



SoftMax® Pro

Data Acquisition and Analysis Software Automation

Version 7.0

API Reference Guide



This document is provided to customers who have purchased Molecular Devices equipment, software, reagents, and consumables to use in the operation of such Molecular Devices equipment, software, reagents, and consumables. This document is copyright protected and any reproduction of this document, in whole or any part, is strictly prohibited, except as Molecular Devices may authorize in writing.

Software that may be described in this document is furnished under a non-transferrable license. It is against the law to copy, modify, or distribute the software on any medium, except as specifically allowed in the license agreement. Furthermore, the license agreement may prohibit the software from being disassembled, reverse engineered, or decompiled for any purpose.

Portions of this document may make reference to other manufacturers and/or their products, which may contain parts whose names are registered as trademarks and/or function as trademarks of their respective owners. Any such usage is intended only to designate those manufacturers' products as supplied by Molecular Devices for incorporation into its equipment and does not imply any right and/or license to use or permit others to use such manufacturers' and/or their product names as trademarks.

Each product is shipped with documentation stating specifications and other technical information. Molecular Devices products are warranted to meet the stated specifications. Molecular Devices makes no other warranties or representations express or implied, including but not limited to, the fitness of this product for any particular purpose and assumes no responsibility or contingent liability, including indirect or consequential damages, for any use to which the purchaser may put the equipment described herein, or for any adverse circumstances arising therefrom. The sole obligation of Molecular Devices and the customer's sole remedy are limited to repair or replacement of the product in the event that the product fails to perform as warranted.

For research use only. Not for use in diagnostic procedures.



The trademarks mentioned herein are the property of Molecular Devices, LLC or their respective owners. These trademarks may not be used in any type of promotion or advertising without the prior written permission of Molecular Devices, LLC.

Patents: <http://www.moleculardevices.com/productpatents>

The Automation API and associated documentation are copyrighted and property of Molecular Devices, LLC. Use of the Automation API and acceptance of the associated documentation is prohibited without the express written consent of Molecular Devices, LLC, which consent shall limit the scope of use of the Automation technology to the authorized user, and shall be non-transferable.

Product manufactured by Molecular Devices, LLC.

1311 Orleans Drive, Sunnyvale, California, United States of America 94089.

Molecular Devices, LLC is ISO 9001 registered.

© 2016 Molecular Devices, LLC.

All rights reserved.

Foreword

Who This Guide is For

This guide is written for those who want to build an automation application that interacts with an instance of the SoftMax® Pro Data Acquisition and Analysis Software Version 6 using the automation interface to control the operation of a Molecular Devices® instrument. This reference guide documents version 7.0 of the SoftMax Pro Automation Interface.

This version of the automation interface can run scripts written for version 6.0 and newer.

Reader Prerequisites

To use this guide, the reader must:

- Understand the basic concept of object-oriented programming and programming using Microsoft .NET including the use of delegates and events.
- Understand assemblies in Microsoft .NET.
- Understand namespaces in Microsoft .NET.
- Have access to and knowledge of development tool to create and customize an assembly.

Changes to SoftMax Pro Software Automation

The technology used for SoftMax Pro Software Automation 6.0 and newer has changed compared to previous versions.

Technology

The underlying communication mechanism is now Windows Communication Foundation (WCF) rather than Windows messaging, which allows:

- Connections to the SoftMax Pro Software over a network, rather than requiring the automation application to be located on the same computer as the SoftMax Pro Software installation.
- A strongly-typed interface, so that the compiler can catch potential issues instead of discovering errors at runtime.
- Improved error reporting.

Enhancements

In addition, the following enhancements have been made:

- Commands are now queued, so that you no longer have to wait for a command to complete before sending the next one.
- The SoftMax Pro Software application GUI is now blocked when it is being controlled by an automation application, preventing another person from interfering while automation is in progress.
- There are more comprehensive documentation and samples, including a sample application which can be used to prototype programs.

Commands

The commands available in the API are the same, although the names of some commands have changed.

The following commands are not available in this version, and will be added in future versions:

- Copy, SelectAll, SetFolderAs, TrayLock
- CDrawer, CloseCDrawer, OpenCDrawer, OpenTDrawer, TDrawer (FlexStation instruments only)

This version of the automation interface can run scripts written for version 6.0 and newer.

Contents

Foreword	3
Who This Guide is For	3
Reader Prerequisites	3
Changes to SoftMax Pro Software Automation	3
Chapter 4: Automation Interface Overview.....	7
Introduction.....	7
Automation Interface Components.....	7
Using the Automation Interface	8
Installing the Automation API	13
Computer System Requirements.....	13
Using Automation with the SoftMax Pro Software GxP Edition	14
Sample Source Code and Applications	14
Creating a New Project in Visual Studio	14
Automation Sample Application	15
Developing SoftMax Pro Excel Visual Basic Macros.....	16
Chapter 5: Using SoftMax Pro Excel Workflows	19
Introduction.....	19
Installing the Excel Add-In	19
Using the Provided Excel Workflows.....	20
Running Excel Workflows.....	21
Creating a New Excel Workflow.....	23
Editing Excel Workflows	23
Initialization Statements.....	24
Automation Command Statements.....	25
Worksheet Formatting Statements.....	25
Workflow Flow Control Statements	38
Custom Excel Formulas.....	42
Instrument Connectivity.....	44
Troubleshooting Excel Workflows	45
Chapter 6: SoftMax Pro Automation Commands	47
AppendData.....	48
AppendTitle	49
CloseDocument	50
CloseAllDocuments	50
CloseDrawer	50
Dispose	51
ExportAs	52
ExportSectionAs	53
GetAutosaveState	53
GetDataCopy	54
GetDrawerStatus.....	55

GetFormulaResult	57
GetGroupNameAssignments	59
GetInstrumentStatus	59
GetNumberPlateSections	60
GetTemperature	61
GetVersion	61
ImportPlateData.....	63
ImportPlateTemplate	64
Initialize.....	65
Logon.....	68
Logoff	68
NewDocument.....	69
NewExperiment	70
NewNotes.....	71
NewPlate	72
OpenDrawer.....	73
OpenFile	75
Quit	76
SaveAs.....	77
SelectNextPlateSection	78
SelectSection	79
SetReader	80
SetShake	81
SetSimulationMode	82
SetTemperature	83
SetTitle	84
SetUserName	85
StartRead	85
StopRead.....	86
Chapter 7: Events	87
CommandCompleted.....	87
InstrumentStatusChanged	88
ErrorReport.....	89
Chapter 8: Examples	91
Append Title Script.....	91
Get Commands Script	92
Multiple Read and Copy Events Script	93
Multiple Read With ID Script.....	94
ExportAs Script.....	95

Automation Interface Overview

Introduction

To integrate Molecular Devices® readers and the SoftMax® Pro Software version 7 with robotic systems from other manufacturers, or to automate the export of data from the SoftMax Pro Software to a desired Laboratory Information Management System (LIMS) package, the SoftMax Pro Software supports an automation interface. Molecular Devices has tested the interface but does not provide technical support for specific integration needs. Molecular Devices strongly recommends that customers consult a third-party automation company if internal software resources or expertise are not available.

All leading automation vendors have integrated the SoftMax Pro Software and Molecular Devices instruments with their robotics systems. A list of approved robotics partners can be found on the Molecular Devices web site.

Automation Interface Components

This SoftMax Pro Automation Interface version 6 is a Microsoft .NET Framework assembly, which can be used with any .NET Language (C#, Visual Basic, etc). This reference guide documents version 7.0 of the SoftMax Pro Automation Interface.

This version of the automation interface can run scripts written for version 6.0 and newer.

The Automation Interface API components are shown in the following figure.

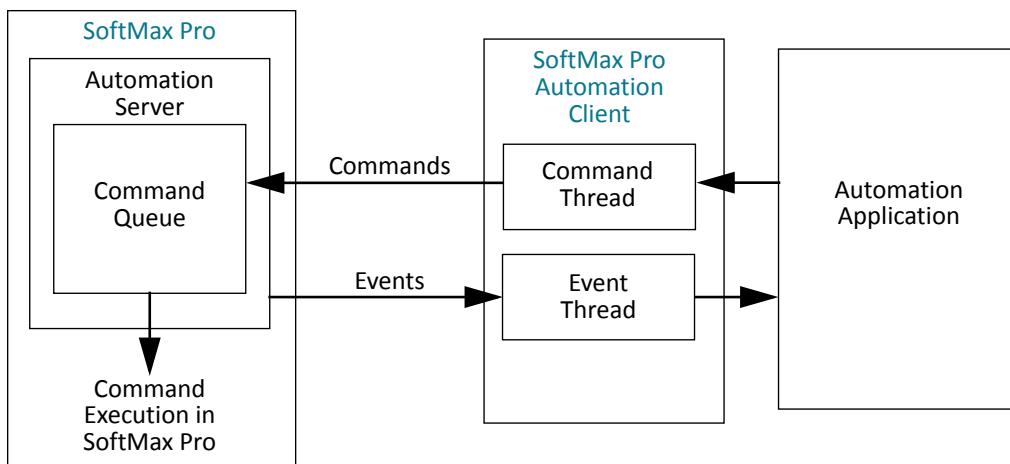


Figure 4-1: SoftMax Pro Automation API

The .NET assembly **SoftMax Pro AutomationClient** supplies the interface that can be used to control the SoftMax Pro Software. This interface contains commands that can be called, as well as Event Handlers that are used to notify the automation application of the status of the SoftMax Pro Software, execution of queued commands, and errors. Any application that wants to control the SoftMax Pro Software must connect through the AutomationClient.

The AutomationClient connects to the **AutomationServer**, which is contained within the SoftMax Pro Software. The AutomationServer maintains a queue of commands that have been sent through the Automation Client, and executes them sequentially.

Commands are sent to the SoftMax Pro Software by calling functions in the automation interface (the client). All commands in the automation interface are asynchronous, meaning that the functions will return control to the application immediately and will not block until complete.

Command results and errors are obtained by subscribing to events on the automation interface.

The Automation Server and the Automation Client are both multi-threaded:

- ◆ Commands are sent by the client and received by the server on one thread.
- ◆ Events are published by the server and passed on by the client on a second thread.

Using the Automation Interface

You use the Automation Interface to control the SoftMax Pro Software. To connect to the SoftMax Pro Software using the automation interface, call the **Initialize()** function. After the automation interface is initialized, the SoftMax Pro Software enters “automation mode.” While in this automation mode, user input by way of the SoftMax Pro Software user interface is disabled.

Automation mode can be ended from the automation interface by calling **Dispose()** or from the SoftMax Pro Software user interface by clicking the **Terminate** button.

After initialization is complete, events that the application would like to monitor must be hooked up. If events are connected, they should be disconnected when the **CommandCompleted** event indicates that the command queue is empty.

All commands in the automation interface return a command ID. The command IDs are generated by the automation interface and maintained in a command queue, as shown in [Figure 4-2](#). Each command ID should be stored, and used to obtain command results when the command completes. An application obtains the results of a command by subscribing to the **CommandCompleted** event, and matching the command ID in the event with the one that was returned when the command was sent.

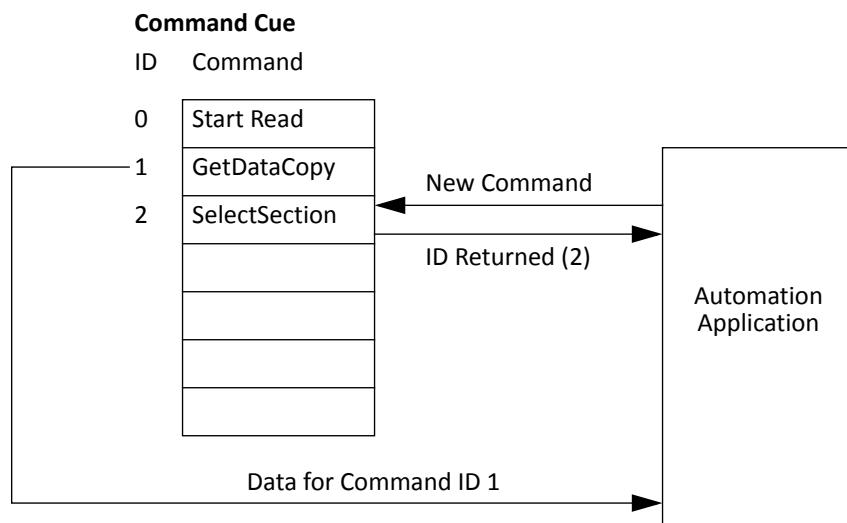


Figure 4-2: Command Queue

In addition, the following two events are published and should be considered when designing and writing an automation application:

ErrorReport event: When an error is detected by the automation server an Error event is published, followed by a CommandComplete event. Additionally, all commands in the automation server command queue are flushed.

InstrumentStatus Event: When the instrument changes state, for example from Idle to Busy, an InstrumentStatus event is published.

For more information about event messages, see [Events on page 87](#).

For help with creating an environment able to run C# and Visual Basic .NET clients, see [Creating a New Project in Visual Studio on page 14](#).

The following example shows a basic code structure. This includes a call to **Initialize()**, subscription to an event, and a **Dispose()** call to terminate the connection. The following examples assumes that the Automation Sample Application is being used.

```
// An instrument must be connected to SoftMax Pro
// before running this code

int lastCommandId = -999;
bool connected = false;

public void Main()
{
    // Initialize the interface
    connected = AutomationObject.Initialize();

    if ( connected )
    {
        // Hook up the CommandCompleted event
        AutomationObject.CommandCompleted+= CommandCompleted;
        // Open the reader drawer
        lastCommandId = AutomationObject.OpenDrawer();
    }
}

private void CommandCompleted( object sender,
    SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    // report on the command being completed
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );

    // Disconnect from the automation server
    if( lastCommandId == e.QueueID )
    {
        Results.AppendResult("Command execution complete, disconnecting from server");
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
```

The output generated should be as follows:

```
Command complete Command ID = 0Command execution complete, disconnecting from server
```

The following example is similar logic written in C#.

```
private int lastCommandId = -999;
private bool connected = false;
private SMPAutomationClient client;

public void OpenDrawerExample()
{
    // Create an instance of the automation client
    client = new SMPAutomationClient();

    // Initialize the interface
    connected = client.Initialize();

    if (connected)
    {
        // Hook up the CommandCompleted event
        client.CommandCompleted += CommandCompleted;
        // Open the reader drawer
        lastCommandId = client.OpenDrawer();
    }
}

private void CommandCompleted(object sender, SMPAutomationClient.CommandStatusEventArgs e)
{
    // report on the command being completed
    Console.WriteLine("Command complete Command ID = " + e.QueueID.ToString());

    // Disconnect from the automation server
    if (lastCommandId == e.QueueID)
    {
        Console.WriteLine("Command execution complete, disconnecting from server");
        client.CommandCompleted -= CommandCompleted;
        client.Dispose();
    }
}
```

The following example is similar logic written in Visual Basic .NET.

```

Imports System.Threading
Imports SoftMaxPro.AutomationClient

Module OpenDrawerExample
    Dim lastCommandId = -999
    Dim client As SMPAutomationClient

    Sub Main()
        REM Create an instance of the automation client
        client = New SMPAutomationClient

        REM Initialize the interface
        Dim connected = client.Initialize()

        If (connected) Then
            REM Hook up the CommandCompleted event
            AddHandler client.CommandCompleted, AddressOf CommandCompleted

            REM Open the reader drawer
            WaitForCommandCompletion(client.OpenDrawer())

            REM Disconnect from the automation server
            Console.WriteLine("Command execution complete, disconnecting from server")
            RemoveHandler client.CommandCompleted, AddressOf CommandCompleted
            client.Dispose()
        End If
    End Sub

    Private Sub CommandCompleted(ByVal sender As Object,
                                 ByVal e As SMPAutomationClient.CommandStatusEventArgs)
        Console.WriteLine("Command complete Command ID = " + e.QueueID.ToString())
        lastCommandId = e.QueueID
    End Sub

    Private Sub WaitForCommandCompletion(ByVal commandId As Integer)
        While commandId > lastCommandId
            Thread.Sleep(10)
        End While
    End Sub
End Module

```

Get() Functions

Commands that return information, that are prefixed with Get..., like **GetVersion()** or **GetNumberPlateSections()**. They are queued like all other commands and the information of the command is returned by way of the **CommandStatusEventArgs** class in the **CommandCompleted** event.

The CommandStatusEventArgs class

```
public class CommandStatusEventArgs : EventArgs
{
    public CommandStatusEventArgs( int queueID, bool queueEmpty, int commandID, int intResult, String
stringResult )
    {
        QueueID = queueID;
        QueueEmpty = queueEmpty;
        CommandID = commandID;
        IntResult = intResult;
        StringResult = stringResult;
    }

    public int QueueID;
    public bool QueueEmpty;
    public int CommandID;
    public int IntResult;
    public String StringResult;
}
```

The following example demonstrates how to use functions that return information.

```

int mStatusID;

public void Main()
{
    AutomationObject.Initialize();
    AutomationObject.CommandCompleted+= CommandCompleted;

    mStatusID = AutomationObject.GetVersion();
    Results.AppendResult("Status Command ID = " + mStatusID .ToString() );
}

private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)

{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );

    if( mStatusID== e.QueueID )
    {
        Results.AppendResult( "Result: " +e.StringResult);
    }

    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");

        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}

```

Installing the Automation API

The Automation SDK is automatically installed with the SoftMax Pro Software, and can be found in the file system where the SoftMax Pro Software application is installed.

The default installation path for the automation SDK is in a sub-folder of the **Program Files** folder where the SoftMax Pro Software is installed:

- For a 32-bit installation:
C:\Program Files\Molecular Devices\SoftMax Pro 7.0 Automation SDK
- For a 64-bit installation:
C:\Program Files (x86)\Molecular Devices\SoftMax Pro 7.0 Automation SDK

Computer System Requirements

The minimum and recommended computer system requirements are listed in the SoftMax Pro Software application help and user guide.

- When running both automation and the SoftMax Pro Software on the same computer, use at least the recommended system configuration, not the minimum requirements.
- Molecular Devices recommends that the files used in the automation workflow have a small number of sections, and that documents should be closed when they are no longer needed.
- An Experiment with many plates of data could adversely affect SoftMax Pro Software efficiency. If increasing the computer memory is not an option, Molecular Devices recommends using only one plate per Experiment and also ensuring that the computer is not being used for other purposes while SoftMax Pro Software is running.

Using Automation with the SoftMax Pro Software GxP Edition

If the target application is the SoftMax Pro GxP Software, /R is required in the program command line (for example, "SoftMaxProApp.exe /R") when launching the application to avoid the Log On dialog that requires manual entry of user ID and password. Instead, the Logon remote command can be sent to log the user on through the robotic interface. See [Logon on page 68](#).

Sample Source Code and Applications

Sample automation scripts and the command example scripts shown in this guide are included as a part of the SoftMax Pro Automation API installation package and can be found in the **Scripts** folder in the **SoftMax Pro 7.0 Automation SDK** folder.

The default installation path for the sample automation scripts is:

C:\Program Files (x86)\Molecular Devices\SoftMax Pro 7.0 Automation SDK\Scripts

You can use the sample application to load, view, and run the script examples.

Creating a New Project in Visual Studio

To use the automation interface in the development of custom applications:

- 1 Create a new project in Visual Studio using your preferred programming language.
- 2 Select one of the relevant templates (for example, Windows Form Application, WPF application, or Console application).
3. In the new project, add a reference to each of the following assemblies:
 - ◆ SoftMax Pro AutomationClient.dll
 - ◆ SoftMaxPro.AutomationInterface.dll
 - ◆ SoftMaxPro.AutomationExtensions.dll

These assemblies are contained in a sub-folder of the **Program Files** folder where the SoftMax Pro Software is installed:

- ◆ For a 32-bit installation, the assemblies are in the following folder:
C:\Program Files\Molecular Devices\SoftMax Pro 7.0 Automation SDK
- ◆ For a 64-bit installation, the assemblies are in the following folder:
C:\Program Files (x86)\Molecular Devices\SoftMax Pro 7.0 Automation SDK



Note: The provided Automation Sample Application does not show the referenced assembly because it is loaded dynamically at run time and injected into the scripting interface. Referencing this assembly is required for normal application development.

Automation Sample Application

A Sample Application has been provided which can be used to quickly prototype and test code. The application binds the automation interface to a Microsoft scripting control.



Note: The Sample Application is not intended to be used in an actual laboratory or research environment.

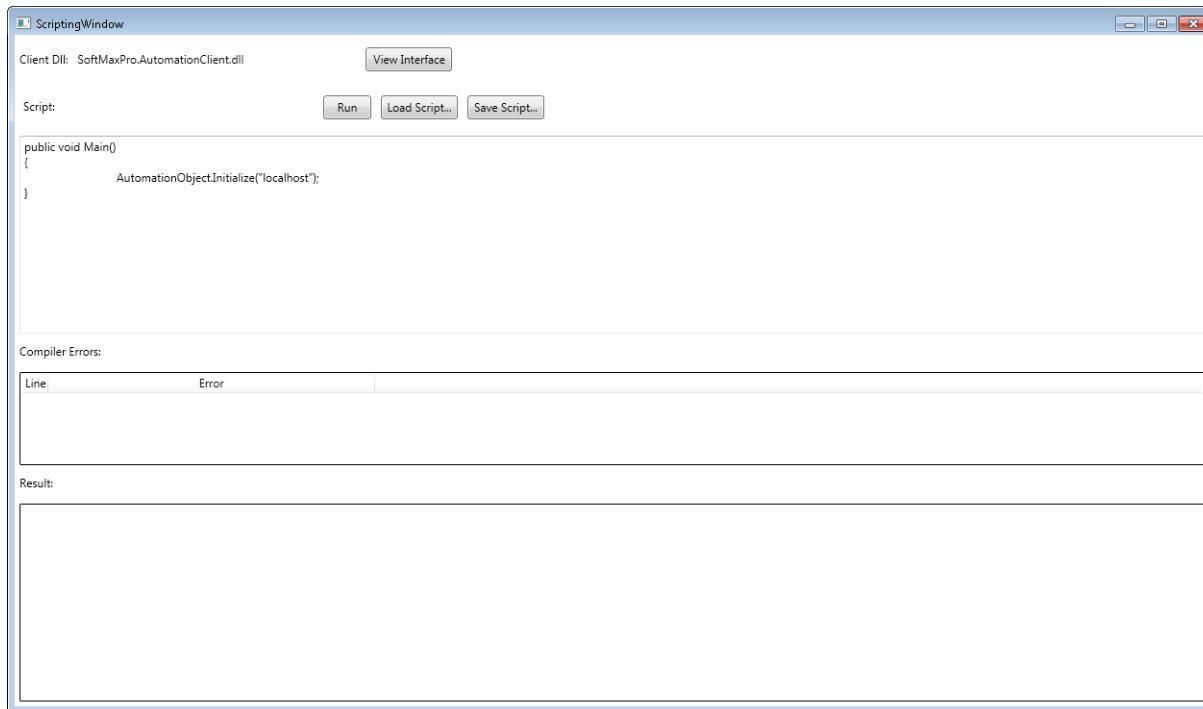


Figure 4-3: Sample Application Scripting Window

The Sample Application is launched by executing the SMP AutomationSampleApp shortcut from the Windows Start Menu. The scripting window that appears is shown in [Figure 4-3](#).

The Sample Application scripting window has buttons that let you view the available interface commands and to load, run, and save scripts. Errors are displayed in Compiler Errors and any output information that is relevant to the script are shown in Results.

To prototype sample code quickly, the scripting engine of the Sample Application has already created the object instance for the automation component, called **AutomationObject**, behind the scenes. In the script, you can use this object to access all automation commands and events to be monitored. If the code is transferred to a stand-alone application, code must be added to create the automation object instance.

The Sample Application also has a “Result” object that can be used to append text data. This can also be used to monitor the execution of commands and the progress in the command queue.

After programming code has been prototyped in the Script section of the Sample Application, it can be used to create a stand-alone robotic/automation application that directly controls SoftMax Pro Software.

Developing SoftMax Pro Excel Visual Basic Macros

This section describes how Visual Basic developers can write Excel Visual Basic macros that access the SoftMax Pro Software through its Automation API. A working knowledge of Excel Visual Basic is required. SoftMax Pro Excel macro development requires the installation of the same Excel Add-In as is described in [Installing the Excel Add-In on page 19](#).



Note: The development process described in this section is significantly different from the Visual Basic .NET example on page 11. Excel Visual Basic macros and Visual Basic .NET are very different software development environments.

Accessing the SoftMax Pro Software Automation API

The SoftMax Pro Software Automation API described in [Automation Interface Overview on page 7](#) is accessed from Excel Visual Basic by using a tool called Excel-DNA. To learn more, go to <http://exceldna.codeplex.com/>. Excel-DNA enables developers to create a wrapper around the SoftMax Pro Software Automation interface, resulting in the following differences.

Automation Command Names

To invoke an automation command from Excel Visual Basic, prefix the SoftMax Pro Software Automation command name with **SMP**.

For example, to invoke the **GetInstrumentStatus** automation API command from Excel Visual Basic, you must invoke **SMPGetInstrumentStatus**.

Invocation Mechanism

Instead of invoking methods directly (for example, **AutomationObject.OpenDrawer()** as shown in the example on page 9), commands are passed as arguments to an invocation of **Application.Run**.

For example, **Application.Run "SMPCloseDrawer"** as shown in [SoftMax Pro Excel VBA Example on page 17](#).

Synchronous Method Calls

Instead of checking the contents of an event for the result of an automation command, the result is returned directly to the Excel Visual Basic statement issuing the command.

For example, **drawerStatus = Application.Run("SMPGetDrawerStatus")** as shown in [SoftMax Pro Excel VBA Example on page 17](#).

Error Detection

Two new methods are available to Excel Visual Basic for error handling.

- **CommandError** = **Application.Run("SMPHasError")**
- **ErrorMessage** = **Application.Run("SMPGetErrorMessage")**

Both are included in [SoftMax Pro Excel VBA Example on page 17](#).

Instrument Status

No events are generated. Instead, invoke **SMPGetInstrumentStatus**.

New Command Names

To work within the constraints of the technology employed in this solution, some Excel Macro Commands do not conform exactly to the SMP prefix mechanism detailed in [Invocation Mechanism on page 16](#).

- The automation API command **SelectSection** has two Excel Visual Basic equivalents:
 - ◆ `SMPSelectSectionByName(string sectionName)`
 - ◆ `SMPSelectSectionByNumber(int sectionNumber)`
- The automation api command **Initialize** has three Excel Visual Basic equivalents:
 - ◆ `SMPIinitialize()`
 - ◆ `SMPIinitializeByServer(string server)`
 - ◆ `SMPIinitializeByServerAndPort(string server, int port)`

SoftMax Pro Excel VBA Example

A working spreadsheet called **VbaMacroExample.xlsxm** is included under the **SoftMax Pro 7.0 Automation SDK\ExcelAddIn** folder, as described in [Installing the Excel Add-In on page 19](#). Please read the notes within the spreadsheet before executing the Visual Basic macro it contains.

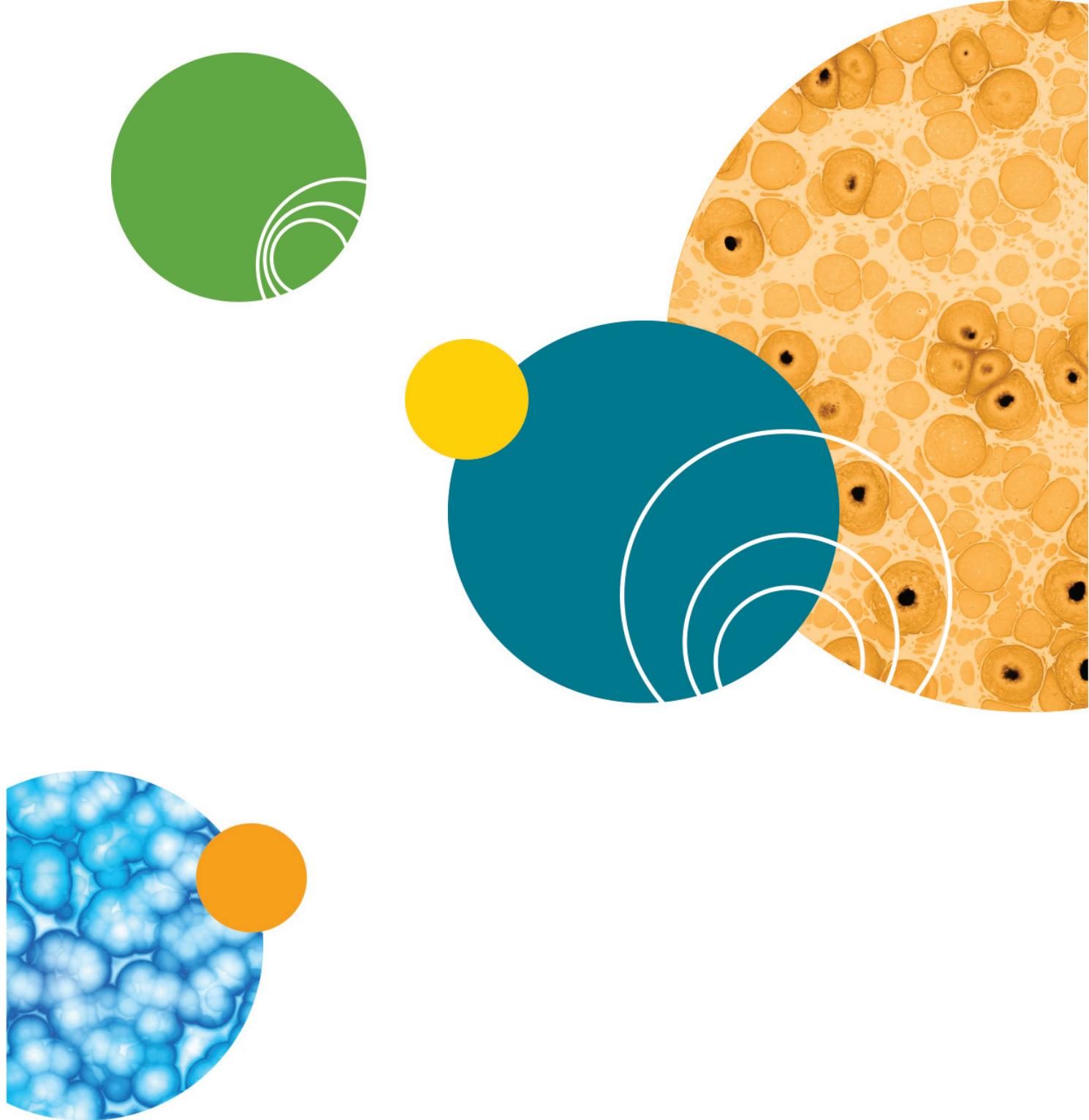
The following shows the code for example Visual Basic macro.

```
Sub AcquireData()
    Dim initialized As Boolean
    Dim disposed As Boolean
    Dim drawerStatus As String
    Dim plateData As String
    Dim commandError As Boolean
    Dim errorMessage As String

    Rem Connect to a running instance of SMP
    initialized = Application.Run("SMPIinitializeByServer", "localhost")

    If initialized Then
        Rem For testing purposes turn on the simulator
        Application.Run "SMPSetSimulationMode", True
        Rem Execute some sample/example commands
        Application.Run "SMPCloseDrawer"
        drawerStatus = Application.Run("SMPGetDrawerStatus")

        Rem Select an initial plate section to make sure we have the experiment selected
        Application.Run "SMPSelectSectionByName", "Plate1@Expt1"
        Rem Check the plate section selection completed correctly, 'HasError' checks the last command executed
        commandError = Application.Run("SMPHasError")
        If commandError Then
            Rem An error has occurred
            errorMessage = Application.Run("SMPGetErrorMessage")
        Else
            Rem Add a new Plate section
            Application.Run "SMPNewPlate"
            numberPlateSections = Application.Run("SMPGetNumberPlateSections")
            Rem Read the plate in the reader
            Application.Run "SMPStartRead"
            Rem Get the data from the plate
            plateData = Application.Run("SMPGetDataCopy")
            Rem Put the data into Excel
            Dim dataArray As Variant
            dataArray = Split(plateData, Chr(9))
            ActiveWorkbook.Sheets("Sheet1").Select
            For iRow = 0 To 8
                For iCol = 0 To 14
                    Cells(iRow + 1, iCol + 1) = dataArray(iRow * 15 + iCol)
                Next iCol
            Next iRow
        End If
        Rem tidy up
        Application.Run "SMPSetSimulationMode", False
        disposed = Application.Run("SMPDispose")
    End If
End Sub
```



Using SoftMax Pro Excel Workflows

Introduction

The SoftMax® Pro Excel workflows are designed to augment the industry-leading handling of Plate Format Data by the SoftMax Pro Software with Excel-based handling of List Format Data. You can use SoftMax Pro Excel workflows to run discontinuous kinetic reads, multiplexed reads, kinetic well scan reads, and temperature-triggered reads.

The SoftMax Pro Automation SDK is the underlying mechanism used by SoftMax Pro Excel workflows to access SoftMax Pro Software functionality. If you want to write your own workflows, you need to be familiar with the available Automation Commands. See [SoftMax Pro Automation Commands on page 47](#).

The SoftMax Pro Excel workflow feature is provided as a standard Excel Add-In and can be used effectively through the application of the Excel Charting & Trend Line features along with the following Microsoft Add-Ins:

- Analysis ToolPak
- Solver

The Excel Add-In is compatible with the following versions of Microsoft Excel:

- Microsoft Excel 2007 (version 12)
- Microsoft Excel 2010 (version 14)
- Microsoft Excel 2013 (version 15)

A supported version of Microsoft Excel must be installed on the same computer as the SoftMax Pro Software.

The Excel Add-In has been tested on 32-bit and 64-bit versions of Windows 7, Windows 8, and Windows 8.1.

Before you can use Excel to create and run automated workflows, you must install the custom Add-In in Excel. With the Add-In installed and macros enabled you can run or edit the provided workflows or create your own.

You can write a new custom formula provided you have knowledge of Excel Visual Basic. All custom formulas can be used in a SoftMax Pro Excel workflow. See [Custom Excel Formulas on page 42](#).

Installing the Excel Add-In

1. Start Microsoft Excel.
2. Click the **Developer** tab in the ribbon and then click **Add-Ins**.
If the **Developer** tab is not enabled, right-click the ribbon and then select **Customize the Ribbon**. In the **Excel Options** dialog, enable the **Developer** tab.
3. In the **Add-Ins** dialog, click **Browse**.
4. In the **Browse** dialog, navigate to the Program Files folder where the SoftMax Pro Software is installed:
 - ◆ For a 32-bit installation, go to **C:\Program Files\Molecular Devices**.
 - ◆ For a 64-bit installation, go to **C:\Program Files (x86)\Molecular Devices**.
5. Navigate to **SoftMax Pro 7.0 Automation SDK\ExcelAddIn**.
6. Select the add-in file for your version of Microsoft Office:
 - ◆ For a 32-bit version of Office, click the **SoftMaxPro6.xll** file.
 - ◆ For a 64-bit version of Office, click the **SoftMaxPro6x64.xll** file.
7. Click **OK**.
8. In the **Add-Ins** dialog, select the **SoftMaxPro6** or **SoftMaxPro6x64** check box.
9. Click **OK**.

Using the Provided Excel Workflows

With the Add-In installed you can run or edit the provided Excel workflows.

Accessing the Provided Excel Workflows

The provided Excel workflows are installed in the same folder as the Add-In files.

- For a 32-bit installation, the workflows are in the following folder:
C:\Program Files\Molecular Devices\SoftMax Pro 7.0 Automation SDK\ExcelAddIn\Examples
- For a 64-bit installation, the workflows are in the following folder:
C:\Program Files (x86)\Molecular Devices\SoftMax Pro 7.0 Automation SDK\ExcelAddIn\Examples

The Excel workflow files have an extension of **.xlsm**.



Note: Before running or editing a provided Excel workflow, save a copy of the workflow on your computer.

Layout of an Excel Workflow

All SoftMax Pro workflows must be contained in one Excel Workbook, including the supplied workflows and any customized workflows.

The supplied workflows are color coded for readability. The different Workflow Statement Types each have their own colors. The color coding does not affect the execution of workflows.

- Initialization statements are in orange.
- Automation Command statements are in yellow.
- Worksheet Formatting statements are in green.
- Workflow Flow Control statements are in gray.

Free-format statement arguments are in pink for each statement type. See [Workflow Statement Types on page 23](#).

Additionally, a list of valid instrument types is provided, so that you can copy and paste the appropriate instrument name into place in the workflow that you are editing. This helps prevent mismatches in the format of the name of the instrument. See [Instrument Connectivity on page 44](#).

Features of the SMPWorkflow Worksheet

The **SMPWorkflow** worksheet contains an **Execute** button that you click to start the workflow.

Each statement of the workflow is on a single row that starts in column B and can continue in the columns to the right with parameters or formatting information. Workflow Loop Control statements start in column A.

After you click **Execute**, the first statement is read and executed.

- The first blank column in the statement row indicates the end of the statement, and then the next statement in the following row is executed.
- The first blank row indicates the end of the workflow.

See [Running Excel Workflows on page 21](#).

Excel Visual Basic Components

A Visual Basic module called **SMPWorkflow** contains logic to initiate the invocation of Workflow Statements, worksheet formatting logic, and custom Excel formulas. See [Custom Excel Formulas on page 42](#).

Visual Basic message can appear with the following types of information:

- The workflow is being validated before execution.
- The workflow is being executed.
- The workflow is paused, and the reason for the pause is stated.
- An error was detected.

Running Excel Workflows

To run a new workflow, you need an open SoftMax Pro Excel workflow with no saved data in the worksheet. The Excel Add-In must be installed with macros enabled. The SoftMax Pro Software must be running.

There are three operations you can perform in a workflow:

- Executing a Continuous Workflow, see page 21
- Continuing a Discontinuous Workflow, see page 21
- Canceling a Workflow, see page 22

In most workflows, the user running the SoftMax Pro Software connects to the instrument before starting the workflow. If you want the workflow to connect to the instrument without user intervention, then see [Instrument Connectivity on page 44](#).

Executing a Continuous Workflow

This procedure runs the workflow once and saves the acquired data in the Excel worksheet.

1. Make sure that the instrument is connected to the computer and that the instrument is powered on.
2. Start the SoftMax Pro Software.
3. Open the Excel workflow that you want to run.
4. In the SMPWorkflow worksheet, click **Execute**.
5. When the workflow completes, save your worksheet.

Continuing a Discontinuous Workflow

This procedure uses the example of running a fluorescence read followed once every hour. The List Format saves the acquired data in separate rows after each read. See [Data Management with Excel-Based List Format Data on page 22](#).

1. Make sure that the instrument is connected to the computer and that the instrument is powered on.
2. Start the SoftMax Pro Software.
3. In the SoftMax Pro Software, connect to the instrument.
4. Open the Excel workflow that you want to run.
5. In the SMPWorkflow worksheet, click **Execute**.
6. When the workflow completes save and close your Excel workbook. The data worksheet has one row of data.
7. After the correct amount of time has passed, open the Excel workflow again.
8. Make sure that the instrument is connected to the computer and that the instrument is powered on.
9. Start the SoftMax Pro Software.
10. In the SoftMax Pro Software, connect to the instrument.
11. In the SMPWorkflow worksheet, click **Execute**.
12. When the workflow completes save and close your Excel workbook. Each worksheet has now has two rows of data.

You can continue these step for as many reads as you require.

Data Management with Excel-Based List Format Data

After executing an Excel workflow to acquire data with the SoftMax Pro Software, the data is written to Excel in List Format style and does not overwrite existing data in Excel worksheets. This can allow for kinetic reads with variable intervals, such as in a discontinuous workflow.

For example, if you execute a read once every hour, then after the first read the worksheet would have one row of data similar to the following:

Table 5-1:

Date/Time	Elapsed Time	Temperature	A1	A2
2/20/2013 18:57	0	31.5	0.30	0.21

After the second read, the worksheet would have two rows of data similar to the following:

Table 5-2:

Date/Time	Elapsed Time	Temperature	A1	A2
2/20/2013 18:57	0	31.5	0.30	0.21
2/20/2013 19:57	3600	31.5	0.31	0.25

See [Continuing a Discontinuous Workflow on page 21](#).

Cancelling a Workflow

When you run a SoftMax Pro Excel workflow, a dialog appears in front of the main SoftMax Pro Software window, to show that the software is in automation mode.

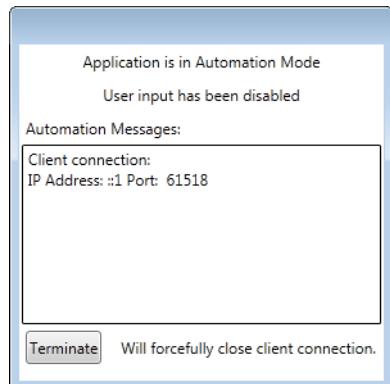


Figure 5-1: SoftMax Pro Software Automation Mode

If you need to stop the automation process and regain control of the SoftMax Pro Software, click **Terminate**. Then in the Excel workflow, close all open dialogs.

Creating a New Excel Workflow

To create a new SoftMax Pro Excel workflow, open an existing workflow and then save it as new workbook. This process ensures that all required macros and dialogs are in place for the new workflow.

Moving the Execute Button

When you create a long workflow, you might want to move the **Execute** button.



To move the **Execute** button, right-click the button and then drag it to its new location.

Editing Excel Workflows

When editing workflows, keep in mind how workflow statements are interpreted when the workflow is run.

- The first blank column in the statement row indicates the end of the statement, and then the next statement in the following row is executed.
- The first blank row indicates the end of the workflow.

Workflow Statement Types

To help understand how to read and write workflows, consider each Workflow Statement to be one of four statement types.

Table 5-3: Workflow Statement Types

Statement Type	Description
Initialization Statements	Initialization statements augment available Automation Commands when setting up the workflow execution environment.
Automation Command Statements	Automation command statements are available as part of the SoftMax Pro Software Automation API.
Worksheet Formatting Statements	Worksheet formatting statements define the format of data to be written to worksheets, and can retrieve the data from the SoftMax Pro Software and write it to the Excel worksheet.
Workflow Flow Control Statements	Workflow flow control statements cause the workflow to repeat a section of statements or pause until a given condition is satisfied.

Initialization Statements

Initialization statements augment available Automation Commands when setting up the workflow execution environment.

In the provided example workbooks, initialization statements are color coded orange.

PlateSize Statement

Purpose

Define the size of the microplate used on the instrument.

Arguments

PlateSize has one argument:

- The number of wells in the microplate.

The following microplate sizes are supported.

- ◊ 24-well microplates
- ◊ 48-well microplates
- ◊ 96-well microplates
- ◊ 384-well microplates

Defaults

If the PlateSize Statement is not coded, the microplate size defaults to 96 wells.

PlateSize Statement Examples

PlateSize	96
PlateSize	384

Automation Command Statements

Automation command statements are available as part of the SoftMax Pro Software Automation API.

In the provided example workbooks, automation command statements are color coded yellow.

For descriptions of and parameters for the available automation commands, see [SoftMax Pro Automation Commands on page 47](#).

The following figure shows examples of some automation command statements. The first column contains the command name, and the columns to the right contain the parameter values for the command.

CloseAllDocuments	
OpenFile	C:\temp\philippe.spr
SetReader	COM3
SetSimulationMode	FALSE
SelectSectionByName	Plate1
NewPlate	
StartRead	
GetDataCopy	

Most automation commands perform an action. However, some commands retrieve data from the SoftMax Pro Software. These commands make their data available to be written to worksheets or used to control the flow of a workflow.

- Worksheet formatting statements define the format of data to be written to worksheets, and can retrieve the data from the SoftMax Pro Software and write it to the Excel worksheet. For an example, see [ProcessCommandResult Statement on page 37](#).
- Workflow flow control statements cause the workflow to repeat a section of statements or pause until a given condition is satisfied. For an example, see [WaitUntil Statement on page 40](#).

Worksheet Formatting Statements

Worksheet formatting statements define the format of data to be written to worksheets, and can retrieve the data from the SoftMax Pro Software and write it to the Excel worksheet.

In the provided example workbooks, formatting statements are color coded green.

There are three categories of worksheet formatting statements.

Table 5-4: Worksheet Formatting Statement Categories

Statement Type	Description
Creating and Selecting Worksheets	As your workflow retrieves data from the SoftMax Pro Software you need to create worksheets in which to save or write this data. This section describes how to create new worksheets and then target these worksheets to receive specific data from within the workflow.
Defining Write Formats	Writing to a worksheet is broken into two parts. First, a workflow has to define the format in which the data is written. Next, the data is written to the worksheet. This section describes how to define formats.
Writing to Worksheets	When a workflow has data to write to an Excel worksheet it is directed to perform the write by defining which Write Format to use. The section describes how to write fixed values, calculated values, and acquired data to a worksheet.

Creating and Selecting Worksheets

As your workflow retrieves data from the SoftMax Pro Software you need to create worksheets in which to save or write this data. This section describes how to create new worksheets and then target these worksheets to receive specific data from within the workflow.

Worksheet Statement

Purpose

This statement can create a new Excel worksheet and select an existing Excel worksheet. When a worksheet is added it is automatically selected. Worksheet formatting statements work mostly against the worksheet currently selected.

Arguments

Worksheet has two arguments:

- Operation

This is either one of two values:

- ◆ Select
- ◆ Add

- Name (optional)

This is the name of the worksheet to be selected or added. This is optional.

Defaults

If the Name argument is not given, a new worksheet will be added. The name of the worksheet will be **Sheetx** where **x** is the lowest number required to create a new worksheet with a unique name in the workbook.

Worksheet Statement Examples

Create a new worksheet named Summary:

Worksheet	Add	Summary
-----------	-----	---------

Select an existing worksheet named Summary:

Worksheet	Select	Summary
-----------	--------	---------

Add a new worksheet named Sheet1 in an almost empty workbook:

Worksheet	Add
-----------	-----

Add a worksheet named Sheet3 in a workbook that already has worksheets named Sheet1 and Sheet2:

Worksheet	Add
-----------	-----

Defining Write Formats

Writing to a worksheet is broken into two parts. First, a workflow has to define the format in which the data is written. Next, the data is written to the worksheet. This section describes how to define formats.

CellFormat Statement

Purpose

Defines a format in which to write data into one Excel worksheet cell. It is used in conjunction with the following two workflow statements:

- [FormatWorksheet Statement, see page 36](#)
- [ProcessCommandResult Statement, see page 37](#)

Arguments

CellFormat has up to four arguments:

- Format Name
This must be unique within the workflow. Both the FormatWorksheet and ProcessCommandResult statements reference this name.
- Target Worksheet Column
Defines which worksheet column will be written into in the following format:
`worksheet-name:worksheet-column or worksheet-column`
If only the worksheet column is specified, then the worksheet row to be used is determined to be the next empty cell in the column, working down from the top of the target worksheet.
- Row Number (optional)
Allows output to always be written to a specific row. If omitted, then the output is written to next free row in the target column. It can be useful for making sure headings are written out only once in discontinuous reads.
- Cell Content (sometimes not required, see defaults below)
Optionally defines what will be written into the worksheet column. When writing an Excel formula (e.g. '=Sum(A1:A5)' the following keywords can be used and will be substituted automatically immediately prior to the formula being written to a worksheet,
 - ◆ #Column: The workflow keeps track of where it writes data into a worksheet. This keyword should be used in conjunction with the FormatWorksheet WriteRow statement to apply operations to whole columns of data. See the following examples.
 - ◆ #RowCount: This provides the number of rows containing data values. As per #Column, this keyword should be used in conjunction with the FormatWorksheet WriteRow statement. See the following examples.
 - ◆ #Worksheet: This references the currently selected worksheet. However, this keyword can be used in conjunction with both the FormatWorksheet WriteRow and FormatWorksheet WriteCell statements. See the following examples.

A formula entered in the workflow evaluates before it is run if not entered with a single quotation mark prefix. For example, instead of entering **=SUM(A1: A5)** enter '**=SUM(A1: A5)**'.

Defaults

If the worksheet name is omitted from the Target Worksheet Column argument, the currently selected worksheet is used as the target for the write. See [Worksheet Statement on page 26](#).

Cell content can be omitted if the value to be written to the cell is the result from a previously issued Automation Command. See [ProcessCommandResult Statement on page 37](#).

CellFormat Statement Example 1

This example shows how to write the word “Temperature” into cell D1 of the Excel worksheet, which is selected when a subsequent FormatWorksheet statement is executed. The name of the format is MyTemperatureHeading.

CellFormat	MyTemperatureHeading	D	1	Temperature
------------	----------------------	---	---	-------------

...

Worksheet	Add
FormatWorksheet	WriteCell

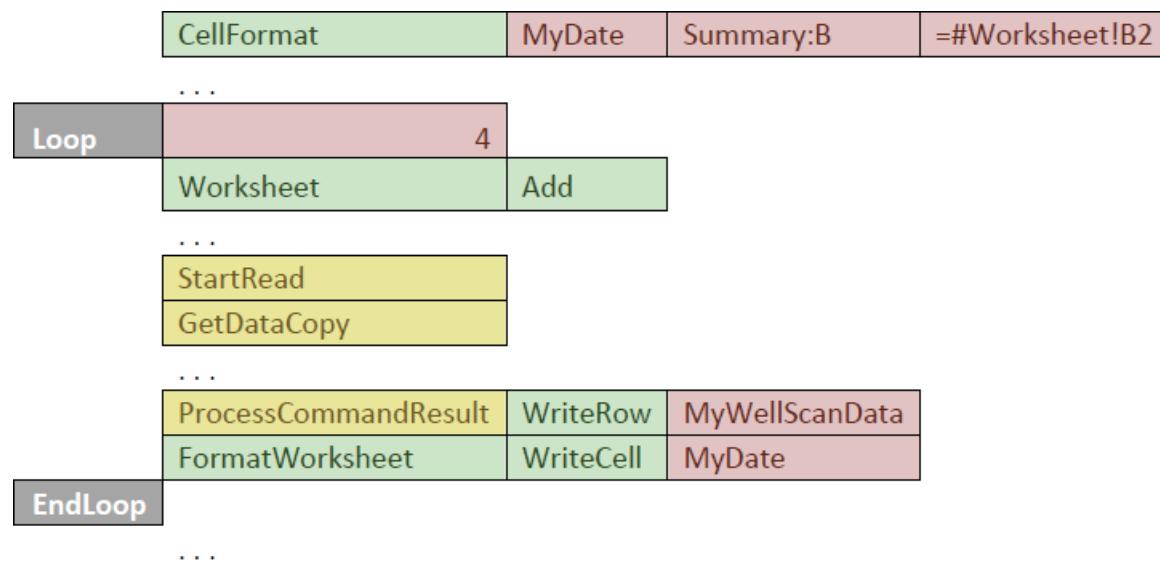
The output written to Excel is highlighted in red in the following figure.

A	B	C	D	E	F
1	Date/Time	Elapsed Time	Temperature	A1	A2
2	2/21/2013 14:52	0	35	1000	
3	2/21/2013 14:52			1841.47098	1841.4
4	2/21/2013 14:52			1909.29743	1909.2
5	2/21/2013 14:52			1141.12001	1141.1
6	2/21/2013 14:52			243.1975	243
7	2/21/2013 14:52			41.07573	41.0
8	2/21/2013 14:52			720.5845	720
9	2/21/2013 14:52			1656.9866	1656
10	2/21/2013 14:52			1989.35825	1989.3
11					
12					
13					
14					
15					

SMPWorkflow Summary Sheet1 Sheet2 Sheet3 Sheet4

Cell Format Statement Example 2

In this example, the workflow performs four Well Scan reads, writing one result into each of Sheets 1 through 4. The CellFormat statement in this example is responsible for putting a reference in the Summary worksheet to the date and time recorded in each of Sheets 1 through 4.



The two most significant CellFormat arguments are interpreted as follows:

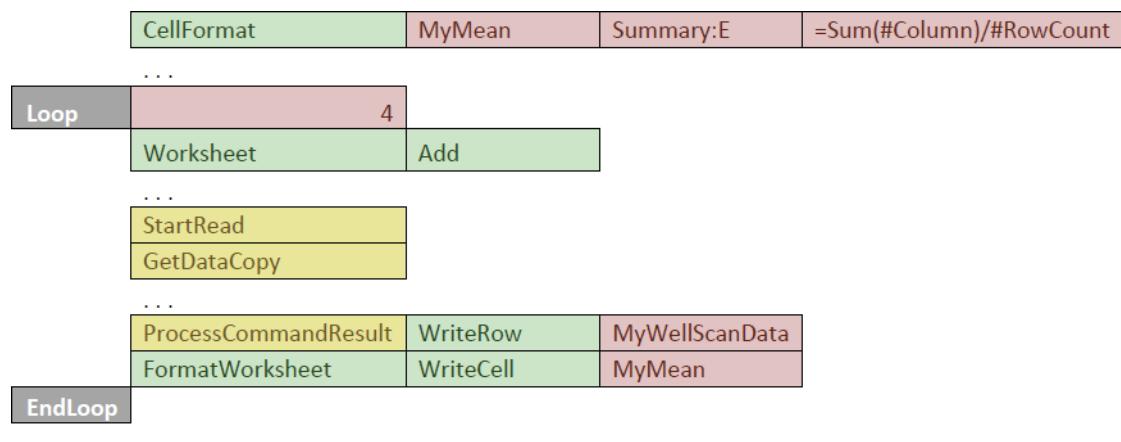
- **Summary:B**
Write output to the next free or empty cell in column B of the Summary worksheet.
- **=#Worksheet!B2**
The information written into column B where **#Worksheet** will be the name of the currently active worksheet, for example, **Sheet1**.

The output written to Excel is highlighted in red in the following figure.

The screenshot shows an Excel spreadsheet with multiple tabs at the bottom: SMPWorkflow, Summary, Sheet1, Sheet2, Sheet3, and Sheet4. The Summary tab is active. The data is organized into columns A through F. Column A contains row numbers from 1 to 15. Column B is labeled 'Date/Time' and contains the values '2/21/2013 14:52', '2/21/2013 14:53', '2/21/2013 14:53', and '2/21/2013 14:55' respectively for rows 2, 3, 4, and 5. Columns C, D, E, and F contain numerical data: Elapsed Time (0, 33, 67, 154), Temperature (35, 35, 35, 35), and two columns A1 and A2 both containing the value '1171.454556'. Rows 6 through 15 are empty.

CellFormat Statement Example 3

In this example, the workflow performs four Well Scan reads, writing one result into each of Sheets 1 through 4. The CellFormat statement in this example is responsible for putting calculating the mean value of each point in a well, for one read, into the Summary worksheet.



The two most significant CellFormat arguments are interpreted as follows:

- **Summary:B**
Start writing write output to the next free or empty row starting in column E of the Summary worksheet.
- **=Sum(#Column)/#RowCount**
The SoftMax Pro Software has tracked where acquired data has been written in each worksheet. Because of this the workflow is able to substitute the #Column and #RowCount with real values that match the currently active worksheet. The substitution is shown in the 'fx' cell in the following figure.

The output written to Excel the final time through the loop is highlighted in red in the following figure.

	A	B	C	D	E	F	G
1		Date/Time	Elapsed Time	Temperature	A1	A2	A3
2		2/21/2013 14:52		0	35	1171.454556	1171.454556
3		2/21/2013 14:53		33	35	1171.454556	1171.454556
4		2/21/2013 14:53		67	35	1171.454556	1171.454556
5		2/21/2013 14:55		154	35	1171.454556	1171.454556
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							

The screenshot shows a Microsoft Excel spreadsheet titled "SNPWorkflow". The active sheet is "Summary". The data is organized into columns A through G. Column A contains row numbers from 1 to 15. Columns B, C, and D contain experimental data: Date/Time, Elapsed Time, and Temperature respectively. Column E contains calculated mean values for each row, starting with "A1" and continuing with "A2" and "A3". The formula for cell E5 is displayed as "=SUM(Sheet4!E2:E10)/9". The last three rows (12, 13, 14) are empty. The bottom of the screen shows the standard Excel ribbon tabs: Back, Forward, Stop, Home, Insert, Page Layout, Formulas, Data, Page Break Preview, and View. The status bar at the bottom right shows the identifier "5049094 A".

CellFormat Statement Example 4

In this example, when subsequent ProcessCommandResult statements reference MyTemperature, they write the returned value of the last executed Automation Command into the first blank cell found in Column D of the Excel worksheet that is selected at the time the ProcessCommandResult is executed.

CellFormat	MyTemperature	D
...		
GetTemperature		
ProcessCommandResult	WriteCell	MyTemperature

The output written to Excel is highlighted in red in the following figure.

	A	B	C	D	E	F
1		Date/Time	Elapsed Time	Temperature	A1	A2
2		2/21/2013 14:52		0 35	1000	
3		2/21/2013 14:52			1841.47098	1841.4
4		2/21/2013 14:52			1909.29743	1909.2
5		2/21/2013 14:52			1141.12001	1141.1
6		2/21/2013 14:52			243.1975	243.
7		2/21/2013 14:52			41.07573	41.0
8		2/21/2013 14:52			720.5845	720.
9		2/21/2013 14:52			1656.9866	1656.
10		2/21/2013 14:52			1989.35825	1989.3
11						
12						
13						
14						
15						

SMPWorkflow Summary Sheet1 Sheet2 Sheet3 Sheet4

CellFormat Statement Example 5

In this example, instead of writing a standard Excel formula to Column C of the selected Excel worksheet, the following statement writes in a call to a custom Visual Basic function called ElapsedTime. For details, see [Custom Excel Formulas on page 42](#).

CellFormat	MyElapsedTime	C	=ElapsedTime(\$B\$2, "B")
------------	---------------	---	---------------------------

The output written to Excel is highlighted in red in the following figure.

	A	B	C	D	E	F
1	Date/Time	Elapsed Time	Temperature	A1	A2	
2	2/21/2013 14:52	0	35	1171.454556	1171.454556	
3	2/21/2013 14:53	33	35	1171.454556	1171.454556	
4	2/21/2013 14:53	67	35	1171.454556	1171.454556	
5	2/21/2013 14:55	154	35	1171.454556	1171.454556	
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

SMPWorkflow Summary Sheet1 Sheet2 Sheet3 Sheet4

RowFormat Statement

Purpose

Defines a format in which to write data into one Excel worksheet row. It is used in conjunction with the following two workflow statements:

- [FormatWorksheet Statement, see page 36](#)
- [ProcessCommandResult Statement, see page 37](#)

The RowFormat statement is designed to make it easy to write a plate of data as a row in an Excel worksheet. It is also useful in writing ancillary information related to plate data, such as well names (A1, A2, and so on) and Group Name Assignments (Control, Unknown, and so on).

As the Automation Command GetDataCopy returns a date followed by the well data, the simplest, default form of this command formats that data. See [RowFormat Statement Example 1 on page 34](#).

Arguments

RowFormat has a variable number of arguments.

- Format Name

This must be unique within the workflow. Both the FormatWorksheet and ProcessCommandResult statements reference this name.

- Start Column (optional)

RowFormat always writes to the currently selected Excel worksheet.

- Row Number (optional)

Allows output to always be written to a specific row. If omitted, then the output is written to next free row in the target column. It can be useful for making sure headings are written out only once in discontinuous reads.

- Row Contents (optional)

The number of subsequent arguments is variable. There are three keywords that can be used to control what is written in a row:

◆ Date/Time: This causes the Date/Time to be written.

◆ Wells: This causes the well related information to be written. For example, 96 columns of data in a 96-well microplate.

◆ Blank: This leaves a column in the row blank, typically to be populated with a CellFormat related entry such as temperature or elapsed time.

Defaults

If no Start Column is given, then the row starts being written in Column A.

If no Row Contents are given, then the default is Date/Time followed by Wells.

For a full workflow using default RowFormat values, see [RowFormat Statement Example 1 on page 34](#).

RowFormat Statement Example 1

This example shows the simplest definition of a RowFormat statement where all default values are taken. The worksheet heading line (row 1) and the data itself (rows 2 through 10) use the same row format.

A	B	C	D
1	RowFormat	MyFormat	
2	Worksheet	Add	
3	CloseAllDocuments		
4	OpenFile	C:\temp\simple.spr	
5	SetReader	OFFLINE	GEMINI EM
6	SetSimulationMode	TRUE	
7	FormatWorksheet	WriteRow	MyFormat
8	StartRead		
9	GetDataCopy		
10	ProcessCommandResult	WriteRow	MyFormat
11	Execute		
12			
13			

This complete workflow creates the following Excel worksheet data file from one nine-point well scan read using the instrument simulator.

A	B	C	D	E	F	G	H	I
1	Date/Time	A1	A2	A3	A4	A5	A6	A8
2	2/21/2013 16:32	1000	1000	1000	1000	1000	1000	1000
3	2/21/2013 16:32	1841.47098	1841.47098	1841.47098	1841.47098	1841.47098	1841.47098	1841.47098
4	2/21/2013 16:32	1909.29743	1909.29743	1909.29743	1909.29743	1909.29743	1909.29743	1909.29743
5	2/21/2013 16:32	1141.12001	1141.12001	1141.12001	1141.12001	1141.12001	1141.12001	1141.12001
6	2/21/2013 16:32	243.1975	243.1975	243.1975	243.1975	243.1975	243.1975	243.1975
7	2/21/2013 16:32	41.07573	41.07573	41.07573	41.07573	41.07573	41.07573	41.07573
8	2/21/2013 16:32	720.5845	720.5845	720.5845	720.5845	720.5845	720.5845	720.5845
9	2/21/2013 16:32	1656.9866	1656.9866	1656.9866	1656.9866	1656.9866	1656.9866	1656.9866
10	2/21/2013 16:32	1989.35825	1989.35825	1989.35825	1989.35825	1989.35825	1989.35825	1989.35825
11								

RowFormat Statement Example 2

In this example, the RowFormat starts writing its rows of data in column C and leaves a blank column between the date/time and the well data.

A	B	C	D	E	F	G
1	RowFormat	MyFormat	C	Date/Time	Blank	Wells
2	Worksheet	Add				
3	CloseAllDocuments					
4	OpenFile	C:\temp\simpletoo.spr				
5	SetReader	OFFLINE	GEMINI EM			
6	SetSimulationMode	TRUE				
7	FormatWorksheet	WriteRow	MyFormat			
8	Loop	5				
9	StartRead					
10	GetDataCopy					
11	ProcessCommandResult	WriteRow	MyFormat			
12	Pause		60			
13	EndLoop					
14						
15						
16						
17	Execute					

This complete workflow creates the following Excel worksheet data file from five endpoint reads using the instrument simulator.

A	B	C	D	E	F	G	H	I	J	K
1		Date/Time		A1	A2	A3	A4	A5	A6	A7
2		2/22/2013 9:44		0.01	0.02	0.03	0.04	0.05	0.06	0.0'
3		2/22/2013 9:46		0.01	0.02	0.03	0.04	0.05	0.06	0.0'
4		2/22/2013 9:47		0.01	0.02	0.03	0.04	0.05	0.06	0.0'
5		2/22/2013 9:48		0.01	0.02	0.03	0.04	0.05	0.06	0.0'
6		2/22/2013 9:49		0.01	0.02	0.03	0.04	0.05	0.06	0.0'
7										
8										
9										
10										
11										
12										
13										
14										
15										
16										
17										

Writing to Worksheets

When a workflow has data to write to an Excel worksheet it is directed to perform the write by defining which Write Format to use. The section describes how to write fixed values, calculated values, and acquired data to a worksheet.

When writing into an Excel worksheet there are two types of write statements that can be placed in a SoftMax Pro Excel workflow.

Table 5-5: Write Statements

Statement Type	Description
FormatWorksheet Statement	This statement writes fixed or calculated values to a worksheet. The running instance of the SoftMax Pro Software does not directly provide data for this type of write.
ProcessCommandResult Statement	This statement takes data returned from a SoftMax Pro Software Automation API command and writes it into a worksheet.

When writing to a worksheet the information to be written should already have been defined by a Write Format. See [Defining Write Formats on page 27](#).

FormatWorksheet Statement

Purpose

This statement writes fixed or calculated values to a worksheet. The running instance of the SoftMax Pro Software does not directly provide data for this type of write.

Examples include writing worksheet column headings and writing Excel formulas into worksheet cells.

Arguments

FormatWorksheet has two arguments:

- Type of Write
This is either one of two values:
 - ◆ WriteCell
 - ◆ WriteRow
- Name of Format
 - ◆ If the first argument specifies WriteCell, then argument this must reference a previously defined CellFormat.
 - ◆ If the first argument specifies WriteRow, then this argument usually references a previously defined RowFormat.
The exception is when a formula using keywords appears in a CellFormat. See [CellFormat Statement Example 3 on page 30](#).

Defaults

No default values

Examples

See the examples given in the [CellFormat Statement on page 27](#) and [RowFormat Statement on page 33](#).

ProcessCommandResult Statement

Purpose

This statement takes data returned from a SoftMax Pro Software Automation API command and writes it into a worksheet.

Examples of commands which return data are GetTemperature, GetGroupNameAssignments, and GetDataCopy.

The last automated command that was executed prior to the ProcessCommandResult statement is the command that is processed.

For descriptions of and parameters for the available automation commands, see [SoftMax Pro Automation Commands on page 47](#).

Arguments

ProcessCommandResult has two arguments:

- Type of Write
 - ◆ This is either one of two values:
 - ◆ WriteCell
 - ◆ WriteRow
- Name of Format
 - ◆ If the first argument specifies WriteCell, then argument this must reference a previously defined CellFormat.
 - ◆ If the first argument specifies WriteRow, then this argument this must reference a previously defined RowFormat.

Defaults

No default values

Examples

See the examples given in the [CellFormat Statement on page 27](#) and [RowFormat Statement on page 33](#).

Workflow Flow Control Statements

Workflow flow control statements cause the workflow to repeat a section of statements or pause until a given condition is satisfied.

In the provided example, Workbooks control statements are color coded gray.

Loop and EndLoop Statements

Purpose

To repeat a series of workflow statements a fixed number of times, **Loop** marks the start of the series and **EndLoop** marks the end of the series.



Note: Loop and EndLoop statements should always be written in worksheet column A to make them easy to detect. All other statements are written in worksheet column B.

Arguments

Loop has one argument:

- The number of times the series of statements should be repeated.

EndLoop has no arguments.

Defaults

No default values.

Example

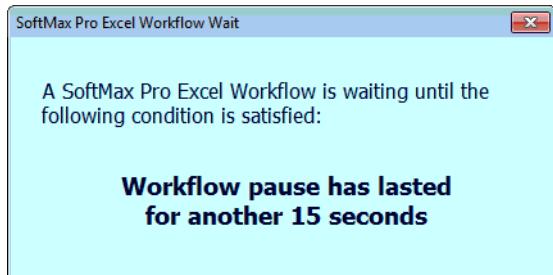
In this example, the series of statements between Loop and EndLoop repeats five (5) times. The SetTemperature and OpenDrawer statements execute only once each.

	SetTemperature	30
Loop	5	
	SelectSectionByName	BlankPlate
	NewPlate	
	StartRead	
	GetTemperature	
	GetDataCopy	
EndLoop		
	OpenDrawer	

Pause Statement

Purpose

This statement pauses workflow execution for a fixed number of seconds. When a workflow is paused a dialog appears.



Arguments

Pause has one argument:

- The number of seconds the workflow should pause

Defaults

No default values

Example

In this example, the Pause within the Loop pauses the workflow for 300 seconds (5 minutes) after each read.

Loop	10	
	StartRead	
	GetDataCopy	
	Pause	300
EndLoop		

WaitUntil Statement

Purpose

This statement pauses workflow execution until a given condition has been satisfied. A dialog appears describing the condition that needs to be satisfied to allow the workflow to continue.

Arguments

WaitUntil has four arguments:

- Automation Command

This command must return a result that can be tested in combination with the second and third arguments.

- Operator

This can be one of the following values:

- ◆ **EQ** meaning equals
- ◆ **LT** meaning less than
- ◆ **GT** meaning greater than
- ◆ **LE** meaning less than or equals
- ◆ **GE** meaning greater than or equals

- Target Value

The target value against which the result from argument one is tested.

- Description

This gives the reason the workflow is waiting and is displayed to the user while the pause is in effect. This optional argument can be used to make the wait reason more descriptive by adding context or using a local language.

Defaults

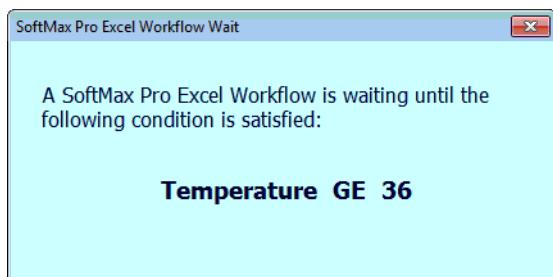
If a description argument is not supplied the first three arguments are concatenated into a default description.

Examples

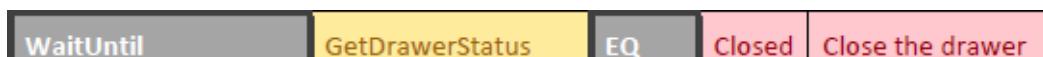
The following WaitUntil statement pauses until the instrument incubator is equal to, or greater than, 36 degrees.



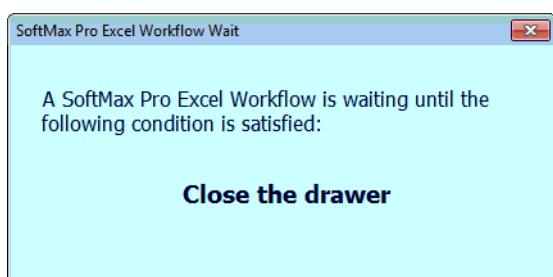
While the workflow is paused the following dialog appears.



The following WaitUntil statement pauses until the instrument drawer is closed.



While the workflow is paused the following dialog appears.



Custom Excel Formulas

You can write a new custom formula provided you have knowledge of Excel Visual Basic. All custom formulas can be used in a SoftMax Pro Excel workflow. See [CellFormat Statement on page 27](#).

The ElapsedTime custom formula is already provided with the SoftMax Pro Excel workflows.

Additional custom formula should be added to the SoftMax Pro Excel Visual Basic module called SMPWorkflow.

ElapsedTime Custom Formula

Purpose

This custom formula writes a column of elapsed time values, in seconds, between successive instrument reads. This can be used to assist in plotting a graph of data collected from instrument reads over time.

Dependency

The date/time of each read must be recorded in the Excel Worksheet.

Usage

=ElapsedTime(Base Date Cell, Date Column)

where:

- **Base Date Cell** is the Excel worksheet cell containing start date/time against which all other dates are compared against
- **Date Column** is the column containing the dates to be compared against the Base Date. The Date Column must be in quotation marks, for example, "C" for column C of the worksheet.

ElapsedTime Example

In this example, the workflow uses the ElapsedTime custom formula to write the elapsed time between runs in Column B of the Sheet1 worksheet.

A	B	C	D	E	F
RowFormat	MyWellFormat	Date/Time	Blank		Wells
CellFormat	MyElapsedTimeHeading	B		Elapsed Time	
CellFormat	MyElapsedTimeFormula	B		=ElapsedTime(\$A\$2, "A")	
Worksheet	Add				
CloseAllDocuments					
OpenFile	C:\temp\simpletoo.spr				
SetReader	OFFLINE	GEMINI EM			
SetsimulationMode	TRUE				
FormatWorksheet	WriteRow	MyWellFormat			
FormatWorksheet	WriteCell	MyElapsedTimeHeading			
Loop	5				
StartRead					
GetDataCopy					
ProcessCommandResult	WriteRow	MyWellFormat			
FormatWorksheet	WriteCell	MyElapsedTimeFormula			
Pause		10			
EndLoop					

The output written to Excel appears in the following figure.

	A	B	C	D	E	F	G
1	Date/Time	Elapsed Time	A1	A2	A3	A4	A5
2	2/22/2013 11:13		0	0.01	0.02	0.03	0.04
3			19	0.01	0.02	0.03	0.04
4	2/22/2013 11:14		38	0.01	0.02	0.03	0.04
5	2/22/2013 11:14		57	0.01	0.02	0.03	0.04
6	2/22/2013 11:14		76	0.01	0.02	0.03	0.04

Visual Basic Code

```
Function ElapsedTime(ByVal startDate As Date, ByVal compareDateCol As String) As Long
    Dim compareDateColIndex As Long
    compareDateColIndex = ColumnLetterToNumber(compareDateCol)
    Cells(ActiveCell.row, compareDateColIndex).NumberFormat = "dd/mm/yy"
    Dim compareDate As Date
    compareDate = Cells(ActiveCell.row, compareDateColIndex)
    ElapsedTime = DateDiff("s", startDate, compareDate)
End Function
```

Instrument Connectivity

In most workflows, the user running the SoftMax Pro Software connects to the instrument before starting the workflow. If you want the workflow to connect to the instrument without user intervention, then see read this section.

To connect to either an instrument or an instrument simulator, use the following Automation commands in the workflow:

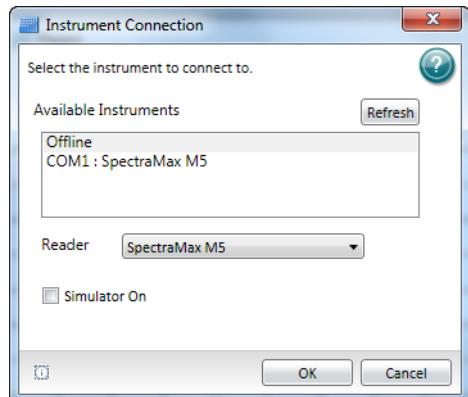
- SetReader
- SetSimulationMode

As the automation API requires the name of the instrument be in a very specific format, each model of instrument is included in the supplied example Excel workbooks. When writing or changing a workflow, you can copy and paste the appropriate instrument name into place in the workflow that you are editing. This helps prevent mismatches in the format of the name of the instrument.

Instrument Connectivity Examples

Connecting to a Real Instrument

The port connected to the instrument should be coded as part of the SetReader command, and the instrument should be taken out of simulation mode.



SetReader	COM1	SPECTRAmax M5
SetSimulationMode	FALSE	

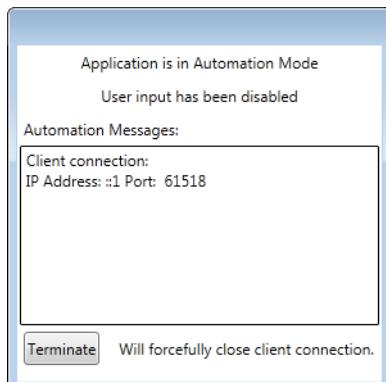
Using a Simulator

The instrument should be set as OFFLINE, and the SetSimulation command should say TRUE.

SetReader	OFFLINE	SPECTRAmax M5
SetSimulationMode	TRUE	

Troubleshooting Excel Workflows

- Make sure that the SoftMax Pro Excel Add-In is enabled in Excel.
- Make sure that the SoftMax Pro Software is started and that the Automation Mode dialog is not displayed.



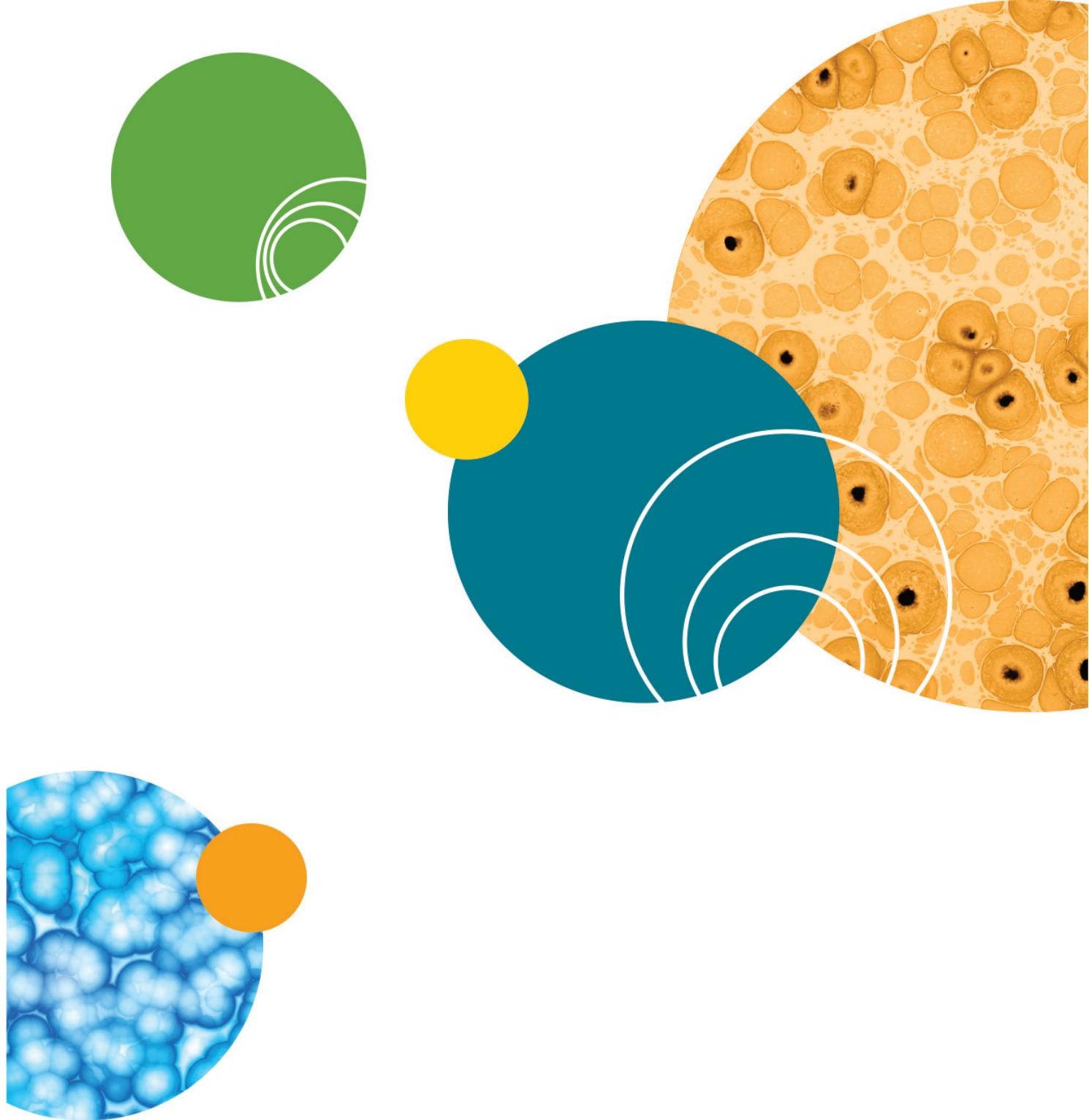
- If a formula entered in the workflow evaluates before it is run then enter with a single quotation mark prefix. For example, instead of entering **=SUM(A1: A5)** enter '**=SUM(A1: A5)**'.

Workflow Errors

Any errors that are detected by the workflow are reported in the **SMPErrors** worksheet as shown in the following figure.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	ERRORS DETECTED DURING VALIDATION OF WORKFLOW:												
2													
3	Row 1 - SoftMax Pro Excel Workflow statement RowFormat has missing/invalid arguments												
4	Row 2 - SoftMax Pro Excel Workflow statement CellFormat has missing/invalid arguments												
5	Row 3 - SoftMax Pro Excel Workflow statement CellFormat has missing/invalid arguments												
6	Row 4 - SoftMax Pro Excel Workflow statement Worksheet has missing/invalid arguments												
7	Row 9 - SoftMax Pro Excel Workflow statement FormatWorksheet has missing/invalid arguments												
8	Row 10 - SoftMax Pro Excel Workflow statement FormatWorksheet has missing/invalid arguments												
9	Row 14 - SoftMax Pro Excel Workflow statement ProcessCommandResult has missing/invalid arguments												
10	Row 15 - SoftMax Pro Excel Workflow statement FormatWorksheet has missing/invalid arguments												
11													
12													
13													
14													
15													
16	SoftMax Pro Excel Workflow errors have been detected. Please see the worksheet called 'SMPErrors' for details.												
17													
18													
19													
20													
21													
22													

You should also check for additional error messages in the SoftMax Pro Software Automation Mode dialog under Automation Messages. The errors reported in this dialog might not be reported to the workflow.



SoftMax Pro Automation Commands

SoftMax Pro automation commands are one or more concatenated words with no spaces and a closing set of parentheses. If parameters are required, the parameters are included within the closing set of parentheses.

Table 6-1: SoftMax Pro Commands

Command	Description
AppendData	Reads the current Plate section and appends the new data to the existing data.
AppendTitle	Appends text to the title of the specified section.
CloseDocument	Closes the current document.
CloseAllDocuments	Closes all the currently open documents.
CloseDrawer	Closes the specified drawer on the instrument.
Dispose	Closes the final automation application dialog.
ExportAs	Exports data in the Columns, Plate, or XML format.
ExportSectionAs	Exports the currently selected section.
GetAutosaveState	Returns the current autosave setting for the active document.
GetDataCopy	Copies data to the client by way of an event.
GetDrawerStatus	Returns the state of the specified drawer on the instrument.
GetFormulaResult	Returns the result from the specified formula.
GetGroupNameAssignments	Returns the group name assignments of the currently selected plate section.
GetInstrumentStatus	Returns the currently connected instrument status.
GetNumberPlateSections	Returns the number of plate sections in the active experiment.
GetTemperature	Returns the currently connected instrument incubator temperature.
GetVersion	Returns the version of the SoftMax Pro Automation Interface.
ImportPlateData	Imports data from a file into a Plate section in the current data file.
ImportPlateTemplate	Imports a template from a file into the current plate section.
Initialize	Initializes the Automation Interface.
NewDocument	Creates a new document.
Logon	Logs a user onto the SoftMax Pro GxP Software.
Logoff	Logs a user off from the SoftMax Pro GxP Software.
NewExperiment	Creates a new Experiment section.
NewNotes	Creates a new Notes section within the current Experiment.
NewPlate	Creates a new Plate section within the current Experiment.
OpenDrawer	Opens the specified drawer on the instrument.
OpenFile	Opens a protocol file or data file.
Quit	Exits the SoftMax Pro Software application.
SaveAs	Saves the current document as a data file or a protocol file.
SelectNextPlateSection	Selects the next plate section from the current Experiment.

Table 6-1: SoftMax Pro Commands (cont'd)

Command	Description
SelectSection	Selects a section by name or by the order of the section within the experiment.
SetReader	Selects a reader type and sets the reader status.
SetShake	Shakes the microplate.
SetSimulationMode	Sets SoftMax Pro software into simulation mode.
SetTemperature	Sets the current instrument incubator temperature.
SetTitle	Sets the title of the selected section.
SetUserName	Used with GxP systems to set the user name for a shielded protocol.
StartRead	Reads the current Plate or CuvetteSet section.
StopRead	Stops the current Plate or CuvetteSet section read.

AppendData

`Int32 AppendData()`

Purpose

Reads the current Plate section and appends the new data to the existing data.

If the current section is not a Plate section, then the next Plate section is read.

See [StartRead](#) on page 85.

Parameters

None

AppendTitle

```
Int32 AppendTitle(String titletext)
```

Purpose

Appends text to the title of the specified section.



Note: This command can alter the names of the sections within an experiment, but it cannot alter the name of the experiment.

Parameters

titletext

Type: String

The text to be appended to the section title.

Example

The following example demonstrates appending text to the title of a plate section.

```
int mAppenTitleID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.SelectSection("Plate01");
    mAppenTitleID = AutomationObject.AppendTitle("- YeOld Append");
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}

private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString());
    if( mAppenTitleID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty )
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
```

CloseDocument

```
Int32 CloseDocument()
```

Purpose

Closes the current document.

If the data has not been saved, the document is closed anyway with no user warning.

Parameters

None

CloseAllDocuments

```
Int32 CloseAllDocuments()
```

Purpose

Closes all the currently open documents.

Parameters

None

CloseDrawer

```
Int32 CloseDrawer()
```

```
Int32 CloseDrawer(String drawerType)
```

Purpose

Closes the specified drawer on the instrument.

For most instruments, this command closes the microplate drawer.

Parameters

The CloseDrawer parameters are recognized by the FlexStation 3 reader only. For all other instruments, the parameters are ignored and can be omitted.

drawerType

Type: String

Must be one of the following strings:

- ◆ "Assay Plate Drawer" [default]
- ◆ "Compound Plate Drawer"
- ◆ "Tips Drawer"

These values are not case sensitive.



Note: The *drawerType* parameter is required for the FlexStation 3 reader, since it has three drawers. If the parameter is omitted, then the assay plate drawer is closed.

Dispose

Void Dispose()

Purpose

Closes the final automation application dialog.

The termination dialog containing the “Terminate” button that appears on the SoftMax Pro 6 Software application after the script has completed execution.

Parameters

None

Example

The following example demonstrates how to use the Dispose function to close the automation dialog and exit the automation application. In The following example the code used to complete command execution and dispose the interface dialog box is included.

```
public void Main()
{
    AutomationObject.Initialize();
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.ErrorReport+= Error;
    AutomationObject.OpenDrawer();
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
```

ExportAs

```
Int32 ExportAs(String path, ExportAsFormat exportAsFormat)
```

Purpose

Exports data in the Columns, Plate, or XML format.

The **ExportAs** command overwrites any existing file without warning.

Parameters

path

Type: String

Fully qualified path name

exportAsFormat

Type: ExportAsFormat

TIME exports data in a single column of text for each well.

COLUMNS exports data in a single column of text for each well.

PLATE exports data in a text matrix corresponding to a microplate grid.

XML exports data in an XML file format.

Example

The following example demonstrates using the ExportAs function to output data in XML, Plate, and Columns formats. In this example, *TIME* is used in the ExportAsFormat parameter to export in Columns format.

```
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.SelectSection("Plate1");
    AutomationObject.SetReader("Offline", "SPECTRAMax M2");
    AutomationObject.SetSimulationMode(true);
    AutomationObject.StartRead();
    var commandID = AutomationObject.ExportAs("C:\\\\test.xml",
    SoftMaxPro.AutomationClient.SMPAutomationClient.ExportAsFormat.XML );
    Results.AppendResult("Command ID = " + commandID.ToString() );
    commandID = AutomationObject.ExportAs("C:\\\\Plate.txt",
    SoftMaxPro.AutomationClient.SMPAutomationClient.ExportAsFormat.PLATE );
    Results.AppendResult("Command ID = " + commandID.ToString() );
    commandID = AutomationObject.ExportAs("C:\\\\COLUMNS.txt",
    SoftMaxPro.AutomationClient.SMPAutomationClient.ExportAsFormat.TIME);
    Results.AppendResult("Command ID = " + commandID.ToString() );
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}

private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );

    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
```

ExportSectionAs

```
Int32 ExportSectionAs(String path, ExportAsFormat exportAsFormat, Bool append)
```

Purpose

Exports the currently selected section.

Parameters

path

Type: String

Fully qualified path name

exportAsFormat

Type: ExportAsFormat

TIME exports data in a single column of text for each well.

COLUMNS exports data in a single column of text for each well.

PLATE exports data in a text matrix corresponding to a microplate grid.

XML exports data in an XML file format.

append

Type: Boolean

Specify whether to append the data to an existing file.

If *append* is set to *false*, the command will overwrite any existing file without warning.

GetAutosaveState

```
Int32 GetAutosaveState()
```

Purpose

Returns the current autosave setting for the active document.

Parameters

None

Returns

This function returns the command id which can be used to retrieve the returned data from the CommandCompleted Event.

Data Returned Through the CommandCompleted Event

Return type: String

Returns the AutoSave state.

The return string is one of the following properties:

- `SMPAutomationClient.AutoSaveState.OFF` = AutoSave is turned off
- `SMPAutomationClient.AutoSaveState.ON` = AutoSave is turned on

GetDataCopy

```
Int32 GetDataCopy()
```

Purpose

Copies data to the client by way of an event.

The format of the copied data copied can vary based on the display settings for the current plate section (normally Raw Data), the read mode, the read type, and the number of wavelengths.

Parameters

None

Returns

This function returns the command id which can be used to retrieve the returned data from the CommandCompleted Event.

Data Returned Through the CommandCompleted Event

Return type: String

Returns the data for the selected plate section.

Example

The following example demonstrates how to copy data to the client by way of an event.

```
int mCopyID;
public void Main()
{
    string now = System.DateTime.Now.ToString();
    Results.AppendResult(now);
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
    AutomationObject.SetReader("Offline", "SPECTRAmax M2");
    AutomationObject.SetSimulationMode(true);
    AutomationObject.SelectSection("Plate1");
    int read1 = AutomationObject.StartRead();
    Results.AppendResult("Read ID = " + read1.ToString());
    read1 = AutomationObject.StartRead();
    Results.AppendResult("Read ID = " + read1.ToString());
    read1 = AutomationObject.StartRead();
    Results.AppendResult("Read ID = " + read1.ToString());
    mCopyID = AutomationObject.GetDataCopy();
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( mCopyID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.InstrumentStatusChanged -= InstrumentStatus;
        AutomationObject.Dispose();
    }
}
private void InstrumentStatus( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.InstrumentStatusEventArgs e)
{
```

```

    Results.AppendResult("Status changed to " + e.Status);
}

```

GetDrawerStatus

```

Int32 GetDrawerStatus()
Int32 GetDrawerStatus(String drawerType)

```

Purpose

Returns the state of the specified drawer on the instrument.

For most instruments, this command returns the state of the microplate drawer.

Parameters

The GetDrawerStatus parameters are recognized by the FlexStation 3 reader only. For all other instruments, the parameters are ignored and can be omitted.

drawerType

Type: String

Must be one of the following strings:

- ◆ “Assay Plate Drawer” [default]
- ◆ “Compound Plate Drawer”
- ◆ “Tips Drawer”

These values are not case sensitive.



Note: The *drawerType* parameter is required for the FlexStation 3 reader, since it has three drawers. If the parameter is omitted, then the status of the assay plate drawer is returned.

Returns

This function returns the command id which can be used to retrieve the returned data from the CommandCompleted Event.

Data Returned Through the CommandCompleted Event

Return type: String

Returns the status of the drawer.

The return string is one of following properties:

- ◆ `SMPAutomationClient.DrawerStatus.OPENED` = The drawer is open
- ◆ `SMPAutomationClient.DrawerStatus.CLOSED` = The drawer is closed

Example

The following example demonstrates how to determine out if each of the drawers on a FlexStation 3 reader is open or closed.

For all other instruments, no parameter is required to determine if the microplate drawer is open or closed.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    mStatusID = AutomationObject.GetDrawerStatus("Assay Plate Drawer");
    cStatusID = AutomationObject.GetDrawerStatus("Compound Plate Drawer");
    tStatusID = AutomationObject.GetDrawerStatus("Tips Drawer");
    Results.AppendResult("Status Command ID = " + mStatusID .ToString());
}
private void Error(object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void CommandCompleted(object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( mStatusID== e.QueueID )
    {
        Results.AppendResult( "Assay Plate Drawer Status: " +e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.ErrorReport -= Error;
        AutomationObject.Dispose();
    }

    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( cStatusID== e.QueueID )
    {
        Results.AppendResult( "Compound Plate Drawer Status: " +e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.ErrorReport -= Error;
        AutomationObject.Dispose();
    }

    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( tStatusID== e.QueueID )
    {
        Results.AppendResult( "Tips Drawer Status: " +e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.ErrorReport -= Error;
        AutomationObject.Dispose();
    }
}
```

GetFormulaResult

```
Int32 GetformulaResult(String results)
```

Purpose

Returns the result from the specified formula.

This command requires the name of the formula, name of either Notes or Group section, and the name of the Experiment. If the name of the Experiment is not specified, it will search from the first Experiment of the document.



Note: The returned value is the calculated value for the formula, not the formula string.

Parameters

results

Type: String

fully qualified formula name, including section and experiment identifiers

Example: *formulaName@sectionName@experimentName*

Returns

This function returns the command id which can be used to retrieve the returned data from the CommandCompleted Event.

Data Returned Through the CommandCompleted Event

Type: String

Calculated value for the formula

Example

The following example shows getting information from a Notes section. If the Notes section does not have a field titled Summary1, the command fails.

```

int mID;
int bAutomix;
System.Collections.Generic.Dictionary<int, string> commandResultFormatter = new
System.Collections.Generic.Dictionary<int, string>();
public void Main()
{
    string now = System.DateTime.Now.ToString();
    Results.AppendResult(now);
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.SelectSection("Notes1");
    mID = AutomationObject.GetFormulaResult("Summary1");
    commandResultFormatter.Add(mID, "Formula Result of Summary1 :- {0}");
    AutomationObject.SelectSection("Notes1");
    mID = AutomationObject.GetFormulaResult("Summary2");
    commandResultFormatter.Add(mID, "Formula Result of Summary2 :- {0}");
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}

private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString());
    if( !string.IsNullOrEmpty(e.StringResult) )
    {
        if(commandResultFormatter.ContainsKey(e.QueueID))
            Results.AppendResult(string.Format(commandResultFormatter[e.QueueID],
e.StringResult));
        else
            Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
}

```

GetGroupNameAssignments

Int32 GetGroupNameAssignments

Purpose

Returns the group name assignments of the currently selected plate section.

Parameters

None

Returns

The function returns the command id which can be used to retrieve the returned data from the CommandCompleted Event.

Data Returned Through the CommandCompleted Event

Return Type: String

Returns a tab-delimited list of group names assigned to wells for all the wells in the plate.
Wells without a group assignment are returned as blank.

GetInstrumentStatus

Int32 GetInstrumentStatus

Purpose

Returns the currently connected instrument status.

Parameters

None

Returns

The function returns the command id which can be used to retrieve the returned data from the CommandCompleted Event.

Data Returned Through the CommandCompleted Event

Return Type: String

For the possible values, see [InstrumentStatusChanged on page 88](#).

GetNumberPlateSections

```
Int32 GetNumberPlateSections()
```

Purpose

Returns the number of plate sections in the active experiment.

Parameters

None

Returns

This function returns the command id which can be used to retrieve the returned data from the CommandCompleted Event.

Data Returned Through the CommandCompleted Event

Return Type: Int32

Returns the number of plate sections in the active experiment.

Example

The following example shows how to determine the number of plate sections in the active experiment.

```
int NumPlateSectionsID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted+= CommandCompleted;
    NumPlateSectionsID = AutomationObject.GetNumberPlateSections();
    Results.AppendResult("Status Command ID = " + NumPlateSectionsID .ToString());
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( e.QueueID == NumPlateSectionsID )
    Results.AppendResult("Number of Plate Sections : " + e.IntResult.ToString() );
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
```

GetTemperature

`Int32 GetTemperature()`

Purpose

Returns the currently connected instrument incubator temperature.

Parameters

None

Returns

The function returns the command id which can be used to retrieve the returned data from the CommandCompleted Event.

Data Returned Through the CommandCompleted Event

Return Type: Double

Current instrument incubator temperature

GetVersion

`Int32 GetVersion()`

Purpose

Returns the version of the SoftMax Pro Automation Interface.

Parameters

None

Returns

This function returns the command id which can be used to retrieve the returned data from the CommandCompleted Event.

Data Returned Through the CommandCompleted Event

Return Type: String

Returns the version of the SoftMax Pro Software application.

The return string is the following property:

`SMPAutomationClient.Version.VERSION6300`

Example

The following example demonstrates getting the Version level of the SoftMax Pro Software application.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.ErrorReport += Error;
    mStatusID = AutomationObject.GetVersion();
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString());
    if( mStatusID== e.QueueID )
    {
        Results.AppendResult( "Version: " +e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.ErrorReport -= Error;
        AutomationObject.Dispose();
    }
}
```

ImportPlateData

```
Int32 ImportPlateData(string importType, params string[] importParameter)
```

Purpose

Imports data from a file into a Plate section in the current data file.

Parameters

importType

Type: String

Must be one of the following format strings:

- ◆ "Plate Format"
- ◆ "XML Format"

importParameter

Type: Params String

Fully qualified path name of the file containing the data to be imported. The file must match the format used for manual data imports.

Example

The following example imports microplate data in Plate format from a file named data_import.txt.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
    AutomationObject.SetReader("Offline", "SPECTRAmax M5");
    AutomationObject.SetSimulationMode(true);
    AutomationObject.SelectSection("Plate1");
    mStatusID = AutomationObject.ImportPlateData("Plate Format", "C:\\\\Program Files\\\\Molecular
Devices\\\\Import Templates\\\\data_import.txt");
    Results.AppendResult("Import Plate Data Status=" + mStatusID.ToString());
}

private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.InstrumentStatusChanged -= InstrumentStatus;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void InstrumentStatus( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.InstrumentStatusEventArgs e)
{
    Results.AppendResult("Status changed to " + e.Status);
}
```

ImportPlateTemplate

```
Int32 ImportPlateTemplate(string path)
```

Purpose

Imports a template from a file into the current plate section.

Parameters

path

Type: String

Fully qualified path name of the file containing the template to be imported. The file must match the format used by for manual template imports.

Example

The following example imports a plate template named platetemplate.txt.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
    AutomationObject.SetReader("Offline", "SPECTRAMax M5");
    AutomationObject.SetSimulationMode(true);
    AutomationObject.SelectSection("Plate1");
    mStatusID = AutomationObject.ImportPlateTemplate("C:\\\\Program Files\\\\Molecular Devices\\\\SoftMax Pro
7.0 Automation SDK\\\\Scripts\\\\PlateTemplate.txt");
    Results.AppendResult("Import Plate Template Status=" + mStatusID.ToString());
}

private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.InstrumentStatusChanged -= InstrumentStatus;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void InstrumentStatus( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.InstrumentStatusEventArgs e)
{
    Results.AppendResult("Status changed to " + e.Status);
}
```

Initialize

```
Boolean Initialize ()  
Boolean Initialize (String server)  
Boolean Initialize (String server, Int32 port)
```

Purpose

Initializes the Automation Interface.

Parameters

server

Type: String
the computer name or IP address of the server

port

Type: Int32
the port address of the server

Returns

Return Type: Boolean

- True = initialization was successful
- False = initialization failed

Examples

There are three ways to use the Initialize command.

- `bool Initialize()`
- `bool Initialize(String server)`
- `bool Initialize(String server, int port)`

bool Initialize()

With no parameters, the Initialize function uses the default host 'localhost' and default port of 9000.

```
bool mInitialize;
public void Main()
{
    mInitialize = AutomationObject.Initialize(); // Replace the target machine IP Address here
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
    Results.AppendResult("Inirialized : " + mInitialize.ToString());
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if(mInitialize)
    {
        Results.AppendResult(e.StringResult);
    }

    Results.AppendResult("Queue empty - disconnecting events");
    AutomationObject.ErrorReport -= Error;
    AutomationObject.CommandCompleted -= CommandCompleted;
    AutomationObject.InstrumentStatusChanged -= InstrumentStatus;
    AutomationObject.Dispose();
}

private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}

private void InstrumentStatus( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.InstrumentStatusEventArgs e)
{
    Results.AppendResult("Status changed to " + e.Status);
}
```

bool Initialize(String server)

The *server* parameter can be either the computer name or IP address and default port.

```
bool mInitialize;
public void Main()
{
    mInitialize = AutomationObject.Initialize("127.0.0.1"); // Replace the target machine IP Address here
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
    Results.AppendResult("Inirialized : " + mInitialize.ToString());
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if(mInitialize)
    {
        Results.AppendResult(e.StringResult);
    }

    Results.AppendResult("Queue empty - disconnecting events");
    AutomationObject.ErrorReport -= Error;
    AutomationObject.CommandCompleted -= CommandCompleted;
    AutomationObject.InstrumentStatusChanged -= InstrumentStatus;
    AutomationObject.Dispose();
}

private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}

private void InstrumentStatus( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.InstrumentStatusEventArgs e)
{
    Results.AppendResult("Status changed to " + e.Status);
}
```

bool Initialize(String server, int port)

You can use the computer name or IP and port.

```
public void Main()
{
    AutomationObject.Initialize("10.212.11.175", 9901)
    AutomationObject.CloseDocument();
    AutomationObject.NewDocument();
}
```

When the default port is not available for SoftMax Pro, then the you can add a registry key in HLM\SOFTWARE\Molecular Devices\Readers\SMP with key "Port" and a 32bit DWORD that contains the port address.

Logon

```
Int32 Logon(String userID, String password)
```

Purpose

Log a user onto the SoftMax Pro GxP Software.

To prevent the Log On dialog from appearing when the application starts, use /R in the program command line (for example, "SoftMaxProApp.exe /R"). See [Computer System Requirements on page 13](#).

Parameters

userID

Type: String

The user ID for the account

password

Type: String

The password for the account

Logoff

```
Int32 Logoff()
```

Purpose

Log a user off from the SoftMax Pro GxP Software.

Parameters

None.

NewDocument

`Int32 NewDocument()`

Purpose

Creates a new document.

This command also reads into the new document the values contained in the “Default Protocol.spr” protocol file.

Parameters

None

Example

The following example demonstrates creating a new document containing the values in the default protocol.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    mStatusID = AutomationObject.NewDocument();
}
private void CommandCompleted( object sender,
    SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
```

NewExperiment

Int32 NewExperiment()

Purpose

Creates a new Experiment section.

Parameters

None

Example

The following example shows creating a new experiment section.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    AutomationObject.NewDocument();
    AutomationObject.NewNotes();
    AutomationObject.NewPlate();
    AutomationObject.NewNotes();
    mStatusID = AutomationObject.NewExperiment();
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
```

NewNotes

```
Int32 NewNotes()
```

Purpose

Creates a new Notes section within the current Experiment.

Parameters

None

Example

The following example shows creating a notes section.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    AutomationObject.NewDocument();
    mStatusID = AutomationObject.NewNotes();
}
private void CommandCompleted( object sender,
    SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
```

NewPlate

Int32 NewPlate()

Purpose

Creates a new Plate section within the current Experiment.

Parameters

None

Example

The following example demonstrates the use of NewPlate command.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
    AutomationObject.NewDocument();
    AutomationObject.NewPlate();
    AutomationObject.NewNotes();
    mStatusID = AutomationObject.NewPlate();
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.InstrumentStatusChanged -= InstrumentStatus;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void InstrumentStatus( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.InstrumentStatusEventArgs e)
{
    Results.AppendResult("Status changed to " + e.Status);
}
```

OpenDrawer

```
OpenDrawer()
CloseDrawer(String drawerType)
OpenDrawer(Int32 xPosition, Int32 yPosition, Bool locked)
```

Purpose

Opens the specified drawer on the instrument.

For most instruments, this command opens the microplate drawer.



Note: For instruments with temperature control, the microplate drawer cannot be opened while the incubator is on. See [SetTemperature on page 83](#).

Parameters for FlexStation 3 Readers

The OpenDrawer *drawerType* parameter is recognized by the FlexStation 3 reader only.

For all other instruments, this parameter is ignored and can be omitted.

drawerType

Type: String

Must be one of the following strings:

- ◆ "Assay Plate Drawer" [default]
- ◆ "Compound Plate Drawer"
- ◆ "Tips Drawer"

These values are not case sensitive.



Note: The *drawerType* parameter is required for the FlexStation 3 reader, since it has three drawers. If the parameter is omitted, then the assay plate drawer is opened.

See also [Parameters for SpectraMax i3x, SpectraMax i3, SpectraMax Paradigm, and FilterMax Instruments on page 74](#).

Parameters for SpectraMax i3x, SpectraMax i3, SpectraMax Paradigm, and FilterMax Instruments

The OpenDrawer *xPosition*, *yPosition*, and *locked* parameters are recognized by the SpectraMax i3x, SpectraMax i3, SpectraMax Paradigm, and FilterMax Instruments only.

For all other instruments, these parameters are ignored and can be omitted.

xPosition

Type: Int32

This parameter defines the left-right offset for the position of the open microplate drawer.

- ◆ Range for the SpectraMax i3x and i3 Instruments: 0 to 2900
- ◆ Range for the SpectraMax Paradigm Instrument: 0 to 2950
- ◆ Range for the FilterMax Instruments: 0 to 2950

yPosition

Type: Int32

This parameter defines the front-rear offset for the position of the open microplate drawer.

- ◆ Range for the SpectraMax i3x and i3 Instruments: 5200 to 6900
- ◆ Range for the SpectraMax Paradigm Instrument: 5200 to 6900
- ◆ Range for the FilterMax Instruments: 5200 to 6900

locked

Type: Bool

When this parameter is true, the microplate is held in a fixed position to allow operations such as dispensing.

Example

The following example shows opening the microplate drawer.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.OpenDrawer();
    mStatusID = AutomationObject.GetDrawerStatus();
    Results.AppendResult("Status Command ID = " + mStatusID .ToString() );
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( mStatusID== e.QueueID )
    {
        Results.AppendResult( "Result: " +e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.CommandCompleted -= CommandCompleted;AutomationObject.Dispose();
    }
}
```

OpenFile

```
Int32 OpenFile(String pathname)
```

Purpose

Opens a protocol file or data file.

If the file is not found, the SoftMax Pro application does nothing.

Parameters

pathname

Type: String

Fully qualified path to protocol file or protocol name

Example

In the following example, the file that is to be opened must exist in the file system. If it does not exist, the command fails.

```
int mStatusID;
string mPath="C:\\Users\\All Users\\Application Data\\Molecular Devices\\SMP6\\Default
Protocols\\Basics\\Spectrum.spr";

public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    mStatusID = AutomationObject.OpenFile(mPath);
}

private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
```

Quit

`Int32 Quit()`

Purpose

Exits the SoftMax Pro Software application.

Parameters

None

Example

The following example demonstrates quitting the SoftMax Pro Software application.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    AutomationObject.NewDocument();
    AutomationObject.NewPlate();
    mStatusID = AutomationObject.Quit();
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
```

SaveAs

```
Int32 SaveAs(String pathname)
```

Purpose

Saves the current document as a data file or a protocol file.

Define the file type for the saved file by the file extension in the path statement. Use the *.sda extension for data files and *.spr for protocol files. If you are using the SoftMax Pro GxP Software, use the *.sdax extension for data files and *.sprx for protocol files.

If a file with the same name already exists, the SoftMax Pro Software automatically overwrites the file with no warning. The SoftMax Pro GxP Software does not allow overwriting existing files, and generates an error message if an overwrite is attempted.

Parameters

pathname

Type: String

Fully qualified path and name of file to be saved, including the file extension.

Example

The following example shows saving the file to an undefined “Temp” path. To define the path, the mPath string needs to be defined with an absolute or relative path that includes the name of the file and its file extension.

```
int mStatusID;
string mPath="C:\\temp\\mydata.sda";
public void Main() {
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    mStatusID = AutomationObject.SaveAs(mPath);
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
```

SelectNextPlateSection

```
Int32 SelectNextPlateSection()
```

Purpose

Selects the next plate section from the current Experiment.

Parameters

None

Example

The following example shows selecting the next plate after Plate1.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.CloseDocument();
    AutomationObject.NewDocument();
    AutomationObject.NewPlate();
    AutomationObject.SelectSection("Plate1");
    mStatusID= AutomationObject.SelectNextPlateSection();
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
```

SelectSection

```
Int32 SelectSection(String sectionName)
Int32 SelectSection(Int32 sectionNumber)
```

Purpose

Selects a section by name or by the order of the section within the experiment.

SelectSection using the System.String parameter can be useful for multi-plate protocols or for selecting Group tables for copying.

Parameters

sectionName

Type: String

The name of the section.

Either the name of a section within the currently selected experiment or a fully qualified section name, including section and experiment identifiers

Examples: sectionName or sectionName@experimentName

sectionNumber

Type: Int32

The order number of the section.

Example

The following example demonstrates using the SelectSection function using the System.String parameter.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    AutomationObject.NewExperiment();
    mStatusID = AutomationObject.SelectSection("Exp02");
    AutomationObject.NewExperiment();
    mStatusID = AutomationObject.SelectSection(2);
    AutomationObject.AppendTitle("_&^**%%$$##@@");
}
private void CommandCompleted( object sender,
    SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }

    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
```

SetReader

```
Int32 SetReader(String port, String instrument)
```

Purpose

Selects a reader type and sets the reader status.

Parameters



Note: The parameters are case-sensitive.

communication port

Type: String

The name of the communication port to look for the instrument (for example, *COM1*).

Offline sets the reader state to offline.

instrument

Type: String

340PC384

DTX 800

DTX 880

Emax

Emax Plus

FilterMax F3

FilterMax F5

Flexstation III

GEMINI EM

GEMINI XPS

PLUS190PC

PLUS384

SpectraMax i3

SpectraMax i3x

SPECTRAmax M2

SPECTRAmax M2e

SPECTRAmax M3

SPECTRAmax M4

SPECTRAmax M5

SPECTRAmax M5e

SpectraMax Paradigm

VersaMaxPLUS

Vmax

Example

See the [Example on page 82](#).

SetShake

```
Int32 SetShake(Boolean shakestate)
```

Purpose

Shakes the microplate.

When this command is set to *true*, starts shaking the microplate tray until the *Shake(false)* command is sent. By default, the shake will stop after 30 seconds in most instruments.



Note: The SetShake command is not supported for the SpectraMax i3x, SpectraMax i3, SpectraMax Paradigm, and FilterMax instruments.

Parameters

shakestate

Type: Boolean

true turns Shake on.

false turns Shake off.

Example

The following example demonstrates turning the shake function on and then off.

```
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    AutomationObject.Initialize();

    AutomationObject.SetReader("Offline", "SPECTRAmax M2");
    AutomationObject.SetSimulationMode(true);
    AutomationObject.SelectSection("Plate1");

    AutomationObject.CloseDrawer();
    AutomationObject.SetShake(true);

    AutomationObject.StartRead();

    AutomationObject.SetShake(false);
    AutomationObject.OpenDrawer();

}

private void CommandCompleted( object sender,
    SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
```

SetSimulationMode

```
Int32 SetSimulationMode(Boolean modestate)
```

Purpose

Sets SoftMax Pro software into simulation mode.

Parameters

`modestate`

Type: Boolean

true enables simulation mode.

false disables simulation mode.

Example

The following example demonstrates selecting a reader, enabling simulation mode and starting a read.

```
int mStatusID;
public void Main()
{
    string now = System.DateTime.Now.ToString();
    Results.AppendResult(now);
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
    AutomationObject.SetReader("Offline", "SPECTRAmax M5");
    mStatusID = AutomationObject.SetSimulationMode(true);
    Results.AppendResult("Simulation Mode Status = " +mStatusID.ToString());
    AutomationObject.SelectSection("Plate1");
    int read1 = AutomationObject.StartRead();
    Results.AppendResult("Read ID = " + read1.ToString());
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.InstrumentStatusChanged -= InstrumentStatus;
        AutomationObject.Dispose();
    }
}
private void InstrumentStatus( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.InstrumentStatusEventArgs e)
{
    Results.AppendResult("Status changed to " + e.Status);
}
```

SetTemperature

```
Int32 SetTemperature(Double temperature)
```

Purpose

Sets the current instrument incubator temperature.
Setting temperature to zero turns off the incubator.

Parameters

Temperature

Type: Double

The number of degrees centigrade

Example

The following example demonstrates setting the incubator temperature to 20°C.

```
int ID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    AutomationObject.SetReader("Offline", "SPECTRAmax M2");
    AutomationObject.SetSimulationMode(true);
    ID = AutomationObject.SetTemperature(20.0);
    Results.AppendResult("Set Temperature Status = " + ID.ToString());
}
private void CommandCompleted( object sender,
    SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if(ID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void InstrumentStatus( object sender,
    SoftMaxPro.AutomationClient.SMPAutomationClient.InstrumentStatusEventArgs e)
{
    Results.AppendResult("Status changed to " + e.Status);
}
```

SetTitle

`Int32 SetTitle()`

Purpose

Sets the title of the selected section.

Checks for uniqueness of the specified section name within the current experiment. If there is a conflict, no change is made.



Note: This command can alter the names of the sections within an experiment, but it cannot alter the name of the experiment.

Parameters

None

Example

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
    AutomationObject.SetReader("Offline", "SPECTRAMax M5e");
    AutomationObject.SetSimulationMode(true);
    AutomationObject.SelectSection("Plate1");
    mStatusID = AutomationObject.SetTitle("Plate-100");
    Results.AppendResult("Set Title Result : "+mStatusID.ToString());
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.InstrumentStatusChanged -= InstrumentStatus;
        AutomationObject.Dispose();
    }
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void InstrumentStatus( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.InstrumentStatusEventArgs e)
{
    Results.AppendResult("Status changed to " + e.Status);
}
```

SetUserName

```
Int32 SetUserName()
```

Purpose

Used with GxP systems to set the user name for a shielded protocol.

Parameters

User name

StartRead

```
Int32 StartRead()
```

Purpose

Reads the current Plate or CuvetteSet section.

If the current section is neither a Plate nor CuvetteSet section, the next Plate or CuvetteSet section is read.

Parameters

None

Example

The following example shows starting a read operation.

```
int mCopyID;
public void Main()
{
    string now = System.DateTime.Now.ToString();
    Results.AppendResult(now);
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
    AutomationObject.CloseDocument();
    AutomationObject.NewDocument();
    AutomationObject.SetReader("Offline", "SPECTRAmax M2");
    AutomationObject.SetSimulationMode(true);
    AutomationObject.SelectSection("Plate1");
    int read1 = AutomationObject.StartRead();
    Results.AppendResult("Read ID = " + read1.ToString());
    read1 = AutomationObject.StartRead();
    Results.AppendResult("Read ID = " + read1.ToString());
    read1 = AutomationObject.StartRead();
    Results.AppendResult("Read ID = " + read1.ToString());
    mCopyID = AutomationObject.GetDataCopy();
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString());
    if( mCopyID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.InstrumentStatusChanged -= InstrumentStatus;
        AutomationObject.Dispose();
    }
}
private void InstrumentStatus( object sender,
```

```
SoftMaxPro.AutomationClient.SMPAutomationClient.InstrumentStatusEventArgs e)
{
    Results.AppendResult("Status changed to " + e.Status);
}
```

StopRead

Int32 StopRead()

Purpose

Stops the current Plate or CuvetteSet section read.

This command is not queued.

Parameters

None

Events

The .NET server generates events which the client application can “listen” for by way of an event handler.

CommandCompleted

```
event EventHandler<object sender,
SoftMaxPro.Automation.SMPAutomationclient.CommandStatusEventArgs> CommandCompleted
```

Purpose

To generate a client callback when a command has completed.

Table 7-1: CommandStatusEventArgs Properties

Name	Type	Description
DoubleResult	Double	Returns a double result from a get command, such as GetTemperature()
IntResult	Int32	Provides the integer result from a get command, such as GetNumberPlateSections()
QueueEmpty	Boolean	True indicates the command queue on the server is empty False indicates the command queue on the server still contains commands
Queueld	Int32	The id of the completed command
StringResult	String	Provides the string result from a get command, such as GetDrawerStatus()

Example

See [Multiple Read and Copy Events Script on page 93](#).

InstrumentStatusChanged

```
event EventHandler(object sender,
SoftMaxPro.Automation.SMPAutomationclient.InstrumentStatusEventArgs)
InstrumentStatusChanged
```

Purpose

To generate a client callback when the status of the connected instrument changes.

Table 7-2: InstrumentStatusEventArgs Property

Name	Type	Description
Status	String	Provides the new status of the instrument. See the possible values listed below.

Returns

The following properties can be used for testing which state has been returned:

- SMPAutomationClient.InstrumentStatus.IDLE
- SMPAutomationClient.InstrumentStatus.INITIALIZING
- SMPAutomationClient.InstrumentStatus.BUSY
- SMPAutomationClient.InstrumentStatus.STOPPING
- SMPAutomationClient.InstrumentStatus.ERROR
- SMPAutomationClient.InstrumentStatus.TIMEOUT
- SMPAutomationClient.InstrumentStatus.OFFLINE

Example

See [Multiple Read and Copy Events Script on page 93](#).

ErrorReport

```
event EventHandler<object sender,
SoftMaxPro.Automation.SMPAutomationclient.ErrorEventArgs>
ErrorReport
```

Purpose

To generate a client callback when an error condition occurs.

Table 7-3: ErrorEventArgs Properties

Name	Type	Description
Error	String	A description of the error
QueueId	Int32	The id of the command with the error

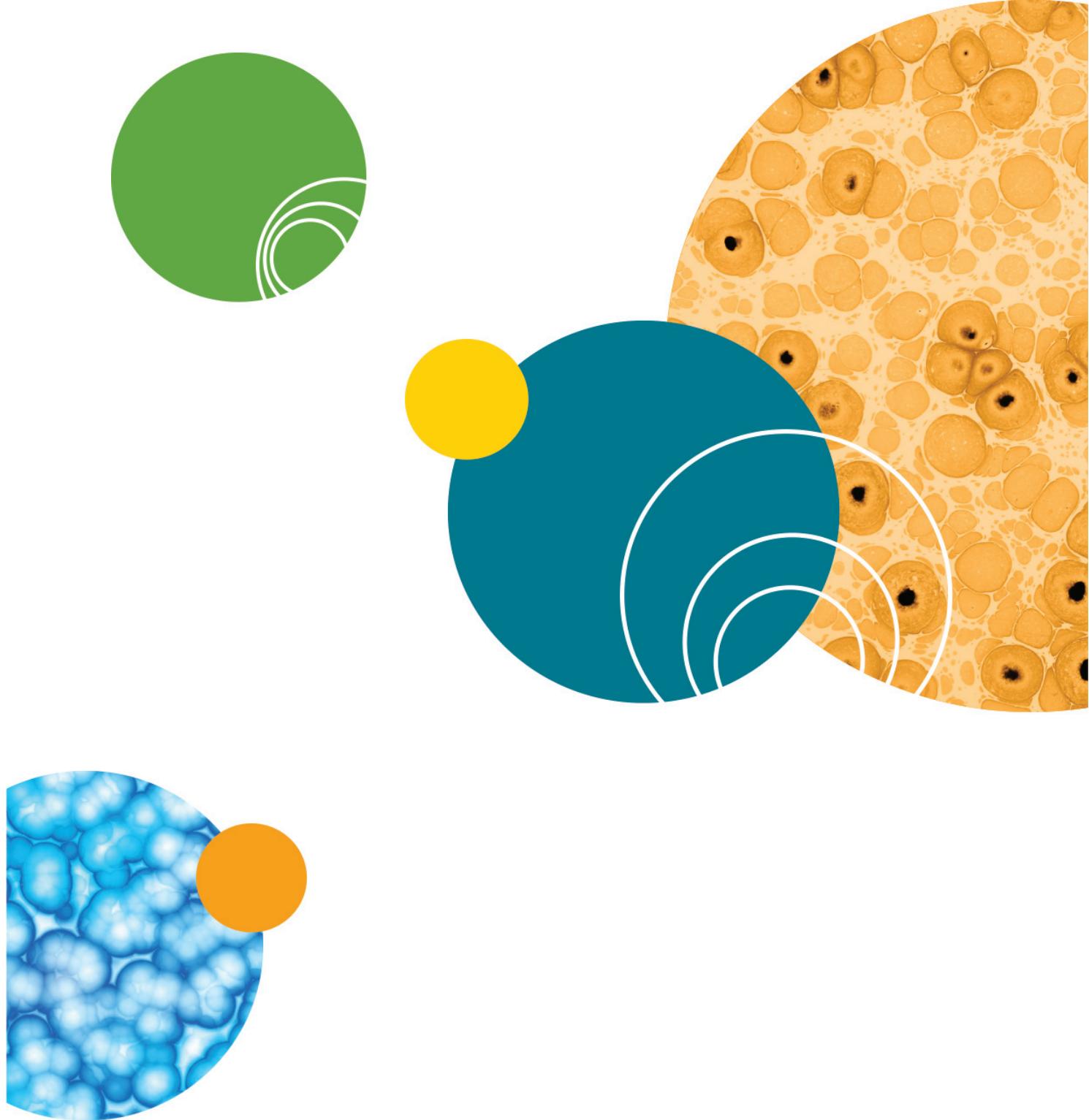
Error Event Related Processing

When an Error event is generated, it is always followed by a CommandComplete event for the command that caused the error.

All commands waiting in the automation server command queue are deleted so that the automation client can control exactly what commands are executed when an error has been reported.

Example

```
public void Main()
{
    AutomationObject.Initialize();
    AutomationObject.ErrorReport += Error;
    AutomationObject.CloseDrawer();
    AutomationObject.SetShake(true);
    AutomationObject.OpenDrawer();
}
private void Error( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " +
e.QueueID.ToString() + " - " + e.Error);
}
```



Examples

This chapter contains some sample scripts.

Append Title Script

```
int mAppenTitleID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.SelectSection("Plate01");
    mAppenTitleID = AutomationObject.AppendTitle("- YeOld Append");
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}

private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString());
    if( mAppenTitleID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
```

Get Commands Script

```

public int AutosaveStateID;
public int NumPlateSectionsID;
public int DrawerStatusID;
public void Main()
{
    string now = System.DateTime.Now.ToString();
    Results.AppendResult(now);
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.CloseDrawer();
    AutosaveStateID = AutomationObject.GetAutosaveState();
    Results.AppendResult("AutosaveStateID= " + AutosaveStateID.ToString());
    NumPlateSectionsID = AutomationObject.GetNumberPlateSections();
    Results.AppendResult("NumPlateSectionsID= " + NumPlateSectionsID.ToString());

    DrawerStatusID = AutomationObject.GetDrawerStatus();
    Results.AppendResult("NumPlateSectionsID= " + DrawerStatusID.ToString());

}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( e.QueueID == AutosaveStateID)
    Results.AppendResult("AutosaveState: " + e.IntResult.ToString() );
    if( e.QueueID == NumPlateSectionsID )
    Results.AppendResult("NumPlateSections: " + e.IntResult.ToString() );
    if( e.QueueID == DrawerStatusID )
    Results.AppendResult("DrawerStatus: " + e.IntResult.ToString() );
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}
}

```

Multiple Read and Copy Events Script

```

int mCopyID;
int read1;
public void Main()
{
    string now = System.DateTime.Now.ToString();
    Results.AppendResult(now);
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
    AutomationObject.SetReader("Offline", "SPECTRAmax M2");
    AutomationObject.SetSimulationMode(true);
    AutomationObject.SelectSection("Plate1");
    read1 = AutomationObject.StartRead();
    Results.AppendResult("Read ID = " + read1.ToString());
    read1 = AutomationObject.StartRead();
    Results.AppendResult("Read ID = " + read1.ToString());
    read1 = AutomationObject.StartRead();
    Results.AppendResult("Read ID = " + read1.ToString());
    mCopyID = AutomationObject.GetDataCopy();
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString());
    if( read1 == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.InstrumentStatusChanged -= InstrumentStatus;
        AutomationObject.Dispose();
    }
}
}

```

Multiple Read With ID Script

```

public void Main()
{
    string now = System.DateTime.Now.ToString();
    Results.AppendResult(now);
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.SetReader("Offline", "SPECTRAmax M2");
    AutomationObject.SetSimulationMode(true);
    AutomationObject.SelectSection("Plate1");
    int read1 = AutomationObject.StartRead();
    Results.AppendResult("Read 1 ID = " + read1.ToString());
    int read2 = AutomationObject.StartRead();
    Results.AppendResult("Read 2 ID = " + read2.ToString());
    int read3 = AutomationObject.StartRead();
    Results.AppendResult("Read 3 ID = " + read3.ToString());
}

private void PrintCurrentTime()
{
    string now = System.DateTime.Now.ToString();
    Results.AppendResult("Read completed at: " + now);
}

private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    PrintCurrentTime();
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}

```

ExportAs Script

```

int mStatusID ;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.SelectSection("Plate1");
    AutomationObject.SetReader("Offline", "SPECTRAmax M2");
    AutomationObject.SetSimulationMode(true);
    AutomationObject.StartRead();
    mStatusID = AutomationObject.ExportAs("C:\\Automation_Test\\Results\\test.xml",
SoftMaxPro.AutomationClient.SMPAutomationClient.ExportAsFormat.XML );
    Results.AppendResult("Status Command ID = " + mStatusID .ToString() );
    mStatusID = AutomationObject.ExportAs("C:\\Automation_Test\\Results\\Plate.txt",
SoftMaxPro.AutomationClient.SMPAutomationClient.ExportAsFormat.PLATE );
    Results.AppendResult("Status Command ID = " + mStatusID .ToString() );
    mStatusID = AutomationObject.ExportAs("C:\\Automation_Test\\Results\\COLUMNS.txt",
SoftMaxPro.AutomationClient.SMPAutomationClient.ExportAsFormat.TIME);
    Results.AppendResult("Status Command ID = " + mStatusID .ToString() );
}
private void Error( object sender, SoftMaxPro.AutomationClient.SMPAutomationClient.ErrorEventArgs e)
{
    Results.AppendResult("Error: Command ID = " + e.QueueID.ToString() + " - " + e.Error);
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArgs e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() );
    if( mStatusID == e.QueueID )
    {
        Results.AppendResult(e.StringResult);
    }
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - disconnecting events");
        AutomationObject.ErrorReport -= Error;
        AutomationObject.CommandCompleted -= CommandCompleted;
        AutomationObject.Dispose();
    }
}

```

Contact Us

Phone: [800.635.5577](tel:8006355577)
Web: www.moleculardevices.com
Email: info@moldev.com
Check our website for a current listing
of worldwide distributors

Regional Offices

USA and Canada	+1.800.635.5577	China (Beijing)	+86.10.6410.8669	Japan (Osaka)	+81.6.7174.8831	Brazil
United Kingdom	+44.118.944.8000	China (Shanghai)	+86.21.3372.1088	Japan (Tokyo)	+81.3.6362.5260	+55.11.3616.6607
Europe*	00800.665.32860	Hong Kong	+852.2248.6000	South Korea	+82.2.3471.9534	

*Austria, Belgium, Denmark, Finland, France, Germany, Ireland, Netherlands, Spain, Sweden, and Switzerland

The trademarks used herein are the property of Molecular Devices, LLC or their respective owners.
Specifications subject to change without notice. Patents: www.moleculardevices.com/productpatents
FOR RESEARCH USE ONLY. NOT FOR USE IN DIAGNOSTIC PROCEDURES

©2016 Molecular Devices, LLC.
All rights reserved.
5049094 A

