TRiLOGI

Version 4.1

Programmer's Reference

Part Two

The TBASIC Reference

Programming Language Support for MODBUS and EMIT3.0 features of M+ series PLC.

Table of Contents

Chapter 5 - Application Programming Examples

Chapter 1 - Upgrading From TRiLOGI Version 3.x to 4.1

TRiLOGI Version 4.1 is the editor, compiler and simulator software for programming the new **M-series** family of PLCs from Triangle Research International Pte Ltd (TRi). Version 4.1 is specifically created for the M and M+ PLC models only and hence it cannot be used to program the H- and the E-series. (Use Version 3.x for the H & E-series PLCs, Please refers to the Programmer's Reference manual for TRiLOGI Version 3.x)

TRiLOGI Version 4.1 expands the ladder logic language of Version 3.x (which is the standard editor for programming the H-series PLCs) and adds a whole new suite of TBASIC commands for handling complex computational tasks that are either impossible or awkward to write using traditional ladder logic language.

To seamlessly integrate TBASIC commands with the ladder diagram, we invented the concept of a "Custom-Function" (abbreviated as CusFn) which can be connected as a special function coil to the end of a ladder rung. TBASIC commands are then entered via a built-in full screen text editor that enables you to define what you want that particular CusFn to do.

Up to 256 CusFns can be defined. A CusFn may be connected as a normal coil [CusFn] to the ladder rung, or as a "Differentiation Up" coil [δCusF] which means that <u>it will only be executed once</u> when its ladder logic condition goes from OFF to ON. A CusFn can also be called by any other CusFns and act like a subroutine.

The next three chapters will describe in detail how you can create and define the CusFn and various TBASIC commands. In this chapter, however, we will focus on the modifications made to the Ladder Logic portion of the software found in TRiLOGI Version 3.x

1. File menu.

a) In TRiLOGI Version 4.1, an "Import File" function has been added to allow other TRiLOGI files to be imported to the current program. You can now import the ladder logic, I/O definitions and Custom functions defined in another TRiLOGI program into the current program. New ladder rungs will be appended to the end of current ladder program. However, please note that If the imported file and the current file contain I/O labels for I/Os of the same location or same Custom Function number, the imported file will take precedence and overwrite the I/O label or CusFn definition. Other I/O or CusFn not used in the imported file will not be affected at all.

You can use this feature to create a library of useful custom functions or ladder programs to be merged to your actual application when required.

b) You can run a small DOS utility program without quitting TRiLOGI by executing the "Run Utilities" command under the "File" menu. All files within your current directory and with the ".EXE" extension will appear in a pop up menu. You can move the highlight bar to the desired file and press <Enter> key to run it.

2. Edit Menu

a) Three new commands: "Edit Custom Function", "Browse All CusFn" and "Search Text in CusF ^F" have been added to the "Edit" pull-down menu which will activate the built-in text editor to enable you to enter or display the TBASIC commands for the selected CusFn. Please refer to Chapter 2 for details on how to create/edit a CusFn.

"Browse All CusFn" or <Ctrl-F7> opens up the text editor in "Read-Only" mode, which means that you may not change the content of the CusFn. This command only opens up CusFns that have been defined but skips those undefined functions. You can use the Up/Down cursor keys to scroll from one CusFn to another quickly, which offers you a quick way to examine all the CusFns that have been defined in your program. You can also activate this command during "Simulation" by pressing the <Ctrl-F7> keys.

This "Search Text in CusF ^F" command allows you to search for a text string across all CusFn. You will be prompted for the text string to find and when found, the corresponding custom function will be opened with the cursor positioned at the location of the last found text. Press <Enter> key to search for next occurrence of the text. You can also use up/down cursor keys to search in the previous or the next custom function. You may also press <Ctrl-F> hot key to use it.

b) TRiLOGI Version 4.1 supports High Speed Timers. Any one or all of the 128 timers can be configured as high speed timers using the TBASIC command "HSTIMER". When a timer is activated as a high speed timer, its set value (SV) in the "Timer" table represents how many 0.01s the timer takes to completely count down. These give rise to a timer range of 0.01 to 99.99 seconds. If it is not defined as high speed, then the set value represents 0.1 to 999.9 seconds as in Version 3.x.

c) In Version 4.1 all the inputs, outputs, relays, timers, counters and their respective present values are accessible as 16-bit integer variables such as INPUT[1], TIMERBIT[3], etc. We modify the CH:Bit column in the I/O tables so that the first number CH represents the index of the variable (1,2,...) and second number ":Bit" represents the bit position (in hex digit 0,1....14,15) that I/O occupies within the 16-bit number.

3. Controller Menu

The following changes have been made to the "Controller" pull-down menu:

a) Modem Connect/Disconnect

TRiLOGI Version 4.1 now supports modem dial-up to connect to a T100MD+ or T100MX+ PLC located at remote location via telephone lines. Please see the "Communication Setup" in the next subsection to select or define your modem type. You use this command to instruct your PC modem to dial the telephone number where the remote modem and PLC are located. You will be prompted to enter the telephone number to dial or accept the last dialed number which is automatically saved in the configuration file when you quit TRiLOGI.

Once the connection has been established, you can perform on-line monitoring/control or transfer program to the PLC just like when you connect the PC to the PLC by hardwired cable. However, please note that there will always be some time delay when the PC sends out a host link command string until it receive a response string from the PLC. This is due to the time it takes the modem to convert (modulate and demodulate) the serial data to and from voice signals for transmission along the public telephone system.

Hence when TRiLOGI is connected to the PLC via modem the overall system will feel more "sluggish" when compared to direct cable connection. This is quite normal and you should not be alarmed.

Password Protection

Notice that anybody with a TRiLOGI 4.1 software and a modem could dial into your PLC+modem at a remote location and make unauthorized modification to your program or remotely activate an I/O. Therefore this will become a very serious security issue that you must be concerned with. You should use the SETPASSWORD command in TBASIC to define a password for the PLC program. Please refer to Chapter 4 for detailed description of the "SETPASSWORD" command.

Disconnecting Modem

Once you have finished your work, you should disconnect the modem from the remote link by using the "Disconnect" command in this menu.

This command instructs the modem to hang up the telephone line. It will also re-arm the password protection in the PLC so that the next caller cannot have unauthorized access to the PLC.

b) Communication Setup

An option for setting up the modem has been added in this command. You can choose from a menu of some pre-defined standard modems (Hayes, Motorola, USRobotics, etc.) or define a modem initialization string for your own user-defined modem type.

Note that reliable connection via modem can only be established at transmission baud rate of 9600bps or slower. This is because the M-series PLCs do not utilize the RTS/CTS handshaking lines of the modem and hence it is not possible to communicate via modem at faster communication rate than 9600bps. Therefore, to allow remote monitoring and control via modem, you must execute the TBASIC function: "SETBAUD 1,3" once during the PLC initialization process to set the Comm port #1 to 9600bps.

Modem Setup

TRiLOGI automatically sets the serial port defined for modem communication to 9600bps. Almost all modems on the market today utilize the so called "AT command" set which was pioneered by the company Hayes Inc. These involve sending ASCII command strings to the modem beginning with the characters "AT" followed by a string of ASCII characters. There are many commands published in the modem's manual or the manufacturer's website that will allow you to configure the modem to various operating modes. Specifically a modem must be defined to the following operating modes to work properly with TL41.EXE and the M-series PLCs:

Modem Operating Mode	AT Commands		
	Hayes Accura	US Robotics	Motorola lifestyle
a) No local echo	E	E	E
b) Disable flow control	&K	&H	\Q \G
c) Constant speed	&Q6	&B1	
d) Forced to communicate only at 9600bps.	N0S37=9	&N6	%L5 %B5
e) Return result codes	Q	Q	Q
f) No Data compression		&K	%C

The table above shows the corresponding AT commands for three popular brands of PC modems. If you are defining your own modem, you need to check your modem reference manual or the manufacturer support website to find out the required ASCII strings to set the modem into the operating modes as defined in the table. All the AT commands can be combined together into a single string beginning with the characters AT. For example, the required initialization string for a Hayes Accura modem should be:

ATE&K&Q6N0S37=9

Please note that a “Hayes compatible” modem really only means it uses the “AT” commands and does not mean that it is 100% compatible with the Haye’s Accura modem’s command set. In fact, US Robotics and Motorola modems mentioned above are all “Hayes compatible” modems but they each require very different initialization string from the others.

For modems which are pre-defined on the Modem Setup menu you need not be bothered with the modem initialization string as TRiLOGI Version 4.1 will take care of sending the correct string according to the selected modem. However, if your brand of modem is different from what is available on the menu then you need to select the user-defined modem type. When you select the User-defined modem you will be prompted to enter the modem initialization strings. The string you entered will be saved in TRiLOGI configuration file.

Modem COM Port Selection

After you have defined the modem string, You will also be prompted to select the COM port that your modem is connected to. TRiLOGI 4.1 allows your computer to use both COM ports 1 and 2 for connection to the PLC. You may use one COM port for direct cable connection and the other for connection via a modem. The COM port defined in the “modem setup” is independent of that defined for direct cable connection.

c) New command: Set PLC’s Clock/Calendar

As mentioned in the last section, the Real-Time-Clock of the PLC can be set by executing this command. TRiLOGI will retrieve the current date and time setting from the PLC and display a data entry form for the Hour, Minute, Second, Year, Month and Day. You can edit the data using the <Del> and <Backspace> keys to erase the original entry and enter the new data. Items requiring no changes can be skipped by pressing the <Enter> key.

Once all the data has been entered, the “RTC.Err” flag in the PLC will be cleared immediately.

d) New commands: 1: Host Timer/Ctr SV --> PLC
2: PLC’s Tim/Ctr SV --> Host

In TRiLOGI Version 4.1 you can modify the Set Value (SV) of any timer and counter on their respective tables (by pressing the <F5> and the <F6> keys) and update their respective values stored within the PLC’s EEPROM without transferring the entire program. This is much faster and opens the possibility of modifying the Set Values of timers and counters without the need for the original source program.

The command “1: Host Timer/Ctr SV --> PLC” transfers the Set Values stored in the Timer/Counter tables of the currently loaded ladder program file into the PLC. Alternatively, you can also capture the SVs of the timers/counters within the PLC and update them on the host ladder program’s Timer/Counter tables. Simply execute the command: “2: PLC’s Tim/Ctr SV --> Host” to do the trick. Note that to update the SVs of a timer or a counter, you must give it a unique label name.

e) Program Access Password protection scheme

In Version 4.1, once you set the program access password the protection will be in place all the time, regardless of how many times you have subsequently transferred the program. In addition, Once you have entered a password, you will be prompted to enter it every time you try to transfer a program to it. This protects the PLC program so that unauthorized person cannot alter the program without your knowledge.

The password can be deleted using a newly added command in the “Target PLC Access” sub-menu named: “Delete Password & Clear Program”. Note that the entire program will be erased when the password is deleted and you need to re-transfer the working software. This provides adequate protection against unauthorized deletion of password by others.

Note: The password mentioned here refer to that defined by the “Target Access -> Set Password” command which is independent of SETPASSWORD command in TBASIC. The password here only affects program downloading/uploading, but does not inhibit normal host communication. The SETPASSWORD statement in TBASIC however will lock out the PLC from any host link commands until a correct password command has been received.

4. Simulate menu.

In TRiLOGI Version 4.1, the user-program memories are expressed in terms of 16-bit “words” instead of “steps”. Every ladder logic element, including timer, counter and special function coil occupy exactly one word. Every TBASIC command occupies from 1 to multiple words, depending on the number of operands and the type of operands (16-bit or 32-bit) involved. The number of words occupied by the ladder logic and the CusFns are reported separately when you execute the “Only Compile” command.

If a CusFn has been defined but is <u>not used</u> in the ladder diagram or called by other CusFn at all, then the compiler <u>will not compile</u> it and it will not report its program size to the user either.

<u>Simulation of Momentary Inputs (e.g., PUSH BUTTON Switch)</u>

A new improvement in the Simulation process is the ability to simulate momentary input by pressing <Ctrl-Enter> when the highlight bar is at the “Input” column of the Simulation Screen. The corresponding input will change state (OFF -> ON or ON->OFF) momentarily and it will then revert back to its original logic state. The ladder program will execute for 1 ladder logic scan during the change of state. This allows easy simulation of “Push Button” type inputs which only be activated when you press the button and de-activated when you release your finger.

<u>New Command: Silent Operation</u>

Some TBASIC instructions such as PRINT, INPUT$, ADC, etc always pop up a message window during simulation. These messages may appear repeatedly and interfere with normal simulation. If you wish to suppress the appearance of these windows, simply execute the “Silent Operation” command and select either to “Suppress All Messages” or “Allow Only ADC inputs”. This setting will not be saved in the configuration file.

The PRINT #, INPUT$ and ADC prompt windows can also be selectively turned OFF during simulation session by pressing the <Ctrl-H> keys when the window is opened. To re-enable these two pop-up windows, press <Ctrl-H> keys again in Simulation screen. The screen will appear when the corresponding commands are next called in the program.

5. Print Menu.

a) A new command “Custom Function” has been added to allow selective printing of CusFn #1 to 256. CusFns that have not yet been defined will not be printed.

b) When you first select to print any item, you will be prompted to select the destination of output. The default is “DOS File” and is useful because the created text file can be read into any word-processor for formatting before final print-out. If you are using a Windows-based word processor, you have to select all the content of the file and change their font to: “MS Linedraw” for proper display of the graphic characters.

   If you accept the print destination as “DOS File”, you will be prompted to enter the name of the text file you wish to save as a print file. This is an improvement over Version 3.x which saves only to a system pre-defined file name.

c) The rarely used print command: “CH:Bit logic” has been deleted from the Print menu.

d) Laser printers which are compatible to HP LaserJet (use HP-PCL printer commands) are directly supported under the “Printer Setup” menu for quick configuration of the starting and ending codes.

6. Special Bits Menu

Two additional clock pulses with periods of 0.05s and 0.5 seconds are added to the “Special Bits” menu, providing more choices and convenience to programmers who need them.

RTC.Err flag - Every M-series PLC incorporates a “Real Timer Clock” (RTC) which keeps track of the Year, Month, Day, Hour, Minute and Second of our “REAL” world (hence the name: “REAL TIME”). However, unless the PLC is equipped with a battery-backed RTC module, the clock will cease to operate when the PLC is switched off. When the PLC is turned on again the CPU will reset the RTC to some factory-preset date and time.

To inform users that the RTC has been reset due to power failure, a new flag: “Real Time Clock Error” has been added to the “Special Bits” menu to alert users that the real-time clock has been reset. You can use this flag to turn on a warning light or an alarm so that the operator can set the clock to the proper time. On some models of the M-series PLC a status LED will also light up when RTC error occurs. This is important if your program utilizes the RTC to turn ON or turn OFF a device at some pre-scheduled times (similar to the preset timers commonly found in a VCR), in which case remedy action must be taken if the clock has been upset due to power failure. You can also use this flag to be the pre-requisite condition for executing preset schedule timer functions. If you don’t use the RTC, then simply ignore this flag.

The RTC can be set by the command: "Set PLC's Clock/Calendar" under the "Controller" menu described in the next section. Once you have set the date and time of the RTC, the "RTC.Err" flag will be turned OFF immediately.

RTC Module: MX-RTC

If the PLC is installed with the optional battery-backed RTC module: "MX-RTC", then the CPU will sense it automatically when powered up and the RTC.Err flag will not be set.

7. Ladder Logic Comments

The "Put Comments" feature of Version 3.x has been improved to allow you to enter up to 4 lines per comment circuit. You can also merge or break comment lines within the comment circuit using the <backspace> and <Enter> keys.

8. Changing the Set Value (S.V.) of Timers and Counters

In TRiLOGI Version 3.x, the S.V. of internal timers and counters are pre-defined in their respective definition tables and cannot be changed by the program during execution.

In Version 4.1 this limitation has been lifted by the introduction of two TBASIC commands: SetTimerSV and SetCtrSV. During execution of the user program the set value of any timer or counter can be changed any time by these commands. If you simulate the program which changes the S.V., the change will be reflected in their respective Timer/Counter definition tables.

During full-screen simulation or On-line monitoring, if you move the highlight bar to the "Timer" or "Counter" window and press <Enter>, an "Edit Present Value" window will be opened as in Version 3.x. However, *you can also see the defined S.V. of the timer or counter*. If it is during On-Line Monitoring and the PLC is executing SetTimerSV or SetCtrSV statement which affects the timer/counter S.V. you can see the changes immediately in this window.

Although changes to the S.V. of timer/counter in the target PLC will be captured on the "Edit Present Value" window in the On-Line Monitoring screen as explained above, such changes however will not be captured into the timer/counter definition tables. To capture and update the S.V. of the timer/counter into their definition tables please use the command

"2: PLC's Tim/Ctr SV -> Host"

in the “Controller” menu as explained in section 3. This apparent inconvenience is actually a deliberate design effort to avoid unintentional modification of the parameters in the source program during On-Line Monitoring.

1. Custom-Functions - An Overview

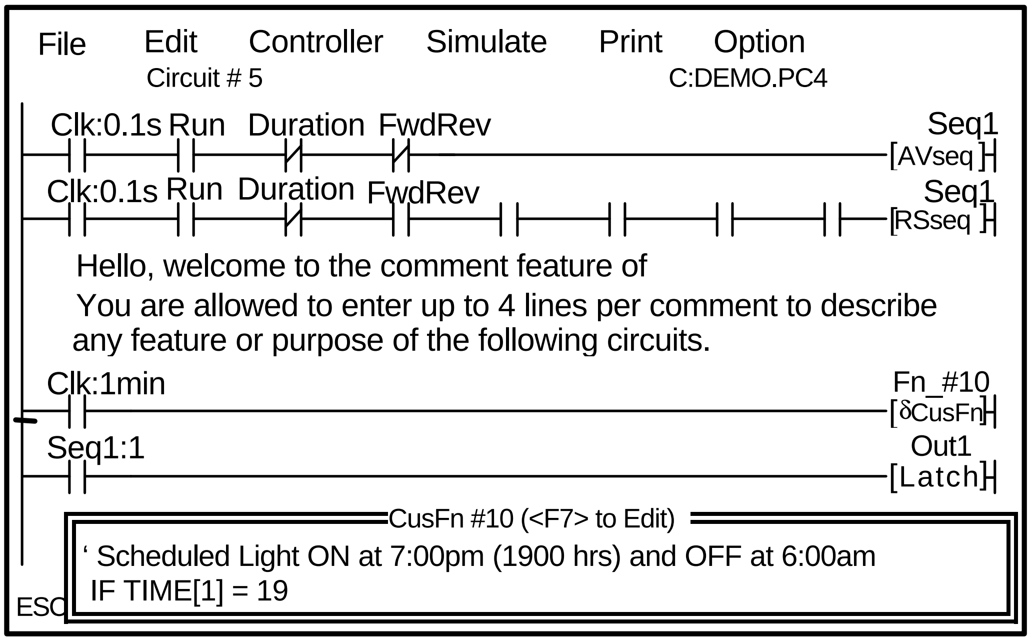
TRiLOGI Version 4.1 supports user-created special functions, known as Custom Functions (the symbol **CusFn** will be used throughout this manual to mean Custom Functions). Up to 256 CusFns can be programmed using a special language: **TBASIC**.

TBASIC is derived from the popular BASIC computer language widely used by microcomputer programmers. Some enhancements as well as simplifications have been made to the language to make it more suitable for use in PLC applications.

There are two simple ways to create a new CusFn:

a) From the "Edit" pull-down menu, select the item "Custom Function" and enter the function number which may range from 1 to 256. You may also use the hotkey <F7>. A Custom Function Editor window will appear.

b) If you have already created a ladder circuit which connects to either a [CusFn] or [δCusF] function (both appear as menu-item within the "Special Function" pop-up menu), move the browse mode cursor to the circuit which connects to that particular CusFn. You will notice a small window pops up at the bottom of the screen as follow:



Hello, welcome to the comment feature of
You are allowed to enter up to 4 lines per comment to describe
any feature or purpose of the following circuits.

```
' Scheduled Light ON at 7:00pm (1900 hrs) and OFF at 6:00am
IF TIME[1] = 19
```

The small window that pops up shows the first two lines of the definition of CusFn #10. If you press the <F7> function key at this point, a full text editor screen will be opened up showing the rest of the TBASIC statements for this function. Up to 60 lines of **TBASIC** statements can be entered for each CusFn. If more lines are required, a **CALL** *n* statement may be executed at the end of the first CusFn which will chain the execution to another CusFn #*n.*

2. Custom Function Editor

The custom function editor window allows creation of up to 60 lines of **TBASIC** program statements. Each line can contain a maximum of 70 characters. The extreme left column of the editor window shows the line number from 1-60. In one screen only 20 lines are visible. You can however easily scroll up and down in pages using the <PgUp>, <PgDn> key and the up/down cursor keys.

The custom function editor is very similar to any standard text editor. Simply key in the text at the location of the cursor. You may use the following keys for editing your program:

<Ctrl-Enter>	-- Insert a new line before the current line.
<Ctrl-Backspace>	-- Delete the current line.
<Cursor keys>	-- navigate within the editor.
<PgUp><PgDn>	-- scroll to the previous page or the next page.
<Backspace>	-- delete text to the left of the cursor.
<Del>	-- delete text to the right of the cursor.
<Home>	-- move cursor to the beginning of current line.
<End>	-- move cursor to the end of current line.
<Enter>	-- insert a line break at the current cursor position.
<Ctrl-Right arrow>	-- Move cursor to the beginning of next word.
<Ctrl-Left arrow>	-- Move cursor to the end of next word.
<Ctrl-C>	-- Copy editor's text into clipboard.
<Ctrl-P>	-- Paste clipboard's content into editor.
<Ctrl-N>	-- Jump to the NEXT Custom Function.
<Ctrl-B>	-- Jump to the CusFn BEFORE the current CusFn.

In TRiLOGI version 4.1 you can break a text line at any text position and push the remaining text to the next line when you press <Enter> key. If you press the <Backspace> key at the beginning of a line it will be merged to the previous line. Similarly if you press the <Del> key at end of a line it will append the next line to the current line.

Note that to insert a new line before the current line, you need to press the <Ctrl> key together with the <Enter> key. To delete the current line, press <Ctrl> key together with the <Backspace> key. If the current line is already deleted, then all the following lines will move one line up towards the current line.

Caution: If 60 lines have already been entered into the editor and a <Ctrl-Enter> or <Ctrl-P> key is pressed which causes a new line(s) to be inserted into the cursor line, then the content of the last line(s) in the editor will be lost.

2.1 Text Copying with Custom Function Editor

TRiLOGI Custom Function Editor allows you to copy a block of program statements to another destination, either within the same CusFn or into another CusFn.

To copy a block of program lines, press <Ctrl> and <C> key simultaneously. A message will be displayed along the bottom of the screen stating:

"Copying Text to Clipboard. Press <Enter> to complete...."

You may move the cursor using the up/down cursor keys or <PgUp> and <PgDn> keys. The range of text selected for copying will be highlighted within the editor window. When you have selected the range of text, press <Enter> key to complete the copying. The selected text is then copied into TRiLOGI's memory buffer known as the "Clipboard". The content of the Clipboard will be not be lost unless it is overwritten by the next <Ctrl-C> command or when the program terminates.

After copying the block of text into the Clipboard, move the cursor to the destination where the block is to be "pasted" and press the <Ctrl-P> key. The destination may be either in the same CusFn, in another CusFn or even in another TRiLOGI program file. The content of the clipboard will be inserted at the cursor position and the existing lines will be pushed downward. You may make as many copies as necessary as the content of the clipboard will not be changed by the "paste" command.

Note that this COPY function provides a convenient way for copying some existing **TBASIC** program codes into a new program for execution. You may also use it to organize your program to improve readability.

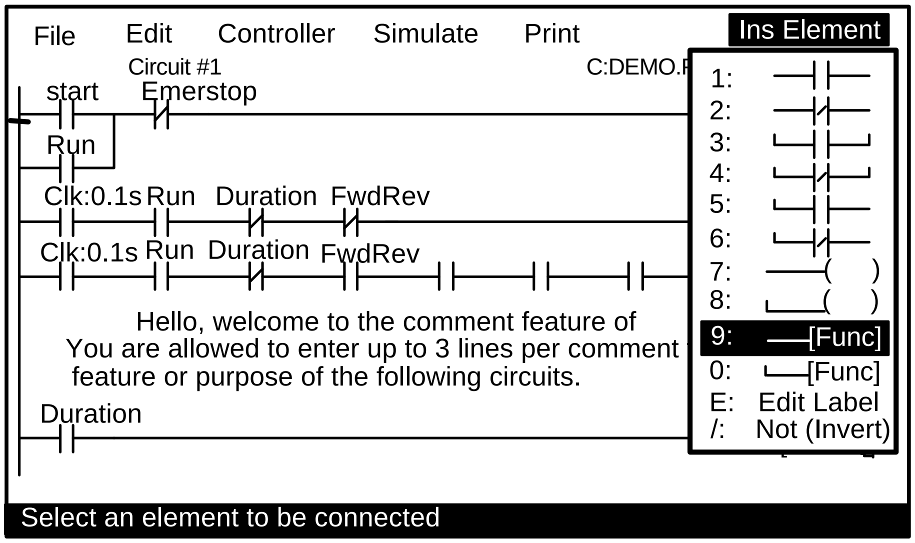
3. Custom Function Execution

It is important to understand when and how a TBASIC-based Custom Function is executed with respect to the rest of the program. There are basically two ways in which a CusFn will be executed:

3.1 Triggered by Ladder Logic Special function coil ——[CusFn]

A custom function may work the same way as any other special functions in the TRiLOGI ladder diagram programming environment. When in ladder circuit editing mode, press <Ins> key to open the "Ins Element" menu.

Select the item 9: ——[FUNC] or 0: └——[FUNC] to create a special function output. A pop-up "Select a Function" menu will appear.



Hello, welcome to the comment feature of
You are allowed to enter up to 3 lines per comment
feature or purpose of the following circuits.

Select an element to be connected

Select either item:

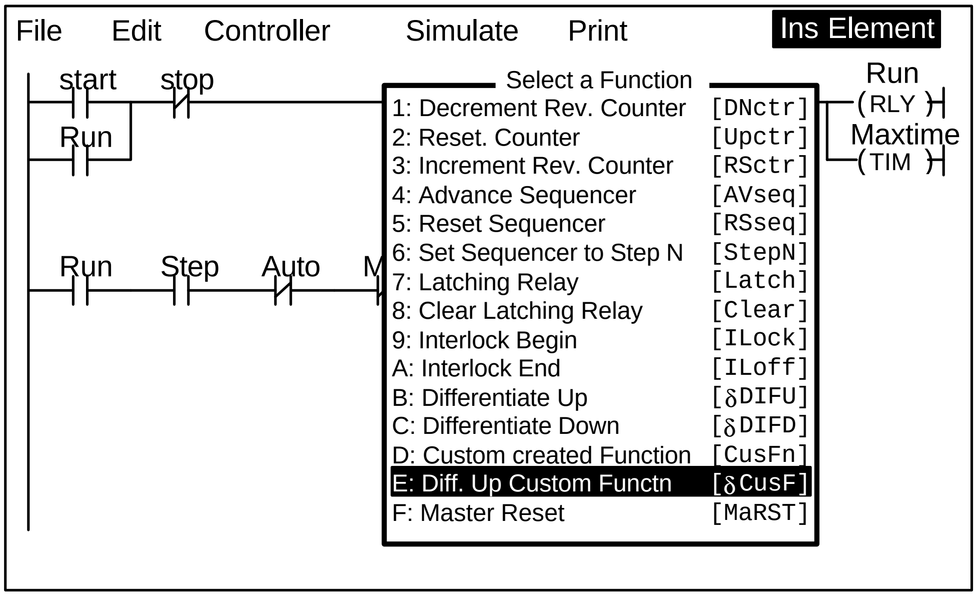
" D :Custom created Function [CusFn]" or item

" E :Diff. Up Custom Functn [δCusF]"

to create a CusFn. You will be required to enter the selected custom function number from 1 to 256. Note that CusFn created using

" E :Diff. Up Custom Functn[δCusF]"

is a "Differentiated Up" instruction. This means that the function will be executed **only once** every time when its execution condition goes from OFF to ON. Nothing will happen when its execution condition goes from ON to OFF.



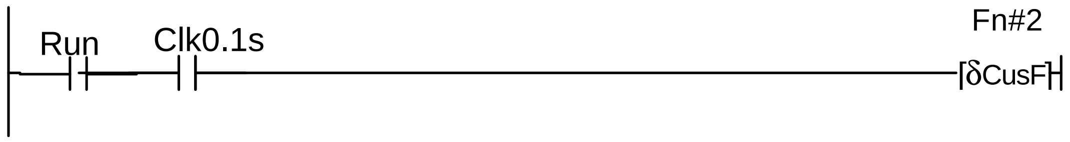
Select a Function

1: Decrement Rev. Counter	[DNctr]
2: Reset. Counter	[Upctr]
3: Increment Rev. Counter	[RSctr]
4: Advance Sequencer	[AVseq]
5: Reset Sequencer	[RSseq]
6: Set Sequencer to Step N	[StepN]
7: Latching Relay	[Latch]
8: Clear Latching Relay	[Clear]
9: Interlock Begin	[ILock]
A: Interlock End	[ILoff]
B: Differentiate Up	[δDIFU]
C: Differentiate Down	[δDIFD]
D: Custom created Function	[CusFn]
E: Diff. Up Custom Functn	[δCusF]
F: Master Reset	[MaRST]

On the other hand, using "D: Custom created Function [CusFn]" will means that the CusFn will be **executed every scan** as long as its execution condition is ON. This is often not desirable and the coil created using this menu item will be highlighted in RED color to serve as an alarm to programmer. You will probably find you will use the differentiated version [δCusF]far more frequently.

Periodic Execution of a Custom Function

There are many situations when you need the PLC to periodically monitor an event or perform an operation. For example, to monitor the temperature reading from a probe or check the real time clock for the scheduled time, and to continuously display changing variables on the LCD display. It is not efficient to use the continuous [CusFn] function for such purpose. It is far more better to use the built-in clock pulses to trigger a differentiated Custom function [δCusF]. You can choose a suitable period from 0.01s, 0.02s, 0.05s, 0.1s, 0.2s, 0.5s, 1.0s, & 1 minute for the application. Other periods can also be constructed with a self-reset timer. The custom function will only be executed once every period controlled by the system clock pulse or the timer, as follow:



You don't need to update the value of a variable displayed on the LCD screen any faster than the human eye can read them. So using a 0.5s clock pulse may be sufficient and this will not take up too much CPU time for the display. For slow process such as heating, a 1.0s clock pulse to monitor temperature change is more than sufficient.

IMPORTANT

1) When the CPU scans the ladder logic to a circuit which contains a CusFn, and the execution condition of the circuit is TRUE, the corresponding CusFn will be immediately executed. This means that the CPU will not execute the remaining ladder circuits until it has completed execution of the current CusFn. Hence if the CusFn modifies a certain I/O or variable, it is possible to affect the running of the remaining ladder program.

2) Note that the INPUT[n] variables contain data obtained at the beginning of the ladder logic scan and <u>not the actual state of the physical input</u> at the time of the CusFn execution. Thus it will be futile to wait for the INPUT[n] variable to change inside a CusFn unless you execute the **REFRESH** statement to refresh the physical I/O before you examine the INPUT[n] variable again.

3) Likewise, any changes to the OUTPUT[n] variable using the **SETBIT** or **CLRBIT** statement <u>will not be transferred to the physical outputs</u> until the end of the current ladder logic scan. Hence do not wait for an event to happen immediately after executing a SETBIT or CLRBIT statement on an OUTPUT[n] because nothing will happen to the physical output until the current ladder logic scan is completed.

   If you want to force the output to change immediately you will need to execute the **REFRESH** statement. Consideration must be given to how such an act may affect the other parts of the ladder program since not the entire ladder program has been executed.

4) Like all ladder circuits, the relative position of the circuit which triggers the CusFn may affect the way the program works. It is important to consider this fact carefully when writing your ladder program and TBASIC CusFns. Always remember that the CPU executes the ladder logic and CusFn sequentially, even though the equivalent circuits in hard-wired relay may seem to suggest that the different rungs of ladder circuits were to work simultaneously.

5) In line with the typical Ladder Logic programming rules, a CusFn may appear only once within the ladder diagram, regardless of whether it

appears in the normal or differentiated form. A compilation error will occur if a CusFn appears in more than one circuit.

However, a CusFn may be “CALLed” as a subroutine by any other CusFn and there is no restriction placed on the number of repeated CALL of a CusFn by more than one CusFn. A CusFn may also modify the logic states of an i/o element or the value of internal timers and counters using its powerful TBASIC commands (such as SetBit, ClrBit). The compiler however will not alarm the user that a CusFn may inadvertently alter the logic state of an I/O already controlled by some other ladder circuit.

This power and flexibility offered by the TBASIC-based custom functions must therefore be handled with greater care by the programmer. It is important to prevent conflicting output conditions due to an i/o being controlled or modified at more than one place within a logic scan. The net result is that the logic state of the i/o appears to be in different states at different parts of the ladder circuit. This could lead to bizarre outcomes that may be difficult to trace and debug.

3.2 Interrupt Service CusFn

A CusFn may also serve as an “Interrupt Service Routine” which is executed asynchronously from the normal ladder logic execution. An interrupt-driven CusFn is run when the condition which causes the interrupt occurs. The response time to execution is very short compared to the scan time of the ladder program. There are several interrupt sources which can trigger a CusFn:

a) Special Interrupt inputs

An M-series PLC contains some special “Interrupt” inputs which, when enabled by the **INTRDEF** statement, will trigger a particular CusFn defined in the INTRDEF statement when the logic level at the interrupt pin changes state (either from OFF to ON or from ON to OFF).

b) High Speed Counter (HSC) Reach Target Count

An M-series PLC contains some “High Speed Counter” inputs which, when enabled by the **HSCDEF** statement, will trigger a particular CusFn defined in the HSCDEF statement when the counter reaches a preset target count value. This enables the CPU to carry out some immediate actions such as stopping a motor or performing some computation.

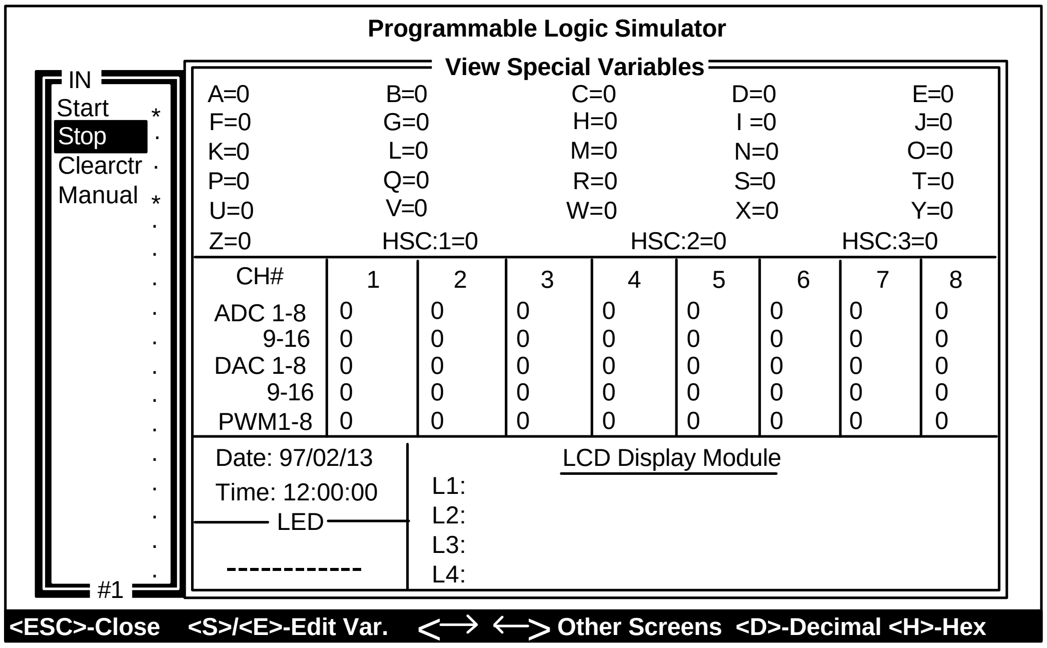
4. Simulation & Examination of TBASIC Variables

4.1 Simulation Run of CusFn.

TRiLOGI fully supports simulation of all **TBASIC** commands. After you have completed coding a CusFn, test the effect of the function by connecting it to an unused input. Run the simulator by pressing <F9> or <Ctrl-F9> key. Execute the CusFn by turning ON its input. If your CusFn executes a command that affects the logic state of any I/O the effect can be viewed on the simulator screen immediately. However, if the computation affects only the variables than you may need to examine the internal variables.

4.2 Viewing TBASIC Variables

The values of the internal variables as a result of the simulation run by pressing the <V> (which stand for "View") key while in the simulation screen. A pop-up window will appear with the values of all the variables as well as special peripheral devices supported by TBASIC. The variables are organized into 3 screens. You can move from screen to screen using the left/right cursor keys:



```
                    Programmable Logic Simulator
 IN            View Special Variables
 Start    *   A=0        B=0      C=0       D=0       E=0
 Stop     .   F=0        G=0      H=0       I =0      J=0
 Clearctr .   K=0        L=0      M=0       N=0       O=0
 Manual   *   P=0        Q=0      R=0       S=0       T=0
          .   U=0        V=0      W=0       X=0       Y=0
          .   Z=0        HSC:1=0      HSC:2=0     HSC:3=0
          .   CH#       1    2    3    4    5    6    7    8
          .   ADC 1-8   0    0    0    0    0    0    0    0
          .       9-16  0    0    0    0    0    0    0    0
          .   DAC 1-8   0    0    0    0    0    0    0    0
          .       9-16  0    0    0    0    0    0    0    0
          .   PWM1-8    0    0    0    0    0    0    0    0
          .   Date: 97/02/13        LCD Display Module
          .   Time: 12:00:00    L1:
          .   ----- LED -----   L2:
          .                     L3:
 #1       .   ------------      L4:
<ESC>-Close  <S>/<E>-Edit Var.  <→ ←> Other Screens  <D>-Decimal <H>-Hex
```

a) System variables Screen

The first screen comprises all 26 32-bit integer variables A-Z, the system DATE and TIME, ADC, DAC, PWM and the resulting values of setLED and setLCD commands. The DATE and TIME figure shown during simulation are taken from the PC internal real-time clock values. The present values of the first 3 high speed counters: HSC:1 to

HSC:3 are also shown on this page. Note that ADC data for any particular A/D channel #n will only be shown if an ADC(n) function has been executed. Otherwise the ADC value shown on screen will not reflect the true current value of the ADC port.

b) Data Memory Screen

The second screen displays, in 25 pages, the values of the 16-bit DM variables from DM[1] to DM[4000]. Each page displays 16 rows x 10 columns = 160 DM variables. You can scroll up and down the pages using the <PgUp> and <PgDn> keys.

c) String Variable Screen

The third screen displays the value of the 26 string variables A$ to Z$ in one or more pages, depending on the length of each string.

If the execution condition is ON and the CusFn is not of the differentiated type, then the CusFn will be continuously executed. The result of the variable will be continuously updated on the viewing window.

4.3 Viewing TBASIC Variables When Editing CusFn

You can also open up the “View Special Variables” window when you are editing or browsing a CusFn by pressing <Ctrl-V> keys.

4.4 Changing the Contents of Variables

While the "View Special Variables" window is open, you may change the contents of the following variables by pressing the <E> key (which stands for EDIT):

```
A-Z, A$ to Z$, DM[n], DATE[n], TIME[n], INPUT[n],
OUTPUT[n], RELAY[n], TIMERBIT[n], CTRBIT[n],
TIMERPV[n], CTRPV[n] and HSCPV[n],
emINT[n], emLINT[n].
```

A text entry window will pop up and you will have to enter the values in the form of assignment statements, such as follow:

```
e.g. A = 5000;
     DM[99]=5678;
     OUTPUT[2]=&H01AB
     B$ = "Welcome to TBASIC"
```

The variable will take up the new value as soon as it is entered, and if the execution condition for any CusFn is ON, the simulator will process the newly entered data immediately and produce the new outcomes. This gives you greater flexibility in controlling the simulation process.

4.5 Showing the Variables' Content

If you press the <S> key when the "View Special Variables" window is opened, you will be prompted to enter the variable name and the value of this variable will be displayed immediately. The variable name is the same as that shown in the last paragraph. This is useful for checking the value of the system variables such as "INPUT[n]", "OUTPUT[n]", etc., which are not shown on the "Special Variables" window. For the EMIT link variables emINT[n] and emLINT[n] these are the only way to show their values.

4.6 Decimal and Hexadecimal Representation

All the numeric data shown in the "Special Variables" window are by default displayed in decimal notation. You can display the number in hexadecimal format by pressing the <H> key. Press the <D> key if you wish to switch back to the decimal format. This feature is very useful for programmer who are familiar with hexadecimal representation of a binary number.

5. On-line Monitoring of TBASIC Variables

If you execute the "On-Line Monitoring/Control" command from the "Controller" pull-down menu, TRiLOGI Version 4.1 will continuously query the PLC for the values of all their internal variables. These variables' values will be updated in real time in the "View Special Variables" window, the "Edit Variable" and the "Show Variable" windows described in the previous section. You may also alter the value of any variables in the PLC using the "Edit Variable" window (by pressing the <E> key when at the "View Special Variables" window.

This ability of TRiLOGI to provide instant and full visibility of the all the PLC's internal variables greatly facilitates the programmers' debugging process. The ease of programming offered by the TRiLOGI programming environment is really what really sets the M-series PLC far ahead of many other PLCs where both programming and debugging are really painstaking tasks. (This is assuming they have been fully equipped with all the expensive "options" to match the M-series built-in capability!)

PAUSE and RESET of Target PLC

During On-Line Monitoring, if the "View Special Variables" window is opened, you can still reset the PLC's internal data by pressing the <Ctrl-R> key. The PLC can also be halted by pressing the <P> key. A halted PLC can be subsequently released from the halted mode by pressing the <P> key again.

Using LCD Display for Debugging

You should take advantage of the built-in LCD display port of the T100MD to display internal data at location where you want to track their values. Especially for EMIT link variables emINT[n] and emLINT[n], it is much easier to see their values continuously on the LCD then using the "Show variable" command on TRiLOGI.

6. Error Handling

Since the CusFn text editor does not restrict the type of text that may be entered into its editor, the TRiLOGI compiler will have to check the syntax of user's TBASIC program to look out for miss-spelling, missing parameters, invalid commands, etc. Such errors which can be tracked down during compilation process are know as "Syntax Errors".

6.1 Syntax Error

TRiLOGI employs a sophisticated yet extremely user-friendly syntax error tracking system: When a syntax error is encountered, the compilation will be aborted immediately and the CusFn which contains the error is automatically opened in the text editor. The location of the offending word is also highlighted and a pop-up message window reports to you the cause of the error. You can then immediately fix the error and re-compile until all the errors have been corrected.

Error Message	Cause / Action
1. Undefined symbol found	Only TBASIC commands and legal variable names are allowed. See Chapter 3.
2. Compiler internal error	Serious trouble, please email to manufacturer support@tri-plc.com
3. ") " found without matching " ("	-
4. Integer expected	Expect to see either an integer variable or integer constant.
5. Value is out-of-range	Check the language reference for allowable range of value for the command.
6. Duplicate line label number	Label for goto must be unique within the same CusFn.
7. Undefined GOTO destination:	Put a matching label at the place where the GOTO statement suppose to go.
8. Invalid GOTO label	@# must be in the range 0-255

9. Type mismatch (numeric and string types may not mix)	In an expression, string and integers may not be mixed unless converted using the conversion function. e.g. STR$, VAL, etc.
10. String is too long	A string is limited to 70 characters
11. Too many line label	There should not be more than 20 GOTO label within the same CusFn.
12. Unknown Keyword	Most likely wrong spelling for TBASIC statement or function.
13. WHILE without ENDWHILE	Every WHILE statement must be ended with a matching ENDWHILE statement. Nested WHILE loop must have proper matching ENDWHILE for each WHILE.
14. IF without ENDIF	Every IF statement must be ended with a matching ENDIF statement to define the boundaries for the block controlled by the IF statement. For multiple IF THEN statement, each IF must be matched by a corresponding ENDIF.
15. FOR without NEXT	Every FOR statement must be ended with a matching NEXT statement to define the boundaries for the block controlled by the FOR statement. For nested FOR loops, each FOR must be matched by a corresponding NEXT.
16. Expect keyword "TO"	Required by FOR statement.
17. Must be an integer	String variable or constant not allowed.
18. Must be an integer variable only	Integer constant not allowed.
19. Must be an integer constant only	Integer variable not allowed.
20. Must be a string	Integer constant or variable not allowed.
21. Must be a string variable only	String constant not allowed.
22. Must be a string constant only	String variable not allowed.
23. Incomplete Expression	Expression not ended properly.
24. String constant missing closing "	String constants must be enclosed between a pair of opening and closing "
25. Must be Integer A to Z only	index for FOR..NEXT loop must be A-Z.

6.2 Run-Time Errors

Certain errors only become apparent during the execution of the program, e.g. A = B/C . This expression is perfectly OK except when C = 0, then you would have attempted to divide a number by zero, which does not yield any meaningful result, in this case a “run-time error” is said to have occurred. Since run-time errors cannot be identified during compilation, TRiLOGI also checks the validity of a command during simulation run and if a run-time error is encountered, a pop-up message window will report to the programmer the cause and the CusFn where the run-time error took place. This helps programmer locate the cause of the run-time errors to enable debugging. The possible run-time errors are listed in the following table and they are generally self-explanatory.

Run-Time Error Message
Divide by zero
Call stack overflow! Circular CALL suspected!
FOR-NEXT loop with STEP = 0!
SET_BIT position out-of-range!
CLR_BIT position out-of-range!
TEST_BIT position out-of-range!
STEPSPEED channel out-of-range!
Illegal Pulse Rate for STEPMOVE!
Illegal acceleration for STEPMOVE!
STEPMOVE channel out-of-range!
STEPSTOP channel out-of-range!
ADC channel out-of-range
DAC channel out-of-range
LED Digit # within (1-12) Only!
PWM Channel out-of-range!
LCD Line # must be (1-4) Only!
PM channel out-of-range!
System Variable Index Out-of-range!
Shifting of (A-Z) Out-of-range!
Illegal Opcode - Please Inform Manufacturer!
Timer or Counter # Out-of-Range!

Chapter 3 - Statement, Functions, Data & Operators

1. Statements, Functions and Delimiter.

1.1 STATEMENT

A STATEMENT is a group of keywords used by TBASIC to perform certain action. A statement may take 0,1,2 or more arguments. The following are some TBASIC statements: PRINT, LET, IF, WHILE, SETLED ...etc.

1.2 FUNCTION

A FUNCTION acts on its supplied arguments and return a value. The returned value may be an integer or a string. A function can usually be embedded within an expression as if it is a variable or a constant, since its content will be evaluated before being used in the expression. e.g.

A$ = "Total is $"+STR$(B+C)

STR$(n) is a function which returns a string and therefore can be used directly in the above string assignment statement.

The most distinguishable feature of a FUNCTION is that its arguments are enclosed within parenthesis "(" and ")". e.g. ABS(n), ADC(n), MID$(A$,n,m), STRCMP(A$,B$).

Note: Statements or functions and their arguments are **NOT** case-sensitive. This means that commands such as `PRINT` and `PriNt` are identical. However, for clarity seek we use a mix of upper and lower case characters in this manual.

1.3 DELIMITER

A TBASIC program consists of many statements. Each statements are usually separated by a different line. The new line therefore acts as a "delimiter" which separate one statement from another. Some statements such as IF..THEN..ELSE..ENDIF span multiple statements and should be separated by proper delimiters.

To make a program visually more compact, the colon symbol ":" may be used to act as delimiter. e.g.

```
IF  A > B THEN
     C = D*5
ELSE
     C = D/5
ENDIF
```

may be written more compactly as

```
IF A >B : C=D*5:ELSE:C=D/5:ENDIF
```

2. Integer Data

The TBASIC compiler in TRiLOGI Version 4.1 supports full 32-bit integer computations. However, only variable A to Z are 32 bits in length which allow them to represent number between -2^{31} to -2^{31}, the remaining system variables and data memory DM[n] are all 16-bit variables which means that they can only store number between -32768 to +32767. However, all numerical computations and comparisons in TBASIC are carried out in 32-bit signed integer, regardless of the bit-length of the variables involved in the numerical expression.

2.1 Integer Constants

These may be entered directly in decimal form, or in *hexadecimal* form by prefixing the number with the symbol "&H". e.g.

```
12345678
&H3EF     =1007 (decimal)
```

If the result of an expression is outside the 32-bit limits, it will overflow and change sign. Care must therefore be exercised to prevent unexpected result from an integer-overflow condition.

A constant may be used in an assignment statement or in an expression as follow:

```
A = 12345
IF A*30 + 2345/123 > 100 THEN ....ENDIF
```

IMPORTANT

When entering an integer constant using the hexadecimal prefix “&H”, it is important to note the sign of the intended value and extend the signs to most significant bit of the 32 bit expression. E.g. to represent a decimal number “-1234”, the hexadecimal representation must be “&HFFFFFB2E” and not “&HFB2E”.

Assuming that a 16-bit variable DM[1] contains the number -1234 and a comparison statement is made to check if the number is -1234. The 32-bit hexadecimal representation of constant -1234 is &HFFFFFB2E. If you enter the constant as 16-bit representation “&HFB2E” as follow:

```
IF   DM[1] <>  &HFB2E  CALL 5
```

TBASIC translates the number “&HFB2E” into a 32-bit decimal number 64302, which when compared to the number “-1234” contained in DM[1] will yield a “False” result which is an error. The following are the correct representation:

a) IF DM[1] < > -1234 CALL 5 : ENDIF
or b) IF DM[1] < > &HFFFFFB2E” CALL 5: ENDIF

2.2 Integer variables:

Variables are memory locations used for storing data for later use. *All Integer variables used in TBASIC are GLOBAL variables* - this means that all these variables are shared and accessible from every custom function.

TBASIC supports the following integer variables:

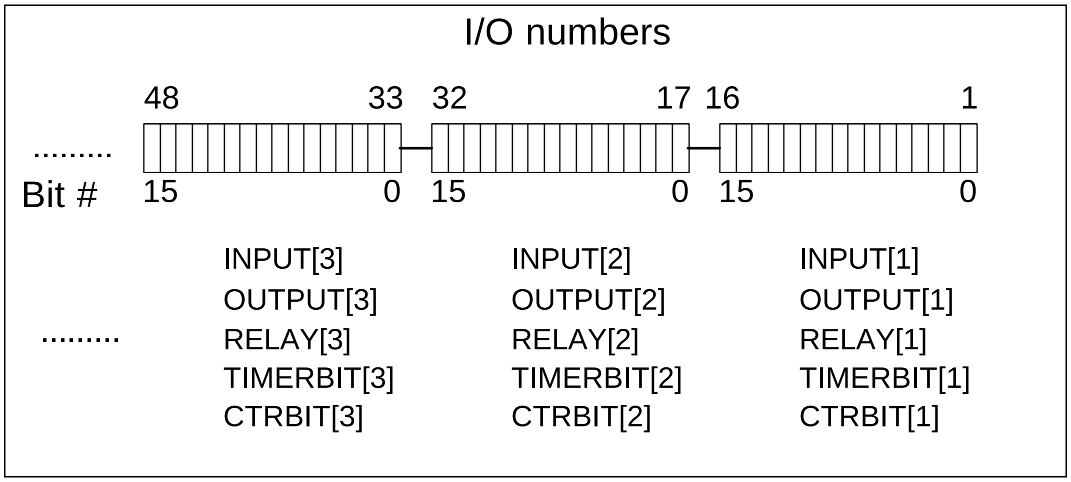
i) 26 Integer variables A, B, C....Z which are 32-bit variables. Note that the variable name must be a single character.

ii) A large, one-dimensional 16-bit integer array from DM[1] to DM[4000], where DM stands for Data Memory. A DM is addressed by its index enclosed between the two square brackets "[" and "]". e.g. DM[3], DM[A+B*5], where A and B are integer variables.

iii) System variables. These are special integer variables which relates to the PLC hardware, as follow:

I/Os, Timers and Counters Contacts

The bit addressable I/Os elements are organized into 16-bit integer variables INPUT[*n*], OUTPUT[*n*], RELAY[*n*], TIMERBIT[*n*] and CTRBIT[*n*] so that they may be easily accessed from within a CusFn. These I/Os are arranged as shown in the following diagram:



Timers and Counters Present Values

The present values (PV) of the 128 timers and 128 counters in the PLC can be accessed directly as system variables:

timerPV[1] to timerPV[128], for timers' present value
ctrPV[1] to ctrPV[128], for counters' present value

DATE and TIME Variables

The PLC's Real-Time-Clock (RTC) derived date and time can be accessed via variables DATE[1] to DATE[3] and TIME[1] to TIME[3], respectively as shown in the following table:

Date		Time	
YEAR	DATE[1]	HOUR	TIME[1]
MONTH	DATE[2]	MINUTES	TIME[2]
DAY	DATE[3]	SECOND	TIME[3]
Day of Week	DATE[4]		

DATE[1] : may contain four digits (e.g. 1998, 2003 etc).
DATE[4] : 1 for Monday, 2 for Tuesday, ... 7 for Sunday.

High Speed Counters

The M-series PLC support High Speed Counters (HSC) which can be used to capture high frequency incoming pulses from positional feedback encoder. These high speed counters are accessible by CusFn using the variables HSCPV[1] to HSCPV[8]. All HSCPV[n] are 32-bit integer variables.

2.3 Integer operators:

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

i) Assignment Operator: An integer variable (A to Z, DM and system variables, etc) may be assigned a value using the assignment statement:

```
A = 1000
X = H*I+J + len(A$)
```

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)`
MOD	Modulus	`Y = Y MOD 10`

iii) Bitwise Logical Operators: logical operations is perform bit-for-bit between two 16-bit integer data.

Symbol	**Operation**	**Example**
&	logical AND	`IF input[1] & &H02 ...`
|	logical OR	`output[1] = A	&H08`
^	Exclusive OR	`A = RELAY[2] ^ B`
~	logical NOT	`A = ~timerPV[1]`

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 **OR** B=1000

v) Functional Operators : TBASIC supports a number of built in functions which operate on integer parameters as shown below:

ABS(*n*), ADC(*n*), CHR$(*n*), HEX$(*n*), STR$(*n*)

For detailed explanation of these functions please refer to the next chapter: “Programming Language Reference”

2.4 Hierachy of Operators:

The hierarchy of operators represent the priority of computation. Eg. X = 3 + 40*(5 - 2). The compiler will generate codes to compute 5 - 2 first because the parentheses has the higher hierarchy, the result is then multiplied by 40 because multiplication has a higher priority then addition. Finally 3 will be added to the result. If two operators are of the same hierarchy, then compiler will evaluate from left to right. e.g. X = 5 + 4 - 3. 5+4 is first computed and then 3 will be subtracted. The following table list the hierarchy of various operator used.

Hierarchy	Symbol	Descriptions
Highest	( )	Parentheses
↓	*, / , MOD	Multiplication/Division
	+, -	Add/Subtract
	-	Negate
	&, |, ^ ,~	Logical AND,OR,XOR,NOT
Lowest	=,<>,>,>=,<,<=	Relational operators

3. String Data

A string is a sequence of alphanumeric characters (8-bit ASCII codes) which collectively form an entity.

3.1 <u>String Constants</u>

A string constant may contain from 0 to 70 characters enclosed in double quotation marks. e.g.

"TBASIC made PLC numeric processing a piece of cake!"
"$102,345.00"

3.2 <u>String Variables</u>

TBASIC supports a maximum of 26 string variables A$, B$ ... Z$. Each string variable may contain from 0 (null string) up to a maximum of 70 characters.

3.3 <u>String Operators</u>

i) <u>Assignment Operator</u>: A string variable (A to Z, DM and system variables, etc) may be assigned a string expression using the assignment statement:

```
A$ = "Hello, Welcome To TBASIC"
Z$ = MID$(A$,3,5)
```

ii) <u>Concatenation Operators</u>: Two or more strings can be concatenated (joined together) simply by using the "+" operator. e.g.

```
M$ = "Hello " + A$ + ", welcome to " + B$
```

If A$ contains "James", and B$ contains "TBASIC", M$ will contain the string: "Hello James, welcome to TBASIC.

iii) Comparison Operators: Two strings may be compared for equality by using the function STRCMP(A$,B$). However, the integer comparator such as "=", "<>", etc cannot be used for string comparison.

iv) Functional Operators: **TBASIC** supports a number of statement and functions which take one or more string arguments and return either an integer or a string value. e.g.

LEN(*x$*),	MID$(*A$,x,y*),	PRINT #1 *A$*,....
SETLCD 1, *x$*	VAL(*x$*),	

Please refer to the next chapter for detailed descriptions of these operators.

4. Link Variables for EMIT 3.0 (Internet Connectivity)

The T100MD+ and T100MX+ PLCs can be linked to the internet via a special "emGateway™" software supplied by the emWare Inc of Salt Lake City, USA. emGateway runs on any Windows 95/98/NT PC. The M+ series PLC incorporates the emMicro code licensed from emWare which allows a JAVA applet to be easily developed so that the PLC's internal data can be accessed by any browser on the internet from literally anywhere in the world!

The emGateway acts as the middle man between the internet and the M+ PLCs. It uses predefined variable names in the PLC and through a JAVA applet allows exchange of data between a JAVA-enabled internet browser such as Netscape or Microsoft Internet Explorer 4.0 and above.

To allow greater flexibility in programming and for protection of internal data, TBASIC does not expose existing internal system variables to the emGateway. Instead, TBASIC creates 32 special system variables for the sole purpose of interacting with emGateway. The user program can therefore control what data is to be exposed or obtained from the internet. The data to be exposed will be copied to the special em-variables and data obtained from the internet can be used selectively by the control program.

Pre-defined Variable Names for emGateway

The following variables name are defined in the emMicro code implemented by M+ series PLCs. These are the names to use when you write the JAVA applet user interface.

a) `emInt1 to emInt16` : These are 16 bit unsigned integer variables
b) `emLInt1 to emLInt16`: These are 32-bit unsigned integer variables.
c) `emStringA` and `emStringB`: These two are byte array of 70 characters each, used mainly as strings variables.

These variables have a one to one correspondence with the following system variables defined in TBASIC:

emMicro (case sensitive)	**TBASIC** (non case sensitive)
`emInt1 to emInt16`	`EMINT[1] to EMINT[16]`
`emLInt1 to emLInt16`	`EMLINT[1] to EMLINT[16]`
`emStringA`	`A$`
`emStringB`	`B$`

Chapter 4 - Programming Language Reference

ABS(*x*)

Purpose : To return the absolute value of the numeric expression *x*

Examples : `A = ABS(2*16-100)`

Comments : A should contain the value 68.

ADC(*n*)

Purpose : To return the value from the Analog-To-Digital Converter channel #*n*. n should be between 1 and 16.

Examples : `A = ADC(2)`

Comments : n may be a numeric expression which returns a value between 1 and 16. If it is out-of-range, a run-time error will be reported and the function will be aborted.

TRiLOGI software is able to support up to 16 channels of 16-bit bipolar ADC (which may has a range of between -32768 to 32767. The actual number of ADC channels and the resolution will depend on the target PLC. On the T100MX, all the A/D are normalized to 12-bit with a range of between 0 and 4096

ASC(*x$*, *n*)

Purpose : To return the numeric value that is the ASCII code for the *n*th character of the string *x$*. If *x$* is a null string, ASC(*x$*,*n*) returns value 0. *n* may start from 1 up to the length of the string.

Examples : `B = ASC("Test String",6)`

Comments : B should contain the value 83 (which is ASCII value of 'S'). If n is less than 1 or greater than string length, **ASC(*x$*, *n*)** *returns a 0.*

See Also : **CHR$(n)**

CALL *n*

Purpose : To call another Custom Function CusFn #*n* as subroutine. When the called function returns, execution will continue from the following statement. *n* must be an <u>integer constant</u> between 1 and 128.

Examples : `IF  B > 5 THEN   CALL 8 : ENDIF`

See Also : RETURN

CHR$(*n*)

Purpose : To convert a number *n* into its corresponding ASCII character. *n* must be a numeric constant (0 to 255)

Examples : `C$ = "This is Message #" + CHR$(&H35)`

Comments : C$ should contain: "This is Message #5", since CHR$(&H35) returns the character '5'.

See Also : **ASC( )**

CLRBIT *v*, *n*

Purpose : To clear the Bit #*n* of the integer variable *v* to '0'. *n* is an integer constant or variable of value between 0 and 15. *v* may be any integer variable or a system variable such as relay[n], output[n], etc. If *v* is a 32-bit integer, **CLRBIT** will only operate on the lower16 bits.

Following digital electronics convention, bit 0 refers to the least significant bit (right most bit) and bit 15 the most significant bit (left most bit) of the 16-bit integer variable. A quick way to find out the bit position and index of an I/O variable is to open their I/O table and check the "CH:BIT" column. Bit position beyond 9 are represented by hexadecimal number A to F.

Examples : `CLRBIT output[2],11`

Comments : Output #28 will be turned OFF.
(Output channel #2 bit #11 = Output #17 +11 = 28)

See Also : **SETBIT, TESTBIT**

---

* CLRIO *labelname* {* Applicable only to **M+** PLC models}
* SETIO *labelname*
* TOGGLEIO *labelname*
* TESTIO (*labelname*)

Purpose : Manipulate the logic states of any input, output, relay, timer or counter contact bit within a CusFn. The *labelname* refers to the label names defined in the input, output, relay, timer or counter tables.

SETIO set a bit to ON, **CLRIO** clear the bit to OFF, and **TOGGLEIO** flip the current logic state of that I/O bit. **TESTIO** function returns a 1 if the bit is ON and a 0 if the bit is OFF.

E.g.
```
SETBIT  alarm
IF  TESTBIT(alarm) THEN … ELSE …ENDIF
```

Comments This function offers a more efficient way of manipulating the I/O bits compared to the SETBIT and CLRBIT function. However, SETBIT and CLRBIT functions has the advantage that they can use variables to indicate the index and bit position of the bit to be affected, whereas the I/O bit that are being affected by the commands here are fixed during compile time.

Note that output bit changed in custom function will only be updated at the physical output at the end of the ladder logic scan unless a "REFRESH" command is being executed.

See Also : **SETBIT, CLRBIT, TESTBIT**

---

* DELAY *n*

{* Applicable only to **M+** PLC models}

Purpose : To provide a time delay of *n* millisecond to the process.

Example :

```
DELAY 100
```

Comments : Provide a 100 ms (0.1s) delay to the current custom function. It is important to note that this is a "brute force" delay method and only to be used with caution. When a DELAY function is executed the CPU waits at the statement until the period specified by the "delay" is over. This means that all the remaining ladder programs and other custom functions will stop responding to changing input conditions, only system services (serial input, countdown timers and host link commands etc) as well as interrupt driven CusFns will work during the period of delay. This may not be desirable if the rest of the process must respond to fast changing inputs. For delays longer than 0.1s a much better way is to invoke the regular PLC timer and use the timer contact to trigger another custom function at the end of the delay.

For T100MD+ and T100MX+, the minimum delay provided by this function is 10ms, and the resolution of the time delay is 10ms. This means that if you execute DELAY 155 the actual delay will be rounded to 160ms, whereas for DELAY 154 the actual delay will be 150ms.

FOR ... NEXT

Purpose : To execute a series of instructions for a specified number of times in a loop.

Syntax :

FOR *variable* **=** *x* **TO** *y* [**STEP** z]

. . . .

NEXT

where *variable* may be any integer variable A to Z only and is used as a counter. *x*, *y* and z are numeric expressions. STEP z is an optional part of the statement.

x is the initial value of the counter, *y* is the final value of the counter. Program lines following the **FOR** statement are executed until the **NEXT** statement is encountered. Then the counter is incremented by the amount specified by **STEP**. If **STEP** is not specified, the increment is assumed to be 1.

A check is performed to see if the value of the counter is greater than the final value *y* if **STEP** is positive (or smaller than the *y* if **STEP** is negative). If it is not greater, the program branches back to the statement after the **FOR** statement, and the process is repeated. If it is greater, execution continues with the statement following the **NEXT** statement. This is called a **FOR-NEXT** loop.

A run-time error will result if **STEP** is evaluated to be 0.

Examples :

```
FOR I=1 TO 10
   FOR J = 100 to 1 STEP -10
        DM[I] = DM[J]
   NEXT
NEXT
```

Comments : FOR-NEXT loops may be nested; i.e. a FOR-NEXT loop may be placed within the context of another FOR-NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. Each Loop must have a separate NEXT statement to mark the end of the loop.

See Also : **WHILE ... ENDWHILE**

GetCtrSV (*n*)
GetTimerSV (*n*)

Purpose : Return the **Set Value (S.V,)** of the Counter #*n* or Timer #n. *n* should be between 1 and 128.

Note : Although the present values (P.V.) of timers and counters #n can be accessed directly as variables "TimerPV[n]" & "CtrPV[n]", the Set Values however can only be obtained by these two functions.

See Also : **SetCtrSV, SetTimerSV**

GETHIGH16(*v*)

Purpose : This function returns the upper 16-bit of a 32-bit integer variable *v*. This can be used to break the value of a 32-bit integer data or variable into two 16-bit values so that they can be saved to the EEPROM or to the DM[n].

Examples :

```
DM[1] = GetHIGH16(A)
save_EEP  GetHIGH16(&H12345678), 10
```

See Also : **SETHIGH16**

GOTO @ *n*

Purpose : To branch unconditionally out of the normal program sequence to a specified line with label @*n* within the present Custom Function.

The destination line must have a corresponding line label marked as "@*n*", where *n* must be a constant within 0-255. Note that the label is local only to the present CusFn. i.e. another CusFn may have a label with the same *n* but the GOTO @*n* will only branch to the line label within the same CusFn.

Examples :

```
@156 SETBIT 0,3
      . . .
GOTO @156
```

Comments : An error message will appear during compilation if the destination label is undefined.

HEX$(*n*)
* HEX$ (*n*, *d*)

{* Applicable only to **M+** PLC models}

Purpose : To return a string that represents the hexadecimal value of the numeric argument *n*. If the second format is used then this function will return a string of '*d*' number of characters.

Examples :

```
A$ = HEX$(1234)
B$ = HEX$(1234,7)
```

Comments : A$ will contain the string : "4D2" , B$ will contain the string "00004D2".

See Also : **HEXVAL( ), STR$( ), VAL( )**

HEXVAL(*x$*)

Purpose : To return the value of a hexadecimal number contained in the argument x$.

Examples : `B = HEXVAL("123")*100`

Comments : B should contain the value 29100 (&H123 =291)

See Also : **HEX$( ), STR$( ), VAL( )**

HSTIMER *n*

Purpose : To define PLC Timer #1 to #n as "High Speed Timers" (HST). A HST counts down every 0.01s instead of every 0.1s for normal timer, and their other properties are identical to normal timer. Those Timers whose number are above n are not affected and remain ordinary timers.

HSCDEF *ch, fn_num, value*

Purpose : Enable and set up parameters for the High Speed Counters channel *ch*. These counters operate independently of the ladder logic scan time and can capture high speed input pulses generated by position encoders.

ch	= channel number (1-8)
fn_num	= Custom Function # to trigger when value is reached.
value	= trigger when HSC reach this (32-bit) integer value.

If the PLC supports quadrature encoder inputs, then the HSC counter variable **HSCPV[*ch*]** will increment/decrement according to direction of rotation. When *value* is reached, the specified custom function activates immediately.

Important : All High Speed Counters are disabled automatically when the PLC is reset unless they are enabled by the **HSCDEF** statement. However, if more than one **HSCDEF** for the same channel *ch* is executed, only the last executed **HSCDEF** statement will take effect. Hence you should put the next HSCDEF statement within the CusFn triggered by the first **HSCDEF**. By chaining the **HSCDEF** statement from one CusFn to another, you can control the motion of the machine using the HSC value to execute a series of CusFn one by one. Within these CusFn you can program what to do to control the motion. E.g. changing the speed, putting on the brake, change direction of motion, etc. You can use the SETBIT, CLRBIT for digital ON/OFF control and setDAC, setPWM for proportional control.

Example :
```
HSCPV[1] = 0
HSCDEF 1,19,-3310003
. . . .
SETLCD 1,1,STR$(HSCPV[1],6)
```

Comments : Enable High-Speed Counter #1 and make it activate function #19 when the counter reaches -33,100,003. Present value of HSC#1 was cleared to 0 before activating it. Note that TRiLOGI Version 4.1 does not perform simulation of the High Speed counter operation since there is no High Speed Counter inputs on the simulator screen.

See Also : **HSCOFF**

HSCOFF *ch*

Purpose : Disable High Speed Counter #*ch* (*ch* = 1 to 8)

If you no longer need the high speed counter, it should be disabled in order not to waste the CPU's time to service the interrupt generated by the change of state at the HSC input..

IF .. THEN .. ELSE .. ENDIF

Purpose : To make a decision regarding program flow based on the result returned by an expression.

Syntax :

IF *expression* **[THEN]**
  .....
[ELSE]
  .....
ENDIF

If the result of the expression is non-zero (logical true), the block of program lines between the **THEN** and the **ELSE** statements will be executed. If the result of the expression is zero (false), the block between the **IF** and **ELSE** will be ignored, and the block between the **ELSE** and **ENDIF** statements will be executed instead.

If there is no ELSE statement, and if the result of the expression is false, the block of program lines between the THEN and the ENDIF statement will be ignored, but execution will continue right after the ENDIF statement.

Nesting of IF statement

Statement blocks within the IF..THEN..ELSE statement may contain other IF..THEN..ELSE blocks (nesting). Note that each IF statement must be ended with the

ENDIF statement. Otherwise an error message "IF without ENDIF" will be reported during compilation.

Testing Equality: Special comparison operators may be used in the expression of the IF statement. Only integer expression may be compared. For comparison of strings, please refer to the "STRCMP(A$, B$)" function.

Equal	=
Not Equal	< >
Greater than	>
Less than	<
Greater than or Equal to	> =
Less than or Equal to	< =

Examples :
```
IF A >= B*5-20*C OR C=20
    B = B-1
ELSE
    B = B*3
ENDIF
```

Comments : A few comparison expressions may be linked with logical-AND (AND statement) or logical-OR (OR statement) operator as shown in the above examples.

INCOMM(*ch*)

Purpose : To return a single 8-bit binary data obtained from comm. channel *ch*.

ch must be a numeric constant between 1 and 8. The actual target hardware determines the valid port #. This function returns -1 if there is no data waiting at serial port.

Example :
```
FOR I=1 to 100
   DM[I] = INCOMM(2):
   IF DM[I]<0 RETURN :ENDIF
NEXT
```

Comments : Usually the PLC buffers the serial data arriving at its COMM port so that the program does not need to continuously check the COMM port for data. When the program is ready to process the data it can use the FOR..NEXT loop shown in the above example to read in all the data in the COMM buffer until it encounters a -1, which indicates that the buffer is empty.

Note: **INCOMM is now supported on all COMM ports of T100MD1616+ and T100MX+ families of PLCs.**

See Also : **OUTCOMM, INPUT$( ), PRINT #**

INPUT$(*ch*)

Purpose : To return a string obtained from communication port *ch*.
ch must be a numeric constant between 1 and 8. The actual target hardware determines the valid port #. This function returns f0 if there is no valid string waiting at serial port.

Example :
```
D$ = INPUT$(2)
```

Comments : A Carriage Return (CR) or ASCII code 13 marks the end of the input string from the communication port. The returned string however will exclude the CR character. In TRiLOGI simulator, the user will be prompted to enter the string in a pop-up window.

See Also : **INCOMM( ), PRINT #, OUTCOMM**

INTRDEF *ch, fn_num, edge*

Purpose : Enable Interrupt Input channel *ch*.
ch = channel number (1-8)
fn_num = Custom Function number to execute when interrupt pin changes according to the defined edge. This is the Interrupt Service Routine ISR.
edge = Positive number means rising edge-triggered. 0 or negative number means falling-edge triggered.

See Also : **INTROFF**

INTROFF *ch*

Purpose: Disable Interrupt Input channel *ch*.

See Also : **INTRDEF**

LEN(*x$*)

Purpose : To return the number of characters in *x$*.
Examples :
```
L = LEN("This is a test string"+CHR$(13))
```

Comments : L = 22 because blanks and non-printing characters are counted.

LET

Purpose : To assign the value of an expression to a variable

Syntax : **[LET]** *variable = expression*

Examples :
```
LET D = 11
A$ = "Welcome to TBASIC"
```

Comments : *LET statement is optional: i.e. the equal sign is sufficient when assigning an expression to a variable name. The variable type on both sides of the equal side must be the same. i.e. string variable may not be assigned to a numeric expression and vice-versa.*

Important : a) When assigning a 16-bit variable to a 32-bit integer, only the lower 16 bits of the 32-bit integer will be assigned. Hence the programmer must take special care if the 32-bit number is out of the range of a 16-bit number (which is between -32768 to 32767).

b) If a negative 16-bit number is assigned to 32-bit integer variable, then the sign bit will be extended to 32 bits.

```
e.g.  DM[1] = -123.
      A = DM[1]
```

The 16-bit hexadecimal value of -123 is &HFF85, but A will be assigned the hexadecimal value &HFFFFFF85. Their decimal representation are however the same.

LOAD_EEP(*addr*)

Purpose : To return a 16-bit integer value saved in the EEPROM by the **SAVE_EEP** statement.

addr - EEPROM address (1-2000) in TRiLOGI. Actual PLC may have less EEPROM space. Please refer to your PLC's reference manual for the upper limit.

Examples : `relay[1] = LOAD_EEP(10):     A = LOAD_EEP(2)`

See Also : **SAVE_EEP**

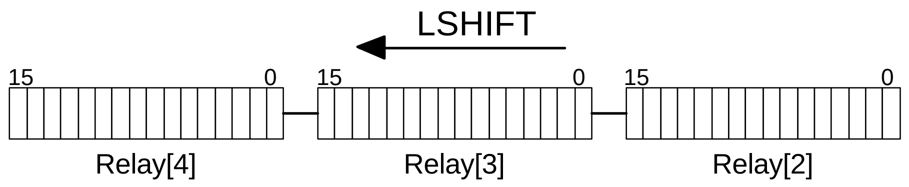
LSHIFT *i*, *n*

Purpose : To shift 1 bit to the left the integer variable *i* which must be either an integer variable, a DM[n] or a system variable such as relay[n], output[n], etc.

LSHIFT instruction permits more than one variable to be chained together before performing a bit shift. The parameter *n* indicates the number of channels to be chained starting from *i* upward. *n* =1 if only one variable is involved.

Examples : `LSHIFT relay[2],3`

Comments : *The relay channels #2,#3, and #4 (which represent relays number #17 to #64) are chained together in the following manner:*



Bits are shifted from the lower channel towards the upper channel. Bit #15 of Relay[2] will be shifted into Bit #0 of Relay[3] and so on. Bit #15 of the highest channel Relay[4] will be lost.

See Also : **RSHIFT**

MID$(*x$, n, m*)

Purpose : This function returns a sub-string of *m* characters from *x$*, beginning with the *n*th character.
x$ - any string expression, variable or constant.
n - any numeric expression producing a result of between 1 to 255
m - any numeric expression producing a result of between 0 to 255.

Examples :
```
A$ = MID$("Welcome to TBASIC",4,7)
```

Comments : *A$ should contain the string :"come to".*

NETCMD$(*ch*, *x$*)

Purpose : This function sends a multi-point host link command string specified in the *x$* via serial port *#ch* to another M-series or H-series PLC. It will then wait for a specified amount of time for a response string from the other PLC and this response string is then returned.

ch - This refer to the communication port #. Please refer to the target PLC for details.

*x$* - contains a valid host link command in multi-point format, <u>excluding the Frame Check Sequence (FCS) and the terminator characters (* and CR)</u>. NETCMD$ function will automatically compute the FCS and append to the end of *x$* and together with the terminator characters will be sent to the other PLC via COMM *#ch*.

Note:
1) If the target PLC does not respond then this function returns an empty string.
2) This function checks the FCS of the response string, and if the FCS is wrong it indicates an error in the serial reception and it will return an empty string.

Examples : `A$ = NETCMD$(3, "@05RI00")`

Comments : *To read the Input channel #0 of the PLC with ID = 05 connected to COMM #3 of this PLC. The response string will be assigned to A$.*

Special **:** If the last character of *x$* is a "~" character, NETCMD$ will send out the string without the '~' character, followed by a Carriage Return (&H0D). It will not append the FCS and '*' to the outgoing string, it will also NOT check the response string for FCS. This allow NETCMD$ to be used to interface to third-party ASCII devices with different command/response formats. E.g. A$ = NETCMD$(3, "Hello World~"). The string "Hello World" will be sent out of serial COMM port #3. A$ will receive the full returned string without applying any FCS check on the return string.

OUTCOMM *n, x*

Purpose : This statement can be used to send an 8-bit byte of data ' *x* ' via Comm port #*n*. This command is added because PRINT #*n* command cannot be used to send out CHR$(0). Zero is treated as the end of a string in TBASIC and will be ignored if you use PRINT #n statement to send out CHR$(0).

Examples : `OUTCOMM 2,225`

PAUSE

Purpose: To set a breakpoint for executing the CusFn. This is used mainly for debugging a CusFn. By Inserting a **PAUSE** statement at the place of interest, you can suspend the program execution when **PAUSE** is encountered, after which you may examine the values of the relevant variables. You can continue to perform on-line monitoring of the PLC that has been paused. Program execution can also be continued by pressing the <P> key during Simulation or On-line Monitoring.

PIDdef *ch, lmt, P, I, D,*

Purpose: To set up the parameters for a Proportional, Integral and Derivative (PID) Controller function. The function **PIDcompute( )** will make use of the parameters defined here for the corresponding channel *ch*.

ch = channel number (1-16)
lmt = Maximum (saturation) limit for the computed result.
P = Proportional Gain (K_P)

I = Integral Gain (K_I)
D = Differential Gain (K_D)

Transfer Function of a PID Controller are defined as follow:

$$G(s) = K_P + \frac{K_I}{s} + K_D s$$

$$K_P = \text{Proportional Gain} = \frac{1}{\text{Porportional Band}}$$

$$K_I = \text{Integral Gain} = \frac{1}{\text{Integral Time Constant}}$$

All four parameters: *lmt, P, I & D* can be either 16 or 32-bit integer constants or integer variables. For the *lmt* term, the computed controller output value by the **PIDcompute( )** function is not allowed beyond the $\pm$ *lmt* value (i.e. *lmt* represents the saturation point of the computed controller output). **PIDcompute( )** function implements "Integrator anti-windup" feature, which will avoid integrating the error signal when output is already saturated .

Important: When this statement is run, the integral and differential terms of channel *ch* is set to zero. Hence **PIDdef** should be run only once during initialization and not repeatedly executed. Otherwise the **PIDcompute( )** function will not run properly because of the loss of integral and differential data.

See Also : **PIDcompute( )**

PIDcompute(*ch, E*)

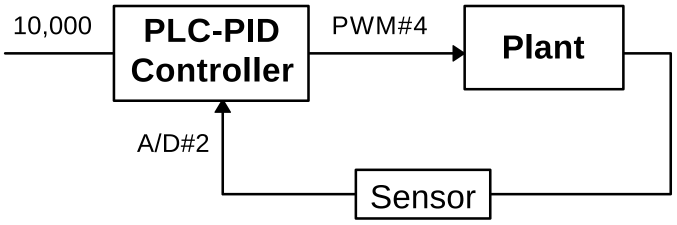
Purpose: This function computes the output for the PID compensator/ controller, using the *P,I,* and *D* Gains defined in the PIDdef statement for the same channel *ch*. The integral and differential values are stored within the channel's internal data space and will be automatically used by the PID computation routine. The PIDcompute( ) function uses the *lmt* (max. limit) term of **PIDdef** statement to limit the results of its computation. If the absolute value of the computed result is greater then "*lmt*", then the result will be set equal to "*lmt*" for +ve number and to "-*lmt*" for negative value. When this happens, the integral term will not accumulate the current error to prevent an "integrator windup" which is very undesirable for the system.

ch = channel number (1-16)

Err = Closed-loop Error.
(i.e. Set point value - Feedback Value)

The controller may obtain feedback from ADC, High Speed Counters, **PULSEFREQUENCY** or other means. The obtained result is then scaled and subtracted from the desired (set point) value to get "Err ". All computations are performed in 32-bit integers and the function returns a 32-bit integer which can be assigned to any variable. Any scaling for actual output (DAC or PWM) will be computed by the user within the same CusFn and sent to the output.

Example :



E.g. Implementing Closed-loop Digital Control with PID computation function

```
E = 10000 - ADC(2)*20
A = PIDcompute(5,E)
setPWM 4, (A + 8000)/100
```

Comments: The set point value is 10000 units, the feedback value is read from ADC channel #2 and then multiplied by 20 to convert (scale) it to the same unit as the parameter to be controlled. PID computation channel #5 (assume somewhere in the program a PIDdef *for channel #5 has been executed before) is then used to compute the desired controller output value using the error signal (= set point - feedback value ADC(2) x 20).*

The desired output (stored in variable A) is then added to the offset value 8000 and then scaled down by a factor of 100 before being sent out physically via PWM Channel #4.

Important: In actual implementation, use a clock pulse such as 0.1s, 0.5s or 1s etc to periodically activate the **PIDcompute( )** function so that digital control in discrete-time can be implemented. The PID sampling period depends on the time constant of the system. For very slow response processes such as the cooking temperature of a large body of water, the time constant is very large and even slower than 1.0 seconds clock may be sufficient. Do not use unnecessarily short sampling time because it increases computation time and slows down overall performance of the system.

PRINT# *n x$; y; z*.... Statement

Purpose : To send a string of ASCII characters formed by its parameter list (*x$; y; z*) out of the PLC to other devices via the communication channel *#n*.

Parameters: *n* must be an integer constant of between 1 and 8. Integer value in the parameter list (*y; z..*) will be converted into the equivalent ASCII representation. Each parameter must be separated by the semicolon(;).

Action : The ASCII string is first formed by the PRINT statement using all the arguments in the argument list and the completed string is then sent out of the serial channel #n at one go. The PRINT statement automatically sends a Carriage Return (CR-ASCII 13) out of the specified serial port after sending out the last character in the argument list. the PRINT statement that ends with a semi-colon “;", will not send the CR character.

If you have a long string to send than you can use ";" to break the whole command into several lines, with each line ending with a ";" except the last lines.

Examples :

```
PRINT #2 "The value of A+B = ";A+B;
PRINT #2 "Units"
```

Comments : IF A=5 and B=100, the string "The value of A+B = 105 Units" and a CR character will be sent out via the comm. port #2. In TRiLOGI simulation mode, the ASCII string will be displayed on a pop-up window to simulate PRINT action.

See Also : **INPUT$( )**

PMON *ch*
PMOFF *ch*

Purpose: **PMON** enables Pulse Measurement Function at channel *#ch*, whereas **PMOFF** disables the channel. After enabling the channel, you may then use the functions **PULSEWIDTH(***ch***)** and **PULSEPERIOD(***ch*) to obtain the width and period of the input pulses arriving at the pulse measurement input pin. You must call PMON once during initialization to enable the pulse measurement hardware. Otherwise the two functions will only return 0. You should avoid repeatedly executing PMON function, otherwise the pulse measurement hardware will be reset repeatedly as well, and accurate measurement cannot be obtained.

If you no longer need to measure the pulse-width or period for a particular channel which has been PMON before, you should disable it using PMOFF to save CPU time because pulse measurement is interrupt driven and consumes CPU time.

Example: `PMON 1 : PMOFF 5`

See Also : **PULSEWIDTH( ), PUSEPERIOD( )**

PULSEFREQUENCY(*ch*)
PULSEPERIOD(*ch*)
PULSEWIDTH(*ch*)

Purpose: Return in Hz the frequency of the last input pulse; Return in microseconds the width and period of the input pulses arriving at channel *ch* of the pulse-measurement pin. The pulse-measurement channel *ch* must have been enabled by the **PMON** statement already. If the pulses stop coming in then PULSEFREQUENCY will return a zero while the other two functions will saturate at a certain maximum value (for T100MD+ it is equivalent to about 3.28 seconds)

ch = channel # (1-8)

Example: `A = PULSEWIDTH(1)`

See Also : **PMON, PMOFF**

---

* READMODBUS (*ch, DeviceID, address*) {* Applicable only to **M+** PLC models}

Purpose : Automatically query a MODBUS ASCII device and return the 16-bit register data using the MODBUS ASCII protocol. The communication baud rate is the default baud rate of that COMM unless it has been changed by the SETBAUD command.

ch	-	PLC COMM port number (1-8)
DeviceID	-	device ID of the MODBUS device (1 to 255)
address	-	zero-offset address of the holding register in the MODBUS device.

Example : `relay [3] =  READMODBUS (3, 5, 101)`

Comments : The relay will contain the 16-bit data obtained from the MODBUS device with ID = 05 and from register offset address 101 (in MODBUS term this refer to the #40102 holding register) . Reading it into the relay[ ] channel allows bit level manipulation by ladder logic. It can of course also be read into any data memory. The command automatically checks the response string received from the slave device for the correct LRC and the slave address. The status of the operation can be checked in the user program by executing the STATUS(2) function, **which will return a '0' if there is any error or if the slave device is not present.**

See Also : WRITEMODBUS, STATUS(2), NETCMD$( )

REFRESH

Purpose : To Force immediate refresh of the physical inputs and outputs. This can be used after executing a **SETBIT** or **CLRBIT** command on an output[n] variable and to force the physical output to change immediately (subject to I/O refresh time delay). Otherwise, the physical output will only be updated during the normal refresh cycle which will occur only at the end of every ladder logic scan.

This is useful for situations which require immediately action such as shutting down a load during an emergency. This command is likely to be used mainly by an Interrupt CusFn.

REM (or ') Statement

Purpose : To allow explanatory remarks to be inserted in a program. The text after the REM statement until the end of the line will be ignored by the compiler. An abbreviation for the REM statement is the apostrophe (')

Examples :
```
REM  Waiting for the right time to turn on
'  This is also a remark line.
```

RESET

Purpose : To perform a software reset of the PLC from within a CusFn. All the variables will be reset to zero or inactive and all the hardware outputs such as DAC and PWM will be turned OFF. The effect is the same as the Master Reset [MaRST] function in the ladder logic. The first scan bit (1st.Scan) will also be turned ON for one scan time.

However, if the program is stuck at some dead loop (such as WHILE, FOR-NEXT) in a CusFn, then [MaRST] would not be executed since the ladder program would not have a chance to scan the ladder rung containing the [MaRST] function. If this command is used by an interrupt service function, then it is possible to get the system out of the dead loop since the interrupt function can interrupt the dead loop and reset the PLC.

RETURN

Purpose : Unconditionally ends the execution of the current CusFn and return to the caller (which is either the ladder program or another CusFn which has executed a CALL command).

Use of the RETURN statement is optional if there is no conditional ending required. After executing the last statement the CusFn will return to the caller automatically.

See Also : **CALL**

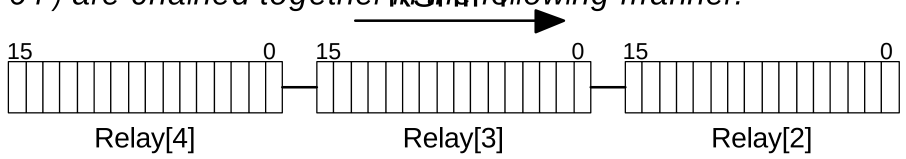
RSHIFT *i*,*n* Statement

Purpose : To shift the integer variable *i* 1 bit to the right. *i* must be either an integer variable, a DM[n] or a system variable such as relay[n], output[n], etc.

RSHIFT instruction permits more than one variable to be chained together before performing a bit shift. The parameter *n* indicate the number of channels to be chained starting from *i* upward. *n* =1 if only one variable is involved.

Examples : `RSHIFT relay[2],3`

Comments : The relay channels #2,#3, and #4 (which represent relays number #17 to #64) are chained together in the following manner:



Bits are shifted from the upper channel towards the lower channel. Bit #0 of Relay[4] will be shifted into Bit #15 of Relay[3] and so on. Bit #0 of the lowest channel Relay[2] will be lost.

See Also : **LSHIFT**

SAVE_EEP *data, addr*

Purpose : To store a 16-bit integer *data* in the user's definable EEPROM address *addr* for non-volatile storage. If you attempt to save a 32-bit data, only the lower 16-bit will be saved. To save the entire 32-bit data, save the upper 16-bit using the **GETHIGH16( )** function and the lower 16-bit directly in two separate locations.

data - may be a 16-bit integer constant or variable.

addr - EEPROM address (1-2000 in TRiLOGI). Actual PLC may have less EEPROM space. Please refer to your PLC's reference manual for the upper limit.

Example : `save_EEP relay[1],100`

See Also : **LOAD_EEP( ), GETHIGH16( ), SETHIGH16**

SETBAUD *ch*, *baud_no*

Purpose : To set the communication "Baud Rate" of the PLC's serial channel #ch. All the M series PLC serial ports are defined as 8 data bit, 1 stop bit, and no parity and each has been preset to a certain default baud rate, which the PLC will assume every time its powers up. The baud rate may or may not be changed, depends on the PLC model. Please refer to the PLC's User's manual for the *baud_no* that represent the baud rate of each serial channel and the range of *baud_no* each of these serial ports may assume.

Caution should be taken when programming the baud rate of the "Host link" port because if a wrong baud value is set the host PC may not be able to communicate with it. If this happens suspend the PLC using its hardware switch and reset the PLC and re-load the program with correct setting.

Examples : `SETBAUD 3,3   ' Set serial port #3 to 9600.`

SETBIT *v*,*n*

Purpose : To set the bit #*n* of the integer variable *v* to '1'. *n* is an integer constant or variable of value between 0 and 15. *v* may be any integer variable or a system variable such as relay[n], output[n], etc. However, if *v* is a 32-bit integer, **SETBIT** will only operate on the lower 16 bits.

Following digital electronics convention, bit 0 refers to the least significant bit (rightmost bit) and bit 15 the most significant bit. (leftmost bit) of the 16-bit integer variable. A quick way to find out the bit position and index of an I/O variable is to open their I/O table and check the "CH:BIT" column. Bit position beyond 9 are represented by hexadecimal number A to F.

Examples : `SETBIT output[2],11`

Comments : *output #28 will be turned ON.*
(Output channel #2 bit #11 = Output #17 +11 = 28)

See Also : **CLRBIT, TESTBIT( )**

SetCtrSV *n*, *value*
SetTimerSV *n*, *value*

Purpose : Change the **Set Value (S.V,)** of the Counter #*n* or Timer #n to *value*. This statement to allow the user to modify the S.V. of the PLC internal timers and counters without changing the source program. A TBASIC function can be written easily to make use of a few digital or analog inputs to modify the SV of these internals timers/counters. The new S.V is also stored in the program EEPROM and hence is non-volatile. (See sample program "set_TCSV.PC4")

n should be between 1 and 128.
value should be between 0 and 9999.

Examples :
```
SetCtrSV 10,1234
SetTimerSV 3, GetTimerSV(3)+10
```

Comments : *Counter #10 will assume a S.V. of 1234..*
: *S.V of Timer #3 will be increased by 10.*

Related : The present values (P.V.) of timers and counters can be read or written directly as integer variables "TimerPV[n]" & "CtrPV[n]". But the Set Values can only be changed by these two functions.

See Also : **GetCtrSV( ), GetTimerSV( )**

SETDAC *n*, *x* Statement

Purpose : To set channel #*n* of the PLC's Digital-to-Analog Converter (DAC) with the 16-bit integer result of the expression *x*. *n* must range between 1 and 16. Once set, the DAC channel will latch the set value until the next SETDAC statement on the same channel is executed.

Examples :
```
SETDAC 5,A+B*16
```

*Comments : DAC channel #5 will be set with the value of A+B*16. A run- time error will result if n is less than 1 or is greater than 16. The actual number of DAC channels depends on the PLC model in use.*

SETHIGH16 *v, data*

Purpose : To assign the upper 16-bit of a 32-bit integer variable *v* to *data*. The lower 16-bit of *v* is unaffected. This can be used to construct the value of a 32-bit integer data using two 16-bit data obtained from either the EEPROM or the DM[n].

Examples :
```
A = DM[2]
SETHIGH16 A,DM[1]
```

See Also : **GETHIGH16( )**

SETLED *n, m, value*

Purpose : To display the integer *value* on the PLC's built-in 7-segment LED displays, starting from the *nth* digit and occupying *m* number of digits. Leading zeros will be added to the left of the display if *value* occupies less digit than that specified by *m*.

However, if *m* is less than 1 (e.g. *m* = 0) then *value* is treated as a single 8-bit ASCII character to be displayed rather than as a numeric value. Special symbols may be displayed on the LED panel if the LED driver is able to display the corresponding ASCII character.

n must be between 1 to 16. The digit position is counted from left to right. i.e. the leftmost LED digit is digit #1. TRiLOGI supports up to 16 LED digits. The actual number of LED on the PLC may vary from 0 to 16, in this case only the available digits will be effective. *Value* may be a 16- or 32-bit integer number. Once set, the LED display will latch the set value until the next SETLED statement on the same digit is executed. On the TRiLOGI simulator, the result of the SETLED is displayed together with the Special Variables screen, which may be viewed by pressing the <V> key while in the simulation mode.

Examples : `SETLED 5,4,89`

Comments : LED digit #5 to #8 (counting from left to right) displays 0089.

SETLCD *n, offset, x$*

Purpose : To display the string expression *x$* on Line #*n* on built-in alphanumeric Liquid Crystal Display (LCD) or compatible Vacuum Fluorescent Display (VFD). *x$* may be formed by concatenation of various strings using the '+' operator (e.g. "Temp ="+STR$(A,3)+CHR$(223)+" C"). Integers must be converted to string using the STR$( ) or HEX$( ) function to be accepted by this function.

> ***Special case**: if n =0 the string x$ will be sent to the LCD's "Instruction-Register" which allows hardware-specific LCD configuration such as clear screen, set cursor ON/OFF etc. (please refer to LCD's manual for details)*

The parameter *offset* = 1 to 40 allows you to send the string *x$* beginning from the *offset*th position. Only the characters position to be occupied by *x$* will be written to the display, other characters of the display remain unaffected.

The PLC may support LCD display modules capable of displaying up to 4 lines x 40 characters per line of alphanumeric characters. If the display has fewer lines or fewer characters per line, the unavailable lines or characters will be ignored by the PLC. Once set, the LCD display will latch the set value until the next SETLCD statement on the same line and same offset is executed. On the TRiLOGI simulator, the result of the SETLCD is displayed together with the Special Variables view screen.

Examples : `SETLCD 1,1,"This is a 1x20 LCD Display"`

SETPWM *n*, *x*, *y*

Purpose : To set channel #*n* of the PLC's Pulse-Width Modulation (PWM) output with duty cycle represented by (*x*/100 %) and at a frequency (in Hz) given by parameter *y*.

n must range between 1 and 8. Once set, the PWM channel will latch the set value until the next **SETPWM** statement on the same channel is executed. *x* should range between 0 and 10000. If x is more than 10000, the duty cycle will be set to 100%

Examples : `SETPWM 1,4995,2000`

Comments : PWM channel #1 will be set to operate at 49.95% duty cycle for PWM that can resolve up to 0.01%. The actual resolution will depend on the PLC's PWM resolution. The PWM frequency is set to 2000 Hz or nearest.
For a 10-bit PWM the best resolution is about 1/1024 = 0.1 %. This means that in the above example the PWM will be rounded to 50%. Please check the target PLC's manual for the actual resolution.

---

* SETPASSWORD *string*

{* Applicable only to **M+** PLC models}

Purpose : When this statement is executed, the PLC will not properly respond to any host link commands sent to it except the command "PWxxxx…xx" which must contains the same string "xxxx…xx" (not more than 19 characters) as defined in the SETPASSWORD command. All other commands will receive a "PWER" response indicating a "password error" state. Once the correct password has been accepted the PLC will work as normal and respond to all host link commands. Execution of "PW" host link command without any string will put the password lock back in force to prevent unauthorized access.

Example : `SETPASSWORD “I love TRiLOGI”`

When using TL41.EXE the editor will automatically prompt you to enter the password string if it encounters a PLC which has been password-locked. Note that the password is case sensitive. Password locked PLC cannot be accessed by older version of TRiLOGI.

Comments : This feature is mainly used to protect an unattended PLC which is linked to an auto-answer modem. Without password protection anybody can dial in with a TL41.EXE and have full control of the PLC, which may be a serious security problem. When TL41.EXE disconnects the modem it automatically executes the "PW" command to re-arm the password lock so that there will not be unauthorized access by subsequent callers. Within the PLC software you may also use a timer to periodically re-arm the PLC with this command for maximum protection. You can also use different passwords for different time of the day or a set of rotating passwords to provide greater security.

STATUS (*n*)

Purpose : Return the status of various system operations.

Function	Returned value
STATUS (1)	0 - Normal power on reset 1 - Reset by Watch Dog Timer (WDT)
STATUS (2)	0 - READMODBUS or WRITEMODBUS failure 1 - READMODBUS or WRITEMODBUS successful
STATUS(8)	PLC's ID address stored in EEPROM for host communication

Examples :

```
IF STATUS(2)            ‘ MODBUS READ/WRITE OK
    …
ELSE                    ‘ MODBUS READ/WRITE failed
    …
ENDIF
```

STEPCOUNT (*ch*)

Purpose : While the stepper motor controller is sending out pulses, this function can be used to monitor the number of stepper pulses sent to the Stepper Motor Channel #*ch* since the execution of the last "STEPMOVE" command. Hence this function returns the relative number of step moves.

This function can also be used to "measure" the physical size of a part if we use the stepper motor to drive a sensor and use the STEPSTOP command and the interrupt input to halt the stepper motor when the edges are detected. The physical size is then computed using the number of steps the stepper motor travels from one edge to another edge. The center position can be easily determined using such data too.

---

*STEPCOUNTABS (*ch*)

{* Applicable only to **M+** PLC models}

Purpose : Returns the absolute position of the stepper motor #ch. This function returns a zero if a **STEPHOME** command had just been executed and the stepper has not been moved since.

---

* STEPHOME *ch*

{* Applicable only to **M+** PLC models}

Purpose : Set the current position counter of stepper # *ch* to zero. This indicates a new "Home" position of that stepper motor. This command should be executed only when the stepper has reached a particular position to be regarded as the home position. All STEPMOVEABS command executed subsequently will be relative to the defined home position.

STEPSPEED *ch*, *pps*, *acc*

Purpose : To set the speed *pps* and acceleration/retardation *acc* parameters for the PLC's stepper motor motion controller (pulse-generator) channel #*ch*.

ch should return a value of between 1 and 8. Speed *pps* is based on no. of pulse per second (pps) output by the pulse generator. The acceleration *acc* determines the total number of steps taken to reach full acceleration from standstill and the number of steps from full speed to a complete stop. The stepper motor calculates and performs the speed trajectory according to these parameters when the command STEPMOVE is executed.

STEPSPEED command should be executed at least once before executing any STEPMOVE command to control the pulse generation. The defined parameters will be remembered until another STEPSPEED statement operating on the same stepper channel is executed again.

Examples : `STEPSPEED 2,2000,20`

Comments : *The PLC's Stepper motor controller channel #2 is configured to send out pulses at 2000 pulses per second when STEPMOVE instruction is executed. It follows a linear acceleration trajectory which takes 20 pulses to reach the full speed of 2000 pps. This is equivalent to an acceleration of*

$$a = \frac{V^2}{2S} = \frac{2000^2}{2x20} = 100{,}000 \text{ pulse/s}^2$$

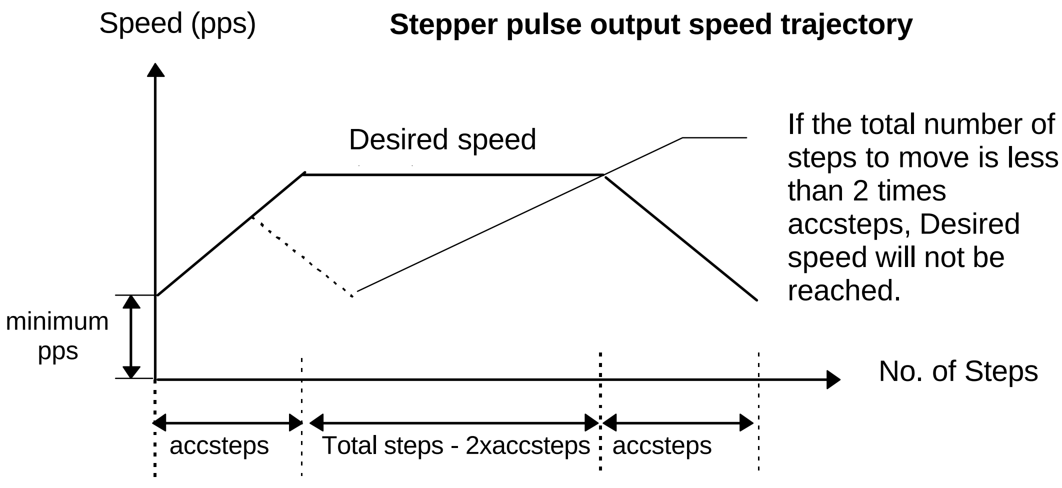
STEPMOVE *ch, count, r*

Purpose : To activate the PLC's built-in stepper motor pulse generator channel #*ch* to output *count* number of pulses. The speed and acceleration parameters for the motion is defined by the STEPSPEED statement on the same channel # *ch*, which must be executed at least once before the first STEPMOVE command is issued. After executing the STEPMOVE command the PLC hardware will take over the actual pulse generation operation. The user's program will continue to execute even though the pulse generation is not yet completed. The internal relay #*r* can be used to signal to the other parts of the ladder program regarding the status of pulse generation, as follow:

When STEPMOVE command is first executed, the internal relay #*r* will be cleared before the first pulse is sent. After the completion of the movement (i.e. when all the pulses have already been sent), the relay #*r* will be set.

ch should be between 1 and 8. *Count* is a 32-bit integer number which allows you to program the stepper motor to move from 1 to $+2^{31}$.(i.e. 2,147,483,647) steps. *Count* can also be an integer variable A-Z. However, If you use a 16-bit variable such as DM[n] for *count* then the range of movement can only be between 1 to 32,767.

Stepper pulse output speed trajectory


Speed (pps)
Desired speed
If the total number of steps to move is less than 2 times accsteps, Desired speed will not be reached.
minimum pps
No. of Steps
accsteps
Total steps - 2xaccsteps
accsteps

Pulse generation can be <u>interrupted</u> by issuing a **STEPSTOP** command in another CusFn, which may occur say, in cases when the hardware hits a limit-switch and must stop the motor immediately.

Important: When a stepper channel is already activated (i.e. mid-way through its pulse generation) repeat execution of STEPMOVE command on the same channel will be ignored by the PLC. Re-execution of the STEPMOVE command on this channel can only take effect after the channel's pulsing operation has been completed by itself or aborted by the STEPSTOP command.

When in TRiLOGI simulation mode, execution of the STEPMOVE command will bring up a pop-up window which displays all the parameters of the motion path.

Examples **:** `STEPMOVE 1,5000,10`

Comments : Send out 5000 pulses on channel 1 and at the end of motion turn ON relay #10.

See Also : **STEPMOVEABS, STEPCOUNT( ), STEPCOUNTABS( ), STEPSPEED, STEPSTOP, STEPHOME**

---

*STEPMOVEABS *ch, position, r* {* Applicable only to **M+** PLC models}

Purpose : This new command allows you to move the stepper motor # *ch* to an absolute position indicated by the *position* parameter. At the end of the move the relay # *r* will be turned ON. Position can be between -2^{31} to $+2^{31}$.(i.e. about $\pm 2 \times 10^9$). The absolute position is calculated with respect to the last move from the "HOME" position. (The HOME position is set when the STEPHOME command is executed). The speed and acceleration profile are determined by the STEPSPEED command as in the original command set.

This command automatically computes the number of pulses and direction required to move the stepper motor to the new position with respect to the current location. The current location can be determined at any time by the **STEPCOUNTABS( )** function.

Once STEPMOVEABS command is executed, re-execution of this command or the STEPMOVE command will have no effect until the entire motion is completed or aborted by the STEPSTOP command.

See Also : **STEPCOUNTABS, STEPHOME , STEPSPEED, STEPMOVE, STEPSTOP, STEPCOUNT**

STEPSTOP *ch*

Purpose : To abort a stepper channel #*ch* which is in motion due to exceptional circumstances.

Examples : `STEPSTOP 2`

Important : Motion aborted by STEPSTOP command will not trigger the end-motion relay #r specified in the STEPMOVE command.

See Also : **STEPCOUNT( ), STEPSPEED, STEPMOVE**

STR$(*n*)
*STR$ (*n, d*)

{* Applicable only to **M+** PLC models}

Purpose : To return a string that represents the decimal value of the numeric argument *n*. If the second format is used then this function will return a string of '*d*' number of characters.

Examples :
```
   A$ = STR$(-1234)
   B$ = STR$(-1234,7)
```

Comments : A$ will contain the string : "-1234" , B$ will contain the string "-001234"

STRCMP(*A$, B$*)

Purpose : Perform a comparison between its two string expressions A$ and B$. IF A$ and B$ are equals, **STRCMP** will return a 0, if A$ is of lower order (in ASCII table order) than B$ the function will return a negative value. Otherwise it returns a positive value.

Examples :
```
IF STRCMP$(A$, B$)=0 THEN
   STEPMOVE 1,1000,1
ENDIF
```

Comments : IF A$ and B$ are the same then turn on the stepper motor #1.

STRUPR$(*A$*)

Purpose : To return a string which is an all-uppercase copy of A$.

Examples : `B$ = STRUPR$(A$)`

```
C$ = STRUPR$(C$)
```

Comments : *The second example shows how to convert a string to upper case.*

STRLWR$(*A$*)

Purpose : To return a string which is an all-lowercase copy of A$.

Examples :
```
B$ = STRLWR$(A$)+Z$
C$ = STRLWR$(C$)
```

Comments : *The second example shows how to convert a string to all lower case.*

TESTBIT (*v*, *n*)

Purpose : To return the logic state of bit #*n* of the variable *v*. The function returns 1 if the bit is '1', otherwise it returns 0.

n is an integer of value between 0 and 15. *v* may be any integer variable, however, if *v* is a 32-bit integer **TESTBIT** will only test the lower significant 16 bits. A quick way to find out the bit position and index of an I/O variable is to open their I/O table and check the "CH:BIT" column. Bit position beyond 9 are represented by hexadecimal number A to F.

Examples :
```
TESTBIT (Input[2],3)
```

Comments : *To test whether input #20 is ON*
(Input channel #2 bit #3 = Input 17 +3 = 20)

See Also : **SETBIT, CLRBIT**

WHILE *expression* .... ENDWHILE

Purpose : To execute a series of statements in a loop as long as a given condition is true.

Syntax : **WHILE** *expression*

. . .

. . .

ENDWHILE

When **WHILE** statement is encountered, the expression will be evaluated and if the result is true, the statements following the expression will be executed until the **ENDWHILE** statement. Thereafter, execution branches back to the **WHILE** statement and the expression is evaluated again. The loop statements will be executed repeatedly until the expression becomes false.

Warning: Be careful that the **WHILE** loop will not be an endless loop as the PLC will appear to freeze up, being trapped in an endless-loop execution. TRiLOGI simulator attempts to detect this situation by giving a warning message if a loop is executed for an unduly large number of loops.

Examples :
```
WHILE S = 1
   IF INPUT[1] & &H0002: S = 0 : ENDWHILE
ENDWHILE
```

Comments : Execution will only be terminated when input #2 is ON. WHILE loops may be nested; i.e. a WHILE loop may be placed within the context of another WHILE loop. Each Loop must have a separate ENDWHILE statement to mark the end of the loop.

---

* WRITEMODBUS *ch, DeviceID, address, data*

{* Applicable only to **M+** PLC models}

Purpose : Automatically write the 16-bit *data* to a MODBUS ASCII device using the MODBUS ASCII protocol. The communication baud rate is the default baud rate of that COMM port unless it has been changed by the SETBAUD command.

ch	- PLC COMM port number (1-8)
DeviceID	- Device ID of the MODBUS device (1 to 255)
address	- Zero-offset address of the holding register in the MODBUS device.
data	- the 16-bit data to be written to the MODBUS device

Example :
```
WRITEMODBUS 3, 8, 1000, 1234
```

Comments : The data 1234 will be written to the MODBUS device with ID=08 at the holding register offset address 1000 (in MODBUS convention this refer to holding register #41001). The command automatically checks the response string received from the slave device for the correct LRC and the slave address. The status of the operation can be checked in the user program by executing the STATUS(2) function, **which will return a '0' if there is any error or if the slave device is not present.**

See Also : READMODBUS( ), STATUS(2), NETCMD$( )

VAL(*x$*)

Purpose : To return a value of a decimal number contained in the argument x$.

Examples : `B = VAL("123")*100`

Comments : B should contain the value 12300

Chapter 5 - Application Programming Examples

1. Important Notes to Programmers of TRiLOGI Version 4.1

a) Understanding Ladder Logic Execution Process

Like all industrial PLCs, the CPU of the M-series PLC first checks the logic states of the physical inputs and copies them into memory. During the ladder logic scan the actual logic states of the physical Inputs (except for interrupt inputs) are ignored by the PLC. The CPU uses the memory copy of the inputs to execute the ladder program.

The CPU executes its ladder logic program starting from the top rung of the program to the bottom rung. When the CPU reaches a ladder rung that activates a {CusFn} or {δCusF} that custom function will be executed. The CPU will only continue to scan the rest of the ladder program when the current custom function ends normally. Hence the order in which a ladder rung is placed within a ladder program can have an effect on the behavior of the program.

Output bits which are changed as a result of the program execution will only be updated to the **physical** outputs at the end of the ladder logic scan. One scan time is defined as the time it takes to execute the 3 steps (read physical inputs, execute program, update physical outputs). The CPU repeats these 3 steps continuously all the time, known as "Ladder Logic Scanning".

Hence, it is important to note that the variables INPUT[n] s and OUTPUT[n] in TBASIC are not the actual physical I/Os of the PLC, but only a memory representation of the actual I/Os which will be updated only during the I/O update cycles. The logic states of physical inputs are copied into the INPUT[n] variables during input scan and the physical outputs are set to the logic states contained in the OUTPUT[n] variables during output updates.

Therefore, one potential error that traditional BASIC programmers tend to commit is to attempt to poll for a change in the variable INPUT[n] within TBASIC such as the following:

✗

```
WHILE  INPUT[1] = 0
    ..
ENDWHILE
```

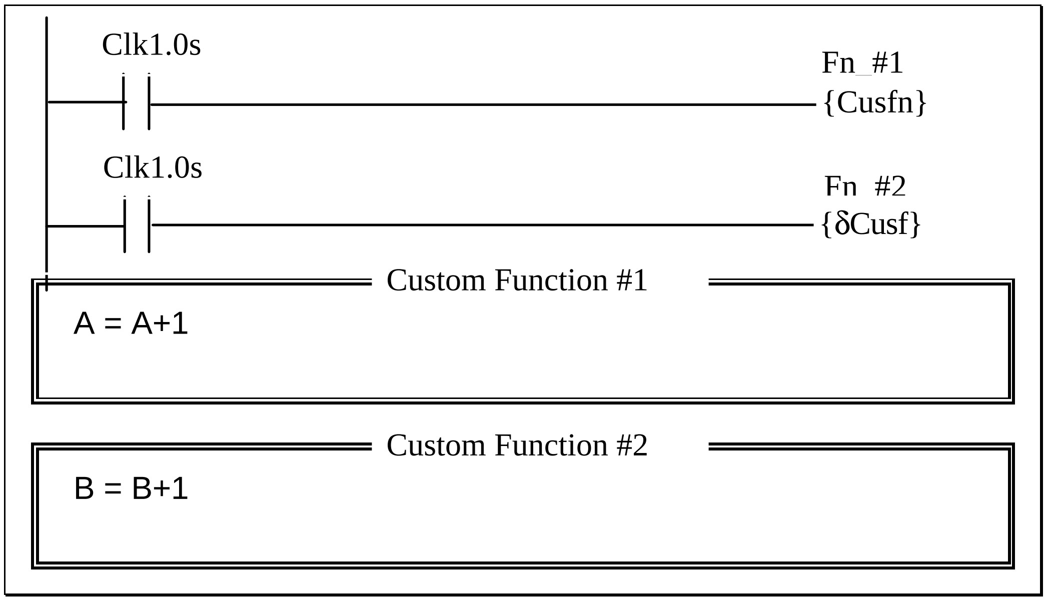
This will result in an endless loop since the value of the variable INPUT[1] will never change during execution of the custom function regardless of the actual logic states of physical input #1 to #8. The only way to force upon a physical I/O update is to use the REFRESH command, but it is not a good practice for ladder logic programming to update physical I/Os in the midst of a program execution. The REFRESH command is meant more for forcing an immediate output to be turned ON or OFF during time-critical situations.

Hence it is important to allow a ladder logic program to finish its scan so that the physical I/Os can be updated. You should never hog the CPU within a particular custom function as this will mean the rest of the ladder program don't have a chance to be executed in a timely manner.

b) The Difference Between {CusFn} and {dCusF}

It is very important to understand the difference between the two formats of the custom functions once you understand how the ladder logic scanning process works as described in the last section. If you use the **{CusFn}**, the custom function will be executed **EVERY SCAN** of the ladder logic program as long as its execution condition is ON.

On the other hand, the **{dCusF}** (known as the differentiated format) is executed only **ONCE** when its execution condition goes from OFF to ON. The execution condition must go OFF and then ON again for the function to be executed again. It is not difficult to see that the differentiated format is used far more frequently than the other one since most custom functions involve arithmetic and when a condition is ON you most likely want the computation to be performed ONCE and not repeatedly in every scan of the ladder logic. You can easily understand the difference between the two formats if you run the following sample program:



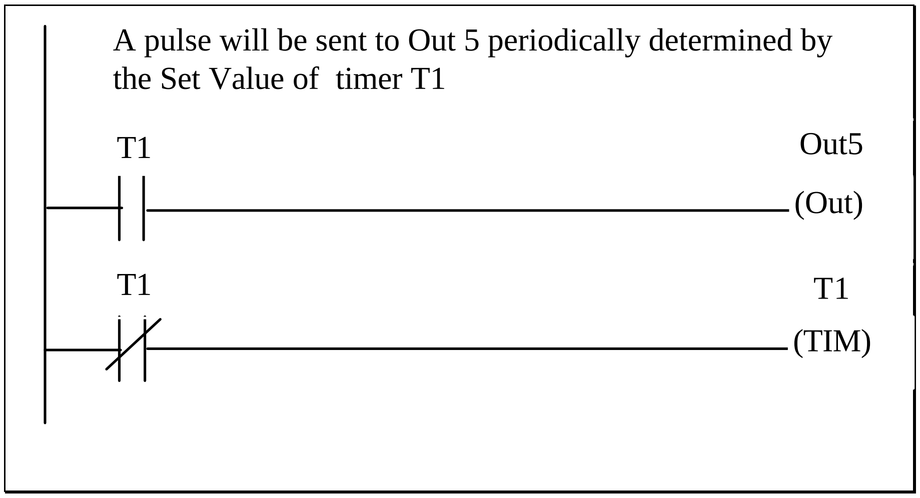
Run the program in simulator and press the <V> key to view the changes in the variables A and B. You will see that B is incremented by one every second, while A is incremented wildly for 0.5s and then stops for 0.5s. Try it! It can be very educational!

If you want to periodically check the status of an analog input or the real time clock, you should use a clock pulse (0.1s, 1.0s etc as shown in the example) and connect to a **{dCusF}.** Connecting to non-differentiated version would mean checking thousands of times for half the period and not at all for the other half period -- certainly not the intended outcome.

c) Timers Contact Updating Process

All the timers' contacts of the PLC, like the inputs and outputs, are updated simultaneously at the beginning of every ladder logic scan and not at the rung which contains the (TIM) coil. So if you are using self-reset timer, please note that if a timer time out its contact will be ON from the beginning of the ladder logic rung until the rung that contains the self-reset circuit. Thereafter the timer contact will be OPEN since the coil has been self-reset.

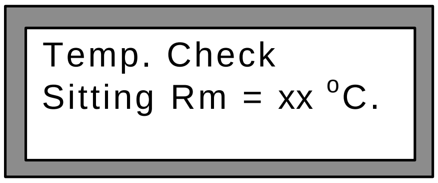
Hence please note that you should place the self-reset timer rung after all the ladder rungs that utilize the said timer contact. This allows those ladder rungs which use the timer contact to have a chance of being executed before the self-resetting rung clears the timer.


A pulse will be sent to Out 5 periodically determined by the Set Value of timer T1
T1
Out5
(Out)
T1
T1
(TIM)

2. Display Alphanumeric Messages on built-in LCD Display

M-series PLC such as the T100MD-1616 supports built-in LCD display port which allows low cost connection to industry standard LCD display module. For such PLC programming of the LCD display is via the SETLCD statement supported by TBASIC language.

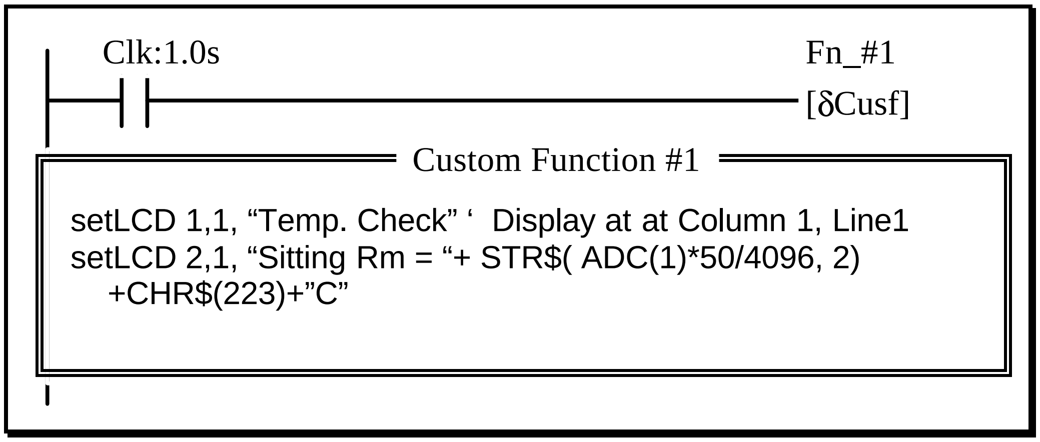
Assignment:
Every 1 second, display a message as follow:



Temp. Check
Sitting Rm = xx ^{o}C.

Where xx depends on reading of A/D #1 which is returned by function `ADC(1)`.

Full scale A/D is 4096.
A/D range (0 to 4096) ⇒Temperature 0 to 50^{o}C



```
setLCD 1,1, "Temp. Check" ‘  Display at at Column 1, Line1
setLCD 2,1, "Sitting Rm = "+ STR$( ADC(1)*50/4096, 2)
   +CHR$(223)+"C"
```

Comments:

Every one second, the special bit Clk:1.0s closes and activates Function #1. Within the Custom Function #1, `ADC(1)` *reads the A/D converter #1 and converts it into degrees. The integer value is then converted into a two-digit string using the* `STR$` *function and concatenated to the rest of the text string for display using the* `SETLCD` *command.*

Simulation of the display string to built-in LCD is supported on TRiLOGI Version 4.0x and above. When in Simulation mode, press <V> key to view the Special Variables and the messages will appear in an LCD Simulation window.

3. Display Alphanumeric Messages on Serial LCD Display MDS100

MDS100 is a 4 line x 20 characters LCD display connected to the PLC's RS485 port (serial port Comm#3).

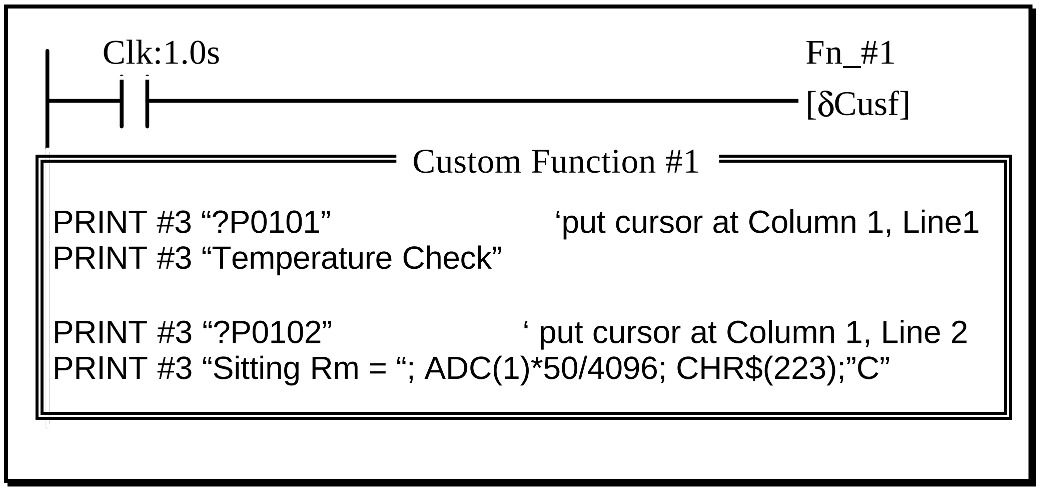
Assignment:

Every 1 second, display a message as follow:

```
Temperature Check
Sitting Rm =  xx °C.
```

Where xx depends on reading of A/D #1 that is returned by function `ADC(1)`.

Full scale A/D is 4096.
A/D range (0 to 4096) ⇒Temperature 0 to 50°C



```
PRINT #3 "?P0101"                  'put cursor at Column 1, Line1
PRINT #3 "Temperature Check"

PRINT #3 "?P0102"                ' put cursor at Column 1, Line 2
PRINT #3 "Sitting Rm = "; ADC(1)*50/4096; CHR$(223);"C"
```

Comments:

Every one second, the special bit Clk:1.0s closes and activates Function #1. Within the Custom Function #1, `ADC(1)` *reads the A/D converter #1 and converts into degrees.* `PRINT #3` *displays the string. The statement* `PRINT #3 "?Pxxyy"` *is a command to put the cursor at column xx, row yy of the display.*

Simulation of the display string to MDS100 is supported on TRiLOGI Version 4.03 and above. When in Simulation mode, press <V> key to view the Special Variables and the messages will appear in a MDS100 Simulation window. You can select whether data sent to Comm3 is meant for the MDS100 or generic RS485 device by the "MDS100 Simulation" option under the "Simulate" pull-down menu.

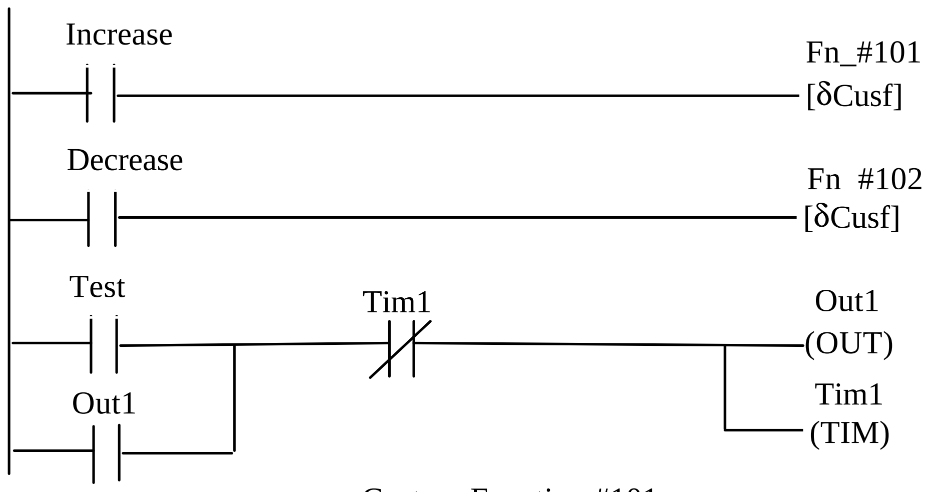
4. Setting Timer/Counter Set Values (S.V.) Using LCD Display

If you have an LCD display, then you can use two push-buttons inputs to change the Set Values (SV) of any selected timers or counters with visual feedback.

Assignment:

- Press push-button "Increase" increment the SV of timer #1 by 0.5s. The upper limit for timer #1 SV is 10s (SV <=100)
- Press push-button "Decrease" decrement the SV of timer #1 by 0.5s

- Press “test” button turns ON output #1 for a duration given by timer #1 and then turns it OFF.



Custom Function #101

```
Z = getTimerSV(1)
IF Z > 100 RETURN: ENDIF      ‘ MAXIMUM 10s
setTimerSV  1, Z+5    ‘Increase the current SV by 5 (0.5s)
SETLCD 1,1,”T1-SV=”+STR$(getTimerSV(1),4)
```

Custom Function #102

```
setTimerSV  1, getTimerSV(1)-5    ‘Decrease the current SV by 5
SETLCD 1,1,”T1-SV=”+STR$(getTimerSV(1),4)
```

Comments:

The **`getTimerSV(1)`** *function returns the current set value of the Timer #1. This value is read into variable Z in CusFn #101 but used directly in CusFn #102 for changing the Set Value of Timer #1. The* **`setTimerSV`** *statement uses the value of its second argument to update Timer #1’s SV accordingly.*

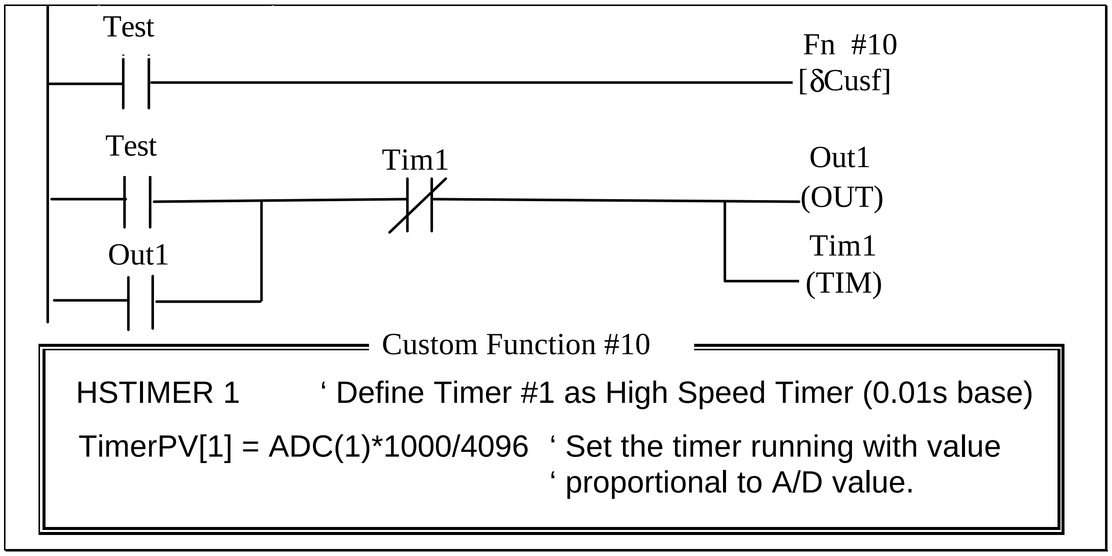
Note that changes to the set value SV will be updated in the program EEPROM memory and is non-volatile. However, EEPROM has a typical life-span of about 100,000 to 1,000,000 erase-write cycle. Exceeding this limit will “wear out” the EEPROM and resulting in a read error when the PLC operates. Hence, you should ***NEVER*** *write a program which excessively changes the set value of the timer or counter (e.g. put it in a non-differentiated form of [CusFn] which executes every scan of the ladder program and continuously changes the content of the EEPROM).*

5. Using a Potentiometer As An Analog Timer

A cheap potentiometer can be connected to the PLC A/D input and provide a user-adjustable "knob" as an analog Set-point input device. A scale can be drawn around the potentiometer to provide visual indication of set point value.

Assignment:

- A potentiometer is connected to A/D #5. Use it to provide a timing range of 0 to 10.00 seconds.
- Pressing the "test" input turns ON output #1 for a duration determined by the potentiometer reading, after that turns output #1 OFF.



```
HSTIMER 1          ' Define Timer #1 as High Speed Timer (0.01s base)
TimerPV[1] = ADC(1)*1000/4096  ' Set the timer running with value
                               ' proportional to A/D value.
```

Comments:

To take full advantage of the resolution of the A/D converter, the timing range of 0-10 seconds is more finely divided when timer is defined as high-speed timer using the `HSTIMER` *command. The time base is now 0.01s. This means that for maximum value of 10.00s, the timer should count down from 1000.*

*The next statement in CusFn #10 computes the ratio of the A/D input with respective to its full scale value of 4096 and multiplies it to the maximum timing value of 1000. I.e., if the potentiometer wiper is at half way, the A/D reading will be around 2048, the computation will results in a timing value = 2048*1000/4096 = 500, or 5.00 second. Note that TRiLOGI does not support floating point arithmetic, hence <u>the multiplication must be carried out before the division</u>. Otherwise, if you compute 2048/4096 *1000, the result of the integer division of 2048/4096 = 0 and the whole expression yields a '0', which is clearly wrong!*

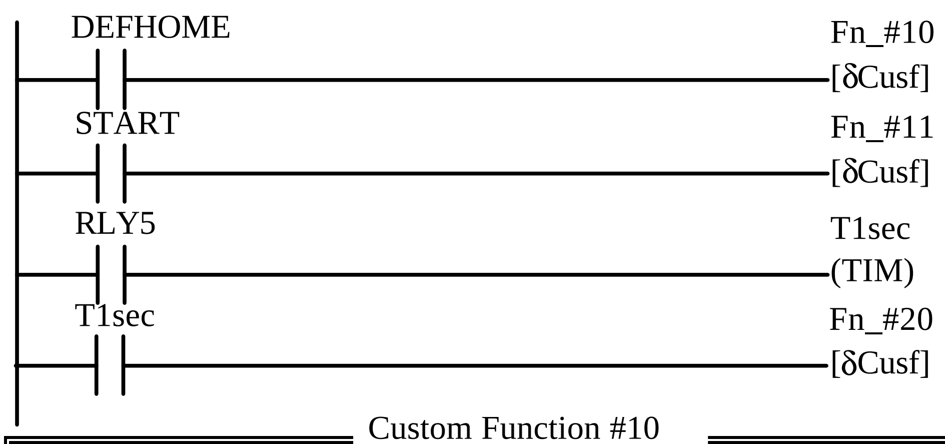
*The timer #1's Present Value (P.V) register is loaded with this number, which will start the timer count-down. In the next logic rung, the timer coil connected to the latched "*OUT1*" is necessary to prevent the timer from resetting itself. But It will not overwrite the PV with its own Set Value (SV), which will not be used at all in this case. This is because the previous ladder program has already started the timer with a value determined by the position of the potentiometer "knob".*

6. Motion Control of Stepper Motor

The M-series PLC can generate pulses to feed to stepper motor driver. The maximum speed, acceleration, deceleration and total number of pulses to generate are definable using TBASIC. Both absolute positioning commands and relative move commands are supported.

Assignment:

- A "DEFHOME" input define the current location as home position.
- Press the "START" input begin Indexing the stepper motor to position at 1500, -2000, 4500 and 9000 steps with respect to the HOME position. Pause for 1 seconds at each position. Return to home at the end of the cycle.
- Maximum speed = 5000 pps, Acceleration=100 steps to full speed.



Custom Function #10

```
STEPHOME(1)                ‘Define the HOME position for stepper 1
```

Custom Function #11

```
DM[1] = 1500: DM[2]= -2000: DM[3]=4500   ‘Store index position
DM[4]=9000: DM[5]=0
N = 1
STEPSPEED 1, 5000,100     ‘Stepper1: Max 5000pps, Acc:100
STEPMOVEABS  1, DM[N], 5    ‘ Move to position stored in DM[1]
                            ‘ at the end, turns ON relay 5
```

Custom Function #20

```
N = N+1
IF N <= 5
   STEPMOVEABS  1, DM[N], 5    ‘ Move to next position in DM[N]
ENDIF                          ‘ at the end, turns ON relay 5
```

Comments:

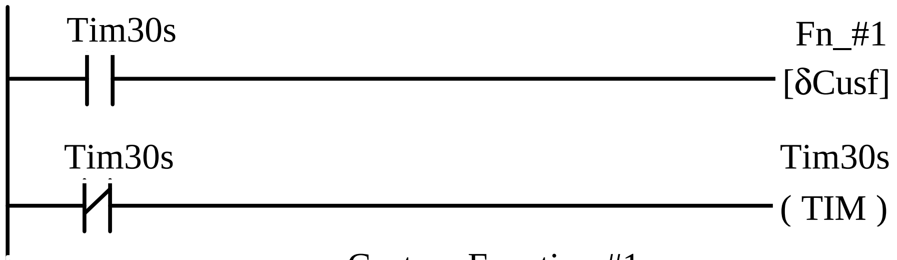
RLY5 *is the label for internal relay #5.* T1sec *is a timer with preset value of 10. At the end of the pulse generation,* RLY5 *will be activated. Ladder logic senses* RLY5 *and executes the* T1sec *timer to cause a 1 second delay, after which custom function #20 is executed which triggers another* STEPMOVEABS *command and the process repeats for the other four indexing positions.*

7. Activate Events at Scheduled Date and Time

All M-series PLCs have built-in Real Time Clock which keeps track of Date and Time and can be used to activate events at scheduled time.

Assignment:

- Every day turn on output #1 at 19:00.
- Turn OFF output #1 at 7:00
- On 1st Jan 2000 at 12:00 turn ON output #5
- On the same day at 18:00 turn OFF output #5



Custom Function #1

```
IF TIME[1]=19 AND TIME[2]=0          ‘ Hour hand at 19
   SETBIT OUTPUT[1],0                ‘ Minute hand at 00
ELSE IF TIME[1]=7 AND TIME[2]=0
   CLRBIT OUTPUT[1],0
ENDIF

IF DATE[1]=2000 AND DATE[2]=1         ‘ Jan, year 2000
  IF DATE[3]=1
    IF TIME[1]=12 SETBIT OUTPUT[1],4: ENDIF
    ELSE IF TIME[1]=18 CLRBIT OUTPUT[1],4: ENDIF
  ENDIF
ENDIF
```

Comments:

1. *Tim30s should have a Set Value = 300 and it activates Function #1 every 30 seconds. It is not necessary to check the clock too often as checking consume CPU execution cycles.*
2. *Output #1 is bit #0 of the variable output[1]. The statement* SETBIT *output[1],0 turns ON output #1. You can find the channel and bit-position number on the 3rd column of the Output definition table in TRiLOGI Ver. 4.03 and above.*
3. *Actually it may not be necessary to check the minute hand since when the RTC turns from 18:59 to 19:00, the output will be turned ON as long as TIME[1]=19. Only when TIME[1]=7, then output #1 needs to be changed.*

8. HVAC (Heating, Ventilation and Air-Conditioning) Control

Assignment:

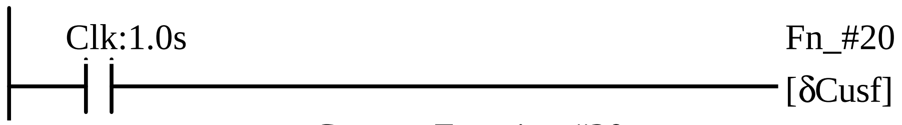
- Read desired temperature setting (S) from a potentiometer connected to A/D #5.
- Read current air temperature (T) from sensor attached to A/D #1 (T)
- Turn ON cold air-conditioner (output #1) if T > S by more than 1.5 °C.

- Turn ON hot air-conditioner if (output #2)
  if S > T by more than 1.5 ^{o}C.
- Turn OFF both hot and cold air-conditioner if T is within $\pm$ 1.5 ^{o}C of S.
- Display both Set Point and Actual Temperature.

Parameters

Full scale A/D is 4096.

Range of Set Point:	A/D #5 = 0	$\Rightarrow$	16.0 ^{o}C
	A/D #5 = 4096	$\Rightarrow$	30.0 ^{o}C
Range of Sensor:	ADC#1 = 0	$\Rightarrow$	-10.0 ^{o}C
	ADC#1 = 4096	$\Rightarrow$	50.0 ^{o}C



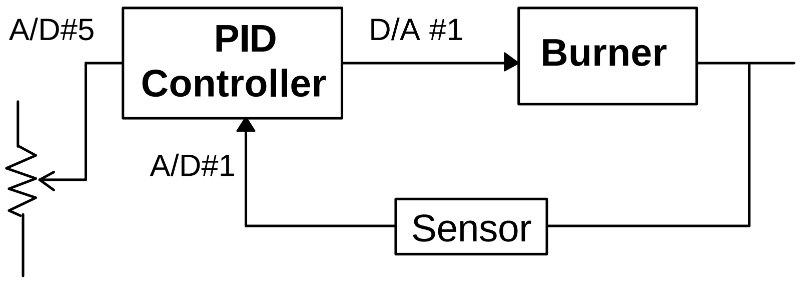
Custom Function #20

```
S = ADC(5)*(300-160)/4096 +160    'Convert to °C x 10
T = ADC(1)*(500+100)/4096 -100    'Convert to °C x 10

IF S -T > 15  : SETBIT OUTPUT[1],0      'Cold Airconditioning ON
ELSE  CLRBIT OUTPUT[1],0 : ENDIF        ' if T is hotter by 1.5 °C

IF S-T < -15 SETBIT OUTPUT[1],1          'Heater ON
ELSE CLRBIT OUTPUT[1],1: ENDIF           ' if T is colder by 1.5 °C

PRINT #3  "?P0101" : PRINT #3  "Set Point :"; S/10; " C"
PRINT #3  "?P0102" : PRINT #3  "Actual :"; T/10; "  C"
```

Comments:

Since TRiLOGI Version 4.x does not support floating point computation, in order to handle decimal value ($\pm$1.5^{o} C) in this application we use a unit integer to represent 0.1 quantity. All temperature readings are x10 times. Hence 16.0^{o}C is represented by 160, -10.0^{o}C is represented by -100. This method, known as fixed-point computation is quite commonly used in industrial control program..

9. Closed-Loop PID Control of Heating Process



E.g. Implementing Closed-loop Digital Control with PID computation function

<u>PID Controller Transfer Function</u>:

$$G(s) = K_P + \frac{K_I}{s} + K_D\, s$$

$$K_P = \text{Proportional Gain} = \frac{1}{\text{Proportional Band}}$$

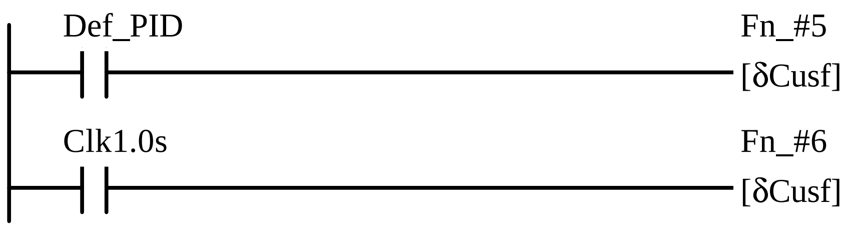
$$K_I = \text{Integral Gain} = \frac{1}{\text{Integral Time Constant}}$$

Assignment:

- Read desired set-point temperature from a potentiometer connected to A/D #5 (S) with temperature range between 50 °C - 200 °C
- Measure the process temperature from a thermocouple + signal conditioner attached to A/D #1(T)
- Compute the Error = S - T. Apply Proportional + Integral + Derivative (P.I.D) algorithm to compute output X.
- Apply output X to Digital-to-Analog converter D/A #1 to control a variable position valve that feed fuel to the flame.
- Sample and compute every 1 second.

<u>Full scale A/D range is 4096.</u>

Range of Set Point:	A/D #5 = 0 $\Rightarrow$ 50 °C
	A/D #5 =4096 $\Rightarrow$ 200 °C
Range of Sensor:	ADC#1 =0 $\Rightarrow$ 0 °C
	ADC#1 = 4096 $\Rightarrow$ 300 °C



Custom Function

```
P = 500: I = 50: D = 0
PIDDEF 1, 2048*100 ,P,I,D  ‘ Use PID Engine #1, maximum limit
                           ‘  = +/- 50% of full scale
```

Custom Function #6

```
S = ADC(5) * (200-50)/4096 + 50    ‘Convert to °C
T = ADC(1) * (300 - 0)/4096

X = PIDcompute(1, S - T)/100 + 2048  ‘ X can vary within ± 50%
setDAC 1, X                          ‘ Write to analog D/A output #1
```

Comments:

1. *We use two decimal places to represent the gains K_P, K_I and $K_{D.}$ Each integer unit represents 0.01. Proportional gain $K_P = 5$ is represented by variable P = 500. Likewise, Integral gains $K_I = 0.5$ is represented by I = 50 and Differential gains = 0 means Differential term is not used (P.I. only). The integrator limits of ± 2048 for the PIDDEF statement must be multiplied by 100 to be put on the same scale as the P,I and D parameters.*

   *Note that since TRiLOGI does not support floating point arithmetic, the multiplication must be carried out before the division. Otherwise, if you compute 150/4096 *ADC(5), the result of the integer division of 150/4096 = 0 and the whole expression yields a ‘0’, which is clearly wrong!*

2. *The value returned by* `PIDcompute()` *function is then divided by 100 to get the real value of controller output.* `PIDcompute()` *returns a signed value which can vary from -limit to + limit. We choose the 50% D/A output (4096/2 = 2048) as the mean control point so that negative values from* `PIDcompute()` *means D/A output will be < 2048, positive values means D/A output will be > 2048.*