TRIANGLE RESEARCH INTERNATIONAL

i-TRiLOGI

Ladder+BASIC

Version 7.0

(Floating-point Edition)

Programmer's Reference Addendum

Revision Sheet

Release No.	Date	Revision Description
1.0	12/1/2013	First release
1.1	2/4/2013	Release software version 7.01 and new RND function

Conditions of Sale and Product Warranty

Triangle Research International Inc. (TRi) and the Buyer agree to the following terms and conditions of Sale and Purchase:

1. TRi Programmable Controllers and Peripherals (the Products) are guaranteed against defects in materials or workmanship for a period of one year from the date of registered purchase. Any unit which is found to be defective will, at the discretion of TRi, be repaired or replaced.

2. TRi will not be responsible for the repair or replacement of any Product damaged by user modification, negligence, abuse, improper installation, or mishandling.

3. TRi is not responsible to the Buyer for any loss or claim of special or consequential damages arising from the use of the product. The Products are NOT certified to be FAILSAFE and hence must **NOT** be used in applications where failure of the Product could lead to physical harm or loss of human life. Buyer is responsible to conduct their own tests to meet the safety regulation of their respective industry.

4. Products distributed, but not manufactured by TRi, carry the full original manufacturers warranty. Such products include, but are not limited to: power supplies, sensors, I/O modules and battery backed RAM.

5. TRi reserves the right to alter any feature or specification at any time.

Notes to Buyer: If you disagree with any of the above terms or conditions you should promptly return the unit to the manufacturer or distributor within 30 days from date of purchase for a full refund.

TABLE OF CONTENTS

Chapter 1 Introduction to I-TRiLOGI 7

1 Introduction – What's New in TL7?

1.1 Overview

This manual for I-TRiLOGI Version 7.xx (TL7) is written as an addendum to, and is intended to be used in tandem with, the i-TRiLOGI Version 6.xx (TL6) Reference Manual which can be downloaded from:

http://www.triplc.com/documents/TL6ReferenceManual.pdf

By structuring the reference manual as an addendum instead of as its own standalone manual, we attempt to provide a clear picture to existing and new users the differences between the two groups of Super PLCs – One group can only be programmed using TL6 and a second group that can make use of the new features available in TL7.

1.2 Floating-point Math Support

The most important capability added to TL7 is the support of **floating-point math** operations. The detail on this capability will be explained in later chapters in this manual.

It is important to realize that TL7 **does not add** floating-point capability to those PLCs that do not support it. It is only meant for programming PLCs that already has the firmware to support the floating-point math and variables.

The following table shows the current PLC models and whether they can be programmed with TL6 or TL7:

PLC Model	i-TRiLOGi
1. Nano-10, FMD88-10, FMD1616-10	Version 6 only
2. F2424, F1616-BA and FCPU	Version 6 only
3. FX2424, FX1616-BA & Fx CPU	Version 7 & Version 6

Of course more new PLCs model may be added in future that are not shown in the above table. Note that PLCs that supports floating-point operation will have firmware >= F90.0 whereas PLC that supports only integers will have firmware <= r8x .

1.3 Local Variables & Function Parameters Passings

Before TL7, all variables are global in scope, which means all variables are accessible to all custom functions, and that changes to any variable inside a custom function will affect all other custom functions that use the same variable. Since all variables are global their value can be monitored and changed via the online monitoring screen.

TL7 now supports the concept of local variables and parameter passing to and from a custom function to make it simpler to create a library of portable custom functions.

Up to 9 floating-point local variables: %[1] to %[9] are now available to a custom function which are local in scope. Since these are elements of a one-dimensional array they can be easily accessed using their index.

In TL7 you can now call a custom function with label name: "MyFunc" by one of the following methods:

1	`CALL MyFunc   ' same as TL6`
2	`FP[1]  =  MyFunc (p1, p2, p3……p9)`
3	`D# =  MyFunc( ) ' no parameter`
4	`Y#  = Function (CusF#, p1, p2, p3…..p9)`

Method 1 is the only method available in TL6 where no parameter is passed to the called function and no parameters are returned. All parameters that MyFunc needs can only be passed via global variables (A to Z, DM[n] and EMINT[n], etc) and what MyFunc need to return to the caller are also passed via global variables. TL7 continues to support this method to maintain compatibility and also for calling functions that do not require parameters passing.

Method 2, 3 are new and allows you to call your custom function by its label name as if it is a built-in function and you can pass the parameters p1, p2, p3… etc enclosed between the parentheses. The variables are separated by comma and you can pass from zero up to a maximum of 9 variables to the called function "MyFunc()". When called this way the function is expected to return a data back to the caller which is assigned to a variable (Y# in this example)

Method 4 is also new and allows you to call a function by using the new keyword "FUNCTION" and specify the custom function by its number or its label name as the first parameter. The actual parameters to be passed are contained from second parameter onwards. It is compiled to the same code as Method 2 and 3.

Parameters Passing

The parameters p1 to p9 can only be either integer or float data type. Integer are automatically converted to float when passed to MyFunc() as part of the parameter list.

The parameters that are passed to MyFunc() are accessible by MyFunc() as the local variables %[1] to %[9]. Basically p1 is passed to %[1], p2 is passed to %[2] and so on.

E.g. `Y# = MyFunc(0.25, 100, DM[1]/3.5)`

When the custom function MyFunc() runs, it will find the following variables already contain the data passed in the parameter list:

```
%[1] = 0.25;  %[2] = 100.0  ; %[3] = result of DM[1]/3.5
```

In the above examples only 3 parameters are passed. So %[4] to %[9] are not used but are always available to the custom function as local variables.

Your custom function "MyFunc" can use any of these 9 local variables without restriction and changes made to these variables does not affect other custom functions. This is what is meant by "local variable" since the local variables are only valid when the function is called and once the custom function ends these variables will no longer contain valid data.

Needless to say any local variables used in MyFunc that is not passed as a parameter must be initialized with valid values before being used since they may contain stale data resulted from execution of the other custom functions.

Returning Data To Calling Function

As explained above in a statement such as

```
Y# = MyFunc(p1, p2, p3)
```

Y# will be assigned a float data returned from MyFunc(). MyFunc will return the data using the following syntax:

```
RETURN = [data to be returned]
```

E.g.
```
RETURN = %[7]
RETURN = FP[20] / %[5]
```

If no valid data need to be returned then a simple RETURN statement suffice but the calling function will receive a stale data that it should discard.

Using #defineLocal

Although the local variables %[1] to %[9] can be directly used in any expression and you may also attach a comment to the variable name such %[3]_index, %[5]_opensasame etc to make its purpose more self-explanatory, a more complex mathematical formula may be more readable if it is written with well known variable names. A new keyword #defineLocal has been introduced in I-TRiLOGI version 7 (which could be ported to I-TRiLOGI version 6.46 b2 onwards by the time you read this document) to simplify programming task in some circumstances.

Although you can still use the global #Define table as in I-TRiLOGI version 6.4x to define a unique label name for each of the local variable used in each custom function, you might find it easier to use the new #defineLocal keyword to define a unique label name to represent the local variable or other expressions.

#defineLocal has only a local scope (i.e. it is only valid within the custom function that it is being defined) and is convenient to use since it can be copied along with the rest of the custom function and pasted into a new program. However, unlike the global #Define label, the help pane does not show up the content of a selected label name defined using the #defineLocal keyword. So we recommend attaching a suffix to a local variable name defined by the #defineLocal keyword:

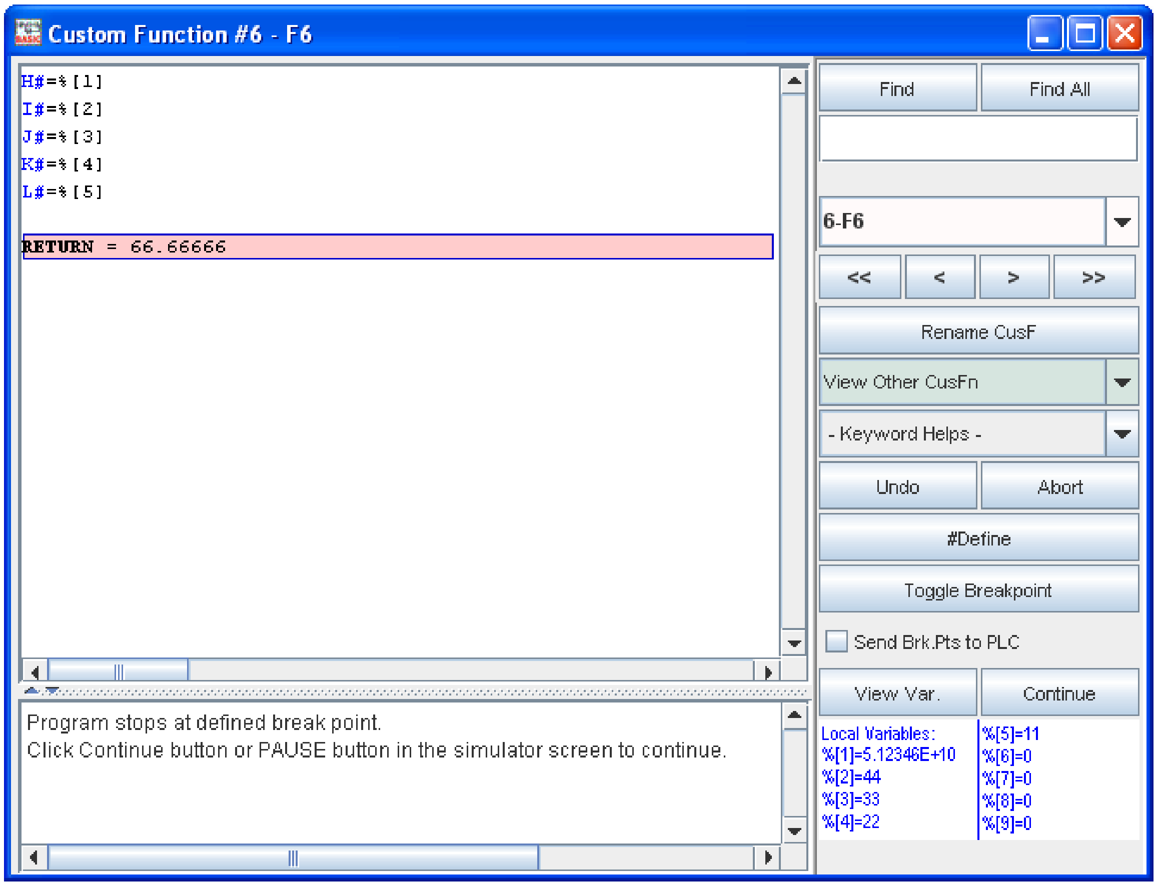
E.g. you can use x1, y2, Z3 etc to represent local variables %[1], %[2] and %[3] etc. The suffix lets you immediately identify the local variable to look for during debugging since all the nine local variables are visible when the program stops at a break within a custom function.

This also prevents the locally defined label name from masking the actual variable names such as X, Y, Z etc.

Monitoring Local Variables

TL6 users may have enjoyed being able to view the value of any global variables via the online monitoring screen at anytime, but may be surprised to learn that it is not quite possible to do the same for the local variables. This is because local variables are only valid within a custom function when it is being run and since the program contains many custom functions the online monitoring screen could not readily associate the local variables to any particular function except when it is being run.

In order to view the values of local variables, you will need to use the same "Breakpoint" feature found in I-TRiLOGI version 6.45. When a custom function is run and reaches a breakpoint the program pauses and the custom function will open at the break point, and you will be able to see the values of the 9 local variables on the right hand panel of the custom function editor windows, as shown below:



```
H#=%[1]
I#=%[2]
J#=%[3]
K#=%[4]
L#=%[5]

RETURN = 66.66666
```

```
Program stops at defined break point.
Click Continue button or PAUSE button in the simulator screen to continue.
```

```
Local Variables:   %[5]=11
%[1]=5.12346E+10   %[6]=0
%[2]=44            %[7]=0
%[3]=33            %[8]=0
%[4]=22            %[9]=0
```

1.4 TL7 and TL6 Compatibility Issues

TL7 is designed to maintain100% backward compatibility to TL6. i.e. Any program written in TL6 can be imported into TL7, compiled and uploaded to the new PLC unmodified.

In addition, the PLC models that support floating-point operation are also programmable by TL6 (obviously without floating-point). This is to ensure true backward compatibility with Super PLCs that support only integer math. This results in a smooth upgrade path for users of existing Super PLCs that do not support float to upgrade to a new PLC model that supports float.

Users of TL6 are aware that TL6 only supports 32-bit integer math operations and 16 or 32-bit integer variables. In order to maintain true backward compatibility to TL6, TL7 simultaneously support integer and floating-point math and automatically does the necessary type casting (i.e. conversion) between the two data types.

That is to say, TL7 does not automatically treat all numerical variables as floating-point data type (aka "float") or perform only float operation. Integer variables that are assigned integer value will continue to keep the data as pure integer with up to 32-bit precision. Mathematical operators that support both integer and float (such as addition, division, etc) will elect to perform the integer operation or float operation at run time based on the operand data type. E.g. a "divide" operator ("/") will perform integer divide on two integer operands but will perform a floating-point divide if any of the two operands is a float. If one of the operand is a float and the other is an integer, the integer operand will be converted to float before float division is performed.

If you supply a float variable or a float constant to any command or function that expects an integer (e.g. the index of a DM[ ] or the channel number of a SETDAC command), TL7 will dynamically convert the float data to integer first and perform the operation as if they are integer data.

E.g. The following statements are valid:

```
FOR %[5] = 1.0 to 10.0
   DM[%[5]] = %[6] * 2.15
NEXT
```

Even though **`%[5]`** (local variable) is a float it will be converted into an integer and hence can be used as an index to access DM[ ].

Compatibility Exception

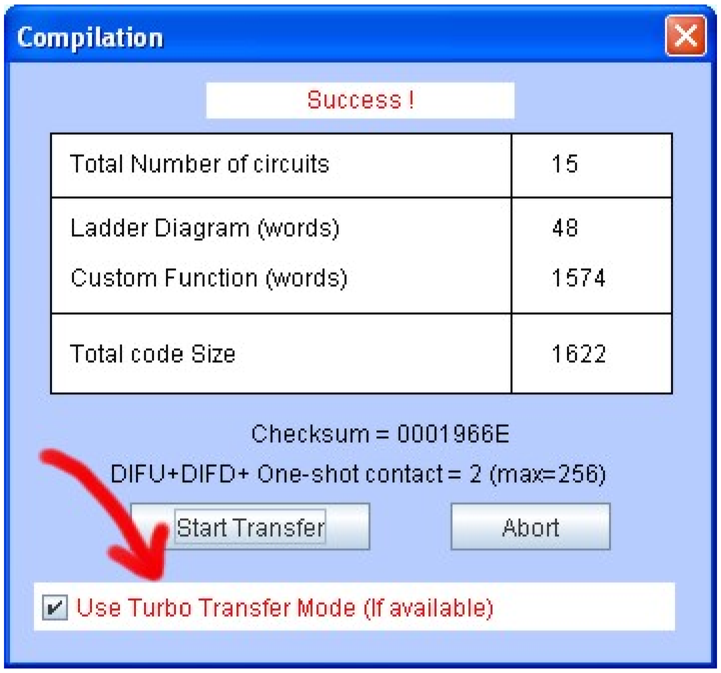
As mentioned earlier, in most mathematic expressions TL7 will handle arithmetic involving floating-point differently than those involving pure integer in order to maintain backward compatibility with TL6.

The only exceptions are the PIDcompute and PIDDef functions, where the E, P, I, and D parameters as well as all internally stored Integral and derivative values are inherently floating-point data. If your program supplies an integer data to a PIDCompute or PIDDef function it will first be converted into float and all PID computation will only be carried out in float.

This however should not become an issue for user upgrading from PLCs without floating-point to one with floating-point, since a floating-point number should be able to represent the range of numbers typically used for PID control, and one of the main reasons for moving to a floating-point PLC is often that it is simpler to handle PID calculation using floating-point than fixed point arithmetic.

1.5 Turbo Transfer Mode

TL7 now supports a default "**Turbo Transfer**" mode, which uses binary FTP protocol to upload the code image to the PLC instead of via ASCII command strings. Turbo Transfer can greatly reduce the program transfer time for very long PC7 program. This can greatly enhance programmer's productivity during program development when small changes to the program happen very frequently.



Success !

Total Number of circuits	15
Ladder Diagram (words)	48
Custom Function (words)	1574
Total code Size	1622

Checksum = 0001966E

DIFU+DIFD+ One-shot contact = 2 (max=256)

☑ Use Turbo Transfer Mode (If available)

However, in order for Turbo Transfer mode to work, it requires that the PLC's FTP server to be accessible in active mode from i-TRiLOGI. Normally this is not an issue when the PC and the PLC are on the same local area network. However, in the case where the PLC is sitting behind a NAT firewall where only port 9080 is mapped, a user using i-TRiLOGI to remotely program the PLC via the Internet will notice that the PC running i-TRiLOGI may not be able to access the FTP server port (21) on the PLC, and therefore the Turbo Transfer mode will fail.

Fortunately, when Turbo Transfer fails, I-TRiLOGI will automatically switch back to regular transfer mode using ASCII string to transfer the codes one block at a time via port 9080. The user can also selectively disable Turbo Transfer mode to reduce the time wasted in trying for Turbo Transfer mode that could not be performed. Turbo transfer de-selection is volatile. When TL7 is re-started Turbo Transfer Mode will be selected by default.

1.6 Backup Zip file

New in TL7: Every time you save the current PC7 program to the hard disk, TL7 will at the same time create a zip archive to store the current program as well as past 4 copies of your PC7 program as "Backup001.PC7" to "Backup004.PC7". This zip serves an additional safe guard for your program in case of unanticipated circumstances you can try to retrieve the backup copy from this zip archive.

We strong recommend that you regularly copy the zip file to a backup flash drive or email it to yourself for safe keeping.

Chapter 2 TL7 Floating-point Math

2 TL7 Floating-point Math

2.1 Floating-point Number Representation

TL7 uses IEEE, 32-bit Single Precision format to represent all float numbers. For more information on how single precision float are encoded into 32-bit of data, please refer to the following link:

http://en.wikipedia.org/wiki/Single-precision_floating-point_format

However, in-depth knowledge of how float are actually encoded are not required for TL7 users since TL7 automatically handles the floating-point data conversion to and from its binary representation. All you need to know are the range of numbers that an IEEE single precision floating-point number can represent, as follow:

Minimum positive value	1.18×10^{-38}
Maximum positive value	3.4×10^{38}.
Minimum negative value	-3.4×10^{38}
Maximum negative value	-1.18×10^{-38}
±Zero	Yes
± Infinity	Yes
NaN (not a number)	Yes

Special formats are used to represent special numbers such as zeros (there are positive zero and negative zeros but they compare as equal), infinity and NaN. These are supported by TL7.

Note: Having the basic knowledge of how the float data are actually encoded in 32-bit binary number is useful when you need to transport the numbers to and from external devices via serial or Ethernet communication. TL7 supports special function "**`Float2Bits(float)`**" to convert a float into its 32-bit IEEE format and the function "**`Bits2Float(integer)`**" to decode a 32-bit IEEE representation (of a single precision float) back into a floating-point number. This makes it very simple to send and retrieve float data via Modbus and Hostlink commands that traditionally support only 16-bit and 32-bit integer data.

2.2 Floating-point Variables

TL7 uses IEEE, 32-bit Single Precision format to represent all float numbers. For more information on how single precision float are encoded into 32-bit of data, please refer to the following link:

TL7 added the following new Floating-point variables

Floating-point variables	Type
`A# to Z#`	Global variable
`FP[1] to FP[1000]`	Global variable
`%[1] to %[9]`	Local variables*

* Support of "Local variables" is a completely new addition to TL7 that allows up to 9 floating-point parameters to be passed to and used within a custom function. Please refer to Section 1.3 in this manual for more detailed explanation of local variables.

2.3 Floating-point Constant

Float constants can be entered into TBASIC using the popular floating-point format as shown in the following examples:

Number	Format 1	Format 2	Format 3	Format 4
123.4567	123.4567	1.234567E2	1.234567e+02	+1.234567e+02
-0.0004567	-0.004567	-4.567E-4	-4.567e-4	-4.56700e-04
12345	12345.0	1.2345E4	+1.2345e04	1.2345e+04
0	+0.0	-0.0		

It is important to note that if you want to use a whole number as a floating-point number in an expression, you should append it with a decimal point and a 0. i.e. 123 should be entered as 123.0. The reason being that certain operators such as "/" (divide by) operate differently on floating-point numbers than that on integers.

```
E.g.   A# =  11.0/2.0  =  5.5
       A# = 11/2 = 5
```

In the second expression, 11/2 is treated as an integer divide between two integers 11 and 2 and the result is an integer 5 without any fractional part. When the result is finally assigned to A# it has already lost its fractional part. But as long as one of the two operands is a float (such as 11.0 or 2.0) the divide operator will perform floating-point division and return a float.

Likewise,
```
A# = 345/0  => Run time error: Divide by Zero
A# = 345/0.0 = +Infinity
```

Special Floating-point Constant

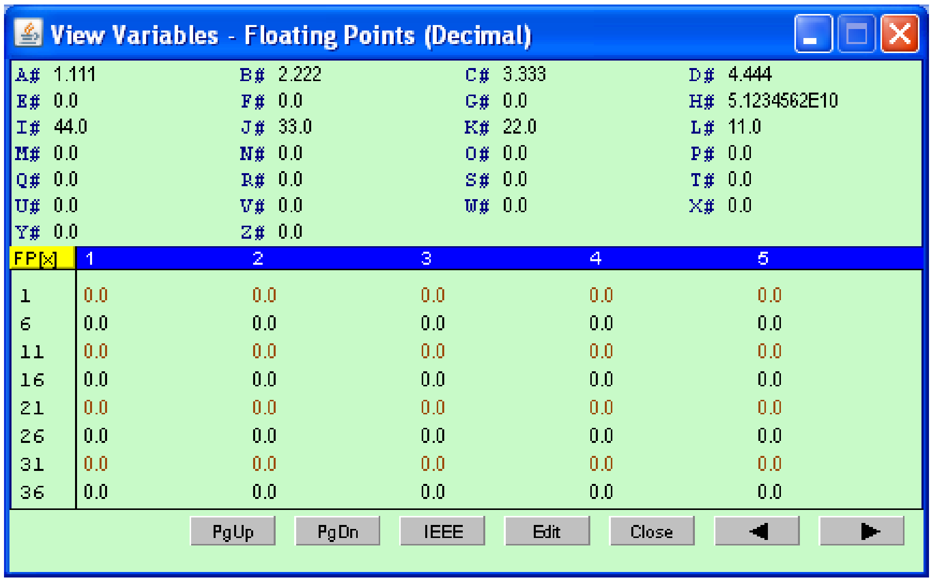
Due to their relatively rare use in control system, TL7 does not create special keywords to represent special float numbers such as +infinity. If you really need to use such special numbers in your program you can use the Bits2Float( ) function to convert their 32-bit integer representation into these special numbers, as follow:

+ Infinity	E.g.	`A# = Bits2Float(&H7F800000)`
- Infinity	E.g.	`B# = Bits2Float(&HFF800000)`
NaN	E.g.	`C# = Bist2Float(&H7FC00000)`

TL7 however thus support the keyword **NaN** (not a number) since this may be used to return to a caller to alert the caller that it has supplied invalid parameters being supplied by the caller.

2.4 Viewing of Floating-point Variables

TL7 added an additional screen to the "View Variable" screen of the simulator/online monitoring screen to display all the new floating-point variables A# to Z# as well as the floating-point array FP[1] to FP[1000] as shown below:



A# 1.111	B# 2.222	C# 3.333	D# 4.444
E# 0.0	F# 0.0	G# 0.0	H# 5.1234562E10
I# 44.0	J# 33.0	K# 22.0	L# 11.0
M# 0.0	N# 0.0	O# 0.0	P# 0.0
Q# 0.0	R# 0.0	S# 0.0	T# 0.0
U# 0.0	V# 0.0	W# 0.0	X# 0.0
Y# 0.0	Z# 0.0		

FP[x]	1	2	3	4	5
1	0.0	0.0	0.0	0.0	0.0
6	0.0	0.0	0.0	0.0	0.0
11	0.0	0.0	0.0	0.0	0.0
16	0.0	0.0	0.0	0.0	0.0
21	0.0	0.0	0.0	0.0	0.0
26	0.0	0.0	0.0	0.0	0.0
31	0.0	0.0	0.0	0.0	0.0
36	0.0	0.0	0.0	0.0	0.0

You can click on the "PgUp" or "PgDn" button on the screen or use the "PgUp" and "PgDn" keys on your keyboard to scroll through all the pages to view all 1000 FP[ ] variables.

The third button: "IEEE" - let you switch the view of the values of these floating-point variables between human readable decimal format and their actual IEEE 32-

bit single precision format. In IEEE mode these values are displayed as 8 characters of hexadecimal digits.

If you are interested in seeing how the floating-point values are represented in IEEE format you can use the following online calculator to verify the data you observe on the screen:

http://www.h-schmidt.net/FloatConverter/

2.5 Floating-point Operators

"Operators" perform mathematical or logical operations on data. TBASIC supports the following integer operators:

i) Assignment Operator (=):

A floating-point variable (FP[], A# to Z#, %[1] to %[9]) may be assigned a numeric value or the result of a numeric expression using the assignment operator " = "

```
A# = 1000
X = H*I+J + len(A$)/FP[5]
```

ii) Arithmetic Operators:

Symbol	Operation	Example
+	Addition	`A# = B+C+25`
-	Subtraction	`Z = TIME[3]-10`
*	Multiplication	`PRINT #1 X*Y`
/	Division	`X = A/(100+B)`

A numeric expression using the above arithmetic operator can include both integers and floating-point numbers. As long as one of the two numbers is a float the integer number will be converted to float and then the floating-point operator will be used and the result is a float. However if both operands of an arithmetic operator are integers then the integer operator will be used and an integer number will be returned.

iv) Relational Operators :

Used exclusively for decision making expression in statement such as **IF** *expression* **THEN** ..... and **WHILE** *expression* ....

Symbol	Operation	Example
=	Equal To	IF A = 100
<>	Not Equal To	WHILE CTR_PV[0]<> 0
>	Greater Than	IF B > C/(D+10)
<	Less Than	IF TIME[3] < 59
>=	Greater Than or Equal To	WHILE X >= 10
<=	Less Than or Equal To	IF DM[I] <= 5678
AND	Relational AND	IF A# >B# **AND** C<=D
OR	Relational OR	IF A#<>0.0 **OR** B# >=1000

The programmer should bear in mind that a decimal floating-point number in a computer is at best an approximation of the real data and not an exact number. Any arithmetic operation carried out on a number could result in rounding or truncation errors of the resulting. So comparison of two numbers using the "Equal" operator may not always work as expected. E.g.

```
A# = 0.0
FOR I = 1 to 1000
     A# = A# + 0.1
NEXT

IF A# = 100.0 THEN
    SETLCD 1,1, "True"
ELSE
    SETLCD 1,1, "False"
ENDIF
```

At first glance you may expect that the result should be "True" since adding 0.1 to A# 100 times should means A# = 100.000 after the program is run? However, when you execute this program you may be surprised that the result is "False". Further examination of the execution result you can see that after adding 0.1 to A# 100 times, A# actually contain the value "99.999", which is quite close to 100.0 but not exactly. The error is due to cumulative truncation errors resulting from floating-point addition. Hence you shoudl avoid using the "Equal" operator to compare two floating-point numbers directly.

2.6 Built-in Floating-point Functions

There are a number of built-in floating-point functions such as trigonometric functions (**`SIN, COS, TAN`**) , Logarithmic functions (**`Ln`** and **`Log`**$_{10}$), power and square root function as well as exponential (e^x) functions. The syntax of these functions are described in details in the next Chapter.

Chapter 3 Floating-point Functions

3 Floating-point Functions

3.1 Overview

A number of new built-in functions that support floating-point math have been added to TL7. Some functions that previously only work with integer math have also been modified to support floating-point operations.

This chapter will list the syntax of these new functions as well as functions that have been modified to accommodate floating-point data. However, functions that simply convert floating-point operand to integer and work on it will not be repeated here. E.g. function such as STATUS(n) - if you supply n as a floating-point data it will simply be converted to integer for use by the STATUS(n) function and there is no difference in the outcome. Please refer to the TBASIC keywords in TL6 Programmer's Reference manual for other functions and commands.

3.2 ARCSIN(*x#*), ARCCOS(*x#*) and ARCTAN(*x#*)

Purpose	:	To compute and return the inverse-sine, cosine or tangent of the operand *x#* in radians. This function will return a NaN if the operand *x#* is invalid.
Examples	:	**`A# = ARCSIN(0.5)`**
Comments	:	A# will receive the value 0.5235987 (unit is radians). To convert to degree multiply by 180/π = 30 degree.
See Also	:	SIN(*x#*), COS(*x#*), TAN(*x#*)

3.3 BITS2FLOAT(*n*)

Purpose	:	*n* should be a 32-bit integer representing the floating-point variable in IEEE single precision format and this function returns the actual float value that it represents.
Examples	:	**`A# = BITS2FLOAT(&H3F9E0610)`**
Comments	:	A# will receive the value 1.23456
See Also	:	FLOAT2BITS(*x#*)

3.4 CEILING(*x#*)

Purpose	:	Returns the smallest integer that is not less than *x#*.

Examples	:	`A# = CEILING (3.456)`
		`B# = CEILING (-3.456)`

Comments	:	A# will be assigned the value 4.0 and B# will be assigned the value –3.0
See Also	:	FLOOR(*x#*), ROUND(x#)

3.5 COS(*x#*)

Purpose	:	Returns the COSINE of the angle *x#* *x#* should be in radian. If your angle in degree you must convert it to radian first. Radian = degree /180 * π
Examples	:	`A# = COS(0.5)`
Comments	:	A# will be assigned the value 0.8776
See Also	:	ARCCOS(*x#*), SIN(*x#*), TAN(*x#*),

3.6 EXP(*x*)

Purpose	:	Returns the exponential e^x e is the mathematical constant (Euler's number) which is approximately = 2.71828,
Examples	:	`A# = EXP(1.5)`
Comments	:	A# will be assigned the value 4.48169
See Also	:	LOG(*x#*) or LN(*x#*)

3.7 FLOAT2BITS (*x#*)

Purpose	:	Return the 32-bit integer representing the IEEE single precision format that of the floating-point number *x#*
Examples	:	`A = FLOAT2BITS (3.1416)`
Comments	:	A will receive the value = &H40490FF9 (1078530041 in decimal)
See Also	:	BITSFLOAT(*n*)

3.8 FLOOR(*x#*)

Purpose	:	Returns the largest integer that is not greater than *x#*.

Examples	:	`A# = FLOOR(3.456)`
		`B# = FLOOR (-3.456)`

Comments	:	A# will be assigned the value 3.0 and B# will be assigned the value –4.0
See Also	:	CEILING(*n#*), ROUND(n#)

3.9 FUNCTION (*fn#, p1, p2,….*)

Purpose	:	One way of calling a custom function using its function # (or label name) and supplying parameters p1, p2 separated by commas.
Examples	:	`A# = FUNCTION (10, B#, FP[5], 0.1245)`
Comments	:	Custom function #10 will be called and the values in B#, FP[5] and the constant 0.1245 will be passed to function #10 as local variables %[1] to %[3]. A# will be assigned the value returned by function #10 via the RETURN = [return value] statement.
See Also	:	Section 1.3 in this manual

3.10 LOAD_EEP# (*n*)

Purpose	:	Return the floating-point data loaded from EEPROM/RAM at address *n*
Examples	:	`A# = LOAD_EEP#(10)`
Comments	:	Note that LOAD_EEP# and SAVE_EEP# uses the identical EEPROM space as LOAD_EEP32 and SAVE_EEP32. For more details please refer to LOAD_EEPXXX command in the TL6 Programmer's Reference manual. Thus, if you are using the EEPROM space to save both integer and float data, you must manage the memory space properly to ensure that this command does not overwrite the memory space reserved for saving 8, 16 or 32-bit integers.
See Also	:	SAVE_EEP# , SAVE_EEP32, LOAD_EEP32

3.11 Log (*x#*)
LN(*x#*)

Purpose	:	Returns the natural logarithm of *x#*. LOG(x) and LN(x) are identical command.
Examples	:	`A# = LOG(120.40)`
Comments	:	A# will be assigned the value 4.7908196
See Also	:	EXP(*x*)

3.12 Log10 (*x#*)

Purpose	:	Returns the base-10 Logarithm (a.k.a. Common Log) of *x#*.
Examples	:	`A# = LOG10(120.40)`
Comments	:	A# will be assigned the value 2.0806265
See Also	:	POWER(*x*,*y*)

3.13 PIDCompute(ch, E) PIDdef ch, lmt, P, I, D

Purpose : These two functions work exactly the same in TL6, except that now E, P, I and D take only floating-point data. If you supply integer data for E, P, I or D the software will convert them into float and used for computation.

For detailed explanation of how these two commands are used please refer to TL6 Reference manual.

3.14 POWER (*x*, *y*)

Purpose : Returns the value of x raised to the power of y. i.e. x^y

Examples :
```
A# = POWER(10.75, 3.54)
```

Comments : A# will be assigned the value 4479.0503

See Also : LOG10 (*n#*)

3.15 ROUND(*x#*)

Purpose : Round the *x#* to the nearest integer and return the integer value.

Examples :
```
A# = ROUND(120.49)
B# = ROUND(-120.49)
C# = ROUND(250.50)
D# = ROUND(-250.50)
```

Comments : Result A# = 120 , B# = -120, C# = 251, D# = -251.

See Also : CEILING, FLOOR

3.16 RND(*x#*)

Purpose : Get a random number n# such that 0.0 < n# < x#

Examples :
```
A# = RND(50)
```

Comments : A# will be between 0.0 and 50.0

Note : This command only available from i-TRiLOGI Version 7.01 and firmware >= F90.2.

3.17 SAVE_EEP# *x#, n*

Purpose : Save the 32-bit IEEE single precision number that represent the floating-point data *x#* to the EEPROM/FRAM address *n*.

Examples : `SAVE_EEP# FP[20], 100`

Comments : Float data contained in variable FP[20] will be saved to the EEPROM address 100.

Note that LOAD_EEP# and SAVE_EEP# uses the identical EEPROM space as LOAD_EEP32 and SAVE_EEP32. For more details please refer to LOAD_EEPXXX command in the TL6 Programmer's Reference manual.

Thus, if you are using the EEPROM space to store both integer and float data, you must manage the memory space properly to ensure that this command does not overwrite the memory space reserved for saving 8, 16 or 32-bit integers.

See Also : LOAD_EEP# , SAVE_EEP32, LOAD_EEP32

3.18 SIN(*x#*)

Purpose : Returns the SINE of the angle *x#*

x# should be in radian. If your angle in degree you must convert it to radian first. Radian = degree /180 * π

Examples : `A# = SIN(0.5)`

Comments : A# will be assigned the value of 0.47942555

See Also : ARCSIN(*x#*), COS(*x#*), TAN(*x#*)

3.19 SQRT(*x#*)

Purpose : Returns the Square Root of the float number *x#*

x# must be >= 0. Otherwise the function will return a NaN.

Examples : `A# = SQRT(50.0)`

Comments : A# will be assigned the value of 7.071068

See Also : POWER (*x, y*)

3.20 STR$ (*x#*)

Purpose : Convert the floating-point variable ***x#*** into its string representation.

This function shares the same function name as the STR$(*n*) found in TL6, When n is an integer STR$(*n*) will return the string representing the integer as described in TL6 programmer's Reference to maintain backward compatibility.

However, when the operand is a floating-point data, STR$(*x#*) will return the string representing the data as either "regular" floating format such as xxx.yyyyy or in scientific notation such as ± x.xxxxxxxE±nn, depending on the range of values x# falls into. The following criteria can be used to predict the format of the string that STR$(x#) will return:

Value range of *x#*	Format	Example

0.0001 < |*x#*| < 1.0	**`±0.ffffffff`**
	1 decimal digit and 7 fractional digits.
		`+0.0012345`
		`-0.4560000`
		`+0.0920000`

1.0 ≤ |*x#*| < 1.0 x10^7	**`±ddd.fffff`**
	n decimal digit and (8-n) fractional digits
		`+123.45600`
		`+90014.016`
		`-55.678000`

|*x#*| ≥ 1.0 x10^7
|*x#*| ≤ 0.0001
	`±d.fffffffE±nn`
	1 decimal digit and 7 fractional digits. 2 exponent digits
		`+2.5670000E+08`
		`-1.3456000E-04`
		`+7.3300000E-09`

Notice that in all formats, number of decimal digit+number of fractional digit = 8. In particular, the scientific notation format only uses the following format:

± x.xxxxxxxE±nn

i.e. the returned string will always start with a sign, then a single decimal digit followed by 7 fractional digits, an E and then the sign of the exponent, followed by the two digit exponents. The length of the returned string is always14 characters when it returns in scientific notation.

Eg. +1.2345000E+10, -9.8763475E-05

This function will return variable significant digit based on the value it evaluated. For fixed number of significant digit, please refer to the next command STR$(***x#*, *w***)

Example : A$ = STR$(0.000012345) => A$ = +1.2344999E-05

Comment : A$ did not get the string "+1.2345000E-05" as expected because this number cannot be precisely represented by the IEEE single precision format and therefore it is approximated by its closest number +1.2344999E-05.

See Also : STR$(*x#, w*)

3.21 STR$ (*x#, w*)

Purpose : Convert the floating-point variable ***x#*** into a string representation with ***w*** number of characters, where either 8 ≤ ***w*** ≤ 14 or -6 ≤ ***w*** ≤ 0

This function shares the same function name as the STR$(n, d) found in TL6, where n is the integer value and d is the number of characters. When n is an **integer** STR$(n, d) will return the string exactly as described in TL6 programmer's Reference to maintain backward compatibility.

However, when the first operand is a float, the function STR$(***x#, w***) will only return the string according to the following rules:

W	**String Format**
a) 0 < ***w*** ≤ 8 (minimum positive)	± **`d.fE±nn`** E.g. 3.4E+05
b) ***w*** >= 14 (maximum positive)	± **`d.fffffffE±nn`** E.g. +1.2345678E-05
c) ***w*** = 0	Rounded to the nearest integer.
d) –6 < ***w*** < 0	| ***w*** | represents the number of decimal places in the returned string.

a) When ***w*** is zero, STR$(x#, 0) returns a string representing an integer equal to rounding of x# to the nearest integer.

b) When ***w*** < 0, the function will **never** return a string in scientific notation, but will **always** return x# as a regular expression nnn.ffff where |***w***| represents the number of decimal places.

```
E.g. STR$(0.0123456, -4) => 0.0123 (4 decimal places)
```

c) When ***w*** is a **positive** number, STR$(x#, w) function will only return a string representation of x# in **scientific notation** in the following format:

± d.fffffffE±nn

The shortest format that this function returns is: **±d.fE±nn** which is is 8 characters wide.

The longest format that this function returns is **±d.fffffffE±nn** which is 14 characters wide

Example : A$ = STR$(0.000012346, 10)
B$ = STR$(-0.000012346,8)
C$ = STR$(-0.000012346,16)
D$ = STR$(-0.000012346,12)

Result : A$ = “+1.235E-05”
B$ = “-1.2E-05”
C$ = “-1.2346000E-05”
D$ = “-1.23460E-05”

Comments : If w is < 8 but > 0, the function will return the shortest form which is 8 characters wide. If w > 14 the function will return the longest form which is 14 characters wide.

The data will be rounded up to the precision level necessary to represent the data in w number of characters.

3.22 TAN(*x#*)

Purpose : Returns the TANGENT of the angle *x#*
x# should be in radian. If your angle in degree you must convert it to radian first. Radian = degree /180 * π

Examples : **A# = TAN(0.5)**

Comments : A# will be assigned the value of 0.5463025

See Also : ARCTAN(*x#*), SIN(*x#*), COS(*x#*)

** THE END **

For detailed information about how to use the i-TRiLOGI software including installation procedure and TLServer functionality, kindly refer to the TL6 Reference Manual, which can be downloaded from:

http://www.triplc.com/documentations.htm