotionTracker functionality
#include "MotionTracker.h"

// Global pointers used for the MotionTracker library
void *module;
MotionTracker* pMT;
destroy_t* destroy_mtobject;

// Dynamically load MotionTracker library
printf("Dynamic load of the MotionTracker library...");
module = dlopen("libmtobject.so",RTLD_NOW);
if (!module) {
        printf("Couldn't open MotionTracker library: %s\n", dlerror());
        return 0;
}
printf("done\n\n");

// Load the symbols for creation and destruction
create_t* create_mtobject = (create_t*) dlsym(module,"create");
destroy_mtobject = (destroy_t*) dlsym(module,"destroy");

if (!create_mtobject || !destroy_mtobject) {
        printf("cannot load symbols\n");
        return 0;
}
// create an instance of the class
pMT = create_mtobject();
```

6.3.2 Filter operation

To use the MT shared object, the same functions are required as with the MT Object.
To set the XMU location, a different function must be used:

```
char cXmuLocation[22] = "2026_24092002_000.xmu";
pMT->MT_SetxmuLocation(cXmuLocation);
```

IMPORTANT: Only needed when using MT9-A. The electronic datasheet containing calibration values of the **MT9-B** (the MotionTracker Specification (MTS) data) are stored in non-volatile memory embedded in the MT9-B and are automatically fetched by the COM-object.

6.3.3 Post-Processing

Post-Processing is essentially the same as with the COM object, except for the input variable, which is not a BSTR, but a pointer to an unsigned char array. The same technique as described in 4.5 can be used.

6.3.4 Destruction of the shared object

```
// Wait for a moment to give MotionTracker object time to close internal thread
// Not essential, but segmentation fault may occur if program ends before object is //
// done
usleep(1000);  // Sleep 1 msec

// Destroy the class
destroy_mtobject(pMT);

// Close the MotionTracker shared object
dlclose(module);
```

6.4 Retrieve data from the shared object

To retrieve the data from the shared object, no special actions are required. Just create a local variable which will contain the data and call the MT_GetOrientationData function with this variable:

```
// Variable of float type to retrieve data from output of shared object
float fOrientationData[9] = {0};
short nNew = 0;

// Ask the object if new data has been calculated. nNew will contain the answer.
pMT->MT_GetOrientationData(&nNew,fOrientationData);
```

The data is, depending on chosen output format 4 (quaternions), 3 (Euler-angles) or 9 (rotation matrix) floats. See the **MT User Manual** for more details.

The function MT_GetCalibratedData outputs 10 floats. See the **MT User Manual** for more details.

6.5 Example command line program

The example C++ command line program is basically the same as the Windows version. Only the included header files and the items discussed above are different. Please refer to chapter 8 and the source code for further detail.

7 Example MFC Program

Included with the SDK is an example program with source code which uses the MT Object. The source code is fully commented and will be useful to get you started quickly in coding your own application using the MT Object.

The program was written using the Microsoft MFC libraries, but they are not necessary for the actual implementation of the COM object. They are merely used to present the user with a 'nice' graphical user interface (GUI), see Chapter 8 for a simple command line example.

To use the program, open the project file (ClientDialog.dsp).

- The project contains 5 classes and 2 global pointers. The first 3 (CAboutDlg, CClientDialogApp and CClientDialogDlg are default MFC.
- The CMotionTrackerSink class implements the sink object.
- The CPollThread class is run as a separate thread to poll the MT Object for data at certain intervals.

When actually implementing the use of the MT Object, only one of the latter two classes is needed, depending on the use of events or polling.

The MT Object is loaded in the CClientDialogDlg::OnInitDialog function. (See chapter 4.1).

Destruction of the object is handled in the CClientDialogDlg::OnDestroy function.

The MT device settings are set just before the object starts processing. This is done when the user presses the Process button on the main window which calls the OnButtonProcess function. Various other variables are also set at the beginning of this function. Except the m_bDoEvents boolean variable, this sets program to use the events method when set true, and polling when set false, all variables set here are settings for the MT Object.

Before first use, all settings of the MT should be checked (COM port number etc). They are all located at the top of the OnButtonProcess function.
IMPORTANT: Only needed when using MT9-A: Set the location of the .XMU file.

Setting the nPortNumber to -1 enables file input. Use a binary log file of the MT9-Software as input file (INPUTFILE define). See sample code for more details.

When using events, data from the MT Object is received in the CClientDialogDlg ::OnNewData function. The data is then put on the screen if Euler output is selected, otherwise, data is ignored.

With polling, data is received in the CPollThread::Run function. The data is then put on the screen if Euler output is selected, otherwise, data is ignored.

Compiling the sample code in Microsoft Visual Studio 6.0 produces no warnings or errors.

7.1 Example MFC program using MT9-B, MTi / MTx functions

This sample program is called ClientDialog_MTB. The code is fully documented. It demonstrates the possibility to store settings into MT and Xbus Master non-volatile memory. The following settings are supported by the MT Object:

Heading (-2 PI – 2 PI)
Location (0 - 255)
Sample Frequency (25, 50, 100, 200, 256, 320, 400, 512)
MT Baudrate (57600, 115200, 230400, 460800)

When using the Xbus Master, the last two are not stored in MT non-volatile memory, but are stored in the Xbus Master. See the example code on how to use these functions.

The program is suitable for both standalone MT9-B, MTi and MTx as well as MT9-A/B connected to the Xbus Master. Select the appropriate one with the radio button on the dialog.

7.2 Example MFC program using Xbus system

The sample program for use with the Xbus is called CClientDialog_XM.
The program setup is the same as the MFC sample program for 1 sensor as described in the previous chapter. The same functions are provided for use with the Xbus system (*XM_** functions). The methodology is identical but some additional function calls are needed (e.g. *XM_QueryXbusMaster(h)*). Please, refer to the example source code for details.

8 Example C++ command line program

Included with the SDK is a sample program with source code which uses the MT Object. The source code is fully commented and will be useful to get you started quickly in coding your own application using the MT Object.

To use the program, open the project file (ClientCmdLine.dsp).

- The project contains no classes, 3 global functions and 2 global variables.
- GetData() queries the MT Object for new data and return an error code on failure. If new data found, this is printed to the screen
- SetupFilter() creates an instance of the MT Object and sets all necessary object parameters
- main(int argc, char *argv[]) main function, handles the COM library, processing and the destruction of the MT Object.

Before first use, all settings of the MT device should be checked (COM port numbert, baudrate, etc). They are all located at the top of the OnButtonProcess function. IMPORTANT: Only needed when using MT9-A: Set the location of the .XMU file.

Compiling the sample code in Microsoft Visual Studio 6.0 produces no warnings or errors.

9 Example application code using the IDispatch interface

9.1 Matlab

MATLAB 6.5 (release 13 nad higer) is compatible with the MT-Object, older versions have not been tested. MATLAB supports events from controls only, not from servers, so with release 13 version of MATLAB a polling sampling method must be used.

Use the *actxserver* function to create and return a handle to the MotionTracker.FilterComponent object.

h=actxserver('MotionTracker.FilterComponent');

Once the MotionTracker.FilterComponent object is created, its functions can be called easily, for example;

MT_SetCOMPort(h,1);

Here, ***h*** is the handle to the MT-Object.

To display the names of the methods for the class of the MotionTracker.FilterComponent object use;

methods(h) or,
methodsview(h)

The *MT_SetxmuLocation(h,XMUfile)* **only needs toe be called if you are using an MT9-A**. If you are using an MTi / MTx or MT9-B, the correct calibration data is embedded in the MT9 itself and retrieved and used by the MT-Object, if needed. The *MT_SetxmuLocation(h,XMUfile)*must always be called with a string (*XMUfile*) pointing to the location of the MT9 unique calibration file (.XMU file that came with your MT9), prior to use.

To start the orientation tracking using the MT simply give;

MT_StartProcess(h)

And then use,

[arg1,arg2] = MT_GetOrientationData(h,1);
[arg1,arg2] = MT_GetCalibratedData(h,1);

or use one of the Matlab functions described below to get a number of consecutive samples from the MT Object.

To release the MT Object from the MATLAB environment the correct sequence of MATLAB function calls is;

MT_StopProcess(h)
delete(h);
clear h;

The included MATLAB -script, "*MT_easy_script.m*", shows all the necessary steps to setup the MT-object and retrieve data. It also demonstrates the use of the MTi, MTx and MT9-B only functions

Improper use, or function sequence calls may cause MATLAB not to be able to release the object[2], resulting in error messages from the MT-object (e.g. unexpected 'could not open COM-port'). Shutting down MATLAB will always release the MT-object.

Please note that the MT-object functions does not return standard MATLAB doubles. For some operations or loops (*for*, *while*) MATLAB is faster if the variable is converted to double (using *double(var)*)

A basic function that can be used to sample data from the MT-object is made available[3], *MT_get_data*. A polling method has been used to grab all available 3D orientation data or calibrated MT data buffered in the object. By using the local buffer in the MT-object it is possible to get ***all*** data transmitted by the MT, even when using polling in MATLAB.

```
function [d, status, N] = MT_get_data(h, Channels);

% [d,status]=MT_get_data(h, Channels);
%
% MTObj needs to be fully configured and started to call this function.
%
% Input variables:
%  h = handle to MTObj
%  Channels = number of Channels of data to retrieve
%                  defined by operation mode, Calibrated data = 10, Euler angles = 3,
Matrix = 9, Quaternion = 4 channels
%
% Output variables:
% d = output data vector
% status = status returned by MTObj, use MT_return_error to display
```

[2] This is caused by the way Matlab 6.5 handles the COM object. This can be handled for easier debugging with a try-catch statement, see example code for details.
[3] Please, note this function is completely re-written from earlier releases (earlier than SDK v.2.3)

```
[str_out]=MT_return_error(arg1)

% this function displays the error message returned from the MTObj in human
% readable format
```

The real-time plot function (calibrated inertial data or 3D orientations) are used to demonstrate the use of the MT-object in MATLAB. In the code you will find how easy it is to create the object in the MATLAB environment and then to use its functionality and configure custom settings through the *MT_Set…* functions. Then using the power of MATLAB to display all the data in real-time, zoom in on the data, or to have a little fun displaying the rotation matrix output in a colourful way. Please, refer to the code itself for details.

```
function MT_DisplayRealtimeData(varargin);

% MT_DisplayRealtimeData(varargin);
%
% Displays MT9 calibrated data or 3D orientation data in real time using Matlab:
%
% varargin: Input arguments
% 1. XMU-file location, [string] with complete path and filename (required)
% 2. COM-port [integer] to which MT9 is connected (default = 1 , i.e. COM1)
% 3. Display mode [string], choose to view:
%                 'caldata'       Calibrated inertial and magnetic data (default)
%                 'euler'         Orientation output in Euler-angles (see Tech Doc for
definition)
%                 'quat'          Orientation output in Quaternions
%                 'matrix'        Orientation output in rotation Matrix format
% 4. zoom-level, [1=12 s chart, 2=8 s chart (default), 3=4 s chart]
% 5. update rate, [1=slow, 2=medium (default), 3=fast]
```

9.1.1 MATLAB using Xbus system

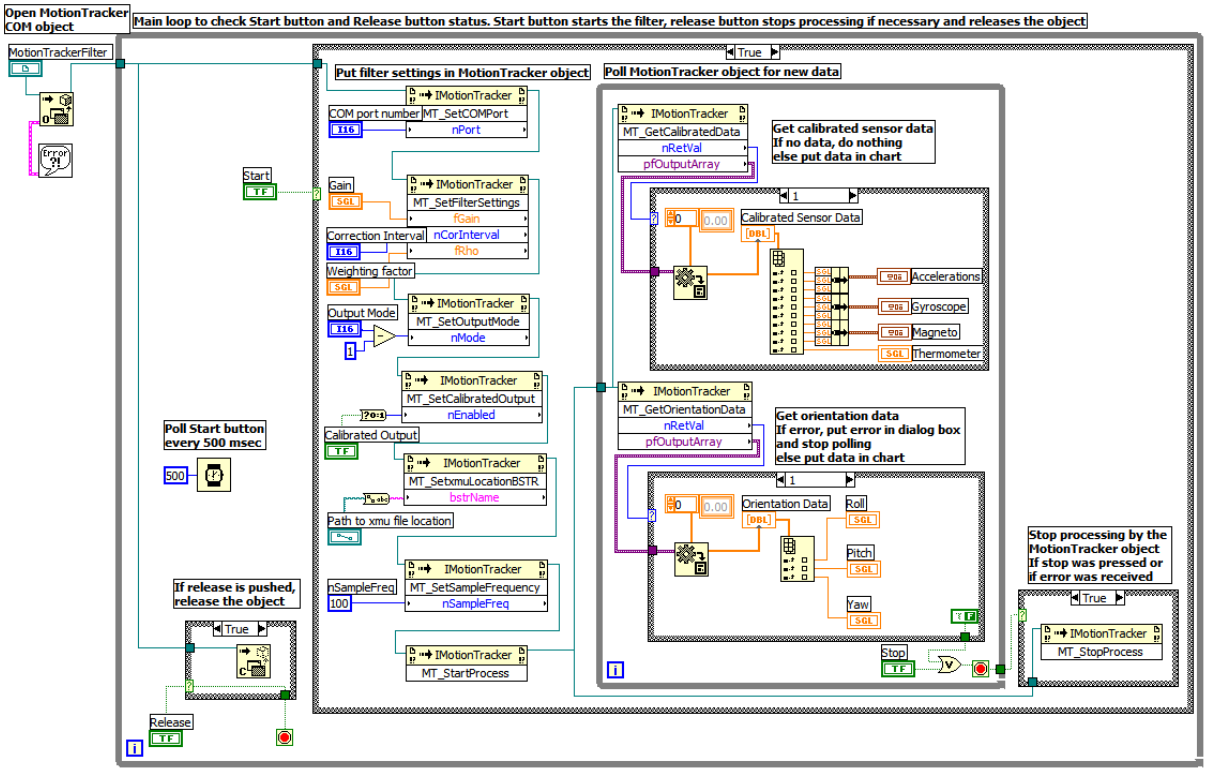
The same functions are provided for use with the Xbus system (*XM_** functions). The methodology is identical but some additional function calls are needed (e.g. *XM_QueryXbusMaster(h)*). Please, refer to the example source code for details.

9.2 LabVIEW

To use the MT Object complete the following steps:

- Create an “Automation Refnum” located in Controls→ActiveX
- Right click this to select the ActiveX Class. Browse to the MotionTracker 2.0 Type Library version 2.0. The Automation Refnum now is labelled “MotionTrackerFilter.IMotionTracker”
- Functions→Communication→ActiveX contains functions for opening and closing ActiveX(COM) objects. The Refnum needs to be opened before use (creates an instance of the MT Object)

At the start of the program, the MotionTrackerFilter is opened. When start is pressed, all object settings functions are called. After MT_StartProcess a loop queries the object for new data and plots new data in a chart. When done processing is stopped. If user wants to quit, the object must be closed with the Automation Close function.



9.2.1 LabVIEW using Xbus system

The same functions are provided for use with the Xbus system (*XM_** functions). The methodology is identical but some additional function calls are needed (e.g. *XM_QueryXbusMaster(h)*). Please, refer to the example source code for details.

9.3 Visual Basic script in MS Excel

To use the MT Object in MS Excel, open the Visual Basic Editor from the Tools→Macro menu. From the Visual Basic Editor, use Tools→References and select the MotionTracker 2.0 Type Library.

The VBA code is divided into 3 files:
Modules→Module1
Microsoft Excel Objects→ThisWorkbook
Microsoft Excel Objects→Sheet1

- Module1 contains the declaration of the public variables objMT9 and DoDemo.
- ThisWorkbook contains code that initializes variables and buttons when the workbook is opened, and checks if the objMT9 was properly closed on exit.
- Sheet1 contains the code that is called when one of the buttons are pressed.
    - CommandButton1_Click is called when the Start/Stop button is pressed. This creates an instance of the MT Object if is does not exist already and runs the DoDemo function. It also stops the processing by setting the DoDemo Boolean to false when called a second time.
    - The DoDemo Sub calls MT_GetOrientationData in a loop to check for new data and perform the correct action depending on the query. (Place data on the screen or report an error. The loop also contains a DoEvents call. This is to let the system also process events, so the system is not blocked while looping.
    - CommandButton2_Click is called when the Reset Orientation button is pressed. This calls the MT_ResetOrientation function of the object.

10 Appendix A - Introduction to COM

In this Appendix a short introduction to COM, a software component standard, will be given. It is by no means exhaustive and for more information please refer to other source of information, some suggestions are given below (Appendix E, reference 1 and 2).

10.1 General COM

COM (Component Object Model) refers to both a specification and implementation developed by Microsoft which provides a framework for integrating components. This framework supports interoperability and reusability of distributed software components by allowing developers to build systems by assembling reusable components from different vendors which communicate via COM.

COM defines an application programming interface (API) to allow for the creation of components for use in integrating custom applications or to allow diverse components to interact. However, in order to interact, components must adhere to a binary structure. As long as components adhere to this binary structure, components written in different programming languages can interoperate.

COM components consist of executable code distributed either as Win32 dynamic link libraries (DLLs) or as executables (EXEs). These are all registered in the Windows registry. The COM library uses this to get the location of a DLL or EXE.

Some of the advantages of the Component Object Model are:

- **Wire Level Standard**. The component users do not have to know anything about the underlying network mechanisms, TCP/IP or Serial Communications, to use the components.
- **Binary Standard**. The client and server components can be developed with different tools and/or different programming languages, and they will all interact properly as long as they adhere to the COM programming model and binary standard.
- **Runtime Polymorphism**. At runtime the client detects the right component it wants and uses its services. This means you do not have to recompile your client every time you make a change to your server. Once, you release a component, if you want to make a change then you release a new component. If the client wants the new services, only then you have to modify your client.

10.2 Objects and Interfaces

An object is an instantiation of some *class*. A "class" is the definition of a set of related variables and functions. The purpose is generally to provide some service to clients.

An object that conforms to COM is a special manifestation of this definition of object. A COM object appears in memory much like a C++ object. Unlike C++ objects, however, a client never has direct access to the COM object in its entirety. Instead, clients always access the object through clearly defined contracts: the **interfaces** that the object supports, *and only those interfaces*.

An **interface** is a strongly-typed group of semantically-related functions, also called "interface member functions." In COM, the name of an interface is always prefixed with an "I" by convention, as in IUnknown. (The real identity of an interface is given by its GUID; names are a programming convenience, and the COM system itself uses the GUIDs exclusively when operating on interfaces.) In addition, while the interface has a specific name (or type) and names of member functions, it defines only how one would use that interface and what behavior is expected from an object through that interface. Interfaces are known to the client by its pointer:

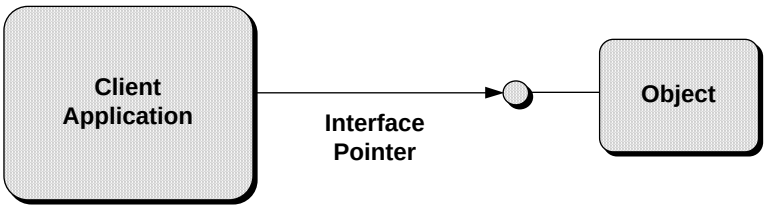


Figure 4 - Interfaces extend towards the clients connected to them.

10.3 Connectable Objects and Events

Interfaces in COM are normally "incoming" (from the perspective of the object). This means all communication is directed towards the COM object. (Settings for the object or retrieve data from the object). However, one would like to be able to receive a notification of the object when for example new data has been calculated. Instead of continuously requesting information it is sufficient to wait for a message from the object. This traffic is called "outgoing."

COM also defines mechanisms where objects can support "outgoing" interfaces. When an object supports one or more outgoing interfaces, it is said to be *connectable.* One of the most obvious uses for outgoing interfaces is for event notification.

A connectable object (also called a *source*) can have as many outgoing interfaces as it likes. Each interface is composed of distinct member functions, with each function representing a single *event, notification*, or *request.* Events and notifications are equivalent concepts (and interchangeable terms), as they are both used to tell the client that something interesting happened in the object.

In all of these cases, there must be some client that listens to what the object has to say and uses that information wisely. It is the client, therefore, that actually implements these interfaces on objects called *sinks*. From the sink's perspective, the interfaces are incoming, meaning that the sink listens through them. A connectable object plays the role of a client as far as the sink is concerned; thus, the sink is what the object's client uses to listen to that object.

An object does not necessarily have a one-to-one relationship with a sink. In fact, a single instance of an object usually supports any number of connections to sinks in any number of separate clients. This is called multicasting. In addition, any sink can be connected to any number of objects.

The COM technology known as Connectable Objects (also called "connection points") supports a generic ability for any object, called in this context a "connectable" object, to express these capabilities:

- The existence of "outgoing" interfaces, such as event sets
- The ability to enumerate the IIDs of the outgoing interfaces
- The ability to connect and disconnect "sinks" to the object for those outgoing IIDs
- The ability to enumerate the connections that exist to a particular outgoing interface.

Support for these capabilities involves four interfaces: IConnectionPointContainer, IEnumConnectionPoints, IConnectionPoint, and IEnumConnections. A "connectable object" implements IConnectionPointContainer to indicate existence of outgoing interfaces. Through this interface a client can enumerate connection points for each outgoing IID (via an enumerator with IEnumConnectionPoints) and can obtain an IConnectionPoint interface to a connection point for each IID. Through a connection point a client starts or terminates an advisory loop with the connectable object and the client's own sink. The connection point can also enumerate the connections it knows about through an enumerator with IEnumConnections.

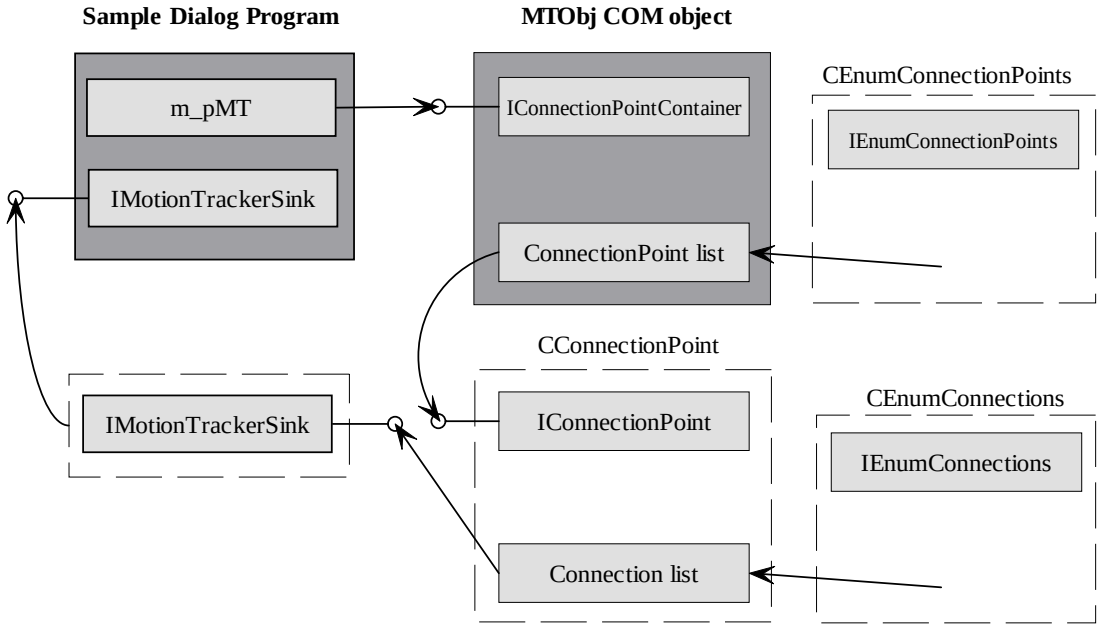


Figure 5 - ConnectionPoint Diagram

10.4 Default COM functions

Each COM object and its interfaces are unique and the interfaces always remain the same once released. When new functionality is added or function definitions change, this is done in a new interface, leaving the old one intact. *This ensures backwards compatibility in your own source code.*

To uniquely identify each object, it is assigned a GUID (Globally Unique Identifier). These numbers are unique in the entire world.

Each interface present in the component is identified with an IID, which is also a GUID. This Interface ID indicates what kind of pointer the caller expects to get returned. Through this pointer, functions in the interface can be accessed.

For COM compatibility, each COM object must implement some functions necessary for the basic functionality. These functions are defined in the IUnknown interface, a class from which every COM object is derived:

```
HRESULT QueryInterface(REFIID riid, void** ppv);
```

riid = GUID of the requested Interface

ppObj = Pointer to the requested interface, if present

Functionality: A function to query a COM object for a specific interface:

COM objects have an internal reference counter to keep track of the number of instances of the object. Following function increments the reference counter:

```
ULONG AddRef(void);
```

When no longer needed, the Release function of the object is called. This decrements the internal reference counter, and if it reaches 0 after this, the object unloads itself from memory.

```
ULONG Release(void);
```

Because the MT object is a DLL, the following 3 functions are also required:

The COM library periodically checks if a dll should remain in memory. It than calls the following function, which returns if it is ok to unload the dll by checking the reference counter

```
STDAPI DllCanUnloadNow();
```

Register the DLL to the COM library (Writes an entry in the windows register):

```
HRESULT __stdcall DllRegisterServer(void)
```

Unregister the DLL at the COM library:

```
HRESULT __stdcall DllUnregisterServer(void)
```

11 Appendix B – CMotionTrackerSink class

This appendix describes the implementation of the clients' internal Sink object. This consists of some standard COM functions, and the implementation of the MT_OrientationChanged function which is called by the MT Object.

11.1 Standard COM functions

To receive events from the MT Object, the client has to implement a small COM object. This object can handle basic COM functions, but is not registered and has no GUIDs assigned to it.

11.1.1 Definition

The definition of this object is as follows:

```
class CMotionTrackerSink : public IUnknown {
public:
        // Main Object Constructor & Destructor.
        CMotionTrackerSink(IUnknown* pUnkOuter,
                           CClientDialogDlg* pMainDlg);
        ~CMotionTrackerSink(void);

        // IUnknown methods. Main object, non-delegating.
        STDMETHODIMP        QueryInterface(REFIID, void**);
        STDMETHODIMP_(ULONG) AddRef(void);
        STDMETHODIMP_(ULONG) Release(void);
private:
        // IMotionTrackerSink methods.
        STDMETHODIMP      MT_OrientationChanged(void);

        // Main Object reference count.
        long            m_cRefs;

        // Outer unknown (aggregation delegation). Used when this COM object
        // is being aggregated.
        IUnknown*       m_pUnkOuter;

        // Pointer to the main object that can service the Sink events.
        CClientDialogDlg*       m_pMainDlg;
};
```

11.1.2 Implementation

The implementation of the default COM functions QueryInterface, AddRef and Release are according to standard COM guidelines:

```
STDMETHODIMP CMotionTrackerSink::QueryInterface(
                                                    REFIID riid,
                                                    void** ppv)
{
        HRESULT hr = E_NOINTERFACE;

        *ppv = NULL;

        if (IID_IUnknown == riid)
          *ppv = this;
        else if (IID__IMotionTrackerEvents == riid)
          *ppv = static_cast<_IMotionTrackerEvents*>(this);
        if (NULL != *ppv)
        {
          // We've handed out a pointer to the interface so obey the COM rules
          // and AddRef the reference count.
          ((LPUNKNOWN)*ppv)->AddRef();
          hr = NOERROR;
        }

        return (hr);
}
```

```
STDMETHODIMP_(ULONG) CMotionTrackerSink::AddRef(void)
{
	ULONG cRefs;

	cRefs = InterlockedIncrement(&m_cRefs);

	return cRefs;
}

STDMETHODIMP_(ULONG) CMotionTrackerSink::Release(void)
{
	ULONG cRefs;

	cRefs = InterlockedDecrement(&m_cRefs);

	if (0 == cRefs)
	{
	  // We artificially bump the main ref count to prevent reentrancy
	  // via the main object destructor.  Not really needed in this
	  // CMotionTrackerSink but a good practice because we are aggregatable
	  // and may at some point in the future add something entertaining like
	  // some Releases to the CMotionTrackerSink destructor.
	  m_cRefs = InterlockedIncrement(&m_cRefs);;
	  delete this;
	}

	return cRefs;
}
```

11.1.3 Event handling function

When the MT_OrientationChanged function is called by the event source (the MT Object) it posts a message to the main dialog indicating new data has been calculated.

The message handler for this message should than retrieve the data from the MT Object.

```
STDMETHODIMP CMotionTrackerSink:: MT_OrientationChanged(void)
{
	m_pMainDlg->PostMessage(WM_NEWDATA,0,0);
	return S_OK;
}
```

12 Appendix C – Connect-/DisConnectMTSink

These functions are used to connect/disconnect the internal sink object to the MotionTracker source object.

The ConnectMTSink function first gets the sink connection point from the MT Object. This is done in the function GetConnectionPoint which first queries the MT Object if it is connectable. If successful, it finds the IMotionTrackerEvents interface and retrieves a pointer to this interface.

On success, the ConnectMTSink function calls the standard COM function Advice, to connect the local sink to the object by passing this pointer to the object.

The DisconnectMTSink function tells the MT Object not to send any notifications to the sink and throw away the pointer to the sink object.

```
IConnectionPoint* CClientDialogDlg::GetConnectionPoint(REFIID riid)
{
	// Internal protected method to obtain a connection point interface.
	IConnectionPointContainer* pConnPointContainer = NULL;
	IConnectionPoint* pConnPoint = NULL;
	IConnectionPoint* pConnPt = NULL;
	HRESULT hr;

	// First query the object for its Connection Point Container. This
	// essentially asks the object in the server if it is connectable.
	hr = pMT->QueryInterface(
					IID_IConnectionPointContainer,
					(void**)&pConnPointContainer);

	if SUCCEEDED(hr) {
	    // Find the requested Connection Point. This AddRef's the returned pointer.
	    hr = pConnPointContainer->FindConnectionPoint(riid, &pConnPt);
	    if (SUCCEEDED(hr))
		pConnPoint = pConnPt;

	    // Release the connection point container. We're done with it.
	    pConnPointContainer->Release();
	    pConnPointContainer = NULL;
	}
	return pConnPoint;
}
```

```
HRESULT CClientDialogDlg::ConnectMTSink(void)
{
	HRESULT hr = E_FAIL;
	DWORD		dwKey;
	IConnectionPoint* pConnPoint;

	// Connect the MotionTrackerSink to the server MTObj COM source.
	if (!m_dwMotionTrackerSink)
	{
		// Get MotionTracker Sink connection point.
		pConnPoint = GetConnectionPoint(IID__IMotionTrackerEvents);
		if (NULL != pConnPoint)
		{
			// Connect the object in the server to the MotionTracker Sink
			hr = pConnPoint->Advise(m_pCMotionTrackerSink, &dwKey);
			if (SUCCEEDED(hr))
				m_dwMotionTrackerSink = dwKey;

			// Release the connection point. We're done with it
			pConnPoint->Release();
			pConnPoint = NULL;
		}
	}
	return hr;
}
HRESULT CClientDialogDlg::DisconnectMTSink(void)
{
	HRESULT hr = E_FAIL;
	IConnectionPoint* pConnPoint;

	// Disconnect the MotionTrackerSink from the server MTObj COM source.
	if (m_dwMotionTrackerSink)	{
		// Get the MotionTracker Sink connection point.
		pConnPoint = GetConnectionPoint(IID__IMotionTrackerEvents);
		if (NULL != pConnPoint)	{
			// Disconnect the object in the server from the MotionTracker Sink.
			hr = pConnPoint->Unadvise(m_dwMotionTrackerSink);
			if (SUCCEEDED(hr))
				m_dwMotionTrackerSink = 0;

			// Release the connection point. We're done with it.
			pConnPoint->Release();
			pConnPoint = NULL;
		}
	}
	return hr;
}
```

13 Appendix D – Function overview table

Function name	Argument(s)	Argument(s) values(s)	Purpose	Default
MT_SetOutputMode	short	0 – Quaternion 1 – Euler 2 – Rotation Matrix	Set type of orientation output	0 – Quaternion
MT_SetSampleFrequency	short	100 – 100 Hz 250 – 250 Hz	Set the sample frequency in the MT-object to match the MT sensor sample frequency.	100
MT_SetCalibratedOutput	short	0 – Disabled 1 – Enabled	Set if calibrated sensor data is required	0 – Disabled
MT_SetCOMport	short int	1 – 15 57600, 115200, 230400 or 460800	Set COM port number of attached MT	1
MT_SetTimeout	short	1 – 120	Set timeout period before an error is returned during processing.	3
MT_SetxmuLocation	BSTR	Pointer to UNICODE Basic STRing.	Set full path to XMU file containing calibration values	-
MT_SetFilterSettings	float short float	Gain: 0.01 – 50.0 Correction Interval: 1 – 100 Rho: 0.0 – 10.0	Set filter gain, correction interval and weighting factor	1.0 1 1.0
MT_SetDoAMD	short	0 – Disabled 1 – Enabled	Set use of adaptive filter	0 – Disabled
MT_SetInputPacket	BSTR	Pointer to Basic STRing.	Set input packet (data) for post-processing purposes.	-
MT_GetOrientationData	short*	Pointer to short containing return value 0 – No new data calculated (No Error) 1 – New data available 2 – No data on COM port 3 – No device ID received from sensor 4 – Incomplete data received (Connection lost) 5 – Checksum error in data from sensor 6 – COM port could not be opened 7 – XMU file with calibration data could not be read	Check for new orientation data and get pointer to this data	-
	VARIANT*	Pointer to VARIANT array containing orientation data		
MT_GetCalibratedData	short*	Pointer to short containing return value 0 – No new data calculated (No Error) 1 – New data available 2 – No data on COM port 3 – No device ID received from sensor 4 – Incomplete data received (Connection lost) 5 – Checksum error in data from sensor 6 – COM port could not be opened 7 – XMU file with calibration data could not be read	Check for new calibrated sensor data and get pointer to this data	-
	VARIANT*	Pointer to VARIANT array containing calibrated sensor data		
MT_GetFilterSettings	float short float	0.01 – 50.0 0 – 100 0.01 – 10.0	Get filter gain, correction interval and weighting factor	-
MT_StartProcess	-	-	Starts processing of MT data by the COM object	-
MT_StopProcess	-	-	Stops processing of MT data by the COM object	-
MT_ResetOrientation	short	0 – Heading 1 – Global 2 – Object 3 – Align	Reset object. See Technical Documentation. Save Reset values after MT_StopProcess	0 – No save
	short	0 – Do not save 1 – Save		
MT_OrientationChanged	-	-	Local Function to receive events from COM object	-

14 Appendix E – Function overview table Xbus

Function name	Argument(s)	Argument(s) values(s)	Purpose	Default
XM_SetOutputMode	short	0 – Quaternion 1 – Euler 2 – Rotation Matrix	Set type of orientation output	0 – Quaternion
XM_SetSampleFrequency	short	100 – 100 Hz 250 – 250 Hz	Set the sample frequency in the MT-object to match the MT9 sensor sample frequency.	100
XM_SetTimeStampOutput	short	0 – Disabled 1 – Enabled	Set if timestamp output is required	0 – Disabled
XM_SetTimeout	short	1 – 120	Set timeout period before an error is returned during processing.	3
XM_SetCalibratedOutput	short	0 – Disabled 1 – Enabled	Set if calibrated sensor data is required	0 – Disabled
XM_SetCOMport	short int	1 – 15 57600, 115200, 230400 or 460800	Set COM port number of attached MT9	1, 115200
XM_SetxmuLocation	BSTR BSTR	Basic STRing with device ID Basic STRing with full XMU path.	Set full path to XMU file containing calibration values	-
XM_SetFilterSettings	BSTR float short float	Basic STRing with device ID Gain: 0.01 – 50.0 Correction Interval: 1 – 100 Rho: 0.0 – 10.0	Set filter gain, correction interval and weighting factor	1.0 1 1.0
XM_SetDoAMD	short	0 – Disabled 1 – Enabled	Set use of adaptive filter	0 – Disabled
XM_QueryXbusMaster	Short* 5 * BSTR*	Number of sensors attached Pointer to Basic STRing.	Set input packet (data) for post-processing purposes.	-
XM_GetOrientationData	short* VARIANT*	Pointer to short containing return value 0 – No new data calculated (No Error) 1 – New data available 2 – No data on COM port 3 – No device ID received from sensor 4 – Incomplete data received (Connection lost) 5 – Checksum error in data from sensor 6 – COM port could not be opened 7 – XMU file with calibration data could not be read Pointer to VARIANT array containing orientation data	Check for new orientation data and get pointer to this data	-
XM_GetCalibratedData	short* VARIANT*	Pointer to short containing return value 0 – No new data calculated (No Error) 1 – New data available 2 – No data on COM port 3 – No device ID received from sensor 4 – Incomplete data received (Connection lost) 5 – Checksum error in data from sensor 6 – COM port could not be opened 7 – XMU file with calibration data could not be read Pointer to VARIANT array containing calibrated sensor data	Check for new calibrated sensor data and get pointer to this data	-
XM_GetFilterSettings	BSTR float short float	Basic STRing with device ID 0.01 – 50.0 0 – 100 0.01 – 10.0	Get filter gain, correction interval and weighting factor	-
XM_StartProcess	-	-	Starts processing of MT9 data by the COM object	-
XM_StopProcess	-	-	Stops processing of MT9 data by the COM object	-
XM_SetXbusMasterSampleFrequency	short	25, 50, 100, 200, 256, 320, 400, 512	Set the sample frequency in the Xbus Master.	-
XM_SetXbusMasterBaudrate	Int	57600, 115200, 230400 or 460800	Set Baudrate of the Xbus Master.	-
XM_GetXbusMasterSampleFrequency	Short*	25, 50, 100, 200, 256, 320, 400, 512	Get the sample frequency in the Xbus Master.	-
XM_GetXbusMasterBaudrate	Int*	57600, 115200, 230400 or 460800	Get Baudrate of the Xbus Master.	-
XM_ResetOrientation	short short	0 – Heading 1 – Global 2 – Object 3 – Align 0 – Do not save 1 – Save	Resetobject. See Technical Documentation. Save Reset values after Stopt	0 – No save
MT_OrientationChanged	-	-	Local Function to receive events from COM object	-

15 Appendix F – Function overview MTi / MTx and MT9-B specific

Function name	Argument(s)	Argument(s) values(s)	Purpose	Default
MT_QueryMotionTrackerBt	BSTR	Basic STRing with device ID	Query attached MT9-B for its DeviceID	1
MT_SetTimeStampOutput	short	0 – Disabled 1 – Enabled	Set if timestamp should be returnd with each data samplet	1
MT_SaveToMTS	BSTR	Basic STRing with device ID	Save MTS data to MotionTracker	-
MT_SetMotionTrackerSampleFrequency	short	25, 50, 100, 200, 256, 320, 400, 512	Set the sample frequency in the sensor. Save to memory wth SaveToMTS function	-
MT_SetMotionTrackerBaudrate	int	57600, 115200, 230400 or 460800	Set Baudrate of the MT9. Save to memory wth SaveToMTS function	-
MT_SetMotionTrackerLocation	short	0-255.	Set Location of the MT9. Save to memory wth SaveToMTS function	-
MT_SetMotionTrackerHeading	float	-2 PI – 2 PI	Set Heading of the MT9. Save to memory wth SaveToMTS function	-
MT_GetMotionTrackerSampleFrequency	short*	25, 50, 100, 200, 256, 320, 400, 512	Get the sample frequency in the sensor. Save to memory wth SaveToMTS function	-
MT_GetMotionTrackerBaudrate	int*	57600, 115200, 230400, 460800	Get Baudrate of the MT9. Save to memory wth SaveToMTS function	-
MT_GetMotionTrackerLocation	short*	0-255.	Get Location of the MT9. Save to memory wth SaveToMTS function	-
MT_GetMotionTrackerHeading	float*	-2 PI – 2 PI	Get Heading of the MT9. Save to memory wth SaveToMTS function	-
MT_SetInputPacketB	BSTR	Pointer to Basic String.	Set input packet (data) for post-processing purposes.	-
MT_GetMTSData	BSTR*	Pointer to Basic String	Get the MTS Data from attached MT9-B	
MT_SetMTSData	BSTR BSTR (Optional 2*)	Basic String with MTSData in a parts Basic String with MTSData in 3 subparts	When using post-processing: Set MTS Data in a part or 3 sub parts	

16 Appendix G – Function overview MT9-B with Xbus

Function name	Argument(s)	Argument(s) values(s)	Purpose	Default
XM_SaveToMTS	BSTR	Basic STRing with device ID	Save MTS data to MotionTracker	-
XM_SetMotionTrackerLocation	BSTR short	Basic STRing with device ID 0-255.	Set Location of the MT9. Save to memory wth SaveToMTS function	-
XM_SetMotionTrackerHeading	BSTR float	Basic STRing with device ID -2 PI – 2 PI	Set Heading of the MT9. Save to memory wth SaveToMTS function	-
XM_GetMotionTrackerLocation	BSTR short*	Basic STRing with device ID 0-255.	Get Location of the MT9.	-
XM_GetMotionTrackerHeading	BSTR float*	Basic STRing with device ID -2 PI – 2 PI	Get Heading of the MT9.	-

17 Appendix H – Literature

1. Inside COM, Microsoft's Component Object Model
   Dale Rogerson, Microsoft Press, 1997
   ISBN 1-57231-349-8

2. http://www.microsoft.com/com/

3. http://msdn.microsoft.com