SoftMax Pro Data Acquisition and Analysis Software

API Reference Guide

Software Version 7.2

5049094 E
August 2024

MOLECULAR DEVICES

Product manufactured by Molecular Devices, LLC.
3860 N. First Street, San Jose, California, 95134, United States of America.
Molecular Devices, LLC is ISO 9001 registered.

Contents

Chapter 1: Introduction

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

Leading automation vendors can integrate the SoftMax Pro Software and Molecular Devices instruments with their robotics systems. See the Molecular Devices web site for a list of approved robotics partners.

Intended Audience

This guide is for people who want to build an automation application that interacts with the SoftMax Pro Software version 7.1.x or later using the automation interface to control the operation of a Molecular Devices instrument.

To use this guide, you 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 tools that enable you to create and customize an assembly.

Computer System Requirements

The computer system requirements are listed in the *SoftMax Pro Data Acquisition and Analysis Software Installation Guides*.

- When you run both automation and the SoftMax Pro Software on the same computer, use at least the recommended system configuration.
- Molecular Devices recommends that the documents you use in the automation workflow have a small number of sections. You should close documents when they are no longer needed.
- An experiment with many plates of data can adversely affect SoftMax Pro Software efficiency. If you cannot increase the computer memory, you should use only one plate per experiment and do not use the computer for other purposes while the SoftMax Pro Software runs.

The automation API uses port 9000 by default. You can change this port in the software if there are conflicts. See *SoftMax Pro Data Acquisition and Analysis Software User Guide* or the application help.

Installing the Automation SDK

The SoftMax Pro Software installation includes the automation SDK and can be found in the file system where you install the SoftMax Pro Software.

The following is the default installation path for the automation SDK:

C:\Program Files\Molecular Devices\SoftMax Pro <n.n> Automation SDK

Automation With the SoftMax Pro Software - GxP Edition

For the SoftMax Pro Software - GxP edition version 7.1.1 and higher, you need to consider the document workflow. See the *SoftMax Pro Data Acquisition and Analysis Software User Guide*.

Manual steps:

- Import a protocol from the Protocol Library to the database or create a new document.
- Save it as protocol.
- Release the protocol after you log in as a user with the equivalent of the Lab Technician Role permissions.

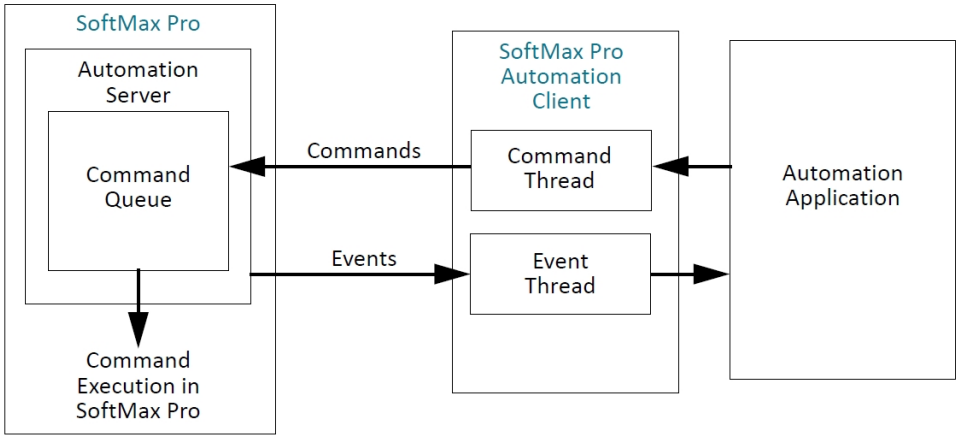
Automation:

- Use the automation to load the protocol. The software creates a new data document based on the opened protocol.
- Save the document as a data document.
- Run the experiment.

Chapter 2: Automation Interface

This SoftMax Pro Software automation interface is a Microsoft .NET Framework assembly that you can use with any .NET Language (C#, Visual Basic, etc.).

The following illustrates the automation interface API components.



The .NET assembly SoftMax Pro Automation Client supplies the automation interface that controls the SoftMax Pro Software. This automation interface supports commands that trigger the event handlers to notify the automation interface application of the status of the SoftMax Pro Software, execute queued commands, and display errors. The applications you use to control the SoftMax Pro Software via the automation interface must connect through the Automation Client.

The Automation Client connects to the Automation Server that is included in the SoftMax Pro Software. The Automation Server maintains a queue of commands that are sent through the Automation Client and executes these commands sequentially.

Commands are sent to the SoftMax Pro Software by calling functions in the automation interface (the client). Commands in the automation interface are asynchronous so that the functions return control to the application immediately.

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:

- The client sends commands and the server receives the commands on one thread.
- The server publishes events and passes the events to the client on a second thread.

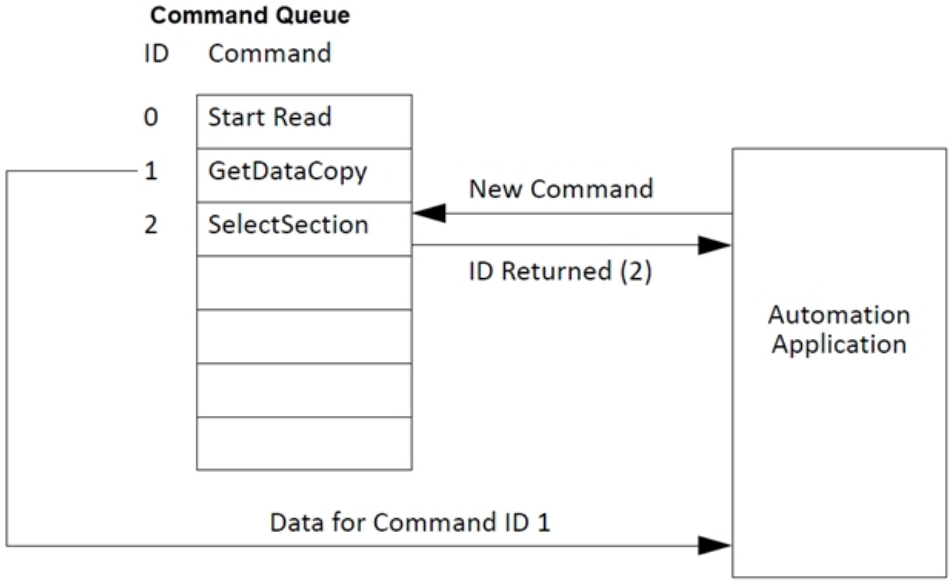
Automation Interface Overview

You use the automation interface to control the SoftMax Pro Software. To use the automation interface to connect to the SoftMax Pro Software, call the Initialize() function. After the automation interface initializes, the SoftMax Pro Software enters automation mode. Automation mode disables the SoftMax Pro Software user interface to prevent user input.

To end automation mode from the automation interface, call Dispose() or click Terminate from the SoftMax Pro Software user interface.

After initialization completes, hook up the events for the application to monitor. You should disconnect events when the CommandCompleted event indicates that the command queue is empty.

Commands in the automation interface return a command ID and the command IDs are maintained in a command queue. Each command ID is stored and then 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 returns when the command is sent.



You should consider the following published events when you design and write an automation application. See Events on page 115.

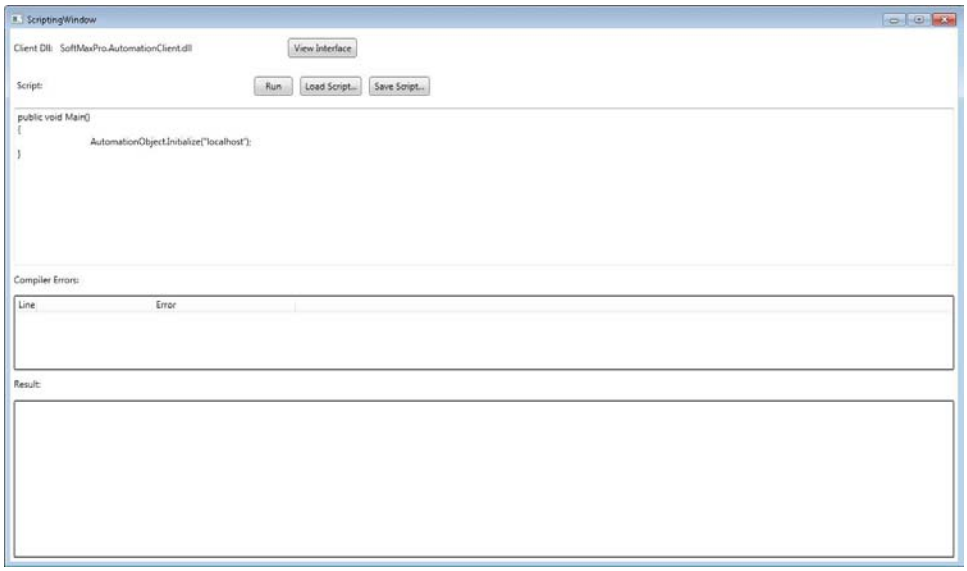
- **ErrorReport Event** - When the automation server detects an error, an Error event is published, followed by a CommandComplete event. All commands in the automation server command queue are flushed.
- **InstrumentStatus Event** - When the instrument changes state, for example from Idle to Busy, an InstrumentStatus event is published.

To create an environment for C# and Visual Basic .NET clients, see Creating Projects With Visual Studio on page 12.

SoftMax Pro Automation Sample Application

The SoftMax Pro Automation Sample Application allows you to prototype and test code. The SoftMax Pro Automation Sample Application binds the automation interface to a Microsoft scripting control.

Note: The SoftMax Pro Automation Sample Application is not intended for use in an actual laboratory or research environment.



Execute the SoftMax Pro Automation Sample Application shortcut from the Windows Start menu to display the Scripting Window dialog.

The Scripting Window dialog allows you view the interface commands and to load, run, and save scripts. Errors display in the Compiler Errors area and output information for the script display in the Result area.

To prototype sample code, the SoftMax Pro Automation Sample Application scripting engine creates 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 monitor. If you transfer the code to a stand-alone application, you must add the code to create the automation object instance.

The SoftMax Pro Automation Sample Application includes a Result object that allows you to append text data. You can use this to monitor the execution of commands and the progress in the command queue.

After you prototype the programming code in the Script area in the Scripting Window dialog, you can use the code to create a stand-alone robotic/automation application that directly controls the SoftMax Pro Software.

Creating Projects With Visual Studio

The automation interface allows you to use your preferred programming language to develop custom applications and create new projects with Visual Studio.

1. Select the relevant template (for example, Windows Form Application, WPF Application, or Console Application).
2. Add a reference to each of the following assemblies:
   - SoftMax Pro AutomationClient.dll
   - SoftMaxPro.AutomationInterface.dll
   - SoftMaxPro.AutomationExtensions.dll
     The default installation places these assembles in the following path:
     C:\Program Files\Molecular Devices\SoftMax Pro n.n Automation SDK

Note: The SoftMax Pro Automation Sample Application does not display the assembly because the assembly loads dynamically at run time and is injected into the scripting interface. You must reference the assembly for normal application development.

Sample Source Code and Applications

Sample automation scripts and the command example scripts that this guide describe are included as a part of the SoftMax Pro Automation API installation package. The default software installation places the samples scripts in the following location:

C:\Program Files\Molecular Devices\SoftMax Pro n.n Automation SDK\Scripts

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

Example

This example of a basic code structure includes a call to Initialize(), a subscription to an event, and a Dispose() call to terminate the connection. These examples assume that you use the SoftMax Pro Automation Sample application.

```
// 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.CommandStatusEventArg 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();
    }
}
```

This generates the following output:

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

C#

This 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.CommandStatusEventArg 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();
    }
}
```

Visual Basic .NET

This 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.CommandStatusEventArg)
        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 you prefix with Get..., like GetVersion() or GetNumberPlateSections(), are queued like other commands. The command returns information by way of the CommandStatusEventArgs class in the CommandCompleted event.

The CommandStatusEventArg class

```
public class CommandStatusEventArg : EventArgs

{
    public CommandStatusEventArg( 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;

}
```

Functions That Return Information

This 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.CommandStatusEventArg 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();
    }

}
```

Developing SoftMax Pro Excel Visual Basic Macros

Visual Basic developers can write Excel Visual Basic macros that access the SoftMax Pro Software through the Automation API. SoftMax Pro Excel macro development requires the installation of the Excel Add-In. See Installing Excel Add-Ins on page 23.

Note: Excel Visual Basic macros and Visual Basic .NET are very different software development environments. The development process in this section is significantly different from the Visual Basic .NET example. See Automation Interface Overview on page 10.

Accessing the SoftMax Pro Software Automation API

Use a tool called Excel-DNA to access the Excel Visual Basic SoftMax Pro Software automation API. To learn more about Excel-DNA, go to http://exceldna.codeplex.com/. Excel-DNA allows you to create a wrapper around the SoftMax Pro Software automation interface which results 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 invoke **SMPGetInstrumentStatus**.

Invocation Mechanism

Instead of invoking methods directly as in the Sample Source Code and Applications on page 12 for the **AutomationObject.OpenDrawer()**, commands are passed as arguments to an invocation of **Application.Run**.

For example, **Application.Run "SMPCloseDrawer"**. See SoftMax Pro Excel VBA Example on page 20.

Synchronous Method Calls

Instead of checking the contents of an event for the result of an automation command, the result returns directly to the Excel Visual Basic statement that issues the command.

For example, **drawerStatus = Application.Run("SMPGetDrawerStatus")**. See SoftMax Pro Excel VBA Example on page 20.

Error Detection

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

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

See SoftMax Pro Excel VBA Example on page 20.

Instrument Status

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

Command Names

To work within the constraints of the technology in this solution, some Excel Macro Commands do not conform exactly to the SMP prefix mechanism. See Invocation Mechanism on page 19.

- 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:
  - SMPInitialize()
  - SMPInitializeByServer(string server)
  - SMPInitializeByServerAndPort(string server, int port)

SoftMax Pro Excel VBA Example

A spreadsheet named VbaMacroExample.xlsm is installed in the SoftMax Pro n.n Automation SDK\ExcelAddIn folder. You should read the notes within the spreadsheet before you execute the Visual Basic macro in the file.

The following is an example of the code for a 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("SMPInitializeByServer", "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
```

Chapter 3: SoftMax Pro Excel Workflows

The SoftMax Pro Excel workflows augment the handling of Plate Format Data by the SoftMax Pro Software with Excel-based handling of List Format Data. SoftMax Pro Excel workflows allow you to run discontinuous kinetic reads, multiplexed reads, kinetic well scan reads, and temperature-triggered reads.

The SoftMax Pro Automation SDK is the underlying mechanism that the SoftMax Pro Excel workflows use to access the SoftMax Pro Software functionality. The SoftMax Pro Excel workflow feature is a standard Excel Add-In that you can use through the application of the Excel Charting & Trend Line features. The Add-In has been tested with Windows 10 to run automated workflows.

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)

When you install the Add-In and enable macros, you can run or edit the provided workflows or create your own. You can write custom formulas if you have knowledge of Excel Visual Basic. See Custom Excel Formulas on page 47.

You can also use the following Microsoft Add-Ins.

- Analysis ToolPak
- Solver

Installing Excel Add-Ins

To install Excel Add-Ins:

1. Start Microsoft Excel.
2. Select the **Developer** tab and click **Excel Add-Ins**.
   If the Developer tab does not display, select the **File** tab, select **Options**, select **Customize Ribbon**, and then select the **Developer** check box.
3. In the Add-Ins dialog, click **Browse**.
4. In the Browse dialog, navigate to the folder where the SoftMax Pro Software is installed:
   C:\Program Files (x86)\Molecular Devices\SoftMax Pro n.n Automation SDK\ExcelAddIn.
5. Click the **SoftMaxPro6x64.xll** file.
6. Click **OK**.
7. In the Add-Ins dialog, select the **SoftMaxPro6x64** check box.
8. Click **OK**.

Provided Excel Workflows

After you install the Add-In, you can run or edit the provided Excel workflows that the SoftMax Pro Software installs in the same folder as the Add-In files.
C:\Program Files (x86)\Molecular Devices\SoftMax Pro n.n Automation SDK\ExcelAddIn\Examples

The Excel workflow files have the .xlsm file extension.

Note: Save a copy of the workflow before you run or edit a provided Excel workflow.

Excel Workflow Layouts

Each Excel workflow must be contained in one workbook. This includes the provided workflows and the workflows you customize.

The provided workflows are color coded for readability. The color coding does not affect the execution of workflow. See Workflow Statement Types on page 28.

Provided Workflow Statement Type Color Codes

Workflow Statement Type	Color Code
Initialization Statements	Orange
Automation Command Statements	Yellow
Worksheet Formatting Statements	Green
Workflow Flow Control Statements	Gray
Free-Format Statement Arguments	Pink

SMPWorkflow Worksheet Features

Each statement in 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. See Running Excel Workflows on page 25.

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

This document contains the list of valid instrument names to prevent mismatches in the format of the name of the instrument. You should copy and paste name into the workflow. See SetReader on page 101.

Excel Visual Basic Components

A Visual Basic module named SMPWorkflow contains logic to initiate the invocation of workflow statements, worksheet formatting logic, and custom Excel formulas. See Custom Excel Formulas on page 47.

The following Visual Basic messages can display:

- The workflow is being validated before execution.
- The workflow is being executed.
- The workflow is paused, with details.
- An error was detected.

Running Excel Workflows

To run a workflow for the first time, you need an open SoftMax Pro Excel workflow that contains no saved data in the worksheet. You must install the Excel Add-In with macros enabled. The SoftMax Pro Software must be running.

Workflows enable you to perform the following operations:

- Executing Continuous Workflows (see below)
- Continuing Discontinuous Workflows, see page 26
- Canceling Workflows, see page 27

In most workflows, you connect the SoftMax Pro Software to the instrument before you start the workflow. You can have the workflow connect the SoftMax Pro Software to the instrument without user intervention. See Instrument Connectivity on page 49.

Executing Continuous Workflows

To run the workflow once and save the data in the Excel worksheet:

1. Physically connect the computer to the instrument and power on the instrument.
2. Start the SoftMax Pro Software.
3. Open the Excel workflow to run.
4. In the SMPWorkflow worksheet, click **Execute**.
5. After the workflow completes, save the worksheet.

Continuing Discontinuous Workflows

After you execute an Excel workflow to acquire SoftMax Pro Software data, the workflow writes the data to Excel in List Format style and does not overwrite existing data in the worksheets. This allows you to do kinetic reads with variable intervals, such as in a discontinuous workflow. The List Format saves data in separate rows after each read.

1. Physically connect the computer to the instrument and power on the instrument.
2. Start the SoftMax Pro Software.
3. Connect the SoftMax Pro Software to the instrument.
4. Open the Excel workflow to run.
5. In the SMPWorkflow worksheet, click **Execute**.
6. After the workflow completes, save the workbook. The data worksheet has one row of data.
7. Close the workbook.
8. After the correct amount of time passes, open the Excel workflow again.
9. Physically connect the computer to the instrument and power on the instrument.
10. Start the SoftMax Pro Software.
11. Connect the SoftMax Pro Software to the instrument.
12. In the SMPWorkflow worksheet, click **Execute**.
13. After the workflow completes save the workbook. Each worksheet now has an additional row of data.
14. Repeat these step for each additional read.
15. Close the workbook.

Example Results

When you execute a read once every hour, after the first read the worksheet has one row of data similar to the following:

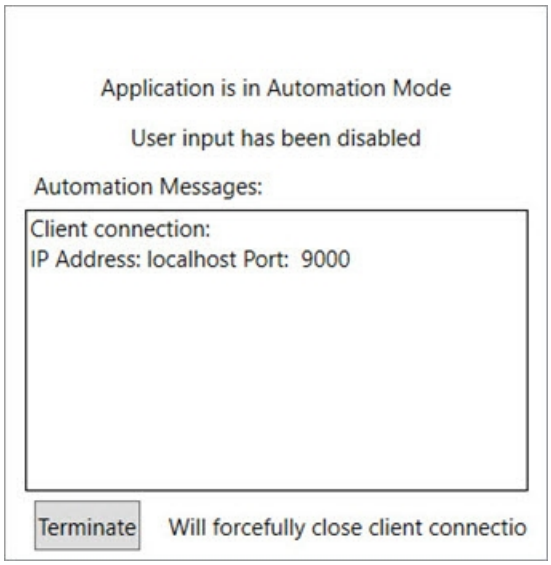
Date/Time	Elapsed Time	Temperature	A1	A2
1/22/2021 18:57	0	31.5	0.30	0.21

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

Date/Time	Elapsed Time	Temperature	A1	A2
1/22/2021 18:57	0	31.5	0.30	0.21
1/22/2021 19:57	3600	31.5	0.31	0.25

Canceling Workflows

When you run an Excel workflow, a message displays in front of the SoftMax Pro Software window, to indicate that the software is in automation mode.



Click **Terminate** to stop the automation mode and regain control of the SoftMax Pro Software. Then close all open dialogs in the Excel workflow.

Creating Excel Workflows

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

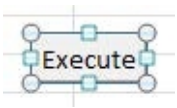
Editing Excel Workflows

When you edit 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 executes.
- The first blank row indicates the end of the workflow.

Moving the Execute Button

When you create a long workflow, you can move the Execute button.



Right-click the button and drag it to the new location.

Workflow Statement Types

To help understand how to read and write workflows, consider each workflow statement to be one of the following statement types.

- **Initialization Statements** - Initialization statements augment automation commands when you set up the workflow execution environment. See Initialization Statements on page 29.
- **Automation Command Statements** - Automation command statements are available as part of the SoftMax Pro Software Automation API. See Automation Command Statements on page 30.
- **Worksheet Formatting Statements** - Worksheet formatting statements retrieve the data from the SoftMax Pro Software, define the format of the data to write to the worksheet, and write it to the worksheet. See Worksheet Formatting Statements on page 30.
- **Workflow Control Statements** - Workflow control statements allow the workflow to repeat a section of statements or pause until a condition is satisfied. See Workflow Flow Control Statements on page 43.

Initialization Statements

Initialization statements augment automation commands when you set up the workflow execution environment. In the provided example workbooks, initialization statements are color coded orange.

PlateSize Statement

The PlateSize statements define the size of the plate you use in the instrument.

Arguments

PlateSize has one argument:

- The number of wells in the plate.
  The following plate sizes are supported.
  - 24-well plates
  - 48-well plates
  - 96-well plates
  - 384-well plates

Defaults

If you do not code the PlateSize Statement, the plate size defaults to 96 wells.

PlateSize Statement Examples

Automation Command Statements

Automation command statements are included with the SoftMax Pro Software Automation API. In the provided example workbooks, automation command statements are color coded yellow. See SoftMax Pro Automation Commands on page 51.

The first column contains the command name and the columns to the right contain the command's parameter values. The following displays examples of automation command statements:

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

Most automation command statements perform an action. Some command statements retrieve data from the SoftMax Pro Software to make data available to be written into worksheets and some command statements control the flow of a workflow.

- **Worksheet Formatting Statements** - Retrieve the data from the SoftMax Pro Software, define the format of the data to write to worksheets, and write the data to the Excel worksheet. See below.
- **Workflow Flow Control Statements** - Allows the workflow to repeat a section of statements or pause until a condition is satisfied. See Workflow Flow Control Statements on page 43.

Worksheet Formatting Statements

Worksheet formatting statements retrieve the data from the SoftMax Pro Software, define the format of the data to write to the worksheet, and write it to the worksheet. In the provided example workbooks, worksheet formatting statements are color coded green.

There are three categories of worksheet formatting statements:

- **Creating and Selecting Worksheets** - Statements that create worksheets in which to write data. See Worksheet Statement on page 31.
- **Defining Write Formats** - Statements that define the format in which to write data. See Defining Write Formats on page 32.
- **Writing to Worksheets** - Statements that define how to write fixed values to a worksheet. See Writing To Worksheets on page 41.

Worksheet Statement

The Worksheet statement creates a new worksheet and then selects a worksheet. As the workflow adds each worksheet, the new worksheet is automatically selected. Other worksheet formatting statements use the selected worksheet.

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 select or add.

Defaults

If you do not give the Name argument, a new worksheet is added with the name Sheet<n> where <n> is the lowest number available 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

Before the application writes to a worksheet, you must have the workflow define the format in which to write data. There are two statements that enable you to define the write format:

- CellFormat Statement (see below)
- RowFormat Statement, see page 38

CellFormat Statement

The CellFormat statement defines the format in which to write data into one Excel worksheet cell. Use the CellFormat statement in conjunction with the following two workflow statements:

- FormatWorksheet Statement, see page 41
- ProcessCommandResult Statement, see page 42

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 to write into in the following format: worksheet-name:worksheet-column or worksheet-column
  If you specify only the worksheet column, the worksheet row to use 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 write to a specific row. If omitted, the output writes to the next free row in the target column. This ensures headings are written out only once in discontinuous reads.
- Cell Content (sometimes not required, see defaults below) - Defines what writes into the worksheet column. When you write an Excel formula (e.g. '=Sum(A1:A5)' you can use the following keywords and this is substituted immediately prior to the formula being written to a worksheet,
  - #Column - The workflow keeps track of where it writes plate data into a worksheet. You should use this keyword in conjunction with the FormatWorksheet WriteRow statement to apply operations to whole columns of data.
  - #RowCount - This provides the number of rows that contain data values. As per #Column, you should use this keyword in conjunction with the FormatWorksheet WriteRow statement.
  - #Worksheet - This references the selected worksheet. You can use this keyword in conjunction with both the FormatWorksheet WriteRow and FormatWorksheet WriteCell statements.

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

Defaults

If the you omit worksheet name from the Target Worksheet Column argument, the currently selected worksheet is used as the target for the write. See Worksheet Statement on page 31.

You can omit cell content if the value to be written to the cell is the result from a previously issued automation command. See ProcessCommandResult Statement on page 42.

CellFormat Statement Example 1

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

CellFormat	MyTemperatureHeading	D	1	Temperature

. . .

Worksheet	Add	
FormatWorksheet	WriteCell	MyTemperatureHeading

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

	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

This example describes how to perform four Well Scan reads and write one result into each of Sheets 1 through 4. This CellFormat statement puts a reference in the Summary worksheet to the date and time recorded in each of Sheets 1 through 4.

```
CellFormat      MyDate      Summary:B      =#Worksheet!B2
...
Loop            4
                Worksheet   Add
...
                StartRead
                GetDataCopy
...
                ProcessCommandResult   WriteRow    MyWellScanData
                FormatWorksheet        WriteCell   MyDate
EndLoop
...
```

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 that writes into column B where #Worksheet is the name of the active worksheet, for example, Sheet1.

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

	A	B	C	D	E	F
1		Date/Time	Elapsed Time	Temperature	A1	A2
2		2/21/2013 14:52	0	35	1171.454556	1171.4
3		2/21/2013 14:53	33	35	1171.454556	1171.4
4		2/21/2013 14:53	67	35	1171.454556	1171.4
5		2/21/2013 14:55	154	35	1171.454556	1171.4
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						

SMPWorkflow | Summary | Sheet1 | Sheet2 | Sheet3 | Sheet4

CellFormat Statement Example 3

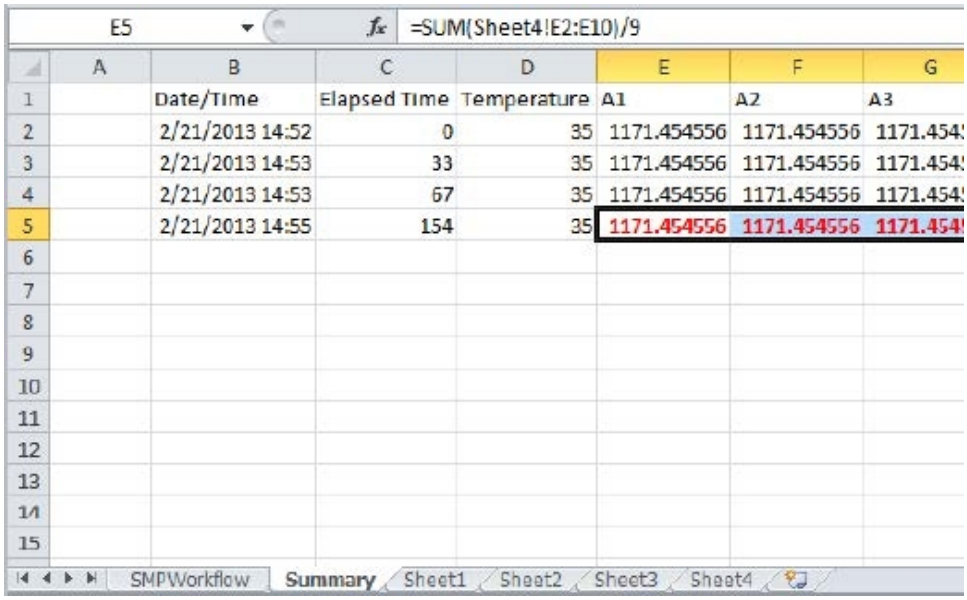
This example describes how to perform four Well Scan reads and write one result into each of Sheets 1 through 4. This CellFormat statement puts the calculated mean value of each point in a well, for one read, into the Summary worksheet.

```
CellFormat      MyMean          Summary:E       =Sum(#Column)/#RowCount
...
Loop            4
                Worksheet       Add
...
                StartRead
                GetDataCopy
...
                ProcessCommandResult    WriteRow        MyWellScanData
                FormatWorksheet         WriteCell       MyMean
EndLoop
...
```

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 tracks where acquired data has been written in each worksheet, which allows the workflow to substitute the #Column and #RowCount with real values that match the active worksheet. The substitution displays 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.



CellFormat Statement Example 4

When subsequent ProcessCommandResult statements reference MyTemperature, they write the returned value of the last executed automation command into the first blank cell in Column D of the Excel worksheet that is selected when the ProcessCommandResult executes.

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						

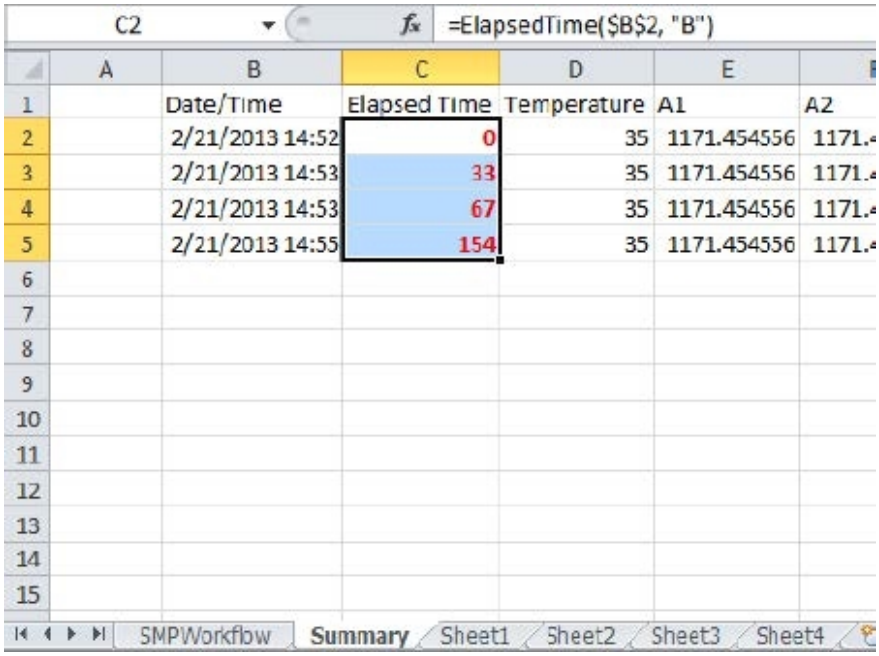
SMPWorkflow | Summary | **Sheet1** | Sheet2 | Sheet3 | Sheet4

CellFormat Statement Example 5

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. See Custom Excel Formulas on page 47.

CellFormat	MyElapsedTime	C	=ElapsedTime(B2, "B")

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



RowFormat Statement

The RowFormat statement allows you to write a plate of data as a row in an Excel worksheet. It also allows you to write ancillary information related to plate data, such as well names (A1, A2, and so on) and Group Name Assignments (Control, Unknown, and so on). Use the RowFormat statement in conjunction with the following workflow statements:

- FormatWorksheet Statement, see page 41
- ProcessCommandResult Statement, see page 42

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 39.

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, the output is written to next free row in the target column. This ensures headings are written out only once in discontinuous reads.
- Row Contents (optional)
  The number of subsequent arguments is variable. There are three keywords you can use 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 plate.
  - 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

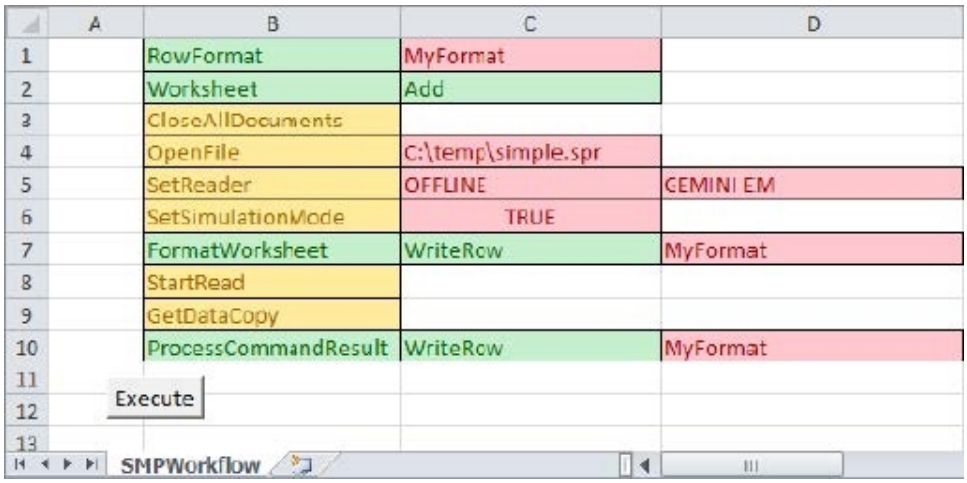
If no Start Column is given, then the row starts to write in Column A.

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

For a full workflow that uses default RowFormat values, see RowFormat Statement Example 1 on page 39.

RowFormat Statement Example 1

This example displays 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.



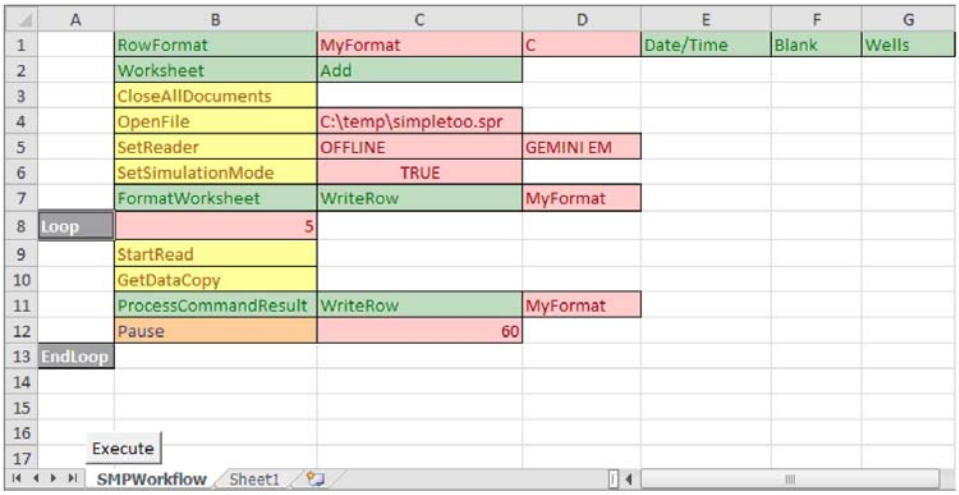
This complete workflow creates the following Excel worksheet data file from one nine-point well scan read that uses the instrument simulator.

	A	B	C	D	E	F	G	H	I
1	Date/Time	A1	A2	A3	A4	A5	A6	A7	A8
2	2/21/2013 16:32	1000	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	1841.47098
4	2/21/2013 16:32	1909.29743	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	1141.12001
6	2/21/2013 16:32	243.1975	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	41.07573
8	2/21/2013 16:32	720.5845	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	1656.9866
10	2/21/2013 16:32	1989.35825	1989.35825	1989.35825	1989.35825	1989.35825	1989.35825	1989.35825	1989.35825
11									

SMPWorkflow | Sheet1

RowFormat Statement Example 2

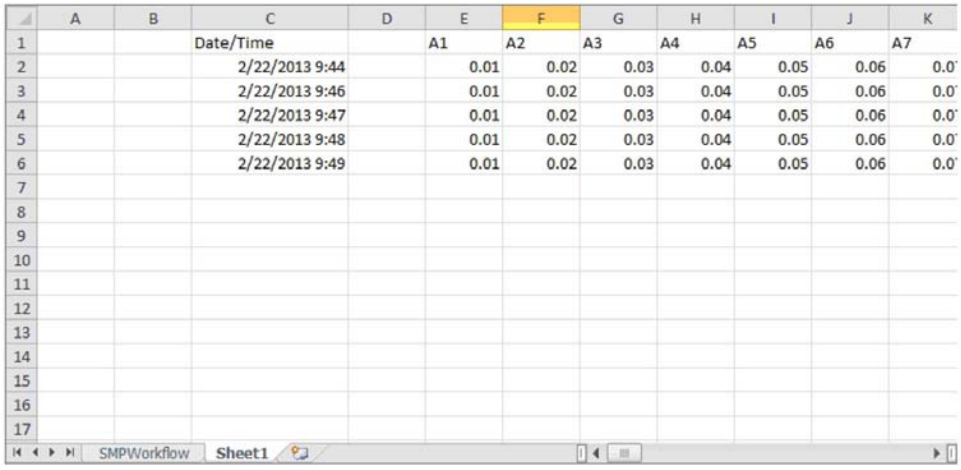
This example describes how the RowFormat statement starts writing rows of data in column C and leaves a blank column between the date/time and the well data.



	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					

SMPWorkflow | Sheet1

This complete workflow creates the following Excel worksheet data file from five endpoint reads that use 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											

SMPWorkflow | Sheet1

Writing To Worksheets

After you have the workflow define the write format, the following statements allow you to have the workflow write the data to the worksheet. 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:

- FormatWorksheet Statement (see below)
- ProcessCommandResult Statement, see page 42

FormatWorksheet Statement

The FormatWorksheet 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 this argument must reference a previously defined CellFormat.
  - If the first argument specifies WriteRow this argument usually references a previously defined RowFormat. The exception is when a formula that uses keywords appears in a CellFormat. See CellFormat Statement Example 3 on page 35.

Defaults

No default values

Examples

See CellFormat Statement on page 32 and RowFormat Statement on page 38.

ProcessCommandResult Statement

The ProcessCommandResult statement takes data that is 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 executes prior to the ProcessCommandResult statement is the command that is processed.

For descriptions of and parameters for the automation commands, see SoftMax Pro Automation Commands on page 51.

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 this argument must reference a previously defined CellFormat.
  - If the first argument specifies WriteRow this argument this must reference a previously defined RowFormat.

Defaults

No default values

Examples

See CellFormat Statement on page 32 and RowFormat Statement on page 38.

Workflow Flow Control Statements

Workflow flow control statements allow 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 (see below)
- Pause Statement, see page 44
- WaitUntil Statement, see page 45

Loop and EndLoop Statements

The Loop statement and the EndLoop statement allow you 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 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.

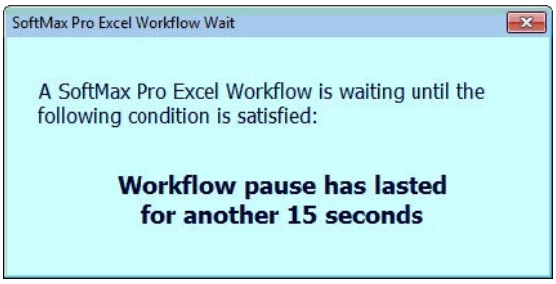
Example

This example describes the series of statements between Loop and EndLoop repeats five (5) times. The SetTemperature and OpenDrawer statements execute once each.

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

Pause Statement

The Pause statement pauses workflow execution for a fixed number of seconds. When the workflow pauses a dialog displays.



Arguments

Pause has one argument:

- The number of seconds the workflow should pause

Defaults

No default values

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

The WaitUntil statement pauses workflow execution until a given condition has been satisfied. A dialog displays 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 displays while the pause is in effect. This optional argument makes the wait reason more descriptive by adding context or using a local language.

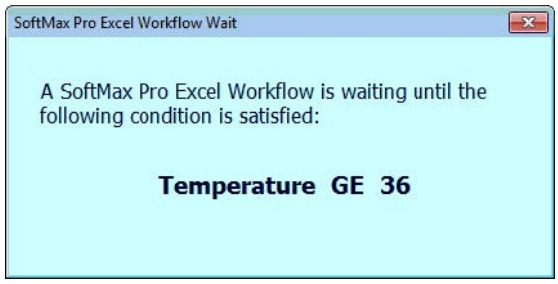
Defaults

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

Examples

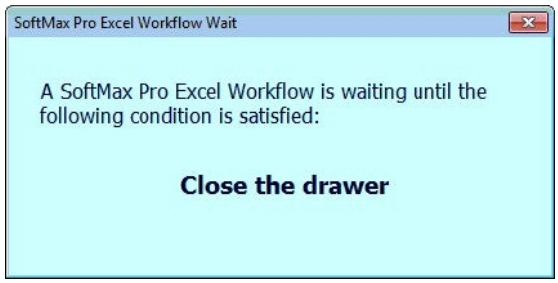
The following WaitUntil statement pauses until the instrument incubator is equal to, or greater than, 36 degrees.

While the workflow pauses the following dialog displays.



The following WaitUntil statement pauses until the instrument drawer closes.

While the workflow pauses the following dialog displays.



Custom Excel Formulas

You can write custom formulas for use in a SoftMax Pro Excel workflow. You should add additional custom formulas to the SoftMax Pro Excel Visual Basic module called SMPWorkflow. See CellFormat Statement on page 32.

ElapsedTime Custom Formula

This custom formula writes a column of elapsed time values, in seconds, between successive instrument reads. This assists in plotting a graph of data collected from instrument reads over time. The ElapsedTime custom formula is provided with the SoftMax Pro Excel workflows.

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 that contains start date/time against which all other dates are compared.
- Date Column is the column that contains the dates to compare against the Base Date. The Date Column must be in quotation marks, for example, “C” for column C of the worksheet.

ElapsedTime Example

This example describes how to use 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(A2, "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	0.05
3	2/22/2013 11:13	19	0.01	0.02	0.03	0.04	0.05
4	2/22/2013 11:14	38	0.01	0.02	0.03	0.04	0.05
5	2/22/2013 11:14	57	0.01	0.02	0.03	0.04	0.05
6	2/22/2013 11:14	76	0.01	0.02	0.03	0.04	0.05

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 the software to the instrument before starting the workflow. This command instructs the workflow to connect to the instrument without user intervention.

To connect to an instrument or to an instrument in Simulator mode, 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 you write or change a workflow, copy and paste the instrument name into place in the workflow. This helps prevent mismatches in the format of the instrument name.

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 Simulator Mode

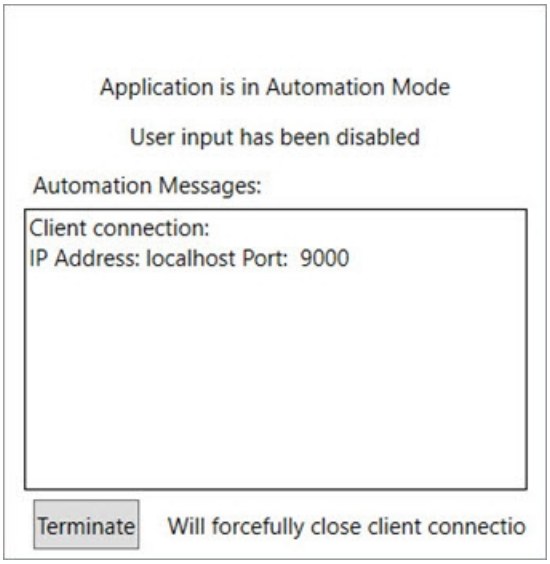
The instrument should be set as Offline and the SetSimulation command should say TRUE.

SetReader	OFFLINE	SPECTRAmax M5
SetSimulationMode	TRUE	

Troubleshooting Excel Workflows

Do the following to troubleshoot the Excel workflows:

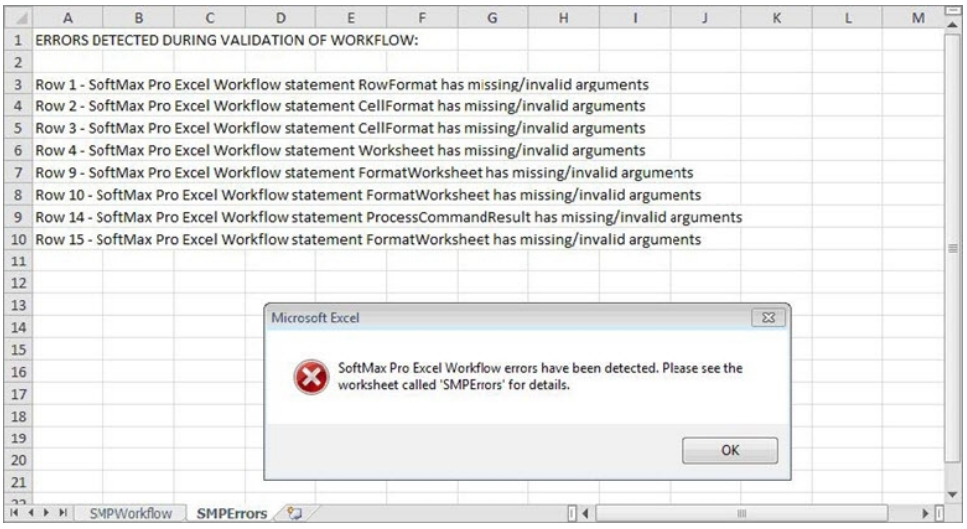
- Make sure the SoftMax Pro Excel Add-In is enabled in Excel.
- Make sure the SoftMax Pro Software is started and the Automation Mode dialog does not display.



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

Workflow Errors

Errors that the workflow detects display in the SMPErrors worksheet.



Check for additional error messages in the SoftMax Pro Software Automation Mode dialog under Automation Messages. This dialog reports errors that the workflow might not report.

Chapter 4: SoftMax Pro Automation Commands

The following topics describe the SoftMax Pro automation commands.

- **AppendData** - Reads the current Plate section and appends the new data to the existing data.
- **AppendTitle** - Appends the text to the title of the specified section.
- **CloseDocument** - Closes the current document.
- **CloseAllDocuments** - Closes all open documents.
- **CloseDrawer** - Closes an instrument drawer.
- **Dispose** - Closes the final automation application dialog.
- **ExportAs** - Exports data in the Column, Plate, or .xml format.
- **ExportSectionAs** - Exports a section.
- **GetAllFolders** - Extracts folder information.
- **GetAutosaveState** - Returns the Auto Save setting for the document.
- **GetDataCopy** - Copies the data to the client by way of an event.
- **GetDocuments** - Extracts document information.
- **GetDrawerStatus** - Returns the state of an instrument drawer.
- **GetFormulaResult** - Returns the result of a formula.
- **GetGroupNameAssignments** - Returns the group name assignments of the Plate section.
- **GetInstrumentStatus** - Returns the instrument status.
- **GetNumberPlateSections** - Returns the number of Plate sections in the experiment.
- **GetTempreature** - Returns the 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 document.
- **ImportPlateTemplate** - Imports a template from a file into a 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 in the document.
- **NewNote** - Creates a new Note section in the experiment.
- **NewPlate** - Creates a new Plate section in the experiment.
- **OpenDrawer** - Opens an instrument drawer.
- **OpenFile** - Opens a document.
- **Quit** - Exits the SoftMax Pro Software.
- **SaveAs** - Saves the document as a protocol or data document.
- **SelectNextPlateSection** - Selects the next Plate section in the experiment.
- **SelectSection** - Selects a section by name or by the order of the sections within the experiment.
- **SetReader** - Selects a reader type and sets the reader status.
- **SetShake** - Shakes the plate.
- **SetSimulationMode** - Sets the SoftMax Pro Software into Simulator mode.
- **SetTemperature** - Sets the instrument incubator temperature.
- **SetTitle** - Sets the title of a section.
- **StartRead** - Reads the Plate section or Cuvette Set section.
- **StopRead** - Stops the read of the Plate section or Cuvette Set section.

AppendRead

```
Int32 AppendRead()
```

The AppendRead command reads the Plate section and appends the new read data to the existing data.

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

See StartRead on page 111.

Parameters

None

AppendTitle

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

The AppendTitle command appends text to the section title. This cannot alter the experiment name.

Parameters

titletext

Type: String

The text to append to the section title.

Example

This example describes how to append text to a Plate section title.

```
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.CommandStatusEventArg 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()
```

The CloseDocument command closes the current document.

Note: For the SoftMax Pro Software - GxP edition, the document must be unlocked, the document status must be In Work, and all statements must be unsigned.

If the data is not saved, the document closes with no warning and data is not saved.

Parameters

None

CloseAllDocuments

```
Int32 CloseAllDocuments()
```

The CloseAllDocuments command closes all open documents.

Parameters

None

CloseDrawer

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

The CloseDrawer command closes the drawer you specify on the instrument. This command closes the plate drawer for most instruments.

Parameters

CloseDrawer parameters are recognized by the FlexStation® 3 Multi-Mode Microplate Reader only. The parameters are ignored and can be omitted for all other instruments.

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. If you omit this parameter the assay plate drawer closes.

Dispose

```
Void Dispose()
```

The Dispose command closes the final automation application dialog. The termination dialog that contains the Terminate button displays in the SoftMax Pro Software after the script execution completes.

Parameters

None

Example

This example describes how to close the automation dialog and exit the automation application. This example includes the code to complete command execution and dispose of the interface dialog.

```
public void Main()
{
   AutomationObject.Initialize();
   AutomationObject.CommandCompleted+= CommandCompleted;
   AutomationObject.ErrorReport+= Error;
   AutomationObject.OpenDrawer();
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArg 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)
```

The ExportAs command exports data in the Columns, Plate, or .xml format. The ExportAs command overwrites any existing file without warning.

Note: The PDF ExportAs behaves the same as the AutoExport PDF. The audit trail does not export to a .pdf format therefore you cannot use the .pdf export function to print an exported document that contains all content.

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 (UTF-16).

XML5 exports data in a .xml format similar to the way the SoftMax Pro Software did in version 5.x (UTF-8).

PDF exports data in a .pdf file format.

custom (also requires the following: Int 32 ExportAs(string path, ExportAsFormat, string styleSheetName)

Example

This example describes how to output data in .xml, Plate, and Columns formats and uses TIME in the ExportAsFormat parameter to export data in the 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.CommandStatusEventArg 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)
```

The ExportSectionAs command exports an experiment 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 that corresponds to a plate 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 overwrites any existing file without warning.

GetAllFolders

```
Int32 GetAllFolders(string folder, string type)
```

The GetAllFolders command returns folder information from the database or file system.

Parameters

folder

Type: String, Default = root folder

Type

Type: String, Default = data

Values:

data - Gets a list of all existing folders with data documents inside. For example, command `GetAllFolders(rootFolder) -> data` gets a list of all subfolders starting with rootfolder that contain data documents.

protocol - Gets a list of all existing folders with protocol documents inside. For example, command `GetAllFolders(rootFolder) -> protocol` gets a list of all subfolders starting with rootfolder that contain protocol documents.

all - Gets both document types.

Example

This example describes how to extract folder information for export.

```
public void Main()
{
   //initalize AutomationObject and hook events
   InitializeAndHookEvents();

   //start of script commands
   AutomationObject.GetAllFolders();
   var indexOfCommand = GetAllFolder(rootFolder, "data")
   //AutomationObject.GetDocuments("Folder","data");
   //AutomationObject.OpenFile("FolderAndName.sdax");

   //var commandID = AutomationObject.ExportAs(mExportFolder + "Plate.txt",
   SoftMaxPro.AutomationClient.SMPAutomationClient.ExportAsFormat.PLATE );
   //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.CommandStatusEventArg e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() + " -
    " + e.StringResult);

    //Script end, Queue is empty
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - Script done.");
        UnhookEventsAndDispose();
    }
}
private void InitializeAndHookEvents()
{
    Results.AppendResult("Initialize AutomationObject");
    AutomationObject.Initialize("localhost");

    Results.AppendResult("Hooking events");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
}
private void UnhookEventsAndDispose()
{
    Results.AppendResult("Disconnecting events");
    AutomationObject.ErrorReport -= Error;
    AutomationObject.CommandCompleted -= CommandCompleted;
    AutomationObject.InstrumentStatusChanged -= InstrumentStatus;

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

GetAutosaveState

```
Int32 GetAutosaveState()
```

The GetAutosaveState command returns the current autosave setting for the active document.

Parameters

None

Returns

This function returns the command ID used to retrieve the data that the CommandCompleted Event returns.

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

GetCopyData

```
Int32 GetDataCopy()
```

The GetCopyData command copies data to the client by way of an event. The format of the copied data depends on the display settings for the 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 used to retrieve the data that the CommandCompleted Event returns.

Data Returned Through the CommandCompleted Event

Return type: String

Returns the data for the Plate section.

Example

This examples describes 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.CommandStatusEventArg 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);
{
```

GetDocuments

`Int32 GetDocuments(folderName)`

The GetDocuments command extracts document information.

Parameters

path

Type: String, Default = root

Type: String, Default = data

Values:

data - Gets a list of all data documents within the folder. For example, command `GetDocuments (rootFolder) -> data` gets a list of all documents in subfolders starting with the root folder.

protocol - Gets a list of all protocol documents within the folder. For example, command `GetDocuments (rootFolder) -> protocol` gets a list of all documents in subfolders starting with root folder.

all - Gets both document types.

Example

This example describes how to extract folder information for export.

```
public void Main()
{
    //initalize AutomationObject and hook events
    InitializeAndHookEvents();

    //start of script commands
    var indexOfCommand = AutomationObject.GetDocuments("C:\xy", "data")

    //AutomationObject.GetDocuments("Test","data");
    //AutomationObject.OpenFile("DocumentAndName.sdax");

    //var commandID = AutomationObject.ExportAs(mExportFolder + "Plate.txt",
    SoftMaxPro.AutomationClient.SMPAutomationClient.ExportAsFormat.PLATE );
    //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.CommandStatusEventArg e)
{
    Results.AppendResult("Command complete Command ID = " + e.QueueID.ToString() + " -
    " + e.StringResult);
```

```
    //Script end, Queue is empty
    if( e.QueueEmpty)
    {
        Results.AppendResult("Queue empty - Script done.");
        UnhookEventsAndDispose();
    }
}
private void InitializeAndHookEvents()
{
    Results.AppendResult("Initialize AutomationObject");
    AutomationObject.Initialize("localhost");

    Results.AppendResult("Hooking events");
    AutomationObject.ErrorReport += Error;
    AutomationObject.CommandCompleted+= CommandCompleted;
    AutomationObject.InstrumentStatusChanged += InstrumentStatus;
}
private void UnhookEventsAndDispose()
{
    Results.AppendResult("Disconnecting events");
    AutomationObject.ErrorReport -= Error;
    AutomationObject.CommandCompleted -= CommandCompleted;
    AutomationObject.InstrumentStatusChanged -= InstrumentStatus;

    Results.AppendResult("AutomationObject disposed.");
    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)
```

The GetDrawerStatus command returns the state of the drawer on the instrument. This command returns the state of the plate drawer for most instruments.

Parameters

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

drawerType

Type: String

Must be one of the following:

- “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. The status of the assay plate drawer is returned if you omit this parameter.

Returns

This function returns the command ID used to retrieve the following from 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

This example describes how to determine if each drawer on a FlexStation 3 is open or closed. All other instruments do not require this parameter.

```
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.CommandStatusEventArg 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)
```

The GetFormulaResult command returns the result from a formula. This command requires the name of the formula, name of either the Note section or the Group section, and the name of the experiment. If you do not specify the experiment name, it searches from the first experiment in 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 used to retrieve the data that the CommandCompleted Event returns.

Data Returned Through the CommandCompleted Event

Type: String

Calculated value for the formula

Example

This example describes how to get information from a Note section. If the Note 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.CommandStatusEventArg 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
```

The GetGroupNameAssignment command returns the group name assignments of a Plate section.

Parameters

None

Returns

The function returns the command ID used to retrieve the data that the CommandCompleted Event returns.

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 return as blank.

GetInstrumentStatus

```
Int32 GetInstrumentStatus
```

The GetInstrumentStatus command returns the instrument status.

Parameters

None

Returns

The function returns the command ID used to retrieve the data that the CommandCompleted Event returns.

Data Returned Through the CommandCompleted Event

Return Type: String

For the possible values, see InstrumentStatusChanged on page 116.

GetNumberPlateSections

```
Int32 GetNumberPlateSections()
```

The GetNumberPlateSections command returns the number of Plate sections in an experiment.

Parameters: None

Returns

Returns the command ID used to retrieve the data that the CommandCompleted Event returns.

Data Returned Through the CommandCompleted Event

Return Type: Int32

Returns the number of Plate sections in the active experiment.

Example

This example describes how to determine the number of Plate sections in an 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.CommandStatusEventArg 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()
```

The GetTemperature command returns the instrument incubator temperature.

Parameters

None

Returns

The function returns the command ID used to retrieve the data that the CommandCompleted Event returns.

Data Returned Through the CommandCompleted Event

Return Type: Double

Instrument incubator temperature

GetVersion

```
Int32 GetVersion()
```

The GetVersion command returns the version number of the SoftMax Pro Software automation Interface.

Parameters

None

Returns

This function returns the command ID used to retrieve the data that the CommandCompleted Event returns.

Data Returned Through the CommandCompleted Event

Return Type: String

Returns the version number of the SoftMax Pro Software.

The return string is the following property:

SMPAutomationClient.Version.VERSION6300

Example

This example describes how to get the version number of the SoftMax Pro Software.

```
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.CommandStatusEventArg 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)
```

The ImportPlateData command imports data from a file into a Plate section in a document.

Note: For the SoftMax Pro Software - GxP edition, the document must be unlocked, the document status must be In Work, and all statements must be unsigned.

Parameters

importType

Type: String

Must be one of the following:

- “Plate Format"
- "XML Format"

importParameter

Type: Params String

Fully qualified path name of the file that contains the data to import. The file must match the format you use for manual data imports.

Example

The example on the following page describes how to import plate 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.CommandStatusEventArg 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)
```

The ImportPlateTemplate command imports a template from a file into a Plate section.

Note: For the SoftMax Pro Software - GxP edition, the document must be unlocked, the document status must be In Work, and all statements must be unsigned.

Parameters

path

Type: String

Fully qualified path name of the file that contains the template to import. The file must match the format you use for manual template imports.

Example

This example describes how to import 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.CommandStatusEventArg 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

The Initialize commands initialize 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 (), see page 81
- bool initialize(string server), see page 82
- bool initialize(String server, int port), see page 83

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.CommandStatusEventArg 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.CommandStatusEventArg 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 the SoftMax Pro Software 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 userName, String password, String projectName)
```

Parameters

userName

Type: String

The user name for the account

password

Type: String

The password for the account

projectName

Type: String

The Project for the user to use

Logoff

```
Int32 Logoff()
```

The Logoff command logs a user off from the SoftMax Pro GxP Software.

Parameters

None.

NewDocument

```
Int32 NewDocument()
```

The NewDocument command creates a new document that uses the Default Protocol.spr protocol settings.

Note: Not available for SoftMax Pro Software - GxP edition.

Parameters

None

Example

This example describes how to create a new document that uses the default protocol settings.

```
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.CommandStatusEventArg 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()
```

The NewExperiment command creates a new experiment in a document.

Note: For the SoftMax Pro Software - GxP edition, the document must be unlocked, the document status must be In Work, and all statements must be unsigned.

Parameters

None

Example

This example describes how to create a new experiment in a document.

```
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.CommandStatusEventArg 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()
```

The NewNotes command creates a new Note section in an experiment.

Note: For the SoftMax Pro Software - GxP edition, the document must be unlocked, the document status must be In Work, and all statements must be unsigned.

Parameters

None

Example

This example describes how to create a Note section.

```
int mStatusID;
public void Main()
{
    AutomationObject.Initialize("localhost");
    AutomationObject.CommandCompleted += CommandCompleted;
    AutomationObject.ErrorReport += Error;
    AutomationObject.NewDocument();
    mStatusID = AutomationObject.NewNotes("<enter note text here>");
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArg 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()
```

The NewPlate command creates a new Plate section in an experiment.

Note: For the SoftMax Pro Software - GxP edition, the user must have the Generate Compliance Data permission.

Note: For the SoftMax Pro Software - GxP edition, the document must be unlocked, the document status must be In Work, and all statements must be unsigned.

Parameters

None

Example

This example describes how to create a Plate section in an experiment.

```
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.CommandStatusEventArg 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

The OpenDrawer commands open a drawer on the instrument. This command opens the plate drawer for most instruments.

Note: For instruments with temperature control, the plate drawer cannot open when the incubator is on. See SetTemperature on page 107.

Parameters for the FlexStation 3

The OpenDrawer drawerType parameter is recognized by the FlexStation 3 only. This parameter is ignored and can be omitted for all other instruments.

drawerType

Type: String

Must be one of the following:

- “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. The assay plate drawer opens if you omit this parameter.

Parameters for Other Instruments

The OpenDrawer xPosition, yPosition, and locked parameters are recognized by the FilterMax™ F3 Multi-Mode Microplate Reader, FilterMax™ F5 Multi-Mode Microplate Reader, SpectraMax® i3 Multi-Mode Microplate Reader, SpectraMax® i3x Multi-Mode Microplate Reader, and SpectraMax® Paradigm® Multi-Mode Microplate Reader only. These parameters are ignored and can be omitted for all other instruments.

xPosition

Type: Int32

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

- Range for the SpectraMax i3 and SpectraMax i3x: 0 to 2900
- Range for the SpectraMax Paradigm: 0 to 2950
- Range for the FilterMax F3 and FilterMax F5: 0 to 2950

yPosition

Type: Int32

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

- Range for the SpectraMax i3 and SpectraMax i3x: 5200 to 6900
- Range for the SpectraMax Paradigm: 5200 to 6900
- Range for the FilterMax F3 and FilterMax F5: 5200 to 6900

locked

Type: Bool

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

Example

This example describes how to open the plate 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.CommandStatusEventArg 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)
```

The OpenFile command opens a protocol or data document. If the file is not found, the SoftMax Pro application does nothing.

Parameters

pathname

Type: String

(SoftMax Pro Software - Standard edition) - Fully qualified path to protocol file or protocol name

(SoftMax Pro Software - GxP edition) - Project name / full path / document name

Example

This example describes how to open a file that exists in the file system.

```
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.CommandStatusEventArg 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()
```

The Quit command exits the SoftMax Pro Software.

Parameters

None

Example

This example describes how to quit the SoftMax Pro Software.

```
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.CommandStatusEventArg 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)
```

The SaveAs command saves the current document as a data document or a protocol. Define the document type by the file extension in the path statement. For the SoftMax Pro Software - Standard edition, use the *.sda file extension for data documents and *.spr file extension for protocols. For the SoftMax Pro Software - GxP edition, enter the .sdax file extension.

If a document with the same name already exists, the SoftMax Pro Software - Standard edition automatically overwrites the document with no warning.

Note: The SoftMax Pro Software - GxP edition does not allow you to save the document as a Protocol and prevents you from overwriting an existing document.

Parameters

pathname

Type: String

Fully qualified path and name of document to save, including the file extension.

Example

This example describes how to save a document to an undefined “Temp” path. To define the path, you need to define the mPath string with an absolute or relative path that includes the name of the document 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.CommandStatusEventArg 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()
```

The SelectNextPlateSection command selects the next Plate section in an experiment.

Parameters

None

Example

This example describes how to select the next Plate section after the Plate1 Plate section.

```
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.CommandStatusEventArg 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)
```

The SelectSection commands select a section by name or by the order of the section within an experiment. SelectSection using the System.String parameter is useful for multi-plate protocols or to select Group tables to copy.

Parameters

sectionName

Type: String

The name of the section.

Either the name of a section within an 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

This example describes how to select a section using the System.String parameter.

```
int mStatusID;
public void Main()
{
   AutomationObject.Initialize("localhost");
   AutomationObject.CommandCompleted += CommandCompleted;
   AutomationObject.ErrorReport += Error;
   AutomationObject.NewExperiment();
   mStatusID = AutomationObject.SelectSection("Pleate1@Exp02");
   AutomationObject.NewExperiment();
   mStatusID = AutomationObject.SelectSection(2);
   AutomationObject.AppendTitle("_&&^^**%%$$##@@");
}
private void CommandCompleted( object sender,
SoftMaxPro.AutomationClient.SMPAutomationClient.CommandStatusEventArg 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)
```

The SetReader command selects a microplate reader type and sets the instrument status. See Examples on page 119.

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 instrument state to Offline.

instrument

Type: String

340PC384

Abs

AbsPlus

DTX 800

DTX 880

Emax

EMax Plus

FilterMax F3

FilterMax F5

Flexstation III

GEMINI EM

GEMINI XPS

PLUS190PC

PLUS384

SpectraMax i3

SpectraMax i3x

SpectraMax iD3

SpectraMax iD5

SPECTRAmax L

SPECTRAmax M2

SPECTRAmax M2e

SPECTRAmax M3

SPECTRAmax M4

SPECTRAmax M5

SPECTRAmax M5e

SpectraMax Mini

SpectraMax Paradigm
VersaMaxPLUS
Vmax

SetShake

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

The SetShake command shakes the plate. When this command is set to true, starts shaking the plate tray until the Shake(false) command is sent. By default, the shake stops after 30 seconds in most instruments.

Note: The SetShake command does not support the FilterMax F3, FilterMax F5, SpectraMax i3, SpectraMax i3x, and SpectraMax Paradigm.

Parameters

shakestate

Type: Boolean

true turns Shake on.

false turns Shake off.

Example

This example describes how to turn 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.CommandStatusEventArg 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);
}
```

SimulationMode

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

The SimulationMode command sets the SoftMax Pro Software into simulator mode.

Parameters

modestate

Type: Boolean

true enables simulator mode.

false disables simulator mode.

Example

This example describes how to select an instrument, enable simulator mode, and start 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.CommandStatusEventArg 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)
```

The SetTemperature command sets the instrument incubator temperature. Set the temperature to zero to turn off the incubator.

Parameters

Temperature

Type: Double

The number of degrees centigrade

Example

This example describes how to set 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.CommandStatusEventArg 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()
```

The SetTitle command sets the title of a section in an experiment. The command confirms that the name is unique within the current experiment and if not, makes no change.

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

Note: For the SoftMax Pro Software - GxP edition, the document must be unlocked, the document status must be In Work, and all statements must be unsigned.

Parameters

None

Example

This example describes how to set a Plate section title to Plate 100.

```
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.CommandStatusEventArg 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);
}
```

StartRead

```
Int32 StartRead()
```

The StartRead command reads a Plate section or CuvetteSet section. If the current section is neither a Plate section nor CuvetteSet section, the command reads the next Plate section or CuvetteSet section.

The SoftMax Pro Software - Standard edition confirms that Auto Export is enabled and the SoftMax Pro Software - GxP edition confirms that Auto Save is enabled.

Parameters

None

Example

This example describes how to start a plate read.

```
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.CommandStatusEventArg 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()
```

The StopRead command stops the read of a Plate section or Cuvette Set section. This command is not queued.

Parameters

None

Chapter 5: 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
```

The CommandCompleted event generates a client callback when a command completes.

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
QueueId	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 122.

InstrumentStatusChanged

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

The InstrumentStatusChanged events generate a client callback when the status of the connected instrument changes.

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 122

ErrorReport

```
event EventHandler(object sender,
SoftMaxPro.Automation.SMPAutomationclient.ErrorEventArgs)
ErrorReport
```

The ErrorReport events generate a client callback when an error condition occurs.

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 generates, it is 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 which commands are executed when an error is 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);
}
```

Chapter 6: 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.CommandStatusEventArg 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 Command 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.CommandStatusEventArg 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.CommandStatusEventArg 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.CommandStatusEventArg 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();
    }
}
```

Obtaining Support

Molecular Devices is a leading worldwide manufacturer and distributor of analytical instrumentation, software, and reagents. We are committed to the quality of our products and to fully supporting our customers with the highest level of technical service.

Our Support website, support.moleculardevices.com, has a link to the Knowledge Base, which contains technical notes, software upgrades, safety data sheets, and other resources. If you still need assistance after consulting the Knowledge Base, you can submit a request to Molecular Devices Technical Support.

You can contact your local representative or Molecular Devices Technical Support at 800-635-5577 x 1815 (North America only) or +1 408-747-1700.
In Europe call +44 (0) 118 944 8000.

To find regional support contact information, visit www.moleculardevices.com/contact.

Contact Us

Phone: +1-800-635-5577
Web: moleculardevices.com
Email: info@moldev.com

Visit our website for a current listing of worldwide distributors.

5049094 E