ActiveX Drivers

About the Company

Melles Griot is an established global force in the design and manufacture of mechanical hardware, motion control systems, vibration isolation systems, machine vision products and multi-element optical systems for fiber-optic, semiconductor and reprographic applications.

We offer customers an in-depth understanding of fiber component manufacture, allowing us to quickly and confidently develop optimal positioning solutions.

As a part of BarloWorld, global providers of world-leading industrial brands, we are committed to providing the service, relationships and attention to detail that make businesses excel.

Revision History

Issue No.	Date	Summary
1	260401	Initial Issue
2	301001	CN2753
3	301101	CN2783
4	190202	CN2869
5	200502	CN2956

Contents

Chapter 1

Introduction

The Melles Griot Nanopositioning Modular System has been designed to allow custom nanopositioning applications to be developed under the Microsoft Windows operating system. This custom programmability is facilitated by control software in the form of a C++ server (MelHost) running on a PC connected, via a Controller Area Network (CAN) bus, to the electronic nanopositioning modules.

The software server exposes functionality for use by third party applications via a high level 'Driver' applications programming interface (API). These high level software functions allow individual modules (motor drives, piezo drives etc.) within the system to be accessed and for command sequences to be created in order to automate a particular positioning application.

Traditionally such programmable functionality was exposed in a Windows environment using dynamic link library (DLL) function calls, and indeed the MelHost software ships with an extensive library of DLL functions (see *Handbook HA0093 Dynamic Link Library Reference*). Increasingly however, the use of direct DLL calls, and the subsequent problems this causes with mixed language software development, has given way to the use of ActiveX Interfacing technology.

With the MelHost server system all key high level commands, settings and system parameters are exposed through a set of ActiveX automation objects (formally called OLE Automation), known collectively as the **ActiveX Drivers**. These exposed objects allow the modular electronics system to be 'driven' from applications written by the user without the need to understand or alter the core system software. In addition, ActiveX technology is language independent, allowing custom applications development to be undertaken using any language or development system that supports ActiveX. Current development systems that support ActiveX include Visual Basic, LabVIEW, Visual C++, Delphi, Borland C++ Builder and, via VBA, Microsoft Office applications such as Excel and Word.

Note. Visual Basic is a relatively easy development system to use and is recommended for those users who are new to Windows applications development. In this documentation, implementation specific explanations and code samples are written using Visual Basic syntax. Some familiarity of software development using Visual Basic is therefore assumed. Refer to the documentation supplied with other development environments for information on accessing and programming ActiveX objects.

The ActiveX Drivers (or simply Drivers) described in this handbook are used with the range of modular nanopositioning equipment supplied by Melles Griot. Before using the software, it is advisable to be familiar with the configuration program MG17_Config as described in the handbook *HA0088 – Main Rack and Controller*. Familiarization with the set up and basic operation of any relevant hardware is also advised – see the handbooks supplied with the individual modules for more information.

Note. Earlier versions of the modular electronics MelHost server software (17CDM001) relied on the use of LabVIEW (17LVD002) or Visual Basic (17VBD002) drivers to provide high level software support. These drivers were shipped as a set of virtual instruments or *.bas (source) files for inclusion in LabVIEW or Visual BASIC projects respectively. The ActiveX Drivers described in this document are designed to replace both the 17LVD002 and 17VBD002 software drivers in new applications development. Legacy applications that rely on these original source code drivers are still supported by new versions of the MelHost server (17CDM002) via the existing DLL function calls that remain.

1.1 Installation

Formally the LabVIEW (17LVD002) and Visual Basic (17VBD002) Drivers were installed separately to the main MelHost server (17CDM001) installation. The ActiveX Drivers described in this handbook are installed automatically with newer versions of the MelHost server software (17CDM002). To install the server and ActiveX Drivers run 'setup.exe' from the MelHost folder on the installation CD and follow the on screen instructions.

1.2 ActiveX Components, Servers, and Clients

Any collection of programmable objects implemented using ActiveX technology is referred to as an ActiveX component. The ActiveX Drivers that expose objects in this way are therefore an ActiveX component.

An application that manipulates the objects exposed by an ActiveX component is often referred to as the ActiveX 'client' or 'client application'.

An ActiveX component that exposes programmable objects to other applications is often referred to as an ActiveX 'server'. The ActiveX Drivers are therefore an ActiveX server.

1.3 Objects, Methods, and Properties

The ActiveX Drivers consist of several 'objects', one for each module type, e.g. **Config**, **NanoSteps**, **NanoTraks** and **Piezos**. These objects have methods and properties as their external interface.

Methods are typically functions that are called to carry out some action. For example, the **Config** object contains a method, **ClearCallLog**, used to clear the Call Log. The **NanoSteps** object contains methods to initiate motor moves (e.g. **MoveRelativeAndWait**) and to abort such moves (**Halt**). The **NanoTraks** object contains a method to read the power level from NanoTrak modules (**GetRelativePower**)

The following lists show some of the methods and properties exposed by the **Config** object (described in more detail in Chapter 3):-

Methods:

SetupAConfiguration

ShowCallLogDlg

ClearCallLog

ShowErrorLogDlg

GetLastErrorInfo

ClearErrorLog

ReleaseAConfiguration

GetNanoStepNamesEx

GetNanoTrakNamesEx

GetPiezoNamesEx

GetDigitalIONamesEx

GetConfigurationListEx

GetNumNanoSteps

GetNumNanoTraks

GetNumPiezos

GetNumDigitalIOs

GetNanoStepNames

GetNanoTrakNames

GetPiezoNames

GetDigitalIONames

GetNumConfigs

GetConfigurationList

Properties are named attributes (or parameters) of the object and are typically read/write attributes (although some are read only). For example, the **MaxCallLogItems** property of the **Config** object is a read/write property used to set or read the maximum number of Call Log entries, which are used to record information about calls made into the server (this is discussed further in Chapter 2 onwards).

Properties:

ShowErrorLogDlgOnError

MaxCallLogItems

MaxErrorLogItems

1.4 Type Libraries

In order to use an ActiveX component such as the ActiveX Drivers, a client application must have information about the properties, methods and constants exposed by the available objects. Properties have data types; methods often return values and accept parameters. The client application requires information about the data types of all of these in order to access them.

This type information is contained in a Type Library file (extension .tlb). The MelHost server installation includes a type library file called 'MG17_Drivers.tlb' which is installed in the Windows System folder. The information contained in a type library is accessed in different ways depending on the particular development environment being used to develop the client application.

Brief details on accessing the Drivers type library information using Visual Basic are given below. If you are using a different development system then refer to the documentation supplied.

1.4.1 Referencing the ActiveX Drivers

A reference to the ActiveX Drivers must first be added to the Visual Basic design environment before any client applications can be written. To check that a reference to the Drivers exists select the **References** menu item from the **Project** menu. In the list of available references check for the presence of **MG17_Drivers 1.0 Type Library**. Select the check box next to this entry to add the reference to Visual Basic. If this entry does not appear in the references' list, then use the **Browse** button to display a dialogue box, which can be used to find the MG17_Drivers.tlb type library file. Once located (in the Windows\System folder) click the **Open** button to add **MG17_Drivers 1.0 Type Library** to the References list.

After a reference has been added to the Drivers, the information about various objects and their methods and properties can be accessed by using the Object Browser dialogue box. The Object browser in Visual Basic is displayed by selecting the **Object Browser** menu item from the **View** menu. When this dialogue box is displayed, use the **Project/Library** Box to select **MG17_Drivers** from the list of currently referenced ActiveX components. The **Classes** list displays all of the available classes (objects) exposed by the ActiveX Drivers. The **Members of** list displays the methods and properties of the particular Driver object selected in the **Classes** list. The definition of a particular selected method or property with a brief description of its use is shown in Fig. 1.1.

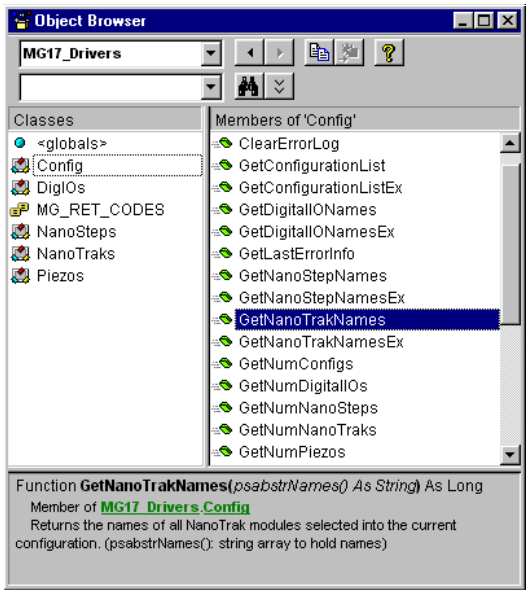


Fig. 1.1 Object browser

Note. The type library associated with the ActiveX Drivers is generated from an IDL (Interface Definition Language) script. With certain programming environments, such as LabVIEW and Visual Basic, it is not usually important to have knowledge of the IDL definitions for each method and property. However it may be helpful when developing client applications with some programming languages (such as C++) to have knowledge of the underlying IDL syntax for exposed methods and properties. In this regard Chapter 11 contains IDL definitions for the ActiveX Drivers.

1.5 ActiveX Driver Objects

Using the Object Browser of Visual Basic (or its equivalent in the development environment being used) the various programmable objects exposed by the ActiveX Drivers can be viewed. The objects currently implemented are listed below:-

Config Main configuration object required by every client application in order to initialize the ActiveX Drivers and obtain information on the number and type of electronics modules selected into the current configuration.
See Chapter 3 – Config Object, for further information.

NanoTraks Object giving access to the NanoTrak module specific commands and settings.
See Chapter 4 – NanoTraks Object, for further information.

Piezos Object giving access to Piezo module specific commands and settings
See Chapter 5 – Piezos Object, for further information.

NanoSteps Object giving access to the NanoStep module specific commands and settings.
See Chapter 6 – NanoSteps Object, for further information.

EncNanoSteps Object giving access to the Encoded NanoStep module specific commands and settings.
See Chapter 7 – Encoded NanoSteps Object, for further information.

DigIOs Object giving access to the Digital IO module specific commands and settings.
See Chapter 8 – DigIOs Object, for further information.

PowerMeter Object giving access to the Optical Power Meter module specific commands and settings.
See Chapter 9 – Power Meter Object, for further information.

As discussed earlier, an object comprises a collection of methods (functions) and properties (read/write parameters). A complete description of the ActiveX Driver objects and their methods/properties is given in the ActiveX Drivers Reference – Chapter 2 onwards.

Note. Melles Griot software is continually updated. To ensure that new additions/enhancements to the ActiveX Drivers are used, always access the latest Type Library file (MG17_Drivers.tlb) installed with upgrade versions

1.5.1 Referencing ActiveX Driver Objects

Before any client application can access the methods and properties exposed by the ActiveX Driver objects, it needs to declare and initialize variables that hold references to the individual objects. The following Visual Basic code example illustrates how to declare an object variable and obtain a reference to it:-

```
' Declare object variable to hold a reference to a Config object.
    Dim objConfig As MG17_Drivers.Config
' Obtain a reference to the ActiveX Drivers Config object.
    Set objConfig = new MG17_Drivers.Config
```

When an object is referenced (assigned to an object variable) a new object of this type is created by the ActiveX Drivers. Creating an object in this way is often referred to as ‘obtaining a new instance’.

When an object variable is no longer required it is good practice to disassociate (de-reference) the variable from the actual object. This releases memory (i.e. destroys the object in the Drivers) and other system resources. The following code sample illustrates how this is achieved using Visual Basic:-

```
' Release the reference to the Config object.
    Set objConfig = Nothing
```

Typically, object variables, and in particular a **Config** object variable, should be declared as global variables and referenced (object instances created) when the client application first loads. While the client application is running the referenced objects remain in existence and are used to access the various ActiveX Driver methods and properties. When the client application closes, the object variables are de-referenced and the associated objects destroyed. In Visual Basic, object variables are typically declared in the 'main' forms declarations section or in a module file, referenced in the 'main' form load event, and de-referenced in the main form unload event.

Note If an object variable is declared locally, it is automatically de-referenced when it goes out of scope, e.g., if an object variable is declared in a button‘s Click Event handler, and is re-referenced when the event handler returns. It is not good programming practice to let ActiveX Driver object variables fall out of scope during operation of the modular electronics, e.g., the Config object is used to set up and release configurations (described later) and should remain in existence throughout the life of a client application.

1.5.2 Single Instances of ActiveX Driver Objects

All objects exposed by the ActiveX Drivers are single instance in nature, i.e., only a single reference to a particular object should be created by a client application. If an application attempts to create more than one reference it will fail with a returned ActiveX error. In order to maintain reasonable compatibility with the legacy Visual Basic Drivers (17VBD002), each module type object is designed to support multiple electronics modules of the associated type by using a string naming convention in method calls. For example a single NanoSteps object can control any number of NanoStep modules (up to the hardware system limit). In this way it is only necessary to create one object of a particular type, even when there are more than one physical module of the associated type in a system configuration. The following code sample shows how a NanoTraks object is used to read power from two NanoTrak modules. The string names of the NanoTrak modules are named using the Melles Griot MG17_Config utility and are assumed to be 'NT 1' and 'NT 2' in this code sample. Refer to Handbook HA0088 – Main Rack and Controller for details on using the MG17_Config utility.

```
' Obtain reference (create Instance) of a NanoTraks object (variable
' Is declared elsewhere).
    Set objNanoTraks = New MG17_Drivers.NanoTraks
' Declare variable used to hold return codes.
    Dim lRetCode As Long
' Declare variables used to hold returned Power Levels.
    Dim dRelPower1 As Double, dRelPower2 As Double
    lRetCode = objNanoTraks.GetRelativePower("NT 1", dRelPower1)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
    lRetCode = objNanoTraks.GetRelativePower("NT 2", dRelPower2)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
```

1.6 ActiveX Driver Return Codes

Each ActiveX Driver method is designed to return a numeric value (return code) to indicate success or failure when called by a client application. A return code of zero indicates the method call was successful. If a method fails for any reason it returns immediately with an appropriate non-zero code to indicate which error was raised.

Note. Depending on the setting of the **ShowErrorLogDlgOnError** property of the **Config** object, an **Error Log** dialogue box is displayed by the ActiveX Drivers giving details of the error that was raised, – see Fig. 1.3. Further details of the error logging capabilities of the ActiveX Driver is given in Chapter 3.

A correctly written client application should trap all return codes and handle those situations when a method call has failed. For convenience, all possible return codes are contained within the type library of the ActiveX Drivers and therefore available within the development environment being used. These return codes are members of an enumeration called MG_RET_CODES. In Visual Basic the members of MG_RET_CODES can be viewed by using the Object Browser as described earlier. The source code sample in the previous section illustrates how these return codes are used. A zero return code
(i.e. MG_RET_CODES.MG17_OK) indicates that a method call has been successful
– see Chapter 10 for a description of each of the ActiveX Driver return codes.

1.7 ActiveX Drivers Call Logging

The ActiveX Drivers have been designed to incorporate full logging capabilities in order to assist software development and debugging. In this respect, all method calls into the Drivers (and the values of associated parameters passed) are logged in a **Call Log**. The maximum number of items that can be contained in this log is specified using the **MaxCallLogItems** property of the **Config** object (described further in Chapter 3 onwards).

Logged Information can be viewed by using the Call Log dialogue box. The Call Log dialogue box can be displayed at any time during execution of a client application by calling the **ShowCallLogDlg** method of the **Config** object. Fig. 1.2 shows typical Information displayed after a sequence of calls into the ActiveX server – see Chapter 3 for further details.

As a tool used during client application development the Call Log is useful for checking that the sequence of calls made into the server and the values of parameters passed are as intended,
e.g., if a motorized stage does not move over the expected distance the reason for the error can be traced using the call log to view the method parameters which initiated the move.

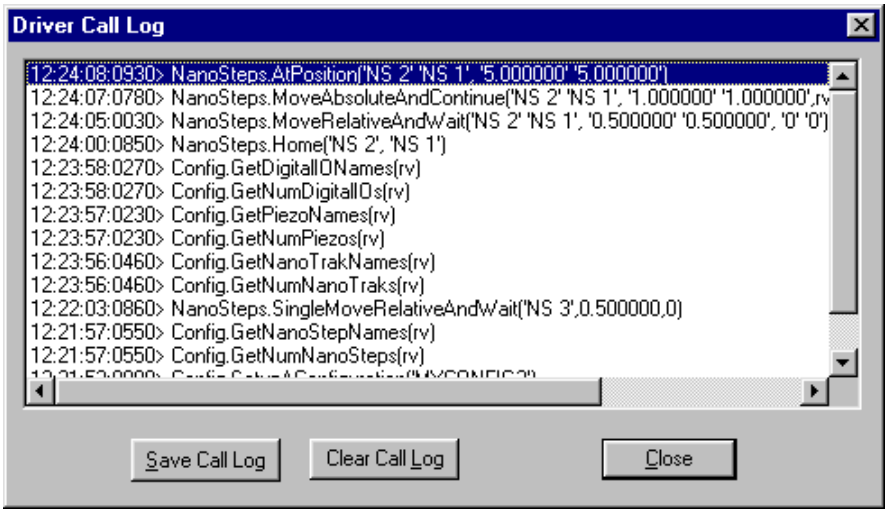


Fig. 1.2 Call log dialogue box

1.8 ActiveX Drivers Error Logging

In addition to the Call Log, the ActiveX Drivers also maintain an **Error Log**. Again the maximum number of error items logged can be specified using the **MaxErrorLogItems** property of the **Config** object. As with the Call Log, the information stored in the Error Log can be viewed by using the Error Log dialogue box. This can be displayed by a client application by calling the **ShowErrorLogDlg** method of the **Config** object.

Note. If an error occurs during execution of a client application, the Error Log dialogue box will be displayed automatically by the ActiveX Drivers if the **ShowErrorLogDlgOnError** property of the **Config** object is set to TRUE. After the Error Log dialogue has been closed by the user, the method that failed will then return with the appropriate return code as displayed in the Error Log dialogue box.

Typically to assist client application development, the **ShowErrorLogDlgOnError** property is set to TRUE, so that error information is displayed immediately a problem occurs. In a final released application, it is expected that a proper error handling scheme has been implemented to allow the client application to handle any error condition in the appropriate manner.

Fig. 1.3 shows the information displayed in the Error Log for a typical error condition. In this example a module name has been passed that is not a recognized part of the current configuration. Further details on configurations are found in section A – see Chapter 3 for further details on using the Error Log dialogue.

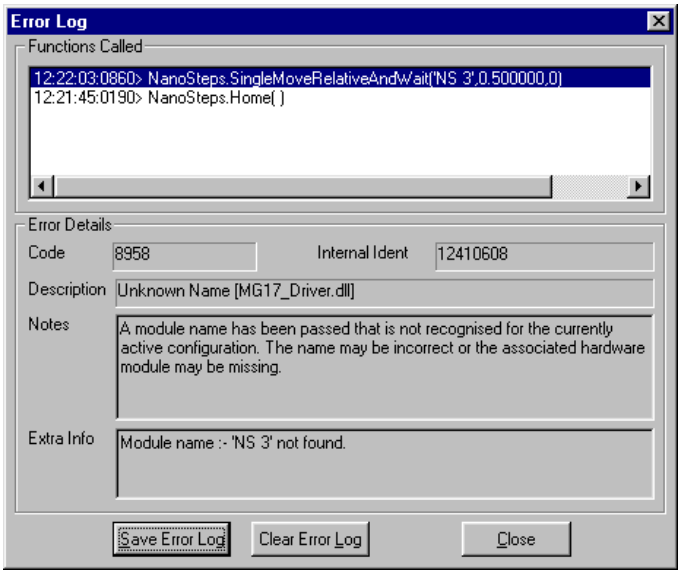


Fig. 1.3 Error log dialogue box

1.9 Configurations

The first task of most client applications is the setting up of the configuration, around which the whole concept of the modular system is based.

A configuration describes the elements of a nanopositioning system, the modules used and the equipment connected to them. A set of equipment may have several configurations, depending on the job it is performing. Each one allocates the type of actuators connected, their names, and the default properties for them. The utility application 'MG17_Config.exe' allows the user to do this interactively (see *Handbook HA0088 'Main Rack and Controller*').

Once a configuration is saved under a meaningful name, a client application can refer to it, and also to the control modules contained within it. The following code (implemented in Visual Basic for this example) must always be present within a client application in order to initialize the ActiveX Drivers with the named configuration and then make method calls to control the hardware modules:-

```
' Set up a named configuration created using the MG17_Config utility.
' This call initializes the ActiveX Drivers
    lRetCode = m_objConfig.SetupAConfiguration("CONFIG1")
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
```

The string parameter is the name of the desired configuration. For further information on making this call and the other coding requirements of a typical client application see Chapter 3.

When the ActiveX Drivers have been initialized with a particular named configuration, information about the numbers and types of hardware modules selected into the configuration can be obtained. For example, to obtain the names of all NanoSteps modules selected into the current configuration, a client application can use the following method calls:-

```
' Resizable string array used to hold names of modules in the
' configuration.
    Dim strNanoStepNames() As String
' Variable used to hold numbers of NanoSteps in the configuration.
    Dim lNumNanoSteps As Long
' Obtain the number of NanoStep modules.
    lRetCode = m_objConfig.GetNumNanoSteps(lNumNanoSteps)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
End If
```

```
' Resize string array to hold names of NanoStep modules.
    ReDim strNanoStepNames(1 To lNumNanoSteps)
' Fill resized string array with names of NanoStep modules.
    lRetCode = m_objConfig.GetNanoStepNames(strNanoStepNames)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
```

Once module names have been retrieved for the current configuration, they can be used in method and property calls to the ActiveX Drivers. Refer to Chapter 2 onwards.

Before an application terminates, you must release the resources allocated by the current configuration in a call to ReleaseAConfiguration. Therefore a client application should contain the following call as a part of it's 'clean up':-

```
' Release the current configuration to free up resources and clean up
' the ActiveX Drivers.
    lRetCode = m_objConfig.ReleaseAConfiguration()
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
```

For descriptions of the driver routines shown above, see Chapter 3.

1.10 Summary: A Typical ActiveX Drivers Client Application Framework

As a summary, in this section we put together many of the concepts discussed so far to describe the essential elements of a typical ActiveX Drivers client application.

In order to access and control the Melles Griot Modular Electronics, all client applications written to access the ActiveX Drivers generally share a common core framework. This usually comprises: creating the required ActiveX Driver objects (particularly the Config object), setting up a configuration, accessing the methods and properties to control the electronics modules as required, and finally releasing the current configuration when the application ends (or when software control of the electronic modules is no longer required). The Config object contains methods that are used by the client application to set up and release a configuration and find out the number, types and names of hardware modules included in the configuration.

In this regard, a Config object is always required by a client application. Further details on the methods and properties of the Config object are given in Chapter 3.

The following Visual Basic source code illustrates the typical coding framework required by an ActiveX Drivers client application in order to control the modular electronics system. In itself, this code sample performs no 'real world' nanopositioning function, but instead is intended to show the essential steps required by any client application.

In this example it is assumed a configuration called 'CONFIG1' has been created using the Melles Griot MG17_Config utility. This configuration contains three NanoStep modules (named NS 1, NS 2, NS 3), three Piezo modules (named P 1, P 2, P 3) and a single NanoTrak module (named NT 1). Refer to comments embedded in the source code for more details on the actions being taken. For brevity, return code handling is not included in this example code. Refer to the documentation supplied with the development environment being used for advice on implementing error handling schemes.

Declare object and other application variables required. This is typically done in a forms' declarations section or in a module (*.bas) file so that these variables have module level or global scope respectively.

```
' Module level declarations.
' Object variables.
    Dim m_objConfig As MG17_Drivers.Config
    Dim m_objNanoSteps As MG17_Drivers.NanoSteps
    Dim m_objNanoTraks As MG17_Drivers.NanoTraks
    Dim m_objPiezos As MG17_Drivers.Piezos
```

```
' Variables used to hold numbers of each type of module in the
' configuration.
    Dim m_lNumNanoSteps As Long
    Dim m_lNumNanoTraks As Long
    Dim m_lNumPiezos As Long

' Resizable string arrays used to hold names of modules in the
' configuration.
    Dim m_strNanoStepNames() As String
    Dim m_strNanoTrakNames() As String
    Dim m_strPiezoNames() As String
```

Next, obtain references to any ActiveX Driver objects required. This is typically carried out in the main forms' Load event:-

```
Private Sub Form_Load()
' Obtain references to (i.e. create Instances of) ActiveX Driver '
' objects required.
    Set m_objConfig = New MG17_Drivers.Config
    Set m_objNanoSteps = New MG17_Drivers.NanoSteps
    Set m_objNanoTraks = New MG17_Drivers.NanoTraks
    Set m_objPiezos = New MG17_Drivers.Piezos
```

After obtaining references use the Config object to set up the named configuration (created using the MG17_Config utility). Again this may be carried out in the main forms' Load event:-

```
' Declare variable used to hold return codes.
    Dim lRetCode As Long
' Set up a named configuration created using the MG17_Config utility.
' This call initializes the ActiveX Drivers
    lRetCode = m_objConfig.SetupAConfiguration("CONFIG1")
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
```

The Config object may then be used to obtain information about the number and type of modules selected into the current configuration and extract the associated module string names (allocated using the MG17_Config utility). As mentioned previously, this is typically carried out in the main forms' Load event:-

```
' Obtain the number of NanoStep modules.
    lRetCode = m_objConfig.GetNumNanoSteps(m_lNumNanoSteps)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure.
    End If
' Obtain the number of NanoTrak modules.
    lRetCode = m_objConfig.GetNumNanoTraks(m_lNumNanoTraks)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
' Obtain the number of Piezo modules.
    lRetCode = m_objConfig.GetNumPiezos(m_lNumPiezos)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
```

```
' Resize string array to hold names of NanoStep modules.
    ReDim m_strNanoStepNames(1 To m_lNumNanoSteps)
' Resize string array to hold names of NanoTrak modules.
    ReDim m_strNanoTrakNames(1 To m_lNumNanoTraks)
' Resize string array to hold names of Piezo modules.
    ReDim m_strPiezoNames(1 To m_lNumPiezos)

' Fill resized string array with names of NanoStep modules.
    lRetCode = m_objConfig.GetNanoStepNames(m_strNanoStepNames)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
' Fill resized string array with names of NanoTrak modules.
    lRetCode = m_objConfig.GetNanoTrakNames(m_strNanoTrakNames)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
' Fill resized string array with names of Piezo modules.
    lRetCode = m_objConfig.GetPiezoNames(m_strPiezoNames)
If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If

    End Sub
```

Note. Alternative methods exist for obtaining the names of modules selected into the current configuration. These methods (having names that end with 'Ex', e.g. GetPiezoNamesEx) are described in Chapter 3.

At this stage the main application administration is carried out. This allows the method and property calls to be made into the ActiveX Drivers so that control of the electronics modules can be achieved. The following code sample illustrates some calls:-

```
Private Sub cmdTESTCALLS_Click()

' Example calls into the ActiveX Drivers.

' Declare looping variable.
    Dim i As Integer
' Declare variable used to hold return codes.
    Dim lRetCode As Long
' Home all NanoSteps.
    lRetCode = m_objNanoSteps.Home(m_strNanoStepNames)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If

' Initiate a positive (away from limit switch) 1mm relative move
' of all NanoSteps'
' Declare and fill an array of the correct size to hold relative
' distances in mm.
```

```
    Dim dDistances(1 To 3) As Double
    dDistances(1) = 1#
    dDistances(2) = 1#
    dDistances(3) = 1#
' Declare and fill an array of the correct size to hold
' Backlash disabled flags.
    Dim bBackLashDisables(1 To 3) As Long
    bBackLashDisables(1) = 0    ' enable backlash correction
    bBackLashDisables(2) = 0    ' enable backlash correction
    bBackLashDisables(3) = 1    ' disable backlash correction
' Do the relative move.
    lRetCode = m_objNanoSteps.MoveRelativeAndWait
    (m_strNanoStepNames, _dDistances, _bBackLashDisables)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
' Home a single NanoStep.
' The configuration 'CONFIG1' contains a NanoStep named 'NS 2'.
    lRetCode = m_objNanoSteps.SingleHome("NS 2")
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If

' Set all Piezo modules to 'closed loop' mode.
For i = 1 To m_lNumPiezos
    lRetCode = m_objPiezos.SetCurrentMode(m_strPiezoNames(i), 1)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If

Next i
' Set NanoTrak mode to tracking in both axes.
' The configuration 'CONFIG1' contains a NanoTrak named 'NT 1'
    lRetCode = m_objNanoTraks.SetMode("NT 1", 1, 0)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If

    End Sub
```

Finally, at the end of any required software access of the ActiveX Drivers (e.g. application shut down), ensure that Driver resources are freed up by calling the ReleaseAConfiguration method and de-referencing all object variables. The latter usually happens automatically when an application ends and the object variables themselves fall out of scope. Typically this may be done in a form Unload event:-

```
Private Sub Form_Unload(Cancel As Integer)
' Release the current configuration to free up resources and clean up
' the ActiveX server.
    lRetCode = m_objConfig.ReleaseAConfiguration()
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
```

```
        End If

    ' De-reference ActiveX Driver objects to free up resources.
        Set m_objConfig = Nothing
        Set m_objNanoSteps = Nothing
        Set m_objNanoTraks = Nothing
        Set m_objPiezos = Nothing

        End Sub
```

Chapter 2

ActiveX Drivers Reference - Summary

This chapter contains a brief description of the methods contained in each object. A detailed description of the various ActiveX Driver objects and their associated methods and properties is given in subsequent chapters. Each method or property name and associated parameter list is written using Visual Basic syntax.

Note. The parameter naming conventions used in the following sections reflects the underlying IDL (Interface Definition Language) definition of each method and property (see Chapter 11). These naming conventions are not intended to reflect standard Visual Basic naming convention.

Some methods use boolean parameters to specify or return certain settings. Boolean parameters in the ActiveX Drivers have been implemented as long (32bit) Integers. In this way when passing boolean parameters a TRUE setting is defined as a non zero value and a FALSE setting is defined as a zero value. When methods return boolean parameters a TRUE setting will be indicated by the value 1 and a FALSE setting by the value 0.

2.1 Config Object

Name	Purpose
ClearCallLog	Clears the Call Log of all current entries.
ClearErrorLog	Clears the Error Log of all current entries.
GetConfigurationList	Returns the names of available configurations created using the MG17_Config.exe utility.
GetConfigurationListEx	Returns the number and names of available configurations created using the MG17_Config.exe utility.
GetDigitalIONames	Returns the names of DigitalIO modules selected into the current configuration.
GetDigitalIONamesEx	Returns the number and names of DigitalIO modules selected into the current configuration.
GetLastErroInfo	Returns the details of the last error raised.
GetNanoStepNames	Returns the names of NanoStep modules selected into the current configuration.
GetNanoStepNamesEx	Returns the number and names of NanoStep modules selected into the current configuration
GetNanoStepEncNames	Returns the names of Encoded NanoStep modules selected into the current configuration.
GetNanoStepEncNamesEx	Returns the number and names of Encoded NanoStep modules selected into the current configuration
GetNanoTrakNames	Returns the names of NanoTrak modules selected into the current configuration.
GetNanoTrakNamesEx	Returns the number and names of NanoTrak modules selected into the current configuration.
GetNumConfigs	Returns the number of available configurations created using the MG17_Config.exe utility.
GetNumDigitalIOs	Returns the number of DigitalIO modules selected into the current configuration.
GetNumNanoSteps	Returns the number of NanoStep modules selected into the current configuration.

Name	**Purpose**
GetNumNanoStepsEnc	Returns the number of Encoded NanoStep modules selected into the current configuration.
GetNumNanoTraks	Returns the number of NanoTrak modules selected into the current configuration.
GetNumPiezos	Returns the number of Piezo modules selected into the current configuration.
GetNumPowerMeters	Returns the number of Power Meter modules selected into the current configuration.
GetPiezoNames	Returns the names of Piezo modules selected into the current configuration.
GetPiezoNamesEx	Returns the number and names of Piezo modules selected into the current configuration.
GetPowerMeterNames	Returns the names of Power Meter modules selected into the current configuration.
GetPowerMeterNamesEx	Returns the number and names of Power Meter modules selected into the current configuration.
MaxCallLogItems	Sets or Returns the number of entries stored in the Call Log.
MaxErrorLogItems	Sets or Returns the number of entries stored in the Error Log
ReleaseAConfiguration	Releases the current configuration.
SaveCallLog	Saves the information stored in the Call Log to the specified disk file.
SaveErrorLog	Saves the information stored in the Error Log to the specified disk file.
SetupAConfiguration	Loads a named configuration to initialise the ActiveX Drivers
ShowCallLogDlg	Displays the Call Log dialogue box.
ShowErrorLogDlg	Displays the Error Log dialogue box.
ShowErrorLogDlgOnError	Sets or Returns flag indicating if Error Log dialogue box is displayed when an error is raised.

2.2 NanoTraks Object

Name	Purpose
GetAbsolutePower	Returns the absolute power in mW measured by the specified NanoTrak
GetCircleDiameter	Returns the circle diameter and circle control mode currently set for the specified NanoTrak
GetCirclePosition	Returns the circle position currently set for the specified NanoTrak
GetFrequency	Returns the circle scanning frequency currently set for the selected NanoTrak.
GetGain	Returns the feedback loop gain and gain mode currently set for the selected NanoTrak
GetMode	Returns the current tracking operating mode for the specified NanoTrak
GetPhaseOffset	Returns the currently set phase offset (phase compensation) value for the specified NanoTrak
GetRange	Returns the Internal power meter front panel controls state, range setting and mode (auto or manual) for the specified NanoTrak.
GetRelativePower	Returns the relative power measured by the specified NanoTrak
IsItTracking	Returns the tracking status of the specified NanoTrak
SetCircleDiameter	Sets the circle diameter and circle control mode for the specified NanoTrak
SetCirclePosition	Sets the circle position for the specified NanoTrak
SetFrequency	Sets the circle scanning frequency for the selected NanoTrak.
SetGain	Sets the feedback loop gain and gain mode for the selected NanoTrak.
SetMode	Sets the current tracking operating mode for the specified NanoTrak
SetPhaseOffset	Sets the phase offset (phase compensation) value for the specified NanoTrak
SetRange	Sets the Internal power meter front panel controls state, range setting and mode (auto or manual) for the specified NanoTrak.

2.3 Piezos Object

Name	Purpose
GetCurrentMode	Retrieves the current operating mode for the specified Piezo module (closed-loop or open-loop)
GetCurrentPosition	Retrieves the current position, in microns, of the actuator driven by the specified Piezo module
GetDisplay	Retrieves the current display setting (voltage or position) for the specified Piezo module
GetInputs	Retrieves the current analogue input settings for the specified Piezo module
GetInputsEx	Retrieves the current analogue input settings for the specified Piezo module
GetTravel	Retrieves the maximum travel of the actuator connected to the specified Piezo module
GetVoltage	Retrieves the voltage currently applied to the actuator by the specified Piezo module
SetCurrentMode	Sets the operating mode of the specified Piezo module to be closed-loop or open-loop
SetCurrentPosition	Sets the current position of the actuator driven by the specified Piezo module
SetDisplay	Sets the display on the specified Piezo module to read voltage or position
SetInputs	Determines which analogue inputs the specified Piezo module responds to
SetInputsEx	Determines which analogue inputs the specified Piezo module responds to
SetVoltage	Sets the voltage applied to the actuator driven by the specified Piezo module
ZeroSensor	Sets the zero reference for position measurement

2.4 NanoSteps Object

Name	Purpose
AtHome	Waits for all homing NanoSteps to complete their moves
AtPosition	Waits for the last instructed motion of specified Nanosteps to be completed
GetPosition	Retrieves the current absolute positions of the specified Nanosteps
Halt	Stops the specified Nanosteps, immediately or gradually.
Home	Re-initialises the absolute positions of the specified Nanosteps to their pre-set home positions
HomeAndContinue	Moves the specified Nanosteps to their home position, and returns without waiting for the moves to complete
MoveAbsoluteAndContinue	Moves the specified Nanosteps to the given absolute positions, and returns without waiting for the moves to complete
MoveAbsoluteAndContinueEx	Moves the specified Nanosteps to the given absolute positions in the specified direction, and returns without waiting for the moves to complete
MoveAbsoluteAndWait	Moves the specified Nanosteps to the given absolute positions, in the given direction, with optional backlash correction, and only returns when the moves have completed.
MoveAbsoluteAndWaitEx	Moves the specified Nanosteps to the specified absolute positions, with optional backlash correction, and only returns when the moves have completed.
MoveRelativeAndContinue	Moves the specified Nanosteps by the given relative distances, and returns without waiting for the moves to complete
MoveRelativeAndWait	Moves the specified Nanosteps by the given relative distances, with optional backlash correction, and only returns when the moves have completed.
SingleAtPosition	Waits for the last instructed motion of specified single Nanostep to be completed.
SingleGetPosition	Returns the current absolute position of the specified single Nanostep.
SingleGetMinMaxPosition	Returns the minimum and maximum limits (in mm or degrees) of the stage associated with the specified single Nanostep.
SingleGetStageDetails	Returns information (e.g. Stage name, units, zero offset distance etc.) for the stage associated with the specified single Nanostep.
SingleGetVelocityProfile	Returns the velocity profile of the specified single NanoStep.
SingleHalt	Stops the specified single Nanostep, immediately or gradually.
SingleHome	Re-initialises the absolute position of the specified single Nanostep to its pre-set home position.

Name	Purpose
SingleHomeAndContinue	Moves the specified single Nanostep to the home position, and returns without waiting for the move to complete
SingleMoveAbsoluteAndContinue	Moves the specified single Nanostep to the given absolute position, and returns without waiting for the move to complete.
SingleMoveAbsoluteAndWait	Moves the specified single Nanostep to the given absolute position, with optional backlash correction, and only returns when the move has completed.
SingleMoveRelativeAndContinue	Moves the specified single Nanostep by the given relative distance, and returns without waiting for the move to complete
SingleMoveRelativeAndWait	Moves the specified single Nanostep by the given relative distances, with optional backlash correction, and only returns when the move has completed.
SingleSetVelocityProfile	Sets the velocity profile of the specified single NanoStep.
LLCheckNanoStepWhenAtPosition	Wrapper for low level DLL function CheckNanoStepWhenAtPosition.

2.5 Encoded NanoSteps Object

The Encoded Nanosteps object provides all the Methods of the Nanosteps object, together with the following:

Name	Purpose
LLCheckNanoStepEncWhenAtPosition	Wrapper for low level DLL function CheckNanoStepEncWhenAtPosition.
SingleGetStatusFlag	Returns a ‘True’ or ‘False’ value for the specified status flag on the specified single Nanostep.
SingleGetEncodedFlag	Returns a ‘True’ (1) value if the specified single NanostepEnc module is an encoded variant or ‘False’(0) if a non-encoded variant.

2.6 DigIOs Object

Name	Purpose
ReadAllInputs	Reads the states of the input channels on the specified Digital IO module.
ReadAllOutputs	Reads the states of the output channels on the specified Digital IO module.
ReadAnInput	Reads the state of a specified input channel on the specified Digital IO module.
ReadAnOutput	Reads the state of a specified output channel on the specified Digital IO module.
SetAllOutputs	Sets the states of the output channels on the specified Digital IO module.
SetAnOutput	Sets the state of a specified output channel on the specified Digital IUO module.
TurnOff24V	Turns off the 24V supply output on the specified Digital IO module.
TurnOn24V	Turns on the 24V supply output on the specified Digital IO module.

2.7 Optical Power Meter Object

Name	Purpose
AbortCapture	Aborts the capture process for the specified PowerMeter module.
GetAutoRangeLimits	Returns the upper and lower auto ranging limits for the specified PowerMeter module.
GetAttenuationFactor	Retrieves the power attenuation factor for the specified PowerMeter module.
GetCaptureData	Returns the capture data associated with the specified PowerMeter module.
GetCaptureParams	Returns the capture set up parameters for the specified PowerMeter module.
GetCaptureStatus	Returns the status of the capture process associated with the specified PowerMeter module.
GetDisplayBrightness	Retrieves the display brightness setting for the 7-segment and alphanumeric displays on the specified PowerMeter module.
GetFilterInBeamFlag	Retrieves a flag indicating whether an in-beam filter is selected for the specified PowerMeter module.
GetHeadInformation	Returns the model number, serial number and type of the detector head associated with the specified PowerMeter module.
GetLowPassFilter	Retrieves an index value relating to the bandwidth of the low pass filter selected for the specified PowerMeter module.
GetMeasurement	Retrieves the latest power reading for the specified PowerMeter module.
GetMeasurementParams	Retrieves an index representing the type of measurement being returned by the GetMeasurement method for the specified PowerMeter module.
GetPowerSignalOutput	Retrieves an index representing the rack backplane channel used to route the specified PowerMeter module voltage signal.
GetRangeParams	Retrieves the current range and range mode settings of the specified PowerMeter module.
GetResponsivity	Returns the responsivity value for the specified PowerMeter module.
GetWavelength	Retrieves the current wavelength setting for the specified PowerMeter module.
PrimeCapture	Clears the on board data buffer and sets up the specified PowerMeter module to commence capturing samples on receipt of a hardware trigger.
SetAttenuationFactor	Sets the power attenuation factor for the specified PowerMeter module.
SetAutoRangeLimits	Sets the upper and lower auto ranging limits for the specified single PowerMeter module.
SetCaptureParams	Sets the capture parameters for the specified single PowerMeter module.
SetDisplayBrightness	Sets the display brightness for the 7-segment and alphanumeric displays on the specified PowerMeter module.
SetFilterInBeamFlag	Sets the flag indicating whether an in-beam filter is selected for the specified PowerMeter module. (1 = filter selected)
SetLowPassFilter	Sets the bandwidth of the low pass filter selected for the specified PowerMeter module.

SetMeasurementParams	Sets the type of measurement (average, peak, peak-to-peak, dcrms or acrms) for the specified PowerMeter module.
SetPowerSignalOutput	Sets the rack backplane channel used to route the specified PowerMeter module voltage signal.
StartCapture	Clears the on board data buffer and starts the data sampling capture process for the specified single PowerMeter, independently of any pending hardware trigger.
SetRangeParams	Sets the range (30pA full scale to 1mA full scale) and range mode (auto or manual) of the specified PowerMeter module.
SetWavelength	Sets the wavelength for the specified PowerMeter module.

Config Object

The Config object is the main configuration object required by every client application in order to use the nanopositioning modular system. This object contains methods to allow the names of all configurations created using the MG17_Config utility to be retrieved, and for a particular configuration to be set up to initialize the ActiveX Drivers. Methods also exist to allow a client application to extract information on the number, type and names of electronics modules selected into a particular configuration. The Config object also provides methods and properties to enable Call and Error logging capabilities to be used in the development of client applications.

Function ClearCallLog() As Long

Returns

MG return code (see Chapter 11)

Purpose

Clears the Call Log of all current entries.

Details

Refer to ShowCallLogDlg method for a detailed description of the Call Log.

Function ClearErrorLog() As Long

Returns

MG return code (see Chapter 11)

Purpose

Clears the Error Log of all current entries.

Details

Refer to ShowErrorLogDlg method for a detailed description of the Error Log.

Function GetConfigurationList(psabstrNames() As String) As Long

Params

psabstrNames: returned string array of configuration names

Returns

MG return code (see Chapter 11)

Purpose

Returns the names of all available configurations.

Details

A one dimensional string array *psabstrNames* of the correct size (number of elements) needs to be passed to this method in order to receive all available configuration names created using the MG17_Config.exe utility. Use the GetNumConfigs method to obtain the number of configuration names in order to correctly size the string array passed.

See the GetConfigurationListEx method as an alternative way of obtaining configuration names.

The following VisualBasic example is a typical use of the GetConfigurationList and GetNumConfigs methods:-

```
' Object variables already assumed to be defined and referenced.
' Variable used to hold number of configurations.
    Dim lNumConfigs As Long
' Resizable string array used to hold names of configurations.
    Dim strConfigNames() As String
' Variable used to hold return codes.
    Dim lRetCode As Long
' Obtain the number of configurations.
    lRetCode = m_objConfig.GetNumConfigs(lNumConfigs)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
' Resize string array to hold names of configurations.
    ReDim strConfigNames(1 To lNumConfigs)
' Fill resized string array with names of configs.
    lRetCode = m_objConfig.GetConfigurationList(strConfigNames)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
```

Function GetConfigurationListEx(pvConfigNames, plNumConfigs As Long) As Long

Params

pvConfigNames: returned string array of configuration names

plNumConfigs: returned number of configurations

Returns

MG return code (see Chapter 11)

Purpose

Returns the number and names of all available configurations.

Details

The variant parameter *pvConfigNames* passed to this method is converted into a zero based string array of the correct size (number of elements) and filled with the names of all available configurations created using the MG17_Config.exe utility. The *lNumConfigs* parameter is initialized to the number of strings returned in *pvConfigNames*.

See the GetConfigurationList method as an alternative way of obtaining configuration names.

The following VisualBasic example is a typical use of the GetConfigurationListEx method:

```
' Object variables already assumed to be defined and referenced.
' Variable used to hold number of configurations.
    Dim lNumConfigs As Long
' Variant used to hold names of configurations.
    Dim vConfigNames As Variant
' Variable used to hold return codes.
    Dim lRetCode As Long
' Other required variables.
    Dim i As Integer
    Dim strtemp As String
' Fill variant with a string array of configuration names.
    lRetCode = m_objConfig.GetConfigurationListEx(vConfigNames, lNumConfigs)
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
' Access each configuration name returned in a loop.
    For i = LBound(vConfigNames) To UBound(vConfigNames)
    strtemp = vConfigNames(i)
    Next i
```

Function GetDigitalIONames(psabstrNames() As String) As Long

Params

psabstrNames: returned string array of Digital IO module names

Returns

MG return code (see Chapter 11)

Purpose

Returns the names of all Digital IO modules selected into the current configuration.

Details

A one dimensional string array *psabstrNames* of the correct size (number of elements) needs to be passed to this method in order to receive the names of all Digital IO modules selected into the current configuration created using the MG17_Config.exe utility. Use the GetNumDigitalIOs method to obtain the number of Digital IO modules in order to correctly size the string array passed. **Note**. The returned array lists the names in reverse order, i.e. last name first.

See the GetDigitalIONamesEx method as an alternative way of obtaining the names of Digital IO modules.

See the GetConfigurationList method.as an indication of how to use GetDigitalIONames.

Function GetDigitalIONamesEx(pvNames, plNum As Long) As Long

Params

pvNames: returned string array of Digital IO module names

plNum: returned number of Digital IO modules

Returns

MG return code (see Chapter 11)

Purpose

Returns the number and names of all Digital IO modules selected into the current configuration.

Details

The variant parameter *pvNames* passed to this method is converted into a zero based string array of the correct size (number of elements) and filled with the names of all Digital IO modules selected into the current configuration created using the MG17_Config.exe utility. The *plNum* parameter is initialised to the number of strings returned in *pvNames*. **Note**. The returned array lists the names in reverse order, i.e. last name first.

See the GetDigitalIONames method as an alternative way of obtaining the names of Digital IO modules.

Refer to the code sample for GetConfigurationListEx as an indication of how to use GetDigitalIONamesEx.

Function GetLastErrorInfo(pbstrFuncName As String, plErrCode As Long, pbstrErrDesc As String, plInternalIdent As Long) As Long

Params

pbstrFuncName: returned name and parameter list of the last method call that raised an error.

plErrCode: returned code of the last error raised.

pbstrErrDesc: returned description of the last error raised.

plInternalIdent: returned internal identifier of the last error raised.

Returns

MG return code (see Chapter 11)

Purpose

Returns the details of the last error raised by the ActiveX Drivers.

Details

This method can be used to retrieve useful information about the latest error raised by the ActiveX Drivers. This method is typically called when a non zero code has been returned by the Drivers. The string parameter *pbstrFuncName* contains the name and parameter list of the method call that raised the error. A brief description of the error that occurred is returned in the pbstrErrDesc string parameter and the associated return (error) code is returned in plErrCode. The *plInternalIdent* parameter contains an internal identifier that should be quoted to Melles Griot in the event of any support request relating to unresolved errors raised by the ActiveX Drivers.

Refer to the ShowErrorLogDlg method for a detailed description of the error logging capabilities of the ActiveX Drivers.

Function GetNanoStepNames(psabstrNames() As String) As Long

Params

psabstrNames: returned string array of NanoStep module names

Returns

MG return code (see Chapter 11)

Purpose

Returns the names of all NanoStep modules selected into the current configuration.

Details

A one dimensional string array *psabstrNames* of the correct size (number of elements) needs to be passed to this method in order to receive the names of all NanoStep modules selected into the current configuration created using the MG17_Config.exe utility. Use the GetNumNanoSteps method to obtain the number of NanoStep modules in order to correctly size the string array passed. **Note**. The returned array lists the names in reverse order, i.e. last name first.

See the GetNanoStepNamesEx method as an alternative way of obtaining the names of NanoStep modules.

Refer to the code sample for GetConfigurationList as an indication of how to use GetNanoStepNames.

Function GetNanoStepNamesEx(pvNames, plNum As Long) As Long

Params

pvNames: returned string array of NanoStep module names

plNum: returned number of NanoStep modules

Returns

MG return code (see Chapter 11)

Purpose

Returns the number and names of all NanoStep modules selected into the current configuration.

Details

The variant parameter *pvNames* passed to this method is converted into a zero based string array of the correct size (number of elements) and filled with the names of all NanoStep modules selected into the current configuration created using the MG17_Config.exe utility. The *plNum* parameter is Initialised to the number of strings returned in pvNames. **Note**. The returned array lists the names in reverse order, i.e. last name first.

See the GetNanoStepNames method as an alternative way of obtaining the names of NanoStep modules.

Refer to the code sample for GetConfigurationListEx as an indication of how to use GetNanoStepNamesEx.

Function GetNanoStepEncNames(psabstrNames() As String) As Long

Params

psabstrNames: returned string array of Encoded NanoStep module names

Returns

MG return code (see Chapter 11)

Purpose

Returns the names of all Encoded NanoStep modules selected into the current configuration.

Details

A one dimensional string array *psabstrNames* of the correct size (number of elements) needs to be passed to this method in order to receive the names of all encoded NanoStep modules selected into the current configuration created using the MG17_Config.exe utility. Use the GetNumNanoStepsEnc method to obtain the number of encoded NanoStep modules in order to correctly size the string array passed. **Note**. The returned array lists the names in reverse order, i.e. last name first.

See the GetNanoStepEncNamesEx method as an alternative way of obtaining the names of encoded NanoStep modules.

Refer to the code sample for GetConfigurationList as an indication of how to use GetNanoStepEncNames.

Function GetNanoStepEncNamesEx(pvNames, plNum As Long) As Long

Params

pvNames: returned string array of encoded NanoStep module names

plNum: returned number of encoded NanoStep modules

Returns

MG return code (see Chapter 11)

Purpose

Returns the number and names of all encoded NanoStep modules selected into the current configuration.

Details

The variant parameter *pvNames* passed to this method is converted into a zero based string array of the correct size (number of elements) and filled with the names of all encoded NanoStep modules selected into the current configuration created using the MG17_Config.exe utility. The *plNum* parameter is Initialised to the number of strings returned in pvNames. **Note**. The returned array lists the names in reverse order, i.e. last name first.

See the GetNanoStepEncNames method as an alternative way of obtaining the names of encoded NanoStep modules.

Refer to the code sample for GetConfigurationListEx as an indication of how to use GetNanoStepEncNamesEx.

Function GetNanoTrakNames(psabstrNames() As String) As Long

Params

psabstrNames: returned string array of NanoTrak module names

Returns

MG return code (see Chapter 11)

Purpose

Returns the names of all NanoTrak modules selected into the current configuration.

Details

A one dimensional string array *psabstrNames* of the correct size (number of elements) needs to be passed to this method in order to receive the names of all NanoTrak modules selected into the current configuration created using the MG17_Config.exe utility. Use the GetNumNanoTraks method to obtain the number of Digital IO modules in order to correctly size the string array passed. **Note**. The returned array lists the names in reverse order, i.e. last name first.

See the GetNanoTrakNamesEx method as an alternative way of obtaining the names of NanoTrak modules.

Refer to the code sample for GetConfigurationList as an indication of how to use GetNanoTrakNames.

Function GetNanoTrakNamesEx(pvNames, plNum As Long) As Long

Params

pvNames: returned string array of NanoTrak module names

plNum: returned number of NanoTrak modules

Returns

MG return code (see Chapter 11)

Purpose

Returns the number and names of all NanoTrak modules selected into the current configuration.

Details

The variant parameter *pvNames* passed to this method is converted into a zero based string array of the correct size (number of elements) and filled with the names of all NanoTrak modules selected into the current configuration created using the MG17_Config.exe utility. The *plNum* parameter is initialised to the number of strings returned in pvNames. **Note**. The returned array lists the names in reverse order, i.e. last name first.

See the GetNanoTrakNames method as an alternative way of obtaining the names of NanoTrak modules. Refer to the code sample for GetConfigurationListEx as an indication of how to use GetNanoTrakNamesEx.

Function GetNumConfigs(plNum As Long) As Long

Params

plNum: returned number of available configurations

Returns

MG return code (see Chapter 11)

Purpose

Returns the number of all available configurations.

Details

The long parameter *plNum* passed to this method is initialised with the number of all available configurations created using the MG17_Config.exe utility. This method is typically used in conjunction with the GetConfigurationList method in order to retrieve the names of all available configurations.

Refer to the code sample for GetConfigurationList as an indication of how to use the GetNumConfigs method.

Function GetNumDigitalIOs(plNum As Long) As Long

Params

plNum: returned number of Digital IO modules

Returns

MG return code (see Chapter 11)

Purpose

Returns the number of all Digital IO modules selected into the current configuration.

Details

The long parameter *plNum* passed to this method is initialised with the number of all Digital IO modules selected into the current configuration created using the MG17_Config.exe utility. This method is typically used in conjunction with the GetDigitalIONames method in order to retrieve the names of Digital IO modules.

Refer to GetDigitalIONames for further information on retrieving the names of Digital IO modules.

Function GetNumNanoSteps(plNum As Long) As Long

Params

plNum: returned number of NanoStep modules

Returns

MG return code (see Chapter 11)

Purpose

Returns the number of all NanoStep modules selected into the current configuration.

Details

The long parameter *plNum* passed to this method is initialised with the number of all NanoStep modules selected into the current configuration created using the MG17_Config.exe utility. This method is typically used in conjunction with the GetNanoStepNames method in order to retrieve the names of NanoStep modules.

Refer to GetNanoStepNames for further information on retrieving the names of NanoStep modules.

Function GetNumNanoStepsEnc(plNum As Long) As Long

Params

plNum: returned number of Encoded NanoStep modules

Returns

MG return code (see Chapter 11)

Purpose

Returns the number of all Encoded NanoStep modules selected into the current configuration.

Details

The long parameter *plNum* passed to this method is initialised with the number of all Encoded NanoStep modules selected into the current configuration created using the MG17_Config.exe utility. This method is typically used in conjunction with the GetNanoStepEncNames method in order to retrieve the names of Encoded NanoStep modules.

Refer to GetNanoStepEncNames for further information on retrieving the names of Encoded NanoStep modules.

Function GetNumNanoTraks(plNum As Long) As Long

Params

plNum: returned number of NanoTrak modules

Returns

MG return code (see Chapter 11)

Purpose

Returns the number of all NanoTrak modules selected into the current configuration.

Details

The long parameter *plNum* passed to this method is Initialised with the number of all NanoTrak modules selected into the current configuration created using the MG17_Config.exe utility. This method is typically used In conjunction with the GetNanoTrakNames method in order to retrieve the names of NanoTrak modules.

Refer to GetNanoTrakNames for further information on retrieving the names of NanoTrak modules.

Function GetNumPiezos(plNum As Long) As Long

Params

plNum: returned number of Piezo modules

Returns

MG return code (see Chapter 11)

Purpose

Returns the number of all Piezo modules selected into the current configuration.

Details

The long parameter *plNum* passed to this method is initialised with the number of all Piezo modules selected into the current configuration created using the MG17_Config.exe utility. This method is typically used in conjunction with the GetPiezoNames method in order to retrieve the names of Piezo modules.

Refer to GetPiezoNames for further information on retrieving the names of Piezo modules.

Function GetNumPowerMeters(plNum As Long) As Long

Params

plNum: returned number of Power Meter modules

Returns

MG return code (see Chapter 11)

Purpose

Returns the number of all Mower Meter modules selected into the current configuration.

Details

The long parameter *plNum* passed to this method is initialised with the number of all Power Meter modules selected into the current configuration created using the MG17_Config.exe utility. This method is typically used in conjunction with the GetPowerMeterNames method in order to retrieve the names of Power Meter modules.

Refer to GetPowerMeterNames for further information on retrieving the names of Power Meter modules.

Function GetPiezoNames(psabstrNames() As String) As Long

Params

psabstrNames: returned string array of Piezo module names

Returns

MG return code (see Chapter 11)

Purpose

Returns the names of all Piezo modules selected into the current configuration.

Details

A one dimensional string array *psabstrNames* of the correct size (number of elements) needs to be passed to this method in order to receive the names of all Piezo modules selected into the current configuration created using the MG17_Config.exe utility. Use the GetNumPiezos method to obtain the number of Piezo modules in order to correctly size the string array passed. **Note**. The returned array lists the names in reverse order, i.e. last name first.

See the GetPiezoNamesEx method as an alternative way of obtaining the names of Piezo modules.

Refer to the code sample for GetConfigurationList as an indication of how to use GetPiezoNames.

Function GetPiezoNamesEx(pvNames, plNum As Long) As Long

Params

pvNames: returned string array of Piezo module names

plNum: returned number of Piezo modules

Returns

MG return code (see Chapter 11)

Purpose

Returns the number and names of all Piezo modules selected into the current configuration.

Details

The variant parameter *pvNames* passed to this method is converted into a zero based string array of the correct size (number of elements) and filled with the names of all Piezo modules selected into the current configuration created using the MG17_Config.exe utility. The *plNum* parameter is initialised to the number of strings returned in pvNames. **Note**. The returned array lists the names in reverse order, i.e. last name first.

See the GetPiezoNames method as an alternative way of obtaining the names of Piezo modules.

Refer to the code sample for GetConfigurationListEx as an indication of how to use GetPiezoNamesEx.

Function GetPowerMeterNames(psabstrNames() As String) As Long

Params

psabstrNames: returned string array of Power Meter module names

Returns

MG return code (see Chapter 11)

Purpose

Returns the names of all Power Meter modules selected into the current configuration.

Details

A one dimensional string array *psabstrNames* of the correct size (number of elements) needs to be passed to this method in order to receive the names of all Power Meter modules selected into the current configuration created using the MG17_Config.exe utility. Use the GetNumPowerMeters method to obtain the number of Power Meter modules in order to correctly size the string array passed. **Note**. The returned array lists the names in reverse order, i.e. last name first.

See the GetPowerMeterNamesEx method as an alternative way of obtaining the names of Power Meter modules.

Refer to the code sample for GetConfigurationList as an indication of how to use GetPowerMeterNames.

Function GetPowerMeterNamesEx(pvNames, plNum As Long) As Long

Params

pvNames: returned string array of Power Meter module names

plNum: returned number of Power Meter modules

Returns

MG return code (see Chapter 11)

Purpose

Returns the number and names of all Power Meter modules selected into the current configuration.

Details

The variant parameter *pvNames* passed to this method is converted into a zero based string array of the correct size (number of elements) and filled with the names of all Power Meter modules selected into the current configuration created using the MG17_Config.exe utility. The *plNum* parameter is initialised to the number of strings returned in pvNames. **Note**. The returned array lists the names in reverse order, i.e. last name first.

See the GetPowerMeterNames method as an alternative way of obtaining the names of Power Meter modules.

Refer to the code sample for GetConfigurationListEx as an indication of how to use GetPowerMeterNamesEx.

Property MaxCallLogItems As Long

Purpose

Sets or returns the maximum number of entries stored in the Call Log. Range 1 to 5000.

Details

This property sets or returns the maximum number of entries stored in the ActiveX Drivers Call Log. Once the maximum number of entries has been reached, older entries are discarded in order to free storage for each new entry. In this way the Call Log will contain entries for the last n calls made into the Drivers where n does not exceed the value specified by MaxCallLogItems.

Refer to the ShowCallLogDlg method for more details on the call logging functionality of the ActiveX Drivers.

Property MaxErrorLogItems As Long

Purpose

Sets or returns the maximum number of entries stored in the Error Log. Range 1 to 500.

Details

This property sets or returns the maximum number of entries stored in the ActiveX Drivers Error Log. Once the maximum number of entries has been reached, older entries are discarded in order to free storage for each new entry. In this way the Error Log will contain entries for the last n calls made into the Drivers where n does not exceed the value specified by MaxErrorLogItems.

Refer to the ShowErrorLogDlg method for more details on the error logging functionality of the ActiveX Drivers.

Function ReleaseAConfiguration() As Long

Params

None

Returns

MG return code (see Chapter 11)

Purpose

Releases the current configuration.

Details

This method should be called to free up system resources allocated when the SetUpAConfiguration method is called. Typically ReleaseAConfiguration will be called by a client application when it no longer requires access to the ActiveX Drivers, usually when the application is shutting down.

Refer to SetUpAConfiguration for further details on setting up a configuration.

Function SaveCallLog(bstrPathName As String) As Long

Params

bstrPathName: path and filename of the text file to create.

Returns

MG return code (see Chapter 11)

Purpose

Saves the current entries in the ActiveX Drivers Call Log to a text file.

Details

This method saves the information about each entry in the Call Log to a text file with a name and location specified by the *bstrPathName* string parameter. Note that if this method is called with the same path and name as an existing file, this file will be overwritten (not appended) with the new Call Log Information. If the path and filename passed is not valid for any reason (e.g. drive letter does not exist) then the SaveCallLog method will fail with a non zero return code.

The following VisualBasic example is a typical use of the Call Log file.

```
' Object variables already assumed to be defined and referenced.
' Variable used to hold return codes.
    Dim lRetCode As Long
' Save Call Log Information to a text file called TestCallLog.txt.
    lRetVal = m_objConfig.SaveCallLog("c:\TestCallLog.txt")
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
```

The following listing shows the typical contents of a Call Log file:-

```
Call Log generated 11:55:28 Thursday, January 04, 2001
Melles Griot MG17_Drivers ActiveX Server (Built: Jan. 4 2001 10:20:24).
Server Version:- 5:3:0:1.
11:55:09:0930> Config.ShowCallLogDlg()
11:55:06:0470> Config.ShowErrorLogDlg()
11:55:00:0870> Piezos.SetCurrentPosition('P 1',3.000000)
11:54:57:0190> Piezos.SetDisplay('P 1',1)
11:54:53:0070> Piezos.SetCurrentMode('P 1',1)
11:54:47:0470> NanoTraks.SetCircleDiameter('NT 1',2.000000,1)
11:54:30:0880> NanoSteps.SingleSetVelocityProfile('NS 1','1.000000' '2.000000' '2.000000')
11:54:18:0630> NanoSteps.AtPosition('NS 3' 'NS 2' 'NS 1', '5.000000' '5.000000' '5.000000')
11:54:17:0640> NanoSteps.MoveRelativeAndContinue('NS 3' 'NS 2' 'NS 1', '0.500000' '0.500000' '0.500000',rv)
11:54:13:0630> NanoSteps.Home('NS 3', 'NS 2', 'NS 1')
11:54:03:0910> Config.GetDigitalIONames(rv)
11:54:03:0910> Config.GetNumDigitalIOs(rv)
11:54:03:0310> Config.GetPiezoNames(rv)
11:54:03:0310> Config.GetNumPiezos(rv)
11:54:02:0810> Config.GetNanoTrakNames(rv)
11:54:02:0810> Config.GetNumNanoTraks(rv)
11:53:59:0460> Config.GetNanoStepNames(rv)
11:53:59:0460> Config.GetNumNanoSteps(rv)
11:53:48:0260> Config.SetupAConfiguration('MYCONFIG2')
11:53:42:0710> rv = Config.MaxCallLogItems
11:53:42:0660> rv = Config.MaxErrorLogItems
11:53:42:0660> Config.ShowErrorLogDlgOnError = 1
11:53:42:0660> rv = Config.ShowErrorLogDlgOnError
```

Refer to the ShowCallLogDlg method for more details on the call logging functionality of the ActiveX Drivers.

Function SaveErrorLog(bstrPathName As String) As Long

Params

bstrPathName: path and filename of the text file to create.

Returns

MG return code (see Chapter 11)

Purpose

Saves the current entries in the ActiveX Drivers Error Log to a text file.

Details

This method saves the information about each entry in the Error Log to a text file with a name and location specified by the *bstrPathName* string parameter. Note that if this method is called with the same path and name as an existing file, this file will be overwritten (not appended) with the new Error Log Information. If the path and filename passed is not valid for any reason (e.g. drive letter does not exist) then the SaveErrorLog method will fail with a non zero return code.

The following listing shows the typical contents of an Error Log file:-

Error Log generated 11:59:10 Thursday, January 04, 2001

Melles Griot MG17_Drivers ActiveX Server (Built: Jan. 4 2001 10:20:29).

Server Version:- 5:3:0:1.

11:58:41:0670> NanoSteps.SingleMoveRelativeAndWait

('NS 3',1000.000000,0)

Error Code = 8514; Internal Error Code = 16410507.

Description:-

Invalid High Level Parameter [MG17_HiLevel.dll].

Notes:-

An attempt has been made to call a high level DLL function passing a parameter that is invalid or out of range. In the case of motor related commands this error may occur for move requests that exceed the stage travel or that exceed the range of calibration data (if used).

Extra Info:-

Internal error occurred calling StartNanoStepRelativeDistance

('NS 3',1,1000.000000) for NanoStep module NS 3.

11:57:59:0650> Piezos.SetVoltage('P 1',4.000000)

Error Code = 8771; Internal Error Code = 12371007.

Description:-

Set Voltage Invalid [MG17_Driver.dll].

Notes:-

The output voltage cannot be set when the Piezo module is in 'closed loop' mode.

Extra Info:-

No extra error information.

11:55:37:0400> NanoTraks.SetCircleDiameter('NT 1',3.000000,1)

Error Code = 8955; Internal Error Code = 12170301.

Description:-

Invalid Driver Level Parameter [MG17_Driver.dll].

Notes:-

An attempt has been made to call a Driver method passing a parameter that is invalid or out of range.

Extra Info:-

Diameter parameter out of range for NanoTrak 'NT 1' [range 0.0 to 2.5].

Refer to the ShowErrorLogDlg method for more details on the error logging functionality of the ActiveX Drivers.

Function SetupAConfiguration(bstrConfigName As String) As Long

Params

bstrConfigName: name of configuration to set up.

Returns

MG return code (see Chapter 11)

Purpose

Loads a named configuration to initialize the ActiveX Drivers.

Details

This method is called to initialize the ActiveX Drivers with information contained in the configuration specified by the *bstrConfigName* string parameter.

A configuration describes the elements of a nanopositioning system: the modules used and the equipment connected to them. A set of equipment may have several configurations, depending on the job it is performing at the time. Each one allocates the type of actuators connected, their names, and the default properties for them. Configurations are created and saved using the '17_Config.exe' application as described in the 'Main Rack and Controller' manual HA0088.

Once a configuration is saved under a meaningful name, a client application can refer to it, and also to the control modules contained within it. The following code (shown in VisualBasic as an example) must always be present within a client application in order to initialize the ActiveX Drivers with the named configuration:-

```
' Set up a named configuration created using the MG17_Config utility.
' This call initializes the ActiveX Drivers
    lRetCode = m_objConfig.SetupAConfiguration("CONFIG1")
    If lRetCode <> MG_RET_CODES.MG17_OK Then
' Method failure. Do error handling here.
    End If
```

This method must always be called before calling any of the hardware module specific methods and properties exposed by the ActiveX Driver objects.

Prior to calling SetupAConfiguration, the list of configuration names created using the 'MG17_Config.exe' utility can be obtained using the GetConfigurationList and GetConfigurationListEx methods.

When software control of the modular electronics is no longer required (e.g. when the application shuts down) the memory and resources allocated by SetupAConfiguration must be freed by calling ReleaseAConfiguration.

Function ShowCallLogDlg() As Long

Params

None

Returns

MG return code (see Chapter 11)

Purpose

Displays the Call Log dialogue box.

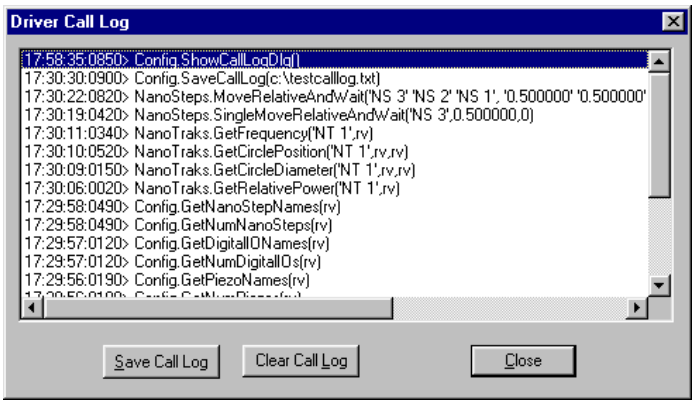
Details

The ActiveX Drivers have been designed to include full Call logging capabilities. To this end the details of each method call made into the Drivers are stored In a Call Log. These details include the time the call was made as well as the name of the method and the parameter values passed (if any). This Call Log can prove useful during the development of a client application to confirm the sequence of actions carried out by the ActiveX Drivers and check that this occurs as expected. It can also prove useful to confirm the integrity of any parameters that are passed through to the Drivers. The maximum number of Items that can be stored in the call log is limited (for resource reasons) to the value specified using the MaxCallLogItems property. If the number of calls made exceed this maximum value then older entries are discarded to free up memory to allow new entries to be added to the Call Log.

The information stored in the Call Log can be viewed at any time during execution of a client application by calling the ShowCallLog method.

This method displays the Call Log dialogue as shown in the following screen shot:

-



For each Call Log entry, the time the call was made is displayed followed by the name and parameter list of the method. Note that some parameters are marked up as 'rv' referring to those variables that are used as place holders for 'returned values'. The Call Log can be cleared at any time by using the **Clear Call Log** button on the Call Log dialogue or by using the ClearCallLog method. The information stored in the Call Log can be saved to a text file at any time by using the **Save Call Log** button on the Call Log dialogue or by calling the SaveCallLog method passing through the path and name of a text file.

Function ShowErrorLogDlg() As Long

Params

None

Returns

MG return code (see Chapter 11)

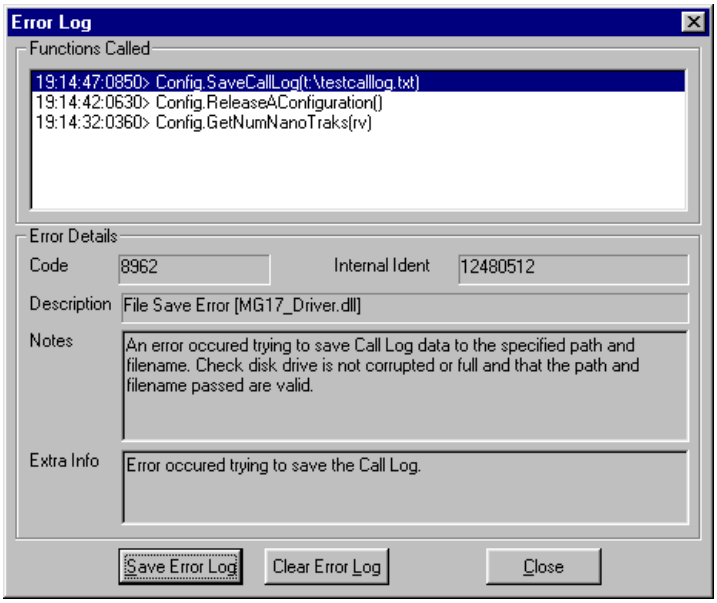
Purpose

Displays the Error Log dialogue box.

Details

The ActiveX Drivers have been designed to include full Error logging capabilities. In this regard the details of any error raised by the Drivers during execution of a client application are logged in an Error Log. Details about each error raised include the time the call that generated the error was made, the name of the method and the parameter values passed (if any) and relevant textual descriptions of the error that occurred. The maximum number of Items that can be stored in the Error Log is limited (for resource reasons) to the value specified using the MaxErrorLogItems property. If the number of errors generated exceed this maximum value then older entries are discarded to free up memory to allow new entries to be added to the Error Log.

The information stored in the Error Log can be viewed at any time during execution of a client application by calling the ShowErrorLog method. This method displays the Error Log dialogue as shown in the following screen shot:-



For each Error Log entry, the time the call that generated the error was made is displayed followed by the name and parameter list of the method. Those parameters marked up as 'rv' referring to variables that are used as place holders for 'returned values'. In addition, the Error Log dialogue box displays the 4 digit method return code in the **Code** field and an Internal Ident code In the **Internal Ident** field. The 4 digit return code and internal ident code should be quoted to Melles Griot in the event of a technical support request relating to unresolved errors being generated by the ActiveX Drivers. The Error Log dialogue also displays textual Information describing the error in the **Description**, **Notes** and **Extra Info** fields. By clicking on a particular method name in the **Functions Called** list, the relevant information fields in the lower half of the dialogue box are updated. The Error Log can be cleared at any time by using the **Clear Error Log** button on the Error Log dialogue or by using the ClearErrorLog method. The information stored in the Error Log can be saved to a text file at any time by using the **Save Error Log** button on the Error Log dialogue or by calling the SaveErrorLog method passing through the path and name of a text file.

Property ShowErrorLogDlgOnError As Long

Purpose

Sets or Returns flag indicating if Error Log dialogue box is displayed when an error is raised.

Details

By default the ActiveX Drivers will display the Error Log dialogue box whenever an error occurs. When this dialogue is closed the method call that generated the error returns with the relevant return code. During development of a client application this default behavior can be useful for debugging purposes. Generally however, a client application will be written to handle errors and inform the user/operator what action to take when an error occurs. In this instance it may be appropriate to suppress the display of the Error Log dialogue box when an error occurs. By setting the ShowErrorLogDlgOnError property to FALSE the ActiveX Drivers will not display the Error Log dialogue box when an error condition occurs. The method call that generated the error will still return with the appropriate return code to allow the client application to deal with the error.

Refer to the ShowErrorLogDlg method for more details on the error logging functionality of the ActiveX Drivers.

NanoTraks Object

The NanoTraks object provides the functionality required for a client application to control one or more NanoTrak modules fitted to a system. After setting up a configuration using the Config object, use the methods of the NanoTraks object to carry out activities such as latching/unlatching, reading power levels, obtaining/setting circle size and position and determining if 'Nano-Tracking' is currently taking place. To supplement the information contained in this NanoTraks object section please refer to *handbook HA0086 - NanoTrak Module* for further operating instructions.

All method calls on the NanoTraks object require a string parameter to be passed containing a valid name (assigned using the MG17_Config.exe utility) of a NanoTrak module selected into the current configuration. Valid string names of all NanoTraks selected into the current configuration can be obtained by calling the GetNanoTrakNames or GetNanoTrakNamesEx methods of the Config object.

Function GetAbsolutePower(bstrName As String, pfAbsolutePower As Double) As Long

Params

bstrName: name of specific NanoTrak module

pfAbsolutePower: returned absolute power value

Returns

MG return code (see Chapter 11)

Purpose

Returns the absolute power in mW measured by the specified NanoTrak.

Details

This method obtains the absolute power reading in the *pfAbsolutePower* double parameter from the NanoTrak module specified by the *bstrName* string parameter.

The absolute power is calculated on the basis of the relative input power, the power meter range setting and a PIN diode input with detector sensitivity of 0.164 A/W. It is an approximate calculation and is not intended to represent a fully calibrated reading. As with manual operation, the user must ensure when the power meter range setting is at a fixed value, that the relative input power reading does not saturate.

Function GetCircleDiameter(bstrName As String, pfDiameter As Double, plMode As Long) As Long

Params

bstrName: name of specific NanoTrak module

pfDiameter: returned circle diameter value

plMode: returned circle adjustment mode

Returns

MG return code (see Chapter 11)

Purpose

Returns the circle diameter and adjustment mode currently set for the specified NanoTrak.

Details

This method obtains the circle diameter setting in the *pfDiameter* double parameter and the circle adjustment mode setting in the *plMode* long parameter from the NanoTrak module specified by the *bstrName* string parameter.

The circle diameter is measured on a scale (NanoTrak units) such that 10 units is the width and height of the screen. If, for example, the NanoTrak is driving a 20 micron actuator, a circle diameter of 1 unit will result in a real diameter of 2 microns. The scanning circle diameter can have values in the range 0 to 2.5 with a value of 2.5 corresponding to one quarter the range of piezo motion. The diameter of the scanning circle can be adjusted in three ways, each exclusive of the others. In this regard, the *plMode* adjustment mode parameter can have one of the following values:-

0 Potentiometer (front panel) control

1 Software control

2 Automatic sizing dependent on the Input power.

Note. The scanning circle diameter and mode can be set using the SetCircleDiameter method.

Function GetCirclePosition(bstrName As String, pfHorizontalPosition As Double, pfVerticalPosition As Double) As Long

Params

bstrName: name of specific NanoTrak module

pfHorizontalPosition: returned circle horizontal position

pfVerticalPosition: returned circle vertical position

Returns

MG return code (see Chapter 11)

Purpose

Returns the circle position currently set for the specified NanoTrak.

Details

This method obtains the horizontal and vertical circle position in the *pfHorizontalPosition* and *pfVerticalPosition* double parameters respectively from the NanoTrak module specified by the *bstrName* string parameter.

Both position outputs can have values in the range 0.0 – 10.0 (NanoTrak units), with (5.0,5.0) representing the display's origin (screen center).

Function GetFrequency(bstrName As String, pfFrequency As Double) As Long

Params

bstrName: name of specific NanoTrak module

pfFrequency: returned circle frequency

Returns

MG return code (see Chapter 11)

Purpose

Returns the circle scanning frequency currently set for the specified NanoTrak.

Details

This method obtains the circle scanning frequency in the *pfFrequency* double parameter from the NanoTrak module specified by the *bstrName* string parameter.

The circle scanning frequency lies in the range 0 to 250Hz. The factory default setting for the scanning frequency is 42Hz. This means that a stage driven by the NanoTrak makes 42 circular movements per second. Different frequency settings allow more than one NanoTrak to be used in the same alignment scenario. Refer to the NanoTrak handbook (HA0086) for more information.

Function GetGain(bstrName As String, pbAutomatic As Long, pfGain As Double) As Long

Params

bstrName: name of specific NanoTrak module

pbAutomatic: returned gain mode

pfGain: returned gain setting

Returns

MG return code (see Chapter 11)

Purpose

Returns the feedback loop gain and gain mode currently set for the specified NanoTrak.

Details

This method obtains the feedback loop gain and gain mode settings in the *pfGain* double and *pbAutomatic* long parameters respectively from the NanoTrak module specified by the *bstrName* string parameter.

The NanoTrak module exposes a gain setting that is used to ensure the DC level of the input (feedback loop) signal lies within the dynamic range of the input. The NanoTrak module can be set to auto gain control (AGC) such that the gain is adjusted automatically during operation. The pbAutomatic parameter returns as TRUE if auto gain control mode is enabled or FALSE if fixed gain mode is enabled. The gain setting in fixed gain mode is returned In the parameter pfGain and lies in the range 0.1 to 200.0 (with a factory set default value of 10.0). The gain mode (fixed or auto) and gain value (fixed gain mode only) can be set using the SetGain method.

Function GetMode(bstrName As String, pbTrack As Long, pbTracking As Long, pbSingleAxis As Long) As Long

Params

bstrName: name of specific NanoTrak module

pbTrack: returned tracking-latched status

pbTracking: returned tracking status

pbSingleAxis: returned circle axis mode

Returns

MG return code (see Chapter 11)

Purpose

Returns the current tracking operating mode for the specified NanoTrak.

Details

This method obtains the tracking mode settings from the NanoTrak module specified by the *bstrName* string parameter. The *pbTrack* boolean parameter indicates if the NanoTrak module is set to latched (FALSE) or track/ unlatched (TRUE). If the NanoTrak module is unlatched the *pbTracking* boolean parameter Indicates if it is currently tracking on an input (feedback) signal (TRUE) or not (FALSE). The *pbSingleAxis* boolean parameter indicates if the scanning circle is currently set to single axis (TRUE) or dual axis (FALSE) mode. In single axis mode the circular scan becomes a horizontal line scan. These six modes of operation when an input signal is connected are summarized in table 4.1

Table 4.1 Modes of operation

	Track mode	Tracking	Single axis
Latch, dual axis	F	F	F
Track, dual axis, not yet tracking	T	F	F
Track, dual axis, tracking	T	T	F
Latch, single axis	F	F	T
Track, single axis, not yet tracking	T	F	T
Track, single axis, tracking	T	T	T

Note. The latched state and single axis mode can be set using the SetMode method.

Function GetPhaseOffset(bstrName As String, pfPhaseOffset As Double) As Long

Params

bstrName: name of specific NanoTrak module

pfPhaseOffset: returned phase offset setting

Returns

MG return code (see Chapter 11)

Purpose

Returns the phase offset (phase compensation) value currently set for the specified NanoTrak.

Details

The feedback loop scenario in a typical NanoTrak application will involve the operation of various electronic and electromechanical components (e.g. power meters and piezo actuators) that can introduce phase shifts around the loop. The phase compensation setting is a means of cancelling unwanted phase shifts in the system and therefore improving tracking stability.

This method obtains the phase offset in the *pfPhaseOffset* double parameter from the NanoTrak module specified by the *bstrName* string parameter. The phase offset value is specified in degrees and lies in the range -90.0 to 90.0 degrees.

The phase offset can be set using the SetPhaseOffset method.

Function GetRange(bstrName As String, pbFrontPanelDisabled As Long, plRangeValue As Long, pbAutoRange As Long) As Long

Params

bstrName: name of specific NanoTrak module

pbFrontPanelDisabled: returned range button disabled status

plRangeValue: returned range setting

pbAutoRange: returned ranging mode

Returns

MG return code (see Chapter 11)

Purpose

Returns the internal power meter front panel controls state, range setting (1 - 8), and ranging mode (auto or manual) for the specified NanoTrak.

Details

The NanoTrak unit is equipped with an internal power meter and associated front panel range/power level displays and control buttons. This power meter operates when an external detector head is connected to the PIN connector on the front panel. There are 8 range settings (1 - 8) that can be used to select the best range to measure the relative input power (displayed on the front panel relative input power LED bar display). The range can be set using front panel 'up' and 'down' buttons. Ranges 1-8 refer to the power range settings given in the ‘Specifications’ section of the handbook for the NanoTrak Control Module. Note that an auto-ranging mode can be enabled (with the front panel 'auto' button) so that range changes occur whenever the relative input power signal reaches the upper or lower end of the currently set range.

This method obtains information regarding the range status of the internal power meter on the NanoTrak module specified by the *bstrName* string parameter. The *pbFrontPanelDisabled* boolean parameter indicates the enabled/disabled status of the front panel range control buttons (up, down and auto). The control buttons are disabled (locked) if pbFrontPanelDisabled returns as TRUE and enabled if FALSE. The *plRangeValue* parameter indicates the currently set range (1 - 8) and the *pbAutoRange* parameter indicates if the auto ranging is enabled (TRUE) or disabled (FALSE).

The ranging mode (auto or fixed) and range setting can be set using the SetRange method.

Function GetRelativePower(bstrName As String, pfRelativePower As Double) As Long

Params

bstrName: name of specific NanoTrak module

pfRelativePower: returned relative power

Returns

MG return code (see Chapter 11)

Purpose

Returns the relative power level for the specified NanoTrak.

Details

This method returns the relative power level in the pfRelativePower double parameter for the NanoTrak module specified by the bstrName string parameter. This power reading lies in the range 0.0 to 10.0 and corresponds to the extension of the relative input power LED bar display on the front panel. Note that the BNC power meter input on the front panel has a 100% overload capability that can result in relative power level values in the range 10.0 to 20.0 being returned by this method even though the LED bar display indicates a saturated input.

Function IsItTracking(bstrName As String, pbTracking As Long) As Long

Params

bstrName: name of specific NanoTrak module

pbTracking: returned tracking status

Returns

MG return code (see Chapter 11)

Purpose

Returns the tracking status of the specified NanoTrak.

Details

The *pbTracking* flag is returned as TRUE if the NanoTrak is tracking, FALSE otherwise. If the NanoTrak is tracking the green tracking LED on the front panel illuminates. This occurs when the NanoTrak is in tracking mode and a certain input power threshold is exceeded.

Function SetCircleDiameter(bstrName As String, fDiameter As Double, lMode As Long) As Long

Params

bstrName: name of specific NanoTrak module

fDiameter: circle diameter value

lMode: circle adjustment mode

Returns

MG return code (see Chapter 11)

Purpose

Sets the circle diameter and adjustment mode for the specified NanoTrak.

Details

This method sets the circle diameter and adjustment mode contained in the *fDiameter* double and lMode long parameters respectively for the NanoTrak module specified by the *bstrName* string parameter.

The circle diameter is measured on a scale (NanoTrak units) such that 10 units is the width and height of the screen. If, for example, the NanoTrak is driving a 20 micron actuator, a circle diameter of 1 unit will result in a real diameter of 2 microns. The scanning circle diameter can have values in the range 0 to 2.5 with a value of 2.5 corresponding to one quarter the range of piezo motion. The diameter of the scanning circle can be adjusted in three ways, each exclusive of the others. These three adjustment modes are specified by the *lMode* parameter as follows:-

0 Potentiometer (front panel) control

1 Software control

2 Automatic sizing dependent on the Input power.

Note. The scanning circle diameter and mode can be obtained using the GetCircleDiameter method.

Function SetCirclePosition(bstrName As String, fHorizontalPosition As Double, fVerticalPosition As Double) As Long

Params

bstrName: name of specific NanoTrak module

fHorizontalPosition: circle horizontal position

fVerticalPosition: circle vertical position

Returns

MG return code (see Chapter 11)

Purpose

Sets the circle position for the specified NanoTrak.

Details

This method sets the horizontal and vertical circle position contained in the *pfHorizontalPosition* and *pfVerticalPosition* double parameters respectively for the NanoTrak module specified by the *bstrName* string parameter.

Both position outputs can have values in the range 0.0 – 10.0 (NanoTrak units), with (5.0,5.0) representing the display's origin (screen center).

Function SetFrequency(bstrName As String, pfFrequency As Double) As Long

Params

bstrName: name of specific NanoTrak module

pfFrequency: circle frequency

Returns

MG return code (see Chapter 11)

Purpose

Sets the circle scanning frequency for the specified NanoTrak.

Details

This method sets the circle scanning frequency contained in the *pfFrequency* double parameter for the NanoTrak module specified by the *bstrName* string parameter.

The circle scanning frequency lies in the range 0 to 512Hz. The factory default setting for the scanning frequency is 42Hz.

Function SetGain(bstrName As String, bAutomatic As Long, fGain As Double) As Long

Params

bstrName: name of specific NanoTrak module

bAutomatic: gain mode

fGain: gain setting (fixed gain mode only)

Returns

MG return code (see Chapter 11)

Purpose

Sets the NanoTrak module gain mode and gain value for the specified NanoTrak.

Details

This method sets the feedback loop gain and gain mode settings contained in the *fGain* double and *bAutomatic* long parameters respectively for the NanoTrak module specified by the *bstrName* string parameter.

The NanoTrak module exposes a gain setting that is used to ensure the DC level of the input (feedback loop) signal lies within the dynamic range of the input. The NanoTrak module can be set to auto gain control (AGC) such that the gain is adjusted automatically during operation. The bAutomatic parameter is set to TRUE to enable auto gain control mode or FALSE to enable fixed gain mode. The fGain gain setting lies in the range 0.1 to 200.0 and can only be set if the pAutomatic parameter is set to FALSE. The gain mode (fixed or auto) and gain value (fixed gain mode only) can be obtained from the specified NanoTrak module using the GetGain method.

Function SetMode(bstrName As String, bTrack As Long, bSingleAxis As Long) As Long

Params

bstrName: name of specific NanoTrak module

bTrack: tracking-latched status

bSingleAxis: circle axis mode

Returns

MG return code (see Chapter 11)

Purpose

Sets the current tracking operating mode for the specified NanoTrak.

Details

This method sets tracking mode settings for the NanoTrak module specified by the *bstrName* string parameter. The *bTrack* boolean parameter is set to TRUE to unlatch the NanoTrak module and FALSE the latch it. The *bSingleAxis* parameter Is used to set single (TRUE) or dual axis (FALSE) tracking mode. In single axis mode the circular scan becomes a horizontal line scan. Note that the single axis mode (bSingleAxis parameter) can only be changed if the bTrack parameter is set to TRUE (i.e. NanoTrak unlatched) when the SetMode method call is made. In summary, the NanoTrak module can be set to operate in three modes:-

LatchedbTrack is FALSE, bSingleAxis is FALSE

Track, dual axisbTrack is TRUE, bSingleAxis is FALSE

Track, single axisbTrack is TRUE, bSingleAxis is TRUE

The latched state and single axis mode can be obtained using the GetMode method.

Function SetPhaseOffset(bstrName As String, fPhaseOffset As Double) As Long

Params

bstrName: name of specific NanoTrak module

fPhaseOffset: phase offset setting

Returns

MG return code (see Chapter 11)

Purpose

Sets the phase offset (phase compensation) value for the specified NanoTrak.

Details

The feedback loop scenario in a typical NanoTrak application will involve the operation of various electronic and electromechanical components (e.g. power meters and piezo actuators) that can introduce phase shifts around the loop. The phase compensation setting is a means of cancelling unwanted phase shifts in the system and therefore improving tracking stability.

This method sets the phase offset value contained in the pfPhaseOffset double parameter for the NanoTrak module specified by the bstrName string parameter. The phase offset value is specified in degrees and lies in the range -90.0 to 90.0 degrees.

The phase offset can be obtained using the GetPhaseOffset method.

Function SetRange(bstrName As String, bFrontPanelDisabled As Long, lRange As Long) As Long

Params

bstrName: name of specific NanoTrak module

bFrontPanelDisabled: range button disabled status

lRange: range setting

Returns

MG return code (see Chapter 11)

Purpose

Sets the internal power meter front panel controls state, range setting and mode (auto or manual) for the specified NanoTrak.

Details

The NanoTrak unit is equipped with an internal power meter and associated front panel range/power level displays and control buttons. This power meter operates when an external detector head is connected to the PIN connector on the front panel. There are 8 range settings (1 - 8) that can be used to select the best range to measure the relative input power (displayed on the front panel relative input power LED bar display). The range can be set using front panel 'up' and 'down' buttons. Ranges 1-8 refer to the power range settings given in the ‘Specifications’ section of the handbook for the NanoTrak Control Module. Note that an auto-ranging mode can be enabled (with the front panel 'auto' button) so that range changes occur whenever the relative input power signal reaches the upper or lower end of the currently set range.

This method sets the range value and range mode of the internal power meter on the NanoTrak module specified by the *bstrName* string parameter. The *pbFrontPanelDisabled* boolean parameter is used to enable or disable the front panel control buttons. The control buttons are disabled (locked) if pbFrontPanelDisabled is set to TRUE and enabled if set to FALSE. The *plRange* parameter contains 0 to set auto-ranging mode or 1 to 8 to set the required fixed range.

Note. Auto ranging does not function when the piezo signals are routed via the backplane.

The ranging mode (auto or fixed) and range setting can be obtained using the GetRange method.

Piezos Object

The Piezos object provides the functionality required for a client application to control one or more Piezo modules fitted to a system. After setting up a configuration using the Config object, use the methods of the Piezos object to carry out activities such as switching between open and close loop mode, reading or setting position and output voltage and configuring how the input signals are summed to generate the output. To supplement the information contained in this Piezos object section please refer to the Piezo module handbook (number HA0087) for further operating instructions.

All method calls on the Piezos object require a string parameter to be passed containing a valid name (assigned using the MG17_Config.exe utility) of a Piezo module selected into the current configuration. Valid string names of all Piezos selected into the current configuration can be obtained by calling the GetPiezoNames or GetPiezoNamesEx methods of the Config object.

Function GetCurrentMode(bstrName As String, pbClosedLoop As Long) As Long

Params

bstrName: name of specific Piezo module

pbClosedLoop: returned closed loop status

Returns

MG return code (see Chapter 11)

Purpose

Returns the current operating mode for the specified Piezo module (closed-loop or open-loop).

Details

This method returns the current operating mode setting in the *pbClosedLoop* boolean parameter from the Piezo module specified by the *bstrName* string parameter. The *pbClosedLoop* parameter will be set to TRUE if the module is operating in closed loop mode or FALSE if operating in open loop mode.

Function GetCurrentPosition(bstrName As String, pfPosition As Double) As Long

Params

bstrName: name of specific Piezo module

*pfPositi*on: returned position

Returns

MG return code (see Chapter 11)

Purpose

Returns the current position, in microns, of the actuator driven by the specified Piezo module.

Details

This method returns the current position value in the *pfPosition* double parameter from the Piezo module specified by the *bstrName* string parameter. The position of the actuator is relative to the datum set for the arrangement using the method ZeroSensor. Once set, the extension of the actuator will be read and scaled automatically to a position in microns, for

whichever actuator is connected to the control module (actuators fitted with feedback circuitry can be interrogated by the Piezo module for the maximum travel in microns).

Function GetDisplay(bstrName As String, pbMicrons As Long) As Long

Params

bstrName: name of specific Piezo module

pbMicrons: returned display mode

Returns

MG return code (see Chapter 11)

Purpose

Returns the current display mode setting (voltage or position) for the specified Piezo module.

Details

This method returns the current display mode setting in the *pbMicrons* boolean parameter from the Piezo module specified by the *bstrName* string parameter. The display shows either the voltage applied to the actuator, or the position in microns as read by the position sensor. A *pbMicrons* value of FALSE indicates the display is showing volts, TRUE indicates the display is showing microns.

Function GetInputs(bstrName As String, pbPotEnabled As Long, pbAnalogueSourceEnabled As Long) As Long

Params

bstrName: name of specific Piezo module

pbPotEnabled: returned potentiometer enabled status

pbAnalogueSourceEnabled: returned analogue (BNC) source enabled status

Returns

MG return code (see Chapter 11)

Purpose

Returns the current analogue input enabled settings for the specified Piezo module.

Details

This method returns the potentiometer and front panel (BNC) analogue input enabled settings in the *pbPotEnabled* and *pbAnalogueSourceEnabled* boolean parameters respectively from the Piezo module specified by the *bstrName* string parameter.

A Piezo drive module can respond to four sources of input:

- Analogue source (front panel BNC)
- Potentiometer (front panel)
- Backplane Analogue Signal
- Software control.

The input sources that are currently enabled sum together to form the total input to the module. The front panel potentiometer input is enabled if *pbPotEnabled* returns as TRUE and disabled if FALSE. Similarly the front panel analogue (BNC) input is enabled if *pbAnalogueSourceEnabled* returns as TRUE and disabled if FALSE. Signal input from the backplane is typically configured for the specified Piezo module by using the MG17_Config.exe utility although it may also be set using the SetInputsEx method. Software control of the piezo output voltage is always enabled.

Function GetInputsEx(bstrName As String, pbPotEnabled As Long, pbAnalogueSourceEnabled As Long, plBackPlaneSignal As Long) As Long

Params
bstrName: name of specific Piezo module

pbPotEnabled: returned potentiometer enabled status

pbAnalogueSourceEnabled: returned analogue (BNC) source enabled status

plBackPlaneSignal: returned back plane analogue source enabled setting

Returns
MG return code (see Chapter 11)

Purpose
Returns the current analogue input enabled settings for the specified Piezo module.

Details
This method returns the potentiometer and front panel analogue (BNC) input enabled settings in the *pbPotEnabled* and *pbAnalogueSourceEnabled* boolean parameters respectively from the Piezo module specified by the *bstrName* string parameter. As an enhancement to the GetInputs method, this method also returns the enabled status of the back plane analogue inputs In the *plBackPlaneSignal* long parameter.

A Piezo drive module can respond to four sources of input:

- Analogue source (front panel BNC)
- Potentiometer (front panel)
- Backplane Analogue Signal
- Software control.

The Input sources that are currently enabled sum together to form the total input to the module. The front panel potentiometer input is enabled if *pbPotEnabled* returns as TRUE and disabled if FALSE. Similarly the front panel analogue (BNC) input is enabled if *pbAnalogueSourceEnabled* returns as TRUE and disabled if FALSE. Signal input from the backplane is typically configured for the specified Piezo module by using the MG17_Config.exe utility although it may also be set using the SetInputsEx method. If no backplane signal input is enabled, plBackPlaneSignal will contain 0 otherwise it will contain the index of the enabled input in the range 1 to 8. Software control is always enabled. Note that the front panel analogue (BNC) input cannot be simultaneously enabled with a backplane signal input.

Function GetTravel(bstrName As String, plStageTravel As Long) As Long

Params
bstrName: name of specific Piezo module

plStageTravel: returned stage travel

Returns
MG return code (see Chapter 11)

Purpose
Returns the maximum travel, in microns, of the actuator connected to the specified Piezo module.

Details
In the case of actuators with position sensing built in, the Piezoelectric Control module can detect the range of travel of the actuator since this information is programmed in the electronic circuit inside the actuator.

Function GetVoltage(bstrName As String, pfVoltage As Single) As Long

Params
bstrName: name of specific Piezo module

pfVoltage: returned voltage output

Returns
MG return code (see Chapter 11)

Purpose
Returns the voltage currently applied to the actuator by the specified Piezo module

Details

This method returns the output voltage (range 0.0 to 75.0) in the pfVoltage single parameter from the Piezo module specified by the *bstrName* string parameter.

Function SetCurrentMode(bstrName As String, bClosedLoop As Long) As Long

Params

bstrName: name of specific Piezo module

bClosedLoop: closed loop status

Returns

MG return code (see Chapter 11)

Purpose

Sets the current operating mode for the specified Piezo module (closed-loop or open-loop)

Details

This method sets the operating mode specified by the *pbClosedLoop* boolean parameter for the Piezo module specified by the *bstrName* string parameter. Setting *bClosedLoop* to FALSE enables open-loop mode, setting it to TRUE enables closed-loop mode. Closed-loop mode is only possible when using actuators equipped with position sensing.

Function SetCurrentPosition(bstrName As String, fPosition As Double) As Long

Note. *SetCurrentPosition* is applicable only to actuators equipped with position sensing.

Params

bstrName: name of specific Piezo module

fPosition: position value

Returns

MG return code (see Chapter 11)

Purpose

Sets the current position, in microns, of the actuator driven by the specified Piezo module

Details

This method sets the position value contained in the *fPosition* double parameter for the Piezo module specified by the *bstrName* string parameter. The position of the actuator is relative to the datum set for the arrangement using the method ZeroSensor. SetCurrentPosition is applicable only to actuators equipped with position sensing.

Function SetDisplay(bstrName As String, bMicrons As Long) As Long

Params

bstrName: name of specific Piezo module

bMicrons: display mode setting

Returns

MG return code (see Chapter 11)

Purpose

Sets the display on the specified Piezo module to read voltage or position

Details

This method sets the display mode specified by the *bMicrons* boolean parameter for the Piezo module specified by the *bstrName* string parameter. The display shows either the voltage applied to the actuator, or the position in microns as read by the position sensor. Setting *bMicrons* to FALSE sets the display to show volts and TRUE to show microns.

Function SetInputs(bstrName As String, bPotEnabled As Long, bAnalogueSourceEnabled As Long) As Long

Params

bstrName: name of specific Piezo module

bPotEnabled: potentiometer enabled status

bAnalogueSourceEnabled: analogue (BNC) source enabled status

Returns

MG return code (see Chapter 11)

Purpose

Sets the current analogue inputs for the specified Piezo module.

Details

This method sets the front panel potentiometer and analogue input enabled settings specified by the *bPotEnabled* and *bAnalogueSourceEnabled* boolean parameters respectively for the Piezo module specified by the bstrName string parameter.

A Piezo drive module can respond to four sources of input:

- Analogue source (front panel BNC)
- Potentiometer (front panel)
- Backplane Analogue Signal
- Software control.

The Input sources that are currently enabled sum together to form the total input to the module. The front panel potentiometer input is enabled if *bPotEnabled* is set to TRUE and disabled if set to FALSE. Similarly the front panel analogue (BNC) input is enabled if *bAnalogueSourceEnabled* is set to TRUE and disabled if set to FALSE. Signal input from the backplane is typically configured for the specified Piezo module by using the MG17_Config.exe utility although it may also be set using the SetInputsEx method. Software control is always enabled. Note that front panel BNC and backplane analogue sources cannot be enabled simultaneously. Thus if the SetInputs method is used to enable the BNC input, the backplane signal input is disabled. In this instance use the SetInputsEx method to re-establish the backplane signal input.

Function SetInputsEx(bstrName As String, bPotEnabled As Long, bAnalogueSourceEnabled As Long, lBackPlaneSignal As Long) As Long

Params
bstrName: name of specific Piezo module

bPotEnabled: potentiometer enabled status

bAnalogueSourceEnabled: analogue (BNC) source enabled status

lBackPlaneSignal: back plane analogue source enabled setting

Returns
MG return code (see Chapter 11)

Purpose
Sets the current analogue inputs the specified Piezo module.

Details
This method sets the potentiometer and front panel analogue (BNC) input enabled settings as specified by the *pbPotEnabled* and *pbAnalogueSourceEnabled* boolean parameters respectively for the Piezo module specified by the *bstrName* string parameter. As an enhancement to the SetInputs method, this method can also be used to enable a back plane analogue input as specified by the index value in the *lBackPlaneSignal* long parameter.

A Piezo drive module can respond to four sources of input:

- Analogue source (front panel BNC)
- Potentiometer (front panel)
- Backplane Analogue Signal
- Software control.

The Input sources that are currently enabled sum together to form the total input to the module. The front panel potentiometer control is enabled if *bPotEnabled* is set to TRUE and disabled if set to FALSE. Similarly the front panel analogue (BNC) input is enabled if *bAnalogueSourceEnabled* is set to TRUE and disabled if set to FALSE. Signal input from the backplane is typically configured for the specified Piezo module by using the MG17_Config.exe utility. However it may be set using this method by specifying the index of the backplane line in the lBackPlaneSignal parameter (range 1 to 8). To disable backplane signal input set lBackPlaneSignal to 0. Software control is always enabled. Note that the front panel analogue (BNC) input cannot be simultaneously enabled with a backplane signal input. If an attempt to enable both is made only the BNC input will be active.

Function SetVoltage(bstrName As String, fVoltage As Single) As Long

Params
bstrName: name of specific Piezo module

fVoltage: voltage output setting

Returns
MG return code (see Chapter 11)

Purpose
Sets the voltage applied to the actuator driven by the specified Piezo module.

Details
This method sets the output voltage specified by the *fVoltage* single parameter for the Piezo module specified by the *bstrName* string parameter, where *fVoltage* lies in the range 0.0 to 75.0.

Function ZeroSensor(bstrName As String) As Long

Params
bstrName: name of specific Piezo module

Returns
MG return code (see Chapter 11)

Purpose
Sets the zero reference for position measurement.

Details
This method applies a voltage of zero volts to the actuator, and then reads the position. This reading is then taken to be the zero reference for all subsequent measurements. This routine is typically called during the initialisation or re-initialisation of the Piezo arrangement.

Chapter 6

NanoSteps Object

The NanoSteps object provides the functionality required for a client application to control one or more NanoStep modules fitted to a system. After setting up a configuration using the Config object, use the methods of the NanoSteps object to carry out activities such as homing of stages, absolute and relative moves and changing velocity profile settings. To supplement the information contained in this NanoSteps object section please refer to the NanoStep module handbook (number HA0085) for further operating instructions.

Method calls on the NanoSteps object require either single string parameter or string parameter arrays to be passed containing valid names (assigned using the MG17_Config.exe utility) of NanoStep modules selected into the current configuration. Valid string names of all NanoSteps selected into the current configuration can be obtained by calling the GetNanoStepNames or GetNanoStepNamesEx methods of the Config object.

Function AtHome As Long

Params
None

Returns
MG return code (see Chapter 11)

Purpose
Waits for the ‘move to home’ of all homing NanoSteps to be completed

Details
This method waits for all NanoStep modules that are homing to complete their instructed moves. No Name parameters need be specified because the method acts on any/all motors that have received a home instruction.

The main purpose of this method is to suspend further system activity until the position of all the motors is referenced.

Function AtPosition(psabstrNames() As String, psafTravelTimes() As Double) As Long

Params
psabstrNames: names of specified NanoStep modules

psafTravelTimes: travel times

Returns
MG return code (see Chapter 11)

Purpose
Waits for the last instructed motion of the specified NanoSteps to be completed.

Details
This method waits for the NanoStep modules specified by the *psabstrNames* string array to complete last Instructed moves within the travel times specified by the mirrored *psafTravelTimes* double array.

The AtPosition method is generally used in conjunction with the 'move and continue' methods such as MoveRelativeAndContinue. These move methods include a double array parameter psafTravelTimes which is returned containing the estimated travel time for each of the NanoStep moves to complete. Typically this double array of returned travel times is used with the AtPosition method call, although an array of user defined values may be passed instead. The former is preferable since the ‘move and continue’ type methods calculate the expected travel times taking into account current velocity profile settings. If a NanoStep does not complete a move within the estimated travel time, AtPosition will return a time-out error code. Note that the estimated travel times return by the 'move and wait' type methods are generally greater than the actual time taken to give some safety margin before a move time-out error is generated

The combination of a 'move and continue' type method call and AtPosition method call allows a client application to initiate one or more NanoStep moves and then carry out some other processing before calling AtPosition to wait for the moves to complete. This can be more convenient for lengthy motor moves compared to the 'move and wait' type methods (e.g. MoveAbsoluteAndWait) that do not return until all moves are complete.

The following VisualBasic example is a typical use of the AtPosition method with MoveRelativeAndWait:-

```
' Initiates relative moves for 3 NanoStep modules named 'NS 1',
"NS 2' and 'NS 3'.
' Use the MoveRelativeAndContinue move method to allow other program
```

```
' activity during motor moves before calling AtPosition to wait for
' moves to complete.
' Object variables already assumed to be defined and referenced.

' String array to hold names of NanoStep modules.
    Dim strNSNames(1 To 3) As String
' Double array to hold relative distance to move for each NanoStep.
    Dim dDistances(1 To 3) As Double
' Double array to hold returned travel times for each NanoStep move.
    Dim dTravelTimes(1 To 3) As Double
' Variable used to hold return codes.
    Dim lRetVal As Long
' Other required variables.
    Dim i As Integer
' Initialize name array. This could be done in a call to
' GetNanoStepNames.
    strNSNames(1) = “NS 1"
    strNSNames(2) = “NS 2"
    strNSNames(3) = “NS 3"
' Initialize array of relative distances to move.
    For i = 1 To 3
' positive 1mm relative move for each NanoStep axis.
    dDistances(i) = 1#
    Next i
' Call method to initiate relative moves.
    lRetVal = objNanoSteps.MoveRelativeAndContinue(strNSNames, dDistances, dTravelTimes)
    If Not lRetVal = MG17_Drivers.MG17_OK Then
' Method failure. Do error handling here.
    End If
'*************************************************************************
' Carry out application specific activity here while motors are moving.
'*************************************************************************
' Call AtPosition to wait for moves to complete.
' Note the travel times array returned by MoveRelativeAndContinue is
' passed to AtPosition.
    lRetVal = objNanoSteps.AtPosition(strNSNames, dTravelTimes)
    If Not lRetVal = MG17_Drivers.MG17_OK Then
' Method failure. Do error handling here.
    End If
```

Function GetPosition(psabstrNames() As String, psafPositions() As Double) As Long

Params
psabstrNames: names of specified NanoStep modules

psafPositions: returned absolute positions

Returns
MG return code (see Chapter 11)

Purpose
Returns the current absolute positions of the specified Nanosteps.

Details
This method returns the current absolute positions in the *psafPositions* double array from the NanoStep modules specified by the mirrored *psabstrNames* string array. The absolute position of the associated stage/axis is determined by its displacement from the 'home' position, as set when the Home method is called.

The precision of the values contained in Positions is constrained by the specification of the associated stage axes e.g. a NanoStep 100mm linear stage has a resolution of 0.00005 mm, so this is the smallest increment you can read. Ordinarily though, double-precision values have 34 decimal place accuracy, and this must be born in mind when performing a comparison with position values obtained with this method.

Function Halt(psabstrNames() As String, bImmediate As Long) As Long

Params
psabstrNames: names of specified NanoStep modules

bImmediate: halt mode

Returns
MG return code (see Chapter 11)

Purpose
Stops the specified Nanosteps, immediately or gradually.

Details
This method brings the stage axes driven by the NanoStep modules specified by the *psabstrNames* string array to a stop gradually or immediately depending on the *bImmediate* boolean parameter setting. If *bImmediate* is set to FALSE, a gradual deceleration is used to halt the associated stage to avoid loss of steps, and therefore maintain positional integrity. If *bImmediate* is set TRUE, the stage axis stops instantaneously and steps may be lost.

Function Home(psabstrNames() As String) As Long

Params
psabstrNames: names of specified NanoStep modules

Returns
MG return code (see Chapter 11)

Purpose
Re-initializes the absolute positions of the specified NanoSteps by moving to their pre-set home (zero) positions.

Details
This method 'homes' the stage axes driven by the NanoStep modules specified by the *psabstrNames* string array.

Each NanoStep driven stage axis has default settings for the parameters Zero offset, and Minimum position. The first value is the distance between the negative limit switch and the stage's edge of travel. The second value is the minimum absolute position that can be set for the stage axis, which is a negative value or zero depending on the stage type.

The Home method sends the stage axis to its negative limit and then moves forward by a distance (Zero offset - Minimum position). The absolute position count is then reset to zero to provide the reference point for all subsequent absolute moves. If position is lost on a stage axis, the Home method should be called to re-establish the zero (home) position.

Note. The 'Home' method returns only when all required homing moves have finished. For some stage types this can take many 10's of seconds.

Function HomeAndContinue(psabstrNames() As String) As Long

Params
psabstrNames: names of specified NanoStep modules

Returns
MG return code (see Chapter 11)

Purpose
Re-initializes the absolute positions of the specified NanoSteps by moving to their pre-set home (zero) positions and returns without waiting for the moves to complete.

Details
This method 'homes' the stage axes driven by the NanoStep modules specified by the *psabstrNames* string array.

Each NanoStep driven stage axis has default settings for the parameters ‘Zero offset’, and ‘Minimum position’. The first value is the distance between the negative limit switch and the stage’s edge of travel. The second value is the minimum absolute position that can be set for the stage axis, which is a negative value or zero depending on the stage type.

The Home method sends the stage axis to its negative limit and then moves forward by a distance (Zero offset - Minimum position). The absolute position count is then reset to zero to provide the reference point for all subsequent absolute moves. If position is lost on a stage axis, the Home method should be called to re-establish the zero (home) position.

Note. This method differs from the ‘Home’ method in that it returns as soon as the home moves have been initiated, to allow the client application to perform other processing tasks (e.g. signal measurement) while the moves are still in progress. If there is no requirement to carry out other processing during a motor move then use the equivalent Home method.

Function MoveAbsoluteAndContinue(psabstrNames() As String, psafPositions() As Double, psafTravelTimes() As Double) As Long

Params
psabstrNames: names of specified NanoStep modules

psafPositions: absolute positions

psafTravelTimes: returned travel times

Returns
MG return code (see Chapter 11)

Purpose
Moves the specified NanoSteps to the given absolute positions, and returns without waiting for the moves to complete.

Details
This method moves the stage axes driven by the NanoStep modules specified by the *psabstrNames* string array to the absolute positions specified by the mirrored *psafPositions* double array. The estimated travel times for the moves are returned in the mirrored *psafTravelTimes* double array.

This method returns as soon as the absolute moves have been initiated to allow the client application to perform other processing tasks (e.g. signal measurement) while the moves are still in progress. For clean up purposes (phase current and status LED handling) this method is usually used in conjunction with the AtPosition method. See the description of the AtPosition method for more details on using the 'move and continue' type methods. If there is no requirement to carry out other processing during a motor move then use the equivalent MoveAbsoluteAndWait method. See the description of MoveAbsoluteAndWait for more details.

This method does not Implement any backlash correction. It is up to the client application to issue backlash correction moves when the AtPosition method returns.

If any of the moves requested falls outside the travel of the associated axis, it is ignored and an error message is generated.

Function MoveAbsoluteAndContinueEx(psabstrNames() As String, psafPositions() As Double, psalDirections() As Long, psafTravelTimes() As Double) As Long

Params

psabstrNames: names of specified NanoStep modules

psafPositions: absolute positions

psalDirections: direction for the move (rotational stages only)

psafTravelTimes: returned travel times

Returns

MG return code (see Chapter 11)

Purpose

Moves the specified NanoSteps to the given absolute positions, and returns without waiting for the moves to complete. If the stage is a rotational type, the move is performed in the specified direction.

Details

This method moves the stage axes driven by the NanoStep modules specified by the *psabstrNames* string array to the absolute positions specified by the mirrored *psafPositions* double array. For rotational stages, the move is performed in the direction specified by the *psalDirections* parameter:

0 = negative, 1 = positive, 2 = nearest

For non-rotational stages, the *psalDirections* parameter is ignored.

The estimated travel times for the moves are returned in the mirrored *psafTravelTimes* double array.

This method returns as soon as the absolute moves have been initiated to allow the client application to perform other processing tasks (e.g. signal measurement) while the moves are still in progress. For clean up purposes (phase current and status LED handling) this method is usually used in conjunction with the AtPosition method. See the description of the AtPosition method for more details on using the 'move and continue' type methods. If there is no requirement to carry out other processing during a motor move then use the equivalent MoveAbsoluteAndWait method. See the description of MoveAbsoluteAndWait for more details.

This method does not Implement any backlash correction. It is up to the client application to issue backlash correction moves when the AtPosition method returns.

If any of the moves requested falls outside the travel of the associated axis, it is ignored and an error message is generated.

Function MoveAbsoluteAndWait(psabstrNames() As String, psafPositions() As Double, psabBLashDisables() As Long) As Long

Params

psabstrNames: names of specified NanoStep modules

psafPositions: absolute positions

psabBLashDisables: backlash correction enabled flags

Returns

MG return code (see Chapter 11)

Purpose

Moves the specified Nanosteps to the given absolute positions, with optional backlash correction, and only returns when the moves have completed.

Details

This method moves the stage axes driven by the NanoStep modules specified by the *psabstrNames* string array to the absolute positions specified by the mirrored *psafPositions* double array. The backlash disabled settings for each of the axes is contained In the mirrored *psabBLashDisables* boolean array.

This method only returns when the absolute moves initiated have completed. If an application needs to carry out other processing activity during the stage moves then use the equivalent MoveAbsoluteAndContinue method. See the description of MoveAbsoluteAndContinue for more details.

To enable backlash correction for a particular axis set the corresponding flag in the *psabBLashDisables* array to FALSE. To disable backlash correction set the flag to TRUE. Note that for multiple moves (i.e. multiple NanoStep names in psabstrNames) a combination of TRUE and FALSE backlash disabled flag settings are allowable.

If any of the moves requested falls outside the travel of the associated axis, it is ignored and an error message is generated.

Function MoveAbsoluteAndWaitEx(psabstrNames() As String, psafPositions() As Double, psalDirections() As Long, psabBLashDisables() As Long) As Long

Params

psabstrNames: names of specified NanoStep modules

psafPositions: absolute positions

psalDirections: direction for the move (rotational stages only)

psabBLashDisables: backlash correction enabled flags

Returns

MG return code (see Chapter 11)

Purpose

Moves the specified Nanosteps to the given absolute positions, with optional backlash correction, and only returns when the moves have completed. If the stage is a rotational type, the move is performed in the specified direction.

Details

This method moves the stage axes driven by the NanoStep modules specified by the *psabstrNames* string array to the absolute positions specified by the mirrored *psafPositions* double array. For rotational stages, the move is performed in the direction specified by the *psalDirections* parameter:

0 = negative, 1 = positive, 2 = nearest

For non-rotational stages, the *psalDirections* parameter is ignored.

The backlash disabled settings for each of the axes is contained In the mirrored *psabBLashDisables* boolean array.

This method only returns when the absolute moves initiated have completed. If an application needs to carry out other processing activity during the stage moves then use the equivalent MoveAbsoluteAndContinue method. See the description of MoveAbsoluteAndContinue for more details.

To enable backlash correction for a particular axis set the corresponding flag in the *psabBLashDisables* array to FALSE. To disable backlash correction set the flag to TRUE. Note that for multiple moves (i.e. multiple NanoStep names in psabstrNames) a combination of TRUE and FALSE backlash disabled flag settings are allowable.

If any of the moves requested falls outside the travel of the associated axis, it is ignored and an error message is generated.

Function MoveRelativeAndContinue(psabstrNames() As String, psafDistances() As Double, psafTravelTimes() As Double) As Long

Params

psabstrNames: names of specified NanoStep modules

psafDistances: relative distances

psafTravelTimes: returned travel times

Returns

MG return code (see Chapter 11)

Purpose

Moves the specified NanoSteps by the given relative distances, and returns without waiting for the moves to complete.

Details

This method moves the stage axes driven by the NanoStep modules specified by the *psabstrNames* string array over the relative distances specified by the mirrored *psafDistances* double array. The estimated travel times for the moves are returned in the mirrored *psafTravelTimes* double array.

This method returns as soon as the relative moves have been initiated to allow the client application to perform other processing tasks (e.g. signal measurement) while the moves are still in progress. For clean up purposes (phase current and status LED handling) this method is usually used in conjunction with the AtPosition method. See the description of the AtPosition method for more details on using the 'move and continue' type methods. If there is no requirement to carry out other processing during a motor move then use the equivalent MoveRelativeAndWait method. See the description of MoveRelativeAndWait for more details.

This method does not Implement any backlash correction. It is up to the client application to issue backlash correction moves when the AtPosition method returns.

Function MoveRelativeAndWait(psabstrNames() As String, psafDistances() As Double, psabBLashDisables() As Long) As Long

Params

psabstrNames: names of specified NanoStep modules

psafDistances: relative distances

psabBLashDisables: backlash correction enabled flags

Returns

MG return code (see Chapter 11)

Purpose

Moves the specified NanoSteps by the given relative distances, with optional backlash correction, and only returns when the moves have completed.

Details

This method moves the stage axes driven by the NanoStep modules specified by the *psabstrNames* string array over the relative distances specified by the mirrored *psafDistances* double array. The backlash disabled settings for each of the axes is contained in the mirrored *psabBLashDisables* boolean array.

This method only returns when the relative moves initiated have completed. If an application needs to carry out other processing activity during the stage moves then use the equivalent MoveRelativeAndContinue method. See the description of MoveRelativeAndContinue for more details.

To enable backlash correction for a particular axis set the corresponding flag in the psabBLashDisables array to FALSE. To disable backlash correction set the flag to TRUE. Note that for multiple moves (i.e. multiple NanoStep names in psabstrNames) a combination of TRUE and FALSE backlash disabled flag settings are allowed.

Function SingleAtPosition(bstrName As String, fTravelTime As Double) As Long

Params

bstrName: name of specific NanoStep module

fTravelTime: travel time

Returns

MG return code (see Chapter 11)

Purpose

Waits for the last instructed motion of the specified single NanoStep to be completed.

Details

This method waits for the NanoStep module specified by the *bstrName* string to complete it's last instructed move within the travel time specified by the *fTravelTime* double parameter.

This method is functionally equivalent to the AtPosition method but is for convenience implemented for a single NanoStep module only. Refer to the description of AtPosition for full details.

Function SingleGetMinMaxPosition(bstrName As String, pfMinPositionand pfMaxPosition As Double) As Long

Params

bstrName: name of specific NanoStep module

fMinPosition: minimum limit position

fMaxPosition: maximum limit position

Returns

MG return code (see Chapter 11)

Purpose

Returns the position of the minimum and maximum limits (in millimeters or degrees) for the specified single NanoStep.

Details

This method returns the position of the minimum and maximum limits (in millimeters or degrees) in the *fMinPosition* and *fMaxPosition* double parameters from the NanoStep module specified by the *bstrName* string parameter.

The minimum and maximum positions of the associated stage/axis are determined by their displacement from the 'home' position.

Function SingleGetStageDetails (bstrName As String, pbstrStageName As String, pbstrUnits As String, plStepsToUnits As Long, pfBackLashDist As Double, pfZeroOffset As Double) As Long

Params
bstrName: name of specific NanoStep module

pbstrStageName: name of stage associated with specific NanoStep module

pbstrUnits: units (millimeters or degrees) associated with specific NanoStep module

plStepsToUnits: microsteps – real world units (mm or degrees) calibration factor

pfBackLashDist: the size of the overshoot (in millimeters or degrees) to overcome backlash. (applicable only to moves which include backlash correction)

pfZeroOffset: the distance (in millimeters or degrees) of the Home position from the lower (minimum) limit

Returns
MG return code (see Chapter 11)

Purpose
Returns information concerning the stage associated with the specified single NanoStep.

Details
This method returns information for the stage specified by the *pbstrStageName* parameter.

The *pbrstUnits* string parameter contains the units of measure for stage motion, either millimeters or degrees, whichever is relevant.

The *plStepsToUnits* parameter returns calibration value mapping microsteps to real world units.

The *pfBackLashDist* parameter returns the amount of overshoot set to compensate for backlash.

The *pfZeroOffset* parameter returns the distance between the Home position and the lower limit of travel – see *Handbook HA0097 Rotary Encoded NanoStep Control Module* for further details.

Function SingleGetPosition(bstrName As String, pfPosition As Double) As Long

Params
bstrName: name of specific NanoStep module

fPosition: absolute position

Returns
MG return code (see Chapter 11)

Purpose
Returns the current absolute position of the specified single NanoStep.

Details
This method returns the current absolute position in the *fPosition* double parameter from the NanoStep module specified by the *bstrName* string parameter.

This method is functionally equivalent to the GetPosition method but is for convenience implemented for a single NanoStep module only. Refer to the description of GetPosition for full details.

Function SingleGetVelocityProfile(bstrName As String, psafVelParams() As Double) As Long

Params
bstrName: name of specific NanoStep module

psafVelParams: returned initial, acceleration and final velocity settings

Returns
MG return code (see Chapter 11)

Purpose
Returns the velocity profile currently set for the specified single NanoStep.

Details
This method returns the current velocity profile settings in the psafVelParams double array from the NanoStep module specified by the bstrName string parameter.

The psafVelParams array parameter is a three element array containing the returned velocity parameters in the order:- initial velocity, acceleration and final velocity. The velocity profile parameters are returned in the appropriate units (mm or degrees) for the selected stage.

Function SingleHalt(bstrName As String, bImmediate As Long) As Long

Params
bstrName: names of specific NanoStep module

bImmediate: halt mode flag

Returns
MG return code (see Chapter 11)

Purpose
Stops the specified single Nanostep, immediately or gradually.

Details
This method brings the stage axis driven by the NanoStep module specified by the *bstrName* string parameter to a stop, gradually or immediately, depending on the *bImmediate* boolean parameter setting.

This method is functionally equivalent to the Halt method but is for convenience implemented for a single NanoStep module only. Refer to the description of Halt for full details.

Function SingleHome(bstrName As String) As Long

Params
bstrName: name of specific NanoStep module

Returns
MG return code (see Chapter 11)

Purpose
Re-initializes the absolute position of the specified single Nanostep to its pre-set home position.

Details
This method 'homes' the stage axis driven by the NanoStep module specified by the *bstrName* string parameter.

This method is functionally equivalent to the Home method but is for convenience implemented for a single NanoStep module only. Refer to the description of Home for full details.

Function SingleHomeAndContinue(bstrName As String) As Long

Params
bstrName: name of specific NanoStep module

Returns
MG return code (see Chapter 11)

Purpose
Re-initialises the absolute position of the specified single Nanostep to its pre-set home position.

Details
This method 'homes' the stage axes driven by the NanoStep module specified by the *bstrName* string parameter.

Note. This method is functionally equivalent to the HomeAndContinue method but is for convenience implemented for a single NanoStep module only. Refer to the description of HomeAndContinue for full details.

Function SingleMoveAbsoluteAndContinue(bstrName As String, fPosition As Double, pfTravelTime As Double) As Long

Params

bstrName: name of specific NanoStep module

fPosition: absolute position

pfTravelTime: returned travel time

Returns

MG return code (see Chapter 11)

Purpose

Moves the specified single Nanostep to the given absolute position, and returns without waiting for the move to complete.

Details

This method moves the stage axis driven by the NanoStep module specified by the *bstrName* string parameter to the absolute position specified by the *fPosition* double parameter. The estimated travel time for the move is returned in the *pfTravelTime* double parameter.

This method is functionally equivalent to the MoveAbsoluteAndContinue method but is for convenience implemented for a single NanoStep module only. Refer to the description of MoveAbsoluteAndContinue for full details.

Function SingleMoveAbsoluteAndWait(bstrName As String, fPosition As Double, bBLashDisabled As Long) As Long

Params

bstrName: name of specific NanoStep module

fPosition: absolute position

bBLashDisabled: backlash correction enabled flag

Returns

MG return code (see Chapter 11)

Purpose

Moves the specified single Nanostep to the given absolute position, with optional backlash correction, and only returns when the move has completed.

Details

This method moves the stage axis driven by the NanoStep module specified by the *bstrName* string parameter to the absolute position specified by the *fPosition* double parameter. The backlash disabled setting is contained in the *bBLashDisabled* boolean parameter.

This method is functionally equivalent to the MoveAbsoluteAndWait method but is for convenience implemented for a single NanoStep module only. Refer to the description of MoveAbsoluteAndWait for full details.

Function SingleMoveRelativeAndContinue(bstrName As String, fDistance As Double, pfTravelTime As Double) As Long

Params

bstrName: names of specific NanoStep module

fDistance: relative distance

psafTravelTime: returned travel time

Returns

MG return code (see Chapter 11)

Purpose

Moves the specified single Nanostep by the given relative distance, and returns without waiting for the move to complete.

Details

This method moves the stage axis driven by the NanoStep module specified by the *bstrName* string parameter over the relative distance specified by the *fDistance* double parameter. The estimated travel time for the move is returned In the *pfTravelTime* double parameter.

This method is functionally equivalent to the MoveRelativeAndContinue method but is for convenience implemented for a single NanoStep module only. Refer to the description of MoveRelativeAndContinue for full details.

Function SingleMoveRelativeAndWait(bstrName As String, fDistance As Double, bBLashDisabled As Long) As Long

Params

bstrName: name of specific NanoStep module

fDistance: relative distance

bBLashDisabled: backlash correction enabled flag

Returns

MG return code (see Chapter 11)

Purpose

Moves the specified NanoStep by the given relative distance, with optional backlash correction, and only returns when the move has completed.

Details

This method moves the stage axis driven by the NanoStep module specified by the *bstrName* string parameter over the relative distance specified by the *fDistance* double parameter. The backlash disabled setting is contained in the *bBLashDisabled* boolean parameter.

This method is functionally equivalent to the MoveRelativeAndWait method but is for convenience implemented for a single NanoStep module only. Refer to the description of MoveRelativeAndWait for full details.

Function SingleSetVelocityProfile(bstrName As String, psafVelParams() As Double) As Long

Params

bstrName: name of specific NanoStep module

psafVelParams: initial, acceleration and final velocity settings

Returns

MG return code (see Chapter 11)

Purpose

Sets the velocity profile of the specified single NanoStep.

Details

This method sets the velocity profile values contained in the *psafVelParams* double array for the NanoStep module specified by the *bstrName* string parameter.

The *psafVelParams* array parameter is a three element array containing the velocity parameters in the order:- initial velocity, acceleration and final velocity.

To configure a move at constant velocity, as opposed to a profiled move, set the acceleration and final velocity values to zero. The move then takes place at the value of the initial velocity.

By default, the NanoStep module will use the maximum values of initial velocity, final velocity and acceleration for all moves except for ‘homing’, which is done at the initial velocity setting. A stage’s default velocity settings can be viewed in the Stages menu of the application MG17_Config.exe

Encoded Nanosteps Object

The Encoded NanoSteps object includes all the functionality of the Nanosteps object, together with the extra functionality described in the following methods:

Function SingleGetEncodedFlag (bstrName As String, pbFlag As Long) As Long

Params
bstrName: name of specific Encoded NanoStep module

pbFlag: True or False value for flag

Returns
MG return code (see Chapter 11)

Purpose
Returns a status flag to ascertain whether the specified single Encoded NanoStep is an encoded or non-encoded variant.

Details
This method returns a status flag for the Encoded NanoStep module specified by the bstrName string parameter.

If pbFlag is True (1), then the specified NanoStep module is an encoded version, if the flag is false (0) then the module is a non-encoded variant.

Function SingleGetStatusFlag (bstrName As String, lIdent As Long, pbValue As Long) As Long

Params
bstrName: name of specific Encoded NanoStep module

lIdent: Index number of the requested flag (0 to 18)

pbValue: True or False value for requested flag

Returns
MG return code (see Chapter 11)

Purpose
Returns a status flag for a specified condition appertaining to the specified single Encoded NanoStep.

Details
This method returns a status flag for conditions appertaining to the Encoded NanoStep module specified by the *bstrName* string parameter.

If *pbValue* is True (1), then the specified condition is true and vice versa.

The Status Flags are called using *lIdent* parameter values as follows:

0 NSE_AT_POSITION – returns a TRUE value if the stage associated with the specified Encoded NanoStep module has completed its last instructed move within the requisite travel time.

1 NSE_MOTION_DIRECTION – returns a TRUE or FALSE value dependent upon the direction of travel of the specified Encoded NanoStep module, TRUE = Forward/Clockwise, FALSE = Reverse/Counterclockwise. **Note.** This flag is valid only if the motor is in motion.

2 Reserved for future implementation.

3 Reserved for future implementation.

4 NSE_JOGGING_MODE – returns a TRUE value if the ‘Jog’ buttons on the front panel of the specified NanoStep are pressed.

5 NSE_ACTIVE_LED_ON – returns a TRUE value if the ‘Active/Status’ LED on the front panel of the specified Encoded NanoStep is illuminated.

6 NSE_CAN_FAULT – returns a TRUE value if there has been a ‘CAN bus’ error.

7 NSE_POWER_OK – returns a TRUE value if the Encoded NanoStep module is powered correctly.

8 Reserved for future implementation.

9 NSE_USER_LIMIT_POS –returns a TRUE value if the Positive travel limit switch has been activated

10 NSE_USER_LIMIT_NEG – returns a TRUE value if the Negative travel limit switch has been activated

11 Reserved for future implementation.

12 Reserved for future implementation.

13 NSE_JOGGING_FWD – returns a TRUE value if the 'Jog Fwd' button on the front panel of the specified Encoded NanoStep is pressed.

14 NSE_JOGGING_REV – returns a TRUE value if the 'Jog Rev' button on the front panel of the specified Encoded NanoStep is pressed.

15 NSE_MOTOR_ENABLED – returns a TRUE value if the 'Drive Enabled' LED on the front panel of the specified Encoded NanoStep is illuminated.

16 NSE_MOTOR_HOMED – returns a TRUE value if the stage associated with the specified Encoded NanoStep has been 'homed'.

17 NSE_POSITION_ERROR – returns a TRUE value if a position error is detected. (FALSE = OK)

Note. If the position reported by the rotary encoder does not equal the intended position requested via software (e.g. absolute or relative move commands), then the flag returns TRUE. The position error flag can be cleared by homing the stage associated with the specified NanoStep. It is good programming practice to check the status of this flag after a move. If the motor is disabled and moved manually, this flag will be set to true, even though the position reported by 'GetPosition' or SingleGetPosition' methods should be correct once the motor is re-enabled. This provides a means by which the client application can recognise if a motor has been moved manually.

18 NSE_MOTOR_CONNECTED – returns a TRUE value if a motor is connected to the 'Drive Output Connector' on the front panel of the specified Encoded NanoStep module.

DigIOs Object

The DigIOs object provides the functionality required for a client application to control one or more Digital IO modules fitted to a system. After setting up a configuration using the Config object, use the methods of the Digital IOs object to carry out activities such as enabling/disabling outputs and reading the status of inputs.

All method calls on the DigIOs object require a string parameter to be passed containing a valid name (assigned using the MG17_Config.exe utility) of a Digital IO module selected into the current configuration. Valid string names of all Digital IO's selected into the current configuration can be obtained by calling the GetDigitalIONames or GetDigitalIONamesEx methods of the Config object.

Function ReadAllInputs(bstrName As String, plStatus As Long) As Long

Params

bstrName: name of the specific Digital IO module

plStatus: input channel status

Returns

MG return code (see Chapter 11)

Purpose

Reads the states of all the input channels on the specified Digital IO module

Details

This method returns the states of all input channels in the *lpStatus* long parameter from the Digital IO module specified by the *bstrName* string parameter.

The input states are contained in the lowest 16 bits of the 32bit lpStatus parameter. Bit 0 (lsb) refers to input channel 0, bit 1 refers to channel 1 and so on. For each bit, a value of 1 indicates current is flowing (between the associated + and - pins) and 0 indicates no current flow.

Function ReadAllOutputs(bstrName As String, plStatus As Long) As Long

Params

bstrName: name of the specific Digital IO module

plStatus: output channel status

Returns

MG return code (see Chapter 11)

Purpose

Reads the states of all the output channels on the specified Digital IO module

Details

This method returns the states of all output channels in the *plStatus* long parameter from the Digital IO module specified by the *bstrName* string parameter.

The output states are contained in the lowest 16 bits of the 32bit lpStatus parameter. Bit 0 (lsb) refers to output channel 0, bit 1 refers to output 1 and so on. For each bit, a value of 1 indicates the output is switched on and 0 indicates the output is switched off.

Function ReadAnInput(bstrName As String, lChannel As Long, pbStatus As Long) As Long

Params
bstrName: name of the specific Digital IO module

lChannel: input channel index

pbStatus: input channel status

Returns
MG return code (see Chapter 11)

Purpose
Reads the state of the specified input channel for the specified Digital IO module

Details
This method returns the state of the specified input channel in the *pbStatus* boolean parameter from the Digital IO module specified by the *bstrName* string parameter. The index of the input channel of interest is specified by the *lChannel* long parameter in the range 0 to 15.

For the specified channel a *pbStatus* value of TRUE indicates current is flowing (between the associated + and - pins) and FALSE indicates no current flow.

Function ReadAnOutput(bstrName As String, lChannel As Long, pbStatus As Long) As Long

Params
bstrName: name of the specific Digital IO module

lChannel: output channel index

pbStatus: output channel status

Returns
MG return code (see Chapter 11)

Purpose
Reads the state of the specified output channel for the specified Digital IO module

Details
This method returns the state of the specified output channel in the *pbStatus* boolean parameter from the Digital IO module specified by the *bstrName* string parameter. The index of the output channel of interest is specified by the *lChannel* long parameter in the range 0 to 15.

For the specified channel a pbStatus value of TRUE indicates the output is switched on and a value of FALSE indicates the output is switched off.

Function SetAllOutputs(bstrName As String, lStatus As Long) As Long

Params
bstrName: name of the specific Digital IO module

lStatus: output channel status

Returns
MG return code (see Chapter 11)

Purpose
Sets the states of all the output channels on the specified Digital IO module

Details
This method sets the output channel states contained in the *lStatus* long parameter for the Digital IO module specified by the *bstrName* string parameter.

The output states are contained in the lowest 16 bits of the 32bit lStatus parameter. Bit 0 (lsb) refers to output channel 0, bit 1 refers to output 1 and so on. For each bit, a value of 1 switches the output on and 0 switches the output off.

Function SetAnOutput(bstrName As String, lChannel As Long, bStatus As Long) As Long

Params
bstrName: name of the specific Digital IO module

lChannel: output channel index

bStatus: output channel status

Returns
MG return code (see Chapter 11)

Purpose
Sets the state of the specified output channel for the specified Digital IO module

Details
This method sets the specified output channel state contained in the *bStatus* boolean for the Digital IO module specified by the *bstrName* string parameter. The index of the output channel of Interest is specified by the *lChannel* long parameter in the range 0 to 15.

For the specified channel a bStatus value of TRUE switches the output on and a value of FALSE switches the output off.

Function TurnOff24V(bstrName As String) As Long

Params
bstrName: name of the specific Digital IO module

Returns
MG return code (see Chapter 11)

Purpose
Turns off the 24V supply output on the specified Digital IO module

Details
This method turns off the 24V supply output on the Digital IO module specified by the *bstrName* string parameter.

Function TurnOn24V(bstrName As String) As Long

Params
bstrName: name of the specific Digital IO module

Returns
MG return code (see Chapter 11)

Purpose
Turns on the 24V supply output on the specified Digital IO module

Details
This method turns on the 24V supply output on the Digital IO module specified by the *bstrName* string parameter.

PowerMeter Object

The PowerMeter object provides the functionality required for a client application to control one or more Power Meter modules fitted to a system. After setting up a configuration using the Config object, use the methods of the PowerMeter object to carry out activities such as selecting low pass filters and gain, and retrieving power readings.

All method calls on the PowerMeter object require a string parameter to be passed containing a valid name (assigned using the MG17_Config.exe utility) of a Power Meter module selected into the current configuration. Valid string names of all Power Meters selected into the current configuration can be obtained by calling the GetPowerMeterNames or GetPowerMeterNamesEx methods of the Config object.

Function AbortCapture(bstrName As String) As Long

Params

bstrName: name of specific Power Meter module

Returns

MG return code (see Chapter 11)

Purpose

Aborts the capture process for the specified single Power meter.

Details

This method aborts any data sample capturing currently in progress for the Power meter specified by the *bstrName* parameter and retains the contents of the sample buffer.

Function GetAttenuationFactor(bstrName As String, pfFactor As Double) As Long

Params

bstrName: name of specific Power Meter module

pfFactor: returned attenuation factor

Returns

MG return code (see Chapter 11)

Purpose

Returns the power attenuation factor associated with the specified single Power meter.

Details

This method returns a factor representing the amount of attenuation used with the Power meter specified by the *bstrName* parameter.

The attenuation factor allows an adjustment to be made to the power readings generated by the specified power meter module. This adjustment is required when the power signal from the source is known to have been attenuated, (e.g. due to system loss or use of integrating spheres), before being measured by the power meter module.

The *pfFactor* parameter returns a value representing the percentage attenuation. (Range 0.0% to 100.0% in 0.1% steps), i.e. a setting of 50% would result in a power reading of 1nW being doubled and displayed as 2nW. The parameter value is set using the SetAttenuationFactor method.

Function GetAutoRangeLimits(bstrName As String, pfLowLimit As Double, pfHighLimit As Double) As Long

Params

bstrName: name of specific Power Meter module

pfLowLimit: lower 'Range Down' limit

pfHighLimit: upper 'Range Up' limit

Returns

MG return code (see Chapter 11)

Purpose

Returns the lower and upper auto ranging limits for the specified single Power meter.

Details

This method returns the lower and upper auto ranging limits for the Power meter specified by the *bstrName* parameter. The *pfLowLimit* parameter defines a signal level below which the power meter automatically switches to the next lower range. Similarly, the *pfHighLimit* parameter defines a level above which the power meter switches to the next higher range. The values are returned as a percentage of the full range.

See the *SetAutoRangeLimits* method for details on setting the auto ranging limits.

See *Handbook HA 0080 Optical Power Meter Module* for further details on the use of auto ranging.

Function GetCaptureData(bstrName As String, psafData() As Single) As Long

Params

bstrName: name of specific Power Meter module

psafData: returned capture data

Returns

MG return code (see Chapter 11)

Purpose

Returns the capture data associated with the specified single Power meter.

Details

This method returns the captured power reading data for the Power meter specified by the *bstrName* parameter.

The number of data samples returned is determined by the number of elements in the *psafData* array. The data is always drawn from the start of the power meter on board data buffer. If there are more elements in the array than data samples stored, then the remaining elements are filled with whatever irrelevent data may reside in the buffer.

The following VisualBasic example is a typical use of the GetCaptureData method:

```
    Dim fData()as single
' Object variables already assumed to be defined and referenced.
' Size capture array to correct size.
' m_l Points is the number of data points to retrieve
    ReDim fData (1 to m_l Points)
    lRetVal = m_objPowerMeters.GetCaptureData (txtPMNAME.Text, fData)
    If Not lRetVal = MG17_Drivers.MG17_OK Then
' Method failure. Do error handling here.
    End If
```

See the *SetCaptureParams* method for details on how to set the number of samples to capture.

Function GetCaptureParams(bstrName As String, plRate As Long, plPoints As Long) As Long

Params

bstrName: name of specific Power Meter module

plRate: returned capture rate

plPoints: returned number of samples to capture

Returns

MG return code (see Chapter 11)

Purpose

Returns the capture set up parameters for the specified single Power meter.

Details

This method returns the capture set up parameters for the Power meter specified by the *bstrName* parameter.

The capture rate (1 to 16384 samples per second) is returned in the *plRate* parameter. The number of samples to capture (1 to 16384) is returned in the *plPoints* parameter.

See the *SetCaptureParams* method for details on how to set the capture parameters.

Function GetCaptureStatus(bstrName As String, plStatus As Long, plCapturedPoints As Long) As Long

Params

bstrName: name of specific Power Meter module

plStatus: the present capture status

plCapturedPoints: number of points captured

Returns

MG return code (see Chapter 11)

Purpose

Returns the status of the current capture process.

Details

This method returns an index in the *plStatus* parameter, indicating the status of the current capture process for the Power meter specified by the *bstrName* parameter.

0 – Capture off. Sample buffer has not been cleaned.

1 – Waiting for trigger. Sample buffer cleaned.

2 – Capturing.

3 – Stopped. Capture has been aborted. Sample buffer has not been cleaned.

If a capture run is in progress when the method is called, the *plCapturedPoints* parameter returns the number of points captured so far during the capture process. If the capture run is complete, the parameter returns the number of points captured during that capture run. See the *AbortCapture*, *PrimeCapture and StartCapture* methods for further details on the sample capture process.

Function GetDisplayBrightness(bstrName As String, pl7SegBrt As Long, plAlphaBrt As Long) As Long

Params

bstrName: name of specific Power Meter module

pl7SegBrt: returned brightness setting for the 7-segment numerical display

plAlphaBrt: returned brightness setting for the alpha-numeric display

Returns

MG return code (see Chapter 11)

Purpose

Returns the brightness setting for the 7-segment and alphanumeric displays of the specified single Power meter.

Details

This method returns a value representing the brightness setting for the 7-segment and alphanumeric displays of the Power meter specified by the *bstrName* parameter.

The *pl7SegBrt* parameter returns a value 0 to 15 representing the brightness setting of the 7-segment display, where 0 is the darkest and 15 is the brightest. The *plAlphaBrt* parameter returns a similar value for the alphanumeric display. The values are set using the SetDisplayBrightness method.

Function GetFilterInBeamFlag(bstrName As String, pbFilterInBeam As Long) As Long

Params

bstrName: name of specific Power Meter module

pbFilterInBeam: returned Filter in Beam flag

Returns

MG return code (see Chapter 11)

Purpose

Identifies whether a calibrated in-beam filter is used in the beam path associated with the specified single Power meter.

Details

This method returns a flag stating if a calibrated in-beam filter is used in the beam path of the Power meter specified by the *bstrName* parameter.

On some detector heads, a second wavelength calibration look up table is included which contains responsivity data required when using a known specified in-beam filter (normally a connectorized option that can be fitted to the detector head). This calibration table is accessed by the specified power meter module to generate calibrated readings if the *pbFilterInBeam* flag is set to True (1). The flag is set using the SetFilterInBeam method.

Function GetHeadInformation(bstrName As String, pbstrModelNo as String, pbstrSerialNo as String, plHeadType As Long) As Long

Params

bstrName: name of specific Power Meter module

pbstrModelNo: model number of detector head

pbstrSerialNo: serial number of detector head

plHeadType: type of detector head

Returns

MG return code (see Chapter 11)

Purpose

Returns the model number, serial number, and type of the detector head associated with the specified single Power Meter.

Details

This method returns the model number and serial number of the detector head associated with the Power meter specified by the *bstrName.* The values are extracted from the responsivity data stored in the calibrated detector head, and are returned in the *pbstrModelNo* (e.g. 17NTA005) and *pbstrSerialNo* (e.g.CO/EO647) parameters. An index indicating the type of detector head is returned in the *plHeadType* parameter:

1 – Germanium (Ge)

2 – Silicon (Si)

3 – Indium Galium Arsenide (InGaAs)

Function GetLowPassFilter(bstrName As String, plLPFilter As Long) As Long

Params

bstrName: name of specific Power Meter module

plLPFilter: returned Low Pass Filter value

Returns

MG return code (see Chapter 11)

Purpose

Returns the frequency of the low pass filter used in the input amplifier stage of the specified single Power meter.

Details

A low pass filter can be selected to clean up any high frequency PIN diode noise and thereby improve display stability.

This method returns an index value relating to the low pass filter selected in the input stage electronics of the Power meter specified by the bstrName parameter. The index values are:

0 30Hz

1 300Hz

2 3 KHz

3 No filter selected

The index is set using the SetLowPassFilter method.

Function GetMeasurement(bstrName As String, lSmoothType As Long, pfPower As Single, plInRange As Long) As Long

Params

bstrName: name of specific Power Meter module

lSmoothType: passed smoothing type

pfPower: returned power reading

plInRange: returned 'in range' index

Returns

MG return code (see Chapter 11)

Purpose

Returns the latest power reading of the specified single Power meter.

Details

This method obtains the power reading in the *pfPower* parameter for the Power meter specified by the *bstrName* parameter. The passed *lSmoothtype* parameter is an index which specifies the smoothing option required as follows:

0 Maximum smoothing

1 Minimum smoothing

2 No smoothing

The module samples the power readings at 16KHz. If 'no smoothing' is requested, the value obtained in the pfPower parameter is the value of the latest sample. If smoothing is selected, the value obtained is averaged over several samples (minimum smoothing 32 samples (2ms) maximum smoothing 4096 samples (250ms)). The returned value also reflects the measurement type and measurement units set in the SetMeasurementParams method. **Note**. the measurement values returned are in Watts (Power or dBm display mode) or Amps (Current display mode).

The *plInRange* parameter returns an index which indicates whether the reading is in range as follows:

0 In range

1 Over range

2 Under range

When manual ranging is selected it is important to check this index in order to guarantee a valid reading (i.e. plInRange = 0). If plInRange = 1 (over range) then the power meter range should be increased and visa versa for an under range reading.

Function GetMeasurementParams(bstrName As String, plType As Long, plUnits As Long) As Long

Params

bstrName: name of specific Power Meter module

plType: measurement type

plUnits: measurement units

Returns

MG return code (see Chapter 11)

Purpose

Obtains the measurement information for the specified single Power Meter.

Details

This method obtains measurement information for the Power Meter module specified by the *bstrName* parameter.

The *plType* parameter returns an index which relates to the measurement type as follows:

0 Average – the numerical average of the sampled data

1 Peak – the average of the highest values taken over 16 equal intervals

2 Peak-to-Peak – the difference between the peak value and the minimum value over each sample period

3 DC rms – the rms value of the overall d.c. signal value. This option differs from 'Average' in that the result is a true root men squared evaluation of the input signal d.c. component.

4 AC rms – the rms value of the a.c. signal component after removal of any d.c. offset.

The *plUnits* parameter returns an index relating to the units displayed on the front panel:

0 Power units

1 dBm units

2 Current units

The index is set using the SetMeasurementParams method. The returned index also reflects the values returned in the GetMeasurement method.

Note. the measurement values returned are in Watts (Power or dBm display mode) or Amps (Current display mode).

Function GetPowerSignalOutput(bstrName As String, plChannel As Long) As Long

Params

bstrName: name of specific Power Meter module

plChannel: returned main rack backplane channel used to route the Power Meter module output signal

Returns

MG return code (see Chapter 11)

Purpose

Identifies the backplane channel used to route the power meter output voltage signal.

Details

This method returns an index value in the *plChannel* parameter which represents the main rack backplane channel used to route the analog output signal from the power meter module specified by the *bstrName* parameter.

Typically, the analog signal that represents the power reading from the detector head is routed along the backplane to a NanoTrak module. The index is interpreted as follows:

0 Routing off (i.e. via front panel BNC connector only)

1-8 Backplane channels 1 to 8

The index is set using the SetPowerSignalOutput method.

Function GetRangeParams(bstrName As String, plMode As Long,plRange As Long,pfRangeMax as Single) As Long

Params

bstrName: name of specific Power Meter module

plMode: returned current ranging mode (auto or manual)

plRange: returned current range

pfRangeMax: returned range maximum

Returns

MG return code (see Chapter 11)

Purpose

Identifies the current range mode and range for the specified single Power Meter.

Details

This method is used to obtain the current ranging mode and range setting of the power meter module specified by the *bstrName* parameter.

The Power Meter can operate in either Auto Ranging or Manual Ranging modes, and an index value is returned in the *plMode* parameter to represent the mode currently set:

0 Auto Ranging

1 Manual Ranging

The *plRange* parameter returns an index which represents the range set. If manual ranging is currently selected, the index is set using the SetRangeParams method.

1	30pA full scale	9	300nA full scale
2	100pA full scale	10	1µA full scale
3	300pA full scale	11	3µA full scale
4	1nA full scale	12	10µA full scale
5	3nA full scale	13	30µA full scale
6	10nA full scale	14	100µA full scale
7	30nA full scale	15	300µA full scale
8	100nA full scale	16	1mA full scale

If using a detector head with a low responsivity, the actual maximum range value may be different to the range set. The *pfRangeMax* parameter returns the effective 'real world' maximum reading for the selected range.

Note. On the lower ranges, Low Pass filtering may be required to obtain a reasonable signal/noise ratio from the detector head.

Function GetResponsivity(bstrName As String, pfResponsivity as Double) As Long

Params

bstrName: name of specific Power Meter module

pfResponsivity: wavelength dependent responsivity value

Returns

MG return code (see Chapter 11)

Purpose

Returns the responsivity value (in Amps per Watt) for the specified single Power Meter.

Details

This method returns the responsivity value, in the *pfResponsivity* parameter, for the Power meter specified by the *bstrName.* The wavelength-dependent responsivity value is extracted from the calibration table stored in the calibrated detector head. See the *SetWavelength* method for details on setting the wavelength.

Function GetWavelength(bstrName As String, plWavelength As Long) As Long

Params

bstrName: name of specific Power Meter module

plWavelength: returned wavelength setting

Returns

MG return code (see Chapter 11)

Purpose

Returns the current wavelength setting for the specified single Power Meter.

Details

This method is used to obtain the current wavelength setting for the power meter module specified by the *bstrName* parameter.

The *plWavelength* parameter returns the current wavelength setting in nm (max range 300nm to 2000 nm), The displayed power readings are computed from the wavelength-dependent responsivity of the calibrated detector head. This method enables the module to select the correct responsivity value from the lookup table stored within the detector head.

Function PrimeCapture(bstrName As String, lTriggerEdge as Long) As Long

Params

bstrName: name of specific Power Meter module

lTriggerEdge: type of trigger (rising or falling edge) to initiate the capture

Returns

MG return code (see Chapter 11)

Purpose

Clears the on board data buffer and sets up (primes) the specified Power meter to commence capturing samples on receipt of a hardware trigger.

Details

This method primes the Power meter specified by the *bstrName* to start capturing samples on receipt of the next hardware TTL trigger event on the front panel BNC connector. The type of trigger to wait for is set in the *lTriggerEdge* parameter:

0 – rising edge

1 – falling edge.

Function SetAttenuationFactor(bstrName As String, fFactor As Double) As Long

Params

bstrName: name of specific Power Meter module

fFactor: attenuation attributable to the in-beam filter

Returns

MG return code (see Chapter 11)

Purpose

Sets the power attenuation factor for in-beam filter used in the beam path associated with the specified single Power meter.

Details

This method sets a factor representing the amount of attenuation used with the Power meter specified by the *bstrName* parameter.

The attenuation factor allows an adjustment to be made to the power readings generated by the specified power meter module. This adjustment is required when the power signal from the source is known to have been attenuated, (e.g. due to system loss or use of integrating spheres), before being measured by the power meter module.

The *pfFactor* parameter sets a value representing the percentage attenuation. (Range 0.0% to 100.0% in 0.1% steps), i.e. a setting of 50% would result in a power reading of 1nW being doubled and displayed as 2nW. The parameter value is obtained using the GetAttenuationFactor method.

Function SetAutoRangeLimits(bstrName As String, fLowLimit As Double, fHighLimit As Double) As Long

Params

bstrName: name of specific Power Meter module

fLowLimit: lower 'Range Down' limit

fHighLimit: upper 'Range Up' limit

Returns

MG return code (see Chapter 11)

Purpose

Sets the lower and upper auto ranging limits for the specified single Power meter.

Details

This method sets the lower and upper threshold limits used by the Power meter, specified by the *bstrName* parameter, during auto ranging. The *fLowLimit* parameter defines a signal level below which the power meter automatically switches to the next lower range. Similarly, the *fHighLimit* parameter defines a level above which the power meter switches to the next higher range. The values are set as a percentage of the full range.

Factory defaults are usually adequate for most auto ranging operations and do not normally require adjustment. However, with very noisy signals, it may be necessary to increase the low limit and decrease the high limit to prevent auto range toggling.

See the *GetAutoRangeLimits* method for details on obtaining the auto ranging limits.

See *Handbook HA 0080 Optical Power Meter Module* for further details on the use of auto ranging.

Function SetCaptureParams(bstrName As String, lRate As Long, lPoints As Long) As Long

Params

bstrName: name of specific Power Meter module

lRate: capture rate

lPoints: number of samples to capture

Returns

MG return code (see Chapter 11)

Purpose

Sets the capture parameters for the specified single Power meter.

Details

This method sets the capture parameters for the Power meter specified by the *bstrName* parameter.

The capture rate (1 to 16384 samples per second in powers of 2, ie. 1, 2, 4, 8, 16 etc.) is set in the *lRate* parameter. The number of samples to capture (1 to 16384) is set in the *plPoints* parameter.

See the *GetCaptureParams* method for details on how to obtain the capture parameters.

Function SetDisplayBrightness(bstrName As String, l7SegBrt As Long, lAlphaBrt As Long) As Long

Params

bstrName: name of specific Power Meter module

l7SegBrt: brightness setting for the 7-segment numerical display

lAlphaBrt: brightness setting for the alpha-numeric display

Returns

MG return code (see Chapter 11)

Purpose

Sets the brightness setting for the 7-segment and alphanumeric displays of the specified single Power meter.

Details

This method sets a value representing the brightness of the 7-segment and alphanumeric displays of the Power meter specified by the *bstrName* parameter.

The *l7SegBrt* parameter sets a value 0 to 15 representing the brightness of the 7-segment display, where 0 is the darkest and 15 is the brightest. The *lAlphaBrt* parameter sets a similar value for the alphanumeric display. The values are obtained using the GetDisplayBrightness method.

Function SetFilterInBeamFlag(bstrName As String, bFilterInBeam As Long) As Long

Params

bstrName: name of specific Power Meter module

bFilterInBeam:Filter in Beam flag

Returns

MG return code (see Chapter 11)

Purpose

Sets whether a calibrated in-beam filter is used in the beam path associated with the specified single Power Meter.

Details

This method sets a flag stating if a calibrated in-beam filter is used in the beam path of the Power meter specified by the *bstrName* parameter.

On some detector heads, a second wavelength calibration look up table is included which contains responsivity data required when using a known specified in-beam filter (normally a connectorized option that can be fitted to the detector head). This calibration table is accessed by the specified power meter module to generate calibrated readings if the *pbFilterInBeam* flag is set to True (1). The flag is obtained using the GetFilterInBeam method.

Function SetLowPassFilter(bstrName As String, lLPFilter As Long) As Long

Params

bstrName: name of specific Power Meter module

lLPFilter: Low Pass Filter value

Returns

MG return code (see Chapter 11)

Purpose

Sets the frequency of the low pass filter used in the input amplifier stage of the specified single Power meter.

Details

A low pass filter can be selected to clean up any high frequency PIN diode noise and thereby improve display stability.

This method sets an index relating to the frequency of the low pass filter in the input stage electronics of the Power meter specified by the bstrName parameter. The index values are:

0 30Hz

1 300Hz

2 3KHz

3 No filter selected

The index is obtained using the GetLowPassFilter method.

Function SetMeasurementParams(bstrName As String, lType As Long, lUnits As Long) As Long

Note. The power meter converts current to 'power' or 'dBm' using a conversion factor stored in the detector head. If an uncalibrated head is connected AND 'Power' or 'dBm' display mode is selected, an error message is returned.

Params

bstrName: name of specific Power Meter module

lType: measurement type

lUnits: measurement units

Returns

MG return code (see Chapter 11)

Purpose

Sets the measurement information for the specified single Power Meter.

Details

This method sets measurement information for the Power Meter module specified by the *bstrName* parameter.

The *lType* parameter sets an index which relates to the measurement type:

0 Average – the numerical average of the sampled data

1 Peak – the average of the highest values taken over 16 equal intervals

2 Peak-to-Peak – the difference between the peak value and the minimum value over each sample period

3 DC rms – the rms value of the overall d.c. signal value. This option differs from 'Average' in that the result is a true root mean squared evaluation of the input signal d.c. component.

4 AC rms – the rms value of the a.c. signal component after removal of any d.c. offset.

The *lUnits* parameter sets an index relating to the units displayed on the front panel:

0 Power units

1 dBm units

2 Current units

The index is obtained using the GetMeasurementParams method. The index also reflects the values returned in the GetMeasurement method.

Note. The measurement values returned are in Watts (Power or dBm display mode) or Amps (Current display mode).

Function SetPowerSignalOutput(bstrName As String, lChannel As Long) As Long

Params

bstrName: name of specific Power Meter module

lChannel: main rack backplane channel used to route the Power Meter module output signal

Returns

MG return code (see Chapter 11)

Purpose

Selects the backplane channel used to route the power meter output voltage signal.

Details

This method sets an index value in the lChannel parameter which represents the main rack backplane channel used to route the analog output signal from the power meter module specified by the bstrName parameter.

Typically, the analog signal that represents the power reading from the detector head is routed along the backplane to a NanoTrak module. The index is interpreted as follows:

0 Routing off (i.e. via front panel BNC connector only)

1-8 Backplane channels 1 to 8

The index is obtained using the GetPowerSignalOutput method.

Function StartCapture(bstrName As String) As Long

Params

bstrName: name of specific Power Meter module

Returns

MG return code (see Chapter 11)

Purpose

Clears the on board data buffer and starts the data sample capturing process for the specified single Power meter, independently of any pending hardware trigger.

Details

This method initiates the data sample capturing process for the power meter module specified by the *bstrName* parameter. Any pending hardware trigger is ignored (i.e. capture commences immediately).

See the *GetAutoRangeLimits* method for details on obtaining the auto ranging limits.

See *Handbook HA 0080 Optical Power Meter Module* for further details on the use of auto ranging.

Function SetRangeParams(bstrName As String, lMode As Long, lRange As Long) As Long

Params

bstrName: name of specific Power Meter module

lMode: current ranging mode (auto or manual)

lRange: current range

Returns

MG return code (see Chapter 11)

Purpose

Sets the current range mode and range for the specified single Power Meter.

Details

This method is used to set the ranging mode and range setting of the power meter module specified by the *bstrName* parameter.

The Power Meter can operate in either Auto Ranging or Manual Ranging modes, and an index value is set in the *lMode* parameter to represent the mode required:

0 Auto Ranging

1 Manual Ranging

If manual ranging is selected, the *lRange* parameter sets an index which represents the range. The index is obtained using the GetRangeParams method.

1 30pA full scale

2 100pA full scale

3 300pA full scale

4 1nA full scale

5 3nA full scale

6 10nA full scale

7 30nA full scale

8 100nA full scale

9 300nA full scale

10 1µA full scale

11 3µA full scale

12 10µA full scale

13 30µA full scale

14 100µA full scale

15 300µA full scale

16 1mA full scale

Note. On the lower ranges, Low Pass filtering may be required to obtain a reasonable signal/noise ratio from the detector head.

Function SetWavelength(bstrName As String, lWavelength As Long) As Long

Params

bstrName: name of specific Power Meter module

lWavelength: current wavelength setting

Returns

MG return code (see Chapter 11)

Purpose

Sets the wavelength for the specified single Power Meter.

Details

This method is used to select the wavelength setting for the power meter module specified by the *bstrName* parameter.

The *lWavelength* parameter sets the wavelength in nm (max range 300nm to 2000 nm), The displayed power readings are computed from the wavelength-dependent responsivity of the calibrated detector head. This method enables the module to select the correct responsivity value from the lookup table stored within the detector head.

Chapter 10

ActiveX Drivers Return Codes

This chapter lists the ActiveX Drivers return codes.

10.1 General Return Codes

MG17_OK = 0

No Error.

MG17L_SERIAL_NUMBER_ALREADY_ALLOCATED = 8001

Serial Number Already Allocated [MG17_LoLevel.dll].

An attempt has been made to create a server module object that has already been created.

MG17L_SERIAL_NUMBER_UNKNOWN = 8002

Unknown Serial Number [MG17_LoLevel.dll].

An attempt has been made to create a server module object with an unrecognized serial number. This can occur if the associated hardware module has been removed or has been replaced with another module without updating the configuration file.

MG17L_LOW_LEVEL_HANDLE = 8003

Unknown Low Level Handle [MG17_LoLevel.dll].

An attempt has been made to call a low level DLL function using an invalid handle (32bit integer). This can occur if the associated hardware module has been removed or has been replaced with another module without updating the configuration file.

MG17L_LOW_LEVEL_PARAMETER = 8004

Invalid Low Level Parameter [MG17_LoLevel.dll].

An attempt has been made to call a low level DLL function passing a parameter that is invalid or out of range.

MG17L_LOW_LEVEL_CANCEL = 8005

Module Selection Cancel [MG17_LoLevel.dll].

The user hit Cancel on the Module Selection Dialog.

MG17L_CREATE_NANOTRACK_MANAGER = 8006

Failed To Create NanoTrak Manager [MG17_LoLevel.dll].

This may be indicative of COM/OLE problems resulting from an unstable Windows system. Alternatively the MG17_Server application may have hung.

MG17L_CREATE_NANOSTEP_MANAGER = 8007

Failed To Create NanoStep Manager [MG17_LoLevel.dll].

This may be indicative of COM/OLE problems resulting from an unstable Windows system. Alternatively the MG17_Server application may have hung.

MG17L_CREATE_PIEZO_MANAGER = 8008

Failed To Create Piezo Manager [MG17_LoLevel.dll].

This may be indicative of COM/OLE problems resulting from an unstable Windows system. Alternatively the MG17_Server application may have hung.

MG17L_LOW_LEVEL_TIMEOUT = 8009

Low Level Time-out [MG17_LoLevel.dll].

This happens when calling one of the low level Get Parameter DLL functions. Possible causes include a hung MG17_Server.exe or a hung or faulty module. In the case of motor moves a time out may occur if the motor time-out specified is too short for the move to complete or the motor has stalled.

MG17L_CREATE_RACK_MANAGER = 8010

Failed To Create Rack Manager [MG17_LoLevel.dll].

This may be indicative of COM/OLE problems resulting from an unstable Windows system. Alternatively the MG17_Server application may have hung.

MG17L_SERVER_OLE_BUSY = 8011

Server OLE Busy [MG17_LoLevel.dll].

This may be indicative of COM/OLE problems resulting from an unstable Windows system. Alternatively the MG17_Server application may have hung.

MG17L_CREATE_DIGITALIO_MANAGER = 8013

Failed To Create Digital I/O Manager [MG17_LoLevel.dll].

This may be indicative of COM/OLE problems resulting from an unstable Windows system. Alternatively the MG17_Server application may have hung.

MG17H_HIGH_LEVEL_NAME = 8513

Unknown High-Level Name [MG17_HiLevel.dll].

A module name has been passed to a high level DLL function that is not recognized for the currently active configuration. The name may be incorrect or missing from the current configuration.

MG17H_HIGH_LEVEL_PARAMETER = 8514

Invalid High Level Parameter [MG17_HiLevel.dll].

An attempt has been made to call a high level DLL function passing a parameter that is invalid or out of range. In the case of motor related commands this error may occur for move requests that exceed the stage travel or that exceed the range of calibration data (if used).

MG17H_ROTARY_DEVICE = 8515

Not A Rotary Device [MG17_HiLevel.dll].

An attempt has been made to call a rotary move (high level) DLL function on a motor that is associated with a linear stage.

MG17H_HIGH_LEVEL_HANDLE = 8516

Unknown High Level Handle [MG17_HiLevel.dll].

An attempt has been made to obtain the name of a module using an invalid/unknown handle (32bit integer).

MG17D_UNKNOWN_CONFIGURATION = 8769

Unknown Configuration [MG17_Driver.dll].

The configuration name passed is unknown.

MG17D_SET_POSITION_INVALID = 8770

Set Position Invalid [MG17_Driver.dll].

The output position (setpoint) cannot be set when the Piezo module is in 'open loop' mode.

MG17D_SET_VOLTAGE_INVALID = 8771

Set Voltage Invalid [MG17_Driver.dll].

The output voltage cannot be set when the Piezo module is in 'closed loop' mode.

MG17D_AT_POSITION_TIMEOUT = 8773

At Position Time Out [MG17_Driver.dll].

A time out occurred while moving to the requested position.

MG17D_NO_NANOSTEPS = 8778

No NanoStep Names [MG17_Driver.dll].

A driver method call was made that required an string array parameter containing at least one NanoStep name.

MG17D_AXES_MOVING = 8779

Motor Axis Moving [MG17_Driver.dll].

A command has been issued to a motor (axis) that is already moving.

MG17D_AXES_BAD_DATUM = 8780

Invalid Axis Datum [MG17_Driver.dll].

The current absolute position is invalid. This can occur when an absolute move method call is made before the associated axis has been homed.

MG17D_MISSING_CONFIG_INI = 8950

Missing Configuration Information [MG17_Driver.dll].

A configuration ini file can not be accessed.

MG17D_EXISTING_CONFIGURATION_ACTIVE = 8951

Configuration Already Set Up [MG17_Driver.dll].

An existing configuration is already set up. Release the existing configuration (by calling ReleaseAConfiguration) before setting up a new configuration.

MG17D_CONFIGURATION_NOT_ACTIVE = 8952

Configuration Not Set Up [MG17_Driver.dll].

A configuration must be set up (by calling SetupAConfiguration) before calling this method.

MG17D_UNKNOWN_ERROR = 8953

Unknown Driver Error [MG17_Driver.dll].

An unspecified internal error has occurred.

MG17D_MEMORY_ERROR = 8954

Memory Error [MG17_Driver.dll].

An internal memory allocation or deallocation error has occurred.

MG17D_DRIVER_LEVEL_PARAMETER = 8955

Invalid Driver Level Parameter [MG17_Driver.dll].

An attempt has been made to call a Driver method passing a parameter that is invalid or out of range.

MG17D_OLE_ERROR = 8956

OLE/COM Error [MG17_Driver.dll].

An unexpected OLE/COM error has occurred. This may be the result of an unstable Windows system. Alternatively the MG17_Server application may have hung (possibly due to CAN bus/hardware faults).

MG17D_MULTIPLE_OBJECTS = 8957

Multiple Object Error [MG17_Driver.dll]

An attempt has been made to create more than one object of a particular type. Only single instances of MG17 Driver objects are allowed.

MG17D_DRIVER_NAME = 8958

Unknown Name [MG17_Driver.dll]

A module name has been passed that is not recognized for the currently active configuration. The name may be incorrect or the associated hardware module may be missing.

MG17D_PIEZO_SIGNAL_CONFLICT = 8959

Piezo Input Signal Conflict Error [MG17_Driver.dll]

A backplane signal input and front panel BNC input cannot be enabled simultaneously.

MG17D_THREAD_ERROR = 8960

Thread Error [MG17_Driver.dll]

An internal thread error occurred. This may be the result of an unstable Windows system.

MG17D_POSITION_INVALID = 8961

Motor Position Invalid [MG17_Driver.dll]

The current position information is invalid. This may be the result of an motor that has not been homed.

MG17D_FILE_SAVE_ERROR = 8962

File Save Error [MG17_Driver.dll]

An error occurred trying to save Call Log data to the specified path and filename. Check disk drive is not corrupted or full and that the path and filename passed are valid.

MG17D_DUPLICATE_NAME = 8963

Duplicate Module Name Error [MG17_Driver.dll]

An array of string names has been passed containing one or more duplicate names.

10.2 Encoded NanoStep Return Codes

MG17L_NANOSTEPENC_UNKNOWN_ERROR = 8020

Unknown Encoded NanoStep Command Error [MG17_LoLevel.dll].

An attempt has been made to issue an unrecognised command to an encoded NanoStep module.

MG17L_NANOSTEPENC_MOTOR_NOT_CONNECTED = 8021

Encoded NanoStep Not Connected [MG17_LoLevel.dll].

An command has been issued to an encoded NanoStep module with no motor connected.

MG17L_NANOSTEPENC_MOTOR_DISABLED = 8022

Encoded NanoStep Disabled [MG17_LoLevel.dll].

An command has been issued to an encoded NanoStep module with a motor connected but disabled.

MG17L_NANOSTEPENC_ENCODER_UNITIALISED = 8023

Encoded NanoStep Position Not Initialised [MG17_LoLevel.dll].

The Encoded NanoStep absolute position has not been set (i.e. motor not homed).

MG17L_NANOSTEPENC_MOTOR_JOGGING = 8024

Encoded NanoStep Jogging [MG17_LoLevel.dll].

An attempt has been made to issue a move command while the encoded NanoStep jog buttons are being operated.

MG17L_NANOSTEPENC_MOTOR_DECELERATING = 8025

Encoded NanoStep Decelerating [MG17_LoLevel.dll].

An attempt has been made to issue a 'Move at Velocity' command while the motor which is decelerating to a stop.

10.3 Power Meter Return Codes

MG17L_POWERMETER_UNKNOWN_CMD = 8031

Unknown Power Meter Command [MG17_LoLevel.dll].

An attempt has been made to issue an unrecognised command to a Power Meter module.

MG17L_POWERMETER_AUTONULL_ACTIVE = 8032

Power Meter Auto Nulling [MG17_LoLevel.dll].

An attempt has been made to issue a command while the Power Meter is in Auto Nulling mode (i.e. front panel Null button is pressed).

MG17L_POWERMETER_USERSETUP_ACTIVE = 8033

Power Meter User Setup [MG17_LoLevel.dll].

An attempt has been made to issue a command while the Power Meter is in User Setup mode (i.e. user interaction with the Power Meter front panel).

Chapter 11

IDL Definitions

This chapter lists the Interface Definition Language (IDL) definitions of the methods exposed by the ActiveX Drivers.

11.1 Config Object

HRESULT SetupAConfiguration([in] BSTR bstrConfigName,[out, retval] long* plRetVal);

HRESULT ShowCallLogDlg([out, retval] long* plRetVal);

HRESULT ClearCallLog([out, retval] long* plRetVal);

HRESULT ShowErrorLogDlg([out, retval] long* plRetVal);

HRESULT GetLastErrorInfo([out] BSTR* pbstrFuncName, [out] long* plErrCode, [out] BSTR* pbstrErrDesc, [out] long *plInternalIdent, [out, retval] long* plRetVal);

HRESULT ClearErrorLog([out, retval] long* plRetVal);

HRESULT ReleaseAConfiguration([out, retval] long *plRetVal);

HRESULT GetNanoStepNamesEx([out] VARIANT *pvNames, [out] long *plNum, [out, retval] long* plRetVal);

HRESULT GetNanoTrakNamesEx([out] VARIANT *pvNames, [out] long *plNum, [out, retval] long *plRetVal);

HRESULT GetPiezoNamesEx([out] VARIANT *pvNames, [out] long *plNum, [out, retval] long *plRetVal);

HRESULT GetDigitalIONamesEx([out] VARIANT *pvNames, [out] long *plNum, [out, retval] long *plRetVal);

HRESULT GetConfigurationListEx([out] VARIANT *pvConfigNames, [out] long *plNumConfigs, [out, retval] long *plRetVal);

HRESULT GetNumNanoSteps([out] long *plNum, [out, retval] long *plRetVal);

HRESULT GetNumNanoTraks([out] long *plNum, [out, retval] long *plRetVal);

HRESULT GetNumPiezos([out] long *plNum, [out, retval] long *plRetVal);

HRESULT GetNumDigitalIOs([out] long *plNum, [out, retval] long *plRetVal);

HRESULT GetNanoStepNames([in, out] SAFEARRAY (BSTR) *psabstrNames, [out, retval] long *plRetVal);

HRESULT GetNanoTrakNames([in, out] SAFEARRAY (BSTR) *psabstrNames, [out, retval] long *plRetVal);

HRESULT GetPiezoNames([in, out] SAFEARRAY (BSTR) *psabstrNames, [out, retval] long *plRetVal);

HRESULT GetDigitalIONames([in, out] SAFEARRAY (BSTR) *psabstrNames, [out, retval] long *plRetVal);

HRESULT GetNumConfigs([out] long *plNum, [out, retval] long *plRetVal);

HRESULT GetConfigurationList([in, out] SAFEARRAY (BSTR) *psabstrNames, [out, retval] long *plRetVal);

HRESULT SaveErrorLog([in] BSTR bstrPathName, [out, retval] long *plRetVal);

HRESULT SaveCallLog([in] BSTR bstrPathName, [out, retval] long *plRetVal);

11.2 NanoSteps Object

HRESULT AtHome([out, retval]long *plRetVal);

HRESULT Home([in, out] SAFEARRAY (BSTR) *psabstrNames, [out, retval] long *plRetVal);

HRESULT HomeAndContinue([in, out] SAFEARRAY (BSTR) *psabstrNames, [out, retval] long *plRetVal);

HRESULT Halt([in, out] SAFEARRAY (BSTR) *psabstrNames, [in] BOOL bImmediate, [out, retval] long *plRetVal);

HRESULT SingleHome([in] BSTR bstrName, [out,retval] long *plRetVal);

HRESULT SingleHomeAndContinue([in] BSTR *bstrName, [out, retval] long *plRetVal);

HRESULT SingleHalt([in] BSTR bstrName, [in] BOOL bImmediate, [out, retval] long *plRetVal);

HRESULT MoveRelativeAndContinue([in, out] SAFEARRAY (BSTR) *psabstrNames, [in, out] SAFEARRAY (double) *psafDistances, [in, out] SAFEARRAY (double) *psafTravelTimes, [out, retval] long *plRetVal);

HRESULT SingleMoveRelativeAndContinue([in] BSTR bstrName, [in] double fDistance, [out] double *pfTravelTime, [out, retval] long *plRetVal);

HRESULT MoveRelativeAndWait([in, out] SAFEARRAY (BSTR) *psabstrNames, [in, out] SAFEARRAY (double) *psafDistances, [in, out] SAFEARRAY (BOOL) *psabBLashDisables, [out, retval] long *plRetVal);

HRESULT SingleMoveRelativeAndWait([in] BSTR bstrName, [in] double fDistance, [in] BOOL bBLashDisabled, [out, retval] long *plRetVal);

HRESULT SingleGetVelocityProfile([in] BSTR bstrName,[in, out] SAFEARRAY (double) *psafVelParams, [out, retval] long *plRetVal);

HRESULT SingleSetVelocityProfile([in] BSTR bstrName, [in, out] SAFEARRAY (double) *psafVelParams, [out, retval] long *plRetVal);

HRESULT AtPosition([in, out] SAFEARRAY (BSTR) *psabstrNames, [in, out] SAFEARRAY (double) *psafTravelTimes, [out, retval] long *plRetVal);

HRESULT SingleAtPosition([in] BSTR bstrName, [in] double fTravelTime, [out, retval] long *plRetVal);

HRESULT MoveAbsoluteAndContinue([in, out] SAFEARRAY (BSTR) *psabstrNames, [in, out] SAFEARRAY (double) *psafPositions, [in, out] SAFEARRAY (double) *psafTravelTimes, [out, retval] long *plRetVal);

HRESULT SingleMoveAbsoluteAndContinue([in] BSTR bstrName, [in] double fPosition, [out] double *pfTravelTime, [out, retval] long *plRetVal);

HRESULT MoveAbsoluteAndWait([in, out] SAFEARRAY (BSTR) *psabstrNames, [in, out] SAFEARRAY (double) *psafPositions, [in, out] SAFEARRAY (BOOL) *psabBLashDisables, [out, retval] long *plRetVal);

HRESULT SingleMoveAbsoluteAndWait([in] BSTR bstrName, [in] double fPosition, [in] BOOL bBLashDisabled, [out, retval] long *plRetVal);

HRESULT GetPosition([in, out] SAFEARRAY (BSTR) *psabstrNames, [in, out] SAFEARRAY (double) *psafPositions, [out, retval] long *plRetVal);

HRESULT SingleGetPosition([in] BSTR bstrName, [out] double *pfPosition, [out, retval] long *plRetVal);

HRESULT SingleGetMinMaxPosition([in] BSTR bstrName, [out] double *pfMinPosition, [out] double *pfMaxPosition, [out, retval] long *plRetVal);

HRESULT SingleGetStageDetails([in] BSTR bstrName, [out] BSTR *pbstrStageName, [out]BSTR *pbstrUnits, [out] long *plStepsToUnits, [out] double *pfBackLashDist, [out] double *pfZeroOffset, [out, retval] long *plRetVal);

HRESULT LLCheckNanoStepWhenAtPosition([in] BSTR bstrName, [out] BOOL *bAtPos, [out, retval] long *plRetVal);

11.3 Encoded NanoSteps Object

HRESULT AtHome([out, retval]long *plRetVal);

HRESULT Home([in, out] SAFEARRAY (BSTR) *psabstrNames, [out, retval] long *plRetVal);

HRESULT HomeAndContinue([in, out] SAFEARRAY (BSTR) *psabstrNames, [out, retval] long *plRetVal);

HRESULT Halt([in, out] SAFEARRAY (BSTR) *psabstrNames, [in] BOOL bImmediate, [out, retval] long *plRetVal);

HRESULT SingleHome([in] BSTR bstrName, [out,retval] long *plRetVal);

HRESULT SingleHomeAndContinue([in] BSTR *bstrName, [out, retval] long *plRetVal);

HRESULT SingleHalt([in] BSTR bstrName, [in] BOOL bImmediate, [out, retval] long *plRetVal);

HRESULT MoveRelativeAndContinue([in, out] SAFEARRAY (BSTR) *psabstrNames, [in, out] SAFEARRAY (double) *psafDistances, [in, out] SAFEARRAY (double) *psafTravelTimes, [out, retval] long *plRetVal);

HRESULT SingleMoveRelativeAndContinue([in] BSTR bstrName, [in] double fDistance, [out] double *pfTravelTime, [out, retval] long *plRetVal);

HRESULT MoveRelativeAndWait([in, out] SAFEARRAY (BSTR) *psabstrNames, [in, out] SAFEARRAY (double) *psafDistances, [in, out] SAFEARRAY (BOOL) *psabBLashDisables, [out, retval] long *plRetVal);

HRESULT SingleMoveRelativeAndWait([in] BSTR bstrName, [in] double fDistance, [in] BOOL bBLashDisabled, [out, retval] long *plRetVal);

HRESULT SingleGetVelocityProfile([in] BSTR bstrName,[in, out] SAFEARRAY (double) *psafVelParams, [out, retval] long *plRetVal);

HRESULT SingleSetVelocityProfile([in] BSTR bstrName, [in, out] SAFEARRAY (double) *psafVelParams, [out, retval] long *plRetVal);

HRESULT AtPosition([in, out] SAFEARRAY (BSTR) *psabstrNames, [in, out] SAFEARRAY (double) *psafTravelTimes, [out, retval] long *plRetVal);

HRESULT SingleAtPosition([in] BSTR bstrName, [in] double fTravelTime, [out, retval] long *plRetVal);

HRESULT MoveAbsoluteAndContinue([in, out] SAFEARRAY (BSTR) *psabstrNames, [in, out] SAFEARRAY (double) *psafPositions, [in, out] SAFEARRAY (double) *psafTravelTimes, [out, retval] long *plRetVal);

HRESULT SingleMoveAbsoluteAndContinue([in] BSTR bstrName, [in] double fPosition, [out] double *pfTravelTime, [out, retval] long *plRetVal);

HRESULT MoveAbsoluteAndWait([in, out] SAFEARRAY (BSTR) *psabstrNames, [in, out] SAFEARRAY (double) *psafPositions, [in, out] SAFEARRAY (BOOL) *psabBLashDisables, [out, retval] long *plRetVal);

HRESULT SingleMoveAbsoluteAndWait([in] BSTR bstrName, [in] double fPosition, [in] BOOL bBLashDisabled, [out, retval] long *plRetVal);

HRESULT GetPosition([in, out] SAFEARRAY (BSTR) *psabstrNames, [in, out] SAFEARRAY (double) *psafPositions, [out, retval] long *plRetVal);

HRESULT SingleGetPosition([in] BSTR bstrName, [out] double *pfPosition, [out, retval] long *plRetVal);

HRESULT SingleGetMinMaxPosition([in] BSTR bstrName, [out] double *pfMinPosition, [out] double *pfMaxPosition, [out, retval] long *plRetVal);

HRESULT SingleGetStageDetails([in] BSTR bstrName, [out] BSTR *pbstrStageName, [out]BSTR *pbstrUnits, [out] long *plStepsToUnits, [out] double *pfBackLashDist, [out] double *pfZeroOffset, [out, retval] long *plRetVal);

HRESULT SingleGetStatusFlag([in] BSTR bstrName, [in] long lident, [out] BOOL *pbValue, [out, retval] long *plRetVal);

HRESULT LLCheckNanoStepEncWhenAtPosition([in] BSTR bstrName, [out] BOOL *bAtPos, [out, retval] long *plRetVal);

11.4 NanoTraks Object

HRESULT GetAbsolutePower([in] BSTR bstrName, [out] double *pfAbsolutePower, [out, retval] long *plRetVal);

HRESULT GetCircleDiameter([in] BSTR bstrName, [out] double *pfDiameter, [out] long *plMode, [out, retval] long *plRetVal);

HRESULT GetCirclePosition([in] BSTR bstrName, [out] double *pfHorizontalPosition, [out] double *pfVerticalPosition, [out, retval] long *plRetVal);

HRESULT GetFrequency([in] BSTR bstrName, [out] double *pfFrequency, [out, retval] long *plRetVal);

HRESULT GetGain([in] BSTR bstrName, [out] BOOL *pbAutomatic, [out] double *pfGain, [out, retval] long *plRetVal);

HRESULT GetMode([in] BSTR bstrName, [out] BOOL *pbTrack, [out] BOOL *pbTracking, [out] BOOL *pbSingleAxis, [out, retval] long *plRetVal);

HRESULT GetPhaseOffset([in] BSTR bstrName, [out] double *pfPhaseOffset, [out, retval] long *plRetVal);

HRESULT GetRange([in] BSTR bstrName, [out] BOOL *pbFrontPanelDisabled, [out] long *plRangeValue, [out] BOOL *pbAutoRange, [out, retval] long *plRetVal);

HRESULT GetRelativePower([in] BSTR bstrName, [out] double *pfRelativePower, [out, retval] long *plRetVal);

HRESULT IsItTracking([in] BSTR bstrName, [out] BOOL *pbTracking, [out, retval] long *plRetVal);

HRESULT SetCircleDiameter([in] BSTR bstrName, [in] double fDiameter, [in] long lMode, [out, retval] long *plRetVal);

HRESULT SetCirclePosition([in] BSTR bstrName, [in] double fHorizontalPosition, [in] double fVerticalPosition, [out, retval] long *plRetVal);

HRESULT SetFrequency([in] BSTR bstrName, [in] double fFrequency, [out, retval] long *plRetVal);

HRESULT SetGain([in] BSTR bstrName, [in] BOOL bAutomatic, [in] double fGain, [out, retval] long *plRetVal);

HRESULT SetMode([in] BSTR bstrName, [in] BOOL bTrack, [in] BOOL bSingleAxis, [out, retval] long *plRetVal);

HRESULT SetPhaseOffset([in] BSTR bstrName, [in] double fPhaseOffset, [out, retval] long *plRetVal);

HRESULT SetRange([in] BSTR bstrName, [in] BOOL bFrontPanelDisabled, [in] long lRange, [out, retval] long *plRetVal);

11.5 Piezos Object

HRESULT GetCurrentMode([in] BSTR bstrName, [out] BOOL *pbClosedLoop, [out, retval] long *plRetVal);

HRESULT GetCurrentPosition([in] BSTR bstrName, [out] double *pfPosition, [out, retval] long *plRetVal);

HRESULT GetDisplay([in] BSTR bstrName, [out] BOOL *pbMicrons, [out, retval] long *plRetVal);

HRESULT GetInputs([in] BSTR bstrName, [out] BOOL *pbPotEnabled, [out] BOOL *pbAnalogueSourceEnabled, [out, retval] long *plRetVal);

HRESULT GetTravel([in] BSTR bstrName, [out] long *plStageTravel, [out, retval] long *plRetVal);

HRESULT GetVoltage([in] BSTR bstrName, [out] float *pfVoltage, [out, retval] long *plRetVal);

HRESULT SetCurrentMode([in] BSTR bstrName, [in] BOOL bClosedLoop, [out, retval] long *plRetVal);

HRESULT SetCurrentPosition([in] BSTR bstrName, [in] double fPosition, [out, retval] long *plRetVal);

HRESULT SetDisplay([in] BSTR bstrName, [in] BOOL bMicrons, [out, retval] long *plRetVal);

HRESULT SetInputs([in] BSTR bstrName, [in] BOOL bPotEnabled, [in] BOOL bAnalogueSourceEnabled, [out, retval] long *plRetVal);

HRESULT SetVoltage([in] BSTR bstrName, [in] float fVoltage, [out, retval] long *plRetVal);

HRESULT ZeroSensor([in] BSTR bstrName, [out, retval] long *plRetVal);

HRESULT SetInputsEx([in] BSTR bstrName, [in] BOOL bPotEnabled, [in] BOOL bAnalogueSourceEnabled, [in] long lBackPlaneSignal, [out, retval] long *plRetVal);

HRESULT GetInputsEx([in] BSTR bstrName, [out] BOOL *pbPotEnabled, [out] BOOL *pbAnalogueSourceEnabled, [out] long *plBackPLaneSignal, [out, retval] long *plRetVal);

11.6 DigIOs Object

HRESULT ReadAllInputs([in] BSTR bstrName, [out] long *plStatus, [out, retval] long *plRetVal);

HRESULT ReadAllOutputs([in] BSTR bstrName, [out] long *plStatus, [out, retval] long *plRetVal);

HRESULT ReadAnInput([in] BSTR bstrName, [in] long lChannel, [out] BOOL *pbStatus, [out, retval] long *plRetVal);

HRESULT ReadAnOutput([in] BSTR bstrName, [in] long lChannel, [out] BOOL *pbStatus, [out, retval] long *plRetVal);

HRESULT SetAllOutputs([in] BSTR bstrName, [in] long lStatus, [out, retval] long *plRetVal);

HRESULT SetAnOutput([in] BSTR bstrName, [in] long lChannel, [in] BOOL bStatus, [out, retval] long *plRetVal);

HRESULT TurnOff24V([in] BSTR bstrName, [out, retval] long *plRetVal);

HRESULT TurnOn24V([in] BSTR bstrName, [out, retval] long *plRetVal);

11.7 PowerMeter Object

HRESULT SetLowPassFilter([in] BSTR bstrName, [in] long lLPFilter, [out, retval] long *plRetVal);

HRESULT GetLowPassFilter([in] BSTR bstrName, [out] long *plLPFilter, [out, retval] long *plRetVal);

HRESULT SetMeasurementParams([in] BSTR bstrName, [in] long lType, [in] long lUnits, [out, retval] long *plRetVal);

HRESULT GetMeasurementParams([in] BSTR bstrName, [out] long *plType, [out] long *plUnits, [out, retval] long *plRetVal);

HRESULT SetWavelength([in] BSTR bstrName, [in] long lWavelength, [out, retval] long *plRetVal);

HRESULT GetWavelength([in] BSTR bstrName, [out] long *plWavelength, [out, retval] long *plRetVal);

HRESULT SetRangeParams([in] BSTR bstrName, [in] long lMode, [in] long lRange, [out, retval] long *plRetVal);

HRESULT GetRangeParams([in] BSTR bstrName, [out] long *plMode, [out] long *plRange, [out, retval] long *plRetVal);

HRESULT SetDisplayBrightness([in] BSTR bstrName, [in] long l7SegBrt, [in] long lAlphaBrt, [out, retval] long *plRetVal);

HRESULT GetDisplayBrightness([in] BSTR bstrName, [out] long *pl7SegBrt, [out] long *plAlphaBrt, [out, retval] long *plRetVal);

HRESULT GetMeasurement([in] BSTR bstrName, [in] long lSmoothType, [out] float *pfPower, [out] long *plInRange, [out, retval] long *plRetVal);

HRESULT SetPowerSignalOutput([in] BSTR bstrName, [in] long lChannel, [out, retval] long *plRetVal);

HRESULT GetPowerSignalOutput([in] BSTR bstrName, [out] long *plChannel, [out, retval] long *plRetVal);

HRESULT SetFilterInBeamFlag([in] BSTR bstrName, [in] BOOL bFilterInBeam, [out, retval] long *plRetVal);

HRESULT GetFilterInBeamFlag([in] BSTR bstrName, [out] BOOL *pbFilterInBeam, [out, retval] long *plRetVal);

HRESULT SetAttenuationFactor([in] BSTR bstrName, [in] double fFactor, [out, retval] long *plRetVal);

HRESULT GetAttenuationFactor([in] BSTR bstrName, [out] double *pfFactor, [out, retval] long *plRetVal);

Products and Customer Support

A Comprehensive Product Range

Optical Components,

Singlets, Doublets and Triplets; Cylindrical Optics, Mirrors, Prisms and Retroreflectors, Beamsplitters, Polarization Components, Filters, High Energy Laser Optics, Diode Laser Optics, UV Optics, Machine Vision.

Opto-mechanical Hardware

'MicroLab System, Micro-optics, Lens, Filter and Polarizer Mounts, Mirror/Beamsplitter Mounts and Prism Tables.

Nanopositioning

Stages, Mechanical Accessories, Piezo-electric and Stepper-motor Controllers, Autoalignment, Modular System Controllers.

Optical Tables, Breadboards and Vibration Isolators

Optical Table-tops, Vibration-isolation and support systems, Optical Breadboards and Baseplates, Workstations.

Lasers

Diode-pumped Solid State, Ion, Helium Cadmium, Helium Neon, Diode Laser Assemblies, Laboratory Diode Laser Drivers, Accessories.

Laser Measurement Instrumentation

Laser-beam characterization, Photodiodes, Power and Energy Meters.

Lab Accessories

Technical Support

Melles Griot provides a comprehensive after sales service. Contact us through your local representative, or at the address below:

Melles Griot Ltd

Saint Thomas Place

Ely

Cambridgeshire CB7 4EX, UK

Tel: +44 (0) 1353 654500

Fax: +44 (0) 1353 654555

email: nanosupport@melesgriot.com

Client Warranty

Prior to installation, the equipment referred to in this handbook must be stored in a clean, dry environment, in accordance with any instructions given. Periodic checks must be made on the equipment's condition.

It is always helpful to have detailed and accurate information about any problems encountered by customers

We welcome comments or suggestions about any aspect of the equipment and instruction handbooks.

Printed in UK (0502)

Melles Griot Ltd.
Saint Thomas Place
Ely
Cambridgeshire CB7 4EX, UK
Tel: +44 (0) 1353 654500
Fax: +44 (0) 1353 654555
www.mellesgriot.com

HA 0104 Issue 5