MBScript Reference

Rev. 2.1

Table of Contents

Purp

The Anewin MBScript Reference provides comprehensive information regarding the MBScript language. The MBScript language is integrated into the FiveWire application providing programmable control of the product features. This supports automation of tests and analysis of captured data.

Section 1 MBScript Commands

Overview

The MicroBench MBScript scripting language provides the ability to automate the setup and control of the instrument tool set. It follows the syntactic conventions of the 'C' language with some more modern modifications and extensions. If you are familiar with the 'C' language, MBScript will feel very natural to you.

Differences from C

Interpreted not Compiled

MBScript source files are simple text files. When the script is run, it is checked for certain kinds of errors and processed to remove non-executable content. It is then interpreted sequentially until the last line is reached or the 'exit' command is executed.

Automatic Variables

Unlike 'C', variables in MBScript do not need to be declared. Variables are created as they are encountered during execution. Two types of variables are supported: numeric and string. During evaluation of the variable, if its contents can be interpreted as a number it will be treated as a numeric variable. Otherwise it will be treated as a string. For example:

```
a=5; // Numeric assignment
b=a; // b now contains the number 5
b=c; // Terminates with an error, c is not yet defined
b="c"; // Assigns the string "c" to the variable b
```

The '+' sum operation can be applied to numbers or strings producing either a numeric sum or a concatenated string. Both arguments to the sum operator must be of the same type to avoid ambiguity. The functions toStr(<value>) and toNum(<string>) provide conversion between the types to force the desired operation outcome.

Order of evaluation

Expressions are always evaluated left to right. There is no implicit precident so parenthesis should be used to force correct evaluation. For example:

```
a < 2 && a > 0 // Produces an error
(a < 2) && (a > 0) // Use instead
```

Associative Arrays

MBScript provides a single array type. Each array includes a name and index. The index can be a number or string. Like variables above, an array element must be assigned before it can be accessed. For example:

```
a[0]=5;
a[1]=10;
a[0]=a[2]; // Produces an error – a[2] has not been assigned yet
b[first] = "first"; // Use a string to index the array b
b[second] = 2;
b[first]=b[second]; // b[first] now contains the number 2
```

MBScript by Example

Instrument Commands

Instrument commands are comprised of a command word and any required parameters separated by white space. Commands are sent to one of the tools in the instrument: liveLogic, logicAnalyzer, logicSource, waveformSource, protocol, or system.

There are two methods for sending commands to a tool. The first sends one command at a time and supports algorithmic command generation. It supports capturing any command response. The second sends multiple commands to a selected tool but the commands must be constant strings.

```
// First method
mbCommand("system",  "openWindow liveLogic");
// or
instrument = "liveLogic";
mbCommand("system",  "openWindow " + instrument);
// with return value
toolIsOpen = mbCommand("system",  "isToolOpen liveLogic");
if(toolIsOpen)
{
      // some code
}

// Second method
using system
{
      "openWindow liveLogic";
      "positionWindow liveLogic 100 100";
}
```

Tool commands either return a value or return an empty string. If there is an error in the command a string will be returned that starts with the word "Error".

Some tool commands that return a data array such as data samples.

```
// Get a captured waveform from Live Logic
waveform = mbCommand("liveLogic",  "getWaveform");
// Convert the waveform to an array of samples
numSamples = listToArray(waveform, ",", samples[], 0);
// Process each sample
for(n = 0; n < numSamples; n++)
{
      // Get the channel and time values from the sample
      listToArray(samples[n], ";", values[], 0);
      ch1 = values[0];
      ch2 = values[1];
      time = values[2];

      // Do something with the information
}
```

Descriptions of all of the tool commands are provided at the end of this document along with complete example programs.

String Manipulation

strLen(<string>) returns the length of a string

```
// n is set to 8
n = strLen("FiveWire");
// Show the value in a dialog box
textWindow(n);
```

strFind(<start>, <find>, <string>**)** returns the index of the start of the find string or -1 if not found.

```
// n is set to the location of the second I, 5 in this case
n = strFind(0, "ir", "FiveWire");
// Show the value in a dialog box
textWindow(n);
```

strSub(*<start>*, *<length>*, *<source>***)** returns a substring from the source string.

```
// Sets s to the string "ir"
s = strSub(5, 2, "FiveWire");
// Show the value in a dialog box
textWindow(s);
```

strFormat(<*format* >[, <*value0*>, <*value1*>, ...]) returns a formatted string

```
// Get the Logic Analyzer data for channel 4
ch = 4;
cmd = strFormat("getWaveform  {0}", toStr(ch));
chData = mbCommand("logicAnalyzer", cmd);
```

Predefined variables/constants

true has the value 1and **false** has the value 0

```
// Use a flag to control a loop
n=3;
flag = true;
do
{
      // Do something useful
      n--;
      flag = n > 0;
      textWindow(toStr(flag) + " " + toStr(n));
}
while(flag);

textWindow("Done");
```

date has the string value of the current date

time has the string value of the current time

```
// Get the date and time and display it
textWindow(date + ", " + time);
```

osTick has the value of the system tick in milliseconds

```
// Timed delay of 2000 ms
textWindow("Start");
futureTime = osTick + 2000;
while(osTick < futureTime)
      ;
textWindow("Done");
```

pi returns the constant 3.14159…

```
// The value of pi
textWindow(pi);
```

version has the numeric value of the MBScript version

```
// Check the current version of the MBScript interpreter
if(version < 1.0)
	textWindow("Need to upgrade");
else
	textWindow("Up to date");
```

scriptDirectory returns the complete path to the directory from which the script was started

```
// Get the directory of the executing script
textWindow(scriptDirectory);
```

myDocuments returns the complete path to my documents

```
// Get the directory of My Documents
textWindow(myDocuments);
```

abortFlag has the value of the abort flag cleared at startup and set if the message window 'Abort' button is pressed

```
// Keep executing until the user aborts the program
seconds = 0;
openMessageWindow();
while(!abortFlag)
{
	sleep(1000);
	seconds++;
	writeMessageLine(toStr(seconds) + " seconds", 0);
}
closeMessageWindow();
```

Variable scope

Variables declared outside of any curly braces {} are global to the script. Curly braces define a block of code and variables declared within that block are local to that block.

global

Forces the variable to exist in global scope. If the variable does not exist in global scope a new variable will be created in global scope.

```
// Using global
function setGlob(value)
{
	global glob = value;
}

if(isDefined(glob))
	textWindow(glob);
else
	textWindow("glob not defined");

setGlob(5);

if(isDefined(glob))
	textWindow(glob);
else
	textWindow("glob not defined");
```

local

If the named variable does not exist in the current scope but does exist in a wider scope, the wider scope variable will be used. The local prefix requires the variable to exist in the current scope. If it does not, a new variable will be created in the current scope even if the same variable name exists in a wider scope.

The local prefix can also be used to declare a global array before attempting to assign values to the array

```
// Using local
val = false;
function useLocal()
{
	local val = true;
}
useLocal();
if(val)
	textWindow("global val was changed");
else
	textWindow("global val was not changed");
```

```
// Retrieve LA acquisition
samples = mbCommand("logicAnalyzer", "getWaveform 210");
// Break into a list of samples
n = listToArray(samples, ",", sample[], 0);
// Declare data and time arrays in this scope
local data[];
local time[];
// Break each sample into its data and time components
for(i=0;i<n;i++)
{
      listToArray(sample[i], ";", values[], 0);
      data[i] = values[0];
      time[i] = values[1];
}
// Display the data and time components
openMessageWindow();
for(i=0;i<n;i++)
{
      msg = toStr(data[i]) + " : " + toStr(time[i]);
      writeMessage(msg);
}
// Wait for the user to terminate via abort button
while(!abortFlag)
      ;
closeMessageWindow();
exit()
```

Flow Control

if else statements

```
// Keep executing until the user aborts the program
upToDate = true;
if(version < 1.0)
{
      upToDate = false;
      textWindow("Older version");
}
else
      textWindow("Current version");
```

for statements

```
// For loops
for(n = 0; n < 2; n++)
	textWindow(n);

openMessageWindow();
for(n = 0; n < 10; n++)
{
	writeMessageLine(toStr(n), 0);
	sleep(1000);
	if(abortFlag)
	{
		closeMessageWindow();
		break;
	}
}
closeMessageWindow();
```

while and do-while statements

```
// While loop
n = 2;
textWindow("Start while");
while (n > 0)
{
      n--;
      sleep(1000);
}
textWindow("Done");
// Do loop
n = 2;
```

```
textWindow("Start do");
do
{
      n--;
      sleep(1000);
}
while (n > 0)
textWindow("Done");
```

switch statement

```
// switch - must be numeric variable
n = 3;
switch(n)
{
      case 1:
            textWindow("Value is 1");
            break;
      case 2:
            textWindow("Value is 2");
            break;
      case 3:
            textWindow("Value is 3");
            break;
      default:
            textWindow("Value is not expected");
            break;
}
```

break

Terminates the enclosing while, for, or do-while execution

continue

Restarts the execution of the enclosing while, for, or do-while at the beginning of the block.

User Defined Functions

Function parameters can be a number or string, an array reference such as *myArray[],* or an expression that evaluates to a number or string. Parameters are passed by value. An array reference passes a pointer to the calling functions array. Any changes made affect the callers array variable. If the array does not exist in the callers scope, an array will be created.

```
// Function that sends commands to Live Logic tool
function LL(cmd)
{
	response = mbCommand("liveLogic", cmd);
	return response;
}
// Use the function
resp = LL("run");
if(strLen(resp) > 0)
	textWindow(resp);
else
	textWindow("Done");
```

```
// Copy data and time to their own arrays
function copyValues(v[], da[], ti[])
{
      da[i] = values[0];
      ti[i] = values[1];
}

// Break a sample into its data and time components
function getValues(s[], d[], t[])
{
      for(i=0;i<n;i++)
      {
            listToArray(s[i], ";", values[], 0);
            copyValues(values[], d[], t[]);
      }
}

// Get data from LA and break it into sample array
samples = mbCommand("logicAnalyzer", "getWaveform 10");
n = listToArray(samples, ",", sample[], 0);

// Declare global arrays data and time
//   Could leave these lines out and allow the folling
//   function call to implicitly create them
local data[];
local time[];

// Process samples into data and time arrays
getValues(sample[], data[], time[]);

// Display the data for the user
openMessageWindow();
for(i=0;i<n;i++)
{
      msg = toStr(data[i]) + " : " + toStr(time[i]);
      writeMessage(msg);
}

// Clean up and exit when abort button is pressed
while(!abortFlag)
      ;
closeMessageWindow();

exit()
```

File I/O

MBScript provides the ability to read and write simple text files one line at a time.

```
// Write a single line to a file
data=5;
handle = openWriteFile("C:/someTextFile.txt");
writeFileLine(handle, strFormat("Data = ", data));
closeWriteFile(handle);


// Read from a file
handle=openReadFile("C:/someTextFile.txt");
line=readFileLine(handle);
if(0 != strFind(0,"::EOF:", line))
{
      s = strFormat("Line from file is: {0}", line);
      textWindow(s);
}
closeReadFile(handle);
```

GUI Windows

MBScript provides a number of windows that can display user data and other useful information.

textWindow(*message [, message, ...]***)**

Displays a text window showing the *message* string to the user and an 'OK' button. Script execution stops until the 'OK' button is pressed. One or more message strings can be included separated by commas. Each message will be presented in a sequence of lines in the dialog box.

```
// Display some text
textWindow(date,time);
```

yesNoWindow(*message [, message, ...]***)**

Displays a text window showing the *message* string to the user and two buttons, 'Yes' and 'No'. If the 'Yes' button is pressed *true* is returned. If the 'No' button is pressed *false* is returned. Script execution is stopped until one of the buttons is pressed. One or more message strings can be included separated by commas. Each line will be presented in a sequence of lines in the dialog box.

```
// Get user input
if(yesNoWindow("Yes or no"))
	textWindow("Selected yes");
else
	textWindow("Selected no");
```

openMessageWindow()

Displays a multi-line message window which does not block execution of the script. An Abort button is bound to the abortFlag providing a method to terminate the script.

writeMessage(*msg***)**

Displays the *msg* string in the message window on sequential lines

writeMessageLine(*msg, line***)**

Displays the *msg* string in the message window on the specified *line*

closeMessageWindow()

Closes the message window

```
// Display some data
openMessageWindow();
n = 5;
writeMessageLine("Count down", 0);
while (n > 0)
{
	writeMessage(toStr(n));
	sleep(1000);
	n--;
	if(abortFlag)
	{
		closeMessageWindow();
		break;
	}
}
closeMessageWindow();
```

readInput(*[[label], default input]*)

Returns the user input string or an empty string if canceled

```
// Get some data
userInput = readInput("Enter an amount", "3");
textWindow(userInput);
```

Standard Functions

mbCommand(*"tool", "command"*)

Sends a command string to the specified tool and returns the commands response.

```
// Get some data
runState = mbCommand("liveLogic", "getRunState");
if(runState == "stopped")
	textWindow("Live Logic is stopped");
else
	textWindow("Live Logic is running");
```

isDefined(*var*)

Returns true if *var* has been assigned (created) otherwise false

```
// Verify that vars are defined before using
myVar = "A defined variable";
if(isDefined(test))
	textWindow(test);
else
	textWindow("test is not defined");
if(isDefined(myVar))
	textWindow(myVar);
```

isArrayDefined(*arrayName[], index*)

Returns true if *var[index]* has been assigned (created) otherwise false

```
// Verify that array vars are defined before using
one = "one";
two = "two";
myVar[one] = 1;
if(isArrayDefined(myVar, one))
      textWindow(myVar[one]);
else
      textWindow("myVar[one] is not defined");
if(isArrayDefined(myVar, two))
      textWindow(myVar[two]);
else
      textWindow("myVar[two] is not defined");
```

runScript(*path*)

Executes script at *path*

```
// hello.mbs script in local directory
textWindow("Hello");

// Script that calls hello.mbs script
runScript(scriptDirectory + "hello.mbs");
```

breakpoint [*{string | variable | array}, ...*]

Stops script execution and presents a dialog indicating the source line of the break point and following lines which contain the name and value of listed variables or the contents of the literal string.

```
// Breakpoint debugging
a = 1;
b = 2;
c[0] = "test";
breakpoint "Stop and inspect vars" a b c[0];
```

sleep(*milliseconds***)**

Delays script execution for specified time

```
// Count down seconds
n = 5;
textWindow("Start");
while(n > 0)
{
	n--;
	sleep(1000);
}
textWindow("Done");
```

Binary Math Operations

Numeric: +, -, *, /, %, <<, >> ==, !=, >, <, >=, <=, ||, &&, |, &

String: +, ==, !=

```
// Do some math
//   Evaluation is from left to right
n = 5 + 3 - 2 / 2 * 3;
textWindow("The result is " + toStr(n));
textWindow("1 << 4 = " + toStr(1<<4));
textWindow("5 % 2 = " + toStr(5%2));
textWindow("5 > 4 = " + toStr(5>4));
textWindow("Incorrect! " + toStr(5>4 && 3>2));
textWindow("Correct " + toStr((5>4) && (3>2)));
textWindow("abc == abd = " + toStr("abc"=="abd"));
```

Unary Math Operations

Prefix: -, ++, --

Suffix: ++, --

```
// Do some math
n = 1;
textWindow("n++ " + toStr(n++));
textWindow("++n " + toStr(++n));
```

Equation

=, +=, -=, *=, /=, %=, |=, &=

```
// Do some math
n = 8;
n /= 4;
textWindow(n);
n = 1;
n |= 4;
textWindow(n);
```

Math Functions

sin(deg), cos(deg), tan(deg), rand(smallest, largest), log(), exp(), sqrt()

```
// Do some math
textWindow("sin(45) = " + toStr(sin(45)));
textWindow("rand(0,255) = " + toStr(rand(0,255)));
textWindow("sqrt(2) = " + toStr(sqrt(2)));
```

Libraries

Standard and user defined libraries may be included within the script using the addLibrary command. Libraries are MBScript files that are evaluated at the addLibrary line.

Libraries typically contain function definitions which are used in the main script. If executable code is included in the library it will be executed when the addLibrary line is evaluated.

A library stdLib.mbs in the "Documents/Anewin/FiveWire/userScripts" directory contains useful helper functions for accessing instruments and their data.

```
// Example library in the local file library.mbs
function LL(cmd)
{
      return mbCommand("liveLogic", cmd);
}

// MBScript file test.mbs that uses the local library
addLibrary scriptDirectory + "library.mbs";
LL("run");
```

Section 2 FiveWire System Commands

These commands provide control over functions that are system wide. To execute these commands from MBScript use **mbCommand("system", "*command*")** where *command* is one of the commands described below.

Window Commands

Open, close, or position a tool window

openWindow *tool*

Opens the specified tool window

closeWindow { *tool*, *all* }

Closes the specified tool window OR all tool windows

positionWindow *tool xPixel yPixel*

Moves the tool window to the specified screen location.

minimizeWindow *tool*

Minimizes the tool window to the Windows task bar

restoreWindow *tool*

Restores the tool window from the Windows task bar

Note: *tool* = {*system* | *liveLogic* | *waveformSource* | *protocol* | *logicSource* | *logicAnalyzer* | *pictureWindow*}

Setup and Execution Commands

Run a user defined script - may be .txt, .mbx, or .mbs. if no file extension .txt is assumed

runScript *fileName* [*directoryPath*]

Open or save the current system – same as System Open and System Save As... menus

openSystem [*directoryPath*]

saveSystem [*directoryPath*]

Examples:

```
mbCommand("system", "openWindow liveLogic");
mbCommand("system", "positionWindow liveLogic 20 150");
mbCommand("system", "closeWindow all");

mbCommand("system", "runScript myScript");
mbCommand("system", "runScript myScript C:/mb500/userScripts");

mbCommand("system", "openSystem");
mbCommand("system", "openSystem C:/MB500/myScripts");
```

Management and Control Commands

clickSystemButton

Pulses the system event

instrumentConnected

Returns {true, false}
A flag indicating if the MB-500 is connected to this PC

isToolOpen *tool*

Returns {true, false}
A flag indicating a tool window is open

getRevision

Returns swRev, fwRev, hwRev
The current SW, FW and HW revisions

getSwRevisionValues

Returns swMajor, swMinor, swSub
The current SW revision

Example:

```
openMessageWindow();

// Get the revision from system
revision = mbCommand("system", "getRevision");

// Convert the csv response to a list of components
```

```
listToArray(revision, ",", components[], 0);

// Display the revision
writeMessage("Software revision: " + toStr(components[0]));
writeMessage("Firmware revision: " + toStr(components[1]));
writeMessage("Hardware revision: " + toStr(components[2]));
```

getSerialNumber

Returns serialNumber
The instruments serial number

getCurrentPath

Returns pathToScriptFile
The full directory path being used by the script

Section 3 FiveWire Tool Commands

Tool Commands directly control or return values from the FiveWire tools. Each tool includes its own set of commands.

Live Logic commands

Example:

```
mbCommand("liveLogic",  "trigger 1X");
```

Commands

trigger {XX, 1X, X1, 11, X0, 0X, 00, EXT}

mode {continuous, single}

timeout {0.1, 1, 2, 5, 10, none, <float number i.e 10e-3, 100>}

samples {32, 64, 128, 256, 512, 1024, 2048, <integer between 8 and 2048>}

logicFamily {CMOS_5V, CMOS_3.3V, CMOS_3V, CMOS_2.5V, CMOS_1.8V, TTL}

logicFamily CMOS_custom <decimal value between 0 and 5>

eventIn {none, <tool name>}

enventOut {start, trigger}

run

stop

getRunState

Returns {stopped, running}

getDVM

Returns channel1Voltage,channel2Voltage

getCaptureTime

Returns time

getNumberOfSamples

Returns samples

getTriggerIndex

Returns index

getTriggerState

Returns {0, 1, 2, 3}

getWaveform

Returns ch1Value;ch2Value;time,ch1Value;ch2Value;time$_1$ ...

getCH1Waveform

Returns ch1Value;time,ch1Value;time, ...

getCH2Waveform

Returns ch2Value;time,ch2Value;time, ...

getWaveformTime

Returns time1,time2,...,timeLast

getWaveformTimeFiltered {CH1,CH2} <filter time>

Returns time1,time2,...,timeLast

getWaveformDurations

Returns duration1,duration2,...,durationLast

zoomAll

saveWaveform <file name without extension> [<absolute directory path>]

loadWaveform <file name without extension> [<absolute directory path>]

Waveform Source commands

Example:

```
mbCommand("waveformSource", "mode continuous");
```

Commands

mode {continuous, onEvent, single}

selectWaveformType {sine, triangle, pulse, rc, battery, supply}

autoGenerate {true, false}

selectFastMode {true,false}

sineFrequency <floating point frequency>

sineOffset <floating point voltage>

sineAmplitude <floating point P-P voltage>

trianglePeriod <floating point number time>

triangleDutyCycle <integer 0...100>

pulseDuration <floating point number time>

pulsePeriod <floating point number time>

pulseInvert {true, false}

{triangle, pulse, rc} VL <floating point number voltage>

{triangle, pulse, rc} VH <floating point number voltage>

rcTimeConstant <floating point number time>

rcInvert {true, false}

batteryType {Alkiline, NiCd, LiO, NiMH}

batteryNumCells <decimal value>

batteryDischargeTime <floating point number time>

supplyVolts <decimal value>

eventIn {None, LS, PRA, PRB, LL, LA, SYS}

eventOut {start, end}

maintainOutputOnStop {true, false}

loadWaveform <file name without extension> [<absolute directory path>]

generateWaveform

run

stop

getRunState

Returns {stopped, running}

setValue <index> <voltage> <duration> // also sets mode to next

setMode <index> {stop|next|jumpA|jumpB|jumpC|event}

setJump {jumpA|jumpB|jumpC} {always|never|event|notEvent|wait|loop|loopInner} <target index> [<loop count>]

updateWaveformDisplay <waveform duration> [smooth {true,false}]

Logic Analyzer commands

Example:

```
mbCommand("logicAnalyzer", "trigger A value XX01");
```

Commands

trigger {A, B, C} value {0, 1, X} *1 to 9 digits of 0 or 1 or X*

trigger {A, B, C} type {none, value, duration, range, window, external, immediate}

trigger {A, B, C} minimumTime <float number time>

trigger {A, B, C} maximumTime <float number time>

triggerPosition {start, middle, end}

mode {continuous, single}

timeout {0.1, 1, 2, 5, 10, none} also *float number i.e 10e-3, 100*

displayOrder {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} *1 to 9 digits*

channelLabel {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} <string label>

samples {32, 64, 128, 256, 512, 1024, 2048} *or integer between 8 and 2048*

logicFamily {CMOS_5V, CMOS_3.3V, CMOS_3V, CMOS_2.5V, CMOS_1.8V, TTL}

logicFamily CMOS_custom <decimal threshold value>

eventIn { none, waveformSource, protocolA, protocolB, logicSource, liveLogic, system}

eventOut {start, trigger}

run

stop

getRunState

Returns {running, stopped}

getCaptureTime

Returns time

getNumberOfSamples

Returns samples

getTriggerIndex

Returns index

getTriggerState

Returns {0, 1, 2, 3}

getWaveform <channel list>

Returns value;time,value;time$_1$ …
Channel list is sequence of channels and order. For example "getWaveform 210" would combine channels 2..0 with channel 2 being the MSB and channel 0 being the LSB

getWaveformData <integer AND mask>

Returns value1,value2,...,valueLast

Each value is bit-wise AND'ed with mask value

getWaveformTime

Returns time1,time2,...,timeLast

getWaveformDurations

Returns duration1,duration2,...,durationLast

zoomAll

saveWaveform <file name without extension> [<absolute directory path>]

loadWaveform <file name without extension> [<absolute directory path>]

Logic Source commands

Example:

```
mbCommand("logicSource", "mode single");
```

Commands

mode {continuous, onEvent, single, custom, manual}

Vhigh < decimal voltage>

Vlow <decimal voltage>

eventIn {none, waveformSource, protocolA, protocolB, liveLogic, logicAnalyzer, system}

loadPattern <file name without extension> [<absolute directory path>]

loadPatternMemory

setValue <index> <vector> <duration>

setMode <index> {stop|next|jumpA|jumpB|jumpC|event}

setJump {jumpA|jumpB|jumpC} {always|never|event|notEvent|wait|loop|loopInner} <target index> [<loop count>]

updatePatternDisplay

run

stop

getRunState

Returns {stopped, running}

SPI Protocol commands

Example:

```
mbCommand("protocol", "commandSet spi");
mbCommand("protocol", "mode onEvent");
```

Note: the commandSet spi command must be the first command sent to the protocol tool to indicate the following commands are for the SPI protocol

Commands

commandSet spi

mode {slave, onEvent, single}

dataSize {8 bits, 16 bits, 24 bits, 32 bits}

bitOrder {msbFirst, lsbFirst}

clockRate {1 MHz, 5 MHz}

clockInvert {true, false}

skipJumpOnTrigger {true, false}

selectInputEnable {true, false}

inputThreshold <decimal value>

Vhigh <decimal value>

Vlow <decimal value>

eventIn {None, WS, PRA, PRB, LL, LA, SYS}

trig0 {0, 1, X} *1 to 8 digits*

trig1 {0, 1, X} *1 to 8 digits*

trig2 {0, 1, X} *1 to 8 digits*

trig3 {0, 1, X} *1 to 8 digits*

trigEnabled {true, false}

loadFile <file name without extension> [<directory path>]

run

stop

Trigger

I/O Protocol tool commands

Example:

```
mbCommand("protocol", "commandSet trigger");
mbCommand("protocol", "out1Source Live Logic");
```

Note: the commandSet trigger command must be the first command sent to the protocol tool to indicate the following commands are for the Trigger I/O Protocol

Commands

commandSet trigger

out1Source {None, WS, PRA, PRB, LS, LA, SYS}

out2Source {None, WS, PRA, PRB, LS, LA, SYS}

out3Source {None, WS, PRA, PRB, LS, LA, SYS}

out1Invert {true, false}

out2Invert {true, false}

out3Invert {true, false}

protASource {None, I-TRIG1, I-TRIG2, I-TRIG3}

protBSource {trig1, trig2, trig3, 1or2or3, 1and2and3, 1or2and3, 1or2andNot3, 1and2andNot3}

inputThreshold <decimal value>

Vhigh <decimal value>

Vlow <decimal value>

I2C Protocol commands

Example:

```
mbCommand("protocol", "commandSet i2c");
mbCommand("protocol", "mode master");
```

Note: the commandSet i2c command must be the first command sent to the protocol tool to indicate the following commands are for the I2C protocol

Commands

commandSet i2c

mode {master, slave}

eventIn {None, WS, PRA, PRB, LL, LA, SYS}

slaveAddress {x, X, <decimal address>}

boxLine {-1, <line number>}

masterClock {100KHz, 400KHz}

stopOnAddressNACK {true, false}

stopOnTimeout {true, false}

slaveReadData <d0> <d1> <d2> <d3> <d4> <d5> <d6> <d7>

slaveRunMode {single, stopOnEvent, Continuous}

showStartStop {true, false}

showRestart {true, false}

getReadData <decimal line number>

Read the captured data from a program line

setWriteData <decimal line number> {<d0> | <0x0} {<d1> | <0x1} … {<d7> | <0x7}

Set the write data to a program line

loadFile <file name without extension> [<directory path>]

run

stop

Picture Window commands

The pictureWindow acccepts image formats

pictureWindow Commands

loadFile name

loadFileFullPath path\name

loadSequence label image1 image2 transitionTome

size xPixels yPixels

title string

startTimer

timeoutImage imageName

Section 4 Complete Examples

The following example scripts are complete programs that may be copied to a text file and executed. They are designed to illustrate the use of the MBScript scripting language in practical terms.

Display Live Logic DVM Values

This script opens the Live Logic tool, reads the current input voltages, and displays their values. Before exiting a dialog is presented that pauses execution so that the values can be viewed.

```
// Open the Live Logic tool and stop acquisition
mbCommand("system", "openWindow liveLogic");
mbCommand("liveLogic", "stop");

// Open the message window
openMessageWindow();

do
{
	// Read the DVM values
	result = mbCommand("liveLogic", "getDVM");
	sleep(500);

	//  Break the csv response into a list of string
	listToArray(result, "," , cmp[], 0);


	// Display the measured values
	writeMessageLine("The CH1 voltage is: " + toStr(cmp[0]), 0);
	writeMessageLine("The CH2 voltage is: " + toStr(cmp[1]), 1);
}
while(!abortFlag);

closeMessageWindow();
```

Test Live Logic DVM values against limits

This script shows how to capture a Live Logic DVM reading and determine if it is within defined limits. The Waveform source is used to provide an input voltage for this test. To reproduce this test connect the Live Logic CH1 probe to the Waveform source output.

```
// Set up the Waveform Source to produce 3.0V DC
mbCommand("system", "openWindow waveformSource");
mbCommand("waveformSource", "selectWaveformType supply");
mbCommand("waveformSource", "supplyVolts 3.0");
mbCommand("waveformSource", "run");

// Open the Live Logic tool and stop acquisition
mbCommand("system", "openWindow liveLogic");
mbCommand("system", "positionWindow liveLogic 550 60");
mbCommand("liveLogic", "stop");

// Open the message window
openMessageWindow();
writeMessageLine("Connect CH1 probe to Waveform Source", 0);
writeMessageLine("Test Limits: 2.8V < V < 3.2V");

do
{
	// Read the DVM values
	result = mbCommand("liveLogic", "getDVM");
	sleep(500);

	//  Break the csv response into a list of string
	listToArray(result, "," , components[], 0);

	writeMessageLine("The CH1 voltage is: " + components[1], 2);

	ch1Voltage = components[1];
	if( (ch1Voltage > 3.2) || (ch1Voltage < 2.8) )
		writeMessageLine("Out of range", 3);
	else
		writeMessageLine("In range", 3);
}
while(!abortFlag);

closeMessageWindow();
```

Use time functions to periodically test and log results

This example uses the Live Logic DVM to take a measurement at 10 second intervals for 3 minutes. Results are logged to a local file named testResults.txt. The Waveform Source is used to provide a ramp voltage to make the readings more interesting. To execute this example connect the Live Logic CH1 probe to the Waveform Source.

```
// Open, configure, and start Waveform Source
mbCommand("system", "openWindow waveformSource");
mbCommand("waveformSource", "stop");
mbCommand("waveformSource", "selectWaveformType triangle");
mbCommand("waveformSource", "trianglePeriod 200");
mbCommand("waveformSource", "triangleDutyCycle 100");
mbCommand("waveformSource", "generateWaveform");
mbCommand("waveformSource", "run");
mbCommand("system", "positionWindow waveformSource 600 400");

// Open the Live Logic tool and stop acquisition
mbCommand("system", "openWindow liveLogic");
mbCommand("liveLogic", "stop");

// Open the message window
openMessageWindow();

fileName = "C:/Users/yourUser/Documents/testResults.txt";
file = openWriteFile(fileName);

// Add a header to the file
writeFileLine(file,"Periodic log of CH1 DVM reading");
writeFileLine(file,"Reading frequency is 10 seconds");
writeFileLine(file,"Readings are taken for 3 minutes");

writeMessageLine("Logging voltage values", 0);

// Values to track the time
startTime = osTick; // now
endTime = startTime + (3*60*1000); // 3 minutes from now
count = 1;
```

```
do
{
	// Read the DVM values
	result = mbCommand("liveLogic", "getDVM");

	//  Break the csv response into a list of string
	listToArray(result, "," , components[], 0);


	//  Update the messageWindow
	writeMessageLine(count, 1);
	count++;

	str = strFormat("date : ", components[1]);
	writeMessageLine(str, 2);

	// Add the value to the file
	writeFileLine(file, str);

	// Wait 10 seconds and watch for user "abort"
	local timer = osTick + 10000;
	repeat
	{
	}
	until( (osTick >= timer) || abortFlag);
}
while(!abortFlag && (osTick < endTime));

closeWriteFile();

// Wait for the user
textWindow("Press OK to exit");

closeMessageWindow();
```

Calculate the frequency of a captured waveform

In this example the frequency of a previously captured Live Logic waveform is evaluated to determine its frequency. It requires that more than 100 complete cycles of the wave were captured and assumes that the waveform is periodic. To execute this script connect Live Logic channel 1 to Logic source channel 0 and then run the script.

```
// Some useful functions
function system(cmd)
{
      return mbCommand("system", cmd);
}

function logicSource(cmd)
{
      return mbCommand("logicSource", cmd);
}

function liveLogic(cmd)
{
      return mbCommand("liveLogic", cmd);
}

// Open, configure, and start Waveform Source
system("openWindow logicSource");

logicSource("stop");
logicSource("setValue 0 0 1e-6");
logicSource("setMode 0 next");
logicSource("setValue 1 1 1e-6");
logicSource("setMode 1 jumpA");
logicSource("setJump jumpA always");
logicSource("updateWaveform");
logicSource("run");

system("positionWindow logicSource 600 400");

// Open the Live Logic tool and stop acquisition
system("openWindow liveLogic");

liveLogic("stop");
liveLogic("mode single");
liveLogic("samples 256");
```

```
// Open the message window
openMessageWindow();

writeMessageLine("Example Measure Frequency", 0);

// Get an acquisition with timeout if no trigger
tries = 0;
liveLogic("run");
do
{
      sleep(250);
      runState = liveLogic("getRunState");
      writeMessageLine("Live Logic is "+runState, 1);
}
while( (runState != "stopped") && (tries < 10) );

// Check that there was a trigger
if (tries > 10)
{
      textWindow("Acuqisition failed");
      liveLogic("stop");
      exit;
}

// Get the time between edges csv
edgeTimes = liveLogic("getWaveformTime");

//  Break the csv response into a list of string
n = listToArray(edgeTimes, "," , edgeTimeList[], 0);

if (n < 220)
{
      textWindow("Instufficient number of edges");
      exit;
end

// Average over 100 cycles (200 edges)
edgeTimeFirst = edgeTimeList[5];
edgeTimeLast = edgeTimeList[205];

// Calculate the frequency from 100 periods and display result
frequency = (1/(edgeTimeLast - edgeTimeFirst))*100;
writeMessageLine(strFormat("Frequency is {0:0.00}",frequency), 2);

logicSource("stop");
liveLogic("stop");
```

```
// Wait for the user
textWindow("Press OK to exit");

closeMessageWindow();
```