
Lab-LINK_{TM} for Windows

Part 3
Smart Script User Manual

Table of Content

Chapter1 Introduction

Features _____	1-1
Mode _____	1-2
Command Line _____	1-2

Chapter 2 Editor

User Interface _____	2-1
File Menu _____	2-2
Edit Menu _____	2-2
View Menu _____	2-4
Run Menu _____	2-5

Chapter 3 Applications

Run a Script When Project Starts _____	3-1
Activate a Script Using a SmartPanel Object _____	3-5
Using Loop _____	3-8
Tag Event _____	3-11
Important Issue Regarding Tag Event _____	3-13
Run a Script when a Panel is Opened _____	3-16
Communication Application _____	3-20

Chapter 4 Syntax

Overview _____	4-1
Command Line _____	4-1
Line Label _____	4-1
Character Set _____	4-2
Constants _____	4-3
Variables _____	4-5

Type Conversion	4-8
-----------------	-----

Expressions and Operations	4-8
----------------------------	-----

Chapter 5 Statements and Functions

How to use this chapter	5-1
-------------------------	-----

Keywords by Programming Task	5-2
------------------------------	-----

Statements and Functions	5-3
--------------------------	-----

ABS()	5-3
-------	-----

ACOS()	5-3
--------	-----

ALARM()	5-4
---------	-----

ALMGRP()	5-5
----------	-----

ALMPRI()	5-5
----------	-----

ALMTAG\$()	5-6
------------	-----

ASC()	5-7
-------	-----

ASIN()	5-7
--------	-----

ATAN()	5-8
--------	-----

BEEP	5-8
------	-----

CD	5-9
----	-----

CHOICE()	5-9
----------	-----

CHR\$()	5-10
---------	------

CLOSE	5-10
-------	------

COMMODE	5-11
---------	------

COMOPEN	5-12
---------	------

CONTINUE	5-13
----------	------

COPY	5-13
------	------

COS()	5-14
-------	------

COSH()	5-14
--------	------

CRC16()	5-15
---------	------

CRC32()	5-15
CREATE	5-16
DAY()	5-16
DATETIME\$()	5-17
DEL	5-17
DIM	5-18
DIR\$()	5-18
END	5-19
ERRID()	5-19
ERRORTAG()	5-20
EXEC	5-21
EXIT	5-22
EXP()	5-22
FAC()	5-23
FCHECK()	5-23
FILE\$()	5-24
FLEN()	5-24
FMBCD ()	5-25
FMDBL ()	5-25
FMFLT ()	5-26
FOR ... LOOP	5-27
Format()	5-29
FPOS()	5-33
FPRINT	5-33
GOSUB	5-34
GOTO	5-35
HOUR()	5-35

IDLE	5-36
IF ... ELSEIF ... ELSE ... ENDIF	5-37
INT()	5-38
ISTR\$()	5-38
IVAL()	5-39
LEFT\$()	5-39
LEN()	5-40
LN()	5-40
LOG()	5-41
LOWER\$()	5-41
LTRIM\$()	5-42
MAX()	5-42
MD	5-43
MESSAGE	5-43
MID\$()	5-44
MIN()	5-44
MINUTE()	5-45
MONTH()	5-45
MOVE	5-46
MSGBOARD	5-47
NERR()	5-48
NOW()	5-48
NOW\$()	5-49
OPEN	5-50
PASS	5-51
PLAY	5-52
RAND()	5-53

RD	5-53
READ	5-54
RETURN	5-55
RIGHT\$()	5-55
RSTERR	5-56
RTRIM\$()	5-56
SECOND()	5-56
SEEK	5-57
SETDIR	5-58
SHORTCUT	5-59
SHUTDOWN	5-60
SIN()	5-61
SINH()	5-61
SLEEP	5-62
SQRT()	5-62
STOP	5-62
STR\$()	5-63
STRING\$()	5-63
SUM08()	5-63
SWITCH ... CASE ... DEFAULT ... ENDSW	5-64
SYSINFO\$()	5-65
TAG()	5-66
TAN()	5-67
TANH()	5-67
TICK()	5-68
TIMER()	5-68
TOKEN	5-69

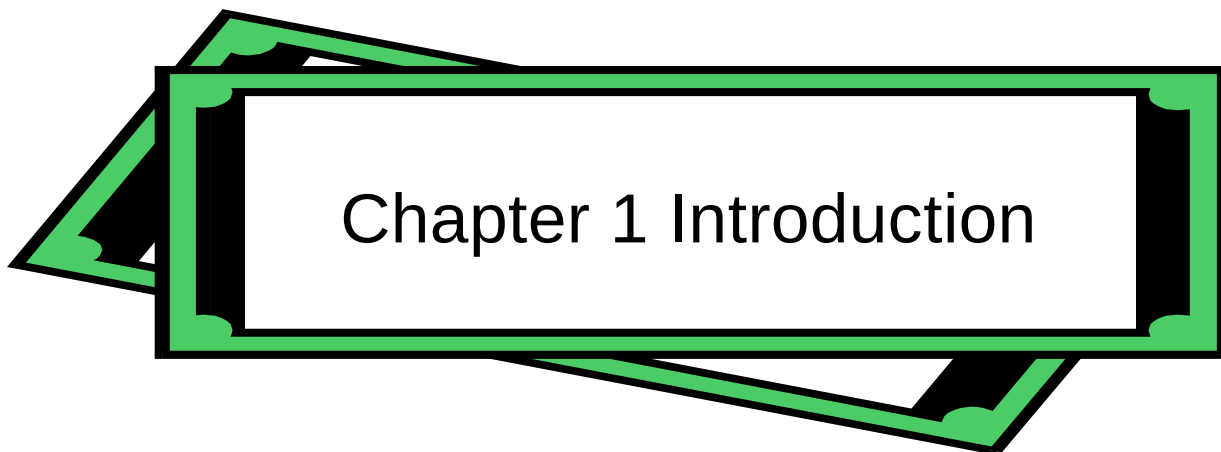
TONE	5-70
TRAPOFF	5-70
TRAPON	5-71
UPPER\$()	5-71
VAL()	5-71
WEEKDAY()	5-72
WHILE ... LOOP	5-72
WRITE	5-73
XOR08()	5-73
YEAR()	5-74

Appendix A Environment Limits

Appendix B Keywords

Appendix C Operator Precedence

Appendix D Error Codes



Chapter 1 Introduction

SmartScript is the built-in script language of **Lab-Link** for Windows. It is simple yet powerful and is closely integrated with Lab-LINK to provide a quick solution when complex logics are involved. With SmartScript, developers can write the control logic or math calculation they need without using a programming language and any other programming tools.

SmartScript provide the basic functions of a script language including data type, variable, loop, conditional branch, file access and IO communication. There are also abundant functions for various needs. A simple editor is also provided to help users edit and debug their script.

This manual will describe the features of SmartScript and the usage of the SmartScript editor. Detail explanation of the syntax, statements and function are also included for developers' reference.

Features

- Access to real time TAG data.
- Program flow control such as conditional branching, loop and subroutine.
- Tag event handling
- File access
- Capable of handling IO communication
- Rich math, string and time functions

Mode

SmartScript module can be executed in two modes: Edit and Run. When executed in Edit mode, the SmartScript editor will be activated to allow editing, compiling and debugging of script files. This mode is usually used in develop stage to write script logics. When executed in Run mode, SmartScript module will load the specified script file and executed the statements in the script.

The execution file of **SmartScript** module is CONTROL.EXE, Its command line parameters will be discussed in the following section.

Command Line

The command line of CONTROL.EXE can contain the following parameters:

CONTROL.EXE [ScriptFile [/R]]

Parameter	Description	Mode
None	Activate SmartScript editor and open a new unnamed script file.	Edit
ScriptFile	Activate SmartScript editor and load the specified script file.	Edit
ScriptFile /R	Run SmartScript and load the specified script file for execution.	Run

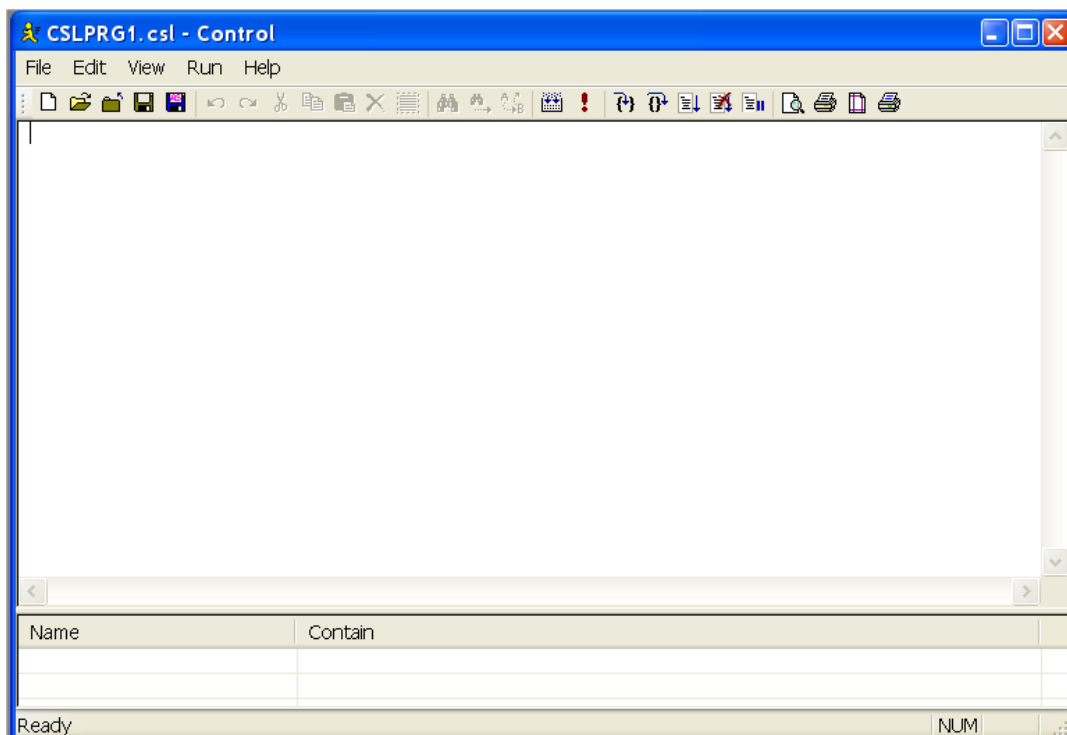
A script file has the extension name of .CSL and is stored in standard text file format.

Chapter 2 Editor

SmartScript is the easy to use developing environment of **SmartScript** module. It provides basic editing, syntax checking and debugging feature to help developer write and test their script logics.

User Interface

When **SmartScript** is executed in Edit mode, the screen below will appear:



If a script file name is included in its command line parameter, the file name will be shown on the title of **SmartScript** editor window. Below the title bar is the menu of the editor which providing File, Edit, View, Run, Help functions. These functions are described as follows.

File Menu

File menu includes theses functions:

New

Open a new script file.

Open

Open an existed script file.

Save

Save the currently editing file. If an untitled file is being edited, the Save As dialog will appear to request for a file name.

Save As

Save the currently editing file as a different name.

Print

Print the currently editing file. A dialog will appear to allow the selection of printer, range and copies.

Print Direct

Print the currently editing file directly without showing the Print dialog.

Print Preview

Show how the file will be printed on the screen.

Print Setup

Define the printing properties.

Exit

Close the editor.

Edit Menu

Edit menu provide these functions:

Undo

Undo the previous editing action.

Redo

Redo the previous undo editing action.

Cut

Cut the selected text.

Copy

Copy the selected text.

Paste

Paste the cut or copied text to the current location of the cursor.

Delete

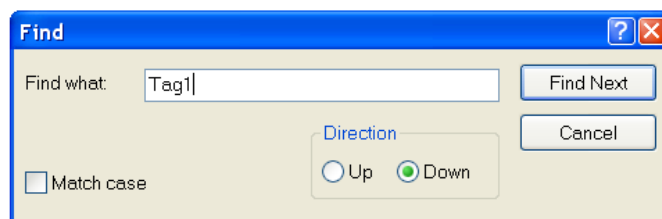
Delete the selected text.

Select All

Select text for editing.

Find

Move the cursor to the target text. A **Find** dialog will appear. Enter the text to be found in the **Find what** field and do either of the followings:



Find Next button

Move cursor to the next appearing of the target string and select the text.

Cancel 按鈕

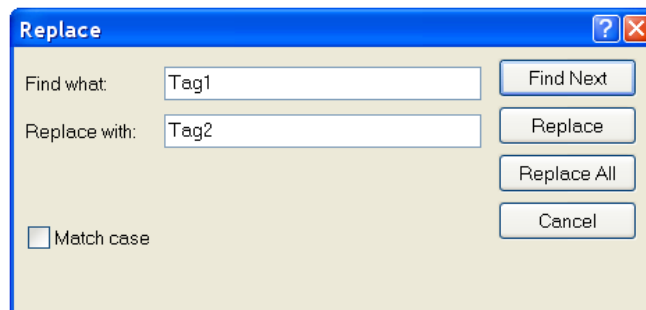
Close the dialog.

Find Next

Find the next appearance of the target string specified in the previous Find action.

Replace

Replace the target text with the specified string. A Replace dialog will appear. enter target and replace string in the **Find what** and **Replace with** respectively. Use the buttons to do the operations below:



Find Next button

Move cursor the next appearance of the target text.

Replace button

Replace the target text with the replace text.

Replace All button

Replace all target text in the file with the replace text.

Cancel button

Close the dialog.

View Menu

View menu provide the following functions:

Tool Bar— To show or hide the tool bar.

Status Bar— To show or hide the status bar.

Run Menu

Run menu provides these functions:

Compile — Check the syntax of the script. Since SmartScript is an interpreter, the compiling only does syntax checking and will not generate any object or execution file. If there is any syntax error, it will be shown in the window below. See Appendix D for error codes.

Execute— Run the script. The execution will continue until the last statement is executed or the execution is interrupted by the user. If any error occurs during the execution, an error message will appear to show error code and message. Execution of script can be interrupted by **Break** or **Stop** in the **Run** menu.

Step Into— Execute the script one statement at a time. Each the menu item is selected, a statement will be executed. After the execution of the statement, the script will stop to wait for user operation. User can either select **Stop Into** again to execute the next statement or select **Step Over** or **Go** to execute the remaining statements. Function key F8 can be used as a hot key of Step Into. For statements in a subroutine, **Step Into** will also execute one statement at a time.

Step Over— Similar to **Step Into** with the exception on the execution of statements in a subroutine. When **Step Over** is used to execute a subroutine call, it will finish all statements in the subroutine and return to the main program before stop for user operation.

Go— Continue running all the remaining statements in the script until the end is reached or any error occurs to interrupt its execution.

Break— Halt the execution of the script. **Step Into**, **Step Over** or **Go** can be used to continue the execution after **Break**.

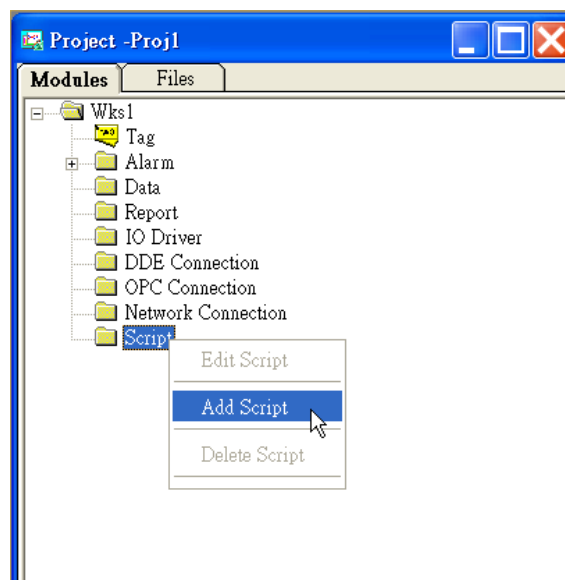
Stop— Stop the execution of the script. All opened files will be closed automatically when the script is stopped.

Chapter 3 Applications

Although **SmartScript** can be run independently, it is usually integrated with a Lab-LINK project to provide extra functions in real applications. Examples may include complex math computation or control logic, file reading and writing or even IO communication. This chapter will illustrate some possible applications using simple examples.

Run a Script When Project Starts

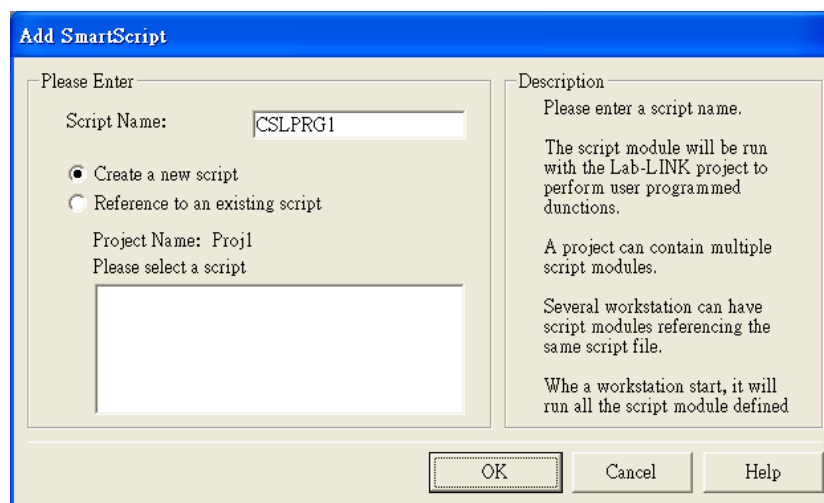
The simplest and also the most common way of running a script is to run it when a Lab-LINK project starts. Add a Script module in the project and the script will be run when the project starts. The script will be terminated when the project ends.



To add a Script into a project, right click on the **Script** node in the **Project** window of **PAM** and select **Add Script** from the popup menu. An **Add SmartScript** dialog will appear to allow the selection of **Script File** source. There are two selections available: **Create a new script** and

Reference to an existing script.

- ☐ **Create a new script** : Create a new blank script file in the CSL subfolder of the project and run **SmrtScript Editor** to load the empty for user to edit
- ☐ **Reference to an existing script**: The new script module will reference to an existed script file.



When a script is added and the project is regenerated, the script will be run when the project starts. The example script below will write a line of message recording the time the system starts into a text file. After the writing, the script will close the file and end.

■ Code

Line	Code
1	FileName\$="LABLINK.LOG"
2	OPEN 1, FileName\$, "W"
3	IF (!FCHECK(1))
4	CREATE 1, FileName\$
5	ENDIF
6	// Write data to the data file
7	SEEK 1, 0, "E"
8	FPRINT 1, YEAR(),"/",MONTH(),"/",DAY()," "
9	FPRINT 1, HOUR(),":", MINUTE(),":",SECOND()," "
10	FPRINT 1, "LABLINK STARTED\r\n"
11	CLOSE 1
12	END

Note: The Line numbers are just for reference and are not part of the script code

■ TAG and Variables

Name	Type	Description
FileName\$	String Variable	Store the name of the file name.

■ Script Description

Line 1: Specify the name of the recording file. The working directory of **SmartScript** is the system folder of **Lab-LINK** (Installation default is C:\LABLINK\SYSTEM4). Complete path should be included in the file name designation if the file does not reside in this system folder. For example, FileName\$="..\CSL\LABLINK.LOG" specify the file path at ..\CSL\LABLINK.LOG. (This is equivalent to the absolute path of C:\LABLINK4\CSL\LABLINK.LOG) Note that “\” is used to replace the “\” in path definition since “\” is a keyword in SmartScript.

Line 2: Open the file in Write mode and assign the file number as 1. The file number will be used in later file access statement until the file is closed.

Line 3-5: Check if the file is opened successfully. If not , create the specified file using CREATE statement.

Line 6: Any texts after “//” will be treated as remark and will not be interpreted and executed..

Line 7: Move the read-write pointer to the end of the file. The new text will be appended to the end of the file.

Line 8-10: Write system date/time and the message “LABLINK STARTED” System date and time are acquired using the time related functions YEAR()、MONTH()、DAY()、HOUR()、MINUTE() and SECOND(). “\r” and “\n” represent the control characters “Carriage Return” and “Line Feed” respectively and used to change line. The three statement can be combine into one by joining the text strings using “+” operators.

Line 11: Close the file. After the file is closed, the file number 1 is also released for use of other file access statements.

Line 12: End of the script. This SmartScript module will also end its execution.

Activate a Script Using a SmartPanel Object

SmartScript can be integrated with **SmartPanel** by running it with a **Executer** object. For example, a script may be run when user presses a button. The following example shows a script which is run when with a LOGIN button to record newly log-in user's identity into a file.

■ **SmartPanel** Object Setting

Object	Properties	TAG
Button	Caption: LOGIN Style: Push Button, Password Note: Pressing this button will set the value of the tag LOGIN to 1. Set Push Button to reset the tag to 0 when the button is release. Set Password to mandate password authentication when a user operate this button.	LOGIN
Executer	File: CONTROL.EXE Parameter: ..\PROJECT\PROJ1\CSL\LOGIN.CSL /R Note: Run the specified SmartScript when the value of LOGIN is 1.	LOGIN

Note: Script files are stored under the CSL subfolder of the project. The project name is PROJ1 and the script file name is "Login.csl "in this example.

■ Code

Line	Code
1	FileName\$="LABLINK.LOG"
2	OPEN 1, FileName\$, "W"
3	IF (!FCHECK(1))
4	CREATE 1, FileName\$
5	ENDIF
6	// Write data to the data file
7	SEEK 1, 0, "E"
8	FPRINT 1, YEAR(),"/",MONTH(),"/",DAY()," "
9	FPRINT 1, HOUR(),":", MINUTE(),":",SECOND()," "
10	FPRINT 1, {\$USER.\$}, " 登入\r\n"
11	CLOSE 1
12	END

Note: The Line numbers are jest for reference and are not part of the script code

■ TAG and Variables

Name	Type	Description
LOGIN	TAG	The tag used to trigger the execution of the script.
\$USER	System TAG	The system Tag whose message field indicate the name of the current logged-in user.
FileName\$	String Variable	The variable used to store the name of the file for log-in logging.

■ Script Description

Line 1: Specify the file name of the Log-in log file.

Line 2: Open the file in Write mode and assign the file number as 1. The file number will be used in later file access statement until the file is closed.

Line 3-5: Check if the file is opened successfully. If not , create the specified file using CREATE statement.

Line 6: Any texts after “//” will be treated as remark and will not be interpreted and executed..

Line 7: Move the read-write pointer to the end of the file. The new text will be appended to the end of the file.

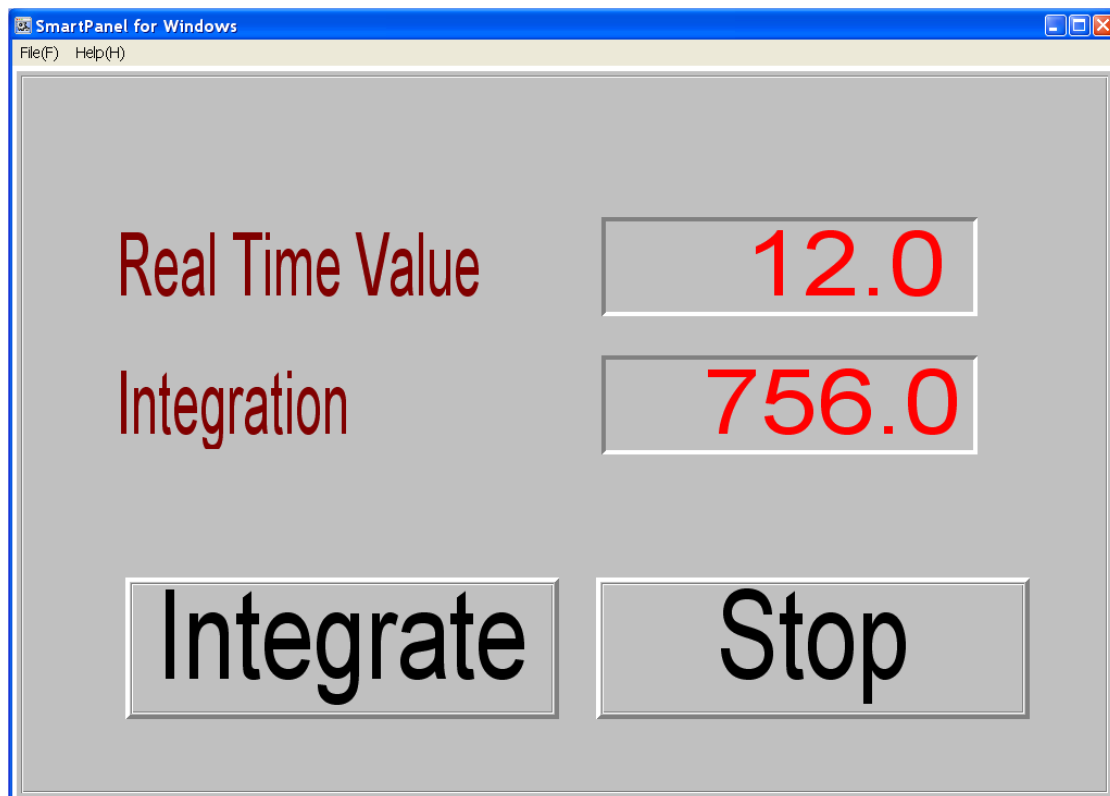
Line 8-10: Write system date/time and the user log-in message into the file. **{\$USER.\$}** is the message field of the system tag **\$USER** and its content is the name of the current log-in user.

Line 11: Close the file. After the file is closed, the file number 1 is also released for use of other file access statements.

Line 12: End of the script. This SmartScript module will also end its execution.

Using Loop

In the two previous examples, **SmartScript** end after all statements are executed. However, some applications may need the script to keep running. In this example, the script will enter a loop to do integration of Tag1 when user press **Integrate** button, The loop will continue until the user press **Stop** button after which the integration loop stop and the script ends.



■ SmartPanel Object Setting

Object	Properties	TAG	Description
Button	Caption: Integrate Style: Set Button	StartInt	Pressing the button will set the value of StartInt to 1.
Button	Caption: Stop Style: Reset Button	StartInt	Pressing the button will set the value of StartInt to 0
Executer	File: CONTROL.EXE Parameter: ..\PROJECT\PROJ1\CSL\INTEGRA L.CSL /R	StartInt	Run the specified script when the value of StartInt changes to 1

Object	Properties	TAG	Description
StaticText	Caption: Real Time Value		Show the text “Real time Value”
StaticText	Cation: Integration		Show the text “Integration”
Editor		Tag1	User can use this object to enter the value of Tag1
TextMeter		Int_Tag1	Show the integral value Int_Tag1

Note: Script files are stored under the CSL subfolder of the project. The project name is PROJ1 and the script file name is “INTEGRAL.csl” in this example.

■ Code

Line	Code
1	PrevTime%={\$TIME}
2	PrevValue={Tag1}
3	{Int_Tag1}=0
4	WHILE({StartInt})
5	{Int_Tag1}={Int_Tag1}+({Tag1}+PrevValue)*({\$TIME}-PrevTime%)/2
6	PrevTime%={\$TIME}
7	PrevValue={Tag1}
8	LOOP
9	END

Note: The Line numbers are jest for reference and are not part of the script code

■ TAG and Variables

Name	Type	Description
------	------	-------------

StartInt	TAG	Use to control the start and stop of integration
Tag1	TAG	Real time value
Int_Tag1	TAG	Integration value
\$TIME	System TAG	Its value will increase by 1 every second. It is used as a one second clock to trigger periodic event.
PrevTime%	Integer Variable	Used to store the previous value of \$TIME
PrevValue	Real Variable	Used to store the previous value of Tag1

■ Script Description

Line 1: Set the initial value of PrevTime%

Line 2: Set the initial value of PrevValue

Line 3: Set the integration value **Int_Tag1** to 0.

Line 4: Use **StartInt**=1 as the entering condition of a While loop. The loop stops and the script ends if **StartInt**=0. **StartInt** is also used as the tag to start running the script.

Line 5: Integrate **Tag1**. PrevValue is the value of Tag in previous integration calculation. $\{\$TIME\} - \text{PrevTime\%}$ is the time elapsed between the previous calculation and current calculation.

Line 6: Update the value of PrevTime%.

Line 7: Update the value of PrevValue.

Line 8: End of the loop.

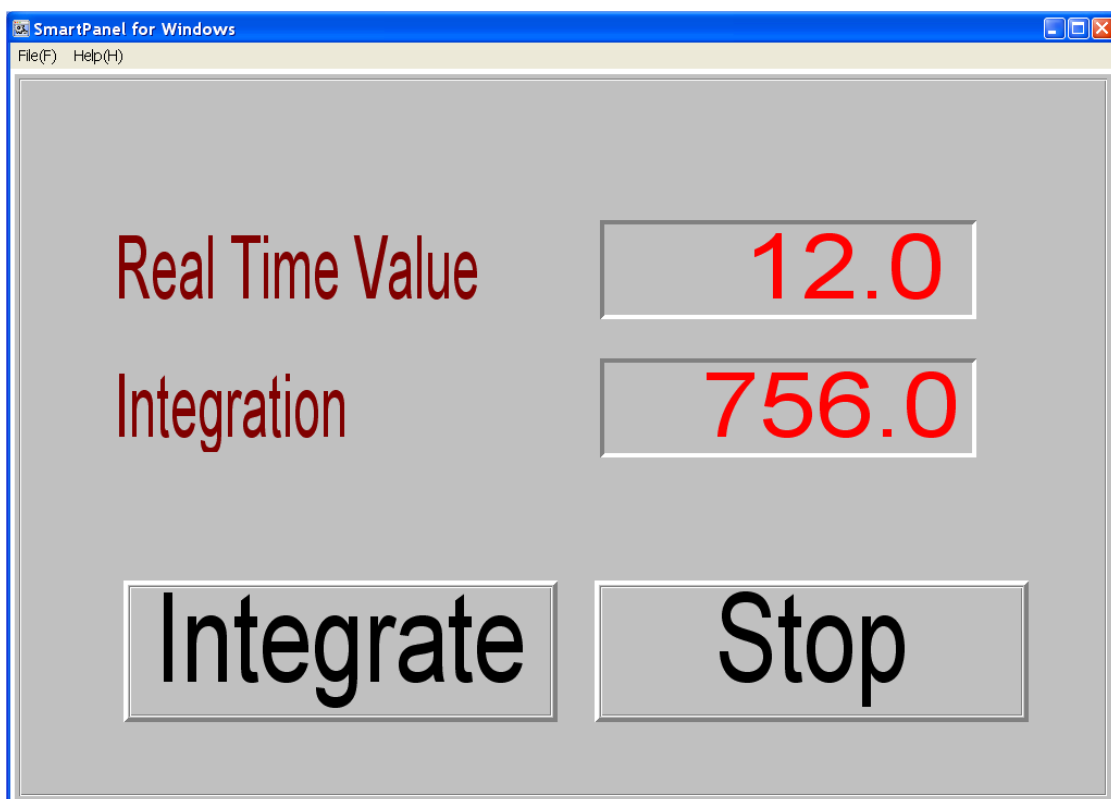
Line9: End of the script. This SmartScript module will also end its execution.

When the user press **Integrate** button and set the value of **StartInt** to 1, the script is executed. The script will first set the integration value to 0 and keep calculating the integration until the **Stop** button is pressed and **StartInt** is set to 0. If the user press Integrate button again, the script will be run again and reinitiate the integration calculation.

To keep the loop running when Lab-LINK is running, the condition of the WHILE loop can be set to 1 to start an infinite loop. A infinite loop will keep running until Lab-LINK runtime ends. However, the infinite will consume system resource.

Tag Event

In some applications, it is more efficient to use Tag events instead of loops. The example in previous section will calculate the integration value at every recurrence of the loop even though the values of **\$TIME** and **Tag1** do not change. The integration script is rewritten in this example to use tag event to calculate integration. The panel object, Tags and variables settings are the same as the previous section , and only the script will be discuss here.



■ Code

Line	Code
1	PrevTime%={\$TIME}
2	PrevValue={Tag1}
3	{Int_Tag1}=0

Line	Code
4	WHILE({StartInt})
5	LOOP
6	END
7	{\$TIME}:
8	{Int_Tag1}={Int_Tag1}+({Tag1}+PrevValue)*({\$TIME}-PrevTime%)/2
9	PrevTime%={\$TIME}
10	PrevValue={Tag1}
11	RETURN

Note: The Line numbers are jest for reference and are not part of the script code

■ Script Description

Line 1: Set the initial value of PrevTime%.

Line 2: Set the initial value of PrevValue.

Line 3: Reset the integration value **Int_Tag1** to 0.

Line 4: Use **StartInt**=1 as entering condition to start a While loop. If **StartInt**=0, the loop will stop and the script ends. **StartInt** is also used in the Executer object in panel to start the running of the script.

Line 5: End of loop.

Line 6: End of the main program and the script as well.

Line 7: Start of tag event of **\$TIME**. By using the Tag as its label, the subroutine below will be executed when the value of **\$TIME** changes.

Line 8: Integrate **Tag1**. PrevValue is the value of Tag in previous integration calculation. **{\$TIME}**-PrevTime% is the time elapsed between the previous calculation and current calculation.

Line 9: Update the value o PrevTime%.

Line 10: Update the value of PrevValue.

Line 11: End of the subroutine. Return to when it was interrupted in the main program and continue,

Similar to the example in the previous section, when the user press **Integrate** button and set the value of **StartInt** to 1, the script is executed. However, the script will enter the empty loop after the initial setting statement in the beginning. The loop itself contains no statement. When the value of **\$TIME** changes (once every second), the tag event subroutine with **{ \$TIME }**: will be run to calculate the integration. Since the calculation statement will be executed only once per second, it is more efficient. To keep the loop running when Lab-LINK is running, the condition of the WHILE loop can be set to 1 to start an infinite loop. A infinite loop will keep running until Lab-LINK runtime ends. A more efficient alternative than the infinite loop is to use the IDLE statement which will not consume system resource. See chapter 5 for syntax and usage of these statements.

Important Issue Regarding Tag Event

Although Tag event is a very useful feature in **SmartScript**, programmers must be very careful to control its behaviors due to its event-driven nature. In general, a Tag event subroutine is a series of **SmartScript** statements that start with a line label which is a Tag name and end with the **RETURN** statement. The code below is a typical example:

```
...                // Other statements
{Tag1}:            // This is the Tag name line label
...                // Script code of Tag1 event subroutine
RETURN
```

As shown below, there can be more than one Tag line label for each event subroutine. Data change of any of the Tags will trigger the execution of the event subroutine.

```
...                // Other statements
{Tag1}:            // This is the first Tag name line label
{Tag2}:            // This is the second Tag name line label
{Tag3}:            // This is the third Tag name line label
...                // Script code of Tag1, Tag2 and Tag3 event subroutine
RETURN
```

When the data of any of the event subroutine label Tag change, **SmartScript** logic execution will jump to the Tag label and execute the statements after the label until the RETURN statement is reached. Script logic will then return to the last statement before the event occurs and continue

with the execution of the remaining statements. The data changes triggering a Tag event include the changes of value, message, data, time and status field of the Tag data.

Since the change of Tag data will trigger a new event, if there is a new event triggered before the finish of a previous even, unpredictable result may occurs if the script logic is not properly controlled. There are two commands, namely **TRAPOFF** and **TRAPON**, designed to help programmers event triggering. **TRAPOFF** is used to suspend event triggering by setting a **TRAPOFF** flag while **TRAPON** is used to do the opposite by resetting **TRAPOFF** flag to allow event triggering. When Tag event triggering is suspended, *SmartScript* will store all new Tag events occur during the period in the **Event Queue**. These buffered events will be executed after the event triggering is enabled again.

To help programmer understand how to use **TRAPOFF** and **TRAPON** in writing Tag event subroutines, the process flow of *SmartScript* event subroutines execution is described as follows.

- During the execution of *SmartScript*, if any Tag event is triggered and the corresponding event subroutine is not in the **Event Queue**, the new Tag event subroutine will be added into the **Event Queue**.
- After the execution of any *SmartScript* statement, the following examination and process will be done:
 - Check **TRAPOFF** flag. If the flag is not set, check if there is any Tag event subroutine in the **Event Queue** waiting for execution. If there is any event in the queue, retrieve the event subroutine from the queue and move the next statement pointer of *SmartScript* to the event subroutine.
 - If the previous executed statement is **TRAPON**, reset the **TRAPOFF** flag.
 - Continue with next statement.
- Based on description above, **TRAPOFF** and **TRAPON** can be used as shown in the example below to guarantee that the execution of an event subroutine will not be interrupted by any other new event.

```

...                // Other statements
{Tag1}:
TRAPOFF            // TRAPOFF flag is set.
                   // Event triggering is disabled
...                // Script code of Tag1 event subroutine
TRAPON             // TRAPOFF flag is reset.
RETURN             // Event triggering is enabled

```

- When Tag1 event is triggered, the first statement executed is **TRAPOFF** (the line label is not considered as a statement). **TRAPOFF** will set **TRAPOFF** flag immediately and all new event will not be triggered but stored in the **Event**

Queue.

- When the execution reaches the **TRAPON** statement, since *SmartScript* will first check the **TRAPOFF** flag (it is not reset yet at this moment) before resetting the **TRAPOFF** flag, it will not execute any new event right away. Instead, *SamrtScript* will wait after the execution of the next statement, **RETURN** in this case, to find that **TRAPOFF** flag is reset and jump to the new event subroutine. By using these two commands, possibilities of interruption by any new event during the execution of the event subroutine can be excluded.
- The following problems may occur if **TRAPOFF** and **TRAPON** statement are not used:
 - During the execution of a Tag event subroutine, it is possible that script logic execution can be switched to a new event. This may result in problems such as the difficulties in determining which event will be handled first and variable values may be changed by new events.
 - If an event subroutine is not completely executed before switching to a new event, the unfinished subroutine will be stored in a stack to wait for execution after the returning from the new event. In the case of large number of events triggered in a short period of time, many unfinished event subroutines will be accumulated in the stack. This may cause the system error of “Stack overflow” eventually and end the execution of the *SmartScript* module.
- Besides the discussion above, there are some more issues need to be considered when using Tag event subroutines:
 - During the execution of a Tag event subroutine, it is always possible that Tag data referenced in the subroutine can be modified by other **Lab-LINK** modules. Programmer should make no assumption that Tag data will remain unchanged during the execution of the subroutine. To lessen the effect of this characteristic, it is recommended that Tag values can be stored in variables at the beginning of the event subroutine. Reference of Tag data can then be replaced by the variable values. However, please note that during the Tag data assignment to the variables, Tag data can still change during this relative short period of time.
 - For each Tag event subroutine, only one record is kept in the **Event Queue**. Therefore, during the period that **TRAPOFF** flag is set.
 - If multiple Tag line labels share the same event subroutine, multiple instances of data changes of these Tags will only add the event routine once into the **Event Queue**. The position of the event subroutine in the **Event Queue** is determined when it is first added into the queue.
 - If an event Tag changes data more than once, the event subroutine will only

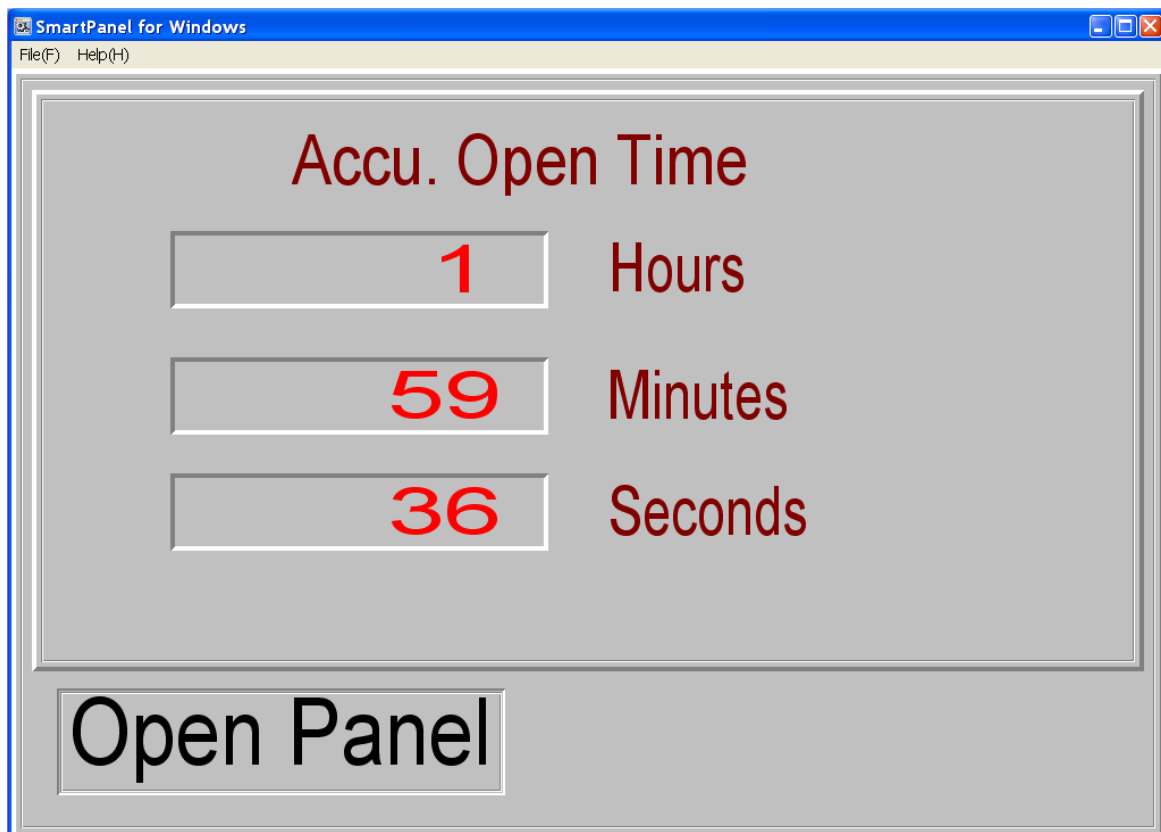
be added once into the **Event Queue**. The position of the event subroutine in the **Event Queue** is determined when it is first added into the queue.

- Due to the capacity of the **Event Queue** (512 events), the time required to run an event subroutine should be kept as short as possible. Otherwise, if more than 512 events are triggered during the **TRAPOFF** period, the events occur after the queue is full will be ignored.

Run a Script when a Panel is Opened

Some applications may need a script to be run only when a panel is opened. This can be achieved by using a similar loop as the previous example but with proper condition. This is useful when the script is only necessary for this display of the panel and should be close to prevent waste of system time when the panel is closed.

In the following example, when user press “Open Panel” button to open a panel file, a script is run to accumulate the time the panel is opened.



■ SmartPanel Object Setting

Two panel files are used in this example: WKS1.PNL(The root panel) and SUBPANEL.PNL. Their settings are as follows:

WKS1.PNL

Object	Properties	TAG	Description
Button	Caption: Open Panel	OpenPnl	Pressing the button will set the value of OpenPnl to 1.
PopMacro	X: 0 Y: 0 Width: 10000 Height: 8500 Panel File: ~1\subpanel.pnl Style: Child Window	OpenPnl	When OpenPnl is 1, a panel file subpanel.pnl will be opened. When OpenPnl is 0, the panel will be closed.

SUBPANEL.PNL

Object	Properties	TAG	Description
StaticText	Caption: Accu. Open Time		Show the text "Accu. Open Time"
StaticText	Caption: Hours		Show the text "Hours"
StaticText	Caption: Minutes		Show the text "Minutes"
StaticText	Caption: Seconds		Show the text "Seconds"
TextMeter	Dec. Digit: 0	OpenHour	Show the hour part of the accumulated time
TextMeter	Dec. Digit: 0	OpenMin	Show the minute part of the accumulated time
TextMeter	Dec. Digit: 0	OpenSec	Show the second part of the accumulated time
TextMeter	File: CONTROL.EXE Parameters: ..\PROJECT\PROJ1\CSL\OPENTIME.CSL /R	OpenPnl	Run the specified script when OpenPnl is 1

Note: Script files are stored under the CSL subfolder of the project. The project name is PROJ1 and the script file name is "OPENTIME.csl" in this example.

■ TAG and Variables

Name	Type	Description
OpenPnl	TAG	Used to control the open of SUBPANEL.PNL
OpenHour	TAG	The hour part of the accumulated open time.
OpenMin	TAG	The minute part of the accumulated open time.
OpenSec	TAG	The second part of the accumulated open time.
\$TIME	System TAG	The system tag whose value will increase by 1 per second. Used as one second clock.
OpenTime%	Integer Variable	Used to store the total accumulation open time in seconds.
PrevTime%	Integer Variable	Used to store the previous value of \$TIME

■ Code

Line	Code
1	OpenTime%={OpenHour}*3600+{OpenMin}*60+{OpenSec}
2	PrevTime%={\$TIME}
3	WHILE({OpenPnl})
4	LOOP
5	END
6	{\$TIME}:
7	OpenTime%=OpenTime%+{\$TIME}-PrevTime%
8	{OpenHour}=OpenTime%/3600
9	{OpenMin}=(OpenTime%\3600)/60
10	{OpenSec}=(OpenTime%\3600)\60
11	PrevTime%={\$TIME}
12	RETURN

Note: The Line numbers are just for reference and are not part of the script code

■ Script Description

Line 1: Calculate the total seconds and store it to OpenTime%.

Line 2: Set initial value of PrevValue%.

Line 3: Use **OpenPnl=1** as entering condition to start a While loop. When SUBPANEL.PNL is closed and caused **OpenPnl=0**, the loop is stopped and the script ends. **OpenPnl** is also the tag used to open and close the panel SUBPANEL.PNL.

Line 4: end of loop.

Line 5: End of the main program and the script as well.

Line 6: Start of tag event of **\$TIME**. By using the Tag as its label, the subroutine below will be executed when the value of **\$TIME** changes.

Line 7: Accumulate the open time of the panel SUBPANEL.PNL. PrevValue% is the value of Tag1 at previous loop recurrence. **{ \$TIME }-PrevTime%** is the time elapsed since last loop recurrence. The statement add the elapsed time to the accumulated time.

Line 8: Calculate the hour part of the accumulated time.

Line 9: Calculate the minute part of the accumulated time.

Line 10: Calculate the second part of the accumulated time.

Line 11: Update the value of PrevTime%.

Line 12: End of the event subroutine and return to the main program.

Communication Application

SmartScript can also be used to write IO communication program to support the communication with IO devices. The example below illustrate the communication with an instrument which use a very simple ASCII protocol. A command “#S00\r” will be sent from the PC to the instrument for reading its data. The returned data from the instrument is always with the fixed length of 11 ASCII character. Data reside at the fourth to the eighth character. The example script will poll the instrument at one second interval, acquire the value part from the returned data and save the data to a tag named **Data-01**. The script can be set to run when the Lab-LINK project similar to the first example of this chapter.

■ TAG and variables

Name	Type	Description
Data-01	TAG	Used to store the data value.
C\$	String Variable	Used to store the command string for reading data.
R\$	String Variable	Used to store the string returned by the instrument.
V\$	String Variable	Used to store the value parsed from the returned string.
TickBeg	Variable	Used to store the previous TICK value. TICK is a time function in mini seconds.
CommLoop	Label	Label for the GOTO statement

■ Code

Line	Code
1	C\$ = "#S00\r"
2	COMOPEN 1, "COM1:9600,N,8,1", 1024, 256
3	IF (! FCHECK(1))
4	MESSAGE " Test Program", "Can not open communication port"
5	STOP
6	ENDIF

Line	Code
7	CommLoop:
8	WRITE 1, C\$
9	WHILE (FLEN(1) < 11)
10	LOOP
11	READ 1, R\$, 11
12	V\$ = MID\$(R\$, 4, 5)
13	{Data-01} = VAL(V\$)
14	TickBeg = TICK()
15	WHILE (TICK()-TickBeg < 1000)
16	LOOP
17	GOTO CommLoop:
18	END

Note: The Line numbers are just for reference and are not part of the script code

■ Script Description

Line 1: Store the data reading command string to the variable C\$. “\r” is the Carriage Return control character.

Line 2: Open communication port COM1 with the communication parameters: 9600, N, 8,1. Set the capacity of Interrupt-driven receive-queue to 1024 characters, and the capacity of Interrupt-driven transmit-queue to 256 characters.

Line 3-6: Handle communication port open failure. Show a message on the screen and end the script after user acknowledgement.

Line 7-17: The GOTO loop to handle communication.

Line 8: Send the command string C\$ to read data.

Line 9-10: Wait until at least 11 characters of return data received in the Interrupt-driven receive-queue.

Line 11: Take the first 11 characters in the Interrupt-driven receive-queue and store it to R\$.

Line 12: Take the fourth to the eighth characters from R\$, the value part, and store it to V\$.

Line 13: Use VAL function to convert V\$ from a string to a number and assign the number to the value of tag **Data-01**.

Line 14-16: Wait 1000tick (1000 msec or 1second).

Line 17: Go back to the beginning of the GOTO loop, the label CommLoop, to repeat the operation.

Line 18: End of script.

Chapter 4 Syntax

Overview

The **SmartScript** Module provides users with a complete programming environment, allowing communication with other modules of **Lab-LINK** for Windows, and acts as a built in program development tool. **SmartScript** Module contains most of the commands and functions found in traditional BASIC or C but also includes additional statements, which deal with specific features of **Lab-LINK**.

SmartScript contains families of commands allowing maintenance of files, control of asynchronous devices, and playing of music and sound. An editor with program debugging capability is provided to simplify software development and code modification.

Command Line

CONTROL.EXE, located in the system folder of **Lab-LINK**, is the execution file of **SmartScript**. Its command line can include the parameters:

CONTROL.EXE [*source-file* [/R]]

Option	Description
<i>source-file</i>	Name of the script file loaded when SmartScript starts.
<i>source-file</i> /R	Causes SmartScript to load and run a program file.

Line Label

Every program line may begin with a line label. Line labels are used as references of branching. The characters allowed in a line labels are alphanumerical characters and underscore. Line label must be within the length between 1 to 16, and the first character must be a letter. Some special type declaration characters are also allowed—see below.

A line label may be a reserved word or may contain an embedded reserved word, though this is not recommended. Reserved words include all commands, statements, function names and operator names.

When a line label enclosed in the curly brackets, it is a special kind of line label called TAG label. In this case the label name within the curly brackets must be a TAG name. Whenever the value of the corresponding TAG changed, the original program flow will be suspended, and the statement block following the label will be executed. After execution of this statement block, the original program flow will be resumed.

Character Set

SmartScript Module character set includes alphabetic characters (A-Z, a-z), numeric characters (0-9 and A-F or a-f for hexadecimal numbers), and special characters. Some characters have special meanings in **SmartScript** Module:

Data-Type Suffixes

- % Integer
- \$ String
- { } TAG

Mathematical Operators

- . Decimal point
- + Plus sign
- Minus sign
- * Multiplication symbol
- / Division symbol (slash)
- \ Integer division symbol (backslash)
- ^ Exponentiation symbol (up arrow or caret)

Relational operators

- > Greater than
- < Less than
- >= Greater than or equal to
- <= Less than or equal to
- == Equal to

!= Not Equal

Logical operators

! Logic NOT

& Logic AND

| Logic OR

Bitwise operators

~ Invert

<< Shift left

>> Shift right

^< Rotate left

>^ Rotate right

Special

= Assignment symbol

: Line label suffix

@ Break point mark

, Parameter delimiter

=> Then symbol

// Comment line

Constants

Constants are the actual values **SmartScript** Module users during execution. There are two types of constants: string and numeric.

String Constant

A string constant is a sequence of characters enclosed in double quotation marks. Examples of string constants:

“\$25,000.00”

“This is a test.”

The characters enclosed in double quotation mark may be alphanumeric or special characters. The escape sequences allow you to use a sequence of characters to represent special characters. Escape sequences are listed below.

Sequence	Name
\n	New Line
\r	Carriage Return
\t	Horizontal Tab
\"	Double Quotation Mark
\\	Backslash
\bnnnnnnnn \Bnnnnnnnn	ASCII character in binary notation
\onnn \Onnn \0nnn	ASCII character in octal notation
\dnnn \Dnnn \nnn	ASCII character in decimal notation
\xnn \Xnn \hnn \Hnn	ASCII character in hexadecimal notation

Numeric Constant

Numeric constants are positive or negative numbers. Numeric constants in SmartScript Module cannot contain commas. There are seven types of numeric constants:

1. Integer Constants
Whole numbers between -2147483648 and +2147483647. Integer constants do not have decimal points.
2. Fixed Point Constants
Positive or negative real numbers, i.e., numbers that contain decimal points.
3. Floating Point Constants
Positive or negative numbers represented in exponential form (similar to scientific notation). A floating-point constant consists of an optionally signed integer or fixed-point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10^{-308} to 10^{+308} , and contains up to 16 significant digits.
Examples:
 $476.983E-6 = 0.000476983$

523E3 = 523000

A special keyword “PI” presents the floating-point constant π (3.141592653589793).

4. Decimal Constants

Decimal numbers with no prefix or the prefix 0d or 0D.

Examples:

1024

0d13

0D27

5. Hexadecimal Constants

Hexadecimal numbers with the prefix 0x, 0X, 0h or 0H.

Examples:

0x41FF

0X0D

0hE6

0H1A0

6. Octal Constants

Octal numbers with the prefix 0o or 0O.

Examples:

0o10

0O361

7. Binary Constants

Binary numbers with the prefix 0b or 0B.

Examples:

0b1010

0B11000011

Variables

Variables are names used to represent values that are used in a **SmartScript** Module Program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

Variable Names and Declaration Characters

The characters allowed in a variable name are letters and numbers, and the underscore. Variable names must be in the length 1 to 16, and the first character must be a letter. Special type declaration characters are also allowed—see below.

A variable name may not be a reserved word, but may contain an embedded reserved word. Reserved words include all commands, statements, function names and operator names.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign (\$) as the last character. For example: A\$ = "Hello the world". The dollar sign is a variable type declaration character; that is, it "declares" that the variable will represent a string.

Numeric variable names may declare integer or real (double precision) values. The type declaration characters for these variable names are as follows:

% Integer variable

The default type for a numeric variable name is real.

Examples of variable names follow.

N% declares an integer value

LIMIT declares a real value

MSG\$ declares a string value

Tag Variables

In **SmartScript**, a special kind of variables called TAG variables is used to access and manipulate TAG data. TAG variables should be named as follows:

{tag-name} *tag-name* is the name of a TAG. If the TAG does not exist, it will be created. **{tag-name}** can be used just like a regular numeric variable (double precision), and it presents the value field of the TAG. Any modification made to the TAG variable will modify the value field of *tag-name* as well.

{tag-name.\$} *tag-name* is the name of a TAG. If the TAG does not exist, it will be created. **{tag-name.\$}** can be used just like a *regular* string variable, and it presents the message field of the TAG. Any modification made to the TAG variable will modify the message field of *tag-name* as well.

{tag-name.t} *tag-name* is the name of a TAG. The variable can also be denoted as **{tag-name.T}**. If the TAG does not exist, it will be created. **{tag-name.t}** can be used just like a *regular* numeric variable. Its integer part represents the date field of the TAG, and its fraction part represents the time field of the TAG. Both integer and fraction parts are of the same unit of days. Any modification made to the TAG date/time variable will modify the date and time field of *tag-name* as well.

Note: When the value or message field of a TAG is modified, the system will update the date and time fields of the TAG automatically to indicate the date and time the TAG data is changed.

`{tag-name.s}` *tag-name* is the name of a TAG. The variable can also be denoted as `{tag-name.S}`. If the TAG does not exist, it will be created. `{tag-name.s}` can be used just like a regular numeric variable but its value represents the 16 bits binary encoded status field of the tag. Definition of the status bits are shown in the table below. Bit operators `>>` (Right shift) and `AND` can be used to acquire the specific status bit(s) of the tag status variable.

Bits	Category	Meaning of Values	Right Shift >>	AND MASK
0~2	Input Status	0: Unknown 1: Uncertain 2: Offline 3: Online	0	0x0007
3	Output Status	0: Success 1: Fault	3	0x0008
4~6	Alarm Status	0: Normal 1: HI alarm 2: LO alarm 3: HH alarm 4: LL alarm	4	0x0070
7	Acknowledge Status	0: Unacknowledged 1: Acknowledged	7	0x0080
8~16	Reserved			

Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example `V(6)` would reference a value in a one-dimension array, `M(2, 7)` would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an array is 3. The maximum number of elements of all dimensions is 8,192.

Memory Space Requirements

Variables		Arrays	Strings
Integer	4 Bytes	Integer 4 Bytes per element	3 bytes overhead plus the present contents of the string
Real	8 Bytes	Real 8 Bytes per element	

Type Conversion

When necessary, **SmartScript** Module will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)

Example:

`N% = 32.64`

After this statement was executed, the value of N% is 32.

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision; i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

Examples:

`A = 15.0 / 2`

The arithmetic equation is performed in double precision and the result was returned in A as a double precision value. After this statement was executed, the value of A is 7.5.

`A = 9 / 5`

The arithmetic was performed in integer precision and the result was returned in A as a double precision value. After this statement was executed, the value of A is 1.0.

3. Logical operators convert their operands to integers and return an integer result.

Example:

`N% = 2.5 XOR 1`

After this statement was executed, the value of N% is 3.

When a floating-point value is converted to an integer, the fractional portion is rounded.

Example:

`N% = 6.67`

`M% = -6.67`

After this statement was executed, the value of N% is 6 and M% is -6.

Expressions and Operations

An expression may be a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by

SmartScript Module may be divided into four categories:

1. Arithmetic
2. Relational
3. Logical
4. Bitwise

Arithmetic Operators

The arithmetic operators, in order of precedence, are:

Operator	Operation	Sample Expression
-	Negation	-X
^	Exponentiation	X ^ Y
*	Multiplication	X * Y
/	Division	X / Y
\	Integer Modulus	X \ Y
+	Addition	X + Y
-	Subtraction	X - Y

To change the order, in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their **SmartScript** Module counterparts.

Algebraic Expression	SmartScript Module Expression
$X + 2Y$	$X + 2 * Y$
$X - \frac{Y}{Z}$	$X - Y / Z$
$\frac{X \cdot Y}{Z}$	$X * Y / Z$
$\frac{X + Y}{Z}$	$(X + Y) / Z$
$(X^2)^Y$	$X ^ 2 ^ Y$
X^{Y^Z}	$X ^ (Y ^ Z)$
$X \cdot -Y$	$X * -Y$

Integer Modulus Arithmetic

Integer modulus arithmetic is denoted by the backslash (\). The operands are rounded to integers before the modulus is performed, and it gives the integer value that is the remainder of an integer division.

Example:

```
N% = 8.9 \ 3
```

After this statement was executed, the value of N% is 2.

Division by Zero

If, during the evaluation of an expression, a division by zero is encountered, the “Division by Zero !!” error message is displayed, and execution stops.

Relational Operators

Relational operators are used to compare two values. The result of the comparison is either “True” (one) or “False” (zero). This result may then be used to make a decision regarding program flow. The relational operators, in order of precedence, are:

Operator	Operation	Sample Expression
>	Greater than	$X > Y$
<	Less than	$X < Y$
>=	Greater than or equal to	$X \geq Y$
<=	Less than or equal to	$X \leq Y$
==	Equality	$X == Y$
!=	Inequality	$X != Y$

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

```
X + Y < (Z + 5) / W
```

is true if the value of X plus Y is less than the value of Z+5 divided by W. More examples:

```
B% = LIMIT > 95.0
```

```
IF (B%) => GOSUB HiAlarm:
```

```
IF (SIN(X) < 0) => GOTO Label1:
```

```
IF (N% \ 2 != 0) => L% = L% + 1
```

```
IF (A$ == "EXIT") => STOP
```

Logical Operators

Logical operators perform tests on multiple relations or Boolean operations. The logical operator returns an result which is either “True” (one) or “False” (zero). The logical operators, in order of precedence, are:

Operator	Operation	Sample Expression
!	Logic NOT	! (X > Y)
&	Logic AND	X > 10.5 & Y < 21.3
	Logic OR	N% < 0x30 N% > 0x39

The outcome of a logical operation is determined as shown in the following table.

! (Logical NOT)	
X	! X
True (not zero)	False (zero)
False (zero)	True (one)

& (Logical AND)		
X	Y	X & Y
True (not zero)	True (not zero)	True (one)
True (not zero)	False (zero)	False (zero)
False (zero)	True (not zero)	False (zero)
False (zero)	False (zero)	False (zero)

(Logical OR)		
X	Y	X Y
True (not zero)	True (not zero)	True (one)
True (not zero)	False (zero)	True (one)
False (zero)	True (not zero)	True (one)
False (zero)	False (zero)	False (zero)

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF statement).

Examples:

IF (A >= 0) & (A <= 100) => GOTO Label1:

B% = (N% < 48) | (N% > 57)

IF ! B% => GOTO Label2:

Bitwise Operators

Bitwise operators perform bit manipulation and return a bitwise result. The operands are rounded to integers before the bit manipulation is performed, and it returns the integer value that is the bitwise result of the operation. The bitwise operators, in order of precedence, are:

Operator	Operation	Sample Expression
~	Invert	~ N%
<<	Shift left	X << 3
>>	Shift right	X >> 8
^<	Rotate left	X ^< 1
>^	Rotate right	X >^ 2
AND	AND	X AND 0x00FF
XOR	XOR	X XOR 0b10101010
OR	OR	X OR Y

The outcome of a bitwise operation is determined as shown in the following table.

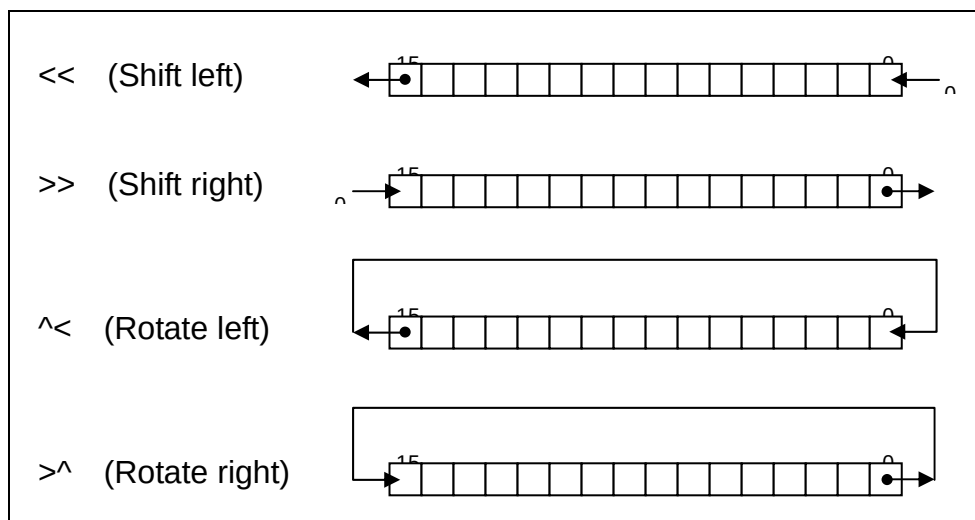
~ (Invert)	
X	~ X
1	0
0	1

AND		
X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

XOR		
X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

OR		
X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

Shift and Rotate



String Operations

Strings may be concatenated using "+".

Example:

A\$ = "ABC"

B\$ = "DEF"

C\$ = A\$ + " " + B\$

After these statements was executed, the value of C\$ is "ABC DEF".

Strings may be compared using the same relational operators that are used with numbers:

> >= < <= == !=

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant.

Examples:

"IS" < "IT"

"OK" != "ok"

"This " > "This"

"kg" > "KG"

"NOT" < "NOTE"

Chapter 5 Statements and Functions

How to use this chapter

This chapter describes in detail the various statements and functions applicable to **SmartScript** Module. Most descriptions consist of five parts, FUNCTION, VERSIONS, FORMAT, REMARKS and EXAMPLE, which are listed below.

FUNCTION	A brief explanation of the statement.
VERSIONS	Indicates which versions of Control Script Module support the particular statement.
FORMAT	Describes the statement syntax according to the following conventions: CAPS Keywords are indicated by capital letters and should be entered as shown. <i>Italics</i> Items in italics represent variable information to be supplied by the user. [] Square brackets indicate optional parameters. ... An ellipsis indicates an item may be repeated as many times as necessary. Vertical bar indicates alternative option. Punctuation must be indicated where indicated except for square brackets and the ellipsis.
REMARKS	Supplies additional information in detail regarding correct statement usage.
EXAMPLE	Illustrates various ways of using the statement and highlights, where applicable, unusual modes of operation.

Keywords by Programming Task

Programming task	Keywords included in this list
Control program flow	IF...ELSE...ELSEIF...ENDIF, CHOICE(), SWITCH...CASE...DEFAULT...ENDSW, FOR...LOOP, WHILE...LOOP, CONTINUE, EXIT, GOTO, GOSUB...RETURN, STOP, END IDLE SLEEP
Perform mathematical calculations	SIN(), COS(), TAN(), ASIN(), ACOS(), ATAN(), SINH(), COSH(), TANH(), EXP(), LN(), LOG(), SQRT(), ABS(), RAND(), FAC(), MIN(), MAX(), INT(), FMBCD(), FMFLT(), FMDBL()
Manipulate strings	LEN(), VAL(), IVAL(), ASC(), STR\$, ISTR\$, CHR\$, STRING\$, LEFT\$, RIGHT\$, MID\$, LOWER\$, UPPER\$, LTRIM\$, RTRIM\$, SUM08(), XOR08(), CRC16(),CRC32(), TOKEN,FORMAT\$, DATETIME\$()
Get the system time	TIMER(), SECOND(), MINUTE(), HOUR(), DAY(), WEEKDAY(), MONTH(), YEAR(), TICK(), NOW(), NOW\$()
File input/output	CREATE, OPEN, COMOPEN, SEEK, READ, WRITE, FPRINT, CLOSE, FCHECK(), FLEN(), FPOS(), CD, MD, RD, COPY, MOVE, DEL, DIR\$, COMMODE, SHORTCUT,FILE\$()
Declare array	DIM
Set traps for TAG events	TRAPON, TRAPOFF
Misc.	PI, BEEP, PLAY, MESSAGE, MSGBOARD, EXEC, SETDIR, ALMTAG\$, ALARM(), TAG(), ALMGRP(), ALMPRI(), ERRID(), ERRORTAG, NERR(),RSTERR, PASS, SHUTDOWN, SYSINFO\$, TONE

Statements and Functions

This section consists of an alphabetical listing of all statements and functions with a detailed description of each.

ABS() Function

FUNCTION ABS returns the absolute value of the specified value.

VERSIONS 1.0 and above

FORMAT $Y = \text{ABS}(\text{numeric-expression})$

REMARKS *numeric-expression* may be any numeric expression.

This function is equivalent to the algebraic expression $y = |x|$.

EXAMPLE $Y = \text{ABS}(45.5 - 100)$ // Y will be 54.5

ACOS() Function

FUNCTION ACOS returns the arccosine of the specified value.

VERSIONS 1.0 and above

FORMAT $Y = \text{ACOS}(\text{numeric-expression})$

REMARKS *numeric-expression* may be any numeric expression.

This *function* is equivalent to the algebraic expression $y = \cos(x)^{-1}$.

The ACOS function returns an angle in radians. To convert from radians to degrees, divided radians by (PI /180).

EXAMPLE $Y = \text{ACOS}(0) / \text{PI} * 180$ // Y will be 90

ALARM() Function

FUNCTION ALARM returns the alarm status of the specified Tag.

VERSIONS 1.1 and above

FORMAT Y = **ALARM**(*string-expression*)

REMARKS *string-expression* must be a string expression representing a Tag name.
The return value of this function is the alarm status of the Tag specified by the string expression. Meanings of alarm status values are explained as follows:

0x00	Normal
0x01	High alarm (Analog alarm) or Alarm (Digital alarm)
0x02	Low alarm(Analog alarm)
0x03	High-high alarm (Analog alarm)
0x04	Low-low alarm (Analog alarm)
0x81	Acknowledged High alarm (Analog alarm) or alarm (Digital alarm)
0x82	Acknowledged Low alarm (Analog alarm)
0x83	Acknowledged High-high alarm (Analog alarm)
0x84	Acknowledged Low-low alarm (Analog alarm)

EXAMPLE N% = ALARM("AI-0001")

ALMGRP() Function

FUNCTION ALMGRP returns the alarm group of the specified Tag

VERSIONS 1.2 and above

FORMAT $N\% = \text{ALMGRP}(\text{string-expression})$

REMARKS *string-expression* must be a string expression representing a Tag name.
Return value of this function is the alarm group the Tag specified by the string expression belongs to. If the Tag specified by the *string-expression* is not defined as an alarm, the return value will be 0. If *string-expression* is an empty string, the return value will be -1.

EXAMPLE $N\% = \text{ALMGRP}(\text{"Tag1"})$

ALMPRI() Function

FUNCTION ALMPRI return the alarm priority of the specified Tag.

VERSIONS 1.2 and above

FORMAT $N\% = \text{ALMPRI}(\text{string-expression})$

REMARKS *string-expression* must be a string expression representing a Tag name.
Return value of the function is the alarm priority of the Tag specified by *string-expression*. If the Tag specified by the *string-expression* is not defined as an alarm, the return value will be 0. If *string-expression* is an empty string, the return value will be -1.

EXAMPLE $N\% = \text{ALMPRI}(\text{"Tag1"})$

ALMTAG\$() 函數

FUNCTION ALMTAG\$ returns the tag name of the active alarm which has the highest priority.

VERSIONS 1.1 and above

FORMAT A\$ = ALMTAG\$()

REMARKS Return value of this function is a string. The string is the tag name of the alarm which has the highest priority or is the newest among all active alarm. Which alarm is the highest priority alarm is determined by the following logics:

- (1) If there are more than one active alarms and some of the alarms are not acknowledged, the unacknowledged alarm that has the highest alarm priority setting is the highest priority alarm. In other word, unacknowledged alarms have higher priority than acknowledged alarms.
- (2) If all active alarms are acknowledged, the acknowledged alarm with the highest alarm priority setting is the highest priority alarm.
- (3) If multiple alarms have the same priority setting, the newest alarm among them is the highest priority alarm.

The highest priority alarm is also the alarm shown in AlarmBar object in SmartPanel.

EXAMPLE A\$ = ALMTAG\$()

ASC() Function

FUNCTION ASC returns the value of the ASCII code of the first character in the specified string.

VERSIONS 1.0 and above

FORMAT $N\% = \text{ASC}(\text{string-expression})$

REMARKS *string-expression* may be any string expression.

EXAMPLE $N\% = \text{ASC}("Q")$ // N% will be 81

ASIN() Function

FUNCTION ASIN returns the arcsine of the specified value.

VERSIONS 1.0 and above

FORMAT $Y = \text{ASIN}(\text{numeric-expression})$

REMARKS *numeric-expression* may be any numeric expression.

This function is equivalent to the algebraic expression $y = \sin(x)^{-1}$.

The ASIN function returns an angle in radians. To convert from radians to degrees, divided radians by (PI /180).

EXAMPLE $Y = \text{ASIN}(1) / \text{PI} * 180$ // Y will be 90

ATAN() Function

FUNCTION ATAN returns the arctangent of the specified value.

VERSIONS 1.0 and above

FORMAT $Y = \text{ATAN}(\text{numeric-expression})$

REMARKS *numeric-expression* may be any numeric expression.

This function is equivalent to the algebraic expression $y = \tan(x)^{-1}$.

The ATAN function returns an angle in radians. To convert from radians to degrees, divided radians by (PI /180).

EXAMPLE `Y = ATAN(1) / PI * 180` `// Y will be 45`

BEEP Statement

FUNCTION BEEP generates a beep sound from your computer's speaker.

VERSIONS 1.0 and above

FORMAT **BEEP**

REMARKS This statement generates a beep sound from your computer's speaker.

EXAMPLE BEEP

CD Statement

FUNCTION Change current working directory to the specified path .

VERSIONS 4.0 and above

FORMAT **CD** *directory-name*

REMARKS *directory-name* Must be a string expression representing a valid path.

EXAMPLE CD “..\Project\Proj1\Dat” // Change current directory to ..\Project\Proj1\Dat\

CHOICE() Function

FUNCTION CHOICE returns the true expression or false expression depends on conditional expression.

VERSIONS 1.0 and above

FORMAT *Y* = **CHOICE**(*cond-expression*, *true-expression*, *false-expression*)

REMARKS *cond-expression* may be any expression. If it evaluates to true (non-zero), then evaluate *true-expression* and return its type and value as the function's result. If it evaluates to false (zero), then evaluate *false-expression* and return its type and value as the function's result.

true-expression may be any numeric expression. It is evaluated only when *cond-expression* evaluates to true.

false-expression may be any numeric expression. It is evaluated only when *cond-expression* evaluates to false.

EXAMPLE Y = CHOICE(Y >= 0, Y + 1, Y - 1)

CHR\$() Function

FUNCTION CHR\$ returns a character from the specified ASCII code.

VERSIONS 1.0 and above

FORMAT M\$ = **CHR\$(numeric-expression)**

REMARKS *numeric-expression* is a numeric expression in the range 0 to 255.

EXAMPLE M\$ = CHR\$(13) + CHR\$(10) // M\$ will be “\r\n”

CLOSE Statement

FUNCTION CLOSE closes an opened file or device.

VERSIONS 1.0 and above

FORMAT **CLOSE** [*file-number*]

REMARKS *file-number* is the number under which the file or device was opened.

CLOSE with no arguments closes all opened files and devices.

EXAMPLE CREATE 1, "TEST.TXT"
CLOSE 1

COMMODE Statement

FUNCTION COMMODE is used to change the communication parameters of an opened communication device.

VERSIONS 4.0 and above

FORMAT **COMMODE** *file-number,comm-param*

REMARKS *file-number* is an integer expression whose value should reference the file number of an opened communication device.

comm-param is a string expression used to specify the communication parameters of the specified communication device. Format of the parameter is identical to the format used in MODE command in Command Prompt shell of Windows.

EXAMPLE COMOPEN 1, "COM2:9600,N,8,1", 2048, 1024
COMMODE 1,"COM2:19200,E,7,1"

COMOPEN Statement

FUNCTION COMOPEN is used to open a communications device file.

VERSIONS 1.0 and above

FORMAT **COMOPEN** *file-number, comm-param, in-queue, out-queue*

REMARKS

file-number	is an integer expression, which evaluates to a valid file number. The number is then associated with the file for as long as it is open and is used by other communications I/O statements to refer to the file. The valid file numbers are 1 to 16.
comm-param	is a string expression that specifies the device-control information for a device. The string must have the same form as the DOS MODE command-line parameter.
in-queue	is an integer expression that specifies the size of the interrupt-driven receive-queue.
out-queue	is an integer expression that specifies the size of the interrupt-driven transmit-queue.

EXAMPLE COMOPEN 1, "COM2:9600,N,8,1", 2048, 1024

CONTINUE Statement

FUNCTION CONTINUE passes control to the next iteration of the smallest enclosing FOR or WHILE statement in which it appears.

VERSIONS 1.0 and above

FORMAT CONTINUE

REMARKS For more information, see FOR ... LOOP, WHILE ... LOOP and EXIT statements.

EXAMPLE FOR N%=0 TO 10
 ...
 IF (N% == 3) => CONTINUE
 ...
 LOOP

COPY Statement

FUNCTION COPY copies the specified file or folder to the specified target path.

VERSIONS 4.0 and above

FORMAT COPY *source-path,target-path*

REMARKS *source-path* is a string expression used to specify the path of the source file or folder.
target-path is a string expression used to specify the path of the target file or folder.

The statement may fail if the source file does not exist, the target path does not exist, or there is other reason prevents its proceeding. The failure will be counted as an internal error but the script will continue its execution. ERRID() function can be used to get the error code.

If the target file exists, COPY statement will overwrite the file with the source file.

EXAMPLE COPY "C:\\Proj1\\Log.txt", "C:\\Proj2\\LogFile.txt"

COS() Function

FUNCTION COS returns the cosine of the specified angle.

VERSIONS 1.0 and above

FORMAT $Y = \text{COS}(\text{numeric-expression})$

REMARKS *numeric-expression* must be an angle expressed in radians.

This function is equivalent to the algebraic expression $y = \cos(x)$.

To convert from degrees to radians, multiply degrees by $(\text{PI} / 180)$.

If the target file exists, COPY statement will overwrite the file with the source file.

EXAMPLE $Y = \text{COS}(180 * (\text{PI} / 180))$ // Y will be -1

COSH() Function

FUNCTION COSH returns the hyperbolic cosine of the specified value.

VERSIONS 1.0 and above

FORMAT $Y = \text{COSH}(\text{numeric-expression})$

REMARKS *numeric-expression* may be any numeric expression.

This function is equivalent to the algebraic expression $y = \cosh(x)$.

EXAMPLE $Y = \text{COSH}(1)$ // Y will be 1.543081

CRC16() Function

FUNCTION CRC16 returns the value of CRC-16 of the specified string. CRC-16 is an algorithm commonly used in communication protocol for error checking.

VERSIONS 1.0 and above

FORMAT *N%* = **CRC16**(*string-expression*)

REMARKS *string-expression* may be any string expression.

EXAMPLE *N%* = CRC16("\x02ABC\x03") // *N%* will be -9832

CRC32() Function

FUNCTION CRC32 returns the value of CRC-32 of the specified string. CRC-32 is an algorithm commonly used in communication protocol for error checking.

VERSIONS 4.0 and above

FORMAT *N%* = **CRC32**(*string-expression*)

REMARKS *string-expression* may be any string expression.

EXAMPLE *N%* = CRC32("\x02ABC\x03") // *N%* will be -1375105033

CREATE Statement

FUNCTION CREATE creates a new file and opens it.

VERSIONS 1.0 and above

FORMAT **CREATE** *file-number*, *file-name*

REMARKS *file-number* is an integer expression, which evaluates to a valid file number. The number is then associated with the file for as long as it is open and is used by other file I/O statements to refer to the file. The valid file numbers are 1 to 16.

file-name is a string expression, which names the file to be opened.

If the file does not exist, the statement creates a new file and opens it for writing. If the file does exist, the statement truncates the file size to zero and opens it for reading and writing. When the statement opens the file, the read-write pointer is set to the beginning of the file.

EXAMPLE CREATE 1, "TEST.TXT"
CLOSE 1

DATETIME\$() Function

FUNCTION DATETIME\$ convert a real number into a date/time string..

VERSIONS 4.0.0.10 and above

FORMAT M\$ = **DATETIME\$(numeric- expression)**

REMARKS *numeric-expression* is an real number expression. Its value should represent a date/time. For example, it can be a TAG data/time variable. The output string has the format of "yyyy/mm/dd hh:ii:ss", where yyyy is the four digits year part, mm is the two digits month part, dd is the two digits day part, hh is the two digits hour part, ii is the two digits minute part, ss is the two digits second part of a complete date.time notation. The length of the output string is fixed at 19 characters.

EXAMPLE M\$= DATETIME\$({Tag1.t}) // M\$ is the date/time filed of Tag1 represented as a
// string

DAY() Function

- FUNCTION** DAY returns the current day of month.
- VERSIONS** 1.0 and above
- FORMAT** *N%* = **DAY()**
- REMARKS** This function returns the current day of month (1 ~ 31)
- EXAMPLE** *N%* = DAY()

DEL Statement

- FUNCTION** DEL is used to delete a file or a folder.
- VERSIONS** 4.0 and above
- FORMAT** **DEL** *path*
- REMARKS** *path* is a string expression used to specify a file path.
- DEL may fail If the specified file does not exist or there is other reason prevents its deletion. The failure will be counted as an internal error but the script will continue its execution. ERRID() function can be used to get the error code.
- DEL can delete the specified file even it has read-only attribute set.
- EXAMPLE** DEL "C:\\Lablink\\Project\\Proj1\\Txt\\Log.txt"

DIM Statement

FUNCTION DIM specifies the maximum values for array variable subscripts and allocates storage accordingly.

VERSIONS 1.0 and above

FORMAT **DIM** *variable*(*numeric-const* [, *numeric-const* [, *numeric-const*]])

REMARKS *variable* may be any array variable name.
numeric-const specifies the maximum values for array subscripts and must be an integer constant.

The DIM statement sets all the elements of the specified array to an initial value of zero. The maximum allowable number of dimensions for an array is 3. The maximum allowable number of elements is 8,192. The minimum value for a subscript is always 1.

This statement is a non-executable statement.

EXAMPLE DIM A(10)
 A(1) = 6.5

DIR\$() Function

FUNCTION DIR\$() can be used to query for current directory, Windows and system directory.

VERSIONS 4.0 and above

FORMAT M% = **DIR\$**("NOW"|"WIN"|"SYS")

REMARKS NOW - Query for current directory.
 WIN - Query for Windows directory.
 SYS - Query for Windows system directory.

EXAMPLE M1\$ = DIR\$("NOW") // M1\$ = "C:\Lablink\System4"
 M2\$ = DIR\$("WIN") // M2\$ = "C:\Windows"
 M3\$ = DIR\$("SYS") // M3\$ = "C:\Windows\System32"

END Statement

FUNCTION END terminates program execution and closes all files.

VERSIONS 1.0 and above

FORMAT END

REMARKS END statements may be placed anywhere in the program to terminate execution. An END statement at the end of a program is optional.

EXAMPLE IF (A > 0) => END

ERRID() Function

FUNCTION ERRID() returns the error code of the latest script internal error.

VERSIONS 4.0 and above

FORMAT N% = ERRID()

REMARKS Return value of ERRID() is the error code of the latest internal error. (See appendix for a list of all error code.) A return value of 0 means there is no error.

Note that any successful execution of a statement or function will clear the error code return by this function. Be sure to place this function right next to the statement or function where error is expected to get its error code.

EXAMPLE N% = ERRID() // N% = 0~37

ERRORTAG Statement

FUNCTION ERRORTAG can specify a Tag name. When the SmartScript end abnormally, the error code and error message will be set to the value and message field of the Tag.

VERSIONS 4.0.1.1 and above

FORMAT **ERRORTAG** *string-expression*

REMARKS *string-expression* is a string expression which is a valid Tag name.

If the Tag specified by the tag name string-expression does not exist, it will be created automatically at runtime. The tag name string-expression should comply with the naming rule of Lab-LINK Tag.

When the script with ERRORTAG statement ends abnormally, the value of the error tag will indicate the error code while its message will be the name of the script file, the error line number and the error message.

If the statement has been used more than once in a SmartScript, only the last executed one will take effect.

If more than one SmartScript are executed at Lab-LINK runtime, it is recommended to define different error tag name for each script to help distinguish in which script an error occurs.

EXAMPLE `ERRORTAG="ETag" // Etag is specified as the error tag`
`A=1/10`

Store the script above is stored as a script file named "CSLPRG.csl". After it is executed and ended abnormally due to the error of divided by zero occurs, the value of ETag will be set to 18 and its message will be "ERROR 18: Division by Zero!! @ CSLPRG1.csl Line: 2".

EXEC Statement

FUNCTION EXEC opens a specified executable file or document file.

VERSIONS 1.0 and above

FORMAT EXEC *file-name, parameter, show-mode*

REMARKS *file-name* must be a string expression that specifies the file to open or the folder to open or explore. The statement can open an executable file or a document file.

parameter must be a string expression that specifies parameters to be passed to the application (if *file-name* specifies an executable file). If *file-name* specifies a document file, *parameter* should be an empty string.

show-mode specifies how the application is to be shown when it is opened. This parameter can be one of the following values:

"MIN"	Minimizes the specified window and activates the next top-level window in the Z order.
"MAX"	Maximizes the specified window.
"HIDE"	Hides the window and activates another window.
"ICON"	Displays the window as a minimized window. The active window remains active.
"NORMAL"	Activates the window and displays it in its current size and position.

EXAMPLE 1. To run Notepad:

```
EXEC "NOTEPAD.EXE", "", "NORMAL"
```

2. To run DBSaver

```
EXEC "DBSAVER.EXE", "PROJ1 WKS1", "NORMAL"
```

3. To run Report

```
EXEC "REPORT.EXE", "..\\Project\\Proj1\\cfg\\Wks1\\report.cat", "NORMAL"
```


EXIT Statement

FUNCTION EXIT terminates the smallest enclosing FOR or WHILE statement in which it appears.

VERSIONS 1.0 and above

FORMAT EXIT

REMARKS For more information, see FOR ... LOOP, WHILE ... LOOP and CONTINUE statements.

EXAMPLE FOR N%=0 TO 10
...
IF (A\$ == "QUIT") => EXIT
...
LOOP

EXP() Function

FUNCTION EXP returns the exponential of the specified value.

VERSIONS 1.0 and above

FORMAT $Y = \text{EXP}(\text{numeric-expression})$

REMARKS *numeric-expression* may be any numeric expression
This function is equivalent to the algebraic expression $y = e^x$

EXAMPLE Y = EXP(1) // Y will be 2.718282

FAC() Function

FUNCTION FAC returns the factorial of the specified value.

VERSIONS 1.0 and above

FORMAT $Y = \text{FAC}(\text{numeric-expression})$

REMARKS *numeric-expression* must be an integer expression.

This function is equivalent to the algebraic expression $y = x!$.

EXAMPLE `Y = FAC(5)` `// Y will be 120`

FCHECK() Function

FUNCTION FCHECK determines whether or not the specified file number is valid.

VERSIONS 1.0 and above

FORMAT $N\% = \text{FCHECK}(\text{file-number})$

REMARKS *file-number* is an integer expression, which evaluates to a file number.

If the file associated with the specified file number was successfully opened, it returns true (one); otherwise it returns false (zero).

EXAMPLE `OPEN 1, "TEST.TXT"`
`IF (! FCHECK(1)) => GOTO Error:`

FILE\$() Function

FUNCTION FILE\$() return the content of the specified file as a string.

VERSIONS 4.0 and above

FORMAT M\$ = **FILE\$**(*filename*)

REMARKS *Filename* is a string expression representing the path of the file to be All characters, including control character such as line-feed, will be stored into a string.

file-number is an integer expression, which evaluates to a file number.

If the file associated with the specified file number was successful opened, it returns true (one); otherwise it returns false (zero).

EXAMPLE M\$ = FILE\$("C:\\PROJ1\\Log.txt")

FLEN() Function

FUNCTION FLEN returns the length of the specified file.

VERSIONS 1.0 and above

FORMAT N% = **FLEN**(*file-number*)

REMARKS *file-number* is an integer expression, which evaluates to a file number.

If the file associated with the specified file number was successful opened, it returns the length of the file. If the file is a communications file, this function returns the number of characters in the receive-queue.

EXAMPLE OPEN 1, "TEST.TXT"

N% = FLEN(1)

CLOSE 1

FMBCD () Function

FUNCTION FMBCD decodes the argument as a BCD number and returns its value as an integer.

VERSIONS 2.0 and above

FORMAT *N%* = **FMBCD**(*numeric-expression*)

REMARKS *numeric-expression* is an integer expression in BCD encoding.

This function is used to decode a BCD number, such as a number read from a PLC register, and convert it into an integer for SCADA usage.

EXAMPLE Y = FMBCD(0x9876) // Y will be 9876

FMDBL () Function

FUNCTION FMDBL combines four arguments as a 4 word data and convert it to a double precision floating number.

VERSIONS 2.0 and above

FORMAT Y = **FMDBL**(*num-expression1*, *num-expression2*, *num-expression3*, *num-expression4*)

REMARKS *num-expression1* is a integer expression representing the first word of a double precision floating number.

num-expression2 is a integer expression representing the second word of a double precision floating number.

num-expression3 is a integer expression representing the third word of a double precision floating number.

num-expression4 is a integer expression representing the fourth word of a double precision floating number.

This function is used to convert 4 word data into a double precision floating number.

EXAMPLE Y = FMDBL((0x0000, 0x0000, 0x8000, 0x4024)) // Y will be 10.25

FMFLT () Function

FUNCTION FMFLT combines its two arguments as two word data and return its value as a single precision floating number.

VERSIONS 2.0 and above

FORMAT $Y = \text{FMFLT}(\text{numeric-expression1}, \text{numeric-expression2})$

REMARKS *numeric-expression1* is a integer expression representing the lower word of a single precision floating number.

numeric-expression2 is a integer expression representing the higher word of a single precision floating number.

This function is used to convert 2 word data into a single precision floating number.

EXAMPLE $Y = \text{FMFLT}(0x0000, 0x4124)$ // Y will be 10.25

FOR ... LOOP Statement

FUNCTION FOR performs a series of instructions in a loop a given number of times.

VERSIONS 1.0 and above

FORMAT **FOR** *variable* = *init-expression* **TO** *final-expression* [**STEP** *inc-expression*]

...

LOOP

REMARKS

<i>variable</i>	is a variable to be used as a counter.
<i>init-expression</i>	is a numeric expression, which is the initial value of the counter.
<i>final-expression</i>	is a numeric expression, which is the final value of the counter
<i>inc-expression</i>	is a numeric expression to be used as an increment. (The default value is 1.)

If the value of the counter is less than or equal to the value of *final-expression* then Control Script Module continues with the statement following the FOR statement.

The program statements following the FOR are executed until the LOOP statement is encountered. At this point the counter is incremented by the STEP value (*inc-expression*) and Control Script Module branches back to the FOR statement where the process begins again.

If the value of the counter is greater than the value of *final-expression*, then Control Script Module branches to the statement following the LOOP statement.

If the value of *inc-expression* is negative, then the test is reversed. The counter is decreased each time through the loop, and the loop is executed until the counter is less than the final value.

Nested Loops

FOR ... LOOP may be nested; that is, one FOR ... LOOP may be placed inside another FOR ... LOOP. When loops are nested, each loop must have a unique variable name as its counter. Each LOOP will match the most recent FOR.

EXAMPLE

```

DIM A(3, 4)
FOR I = 1 TO 3
  FOR J = 1 TO 4
    A(I, J) = (I - 1) * 4 + (J - 1)
  
```

LOOP

LOOP

Y = FMFLT(0x0000, 0x4124) // Y will be 10.25

Format\$() Function

FUNCTION Format convert the integer argument into a string with the format specified by the format string argument.

VERSIONS 2.0 and above

FORMAT $N\$ = \text{FORMAT\$}(\text{string-expression}, \text{numeric-expression})$

REMARKS *string-expression* is a string used to define the format needed. See description below for details.

numeric-expression can be any numeric expression.

Format string consists of the following fields while fields in square brackets are optional and can be omitted:

$\%[\text{flags}] [\text{width}] [.\text{precision}] \text{type}$

Each field in the format string is either a single character or a number and is used to specify a format option. The simplest form of format string contains only a percent symbol % and a *type* character for data type. Ex. %d. The optional fields before the type character are used to control other format feature.

type

type character is the only mandatory field in format string. It must appear after any other format character. *type* character specify the data type of the numeric argument. It can be one of the following characters:

Char	Data type	Output format
d	Int (integer)	Decimal signed integer
i	Int (integer)	Decimal signed integer
o	Int (integer)	Octal signed integer
u	Int (integer)	Decimal unsigned integer
x	Int (integer)	Hexadecimal unsigned integer using "abcdef"
X	Int (integer)	Hexadecimal unsigned integer using "ABCDEF"
e	Double (double precision floating number)	Signed floating number in the format of [–] <i>d</i> . <i>dddd</i> e [<i>sign</i>] <i>ddd</i> where <i>d</i> indicate a single digit decimal number and <i>dddd</i> is a single or multiple digits decimal number. <i>ddd</i> is a number with exact three digit of decimal number and <i>sign</i> can be either + or –.
E	Double (double precision)	Same as e except that capital letter E replaces the lower case letter e for exponential expression.

	floating number)	
f	Double (double precision floating number)	Signed floating number in the format of <code>[-]dddd.dddd</code> where <code>dddd</code> is a single or multiple digits decimal number. The number of digits before the decimal point depends on the value of the number and the number of digits after the decimal point depends on the precision needed.
g	Double (double precision floating number)	Signed floating number using f or e format. The format actually used depends on which format produce a shorter string. Extra trailing 0 will be trimmed. Decimal point appears only when necessary.
G	Double (double precision floating number)	Same as format g except that capital letter E replaces the lower case letter e for exponential expression.

flags

flag character is the first optional field in format string. *flag* is a single character used to specify alignment of sign, space, decimal point, and prefix of octal or hexadecimal number. Format string can contain one or more *flag* characters.

<i>flag</i>	Meaning	Default
-	Align the output to left within the specified width	Align to right
+	If the output of signed data type, add positive/negative sign (+ or -) prior to the output number	Only negative (-) sign is shown 只在負數前顯示負號.
0	If 0 flag appear before width setting, add leading 0 before the number to make up the minimum width. If 0 flag and - flag are set at the same time, 0 flag will be ignored. If 0 flag is used with integer type (i , u , x , X , o , d), 0 flag will be ignored.	No leading 0.
Space (' ')	Add leading space before the number to make up the minimum width if the output it a signed positive number. If Space flag and + flag are used at the same time, Space flag will be ignored.	No leading space
#	When used with o , x or X type, # flag add 0 · 0x or 0X prefix before the number depending on the type. When used with e , E or f type, the output value will always contain the decimal point. When used with g or G type, the output value will always contain the decimal point and the extra trailing 0 after the number will not be removed The flag is ignored if used with d , l or u types.	No prefix Decimal point appears only when there are fractional digits. Decimal point appears only when there are fractional digits. All trailing 0 will be removed.

width

width is the second optional field in format string. *width* is a non-negative decimal integer. It is used to control the minimum number of characters in the output. If the output number contain less characters than the specified *width*, space characters will be added before or after the number based on the existence of – *flag* (which means aligning to left) to make up the specified minimum *width*. If 0 *flag* is added before *width*, trailing 0 will be used to make up the width. 0 *flag* is ignore if left alignment is set.

width setting will not remove any number characters. If the number of characters of the output is greater than the specified *width*, the *width* setting will be ignored and all characters will be output. (The actual number of output characters may still be controlled by the *precision* setting)

precision

precision is the third optional field in the format string. It is a non-negative decimal integer with a period symbol (.) as prefix. It is used to specify number of characters, number of fraction digits and number of significant digits. (See table below). This setting differs from the *width* setting because it will remove some of the output characters and will round up some digits if the output is a floating point number. If *precision* is set to 0 and the output number is also 0, no character will be output.

Y\$ = FORMAT\$("%0d", 0) // Y contains no character

Type setting determines how *precision* will be interpreted and the default if *precision* is not used. See the *table* below:

type	Meaning	Default
d, i, u, o, x, X	<i>Precision</i> specify the minimum number of output digits. If the input argument has fewer digits, leading 0 is added to the left of the output number. If the input number has more digits than <i>precision</i> , the extra digits will not be removed.	Default <i>precision</i> is 1.
e, E	Precision specify the number of fraction digits, the number of digits after the decimal point. The extra digits will be round up.	Default <i>precision</i> is 6. If <i>precision</i> is 0 or there is no number after the period in <i>precision</i> setting, neither decimal point nor any fraction digits will be output.
f	Precision specify the number of digits after the decimal point. At least one digit will appear before the decimal point and the number will be rounded up based on the	Default <i>precision</i> is 6. If <i>precision</i> is 0 or there is no number after the period in <i>precision</i> setting, neither decimal point nor any

	precision specified.	fraction digits will be output.
g, G	Precision specify the maximum number of significant digits.	6 significant digits and all extra trailing 0 will be removed.

FPOS() Function

FUNCTION FPOS returns the current read-write pointer position of the specified file.

VERSIONS 1.0 and above

FORMAT *N%* = **FPOS**(*file-number*)

REMARKS *file-number* is an integer expression, which evaluates to a file number.

If the file associated with the specified file number was successful opened, it returns the current read-write pointer position of the file. If the file is a communications file, this function returns the number of characters in the transmit-queue.

EXAMPLE OPEN 1, "TEST.TXT"
N% = FPOS(1) // *N%* will be 0
 CLOSE 1

FPRINT Statement

FUNCTION FPRINT writes data to the specified file.

VERSIONS 1.0 and above

FORMAT **FPRINT** *file-number*, *expression*, ..., *expression*

REMARKS *file-number* is an integer expression, which evaluates to a file number.

expression may be any numeric or string expression that will be written to the file.

All numeric value will be converted to ASCII string by free format then written to the file. Expressions should be delimited by commas, and no extra blanks will be inserted between expressions.

EXAMPLE A = PI
 CREATE 1, "TEST.TXT"
 FPRINT 1, "A = ", A, "\r\n"
 CLOSE 1

GOSUB Statement

FUNCTION GOSUB branches to the specified subroutine.

VERSIONS 1.0 and above

FORMAT **GOSUB** *line-label*

REMARKS *line-label* is the first line of the subroutine.

A subroutine must be end with a RETURN statement, and cause Control Script Module to branch back to the statement following the most recent GOSUB statement. A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, it may be preceded by a STOP, END or GOTO statement that directs program control around the subroutine.

EXAMPLE

GOSUB ShowMsg:

```
MESSAGE "Test Program", "In main program"
END
```

ShowMsg:

```
MESSAGE "Test Program", "In subroutine"
RETURN
```

GOTO Statement

FUNCTION GOTO branches unconditionally out of the normal program sequence to a specified line label..

VERSIONS 1.0 and above

FORMAT **GOTO** *line-label*

REMARKS *line-label* is a valid line label in the program.

If line label is an executable statement, that statement and those following are executed. If it is a non-executable statement, execution proceeds at the first executable statement encountered after line label.

EXAMPLE GOTO ErrHandle:

HOUR() Function

FUNCTION HOUR returns the current hours after midnight.

VERSIONS 1.0 and above

FORMAT *N%* = **HOUR()**

REMARKS This function returns the current hours after midnight (0 ~ 23).

EXAMPLE *N%* = HOUR()

IDLE Statement

FUNCTION IDLE suspend the execution of the script but can still handle event.

VERSIONS 4.0 and above

FORMAT IDLE

REMARKS IDLE statement is usually used to wait for events to occur. It is similar to running an empty loop (Ex. WHILE(1).....LOOP), but unlike a loop, it will not consume system resource during the waiting.

EXAMPLE IDLE
{EVENT}:
{Value} = {Value} + 1 //{Value} will increase by 1 when {EVENT} changes
RETURN

IF ... ELSEIF ... ELSE ... ENDIF Statement

FUNCTION IF makes a decision regarding program flow based on the result of an expression.

VERSIONS 1.0 and above

FORMAT IF (cond-expression) => statement
 IF (cond-expression)
 ...
 [ELSEIF (cond-expression)]
 ...
 [ELSE]
 ...
 ENDIF

REMARKS *statement* may be any statement.
cond-expression may be any expression.

If the result of the *cond-expression* is true (non-zero), the statement following the “then symbol” or statements following the IF is executed. If the result is false (zero), the statements following the IF are ignored and the statement ELSEIF, if present, is executed. As the IF statement, If the result of the *cond-expression* is true, the statements following the ELSEIF is executed. Otherwise, the next ELSEIF, if present, is executed, and so on. Finally, if no one of the ELSEIF is true, the statements following the ELSE, if present, is executed. If no ELSE present, the statement following the ENDIF is executed.

IF ... ENDIF statements may be nested. Each ENDIF will match the most recent IF.

EXAMPLE IF ((A >= 0) & (A <= 10))
 M\$ = “OK”
 ELSEIF (A > 10)
 M\$ = “> 10”
 ELSE
 M\$ = “< 0”
 ENDIF

INT() Function

FUNCTION INT returns the truncated integer part of the specified value.

VERSIONS 1.0 and above

FORMAT *N%* = **INT**(*numeric-expression*)

REMARKS *numeric-expression* is any numeric expression

EXAMPLE *N%* = INT(99.9) // *N%* will be 99
N% = INT(-4.8) // *N%* will be -4

ISTR\$() Function

FUNCTION ISTR\$ converts a value to a string by the specified notation.

VERSIONS 1.0 and above

FORMAT *M\$* = **ISTR\$**(*numeric-expression* , *radix-expression*)

REMARKS *numeric-expression* may be any numeric expression.

radix-expression is an integer expression, which evaluates to a valid base of value. The valid base is 2 to 36.

This function converts the result of *numeric_expression* to an unsigned long integer, and then converts it to a string by the specified notation.

EXAMPLE *M\$* = ISTR\$(15, 2) // *M\$* will be "1111"

IVAL() Function

FUNCTION IVAL converts a string to a value by the specified notation.

VERSIONS 1.0 and above

FORMAT $Y = \text{IVAL}(\text{string-expression}, \text{radix-expression})$

REMARKS *string-expression* may be any string expression.

radix-expression is an integer *expression*, which evaluates to a valid base of value. The valid base is 2 to 36.

This function converts the result of *string_expression* to an integer by the specified notation.

IVAL returns 0 if no conversion can be performed.

EXAMPLE $A = \text{IVAL}(\text{"FFFF"}, 16)$ // A will be 65535

LEFT\$() Function

FUNCTION LEFT\$ returns a string comprised of the leftmost *N* characters of the specified string.

VERSIONS 1.0 and above

FORMAT $M\$ = \text{LEFT\$}(\text{string-expression}, \text{count-expression})$

REMARKS *string-expression* may be any string expression.

count-expression is an integer expression, which evaluates to a valid count value. The valid count value is 0 to maximum length of the string.

EXAMPLE $M\$ = \text{LEFT\$}(\text{"ABCDEFGF"}, 3)$ // M\$ will be "ABC"

LEN() Function

FUNCTION LEN returns the number of characters in the specified string.

VERSIONS 1.0 and above

FORMAT $N\% = \text{LEN}(\text{string-expression})$

REMARKS *string-expression* may be any string expression.

EXAMPLE $N\% = \text{LEN}(\text{"ABCDEFGH"})$ // N% will be 7

LN() Function

FUNCTION LN returns the natural logarithm of the specified value.

VERSIONS 1.0 and above

FORMAT $Y = \text{LN}(\text{numeric-expression})$

REMARKS *numeric-expression* is a numeric expression and must be greater than zero.

This function is equivalent to the algebraic expression $y = \ln x = \log_e x$. The natural logarithm is the logarithm to the base e.

EXAMPLE $Y = \text{LN}(2)$ // Y will be 0.693147

LOG() Function

FUNCTION LOG returns the logarithm of the specified value.

VERSIONS 1.0 and above

FORMAT $Y = \text{LOG}(\text{numeric-expression})$

REMARKS *numeric-expression* is a numeric expression and must be greater than zero.

This function is equivalent to the algebraic expression $y = \log x = \log_{10} x$. The logarithm is the logarithm to the base 10.

EXAMPLE $Y = \text{LOG}(10)$ // Y will be 1

LOWER\$() Function

FUNCTION LOWER\$ returns a string same as the specified string but all characters were converted to lowercase.

VERSIONS 1.0 and above

FORMAT $M\$ = \text{LOWER\$}(\text{string-expression})$

REMARKS *string-expression* may be any string expression.

EXAMPLE $M\$ = \text{LOWER\$}(\text{"ABCDefg"})$ // M\$ will be "abcdefg"

LTRIM\$() Function

FUNCTION LTRIM\$ returns a string same as the specified string but without the leading space.

VERSIONS 1.0 and above

FORMAT *M\$ = LTRIM\$(string-expression)*

REMARKS *string-expression* may be any string expression.

EXAMPLE M\$ = LTRIM\$(" ABCDEFG") // M\$ will be "ABCDEFG"

MAX() Function

FUNCTION MAX compares two values and returns the larger one.

VERSIONS 1.0 and above

FORMAT *Y = MAX(numeric-expression, numeric-expression)*

REMARKS *numeric-expression* may be any numeric expression.

EXAMPLE Y = MAX(2.15, 4.3) // Y will be 4.3

MD Statement

FUNCTION MD is used to create a new folder.

VERSIONS 4.0 and above

FORMAT **MD** *directory-name*

REMARKS *directory-name* is a string expression representing the path of the new folder to be created.

directory-name can contain subfolders. If any intermediate subfolders do not exist, MD statement will create them automatically.

MD may fail to create the specify folder due to file access privilege setting. Such failure will be counted as an internal error but the script will continue its execution. ERRID() function can be used to get the error code.

EXAMPLE MD "C:\\NewFolder"

MESSAGE Statement

FUNCTION MESSAGE displays a message box.

VERSIONS 1.0 and above

FORMAT **MESSAGE** *title-expression*, *text-expression*

REMARKS *title-expression* is a string expression containing the message to be displayed.

text-expression is a string expression used for the message box title.

The message box contains an application-defined title and message, plus an icon consisting of a lowercase letter *i* in a circle-combination and one push buttons: OK.

Windows does *not* automatically break the lines to fit in the message box, however, so the message string must contain carriage returns to break the lines at the appropriate places.

EXAMPLE MESSAGE "Test Program", "This is line 1\nThis is line 2"

MID\$() Function

FUNCTION MID\$ returns the requested part of the specified string.

VERSIONS 1.0 and above

FORMAT *M\$ = MID\$(string-expression, index-expression, count-expression)*

REMARKS *string-expression* may be any string expression.

index-expression is an integer expression, which evaluates to a valid index value. The valid index value is 1 to maximum length of the string.

count-expression is an integer expression, which evaluates to a valid count value. The valid count value is 0 to maximum length of the string.

EXAMPLE M\$ = MID\$("ABCDEFGH", 3, 2) // M\$ will be "CD"

MIN() Function

FUNCTION MIN compares two values and returns the smaller one.

VERSIONS 1.0 and above

FORMAT *Y = MIN(numeric-expression, numeric-expression)*

REMARKS *numeric-expression* may be any numeric expression..

EXAMPLE Y = MIN(2.15, 4.3) // Y will be 2.15

MINUTE() Function

- FUNCTION** MINUTE returns the current minutes after hour.
- VERSIONS** 1.0 and above
- FORMAT** $N\% = \text{MINUTE}()$
- REMARKS** This function returns the current minutes after hour (0 ~ 59).
- EXAMPLE** $N\% = \text{MINUTE}()$

MONTH() Function

- FUNCTION** MONTH returns the current month.
- VERSIONS** 1.0 and above
- FORMAT** $N\% = \text{MONTH}()$
- REMARKS** This function returns the current month (1 ~ 12).
- EXAMPLE** $N\% = \text{MONTH}()$

MOVE Statement

FUNCTION MOVE is used to move a file or a folder.

VERSIONS 4.0 and above

FORMAT **MOVE** *source-path,target-path*

REMARKS *source-path* is a string expression representing the path of the source file or folder.

target-path is a string expression representing the path of the target file or folder.

When moving a file or a folder, MOVE may fail due to the source file or folder doesn't exist, the target path doesn't exist or any other reason. Such failure will be counted as an internal error but the script will continue its execution. ERRID() function can be used to get the error code.

MOVE will fail if the target file exists. Note that this is different from the result of COPY statement which will overwrite the existing target file automatically.

EXAMPLE MOVE "C:\\Folder1\\Log1.txt"," C:\\ Folder1\\Log2.txt"

MSGBOARD Statement

FUNCTION MSGBOARD opens an “**Important Message**” window and displays a line of message.

VERSIONS 4.0 and above

FORMAT **MSG** *text-expression*

REMARKS *text-expression* is a string expression used for the message to be displayed.

Executing this statement will open an “**Import Message**” window to display the specified message string. Date, time and a [CslMan32] label will be added to the message automatically. After the message is shown, SmartScript will not wait for user to close the window before executing the following statements in the script. If the “**Important Message**” window is already opened when the statement is executed, the message will be added below those messages already shown in the window.

There are two buttons at the bottom of the “**Important Message**” window. If “Save & Close” button is pressed, the system save all the messages in the window to the text file “MsgBoard.log” in the system folder (default path is c:\lablink\system4), clear the messages and close the window; if “Close button” is pressed, the system will clear all messages and close the window immediately.

Each execution of the statement will add one and only one line of message, <Carriage Return> and <Line Feed> character in the message string will be trimmed.

EXAMPLE MSGBOARD “This is a test message”

NERR() Function

FUNCTION NERR() returns the accumulated error count.

VERSIONS 4.0 and above

FORMAT N% = **NERR()**

REMARKS NERR() returns the accumulated error count. Whenever a statement or a function cause an internal error, the error count will be incremented by 1. RSTERR statement can clear the error count.

EXAMPLE N% = NERR() //N% is a non-negative integer

NOW() Function

FUNCTION NOW() returns the current system date/time represented as a real number.

VERSIONS 4.0.0.10 and above

FORMAT N% = **NOW()**

REMARKS NOW() return the current system date/time. The date/time will be represented as a real number. Its integer part represents the date field of the TAG, and its fraction part represents the time field of the TAG. Both integer and fraction parts are of the same unit of days. The returned value of this function can be assign to a TAG variable to modify its date/time field.

EXAMPLE {Tag1.t} = NOW() // The date/time filed of Tag1 are modifie as current system
// date/time.

NOW\$() Function

FUNCTION NOW\$() returns the current system date/time represented as a string.

VERSIONS 4.0.0.10 and above

FORMAT M\$ = NOW\$()

REMARKS NOW\$() return the current system date/time. The date/time will be represented as a string. The output string has the format of "yyyy/mm/dd hh:ii:ss.sss", where *yyyy* is the four digits year part, *mm* is the two digits month part, *dd* is the two digits day part, *hh* is the two digits hour part, *ii* is the two digits minute part, integer part of *ss.sss* is the two digits second and its fraction part is the three digits millisecond part of a complete date.time notation. The length of the output string is fixed at 23 characters.

EXAMPLE M\$ = NOW\$() // M\$ is the system, ex. "2010/01/01 12:00.00.000"

OPEN Statement

FUNCTION OPEN opens the specified file.

VERSIONS 1.0 and above

FORMAT OPEN *file-number*, *file-name*, *access-type*

REMARKS *file-number* is an integer expression, which evaluates to a valid file number. The number is then associated with the file for as long as it is open and is used by other file I/O statements to refer to the file. The valid file numbers are 1 to 16.

file-name is a string expression, which names the file to be opened.

access-type is a string expression that specifies the type of access to the file. This parameter can be one of the following values:

"R"	Specifies read-only access to the file. Data can be read from the file.
"W"	Specifies write-only access to the file. Data can be write to the file.
"RW"	Specifies read-and-write access to the file. Data can be read from and write to the file.

When the statement opens the file, the read-write pointer is set to the beginning of the file.

EXAMPLE OPEN 1, "TEST.TXT", "R"

PASS Statement

- FUNCTION** PASS is used to open an executable file or a document file. It is similar to EXEC, but will wait for the application used to open the specified file terminates before continuing the execution of the statements following this statement.
- VERSIONS** 4.0 and above
- FORMAT** **PASS** "*filename*", "*parameter*", "*showmode*"
- REMARKS**
- | | |
|------------------|--|
| <i>file-name</i> | must be a string expression that specifies the file to open or the folder to open or explore. The statement can open an executable file or a document file. |
| <i>parameter</i> | must be a string expression that specifies parameters to be passed to the application (if <i>file-name</i> specifies an executable file). If <i>file-name</i> specifies a document file, <i>parameter</i> should be an empty string. |
| <i>show-mode</i> | specifies how the application is to be shown when it is opened. This parameter can be one of the following values: |
- | | |
|-----------------|--|
| "MIN" | Minimizes the specified window and activates the next top-level window in the Z order. |
| "MAX" | Maximizes the specified window. |
| "HIDE" | Hides the window and activates another window. |
| "ICON" | Displays the window as a minimized window. The active window remains active. |
| "NORMAL" | Activates the window and displays it in its current size and position. |
- EXAMPLE** PASS "NOTEPAD.EXE", "", "NORMAL"
 MESSAGE "", "NOTEPAD CLOSED"
- The PASS statement will run Notepad and the next statement MESSAGE will be executed after Notepad is closed.

PLAY Statement

FUNCTION PLAY plays a sound specified by the given filename.

VERSIONS 1.0 and above

FORMAT **PLAY** *file-name*, *play-mode*

REMARKS *file-name* is a string expression, which names the wave file to be played.

play-mode is a string expression that specifies the mode of play to the wave file. This parameter can be one of the following values:

""	Attempts to stop the currently playing sound and plays the specified sound.
"LOOP"	Attempts to stop the currently playing sound and plays the specified sound repeatedly.
"NOSTOP"	Plays the specified sound but doesn't attempt to stop the currently playing sound. If a sound cannot be played because the resource needed to generate that sound is busy playing another sound, the statement immediately return and without playing the requested sound.

The sound is played asynchronously and PLAY returns immediately after beginning the sound. To terminate an asynchronously played waveform sound, call PLAY with *file-name* set to empty string.

The sound specified by *file-name* must fit into available physical memory and be playable by an installed waveform-audio device driver. *parameter* must be a string expression that specifies parameters to be passed to the application (if *file-name* specifies an executable file). If *file-name* specifies a document file, *parameter* should be an empty string.

EXAMPLE PLAY "Siren-01.wav", "LOOP"

RAND() Function

FUNCTION RAND returns a random number.

VERSIONS 1.0 and above

FORMAT $Y = \text{RAND}(\text{numeric-expression})$

REMARKS *file-name* is a string expression, which names the wave file to be played.
numeric-expression may be any numeric expression.

This function returns a random number between 0 and the result of *numeric-expression*.
 The random number generator is auto re-seeded, so the different sequence of random numbers is generated each time the program is run.

EXAMPLE $Y = \text{RAND}(5.5)$

RD Statement

FUNCTION RD is used to delete an empty folder.

VERSIONS 4.0 and above

FORMAT **RD** *directory-name*

REMARKS *directory-name* is a string expression representing the path of the folder to be deleted.

RD statement may fail if the specified folder does not exist or is not empty. Such failure will be counted as an internal error but the script will continue its execution. ERRID() function can be used to get the error code.

EXAMPLE RD "C:\\EmptyFolder"

READ Statement

FUNCTION READ reads data from the specified file.

VERSIONS 1.0 and above

FORMAT **READ** *file-number, string-variable, count-expression*

REMARKS *file-number* is an integer expression, which evaluates to a file number.
string-variable is a string variable that receives the data read from the file.
count-expression is an integer expression, which evaluates to a count value. The count value specifies the number of bytes to be read from the file.

If the file associated with the specified file number was successful opened, this statement reads data from the file. If the file is a communications file, this statement reads data from the receive-queue.

EXAMPLE READ 1, A\$, 80

RETURN Statement

FUNCTION RETURN returns from a subroutine.

VERSIONS 1.0 and above

FORMAT RETURN

REMARKS The RETURN statement(s) in a subroutine cause SmartScript Module to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic-dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, it may be preceded by a STOP, END or GOTO statement that directs program control around the subroutine.

EXAMPLE GOSUB ShowMsg:

```
MESSAGE "Test Program", "In main program"
END

ShowMsg:
    MESSAGE "Test Program", "In subroutine"
RETURN
```

RIGHT\$() Function

FUNCTION RIGHT\$ returns a string comprised of the rightmost *N* characters of the specified string.

VERSIONS 1.0 and above

FORMAT *M\$* = **RIGHT\$(string-expression, count-expression)**

REMARKS *string-expression* may be any string expression.
count-expression is an integer expression, which evaluates to a valid count value.
The valid count value is 0 to maximum length of the string..

EXAMPLE *M\$* = RIGHT\$("ABCDEFGH", 4) // *M\$* will be "DEFG"END

RSTERR Statement

- FUNCTION** RSTERR is used to reset the internal error counter of SmartScript module.
- VERSIONS** 4.0 and above
- FORMAT** **RSTERR**
- REMARKS** SmartScript has a counter to count the number of internal error occurs. Whenever a statement or a function cause an internal error, the counter will be incremented by 1 and NERR() can be used to return the error count. Execution of the statement RSTERR will reset the value of the counter to 0. After the execution of this statement, NERR() will return 0.
- EXAMPLE** RSTERR // NERR() = 0.

RTRIM\$() Function

- FUNCTION** RTRIM\$ returns a string same as the specified string but without the trailing Space (ASCII code=32) and control characters including Tab (ASCII code=9), Carriage Return(ASCII code=13) and Line Feed (ASCII code=10).
- VERSIONS** 1.0 and above
- FORMAT** *M\$* = **RTRIM\$(string-expression)**
- REMARKS** *string-expression* may be any string expression.
- EXAMPLE** M\$ = RTRIM\$("ABCDEFGG ") // M\$ will be "ABCDEFGG"

SECOND() Function

- FUNCTION** SECOND returns the current seconds after minute.
- VERSIONS** 1.0 and above
- FORMAT** *N%* = **SECOND()**
- REMARKS** This function returns the current seconds after minute (0 ~ 59).
- EXAMPLE** N% = SECOND()

SEEK Statement

FUNCTION SEEK moves the read-write pointer of the specified file.

VERSIONS 1.0 and above

FORMAT SEEK *file-number*, *count-expression*, *seek-mode*

REMARKS *file-number* is an integer expression, which evaluates to a file number.

count-expression is an integer expression, which evaluates to a count value. The count value specifies the number of bytes the read-write pointer is to be moved.

seek-mode specifies the starting position and direction of the read-write pointer. This parameter can be one of the following values:

"B"	Moves the read-write pointer <i>count-expression</i> bytes from the beginning of the file.
"C"	Moves the read-write pointer <i>count-expression</i> bytes from its current position.
"E"	Moves the read-write pointer <i>count-expression</i> bytes from the end of the file.

If the file associated with the specified file number was successfully opened, it moves the read-write pointer of the file to the specified position. If the file is a communications file, this statement flushes all characters from the receive-queue (*seek-mode* = "C"), transmit-queue (*seek-mode* = "B") or both (*seek-mode* = "E").

When a file is initially opened, the read-write pointer is set to the beginning of the file. SEEK moves the pointer an arbitrary amount without reading data, which facilitates random access to the file's contents.

EXAMPLE SEEK 1, 0, "E"

SETDIR Statement

FUNCTION SETDIR can set the path of *Reference Path* (~0 , ... , ~19) during Lab-LINK runtime

VERSIONS 1.1 and above

FORMAT **SETDIR** *numeric-expression*, *string-expression*

REMARKS *numeric - expression* is a numeric expression with value between 0 and 19. It is used to specify which reference path will be defined.

string - expression is a string expression representing a valid path of a folder. This path is used to define the specified reference path.

Reference Path is a shorthand notation of file path used by Lab-LINK objects. Please see Appendix of SmartPanel Manual for details.

SETDIR can define or redefine the actual path of the specified Reference Path at Lab-LINK runtime. Note that the reference path used by some of the Lab-LINK modules are loaded when Lab-LINK starts and modification of these reference paths at later time will have no effect to these modules. For example, the Data module use reference path ~6 as its default data path and will load the definition of ~6 when Lab-LINK starts. Any change on the reference path ~6 after Lab-LINK starts will not change the path of data files.

EXAMPLE // Set reference path ~8 to ..\project\proj1\dat\01

```
SETDIR 8, "..\PROJECT\PROJ1\DAT\01"
```

SHORTCUT Statement

FUNCTION SHORTCUT is used to create a shortcut for an application or a document file.

VERSIONS 4.0 and above

FORMAT **SHORTCUT** *link-file,filename,parameter,work-directory*

REMARKS	<i>link-file</i>	is a string expression representing the file name of the shortcut.
	<i>filename</i>	is a string expression representing the file name of the application or the document file of the shortcut. The file name should include complete path designation
	<i>parameter</i>	Any parameter required by the application or the document file.
	<i>work-directory</i>	Specify the working directory of the specified application or document.

Shortcut statement may fail if it contains invalid path. Such failure will be counted as an internal error but the script will continue its execution. ERRID() function can be used to get the error code.

If the shortcut file specified by *link-file* already exists, SHORTCUT statement will create a new shortcut to replace the old one.

SHORTCUT statement does not check for correctness of *filename*, *parameter* and *work-directory*. Therefore, it is possible that SHORTCUT statement has successfully created the shortcut but the shortcut cannot be executed correctly. Developers are responsible to verify the validity of these arguments.

EXAMPLE SHORTCUT "Test.lnk","DBSaver.exe","PROJ1 WKS1","C:\\LabLink\\System4\\"

SHUTDOWN Statement

FUNCTION SHUTDOWN can end the execution of Lab-LINK.

VERSIONS 4.0 and above

FORMAT **SHUTDOWN**

REMARKS This command is used to end the execution of Lab-LINK system. Although there is a system tag named \$EXIT can end the execution by setting its value to 1, it is not recommended. It is possible that some Lab-LINK modules may not be terminated when setting \$EXIT to 1 in Smart Script. Therefor, it is recommended to use this statement to end Lab-LINK in Smart Script.

EXAMPLE // End Lab-LINK
SHUTDOWN

SIN() Function

FUNCTION SIN returns the sine of the specified angle.

VERSIONS 1.0 and above

FORMAT $Y = \text{SIN}(\text{numeric-expression})$

REMARKS *numeric-expression* must be an angle expressed in radians

This function is equivalent to the algebraic expression $y = \sin(x)$. To convert from degrees to radians, multiply degrees by (PI / 180).

EXAMPLE $Y = \text{SIN}(90 * (\text{PI} / 180))$ // Y will be 1

SINH() Function

FUNCTION SINH returns the hyperbolic sine of the specified value.

VERSIONS 1.0 and above

FORMAT $Y = \text{SINH}(\text{numeric-expression})$

REMARKS *numeric-expression* must be an angle expressed in radians

This function is equivalent to the algebraic expression $y = \sinh(x)$.

EXAMPLE $Y = \text{SINH}(1)$ // Y will be 1.175201

SLEEP Statement

FUNCTION SLEEP is used to suspend the execution of the script for a specified period of time.

VERSIONS 4.0 and above

FORMAT **SLEEP** *numeric expression*

REMARKS *numeric expression* is a numeric expression. It must be greater than 0 and its unit is second.

EXAMPLE SLEEP 1.5 // Suspend the script for 1.5 seconds.

SQRT() Function

FUNCTION SQRT returns the square root of the specified value.

VERSIONS 1.0 and above

FORMAT $Y = \text{SQRT}(\text{numeric-expression})$

REMARKS *numeric-expression* is a numeric expression and must be greater than or equal to zero.

This function is equivalent to the algebraic expression $y = \sqrt{x}$

EXAMPLE Y = SQRT(2) // Y will be 1.414214

STOP Statement

FUNCTION STOP terminates program execution and closes all files.

VERSIONS 1.0 and above

FORMAT **STOP**

REMARKS STOP statements may be placed anywhere in the program to terminate execution.

EXAMPLE IF (A > 0) => STOP

STR\$() Function

FUNCTION STR\$ returns a string representation of the specified value.

VERSIONS 2.0 and above

FORMAT M\$ = **STR\$(numeric-expression)**

REMARKS *numeric-expression* may be any numeric expression.

EXAMPLE M\$ = STR\$(5.3124) // M\$ will be "5.3124"

STRING\$() Function

FUNCTION STRING\$ returns a string formed by repeating the specified character a specified number of times.

VERSIONS 1.0 and above

FORMAT M\$ = **STRING\$(string-expression, count-expression)**

REMARKS *string-expression* may be any string expression. The first character in the string expression will be repeated.

count-expression is an integer expression, which evaluates to a count value. The count value specifies the number of times to be repeating.

EXAMPLE M\$ = STRING\$("*", 5) // M\$ will be "*****"

SUM08() Function

FUNCTION SUM08 returns the 8-bit checksum value of the specified string.

VERSIONS 1.0 and above

FORMAT N% = **SUM08(string-expression)**

REMARKS *string-expression* may be any string expression.

EXAMPLE N% = SUM08("\x02ABC\x03") // N% will be 203

SWITCH ... CASE ... DEFAULT ... ENDSW Statement

FUNCTION	SWITCH evaluates expression and executes any statement associated with case-expression whose value matches the initial expression.	
VERSIONS	1.0 and above	
FORMAT	SWITCH (<i>switch-expression</i>) [CASE <i>case-expression</i>] ... [CASE <i>case-expression</i>] ... [DEFAULT] ... ENDSW	
REMARKS	<i>switch-expression</i>	may be any numeric expression.
	<i>case-expression</i>	may be any numeric expression.
	The SWITCH and CASE keywords evaluate switch-expression and execute any statement associated with case-expression whose value matches the initial switch-expression. If there is no match with a case-expression, the statement associated with the DEFAULT keyword is executed. If the DEFAULT keyword is not used, control passes to the statement following the ENDSW.	
EXAMPLE	<pre> SWITCH (N%) CASE 0 Y = SIN(X) CASE 1 Y = COS(X) DEFAULT Y = 0.0 ENDSW </pre>	

SYSINFO\$() Function

FUNCTION SYSINFO() returns the specified system information.

VERSIONS 4.0 and above

FORMAT S% = **SYSINFO\$**(*info-name*)

REMARKS *info-name* a string expression which must be one of the following keyword used to specified the system information returned.

HOSTNAME	return the computer name
HOSTADDR	return the IP address. If there are more than one IP addresses associated with the computer, the retuened IP addresses will be separated with SPACE character.

EXAMPLE S\$ = SYSINFO\$("HOSTNAME") // N% is "COMPUTER1"
 T\$ = SYSINFO\$("HOSTADDR") // N% is "192.168.100.1"

TAG() Function

FUNCTION TAG returns the value or message of the specified Tag .

VERSIONS 1.1 and above

FORMAT $Y = \text{TAG}(\text{string-expression})$ or
 $\text{TAG}(\text{string-expression}) = \text{expression}$

REMARKS *string-expression* is a string expression which is a valid Tag name or Tag name with postfix ".\$".
expression may be any numeric or string expression. Its data type should be consistent with the tag field specified by the Tag name *string-expression*.

If the Tag specified by tag name *string-expression* does not exist, it will be created automatically at runtime. The tag name *string-expression* should comply with the naming rule of Lab-LINK Tag. When the last two characters in *string-expression* are ".\$", it indicates that the message field of the specified tag is used in this function. If the last two characters in *string-expression* are ".t" or ".T", it indicates that the date/time field of the specified tag is used in this function.

If the result of *expression* is a string, note that there is a length limit of 80 characters for the message field of a Tag.

When TAG function is placed on the right of the operator "=" in a statement, it will returns the value or message of the specified tag. When TAG function is placed on the left of the operator "=" in a statement, it will assign the result of the expression on the right to the value or message field of the specified tag.

EXAMPLE TAG("Tag"+"1") = 3.9 // Set the value of "Tag1" to 3.9
 TAG("Tag"+"1"+".\$") = "OK" // Set the message of "Tag1" to OK
 {Tag2} = TAG("Tag1") // Assign the value of "Tag1" to "Tag2"

TAN() Function

FUNCTION TAN returns the tangent of the specified angle.

VERSIONS 1.0 and above

FORMAT $Y = \text{TAN}(\text{numeric-expression})$

REMARKS *numeric-expression* must be an angle expressed in radians

This function is equivalent to the algebraic expression $y = \tan(x)$. To convert from degrees to radians, multiply degrees by (PI / 180).

EXAMPLE $Y = \text{TAN}(45 * (\text{PI} / 180))$ // Y will be 1.0

TANH() Function

FUNCTION TANH returns the hyperbolic tangent of the specified value.

VERSIONS 1.0 and above

FORMAT $Y = \text{TANH}(\text{numeric-expression})$

REMARKS *numeric-expression* may be any numeric expression.

This function is equivalent to the algebraic expression $y = \tanh(x)$.

EXAMPLE $Y = \text{TANH}(1)$ // Y will be 0.761594

TICK() Function

FUNCTION TICK returns the number of milliseconds that have elapsed since Windows was started.

VERSIONS 1.0 and above

FORMAT $Y = \text{TICK}()$

REMARKS The return value is the number of milliseconds that have elapsed since Windows was started. The elapsed time is stored as a 32-bit value, which means Windows can record no more than 2^{32} millisecond intervals before the 32-bit value overflows to zero. This is approximately 49.7 days. If you use the elapsed time, check for the overflow condition when comparing times.

EXAMPLE $Y = \text{TICK}()$

TIMER() Function

FUNCTION TIMER returns the current seconds after midnight.

VERSIONS 1.0 and above

FORMAT $N\% = \text{TIMER}()$

REMARKS The return value is the number of seconds that have elapsed since midnight.

EXAMPLE $N\% = \text{TIMER}()$

TOKEN Statement

FUNCTION TOKEN removes the first token that delimited by the delimiters from the specified string.

VERSIONS 1.0 and above

FORMAT **TOKEN** *token-variable*, *source-variable*, *delimiters*

REMARKS *token-variable* is a string variable that will receive the token.

source-variable may be any string variable.

delimiters is a string expression which contains the list of delimiters.

This statement searches for the occurrence of any delimiter (specified by *delimiters*) in the source string, and breaks it to two separated strings. The first string assigns to the *token-variable*, and the second assigns back to the *source-variable*.

EXAMPLE A\$ = "Color = 255, 192, 128"

TOKEN M\$, A\$, "="	// M\$ will be "Color "	A\$ will be " 255, 192, 128"
TOKEN M\$, A\$, ","	// M\$ will be " 255"	A\$ will be " 192, 128"
TOKEN M\$, A\$, ","	// M\$ will be " 192"	A\$ will be " 128"
TOKEN M\$, A\$, ","	// M\$ will be " 128"	A\$ will be ""

TONE Statement

FUNCTION TONE generate a sound with specified frequency and length using the built-in buzzer of the PC.

VERSIONS 4.0 and above

FORMAT TONE *frequency, duration*

REMARKS *frequency* is a numeric expression specifying the frequency of the sound played. Its unit is Hz.

duration is a numeric expression specifying the length of the sound played. Its unit is second.

Script will wait for the end of the sound playing before executing statements following TONE statement.

EXAMPLE TONE 440,3 // Play a sound of frequency 440Hz for 3 seconds

TRAPOFF Statement

FUNCTION TRAPOFF disables the TAG events trapping.

VERSIONS 4.0 and above

FORMAT TRAPOFF

REMARKS This statement disables the TAG events trapping and all TAG events will be ignored.

After the execution of this statement, despite that all other new TAG event won't be executed, these new event will be stored in a TAG event queue and will be executed after TRAPON statement is executed to restore TAG event handling. Since there is a limit on the capacity of the TAG event queue, it is suggested to keep the duration of TRAPOFF as short as possible. Otherwise, an out of stack error may occur if there are too many unexecuted TAG events.

Refer to "Line Label" in chapter 1 for TAG event.

EXAMPLE TRAPOFF

TRAPON Statement

FUNCTION TRAPON enables the TAG events trapping.

VERSIONS 4.0 and above

FORMAT **TRAPON**

REMARKS This statement enables the TAG events trapping and all TAG events will be processed. Refer to “Line Label” in chapter 1.

EXAMPLE TRAPON

UPPER\$() Function

FUNCTION UPPER\$ returns a string same as the specified string but all characters were converted to uppercase.

VERSIONS 1.0 and above

FORMAT *M\$ = UPPER\$(string-expression)*

REMARKS *string-expression* may be any string expression.

EXAMPLE *M\$ = UPPER\$("ABCDefg") // M\$ will be "ABCDEFGG"*

VAL() Function

FUNCTION VAL returns the numerical value of the specified string.

VERSIONS 1.0 and above

FORMAT *Y = VAL(string-expression)*

REMARKS *string-expression* may be any string expression

If the first characters of string-expression are not numeric, then VAL returns 0. VAL also strips leading blanks, tabs, carriage-returns and line-feeds from the specified string.

EXAMPLE *Y = VAL(" -3.84 ") // Y will be -3.84*

WEEKDAY() Function

FUNCTION WEEKDAY returns the current day of week.

VERSIONS 1.0 and above

FORMAT *N%* = **WEEKDAY()**

REMARKS This function returns the current day of week (0 ~ 6, Sunday = 0).

EXAMPLE *N%* =WEEKDAY()

WHILE ... LOOP Statement

FUNCTION WHILE executes a series of statements in a loop as long as a given condition is true.

VERSIONS 1.0 and above

FORMAT WHILE (cond-expression)

...

LOOP

REMARKS *cond-expression* may be any expression.

If cond-expression is true (non-zero), the statements in the loop are executed until the LOOP statement is encountered. Control Script Module then returns to the WHILE statement and checks cond-expression again. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the LOOP statement.

WHILE ... LOOP may be nested to any level. Each LOOP will match the most recent WHILE.

EXAMPLE //Increment the value of X by 1 repeatedly until N is not equal to 1

WHILE(N == 1)

 X = X + 1

LOOP

WRITE Statement

FUNCTION WRITE writes data to the specified file.

VERSIONS 1.0 and above

FORMAT **WRITE** *file-number*, *string-expression*

REMARKS *file-number* is an integer expression, which evaluates to a file number.
string-expression is a string expression that containing the data to be written to the file.

If the file associated with the specified file number was successful opened, this statement writes data to the file. If the file is a communications file, this statement writes data to the transmit-queue.

EXAMPLE WRITE 1, "This is a test.\r\n"

XOR08() Function

FUNCTION XOR08 returns the 8-bit exclusive OR checksum value of the specified string.

VERSIONS 1.0 and above

FORMAT *N%* = **XOR08**(*string-expression*)

REMARKS *string-expression* may be any string expression.

EXAMPLE *N%* = XOR08("\x02ABC\x03") // *N%* will be 65

YEAR() Function

FUNCTION YEAR returns the current year.

VERSIONS 1.0 and above

FORMAT N% = YEAR()

REMARKS This function returns the current year.

EXAMPLE N% = YEAR()

Appendix A Environment Limits

❑ Name, String, and Number Limits

	Maximum	Minimum
Variable-name length	16 characters	1 character
String length	32,767 characters	0 character
Integer	-2147483648	2147483647
Real:		
Positive	1.79769313486231D+308	4.940656458412465D-324
Negative	-4.940656458412465D-324	-1.79769313486231D+308

❑ Array Limits

	Maximum	Minimum
Array size (all elements):	32,767 bytes (32K)	1 byte
Number of dimensions allowed	3	1
Array subscript value	8 , 192	1

❑ Procedure and File Limits

	Maximum	Minimum
Program file size	2G bytes	0 byte
Procedure size	65535 lines	0 line
Number of line labels	65535	0
Number of TAG label	1024	0
Number of variables	65535	0
Data file numbers	16	1
Level of nest	512	0

Appendix B Keywords

ABS()	ACOS()	ALARM()
ALMGRP()	ALMPRI()	ALMTAG\$()
ASC()	ASIN()	ATAN()
BEEP	CASE	CD
CHOICE()	CHR\$()	CLOSE
COMMODE	COMOPEN	CONTINUE
COPY	COS()	COSH()
CRC16()	CRC32()	CREATE
DATETIME\$()	DAY()	DEFAULT
DEL	DIM	DIR\$()
ELSE	ELSEIF	END
ENDIF	ENDSW	ERRID()
ERRORTAG	EXEC	EXIT
EXP()	FAC()	FCHECK()
FILE\$()	FLEN()	FMBCD()
FMDBL()	FMFLT()	FOR
FORMAT\$()	FPOS()	FPRINT
GOSUB	GOTO	HOUR()
IDLE	IF	INT()
ISTR\$()	IVAL()	LEFT\$()
LEN()	LN()	LOG()
LOOP	LOWER\$()	LTRIM\$()
MAX()	MD	MESSAGE
MID\$()	MIN()	MINUTE()
MONTH()	MOVE	MSGBOARD
NERR()	NOW()	NOW\$()
OPEN	PASS	PI
PLAY	RAND()	RD
READ	RETURN	RIGHT\$()
RSTERR	RTRIM\$()	SECOND()
SEEK	SETDIR	SHORTCUT
	SIN()	SINH()
SLEEP	SQRT()	STOP

STR\$()	STRING\$()	SUM08()
SWITCH	SYSINFO\$()	TAG()
TAN()	TANH()	TICK()
TIMER()	TOKEN	TONE
TRAPOFF	TRAPON	UPPER\$()
VAL()	WEEKDAY()	WHILE
WRITE	XOR08()	YEAR()

Appendix C Operator Precedence

Operator	Operation	Direction	Precedence
-	Negation		High ↑ ↓ Low
!	Logic NOT	←	
~	Invert		
^	Exponentiation	→	
*	Multiplication		
/	Division	→	
\	Integer Modulus		
+	Addition	→	
-	Subtraction	→	
<<	Shift left		
>>	Shift right	→	
^<	Rotate left		
>^	Rotate right		
<	Less than		
>	Greater than	→	
<=	Less than or equal to		
>=	Greater than or equal to		
==	Equality	→	
!=	Inequality		
AND	AND	→	
XOR	Exclusive OR	→	
OR	OR	→	
&	Logic AND	→	
	Logic OR	→	

Appendix D Error Codes

Code	Message
1	Out of Memory !!
2	Too many Variable !!
3	Too many Constant !!
4	Too many Label !!
5	Out of Operator Stack !!
6	Invalid Line Label !!
7	Invalid Command !!
8	Syntax Error !!
9	Invalid String Constant !!
10	Invalid Variable !!
11	Expression too Complex !!
12	Extra Argument !!
13	Type Mismatch !!
14	Array Error !!
15	Math Error !!
16	Expression Error !!
17	Undefined Label !!
18	Division by Zero !!
19	RETURN without GOSUB !!
20	ENDIF without IF !!
21	IF without ENDIF !!
22	ENDSW without SWITCH !!
23	SWITCH without ENDSW!!
24	LOOP without FOR or WHILE !!
25	FOR or WHILE without LOOP !!
26	Invalid Option !!
27	Program no Compiled !!
28	End of Program !!
29	Program Break !!
30	Out of Range !!
31	Out of String Length !!
32	Operation fail !!
33	Variable redefined !!
34	File in used !!
35	File not open !!
36	Invalid Tag !!
37	Inner Error !!