



IEC 61131-3 Programming Guide for MAPware-7000

Your Industrial Control Solutions Source

www.maplesystems.com



For use with the following:

- HMC7000 Series HMI + PLC
- HMC3000 Series HMI + PLC
- HMC2000 Series HMI + PLC
- HMC4000 Series HMI + PLC
- MLC Series PLCs

COPYRIGHT NOTICE

This manual is a publication of Maple Systems, Inc., and is provided for use by its customers only. The contents of the manual are copyrighted by Maple Systems, Inc.; reproduction in whole or in part, for use other than in support of Maple Systems equipment, is prohibited without the specific written permission of Maple Systems.

WARRANTY

Warranty Statements are included with each unit at the time of purchase and are available at

www.maplesystems.com

TECHNICAL SUPPORT

This manual is designed to provide the necessary information for trouble-free installation and operation of Maple Systems products. However, if you need assistance, please contact Maple Systems:

- Phone: 425-745-3229
- Email: support@maplesystems.com
- Web: www.maplesystems.com

Table of Contents

Chapter 1 – Introduction	6
What is IEC 61131-3?	6
Available Languages.....	6
The Editor Window	6
Logic Blocks and Execution Style	8
Power Up block.....	9
Main Program block.....	9
Subroutines.....	10
Timer Interrupts.....	10
User Defined Function Block (UDFB)	11
Function Block Instances	11
Tags.....	11
Tag Scope.....	12
Arrays.....	12
Exporting and Importing Logic Blocks.....	14
Connecting to External Devices	14
Ethernet Settings	14
Device Settings.....	16
Creating and Addressing External Tags.....	16
Chapter 2 – Quick Start Sample Project	18
Introduction.....	18
Sample Project Design	18
Create a New Project	18
Add Tags to the Project.....	18
Logic Blocks.....	19
Create a User Defined Function Block (UDFB)	19
Create a Second UDFB to Initialize Instances of the Scale Function Block	23
Use the UDFBs in Ladder Diagram Blocks	25
Define the Main Routine Block 1	28
Create Screen Objects.....	29
Review	31
Chapter 3 – Online Monitoring and Logic Simulation	32
Online Monitoring	32
Simulating IEC 61131-3 Logic on a PC.....	35
Chapter 4 – Ladder Diagram (LD)	38

Overview	38
Execution Order	38
Creating a Ladder Diagram logic block.....	39
The Ladder Diagram Editor Window.....	40
Adding Logic to the Block.....	41
Using the Quick Select Menu	41
Changing Contact and Coil Execution Style.....	41
Configuring Function Blocks	42
Jumps	45
Comments.....	46
Chapter 5 – Function Block Diagram (FBD)	47
Overview	47
Function Block Diagram vs. Ladder Diagram	47
Adding Logic to the Block.....	48
Quick Select Menu	49
Changing Contact and Coil Execution Style.....	59
Adding Function Blocks from the Instruction List	60
Assigning Inputs and Outputs to FBD elements.....	60
Selecting Function Block Instances	61
Jumps and Labels	62
Chapter 6 – Structured Text (ST)	65
Overview	65
Statements.....	65
Expressions	65
Operands	66
Comments.....	66
Adding Logic to the Block.....	67
Operators and Expressions	67
Keywords and Program Flow	69
Preprocessor Commands	73
Quick Select Menu Options	74
Functions, Subroutines and Function Blocks	80
Functions vs. Function Blocks	81
Ordinary Function Calls.....	81
Function Block Calls	82
Chapter 7 – Instruction List (IL)	84

Overview	84
Adding Logic to the Block.....	84
ST Instructions supported.....	84
Instruction List (IL) Instructions	84
Quick Select Menu Options	88
Chapter 8 – Sequential Function Chart (SFC)	93
Overview.....	93
Adding Logic to the Block.....	94
Building a Simple SFC Diagram	94
Steps	97
Transitions	99
Divergences	101
Quick Select Menu Options	103
Child SFC Programs	110
Controlling an SFC Child Program	111
SFC Best Practices	112
SFC Design Patterns	112

Chapter 1 – Introduction

What is IEC 61131-3?

IEC 61131-3 is a section of an International Electro-Technical Committee (IEC) standard that provides a definition for implementing PLC programming software. The standard was first introduced in 1993 as the result of an effort to standardize the myriad PLC logic editors in the automation market place. The goal of the standard is to give automation professionals a familiar environment and set of tools to create PLC programs across vendor platforms. MAPware-7000 has editors implemented for all five programming languages defined by the standard. This manual will give you all the information you need to put these powerful tools to use.

Available Languages

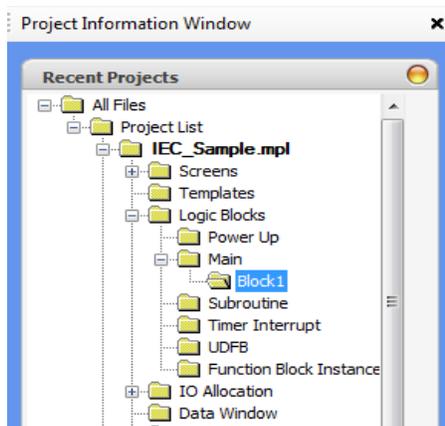
The standard defines five unique programming languages. Editors for each of these languages are available in MAPware-7000. Details on using each of the editors are given in the chapters that follow. The Languages are:

1. **Ladder Diagram (LD)** – Graphical language that simulates an electrical circuit; program instructions are attached as discrete elements in the circuit and are executed when “energized” ([Chapter 4](#))
2. **Function Block Diagram (FBD)** – Graphical language based on logic diagrams. Functions are represented by blocks; complex operations can be built by interconnecting function blocks ([Chapter 5](#))
3. **Structured Text (ST)** – Text based programming language. Programs are built using keywords, operators and function calls. ([Chapter 6](#))
4. **Instruction List (IL)** – Text based procedural programming language ([Chapter 7](#))
5. **Sequential Function Chart (SFC)** – Graphic programming language in which program execution is modeled as a flow chart. Programs are developed by adding blocks to the flow chart ([Chapter 8](#))

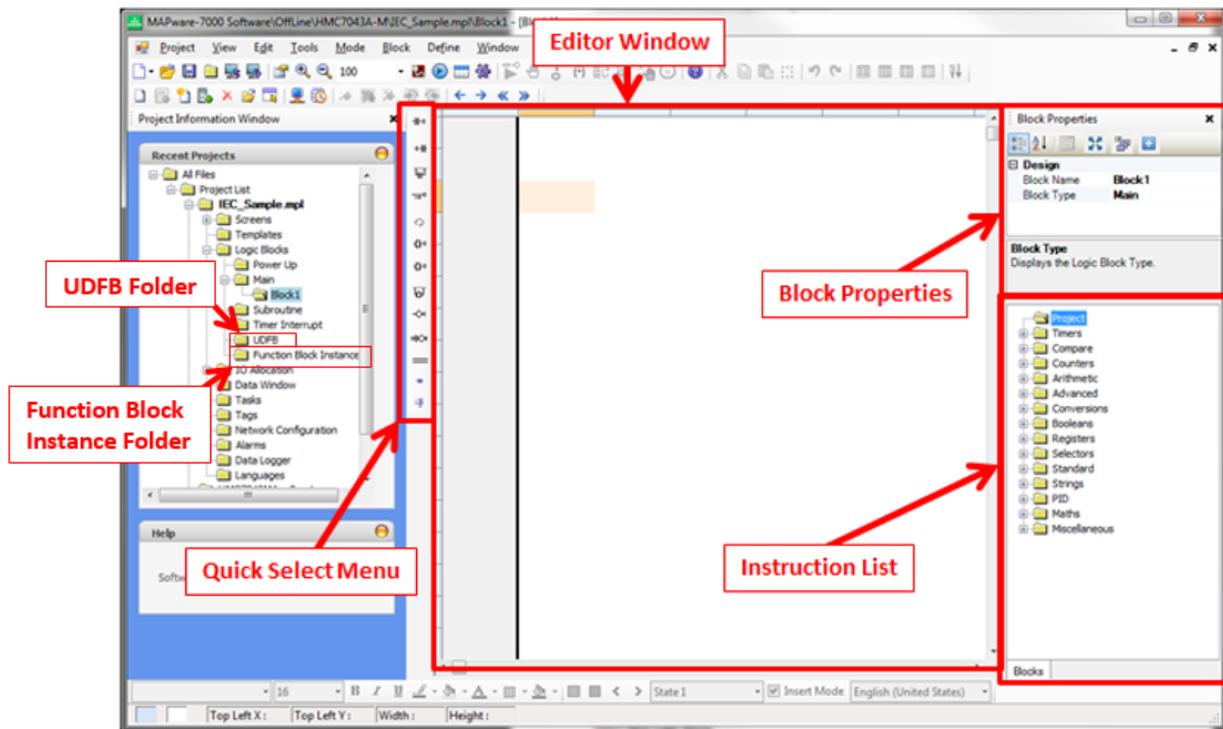
The logic created using these languages is composed of discrete instructions or logic blocks. The available instructions are documented in the MAPware-7000 help file. In MAPware-7000 Double-click a particular instruction in the Instruction List to open that instruction’s help file entry.

The Editor Window

Let’s take a quick tour of some of the aspects of MAPware-7000 that are unique to an IEC 61131-3 project. Projects created using IEC 61131-3 programming mode contain a **Project Information Window** that allows easy navigation to various parts of the project. When a new project is created it contains a default logic block named Block1.



Clicking the folder for Block1 in the **Project Information Window** displays the editor for that block. The default logic block uses the Ladder Diagram editor, so the Ladder Diagram editor is displayed when the Block1 folder is selected in the Project Tree:



The parts of this window specific to the Ladder Diagram editor are covered in [Chapter 4](#). Elements common to all of the editors are:

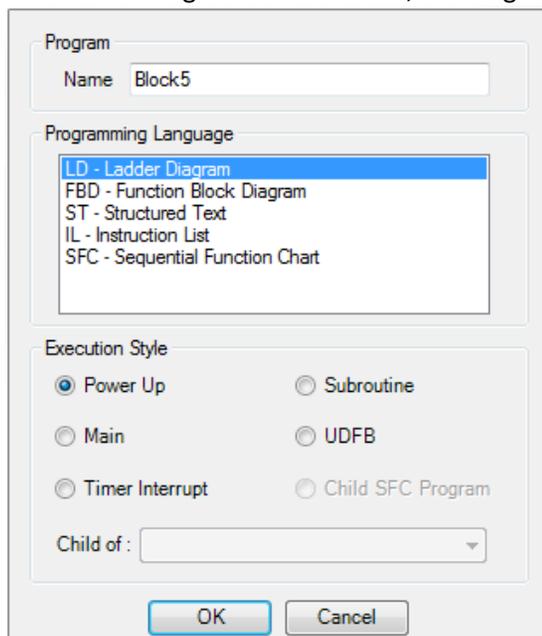
- **Block Properties** – Displays the name and [Execution Type](#) of the block.
- **Instruction List** – Lists all of the available instructions for the project. Instructions are categorized according to functionality. Expand the node for a given category to see individual instructions. To select a particular instruction simply click and drag it into the editor window. Subroutines and UDFBs (Function Blocks that users create) will appear under the Project folder of the instruction list once they are defined. Information on how to configure and use a given function block is available in the help file. To access the help file entry for any of the blocks in the instruction list simply double-click the instruction.
- **Editor Window** – This is where the logic is defined. It displays the graphical or text representation of the logic program.
- **Quick Select Menu** – The options available on this menu depend on the editor in use but provide quick access to common program elements. Click the location in the editor where you want to place the element then click the element in the quick select menu to place it in the editor.
- **UDFB Folder** – This folder contains the definitions for User Defined Function Blocks. To add or edit a UDFB, click the block name in this folder. To use a UDFB in another logic block, select it from the Project folder of the Instruction List.
- **Function Block Instance Folder** – This folder contains a list of all the Function Block Instances in the project. Each block can have multiple instances each instance will have its own private set of data to work with. Thus one type of function block can be utilized for multiple purposes in the project.

Logic Blocks and Execution Style

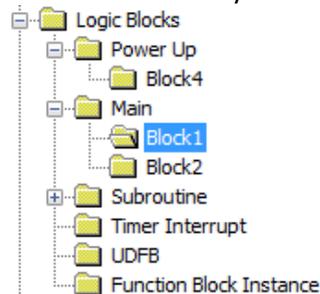
Programs created using the above editors are organized into discrete units called **Blocks**. The job of the programmer is to create logic blocks, define what they do and how they interoperate with each other to model the operation of the system they are building. A MAPware-7000 project can contain many blocks; each created using any of the various editors. In addition to the language used and the logic they contain, blocks are differentiated by Execution Style. Execution Style determines when and how the block is executed. The available Execution Styles are:

- [Power Up](#)
- [Main](#)
- [Subroutine](#)
- [Timer Interrupt](#)
- [User Defined Function Block \(UDFB\)](#)

When a new logic block is created, the language to be used and the Execution Style are specified:

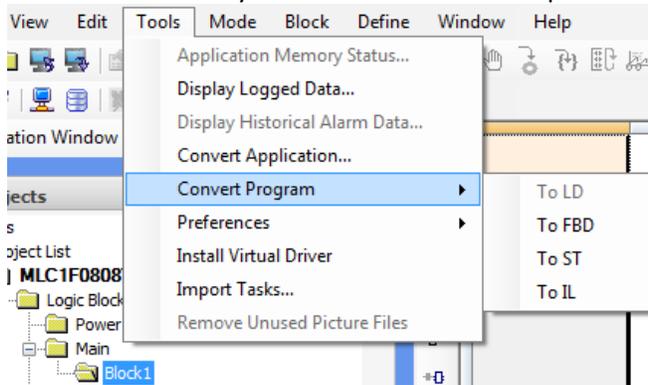


After the block is created, the Execution Style cannot be changed. The current Execution Style of a logic block is indicated by the block's location within the Logic Blocks folder of the project tree:



To change the programming language of a block after it has been created, open the selected logic block in the editor window and select **Tools > Convert Program > To [...]** from the main menu. It is not possible to convert to or from the Sequential Function Chart language, but all other languages are options. The

converted code may need minor edits to compile.

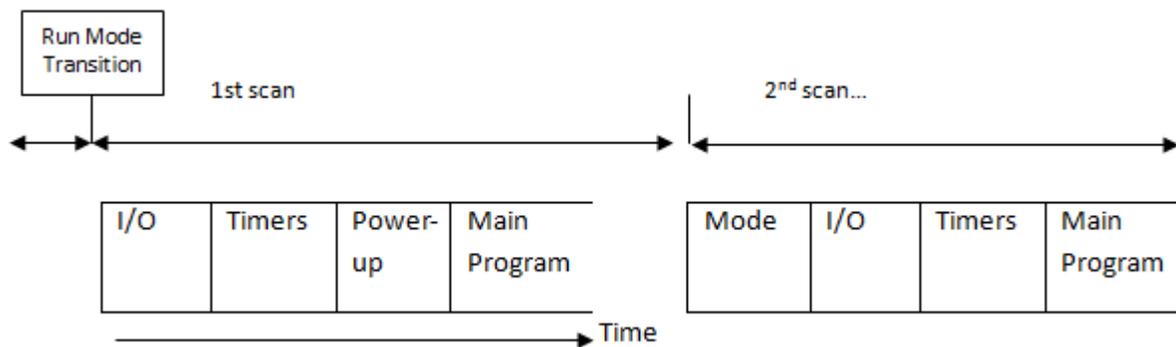


The following sections give more detail on how the various Execution Styles operate and when to use them.

Power Up block

If Power Up blocks are present, they are executed once at the beginning of the first scan (before main block execution). Therefore, Power Up blocks can be used to set initial values into registers.

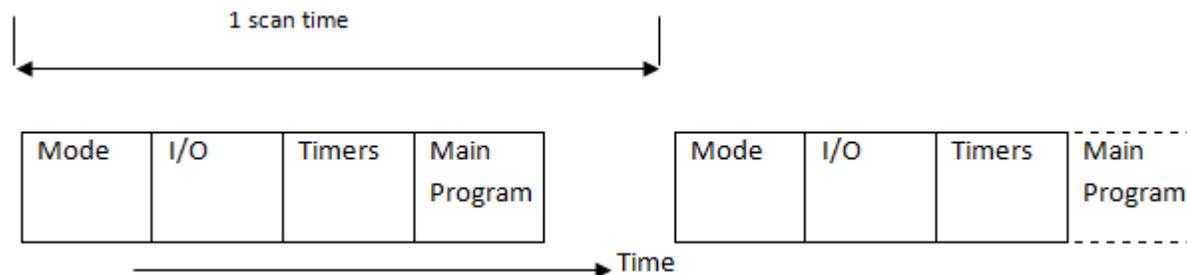
The figure below shows the first scan operation:



Main Program block

Main program blocks are the core of the user program. They are executed once during each scan.

Multiple logic blocks can be created (up to 256) and used as Main Program blocks. During execution, the HMC/MLC starts with the first block listed. When completed, it will execute each block in sequence. The figure below shows a typical scan sequence:



Where:

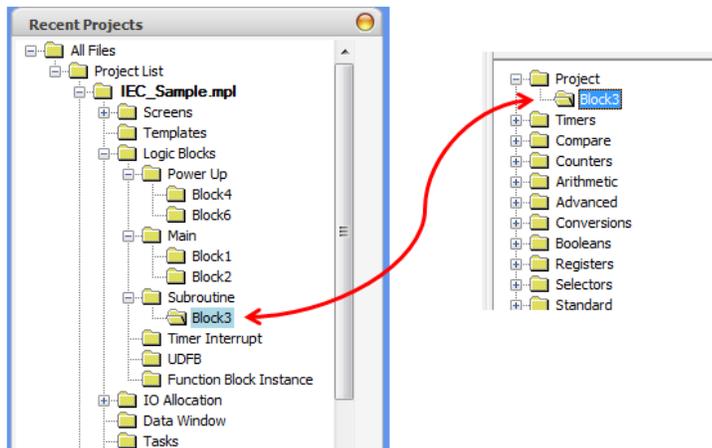
- Mode- determines mode of operation (Run, Halt, etc.)
- I/O- update and process all inputs and outputs
- Timer- update all running timers
- Main Program- all logic blocks created under Main

The length of a scan is not deterministic. It depends on what blocks are executed etc. If a more reliable execution time is required use a timer interrupt routine.

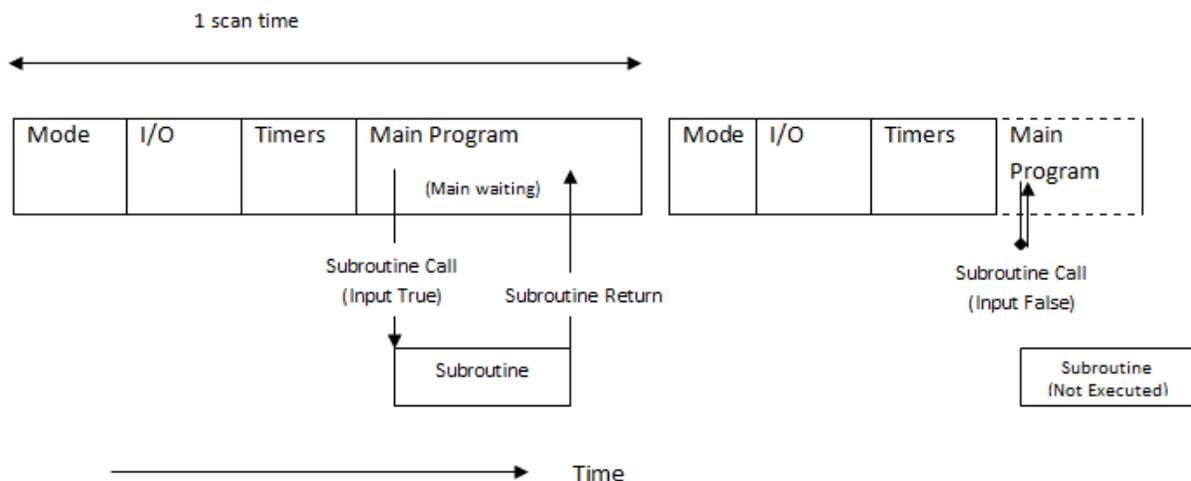
Subroutines

Subroutines operate the same as the main program except they are not executed unless specifically called on by another logic block. Subroutines are useful when you have a set of commands that should be executed only under certain conditions. A maximum of 256 subroutines can be created (dependent upon total memory available).

When a block is created as a Subroutine block, it will appear in the Instruction List as a function block that can be used in another block:

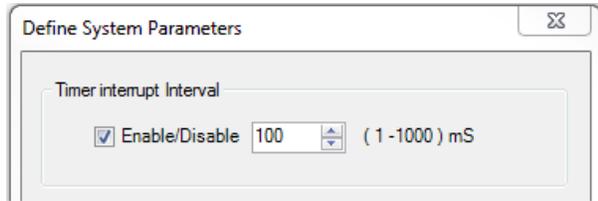


To execute the subroutine, select the instruction and add it to a main logic block. When the input condition to this instruction is true the Subroutine will be executed:



Timer Interrupts

Timer interrupt logic blocks are given the highest priority when the MAPware-7000 program is executed. The timer interrupt is enabled by going to the **Define > System Parameters** dialog box and checking Timer Interrupt Interval.



When enabled, the timer interrupt routine is executed based upon the interval selected (range is 1-1000 milliseconds).

All other operations are suspended whenever the time interrupt activates. Use this feature if you have a continuous operation that is time critical. Note that since timer interrupt routines halt all other activities, it is best to minimize its impact on the performance of the controller by using it sparingly. Design the interrupt routine to be as short as possible and adjust the timer interrupt interval to be the maximum setting that can still meet the requirements of your application.

User Defined Function Block (UDFB)

A User Defined Function Block (UDFB) operates similarly to a subroutine. It is a logic procedure defined by the user that can be executed as a component in another block. Once defined, the UDFB will be available to select from the list of instructions in the lower left of the editor window. The UDFB also allows the user to define input and output parameters making the block easy to reuse throughout the project. The Quick Start section below provides an example of how to create and use UDFBs.

Function Block Instances

One of the major advantages of the IEC61131-3 editor is the ability to modularize and reuse functionality through the use of Function Blocks. Once the logic in a Function Block is defined it can be used to create Function Block Instances. A Function Block Instance contains all of the logic defined in the Function Block as well as its own set of data to operate on. The Function Block can be thought of as a cookie cutter and the Function Block Instance is the cookie that the cutter creates. Multiple instances of the same block can be defined and each will have its *own set of data to work with*. The same block can be used for multiple purposes. Function Block Instances can be passed as parameters to other Function Blocks. Instances can be defined using the built in Function Blocks or User Defined Function Blocks.

The Function Block Instance folder, in the **Project Information Window**, is used to create Function Block Instances and contains a list of all the instances in the project. For details on how to create and use Function Block Instances refer to the [Quick Start](#) section below.

Tags

All of the logic in the various logic blocks operates on tags. Tags are linked to Graphic Objects displayed on the HMC, where the operator can interact with them using the touchscreen, or to the HMC's IO modules, or some other IO mechanism. Tags are defined and listed in the tag database which can be opened by clicking the **Tags** folder of the project tree:

Tag No	Tag Name	Data Type	Attribute	Tag Address	Port	Node	Node Name	Tag Category
142	Scale/MaxScaled	REAL (L)	Read Write	-	-	0	HMC7043A-M	User Defined Tag
139	iMaxRaw	INT	Read Write	-	-	0	HMC7043A-M	User Defined Tag
174	TempF	REAL	Read Write	-	-	0	HMC7043A-M	User Defined Tag
173	TempCelsius	INT	Read Write	-	-	0	HMC7043A-M	User Defined Tag
143	iMaxScaled	INT	Read Write	-	-	0	HMC7043A-M	User Defined Tag
152	ScaleInt/MinOut	INT (L)	Read Write	-	-	0	HMC7043A-M	User Defined Tag
151	ScaleInt/MinIn	INT (L)	Read Write	-	-	0	HMC7043A-M	User Defined Tag
150	ScaleInt/MaxOut	INT (L)	Read Write	-	-	0	HMC7043A-M	User Defined Tag
149	ScaleInt/MaxIn	INT (L)	Read Write	-	-	0	HMC7043A-M	User Defined Tag
165	ScaleInt/MinOut	REAL (L)	Read Write	-	-	0	HMC7043A-M	User Defined Tag
163	ScaleInt/MinIn	REAL (L)	Read Write	-	-	0	HMC7043A-M	User Defined Tag
164	ScaleInt/MaxOut	REAL (L)	Read Write	-	-	0	HMC7043A-M	User Defined Tag

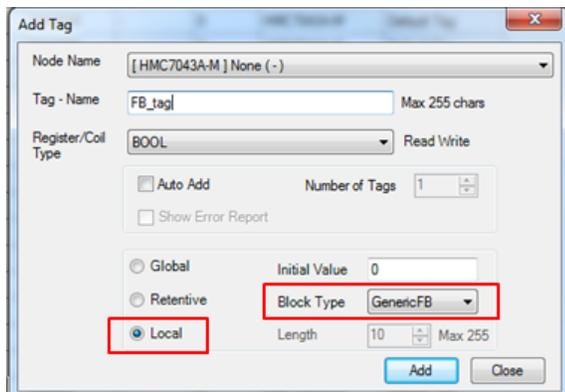
Tags are handled somewhat differently in IEC 61131-3 projects vs. Native Ladder projects. In IEC mode projects tags are not assigned an explicit address. Instead they are given name and a type, MAPware-7000 is responsible for allocating and tracking a memory address for the tag. For more information on creating and using tags refer to the MAPware-7000 programming manual. The following table lists the data types available in IEC 61131-3 programming mode:

Data Type	Description	Bits	Value Range
BOOL	Boolean value can have one of two values: 1 (ON/True) or 0 (OFF/False)	1	0 to 1
BYTE	Unsigned Short (same as USINT)	8	0 to 255
DINT	Signed Double Integer	32	-2,147,483,648 to 2,147,483,647
DWORD	Unsigned Double Integer (same as UDINT)	32	0 to 4,294,967,295
INT	Signed Integer	16	-32,768 to 32,767
REAL	IEEE 754 format, single precision floating point number	32	N/A
SINT	Signed Short Integer	8	-128 to 127
STRING	Variable Length string of ASCII characters	1 to 255 bytes	N/A
TIME	Time of Day	32	N/A
UDINT	Unsigned Double Integer (same as DWORD)	32	0 to 4,294,967,295
UINT	Unsigned Integer	16	0 to 65,535
USINT	Unsigned Short Integer (Same as BYTE)	8	0 to 255
WORD	Unsigned Integer (Same as UINT)	16	0 to 65,535

Tag Scope

Function Block and Function Block Instance Tags

In addition to global scope tags, tags can be associated with a function block instance. Function blocks can have; Input, Output and Internal tags. When an instance of the Function Block is added to the project it will have a copy of each of the tags. Internal tags can be defined in the tag database by setting the Scope to Local and selecting the Function Block to associate the tag with:



Input and Output tags are defined by right-clicking the UDFB's folder in the project tree and selecting the **Edit Parameters** option. Once defined, Function Block Tags will appear in the tag database as <Function Block Name>\<Tag Name>. For example GenericFB\FB_tag is a tag internal to the GenericFB function block named FB_tag.

Arrays

In the MAPware-7000 tag database you can declare an array of a single type of tag (not available for MLC PLCs). Arrays can have up to three dimensions and each dimension can have up to 255 elements. Arrays provide a powerful way to store and index into data at runtime. Arrays can be used for applications such as;

look up tables, recipe storage, FIFO stacks, indirect addressing, or anything that requires an index into a data set.

Note: Arrays can only be indexed in the logic section of the project. To view data from an array on a screen, copy the data from the array to another tag in a logic block, or use the individual element tags created in the tag database. When the array is created the tag database will add individual tags for each element in the array. These tags can then be referenced on HMI screens.

Declaring Arrays

Arrays are declared in the tag database. To declare an array, right-click in the tag database and open the **Add Tag** window, specify a tag name and type for the array. Enter the size for each dimension in the **Dimension** field:

If a multi-dimensional array is required, enter the size for each dimension separated by a comma. The highest order dimension is listed on the left. For example:

- Enter: 8 for a one dimensional, 8 element array
- Enter: 6, 8 for a two dimensional array consisting of six one dimensional, 8 element arrays
- Enter: 4, 6, 8 for a three dimensional array consisting of four 6 x 8 two dimensional arrays

Click **Add** to allocate the array. In the tag database a tag is created for the array itself and for each element in the array. Indices for the individual element tags are specified by the numbers at the end of the tag name. Array elements are enumerated beginning at zero. An 8 element array will have elements 0 through 7.

Using Arrays

To use an array in a logic block, type the name of the array in any field where a tag of the array type can be used. The index of the array is enclosed in square brackets. If there are multiple dimensions, separate the index for each dimension with a comma. The index can be a tag or a literal value. For example, a three dimensional array named *Arr* could indexed with literal values as:

Arr[2,3,1]

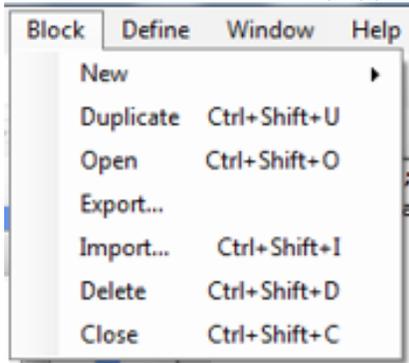
Or, using the tags *Index1*, *Index2*, *Index3* the array can be configured as:

Arr[*Index1*, *Index2*, *Index3*]

As the value of *Index1*, *Index2* and *Index3* change over time, the expression will point at a different array element.

Exporting and Importing Logic Blocks

Logic blocks can be exported for use in other projects. Logic blocks are imported and exported using the Block menu. This menu only appears when a logic block is open in the editor window:



Export

To export a logic block, navigate to the block in the **Project Information Window**, and click the block's folder to open the block for editing. The block can be a normal logic block or a UDFB.

From the **Block** menu select **Export**.

A windows dialog block will appear, allowing you to navigate to name the exported block and select a folder in which to save the block. The exported block consists of two files: an *.xml file which contains the logic used by the block, and a *.txt file which specifies meta data about the block, such as the version used to create it, the inputs and outputs etc. Both files are required to store the logic block for later import.

Import

To import a logic block, navigate to and open any block in the **Project Information Window** to enable the Block menu. Select **Block > Import**. Alternatively, enter <Ctrl>+<Shift>+I from anywhere in the project to open the import block dialog. In the dialog that opens, navigate to the *.xml file that contains the logic for the block to be imported.

Note: the folder containing the *.xml file must also contain the *.txt file for the logic block.

Once the block is imported it will appear under the project folder and will be available to use like any other user created block.

Connecting to External Devices

Details for using a particular driver can be found in the Controller Information Sheet for the device you want to use. Controller Information Sheets are available on the Maple Systems website at:

www.maplesystems.com/support.htm

This section provides some general information about connecting external devices to an HMC or MLC unit in IEC-61131-3 programming mode.

Ethernet Settings

To communicate using the Ethernet port, the Ethernet settings of the HMC or MLC must be configured with an address accessible to the external device. The unit's IP address is configured on the Ethernet tab of the

Project Configuration window (Project > Properties):

The screenshot shows the 'Project Configuration' dialog box with the 'Ethernet' tab selected. The 'Port settings for HMC3070A-M' section includes a 'DHCP' checkbox (unchecked). Below it are input fields for 'IP Address' (192 . 168 . 0 . 254), 'Subnet Mask' (255 . 255 . 255 . 0), 'Download Port' (5000), 'Default Gateway' (0 . 0 . 0 . 0), and 'Monitoring Port' (1100). The range '(1024-65535)' is shown next to the port fields.

The IP address configured in this window must be accessible from the device to be connected to. Generally this means it must be on the same subnet, which boils down to:

- The portion of the IP address masked by the subnet mask must be the same for all devices on the subnet.
- The portion of the IP address not masked by the subnet mask must be unique for each device on the subnet.

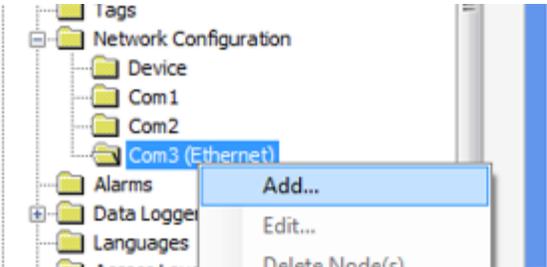
Once the IP address is set, it must be downloaded to the hardware. There is a checkbox in the download window for Ethernet settings. Don't forget to check this box if the IP address needs to be updated:

The screenshot shows the 'Download to device' dialog box. On the left, 'Ethernet' is selected under 'Mode'. The 'Download Options' section has several checkboxes: 'Firmware' (unchecked), 'Application' (unchecked), 'Logic' (checked), 'Font' (unchecked), 'Logged Data' (unchecked), and 'Ethernet Settings' (checked and highlighted with a red box). The 'Device Settings' section has four checkboxes: 'Automatically put unit in halt mode before download' (checked), 'Automatically put unit in run mode after download' (checked), 'Initialize keep memory area after download' (unchecked), and 'Initialize all device registers except keep memory after download' (checked). At the bottom, there is a progress bar at 0% and buttons for 'Options <<', 'Download', 'Abort', and 'Close'. The status bar at the bottom left says 'Ready'.

For HMC models, the IP address is displayed on the screen during the boot process.

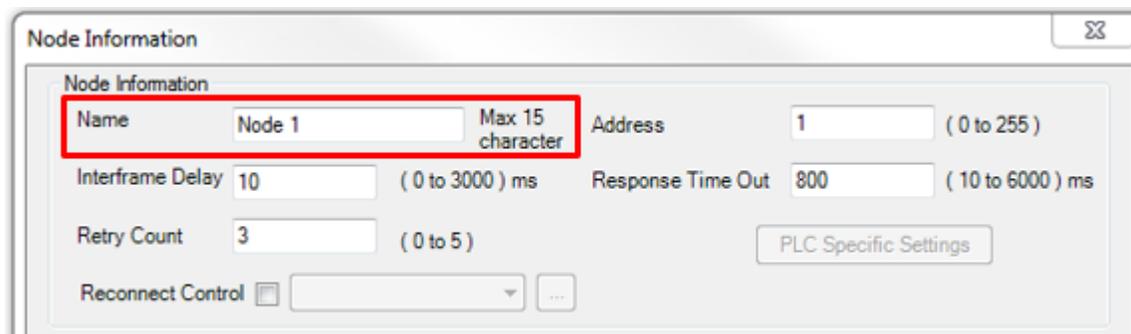
Device Settings

To communicate to an external device such as an HMI or PLC, the communications driver for the device must be configured on the HMC/MLC port that the device is connected to. To add a device to the project, expand the **Network Configuration** folder in the **Project Information Window**, right-click the Com port to be used, and then select **Add** from the context menu.



This will open the **Node Information** window, where you can configure the settings for your device. Detailed information for each communications protocol is available in the Controller Information Sheet for the device you are using. These can be found in the support center at www.maplesystems.com/support.htm.

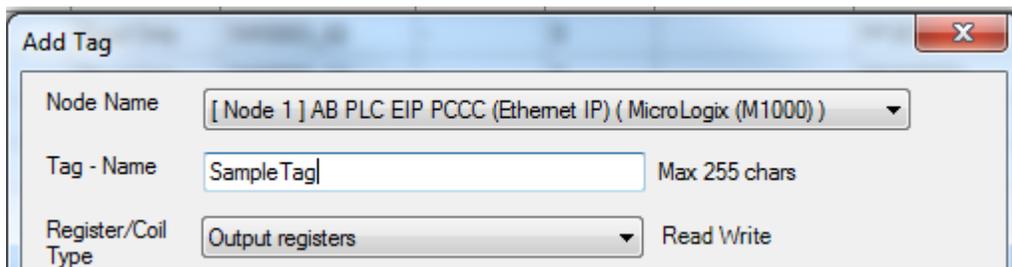
The Node Name configured in this window will be selectable in the tag database in order to create tags addressed to this device:



Creating and Addressing External Tags

HMC/MLC as Master

Once the device is configured it will be available to select as a **Node Name** in the **Add Tag** window. Click the **Tags** folder in the **Project Information Window** to open the **Tag Database**. Right-click in the tag list and select **Add** to open the **Add Tag** window. Select the node in the **Node Name** dropdown list, then enter the name and corresponding address on the device:



Once created, a tag can be used on HMC screens like any other tag. However, tags created on external nodes are not available to use in logic blocks. To use the tag in a logic block, it must be copied from the PLC to the local HMC/MLC onboard memory. This can be done with a task as shown below.

To copy data from an external device to internal tags

Configure a task to continuously copy data to/from an external device:

- Click the **Tasks** folder in the **Project Information Tree** to open the **Tasks** window.
- Select the radio button for the **Global Tasks** section
- From the select task dropdown list, select the:
 - **Copy HMI Block to HMI/PLC Block** to copy data from the external device to the unit.
 - **Copy HMI/PLC Block to HMI Block** to copy data from the unit to an external device.
- Configure the source and destination tags. **Tag A** is the destination and **Tag B** is the source.
- Enter the number of words to copy.
- Click **Add** to add the newly configured task to the **Global Tasks** box.

HMC/MLC as Modbus Slave

An external device can be configured to initiate read and write Modbus commands to the HMC/MLC which will respond as a slave. If the HMC/MLC is connected to the Modbus Master via Ethernet, there is no need to select a Modbus TCP/IP (Slave) driver in the Network Configuration Window. The HMC/MLC supports this driver on the Ethernet port by default. The HMC/MLC will respond to a properly formatted Modbus TCP request sent to the IP address (TCP port 502) configured in the **Ethernet Settings** tab in **Project > Properties**.

If the HMC/MLC is connected to the Modbus Master via serial com port, you must assign the Modbus RTU (Unit as Slave) driver to one of the available serial Com ports in the Network Configuration folder.

Next, you must map tags that are created in the Tag Editor to Modbus addresses. Any tags of the HMC/MLC that are mapped to Modbus addresses can then be accessed by the external Modbus Master device. For example, we wish to map the tags listed below to Modbus addresses:

Tag No	Tag Name	Data Type	Attribute	Tag Address	Port	Node	Com 1	Ethernet	Node Name	Tag Category	Export Tag
195	Coil3	BOOL	Read Write	-	-	0			HMC3070A-M	UserDefined Tag	<input type="checkbox"/>
194	Coil2	BOOL	Read Write	-	-	0			HMC3070A-M	UserDefined Tag	<input type="checkbox"/>
193	Coil1	BOOL	Read Write	-	-	0			HMC3070A-M	UserDefined Tag	<input type="checkbox"/>
192	HoldingRegister3	INT	Read Write	-	-	0			HMC3070A-M	UserDefined Tag	<input type="checkbox"/>
191	HoldingRegister2	INT	Read Write	-	-	0			HMC3070A-M	UserDefined Tag	<input type="checkbox"/>
190	HoldingRegister1	INT	Read Write	-	-	0			HMC3070A-M	UserDefined Tag	<input type="checkbox"/>

Note that the Ethernet column is available by default. This column is used to specify Modbus TCP/IP (Slave) addresses for tags. To see the Com 1 column, you must add the Modbus RTU (Unit as Slave) driver in the Network Configuration folder.

To map a tag to a Modbus Slave driver address, simply enter the desired address in the column corresponding to that Com/Ethernet port being used. Note: click once anywhere on the row of the target tag to highlight (blue), then click the Com or Ethernet cell.

Tag No	Tag Name	Data Type	Attribute	Tag Address	Port	Node	Com 1	Ethernet	Node Name	Tag Category	Export Tag
195	Coil3	BOOL	Read Write	-	-	0	000001	000001	HMC3070A-M	UserDefined Tag	<input type="checkbox"/>
194	Coil2	BOOL	Read Write	-	-	0	000002	000002	HMC3070A-M	UserDefined Tag	<input type="checkbox"/>
193	Coil1	BOOL	Read Write	-	-	0	000003	000003	HMC3070A-M	UserDefined Tag	<input type="checkbox"/>
192	HoldingRegister3	INT	Read Write	-	-	0	400001	400001	HMC3070A-M	UserDefined Tag	<input type="checkbox"/>
191	HoldingRegister2	INT	Read Write	-	-	0	400002	400002	HMC3070A-M	UserDefined Tag	<input type="checkbox"/>
190	HoldingRegister1	INT	Read Write	-	-	0	400003	400003	HMC3070A-M	UserDefined Tag	<input type="checkbox"/>

Chapter 2 – Quick Start Sample Project

Introduction

This section guides you through the steps needed to create and run a simple IEC 61131-3 project. It will use a common engineering task, mapping a value from one scale to another, to demonstrate how the features of the IEC editor can be used to develop solutions. The sample project demonstrates:

- Use of the **Structured Text (ST)**, **Ladder Diagram (LD)** and **Function Block Diagram (FBD)** editors
- Defining a **User Defined Function Block (UDFB)**
- Illustrates the difference between a **Function Block** and a **Function Block Instance**
- Passing an instance of a Function Block as a parameter to another function block

Sample Project Design

The problem this sample project will solve is mapping an input value linearly to produce an output value on a different scale. The sample scales from Celsius to Fahrenheit and from a raw input value to an engineering value (i.e. from a 12-bit raw input to a voltage) but, the solution should be general enough so that the Function Blocks created can be re-used for any linear scaling operation. We also want the calculation to be as efficient as possible. The task of scaling a number can be broken down into two parts.

1. Calculate the slope and offset
2. For a given input calculate the output

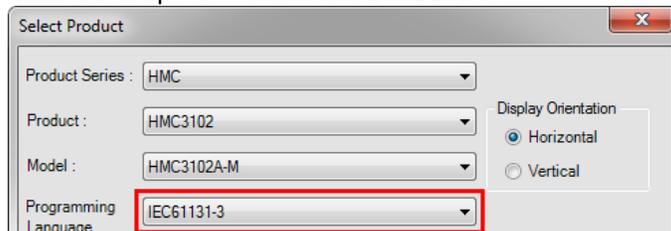
We could do both of these operations in one function block but that would mean that every time the value is scaled the slope and offset are recalculated. Instead we will create two function blocks; one to calculate the slope and offset given maximum and minimum values, and one to actually do the scaling as the input changes.

Creating User Defined Function Blocks to accomplish the task eliminates the need to repeat logic blocks that do the same thing, making the project easier to maintain. If something needs to be changed it can be edited in one place and take effect throughout the project.

Create a New Project

Note: the 3" models (HMC7030A-M, HMC7030A-L) do not support IEC mode projects.

1. To create the project, select **Project > New**. The **Select Product** window is displayed.
2. Select the **Product Series**, **Product**, and **Model**.
3. Leave the **Display Orientation** as **Horizontal**.
4. Native Ladder is the default **Programming Language** and for this sample project must be changed. Click the drop down to select **IEC61131-3**.



5. Click **OK**. A new project is created with a default name.
6. Select **Project > Save** to save the project with a unique name.

Add Tags to the Project

To begin building our sample project we will create some tags to be used later.

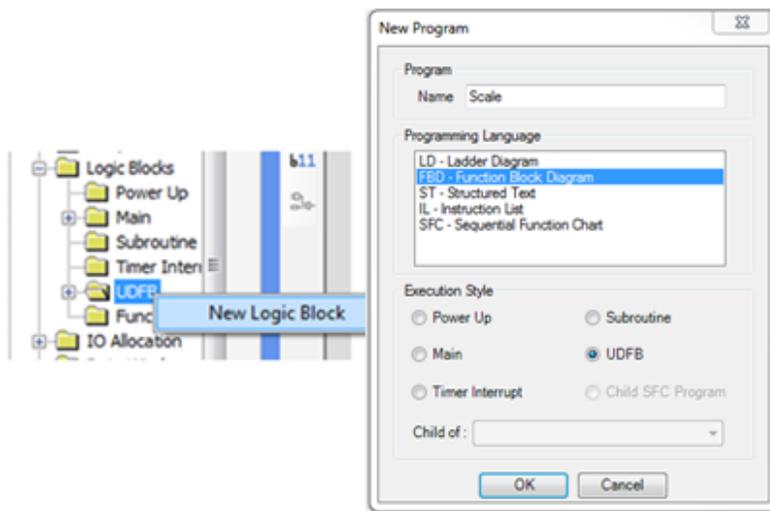
7. Click the **Tag** folder in the **Project Information Window**.
8. Right-click in the **Tag Database** and select **Add** from the context menu.

9. Use the **Add Tag** window to create the following tags.

Tag Name	Type	Scope	Description
<i>Temp1C</i>	INT	Global	Simulated Temp input 1 [C]
<i>Temp2C</i>	INT	Global	Simulated Temp input 2 [C]
<i>Temp1F</i>	REAL	Global	Scaled Temp output 1 [F]
<i>Temp2F</i>	REAL	Global	Scaled Temp output 2 [F]
<i>RawInput</i>	INT	Global	Simulated IO card input
<i>Voltage</i>	REAL	Global	Input scaled to a voltage value

Logic Blocks

Create a User Defined Function Block (UDFB)

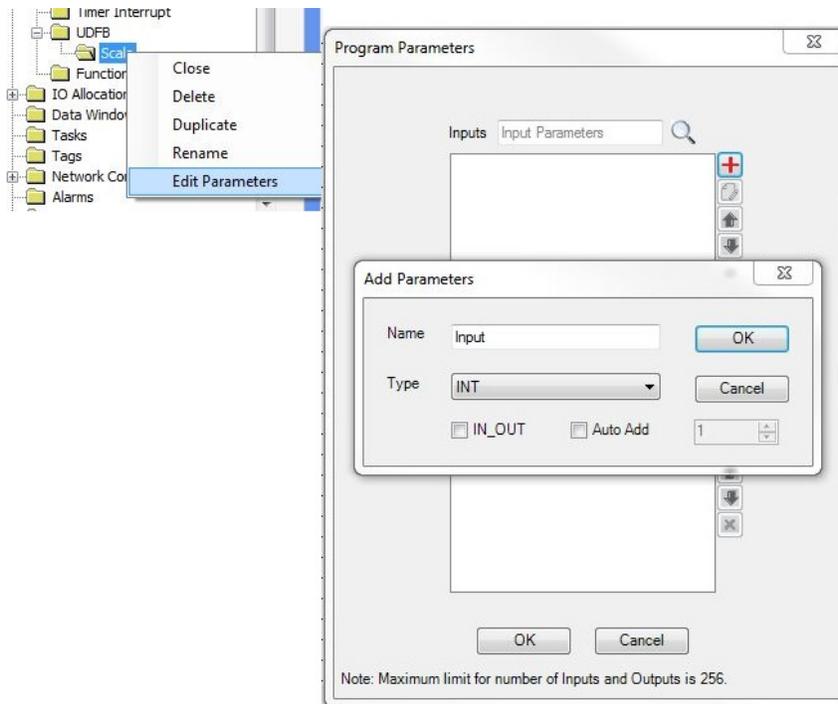


Next we want to create our own User Defined Function Block (UDFB). This block performs the scale operation. We will use the Function Block Diagram (FBD) editor to define this block.

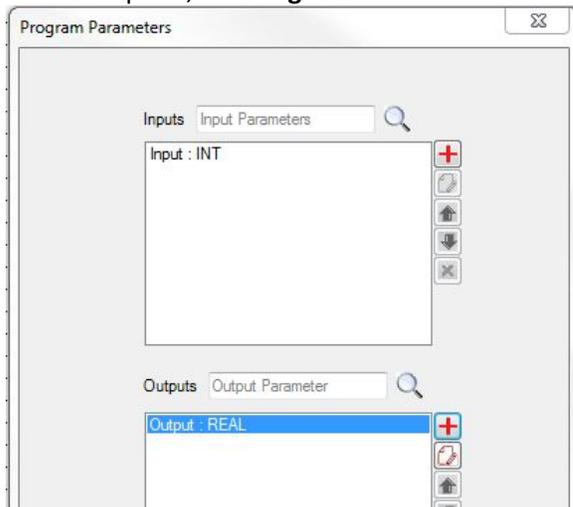
1. Expand the **Logic Blocks** folder in the **Project Information Window**
2. Right-click the **UDFB** subfolder and select **New Logic Block**
3. In the **New Program** window name the function *Scale* and select **FBD – Function Block Diagram** as the **Programming Language** and **UDFB** as the **Execution Style**
4. Click **OK** to create the Logic Block
5. A new block called *Scale* appears in the **UDFB** folder of the **Project Information Window**, and a new instruction called *Scale* appears in the **Project** folder of the **Instruction List**.



Before creating the logic for this block we need to define the inputs and outputs so that other logic blocks can pass data to it.



6. Right-click the *Scale* folder in the Project Information Window and select **Edit Parameters**
7. In the **Program Parameters** window click  'Add Input Parameter' under **Inputs**.
8. The **Add Parameters** pop-up window allows you to specify a name and data type for a new input parameter. This block contains only one input parameter imaginatively named *Input*. Set the type to INT then click *OK*.
9. Next click  'Add Output Parameter' under **Outputs** to create the output. It is named *Output* and has type REAL.
10. Once complete, the **Program Parameters** window should appear as shown here.



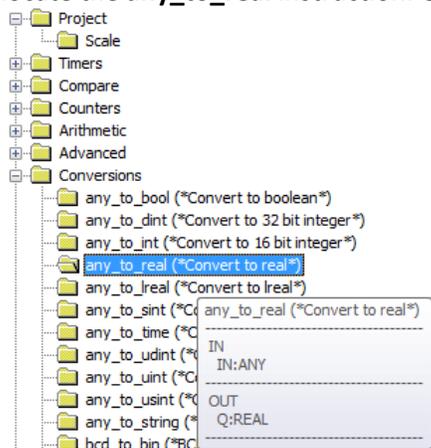
Click **OK** to create the Parameters.

Both tags should now appear in the tag database with the Function Block/Local Tag format we talked about earlier.

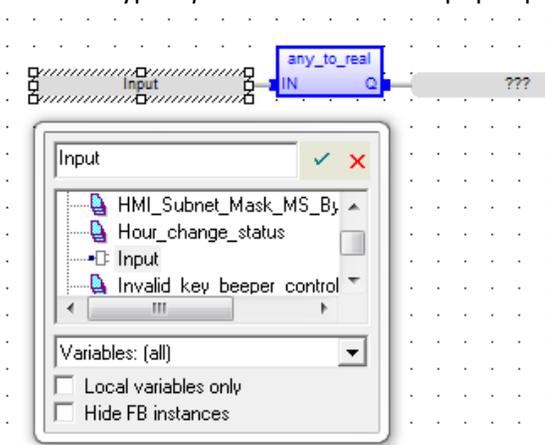
Tag No	Tag Name	Data Type	Attribute	Tag Address	Port	Node	Ethernet	Node Name	Tag Category	Export Tag
1	Temp2C	INT	Read Write	-	-	0		HMC3102A-M	User Defined Tag	<input type="checkbox"/>
2	Temp1C	INT	Read Write	-	-	0		HMC3102A-M	User Defined Tag	<input type="checkbox"/>
3	Temp1F	REAL	Read Write	-	-	0		HMC3102A-M	User Defined Tag	<input type="checkbox"/>
4	Temp2F	REAL	Read Write	-	-	0		HMC3102A-M	User Defined Tag	<input type="checkbox"/>
5	RawInput	INT	Read Write	-	-	0		HMC3102A-M	User Defined Tag	<input type="checkbox"/>
6	Voltage	REAL	Read Write	-	-	0		HMC3102A-M	User Defined Tag	<input type="checkbox"/>
7	Scale/Input	INT (L)	Read Write	-	-	0		HMC3102A-M	User Defined Tag	<input type="checkbox"/>
8	Scale/Output	REAL (L)	Read Write	-	-	0		HMC3102A-M	User Defined Tag	<input type="checkbox"/>

Logic can now be entered into the block to define its functionality. The first thing we want the *Scale* logic block to do is convert the input to floating point (real) format.

- In the instruction list at the bottom right of the editor window expand the **Conversions** folder and locate the **any_to_real** instruction. Click the instruction and drag it into the editor window.

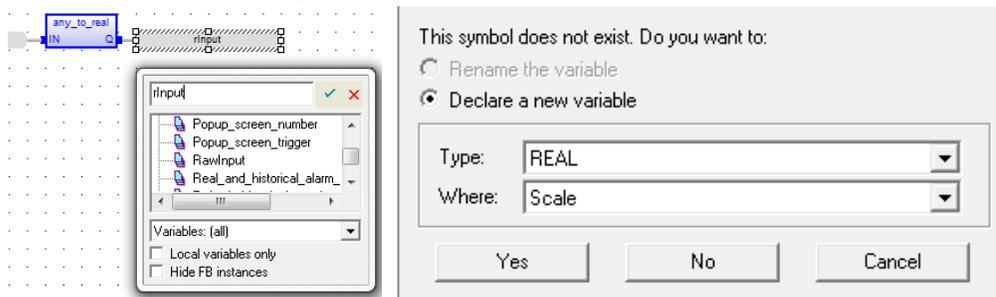


- The input to this instruction will be the UDFB's Input parameter. Double-click the instruction's input area and type *Input* into the box that pops up.



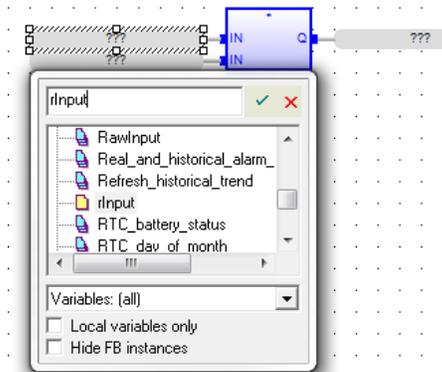
Notice that because *Input* is an Input parameter for this UDFB, it shows up in the context list with an input icon . Clicking the tag in the list will also select it for use in the instruction.

13. For the output, we want to define a new tag that will be local to the Scale function block. Double-click the output area of the **any_to_real** instruction and type *rInput* into the pop up and hit enter.
14. Because a tag with this name does not exist yet, MAPware-7000 will display a dialog box that can be used to define the tag.

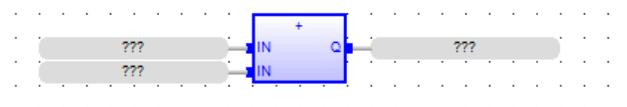


Set the type to **REAL** and **Where** to *Scale*. This sets the scope of the variable to the Scale function block.

15. Click **Yes** to create the new variable.
16. Now that the input is in the correct format, all we have to do is multiply by the slope and add the offset. Expand the **Arithmetic** folder in the instruction list and drag a **(*Multiply*)** instruction into the editor window.
17. The first input here will be the *rInput* tag created in the last step. Double-click in the first input and type *rInput* into the pop up window. Notice that *rInput* is now in the list as a local variable.

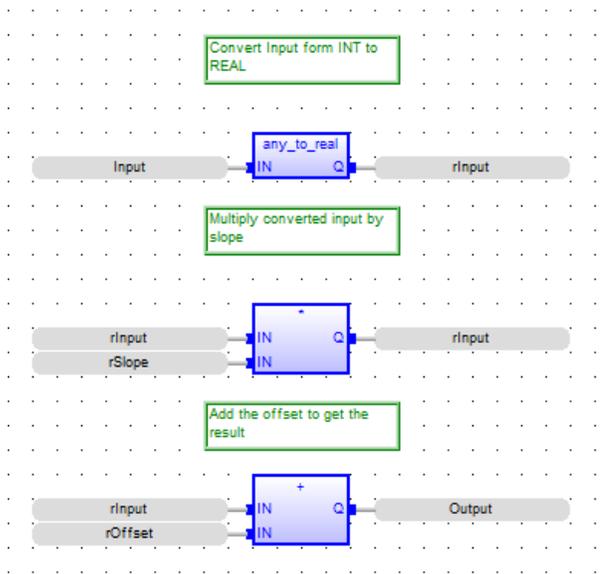


18. The second input will be a local variable called *rSlope*. This tag doesn't exist yet, so click the second input and type *rSlope* to create the new tag.
19. Set **Type** to **REAL** and **Where** to *Scale* using the context window as described above.
20. We will write the result of the multiplication back to the *rInput* variable. Select *rInput* for the output (**Q**) of the instruction.
21. The last step is to add the offset. Drag a **(*Addition*)** instruction from the instruction list into the editor window.



22. The first input for this instruction will again be the *rInput* variable. Double-click the first input and select *rInput* from the tag list.
23. The second input will be a new local variable called *rOffset*. Type *rOffset* in the box for the second input and create a new tag. Make sure the **Type** is **REAL** and **Where** is set to *Scale*.
24. Finally, the output will be the previously created output parameter of the function block. Select *Output* from the menu for the output (**Q**) tag. Notice that it will have an output icon () in the tag list.

25. Here is what the Scale function block should look like when complete (some comments were added for clarity).

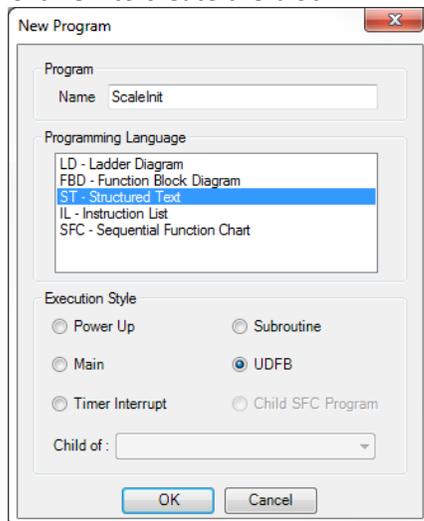


This is a good time to compile (**Project > Compile**) and save the project to make sure there are no errors before moving on.

Create a Second UDFB to Initialize Instances of the Scale Function Block

Before we can use the Scale function block, we have to initialize the offset and slope parameters. These are calculated from maximum and minimum values for the input value and the scaled value. We will create a new **UDFB** to do this calculation and initialize the function block instances. This new block will use the **ST - Structured Text** editor.

1. Right-click the **UDFB** folder in the project information window and select **New Logic Block**.
2. Name this block *ScaleInit*, and select **ST - Structured Text** for the Programming Language.
3. Click **OK** to create the block.

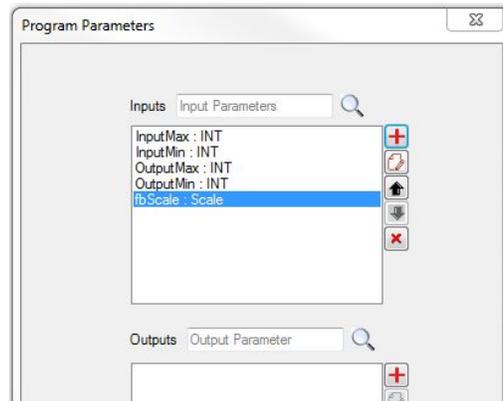


4. Next, we will define the input parameters for the *ScaleInit* function block. Right-click the block's folder in the **Project Information Window** and select **Edit Parameters** from the context menu to open the **Program Parameters** window.

5. This block will have no outputs. Enter the following inputs.

Parameter Name	Type	Description
<i>InputMax</i>	INT	Maximum value for the input
<i>InputMin</i>	INT	Minimum input value
<i>OutputMax</i>	INT	Maximum value of output
<i>OutputMin</i>	INT	Minimum value of input
<i>fbScale</i>	Scale	Function block instance to be initialized

Note. The last parameter is of type *Scale*. This means that a *Scale* function block instance will be passed into the function block. The function block can then operate on the data specific to that function block instance. The input parameters should look something like this when finished.



Now we can start entering code for the *ScaleInit* function block. This block will contain several local variables to perform the calculation. In the **Structured Text Editor**, when enter is pressed at the end of a statement or line of code, the editor will validate all of the parameters in that line. If it finds parameters that don't exist, a dialog box will pop up allowing new variables to be defined, just as in the Function Block Diagram editor. Below is a table of all the local variables to be defined.

Parameter Name	Type	Scope	Description
<i>rInputMax</i>	REAL	ScaleInit	Floating point version of <i>InputMax</i>
<i>rInputMin</i>	REAL	ScaleInit	Floating point version of <i>InputMin</i>
<i>rOutputMax</i>	REAL	ScaleInit	Floating point version of <i>OutputMax</i>
<i>rOutputMin</i>	REAL	ScaleInit	Floating point version of <i>OutputMin</i>

Below is the code to be entered into the *ScaleInit* function block. The first four lines use the **any_to_real** function to convert our input tags into real numbers. Then next line calculates the slope from the given range, and places it into the internal *rSlope* variable of the *fbScale* function block instance. The last line calculates the offset and places it in the internal *rOffset* tag of *fbScale*.

```
// Convert inputs to floating point numbers
rInputMax := any_to_real (InputMax);
rInputMin := any_to_real (InputMin);
rOutputMax := any_to_real (OutputMax);
rOutputMin := any_to_real (OutputMin);

// Calculate slope
fbScale.rSlope := ( rOutputMax - rOutputMin ) / ( rInputMax - rInputMin );

// Calculate offset
fbScale.rOffset := rOutputMax - ( fbScale.rSlope * rInputMax);
```

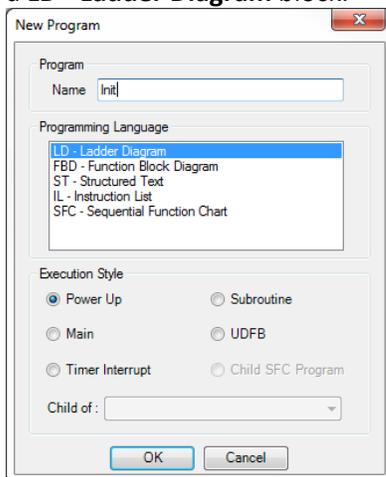
Note. If you copy and paste the code in its entirety from here into the editor, tags will not be created for the internal variables because the code was not entered one statement at a time. You must manually enter each tag into the database in this case.

6. Copy the first statement above, `rInputMax := any_to_real (InputMax);`, and paste it into the **Structured Text Editor**. The green lines beginning with `“//”` are comments and provide context, but are optional and have no effect on the code. Hit the **Enter** button on your keyboard.
7. A popup window to define `rInputMax` appears. Set the **Type** to **REAL** and **Where** to `ScaleInit`. Click **Yes**.
8. Copy the rest of the code into the **Structured Text Editor** one line/statement at a time, pressing **Enter** after each line to define the local tags. Each tag should be of **Type**. **REAL** and local to `ScaleInit`.
9. After all the code has been entered, compile (**Project > Compile**) and save the project to verify that there are no errors, and fix any typos as needed.

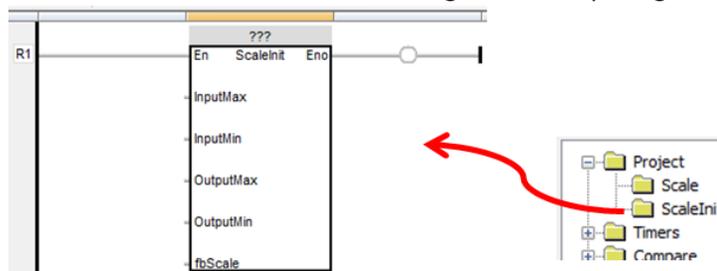
Use the UDFBs in Ladder Diagram Blocks

Now that the function blocks have been created, they can be put to use in other parts of the project. We want to use the `ScaleInit` function block in a power up routine to initialize two `Scale` function block instances.

1. Right-click the **Power Up** folder and select **New Logic Block**. This block will be called `Init` and will be a **LD - Ladder Diagram** block.

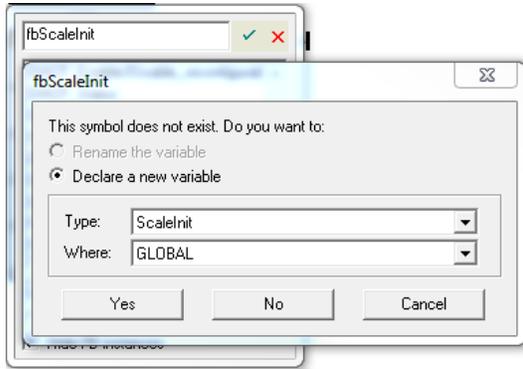


2. Expand the **Project** node in the instruction list. The `Scale` and `ScaleInit` function blocks should appear there.
3. Click the `ScaleInit` instruction and drag it to the top rung of the **Ladder Diagram**.

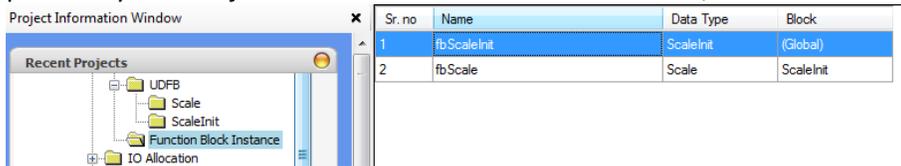


Notice the '???' at the top of the block. This indicates the function block instance has not been selected.

4. Double-click the question marks and enter `fbScaleInit` in the tag selection window. Click the Accept icon  and declare the new variable as **Type**. `ScaleInit` and **Where**. **Global**.



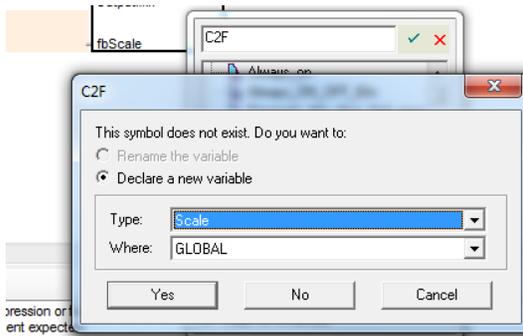
This defines a new instance of the *ScaleInit* Function Block called *fbScaleInit*. This instance will now be listed in the **Function Block Instance** folder in the **Project Information Window** along with the previously created *fbScale* instance of *Scale* defined earlier, local to *ScaleInit*.



- Next we will specify the input parameters. For this sample, we will use literal number values for the maxes and minimums. Double click to the left of each of the first four inputs and type in the following values.

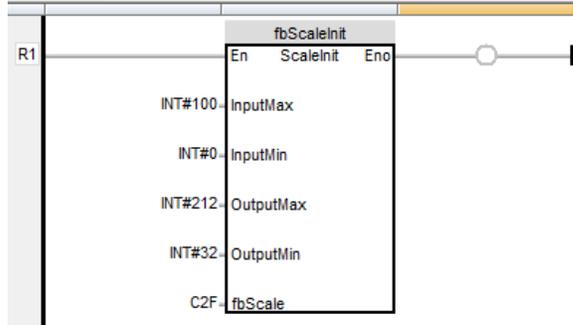
Input Parameter Name	Value	Description
<i>InputMax</i>	100	Celsius input maximum
<i>InputMin</i>	0	Celsius input minimum
<i>OutputMax</i>	212	Fahrenheit equivalent to 100°C (Max °F)
<i>OutputMin</i>	32	Fahrenheit equivalent to 0°C (Min °F)

- The last parameter is the *Scale* function block instance to be initialized. In this case it will be a new instance called *C2F* (Celsius to Fahrenheit). Double-click the *fbScale* input and type *C2F* into the selection box.
- Set the **Type** to *Scale* and **Where** to **GLOBAL** in the popup window and click **Yes** to create the *C2F* instance.



The *C2F* instance is created, added to the **Function Block Instance** Folder and selected as the *fbScale* input parameter.

This is what the instruction should look like now.



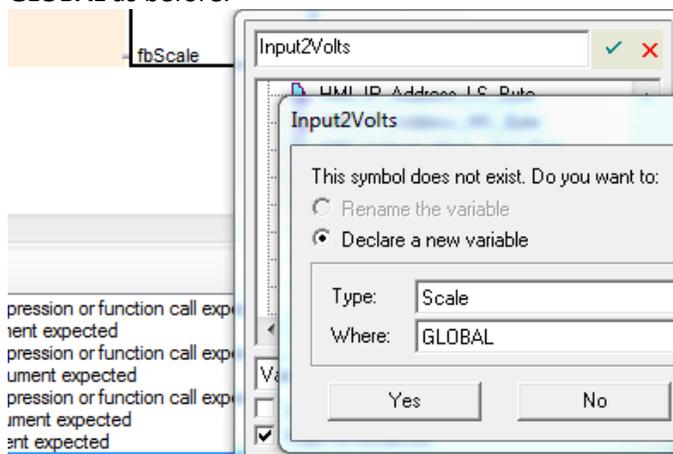
Next we want to initialize a different Scale function block instance called *Input2Volts*. This instance will be used to convert from a raw input to a voltage. The process is the same as above, but with a different function block instance as the input parameter and with different maximum and minimum values.

8. We will use the same instance of the *ScaleInit* function block. Drag a new *ScaleInit* instruction to rung 2 from the instruction list.
9. Click the '???' above the block and again enter *fbScaleInit* in the popup window. This time, the *fbScaleInit* instance will appear in the variable list since it was already defined.
10. Enter the following values for the input parameters.

Input Parameter Name	Value	Description
<i>InputMax</i>	4095	12 bit analog maximum input
<i>InputMin</i>	0	12 bit analog minimum input
<i>OutputMax</i>	5	0-5V maximum
<i>OutputMin</i>	0	0-5V minimum

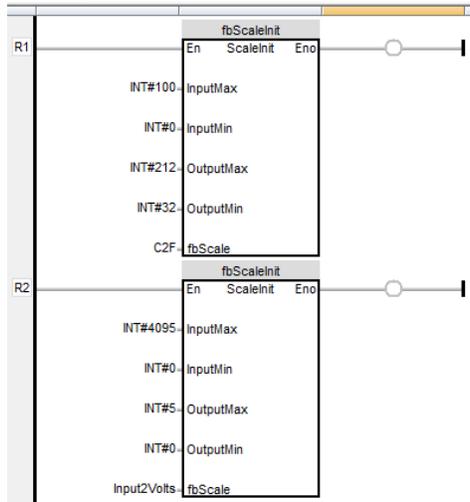
This sets up the *Input2Volts* function block to convert from a 12-bit input to a 0 to 5 V output.

11. Enter *Input2Volts* for the *fbScale* input.
12. The *Input2Volts* function block instance will need to be defined. Set the **Type** to *Scale* and **Where** to **GLOBAL** as before.



This will add a new *Scale* function block instance called *Input2Volts* to the **Function Block Instance** folder.

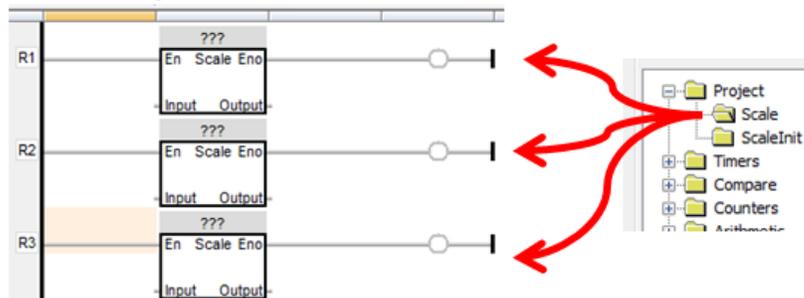
Here is what the *Init* function block should look like when complete.



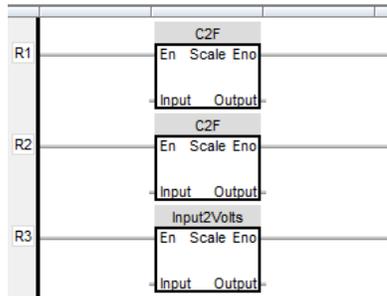
Define the Main Routine Block 1

The final piece of logic in the sample project uses *C2F* and *Input2Volts* instances to continuously convert our simulated inputs to the desired output values as the inputs change over time. We can use the ladder diagram block automatically created by MAPware-7000 to do this.

1. Click the **Block1** folder under **Logic Blocks / Main** in the **Project Information Window** to edit the block.
2. Click and drag three *Scale* blocks from the instruction list into the editor.

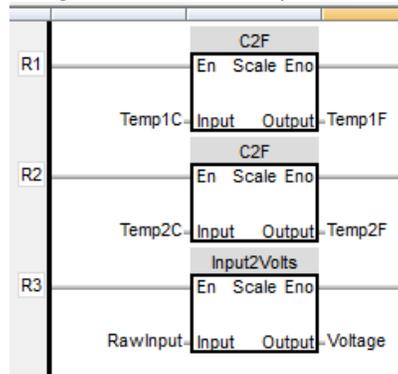


3. Next click the '???' in each block to specify the function block instance to be used. The first two will use *C2F* and the last one will use *Input2Volts*.



4. Specify the input and output parameters for each instruction. We will use the tags created at the beginning of this section.
 - Rung 1 converts *Temp1C* to *Temp1F*
 - Rung 2 converts *Temp2C* to *Temp2F*

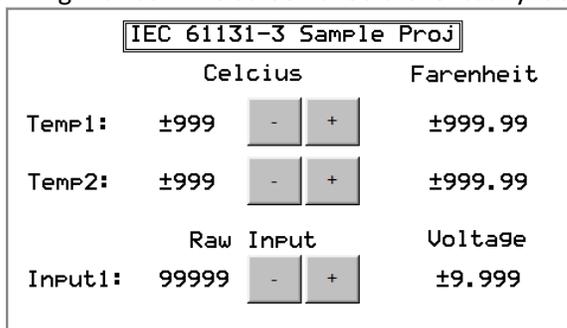
- Rung 3 converts *RawInput* to *Voltage*.



5. That's it for the logic. Check that the project compiles (**Project > Compile**) and save the project to verify that there are no errors. Fix any typos as needed.

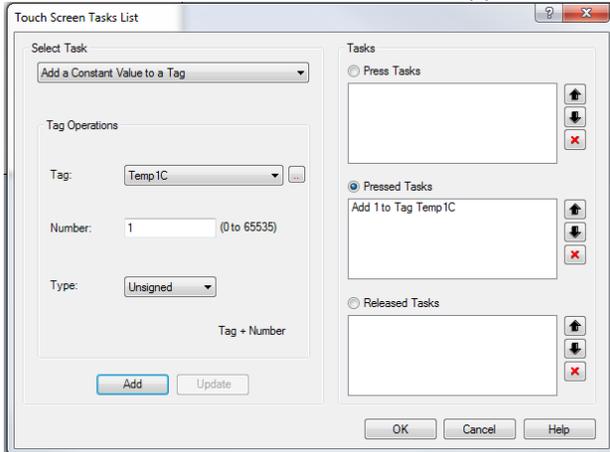
Create Screen Objects

We will create a very simple screen to control the inputs and observe how the outputs change. This manual will not go into detail on creating the objects needed. For more information refer to the MAPware-7000 Programming Manual. The screen should eventually look something like this.



1. Navigate to **Screen1** under the **Base Screens** folder in the **Project Information Window**.
2. Create three **Numeric Entry** objects (**Draw > Input Objects > Data Entry > Numeric Entry**) to display and write to the three inputs; *Temp1C*, *Temp2C* and *RawInput*.
3. Change the **Font** for these objects to **10 x 14** to make them easier to read, and select the appropriate tag in the **Tag Name** property.
4. For the *Temp1C* and *Temp2C* objects set the **Data Type** format to **Signed [-32768 To 32767]** to allow negative temperatures to be displayed correctly.
5. Next we will use **Multi-Task Single-state** buttons (**Draw > Buttons > Multi-Task Single-state**) to create the increment and decrement buttons that control the input tags. Place a button next to the first **Numeric Entry** object.
6. For the increment button, change the **On Text** property to "+".
7. Click the **Tasks** property, then on the  button to configure a task for the button.
8. Before selecting the task, click the **Pressed Task** radio button. This means that the value will increment *while* the button is pressed.
9. Select the **Add a Constant Value to Tag** task from the **Select Task** list. Select *Temp1C* for the **Tag** and enter 1 for the **Number** to add.

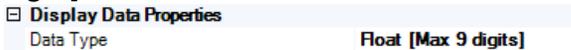
10. Click the **Add** button. The task should appear under **Pressed Tasks**.



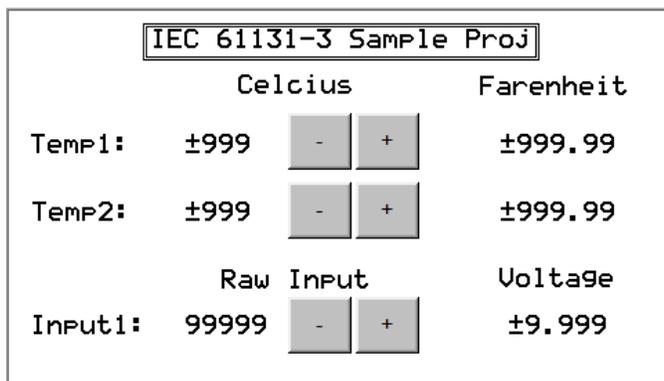
11. Click **OK** to save the task.
 12. Make a copy of the increment button to use as the starting point for the decrement button.
 13. Change the **On Text** to “-“.
 14. Change the **Task** to a **Subtract a Constant Value from Tag** task that subtracts 1 from *Temp1C*.



15. Once you have increment and decrement buttons controlling *Temp1C*, make copies of them to control *Temp2C* and *RawInput*. You can update the task with the correct tag by clicking the task in the **Pressed Tasks** list. When changing the tag, make sure the number is re-entered, and don't forget to click **Update** and **OK** to save the changes. For *RawInput*, you may want to change the add/subtract numbers to 10 or 100 so that the value changes faster.
 16. Finally add three **Numeric Display** objects to display *Temp1F*, *Temp2F* and *Voltage*. These are floating point numbers so, after you select the tag, change the **Data Type** property to **Float [Max 9 digits]**.



After adding these objects, and some text objects for labels, the screen should look something like this.



Compile and save the project one last time. It is ready to download and run on the HMC hardware. Use the increment and decrement buttons to change the inputs and see that the outputs change to the correct scaled value.

Review

Before moving on, let's review some important points about how this sample project is structured that will allow you to take full advantage of IEC 61131-3 features. Although not all of the features available in MAPware-7000 or the HMC Series have been covered, we have taken our first steps in using this software and becoming familiar with device operation.

First, notice how we used multiple instances of a User Defined Function Block. We are scaling three inputs but have defined our scaling logic in only one place, the *Scale* UDFB. If something needs to be changed, it only needs to be changed in one place, and it isn't necessary to hunt through the project to make sure we update the logic everywhere it is used. We reused a single instance of the function block to scale two different inputs and used a separate instance to scale another input on a totally different scale. We could have many more channels and many more scales, but still only need one UDFB. If there was a radically different scaling operation that needed to be done, such as a lookup table, then we would need to create a different UDFB.

Next, note that three different editors were used, and it is possible to call the logic created in one editor from logic created in another. The different languages each have their own strengths. Ladder Diagram provides a clear graphical representation of the logic flow, Function Block Diagram is great for combining operations in a simple to read structure, and Structured Text can be used for more involved operations that might look quite complicated in one of the graphical editors. By combining logic from the different editors we are able to take advantage of the strengths of each.

Finally, note that we were able to separate the logic for initializing the scale block (UDFB-ScaleInt) and actually performing the scale operation (UDFB-Scale) into two separate operations. We did this by passing a UDFB instance as a parameter to another UDFB. This keeps the scaling operation, which occurs frequently while the project is running, as simple and quick as possible. Breaking complex operations into simple building blocks is another way to make a project more maintainable.

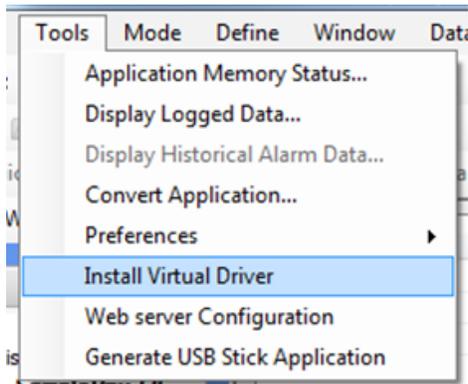
Chapter 3 – Online Monitoring and Logic Simulation

Online Monitoring

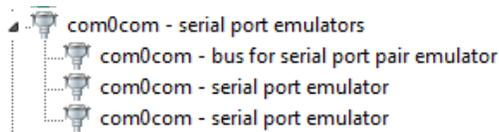
In HMC products, the HMI screen provides a window into what is happening in the HMC; however, for a complicated project with many logic blocks it is often not enough to debug the logic. Online Monitoring allows the programmer to view the logic in real time, and modify data directly in logic blocks. For MLC models, without a screen, this is the only way get a picture of what the logic is doing.

Installing the Virtual Com Port driver

MAPware-7000 requires an additional communication driver, the virtual com port driver, be installed before starting an Online Monitoring session using IEC 61131-3 mode over USB. To install the driver go to **Tools > Install Virtual Driver:**

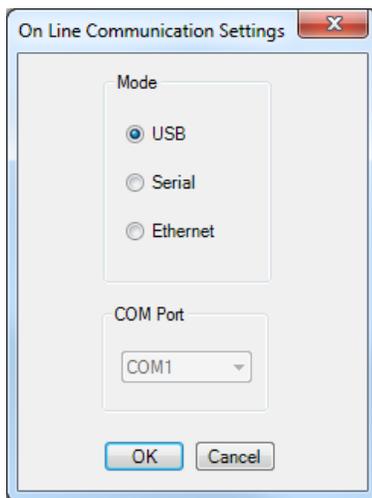


This driver will add several ports in the development PC's Device Manager:



Going Online

With the drivers in place, the HMC/MLC can be monitored with MAPware-7000 while it is executing a program. The connection to the unit can be made using USB, Serial or Ethernet. To select the connection type got to Tools > Preferences > Online Communication Mode.



Note: For Ethernet mode, the IP address of the device must be entered in this window. The IP address here should match what is configured on the Ethernet tab of the Project Properties Dialog. The address of the

device is only updated when the Ethernet Settings box is checked in the download window. The current IP address is shown on the HMI screen when the device is booting up.

There are three options for initiating an online monitoring session:

- With Download – the currently open project will be downloaded to the device before the online monitoring session begins
- Without Upload – the online session will begin with the project that is open and with the project that is in the device. This assumes that the currently open project is the same as the project on the device.
- With Upload – The currently open project will be closed. The project on the device is uploaded and opened, before the online session begins. This is the only option available when no project is opened.

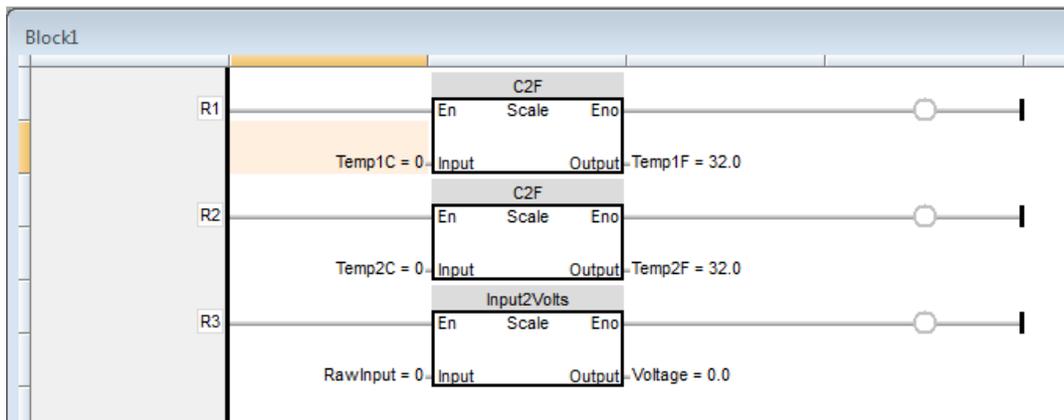
These options are available by selecting Mode > Go Online from the menu. Note: clicking the online icon



is equivalent to selecting Without Upload from the Go Online menu option.

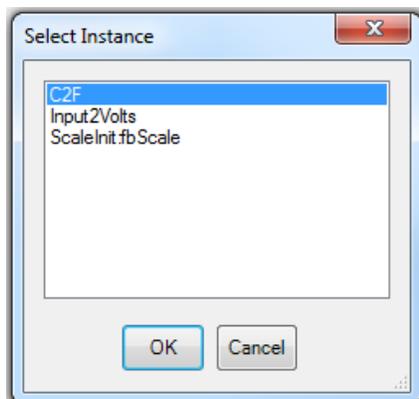
Selecting Logic Blocks and Function Block Instances

Once the online session begins, any open logic blocks will have variables loaded with their real time values:

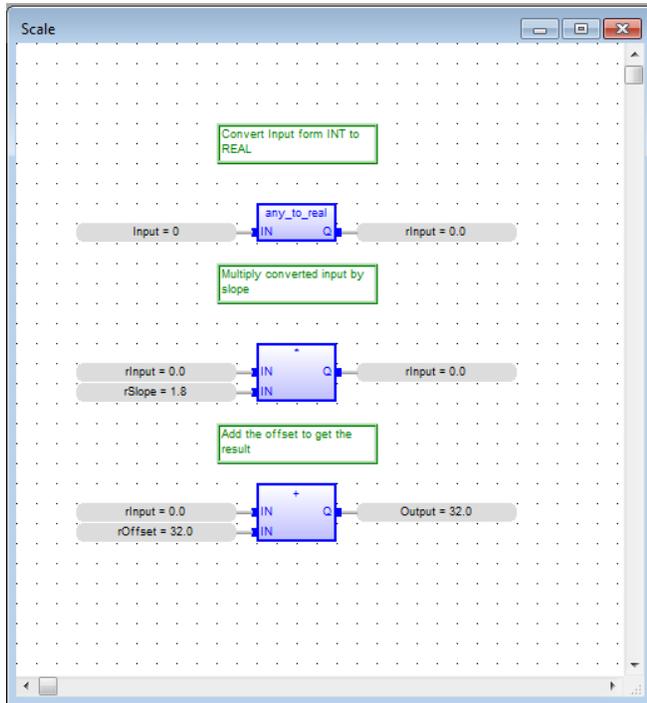


Logic blocks can be opened for monitoring by selecting them in the project tree.

When opening a Function Block that has multiple instances, MAPware-7000 needs to know which instance to open. Thus, when a UDFB that has multiple instances is selected, a popup window will appear listing the available instances of that function block to monitor:

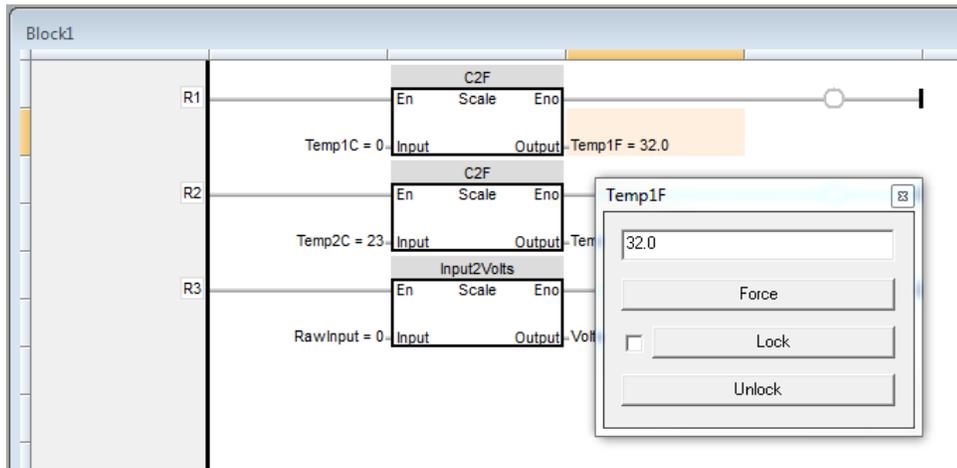


Once the selection is made the function block will be loaded with the real time data for the selected instance:

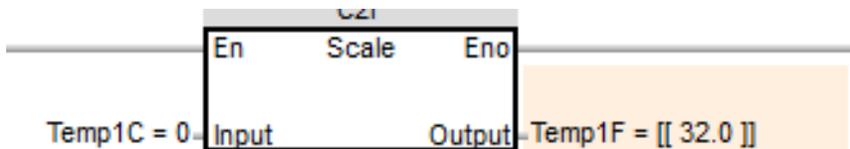


Manipulating Data

To change the value of a parameter simply double-click the parameter:



The popup window allows you to force and or Lock/Unlock the value of the parameter. When a parameter is forced, it is written to once. If some other logic writes to the parameter after it is forced, the forced value will be overwritten. If the value is locked, MAPware will prevent any other logic in the block from overwriting the forced value. Parameters that are locked will appear in double square brackets.



The lock only prevents logic blocks from writing to the value. The value can still be changed by a task, or by entering data in a numeric object on an HMI screen.

Data Monitor Window

In addition to viewing parameters in logic blocks it is often convenient to see data in a spreadsheet view. This allows you to see data that might be related, but might not appear in close proximity in the logic block. Data can also be changed to see how a change in one data point affects other data points. To access the Data Monitor window, select View > Data Monitor Window. The Data Monitor Window can only be configured when MAPware-7000 is in offline mode. To add a tag to the Data Monitor Window, right-click the Data Monitor Window in offline mode and select Add Variables from the context menu. Data from Function Block Instances cannot be added to the Data Monitor Window.

More details on using the Data Monitor Window can be found in the MAPware-7000 Programming manual.

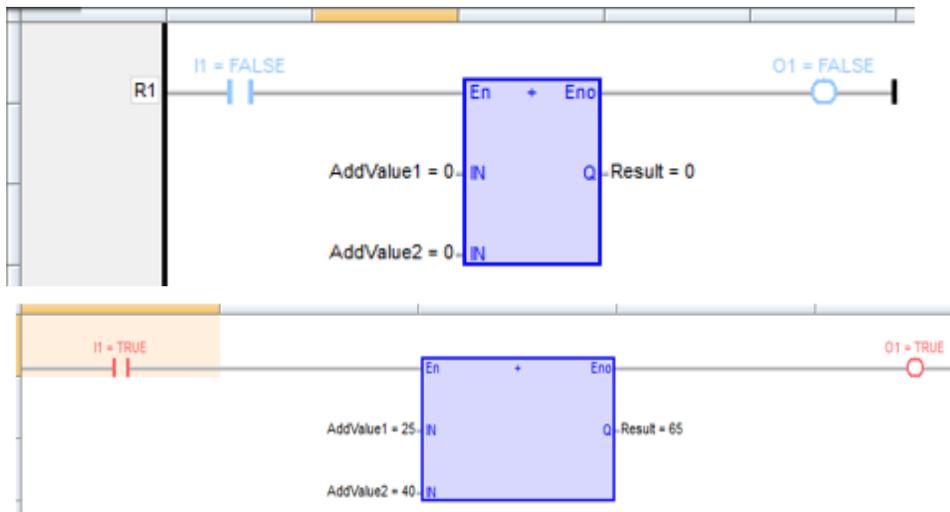
Simulating IEC 61131-3 Logic on a PC

In addition to monitoring logic as it is executed in a device, MAPware-7000 can execute the logic on the development PC using simulation mode.

To simulate a particular logic block, open that block, then select Mode > Start Simulation > Logic Only. The open logic block will be displayed with live values shown for each tag. The appearance of the logic block will vary according to which language type is being simulated.

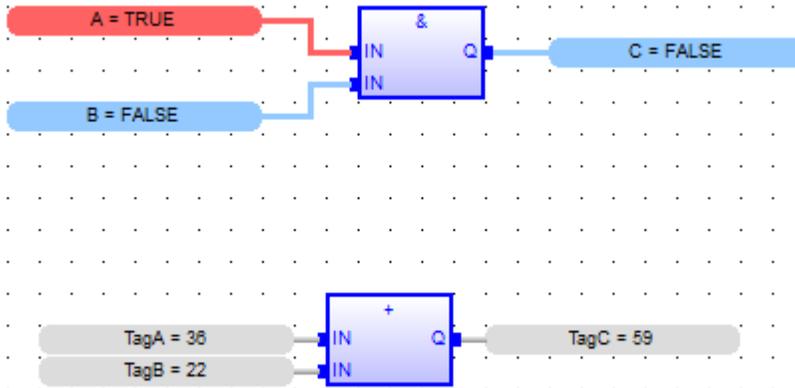
Simulating Ladder Diagram

When simulating LD logic blocks, the current value of variables used as inputs are shown next to the tag where they appear in the logic. Contacts and coils are shown color-coded red for true and blue for false:



Simulating Function Block Diagram

Function Block Diagram Blocks are also shown with tag values next to the tag labels. For boolean values, true is color-coded red and false is blue.



Simulating Structured Text

In structured text logic blocks, you must toggle the Show Value in Text  quick select button to show current values for variables:

```

// ST (Structured Text) Logic Block
O1 FALSE := I1 FALSE AND I2 FALSE ;
O2 FALSE := I3 FALSE OR I4 FALSE ;

IF THEN
  Area 0.0 := POW( Length 0.0 , 3 );
  WordC 0 := WordA 0 AND WordB 0 ;
b11
FOR Counter 1 := MinVal 0 TO MaxVal 0 DO
  Sum 0 := Sum 0 + Result 0 ;
  Counter 1 := INC( Counter 1 );
END_FOR;

IF FALSE Temp 0 > 100 THEN
  MyString 'Temp is Low' := 'Temp is High';
ELSIF Temp 0 < 30 THEN
  MyString 'Temp is Low' := 'Temp is Low';
ELSE
  MyString 'Temp is Low' := 'Temp is Normal';
END_IF;

```

Simulating Instruction List

Instruction list blocks appear similarly to Structured Text blocks in the simulator. Again, the Show Value in Text button  must be pressed to display the live values.

```

LD Real1 10.0
POW Real2 3.0
ST Real3 1000.0

IF THEN
LD( Int1 14 (* in ST: Int5 := (Int1 + (Int2 * Int3)-Int4); *)
ADD( Int2 5
MUL Int3 19 )
SUB Int4 3 )
ST Int5 106

LD Word1 22056 // HI
MAKEDWORD Word2 32457 // LO
ST DWord1 1445494473 // Q

LD Sint1 0 // shl( IN(*ANY*), NbS(*ANY*) )
shl Sint2 2
ST Sint3 0

Start: LD Sint1 0 // Sub if Sint1 < 0, Add if Sint1 = 0, Mul
> 0
JMPC MultiplyVars
LD Sint1 0
< 0
JMPC SubtractVars // jump to SubtractVars if Bool1 is TRUE

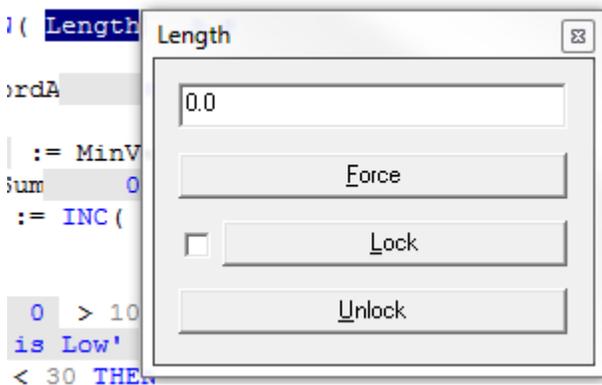
AddVars: LD Int1 14 // otherwise, add the two variables
ADD Int2 5
ST Int3 19
JMP TheEnd

SubtractVars: LD Int1 14 // subtract Int2 from Int1

```

Controlling Values in the Simulator

No matter which language is used, values can be manipulated in the simulator window. To change a value simple double-click the tag or element to get an entry box:



The options here are the same as described in the [online monitoring](#) section above.

Chapter 4 – Ladder Diagram (LD)

Overview

The Ladder Diagram editor provides a graphical representation of the logic block in the form of a schematic of an electro-mechanical circuit. The diagram consists of vertical left and right power rails and logic in the form of horizontal rungs that connect the left power rail to the right. The programmer constructs the logic by adding circuit elements to the horizontal rungs.

There are three main types of elements that can be placed on a Ladder Diagram logic rung:

- **Contacts** – These represent the contact terminals of an electro-mechanical relay. They have two states represented by the state of a Boolean tag. In a contact the state of the tag controls the state of the element, therefore these are generally used as inputs controlling the state of elements placed further to the right along the rung.
- **Coils** - These elements are represented by the coil of an electro-mechanical relay. They also have two states represented as the state of a Boolean tag. In a coil the state of the elements controls the state of the tag, therefore coils are used as outputs of the logic rung, controlled by elements placed to the left closer to the power rail.
- **Function Blocks** – These are more complex operations. They can have several input tags of various types and several output tags of various types. They can also contain quite complicated internal logic as in the case of subroutines and UDFBs.

Power flows through the circuit diagram from the left power rail, through the logic of the rungs, to the right power rail. Contacts and function blocks placed on the left side of a rung (closer to the left power rail), conditionally determine whether power flows to contacts coils and function blocks placed further to the right on the rung. If power is allowed to flow to a coil or function block, that coil or function block is said to be energized. When energized, an output's state is set according to the style of the output. When a function block is energized, the function block operation is executed. If the power is blocked from reaching an output or function block, that output or instruction is said to be de-energized. When de-energized, an output will be set according to the style of the output. A de-energized function block will not be executed and its outputs will not be updated.

Just as in a real circuit, elements can be connected together either in series (one after the other on the same conductor with no branches or junctions), or in parallel (each connected to its own branch from the same junction point).

When contacts are connected together in series this creates a logical AND operation. All of the contacts have to be closed in order for power to flow to downstream circuit elements. When contacts are placed in parallel this creates a logical OR operation. When any one of the contacts is closed power will flow to downstream circuit elements.

Coils and function blocks can also be connected in parallel allowing a single input condition to control several outputs. Series and parallel connections can be combined to create arbitrarily complex logic networks.

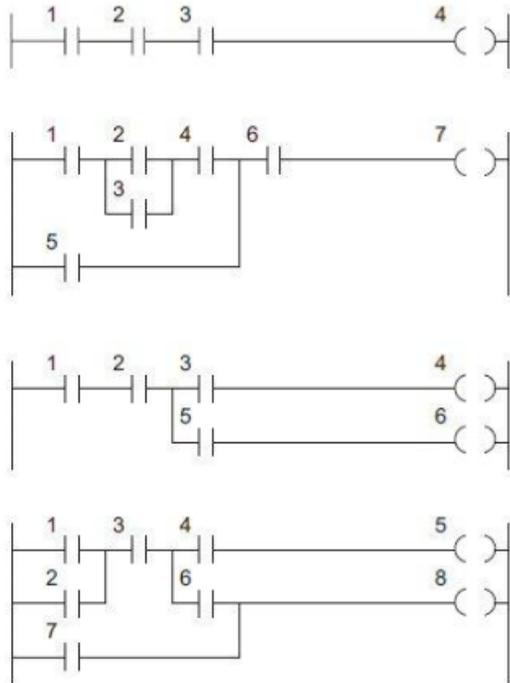
Execution Order

In a real circuit, when a junction point is energized all of the circuit elements attached to that junction point are energized simultaneously. In a Ladder Diagram program this is not the case. The processor can only evaluate one circuit element at a time. The order in which this execution occurs will affect how the program functions. The rules for the order in which circuit elements are evaluated are as follows:

1. Rungs are executed from top to bottom
2. When there is no parallel connection, each element on the rung is executed from left to right.

3. When a junction point is encountered, each branch of the junction is evaluated in turn, from left to right starting with the top branch and moving towards the bottom branch.
4. When a junction point is encountered to the right of a circuit element, elements to the right of the junction point are not evaluated until all branches to the left have been evaluated. Branches are evaluated left to right, top to bottom.

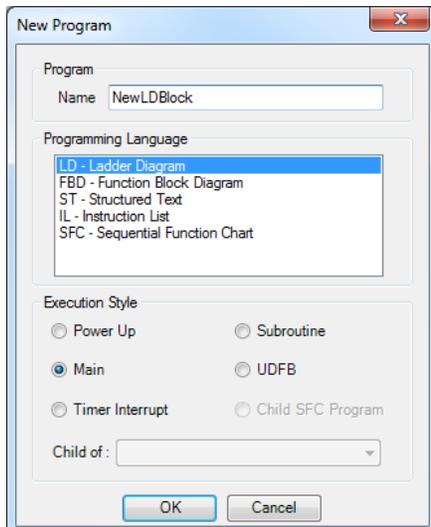
This graphic gives some examples of these rules in practice:



Note: the numbers next to each element indicate the sequence in which they are executed in the program

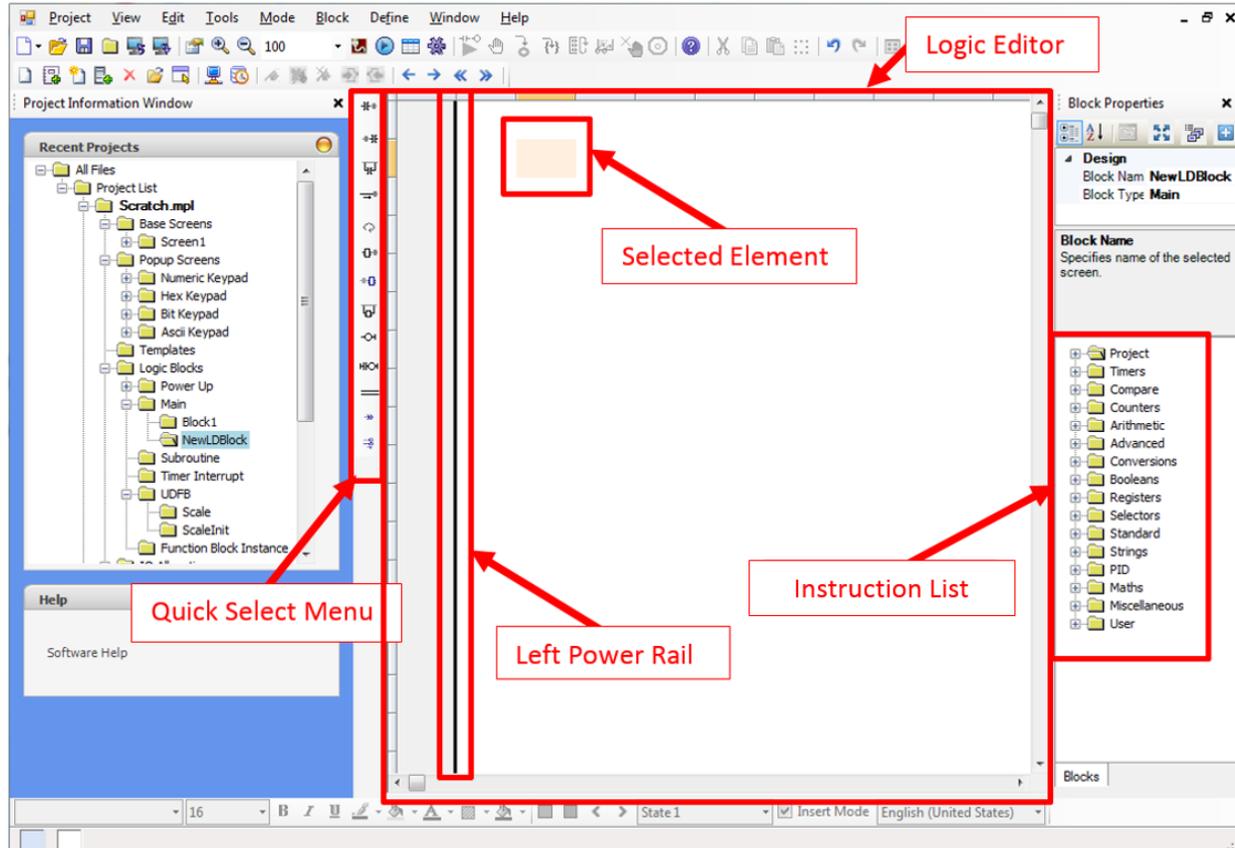
Creating a Ladder Diagram logic block

To create a new LD logic block, right-click the folder of the project tree for the desired Execution Style (see the [section on Execution Styles in Chapter 1](#) for more information). Enter a Name for the logic block, select LD-Ladder Diagram for the Programming Language, and click OK.



The Ladder Diagram Editor Window

Once the block is created the Ladder Diagram Editor Window will be displayed.



The important elements of this window highlighted in red above are:

- **Quick Select Menu** – provides a convenient way to quickly add contacts, outputs and function blocks to a rung.
- **Left Power Rail** – represents the source of power of all instructions added to a rung. In order to energize an instruction or output it must be connected to this power rail. The right power rail is not shown as a continuous vertical rail and is assumed to be connected to the right side of the rightmost element on a rung.
- **Logic Editor** – contains all of the rungs/instructions in the Ladder Diagram logic block.
- **Selected element** – the active element that will receive input from user actions on the Quick Select Menu or the keyboard. Note that the space available for the selected element can be adjusted by clicking and dragging the edge of the editor column or row:



- **Instruction List** – A list of available Function Blocks that can be added to the Ladder Diagram. Click and drag an instruction into the Editor Window to use it on a rung.

Adding Logic to the Block

For the logic block to do anything it must contain rungs with contacts (inputs), coils (outputs) and or function blocks. There are two main ways to add elements to the block:

- Using the Quick Select Menu
- By dragging instructions from the Instruction List

Using the Quick Select Menu

The options in the Quick Select Menu are:

-  **Insert Contact Before** - Inserts a contact on the rung before the selected element.
-  **Insert Contact After** - Inserts a contact on the rung after the selected element.
-  **Insert Contact Parallel** - Inserts a contact that is in parallel with the selected element.
-  **Insert Horizontal Line** - Inserts a horizontal line at the selected element.
-  **Swap Item Style** - Changes the execution style of the highlighted contact or coil. The options cycle each time you click the Swap Item Style icon. The available options are explained in the [Changing Contact and Coil Execution Style](#) section below. Pressing the keyboard space bar is equivalent to selecting this item.
-  **Insert FB Before** - Inserts a function block before (to the left of) the selected element
-  **Insert FB After** - Inserts a function block on the rung after (to the right of) the selected element
-  **Insert FB Parallel** - Inserts a function block on a branch parallel to the selected element.
-  **Insert Coil** - Inserts an output coil at the end of the selected rung.
-  **Insert New Rung** - Inserts a new rung with one contact and one coil above the rung of the selected element.
-  **Insert Comment** - Inserts a comment line above the area selected. See the section on [Comments](#) for more information.
-  **Insert Jump** – Insert a jump instruction. See the section on [Jumps](#) for more information.
-  **Align Coils** - Positions all of the output coils so that they are vertically aligned

Changing Contact and Coil Execution Style

As with real relays, there are several configurations available for contacts and coils in the Ladder Diagram editor. To change the execution style of an element, select that element in the ladder diagram editor window and click the Swap Item Style icon in the Quick Select menu, or press space on your keyboard.

Contact Execution Styles

The available execution styles for contacts are:

-  **Normally Open Contact** - If the value of the tag controlling the contact is 0 (false) no power flows through contact. If the value of the tag controlling the contact is 1 (true) power flows through contact.
-  **Normally Closed Contact** – If the value of the tag controlling the contact is 1 (true) no power flows through contact. If the value of the tag controlling the contact is 0 (false), power flows through contact.

-  **Positive (Rising Edge) Pulse Contact** - if the value of the tag controlling the contact transitions from 0 (false) to 1 (true), power flows through contact for one scan only.
-  **Negative (Falling Edge) Pulse Contact** - if the value of the tag controlling the contact transitions from 1 (true) to 0 (false), power flows through contact for one scan only.

Coil Execution Styles

The available execution styles for coils are:

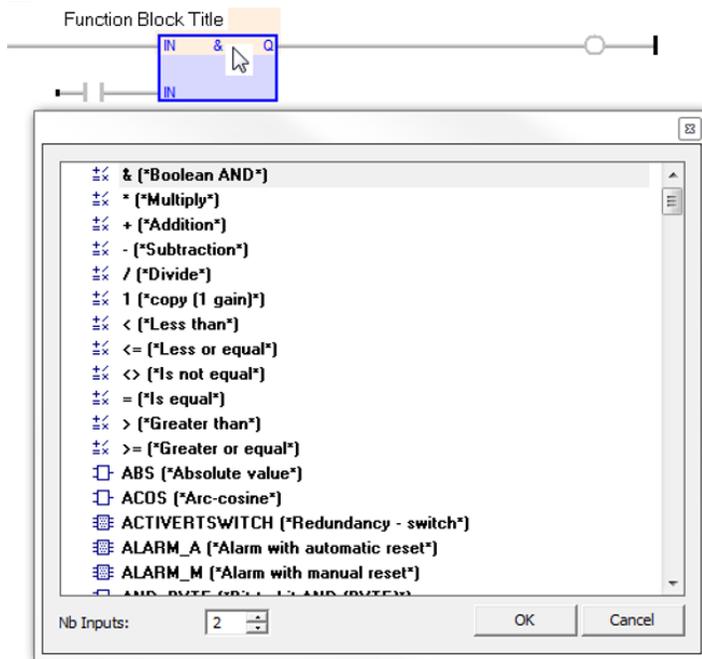
-  **Normal Coil Output** – The tag controlled by the coil is set to 1 (true) as long as power flows to the coil. The tag is set to 0 (false) when power does not flow to the coil.
-  **Inverted Coil Output** – The tag controlled by the coil is set to 0 (false) as long as power flows to the coil. The tag is set to 1 (true) when power does not flow to the coil.
-  **Set Coil Output** – The tag controlled by the coil is set to 1 (true) when power flows to it. The value of the tag will stay at 1 even after power no longer flows. This is also referred to as a latch instruction.
-  **Reset Coil Output** – The tag controlled by the coil is set to 0 (false) when power flows to the coil. The value of the tag will stay at 0 (false) when power no longer flows to the coil. This is also referred to as an unlatch instruction.
-  **Positive Pulsed Coil Output** – The value of the tag controlled by the coil is set to 1 (true) for one scan only when the power that flows to the coil transitions from off to on.
-  **Negative Pulsed Coil Output** – The value of the tag controlled by the coil is set to 1 (true) for one scan only when power that flows to the coil transitions from on to off.

Configuring Function Blocks

Function Blocks will need to be configured after they are placed. You may need to select a different function block, select a function block instance to use, or simply configure the function block inputs and outputs.

Changing the type of a function block

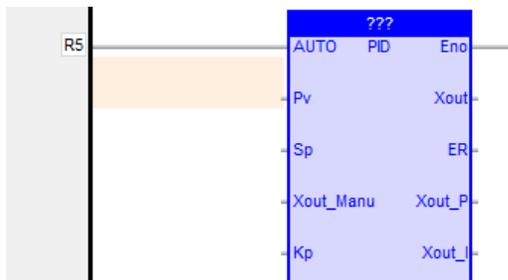
When using the Quick Select Menu to place a function block, MAPware-7000 will always default to using the AND function block. To select a different block, simply double-click the block and use the popup list select the desired block.



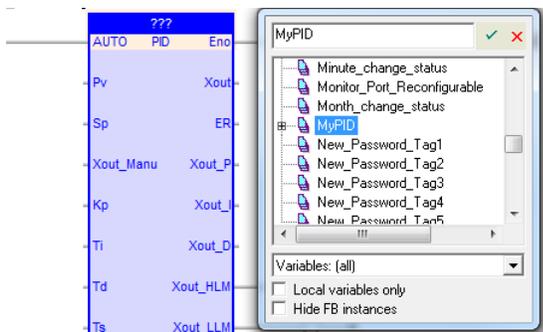
For blocks with variable numbers of inputs (ex. AND, OR, Multiply) this window can also be used to change the number of inputs. If the selected function block does not support this feature, the **No Inputs** selection box will be disabled.

Selecting a Function Block Instance

Some function blocks require that an instance of the function block is added to the project before the block is used. If this is the case, a set of question marks will appear above the function block.



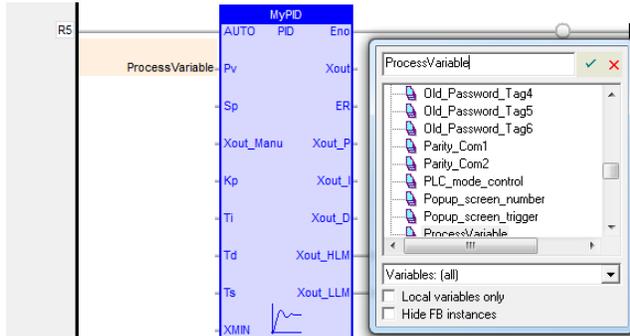
Double-click the question marks and select the desired instance from the popup window.



If there is no instance of this block created yet, simply type the name of the instance you want to create in the search field and hit enter. The instance will be created with the correct type and selected for use in this ladder element.

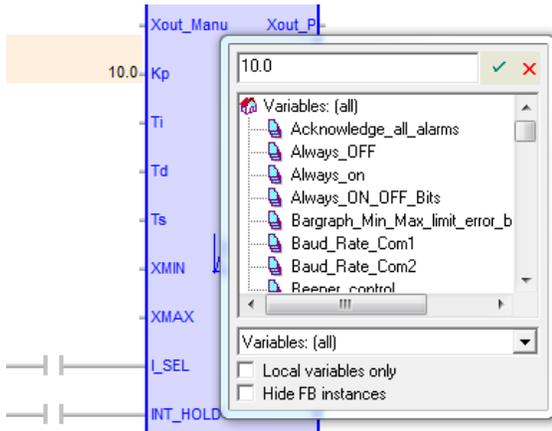
Configuring Input and Output Parameters

Function blocks can have many input and output parameters. To configure a parameter, simply double-click the input or output and select the desired tag from the popup window.



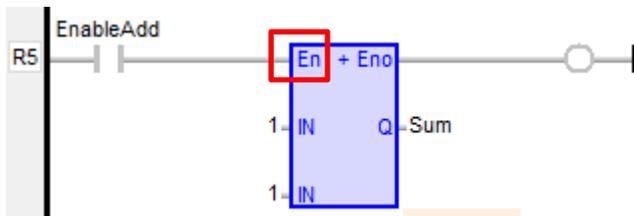
If a suitable tag has not yet been created, you can type the name of a new tag and hit enter. A popup window will appear, allowing you to configure the tag. When finished, the tag is added to the tag database.

You may also use literal values (static numbers that don't change) as inputs to function blocks, by typing the value in the tag selection popup window.



Function Block Enabled and Enabled Output Fields

As mentioned in the overview section, Function Blocks are represented as circuit elements that do more complicated functions than simple inputs and outputs. For the operation to take place, the Function Block needs to be attached to an activated rung (one where power is able to flow from the left power rail). The point of attachment is the Enable (En) input.

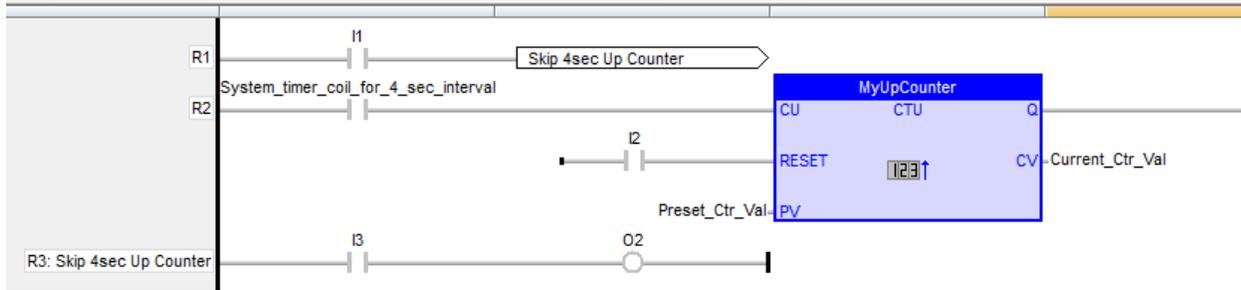


The logic in the function block will not be executed unless this input is true.

Function Blocks will also have an Enable (Eno) output. This allows an output coil to be activated, indicating that the function block is active. This also allows multiple function blocks to be connected in series.

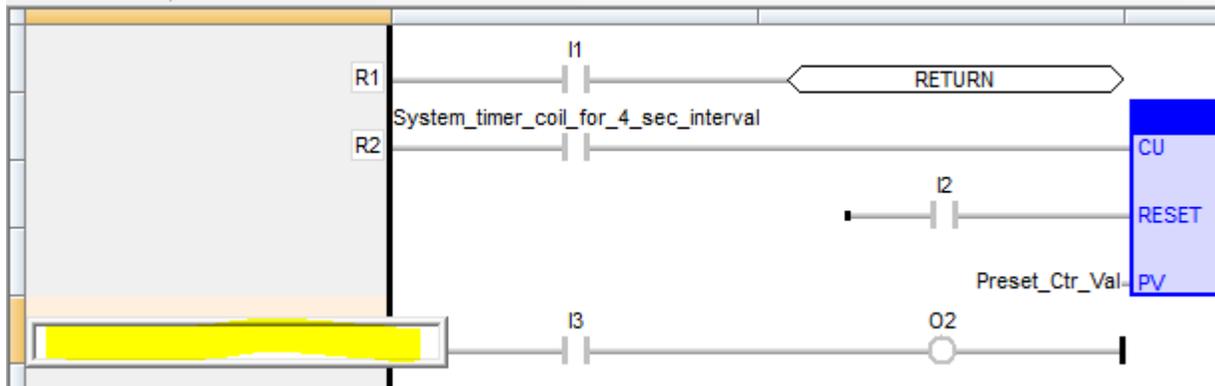
Jumps

A jump is a simple way to skip a section of logic. Here is a simple example:

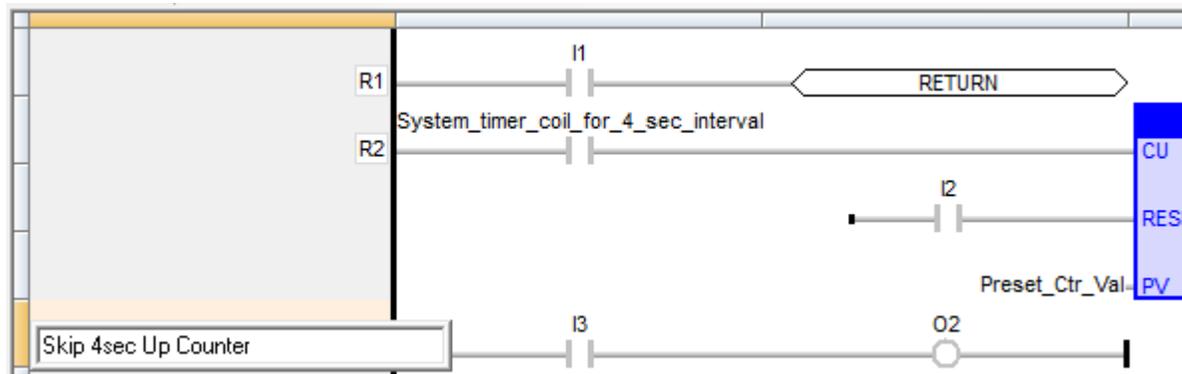


The Up Counter on Rung #2 increments every 4 seconds until it reaches the preset value. To temporarily halt operation of the Up Counter, we could use the Jump command, which is on Rung #1. When input I1 is on, the Jump command forces the HMC/MLC to jump to Rung #3 (Skip 4sec Up Counter), thus not executing any code between (in this case, Rung #2).

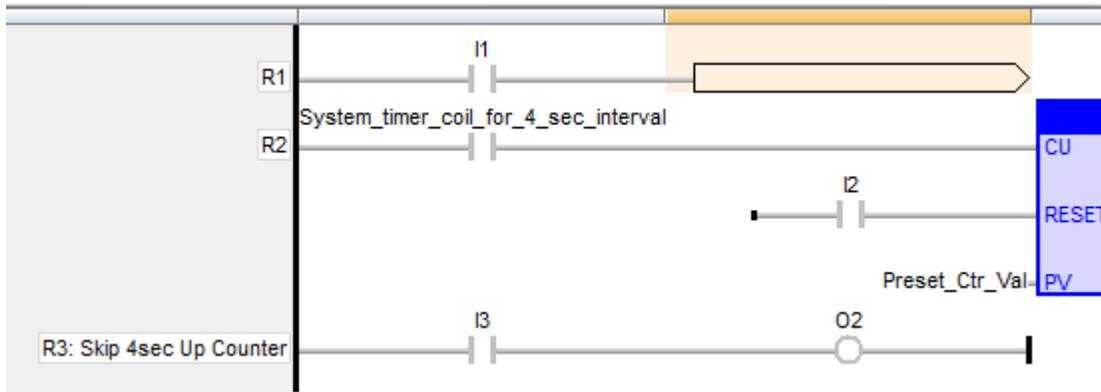
To implement a jump command, you must first assign a label to the rung that the code will jump to. Double-click in the grey column area (left of the logic code area) on the rung that needs a label:



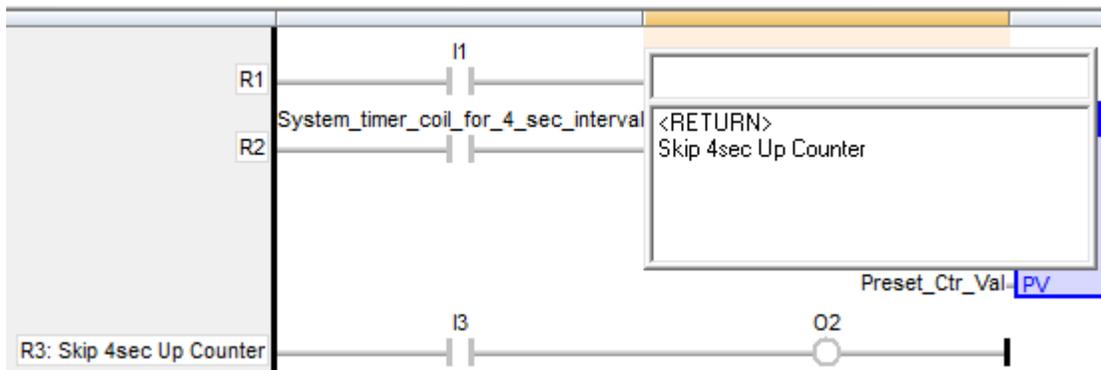
Enter a unique label name.



Then press Enter. Now insert a jump command on an existing or new rung:



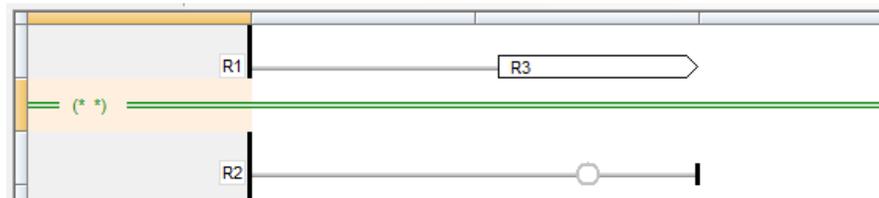
Double-click the jump command to display a popup window that shows all of the current rung labels (as well as a Return command):



Double-click the appropriate rung label.

Comments

Comments are a great way to document a logic block. To add a comment simply click the add comment option in the Quick Select Menu. This will put a single line comment above the selected element.

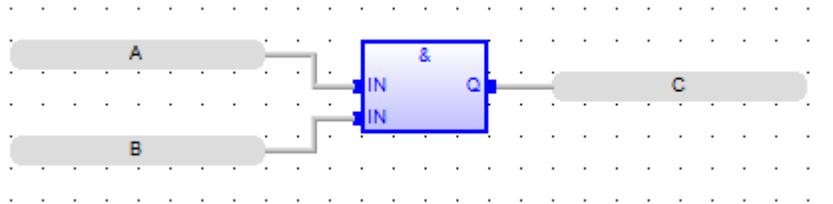


To fill in the comment double-click anywhere on the comment line and enter the comment text.

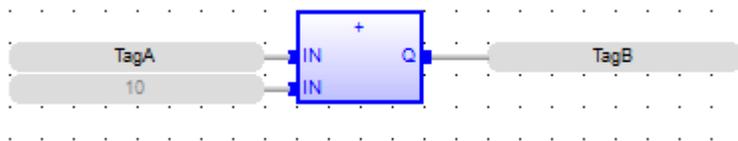
Chapter 5 – Function Block Diagram (FBD)

Overview

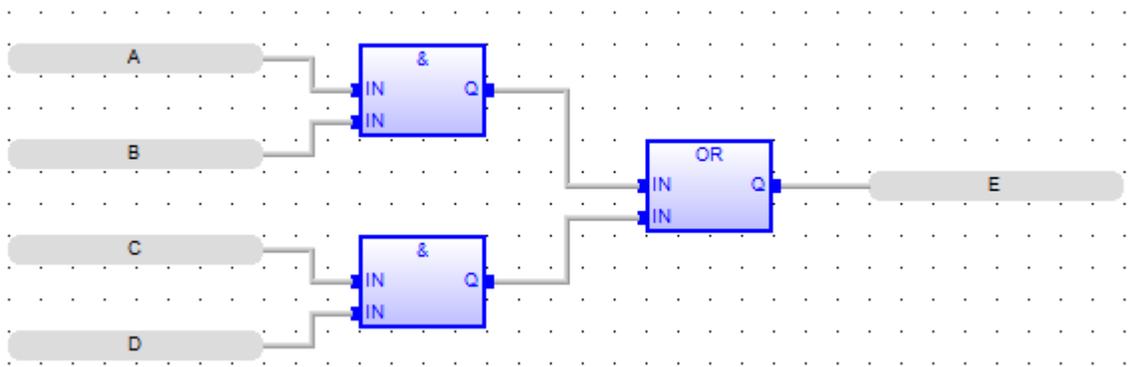
Function Block Diagram is a graphical programming language in which program operations are represented as labeled, rectangular blocks. Input and output parameters are shown as labels attached to the blocks. Inputs are attached to the left and outputs to the right.



Inputs can be tags or literal values.



Function Blocks can be connected together so that the outputs of one block become the inputs of another. This allows the programmer to build complex operations out of simple building blocks.



Function Block Diagram vs. Ladder Diagram

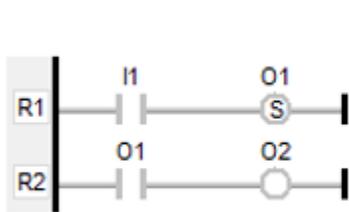
Unlike LD programming, FBD programming does not require the use of power rails for the Function Blocks:



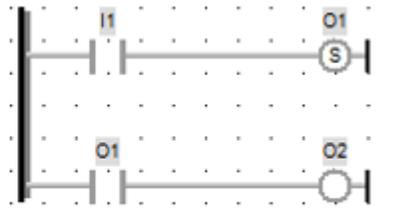
Therefore, with FBD programming, there are no En (Enable or activate instruction) or Eno (Enable output) connections. This means that each Function Block (instruction) is always executed in FBD programming (unless skipped). For this reason, FBD programming is most often used when the order of execution is not as important. Rather, the focus is on creating function blocks (instructions) that process data when the logic block is executed. There is much less emphasis on enabling/disabling functions using contacts in FBD.

However, the FBD editor does contain left power rail, contact and coil objects that can be added to the program using the quick start menu. When using power rails, contacts and coils, both programming languages are very similar:

- every contact requires a connection to the left power rail
- every contact requires that the output be connected to another contact, instruction or coil

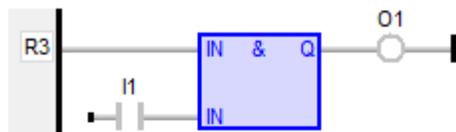


LD contacts and coils

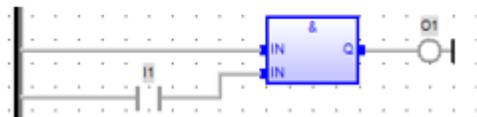


FBD contacts and coils

You can connect a contact directly to the input of a Function Block that requires a Boolean value:

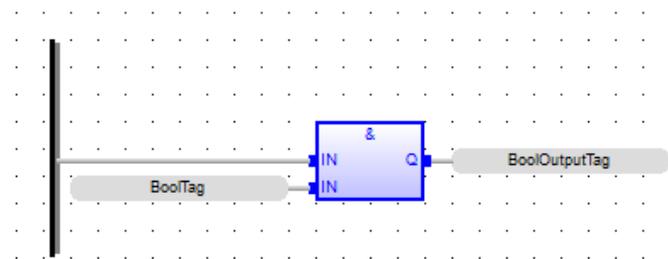


LD contact to instruction



FBD contact to instruction

This is equivalent to using a Boolean tag as the input:



In LD programming, instructions are executed starting with the top rung, then the next rung, and so on. Since FBD programming doesn't require rungs, all objects are executed from Left to Right and Top Down.

Adding Logic to the Block

There are two main ways to add elements to the block:

- Using the Quick Select Menu
- By dragging instructions from the Instruction List

Function Block elements have input(s) on the left side and an output(s) on the right side. *When connecting elements, you can only connect one object's output to another object's input.*

For example, suppose you wish to connect a contact to a coil.

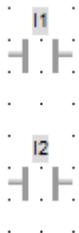


To connect the two, you would simply select the Arc Tool  and then connect the two objects:

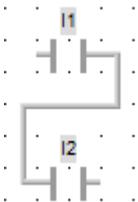


As stated above, if you do this by clicking the input I1, then move to the output O1, you will have no problem connecting the two. However, if you try clicking the output O1, then move to input I1, they will not connect. Why? Because the software is assuming that you want to connect the output of the coil O1 to the input of the contact I1- which is not allowed. Generally speaking, you will find it is much easier to connect objects if you start from the left side and move to the right. *Also, don't forget to connect the input of I1 above to the left power rail.*

Similarly, you might want to connect two contacts in parallel:



You might try to do this by simply clicking the output of I1 then clicking the output of I2. This is what you would get:



Why? Although you are correctly starting with the output of I1, the software thinks you want to connect to the input of I2. To create a parallel connection, you must use the OR bar (more on this tool later).

Note: When adding components to a FBD block, *open connections* will cause a compiler error.

Quick Select Menu

Options available in the FBD editor Quick Select Menu are:

Selection

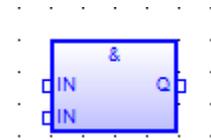
The selection tool is used to select one or more objects already on the editor work area to delete, move, highlight, etc. Note: when using any of the quick select options below, you will notice that your mouse cursor remains defined as that quick select option even after you have used it to place the object in the work area. For example, after placing a function block on the work area using the Add Function Block

option, the mouse cursor continues to be defined as an Add Function Block so that you can easily place additional function blocks on the work area.

To be able to select an existing object in the work area, you must click the Selection quick select option. Another way to do this is move the mouse cursor to a blank area in the work area, then right-click the mouse. This will change it back to the Selection quick select option.

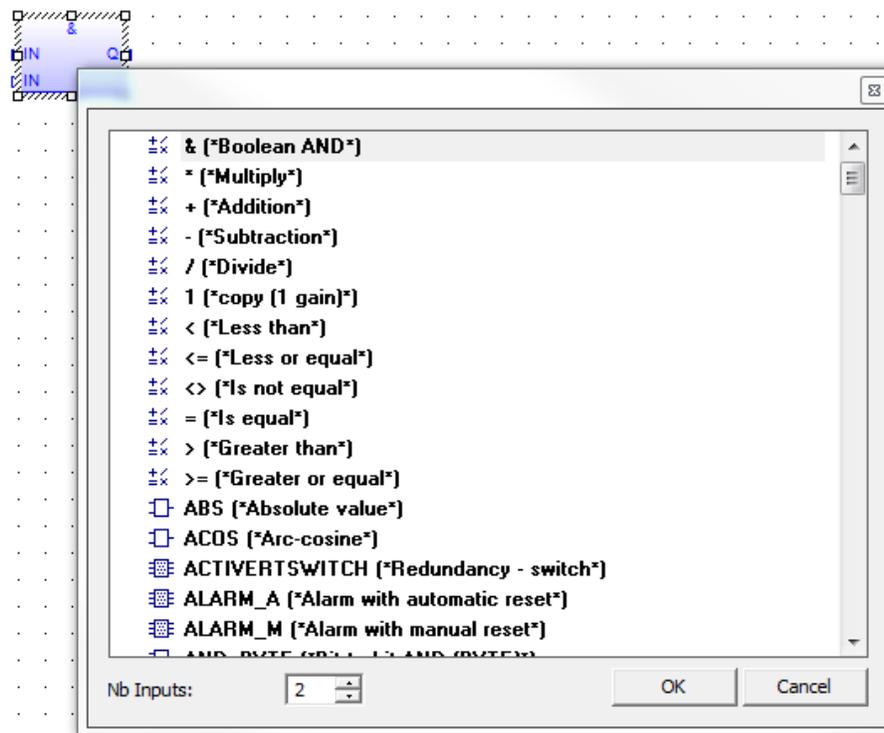
Add Function Block

Click to add a function block to the editor work space. Click this option then click in the editor work space. This will place an AND (&) function block with no variables inputs connected to it.



Click again to add another function block or right-click to switch back to the select tool.

To change the function block type, double-click the block to get a list of available blocks to choose from.

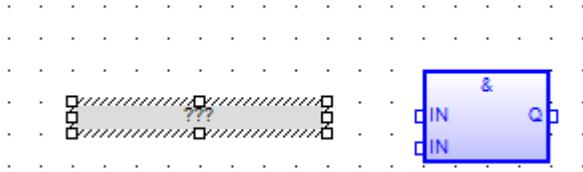


Also, use this menu to change the number of inputs if the selected function block supports that option.

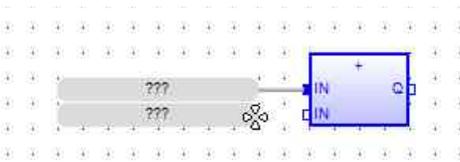
The logic blocks inputs and outputs can be connected to other elements or to variables using the Add Variable option.

Add Variable

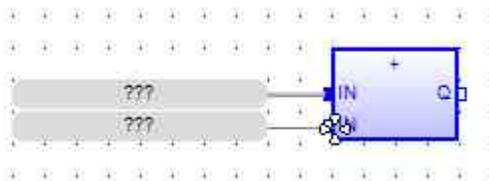
Places a variable field on the logic work area:



A variable field is used to connect tags to the inputs or outputs of function blocks that are not connected to other function blocks or to other elements. If you drag the function block from the Instruction List into the editor it will already have the required variable fields attached. Function blocks added using the quick select menu will not have variable fields attached. To place a Variable Field, click the Add Variable icon, move the mouse cursor to where you wish to place the Variable Field in the editor work area, then click the mouse cursor. Note: if the Variable Field (VF) does not appear after you click the mouse cursor, this means that you are placing the VF too close to another object. Once the VF has been placed on the work area, you must connect it to either a FB input or output. Move the mouse cursor to one side of the VF until the cursor changes to a crosshair:



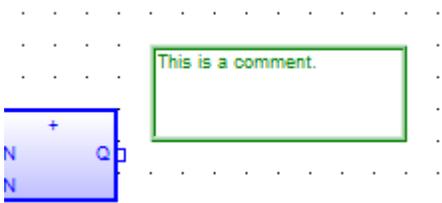
Click and Drag the mouse cursor over to the FB input, until another crosshair cursor appears, then release:



After adding the variable field, double-click to enter a tag.

Add Comment

Places a comment block on the logic work area.



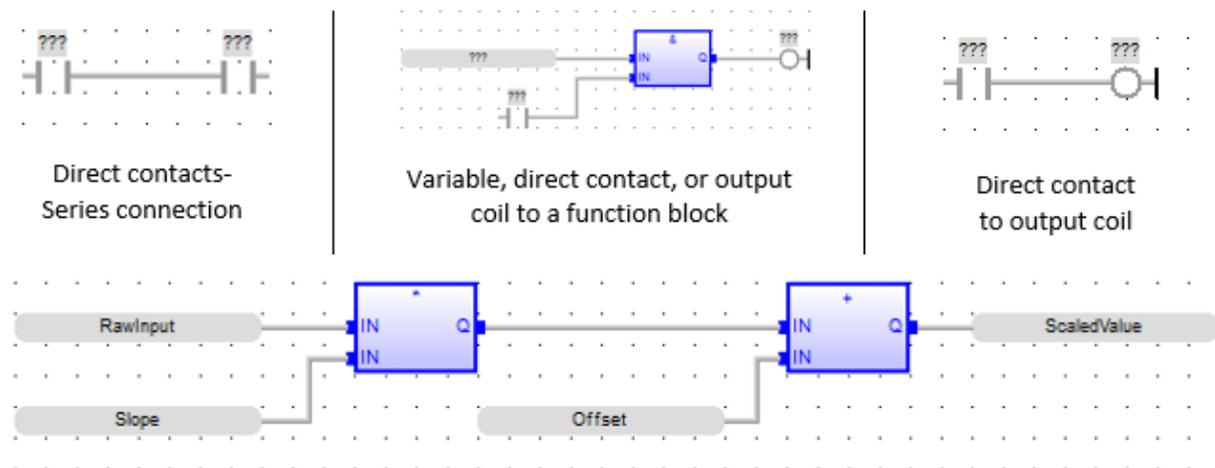
A comment block can be used to provide a description of the FBD or a particular object. To place a Comment Block (CB), click the Add Comment icon, move the mouse cursor to where you wish to place the CB in the work area, then click the mouse cursor. Note: if the Comment Block (CB) does not appear after you click the mouse cursor, this means that you are placing the CB too close to another object. Once the CB has been placed on the work area, double-click to display an input field:



Enter text then click the accept  button. The browse  button is used to open Windows File Explorer. You can then select a bitmap image that can be placed in the comment block.

Add Arc

An arc is used to link the output of one object to the input of another.



Function Block Output to Function Block Input

You can initiate using the Arc tool by clicking the Add Arc quick access key, then pointing the mouse cursor at any object's out point. This will cause the mouse cursor to change to a crosshair .

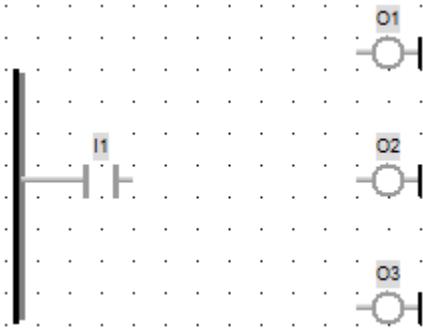
Click/drag the mouse cursor to the second object's in point. When the crosshair appears again, release the mouse cursor:



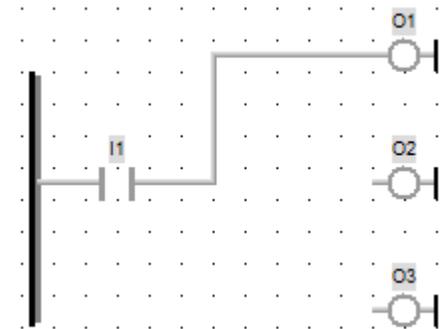
You can also use the Arc tool to click an object's out point, extend a line, and then end with a corner link (see Add Corner below). This allows you to branch off to more than one object.

Add Corner

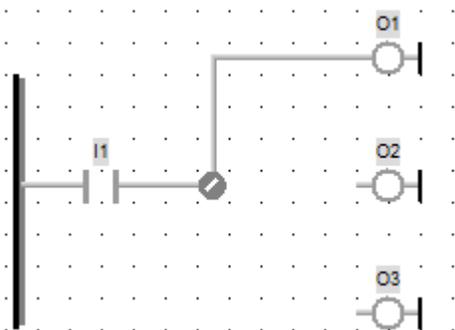
This tool adds a junction point  to an arc, thus providing the ability to branch to more than one object (to the input of those objects). For example, suppose we want to connect one contact to three coils:



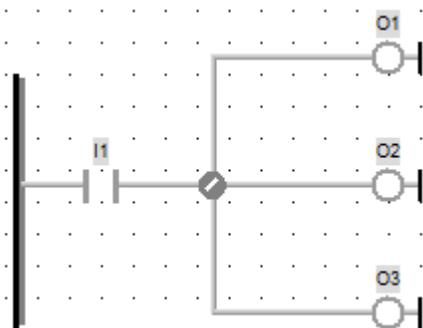
Step 1: Use the arc tool  to draw a connection between I1 and O1:



Step 2: Use the corner tool  to place a corner somewhere on the connecting rung:



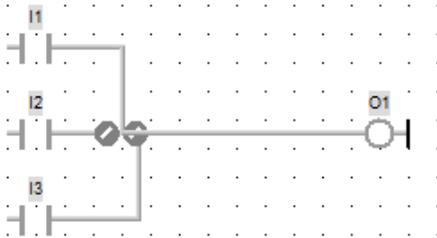
Step 3: Use the arc tool to connect the corner to the other two outputs:



Note that the arc tool only works when trying to connect the output of one object to multiple *inputs* of other objects. Suppose that you want to connect three contacts to one coil?



In this case, you are trying to connect multiple outputs (3 contacts) to a single input of one object (1 coil). You might try something like this:



This won't work- although the software lets you create this, as soon as you try to compile it, you will get several errors:

```
FBD_Block
FBD_Block: (19,8): Output pin not connected
FBD_Block: (18,8): Output pin not connected
```

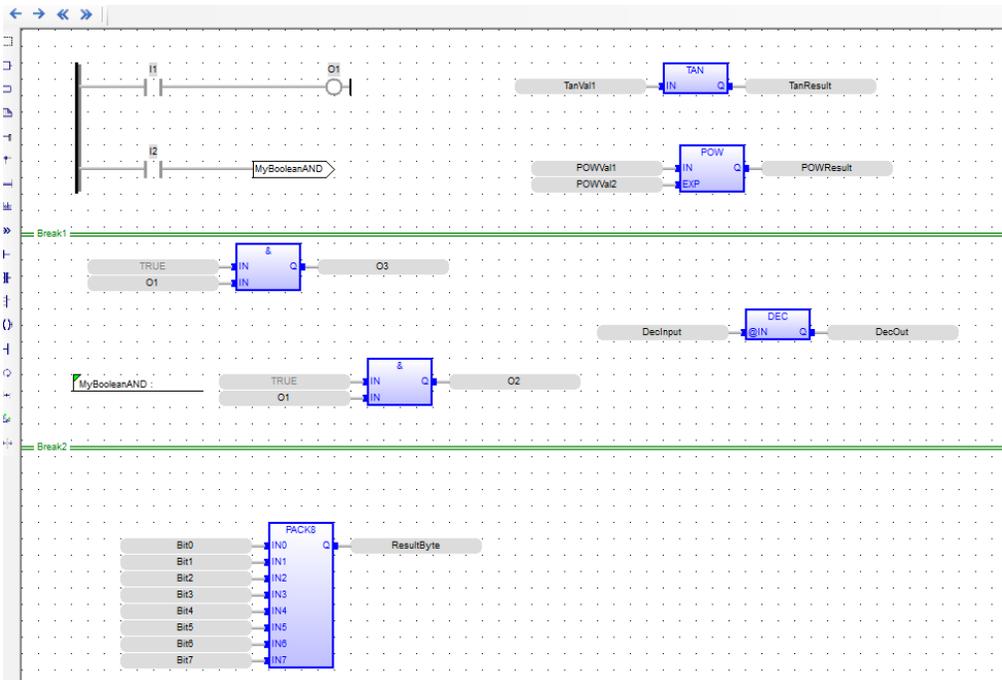


Note: any time you see a corner object in your logic with a line on top of it, the corner object is not actually connected to the line.

The solution would be to use the OR bar (see OR bar below).

Add Break

Inserts a network break (horizontal line).



The network break does not affect the operation of the FBD code. It is only used to help make complex diagrams easier to read by splitting the diagram into networks. To place a break there must be a horizontal section of the editor work area with no objects on it.

Add Label

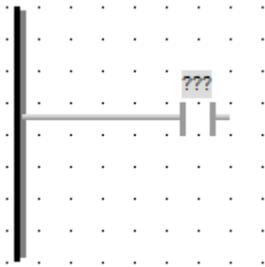
The label is used along with the Jump tool (see below) to skip (not execute) the segment of code between the Jump statement and the Label, when the Jump statement is active. Labels used without a corresponding Jump can be used to simply provide a description of some segment of the FBD logic block. See the section on [Jumps and Labels](#) for more information.

Add Jump

Inserts a Jump statement. The Jump is used with the Label to cause code execution to be redirected. See the section on [Jumps and Labels](#) below for more information.

Add Left Power Rail

This tool is used to place a left power rail on the FBD. Similar to LD programming, whenever you use contacts, you must tie them to a left power rail:

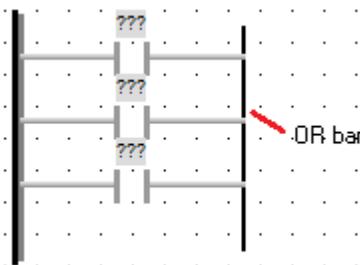


- You can resize the power rail to any length you need or you can place multiple power rails on the FBD
- When placing a direct contact on the FBD, if you first place the left power rail, then place a direct contact- the software will automatically tie the two together.

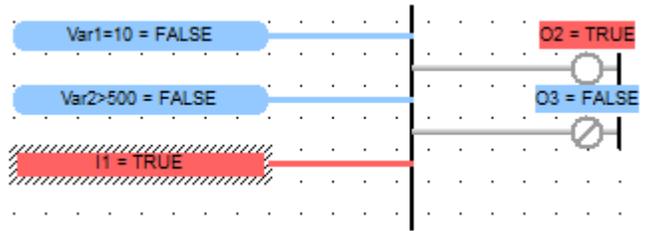
The left power rail can be placed anywhere on the FBD (it does not have to be on the left side).

Add OR bar

The OR bar is required to tie the outputs of multiple direct contacts for a parallel connection:



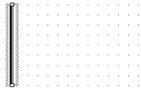
You can also use it to tie the outputs of multiple variables:



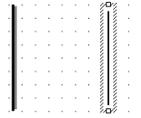
Notice above that you can also tie the OR bar to the inputs of multiple direct coils. Every OR bar must have at least one input and one output tied to it.

The fastest way to connect multiple direct contacts in parallel is to:

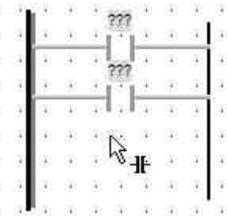
- Place a Left Power Rail on the FBD



- Place an OR bar on the FBD

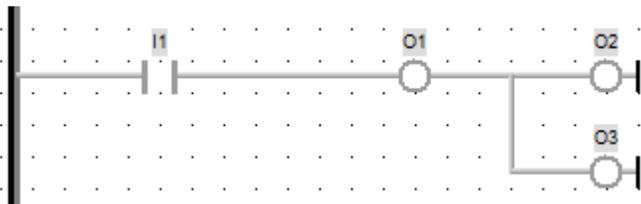


- Place a direct contact between the Left Power Rail and the Or bar. The software automatically connects the elements.



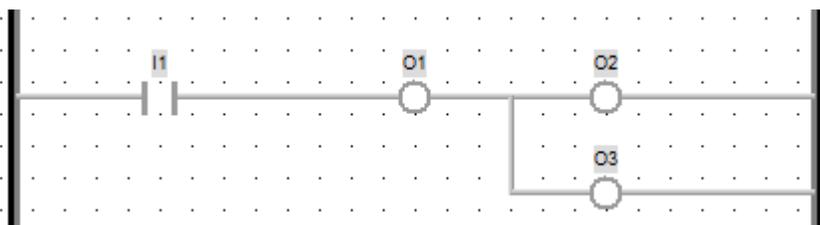
Add direct Coil

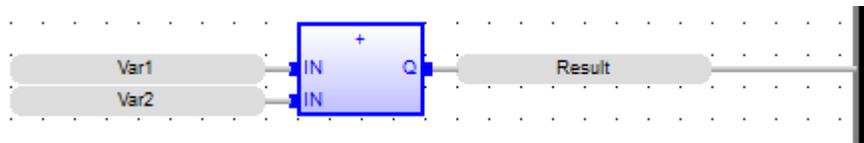
A direct coil  can be used to activate a physical output coil or as a result of code that it is tied to. Direct coils can be placed on the FBD in series or in parallel circuits:



Add Right Power Rail

The Right Power Rail can be used to complete a rung circuit.





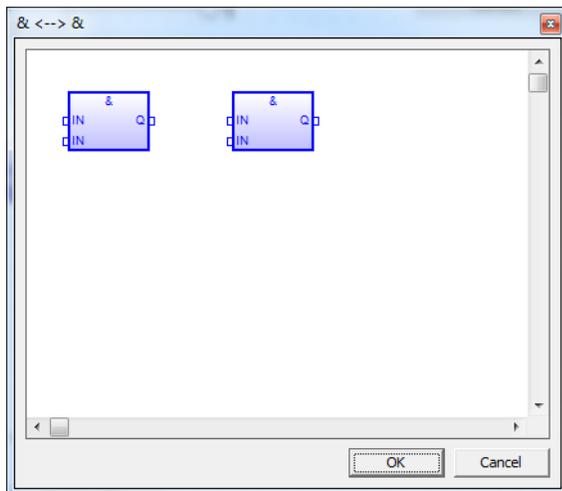
However, for FBD programming it is not necessary to have any direct coils connected to the right power rail. In fact, you don't have to use the Right Power Rail for any connection. It is provided as a convenience for users who are experienced with traditional ladder logic and wish to have it depicted in the FBD.

Swap Item Style

Changes the style of the highlighted contact or coil. The options cycle each time you click the Swap Item Style hotkey. See the section on Changing Item Style below for available contact and coil execution styles.

Connect Two Items

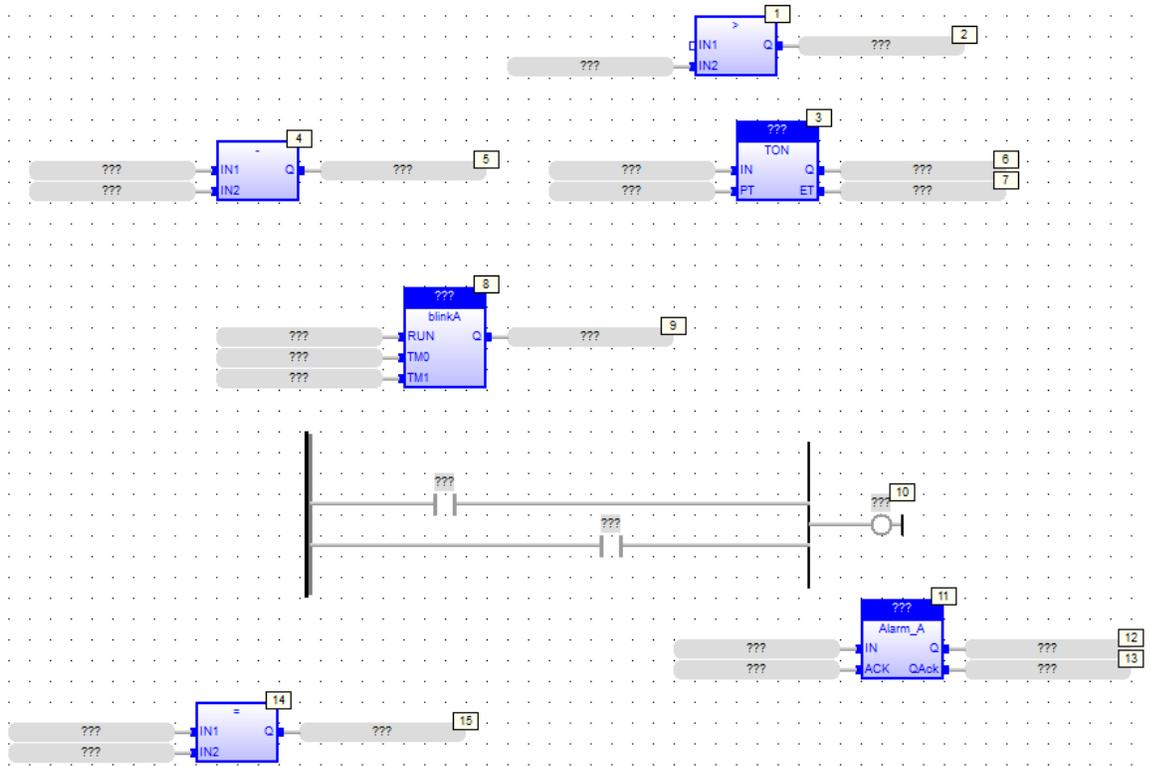
This tool can be useful if you need to connect two function blocks or other items that are far apart on the editor screen. First, using the select tool, hold down the <Ctrl> key and click both items you want to connect. A popup window will appear with both selected items next to each other.



Use the popup window to connect the outputs to the inputs as if the window were the main editor window. Click OK and the software will draw the full connection in the main editor window automatically.

Show Execution Order

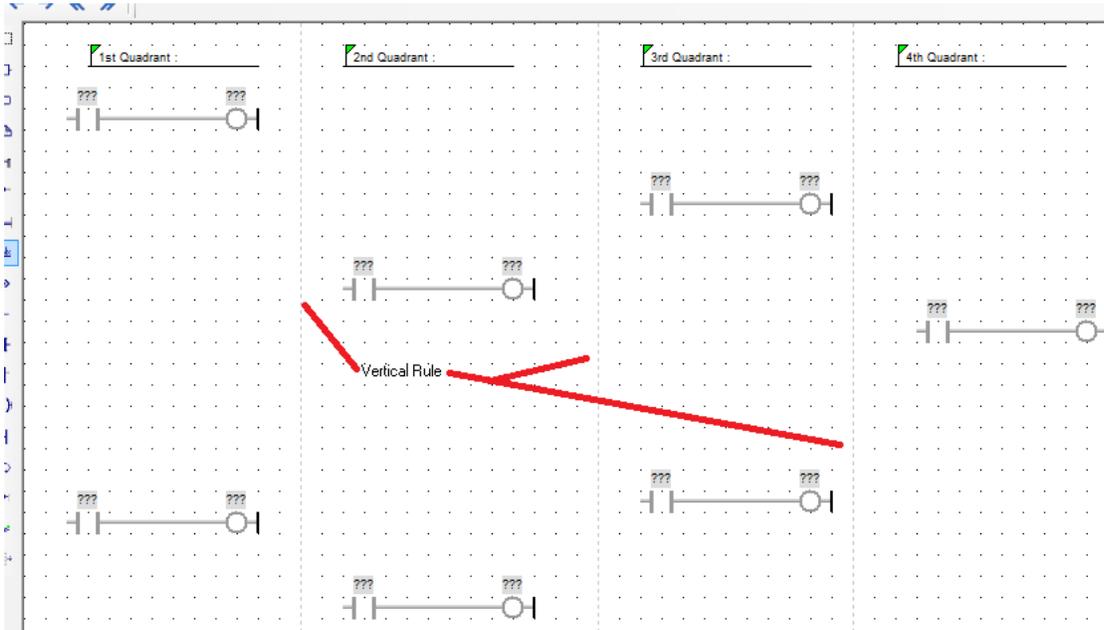
In general, code execution in an FBD logic block moves from left to right, top to bottom. Since function blocks are not required to attach to a left power rail function blocks can be floating in the editor window and it can be difficult to see the exact order. This tool will show you the execution order MAPware-7000 has assigned to all the blocks in the editor:



Note: The Show Execution Order tool is used only to show the execution order- you cannot change the order of execution using this tool.

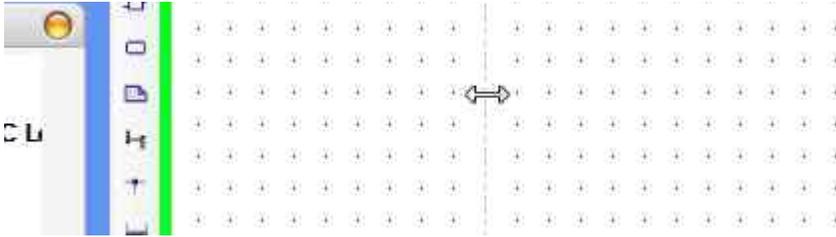
Add Vertical Rule

Similar to the Add Break tool, the Vertical Rule can be used to place vertical dotted lines on the FBD work area:



Vertical Rules have no effect on code execution- the only purpose is to split the FBD logic block into easier to read sections.

Once placed, you can move a vertical rule by moving the mouse cursor until it changes to a double-arrow:



Now click and drag to the new location. To delete a vertical rule, move the cursor until a double-arrow displays, then double-click the mouse.

Changing Contact and Coil Execution Style

As with real relays, there are several configurations available for contacts and coils in the Function Block Diagram editor. To change the execution style of an element, select that element in the editor window and click the Swap Item Style icon in the Quick Select menu, or press space on your keyboard.

Contact Execution Styles

The available execution styles for contacts are:

-  **Normally Open Contact** - If the value of the tag controlling the contact is 0 (false) no power flows through contact. If the value of the tag controlling the contact is 1 (true) power flows through contact.
-  **Normally Closed Contact** – If the value of the tag controlling the contact is 1 (true) no power flows through contact. If the value of the tag controlling the contact is 0 (false), power flows through contact.
-  **Positive (Rising Edge) Pulse Contact** - if the value of the tag controlling the contact transitions from 0 (false) to 1 (true), power flows through contact for one scan only.
-  **Negative (Falling Edge) Pulse Contact** - if the value of the tag controlling the contact transitions from 1 (true) to 0 (false), power flows through contact for one scan only.

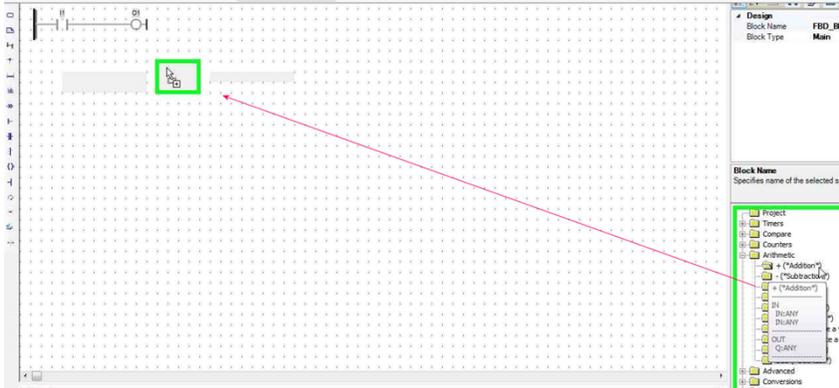
Coil Execution Styles

The available execution styles for coils are:

-  **Normal Coil Output** – The tag controlled by the coil is set to 1 (true) as long as power flows to the coil. The tag is set to 0 (false) when power does not flow to the coil.
-  **Inverted Coil Output** – The tag controlled by the coil is set to 0 (false) as long as power flows to the coil. The tag is set to 1 (true) when power does not flow to the coil.
-  **Set Coil Output** – The tag controlled by the coil is set to 1 (true) when power flows to it. The value of the tag will stay at 1 even after power no longer flows. This is also referred to as a latch instruction.
-  **Reset Coil Output** – The tag controlled by the coil is set to 0 (false) when power flows to the coil. The value of the tag will stay at 0 (false) when power no longer flows to the coil. This is also referred to as an unlatch instruction.

Adding Function Blocks from the Instruction List

In addition to using the Add Function Block quick select menu option, you can simply click-drag on a function block in the Function Block List and drag to the work area:

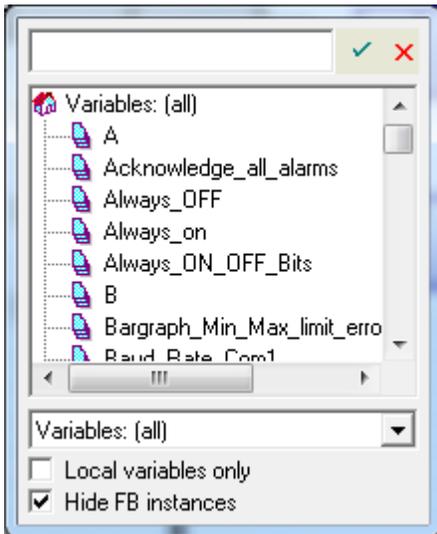


Place the Function Block (instruction) on the FBD work area.

The function block will be configured with Variable Labels for each input and output. These can be configured with tags, or deleted to connect the function block to another function block or element.

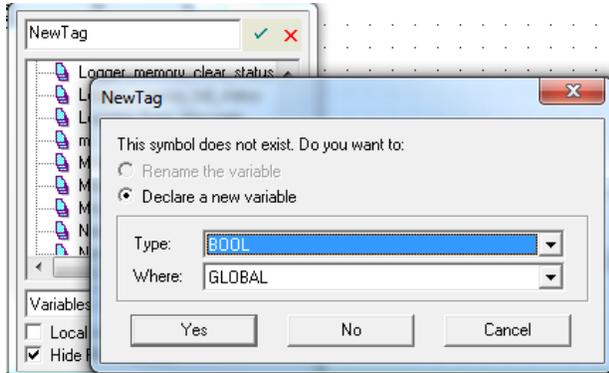
Assigning Inputs and Outputs to FBD elements

After a contact, coil or function block is placed in the editor window, tags must be assigned. To select a tag double-click the element you are trying to configure. A tag selection box will be displayed.



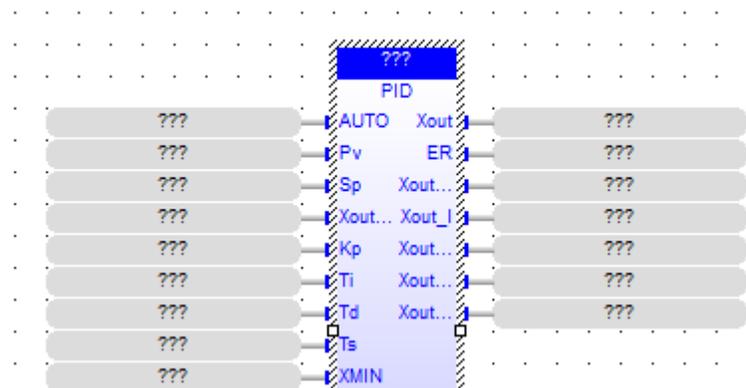
You can scroll through the variable list to find an appropriate tag. If the tag name is known, begin typing the name in the text box and matching tags will be available for auto complete. Once the tag is selected press Enter or the green checkmark to assign the selected tag.

If no tag exists with the name entered, a popup window will appear allowing a new tag to be created.

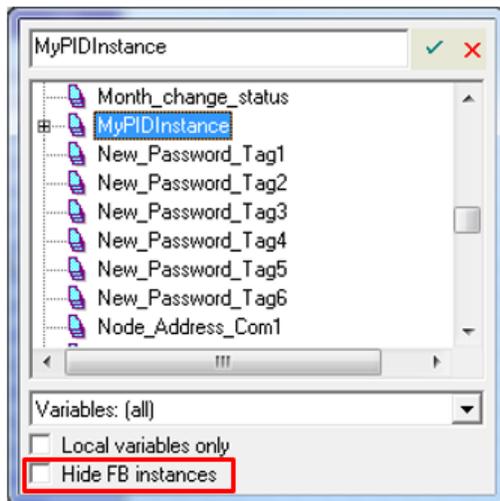


Selecting Function Block Instances

User Defined Function Blocks and many of the built in Function Blocks require that an instance of the function block be selected so that the block can operate on its own set of data. If this is the case, the function block will appear with a set of question marks above the type of the block.

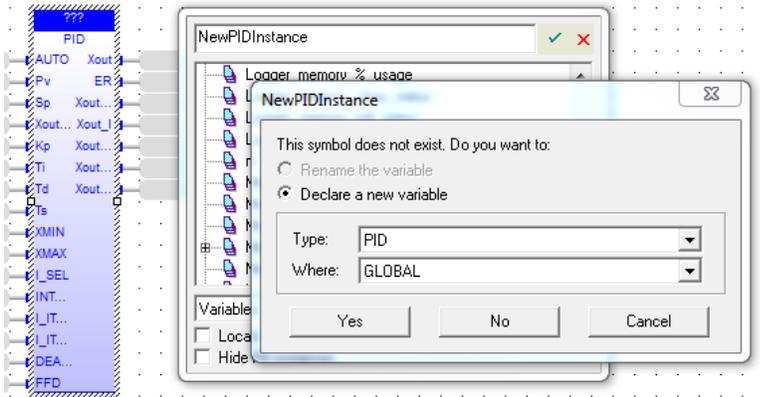


To select an instance to use, simply double-click the question marks to display the Tag selection window. Function Block Instances will appear in this window along with normal tags. Make sure that the Hide FB instances checkbox is not checked.



The list of available Function Block Instances can be found in the Function Block Instance folder of the **Project Information Window**. The type of the function block instance must be the same as the function block being configured. If there is a mismatch this will cause a compiler error.

To create a new instance, simply type the name for the new instance in the text box and hit enter. A popup window will appear allowing the new instance to be configured. The new instance is automatically added to the Function Block Instance folder.

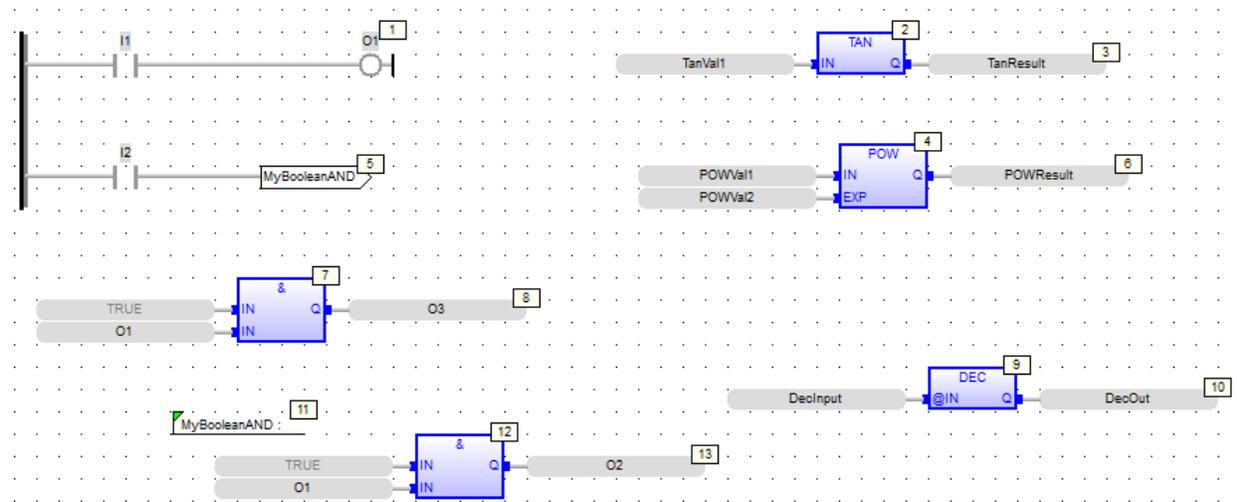


Jumps and Labels

Jumps and Labels allow function blocks to be conditionally skipped (not executed). When a Jump statement is active, the segment of code between the Jump statement and the Label will be skipped. Jump and Label elements can be placed in the editor window using the [quick select](#) menu options.

Note: Jumps skip function blocks based on their execution order which is determined by the location of the function block in the editor. Because function blocks can “float” in the editor window execution order can be ambiguous to the naked eye. To see the execution order MAPware-7000 has assigned to elements use the Show Execution Order quick select tool ().

To see how Jumps and Labels operate, consider this Function Block Diagram:

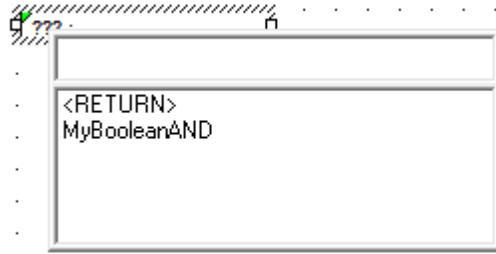


Notice a Jump statement at location 5 with the name: *MyBooleanAND*. Note the Label statement at location 11 with the same name. During execution of this FBD, when input contact I2 is active, the jump statement is activated. This causes the code execution to skip all code between the jump statement and the label statement (locations 6-10).

To implement the label, click the Label tool and place a label on the FBD:

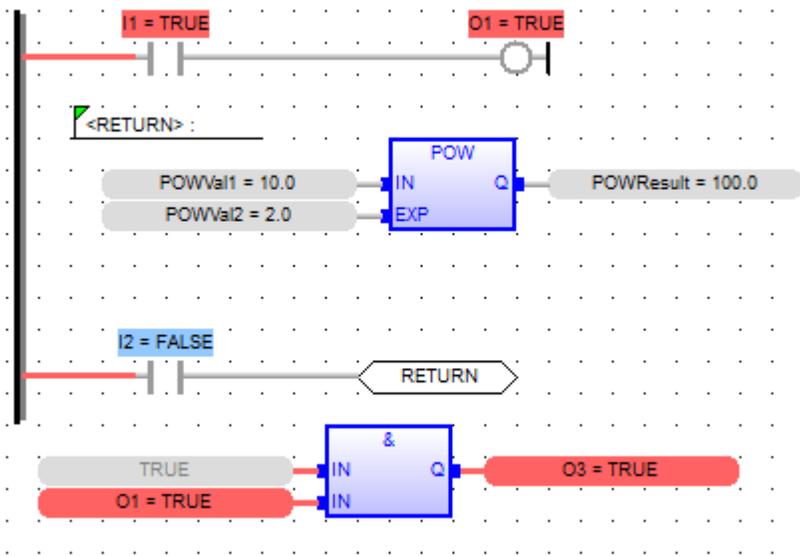


Double-click the label to display the Edit popup window:



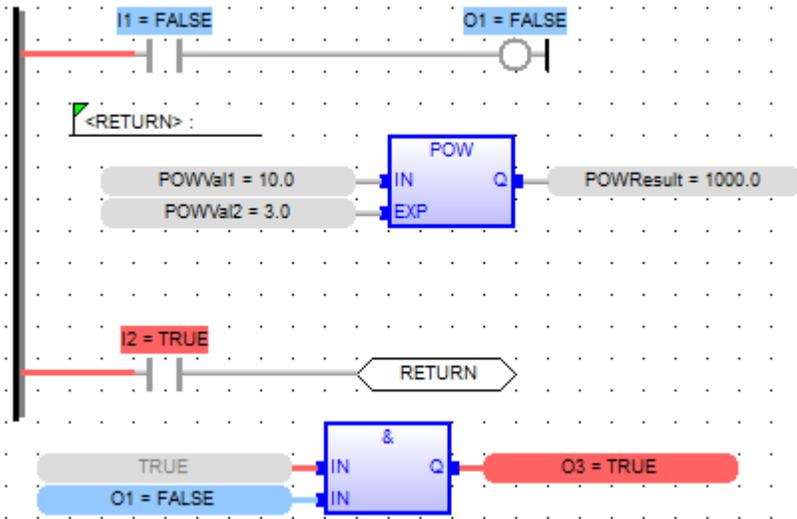
The <RETURN> statement and any Jump statements that have been created are displayed as a list. You can select from the list or enter a new label name.

The <RETURN> statement is a special instance that behaves a little differently. If a Jump statement and a Label statement are assigned to <RETURN>, when the Jump statement is activated, code execution immediately goes to the Label <RETURN>:



In this example, input contact I2 is off so the Jump <RETURN> is not activated. Therefore, code execution proceeds as normal- with I1 in the first rung setting coil O1. The POW function is executed and the AND function at the bottom is executed.

If we set input contact I2 true, then the Jump <RETURN> is activated:



- Now, when we reset input contact I1 to OFF, output coil O1 is set off.
- If we change the inputs to the POW function, the output is updated.
- However, notice that since the Jump <RETURN> is now active, code execution immediately goes to the Label <RETURN> statement- causing a loop in which only the POW statement is executed.
- Notice the AND function after the Jump <RETURN> statement. Although O1 is now False, output O3 remains True since the code is not executed.

The code is now a continuous loop until I2 is reset back to False.

Chapter 6 – Structured Text (ST)

Overview

As the name suggests Structured Text is a text based programming language in which program instructions are entered as discrete statements in a text source file. The processor executes the statements in the source file from left to right top to bottom (subject to the rules of precedence and program flow as described below). This programming method is similar to other high level programming languages such as C or Visual Basic.

Statements

The basic unit of structured text program is a statement. A statement is an instruction to the processor to perform a set of actions. Statements are composed of one or more expressions and end with a semicolon ‘;’ character. Statements can contain other statements and span multiple lines. Multiple statements can be placed on a single line, though limiting source files to one statement per line is a good way to improve readability.

ST programs are lists of statements. The order in which statements are evaluated is, by default, from the top of the source file to the bottom. However, this order can be controlled by the programmer using keywords that tell the processor to change to order of operation. These keywords are said to control program flow. Some examples of keywords are: FOR, IF, END_IF

Some common examples of Structured Text statements are:

```
// Assignment
TagA := 42;

// Conditional
If TagA > 22 then
  B := C;
end_if;

// Function Call
RealNumber := any_to_real(10);

// Iteration
FOR inx := start to stop do
  TagA := inx;
end_for;
```

Expressions

Expressions are the building blocks of ST statements. An expression is any piece of a statement that evaluates to a single value. Expressions may or may not have side effects. A typical ST expression consists of an operator and one or more operands. Some typical ST statements are:

```
// Assignment
TagA := 42

// Function call
any_to_real(13)

// Math operations
TagA + 10

// Comparison
TagA > TagB
```

Note: These expressions are shown by themselves, as such they will cause a compile error. To be a valid ST program, expressions must be organized into statements.

Operands

The operands of expressions are themselves expressions. When an expression is evaluated, it is replaced with the result of the operation. This result is then used as an operand to evaluate the next expression. This process continues until the entire statement has been evaluated. The order in which expressions are evaluated is governed by operator precedence. Expressions that contain higher precedence operators are evaluated first and expressions with lower precedence operators are evaluated later.

Comments

ST programs use familiar mathematical operators, keywords that have meanings in English, and tag names that can describe the data they represent. However, the structure of statements means that it is not always clear what a program doing. Comments should be used liberally in ST programs to make it clear to anyone reading the source file what the program is doing (or at least what it is intended to do).

Comments are simply text in the source file that is visible to anyone looking at the file in the editor, but is ignored by the compiler and is not part of the program executed by the processor.

There are two ways to enter comments in the ST Editor. A block comment is created by enclosing the comment between (*) character sequences:

```
(* This is a block comment *)
```

Block comments can span multiple lines:

```
(*
  This is a
  Multiline
  Block
  Comment.
*)
```

And can be embedded in a line of text:

```
// Conditional
If TagA > 22 then
  B := (* This comment is not really helpfull *) C;
end_if;
```

The second type of comment is an inline comment. An inline comment starts with a // character sequence and continues to the end of a line.

```
// This is an inline comment
```

The // character sequence must be repeated before (to the left of) each line that is to be used as a comment.

```
// This multiline
// comment
// requires
// more typing
```

Inline comments can be placed after (to the right of) a line of code.

```
// Conditional
If TagA > 22 then
  B := C; // This comment is not really helpfull
end_if;
```

But can't be embedded in a single line of code because all of the text to the right of the // is ignored.

```
// Conditional
If TagA > 22 then
  B := // This comment will cause a compile error // C;
end_if;
```

Adding Logic to the Block

Operators and Expressions

This section lists the operators available in ST programs to compose expressions. Operators act on operands to produce some result, the result becomes the value of the expression. Operators may also have side effects. For example, the assignment operator '=' evaluates to the value of the expression on the right hand side of the operator, and has the side effect that the tag on the left hand side is assigned the same value. Another example is a function call. The function call expression evaluates to the function's returned value, and as a side effect, the statements in the function are executed.

In ST programs, the operands of expressions are themselves expressions. The expression to the right is referred to as the RHS operand and the expression to the left is referred to the LHS operand. Some operators only act on one operand, some act on both the RHS and LHS operands. In either case, once the expression is evaluated, the operator and its operand(s) are replaced with the result of the operation. Then the next expression is evaluated. This process repeats until the entire statement is evaluated.

The order in which expressions are evaluated is referred to as precedence. Operator precedence is shown in the table below (precedence of 1 indicates expressions with this operator will be evaluated first, etc.). Operators of the same precedence are evaluated left to right. Parenthesis can change the precedence of operators and should be used liberally to make the intended order of operations explicit.

Operators available to use in ST are given in the following table:

Operator	Name	Description	Example Statement(s)	Priority
()	Parenthesis	Used to change the order in which expressions are evaluated. Expressions within parenthesis are evaluated first.	// with (), TagA = 40 TagA := 10 * (1 + 3); // without (), TagA = 13 TagA := 10 * 1 + 3;	1
func()	Function call	Values are assigned to parameters. Function is executed. Expression evaluated to returned value.		2
**	Exponential	Evaluates to the left hand expression raised to the power of the right hand expression. Note: both operands must be of type REAL.	result := 2.0 ** 2.0; // 4.0	3
-	Negation	Evaluates to the negative of the operand immediately to the right.	TagA := -TagB;	4
NOT	Complement	Returns that opposite logical value of the Boolean operator on the RHS.	State := NOT true; // false State := NOT false; // true	4
*	Multiplication	Evaluates to the LHS multiplied by the RHS	result := 3 * 4; // 12	5
/	Division	Evaluates to the LHS divided by the RHS.		5
MOD	Modulo	Computes the modulus (Remainder) of the LHS divided by the RHS.	result := 5 MOD 2; // 1 result := 9 MOD 3; // 0 result := 12 MOD 5; // 2	5
+	Addition	Computes the sum of the RHS added to the LHS.	result := 5 + 2; // 7 result := 9 + 3; // 12 result := 12 + 5; // 17	6
-	Subtraction	Subtracts the RHS from the LHS		6
<	Less Than	True when LHS is less than the RHS, false otherwise.		7
>	Greater Than	True when the LHS is greater than the RHS, false otherwise		7
<=	Less than or equal	True when the LHS is less than OR equal to the RHS, false otherwise		7
>=	Greater than or equal	True when the LHS is greater than OR equal to the RHS, False otherwise		7
=	Test for equality	True if RHS has the same value as the LHS, False otherwise.		8
<>	Test for inequality	True if the RHS does not have the same value as the LHS, False otherwise		8
&, AND	Boolean AND operation	True if LHS AND RHS are true, false otherwise	result := true & true; // true result := true AND false; // false result := false AND true; // false result := false & false; // false	9
XOR	Exclusive OR	True if the LHS is true while the RHS is false, or the LHS is false while the RHS is true. False if both LHS and RHS are true. False if both LHS and RHS are false.	result := true XOR true; // false result := true XOR false; // true result := false XOR true; // true result := false XOR false; // false	10

Operator	Name	Description	Example Statement(s)	Priority
OR	Boolean OR	False if both LHS AND RHS are false, true otherwise.	result := true OR true; // true result := true OR false; // true result := false OR true; // true result := false OR false; // false	11
:=	Assignment	Writes the value of the RHS to the tag on the LHS. Note: only one assignment operator is allowed per statement. The LHS operand must be a tag.	TagA := 10; // TagA set to 10 TagB := TagA; // TagB set to 10	12

Keywords and Program Flow

ST programs have reserved keywords that have special meaning to the compiler. In general, these keywords are used by the programmer to control when and how statements are executed. The table below lists the keywords available in the ST Editor, the sections that follow describe how they are used.

Keyword	Description	Section
IF	Indicates test condition for IF statement	IF Statement
THEN	Indicates beginning of block of statements controlled by IF statement.	IF Statement
ELSIF	Indicates test condition to control ELSIF block of statement in an IF statement.	IF Statement
ELSE	Indicates beginning of conditional block of statements for ELSE condition in IF statement, or default block in CASE statement.	IF Statement , CASE Statement
END_IF	Indicates end of IF statement	IF Statement
CASE	Indicates input to CASE, multi-selection statement.	CASE Statement
OF	Indicates beginning of condition blocks for CASE, multi-selection statement	CASE Statement
END_CASE	Indicates the end of a case statement	CASE Statement
FOR	Indicates expression used in for loop, and initial value of the for the loop index.	FOR Statement
TO	Indicates expression used for final value of FOR loop index.	FOR Statement
BY	Indicates expression used for increment value in for loop; i.e., the FOR loop index is increased by this much on every iteration.	FOR Statement
DO	Indicates beginning of statement block used in FOR statement and WHILE statement	FOR Statement , WHILE Statement
END_FOR	Indicates end of statement block used in FOR statement	FOR Statement
WHILE	Indicates expression used as condition for while loop.	WHILE Statement
END_WHILE	Indicates the end of a statement block and end of a WHILE loop	WHILE Statement
REPEAT	Indicates the beginning of a block of statements used in a repeat loop	REPEAT Statement
UNTIL	Indicates end of REPEAT statement block and expression used for test condition of REPEAT block. The loop continues until the condition is true.	REPEAT Statement
EXIT	Exit from loop	EXIT Statement
RETURN	Return from a function	RETURN Statement

IF Statement

The **IF** statement is used to conditionally control the execution of a block of statements. When the condition is true the code is executed when it is false the code is not. The Syntax of an **IF** statement is as follows (on next page):

```

IF <Condition> THEN
    <Statement>;
    > .
    <Statement>;
END_IF;

```

<Condition> is an expression that evaluates to a Boolean value (e.g. >, <, <=, >=, OR, AND, XOR, OR).

When the <Condition> expression is true. The Statements are executed. When <Condition> is false the <Statements> are not executed and program execution moves to statements after the END_IF keyword.

Example:

```

// If Statement
If TagA > 22 then
    B := C;
    D := E;
end_if;

```

The IF statement can also contain one or more ELSIF keywords with additional conditions, and an ELSE keyword that is executed when the IF (and any ELSIF) conditions are false. The syntax when using ELSIF and ELSE keywords is:

```

IF <ConditionA> THEN
    [Statement(s)]
ELSIF <ConditionB> THEN
    [Statement(s)]
>
ELSIF <ConditionN> THEN
    [Statement(s)]
ELSE
    [Statement(s)]
END_IF;

```

The condition following an ELSIF keyword is only evaluated if the preceding IF condition is false. If multiple ELSIF keywords are used only the first one with a condition that evaluates to true will be executed. The ELSE statement will be executed if and only if the IF condition and all ELSIF conditions are false.

Example:

```

IF Temp > 100 THEN
    MyString := 'Temp is High';
ELSIF Temp < 30 THEN
    MySTRING := 'Temp is Low';
ELSE
    MySTRING := 'Temp is Normal';
END_IF;

```

CASE Statement

The case statement provides a way of selecting from a set of conditions. The syntax for the CASE statement is as follows:

```

CASE <expression> OF
    <value>:
        [Statement(s)]
    <value>, <value>, > :

```

```

    [Statement(s)]
  <min value> .. <max value>:
    [Statement(s)]
  ELSE
    [Statement(s)]
END_CASE;

```

The **CASE** keyword is immediately followed by an expression which acts as the parameter to the rest of the statement. The **OF** keyword indicates the beginning of a list of possible values or range of values. When the value of the expression is equal to one of the listed values (or within the range or set of values) the statements following that value (or range/set of values) is executed. The **CASE** statement can also contain an **ELSE** keyword in place of the last value. This acts as the default case and is executed if none of the other values match of the expression's value.

Example:

```

// CASE statement
CASE NiceGuy OF
  'Good':
    CharacterPts := 10;
  'Bad':
    CharacterPts := 0;
  'Ugly':
    CharacterPts := 5;
  ELSE
    CharacterPts := -1;
END_CASE;

```

Example:

```

// CASE statement
CASE CharacterPts OF
  10:
    NiceGuy := 'Good';
  1 .. 7:
    NiceGuy := 'Ugly';
  0, 8, 9:
    NiceGuy := 'Bad';
  ELSE
    NiceGuy := '';
END_CASE;

```

In the second example note that, for the first case a single value (10) was used, in the second case a range (1 to 7) was used and in the third case a set of values (0, 8 and 9) were used. The values for each case must be unique, and must be constant values. Tags are not allowed.

FOR Statement

The FOR statement is a means of iterating or repeating a series of statements a specified number of times. The syntax of a FOR statement is:

```

FOR <index expression> := <min> TO <max> BY <step> DO
    [Statement(s)]
END_FOR;

```

The statements between the **DO** and **END_FOR** keywords will be executed repeatedly. The first time they are executed <index expression> is set to <min>. After one execution <index expression> is incremented by the amount specified by <step>, and the statements are evaluated again. The process is repeated until <index expression> is equal to or greater than the value of max.

<index expression>, <min>, <max> and <step> must be a tags. The **BY** keyword is optional. If omitted the index is incremented by 1.

Example:

```
minimum := 0;
maximum := 9;
increment := 1;

FOR index := minimum TO maximum BY increment DO
    squares[index] := index * index;
END_FOR;
```

WHILE Statement

The **WHILE** statement is another iteration statement. It executes a block of code while an expression evaluates as true. The syntax for the while statement is as follows:

```
WHILE <condition> DO
    [Statement(s)]
END_WHILE;
```

As long as the expression <condition> evaluates to true, the statements between **DO** and **END_WHILE** will continue to execute. This means that the programmer must include logic in the while loop's block of statements that will eventually result in the condition expression evaluating to false. If not the loop will never exit.

Example:

```
Count := 0;

WHILE Count < 10 DO
    Result := Result + 1;
    Count := Count + 1;
END_WHILE;
```

REPEAT Statement

The **REPEAT** statement is similar to the **WHILE** statement except that the condition to continue loop execution occurs after the block of statements is executed, and the loop continues until the condition becomes true. Use this statement if you want to ensure the block of statements is executed at least once.

The syntax for the **REPEAT** Statement is as follows:

```
REPEAT
    [statement(s)]
UNTIL <condition>
END_REPEAT;
```

As with the WHILE statement the programmer must be careful to include logic in the statements that makes the condition become true at some point, or the loop will never exit.

Example:

```
// REPEAT statement
Counter := 0;
REPEAT
    SumofFives := SumofFives + 5;
    Counter := Counter + 1;
UNTIL Counter = 100
END_REPEAT;
```

EXIT Statement

The EXIT statement is used to break out of an enclosing loop (FOR, WHILE or REPEAT).

Example:

```
// REPEAT statement
Counter := 0;
REPEAT
    Total := Total + Item;
    INC( Counter );
    IF Total = 250 THEN
        EXIT;
    END_IF;
UNTIL Counter = 100
END_REPEAT;
```

RETURN Statement

The RETURN statement is used to return from a function or a function block and return execution to the calling statement.

Example:

```
IF EarlyEXIT THEN
    RETURN;
END_IF;
```

This keyword can be used in an ST subroutine or User Defined Function Block to return at the end of the function or to return when a certain condition is true, by pairing the RETURN statement with an IF statement as shown in the example.

Preprocessor Commands

In addition to operators, keywords and function calls, MAPware-7000 also has a set of Preprocessor commands. As the name suggests these commands are evaluated before the program is executed. In fact they are evaluated before the program is compiled. These are generally used to define static variables that correspond to constant values. They can make a program easier to read by replacing numbers with meaningful text. The available Preprocessor commands are:

Command	Description
#define	Create a name that represents a constant number.
#ifdef	Tests whether the specified label has been defined.
THEN	Used with #ifdef statement to specify a list of #define statements to evaluate when the variable exists.
#else	Designates a block of #define statements to evaluate if the #ifdef evaluates to false.
#end_if	Designates the end of a #ifdef block.

Preprocessor commands can be placed in a source file directly. Or placed in a special Local Define Area. To open the local define area open the Logic Block that you want to use. Then click **View>Local Define**. The define area appears above the ST Editor window:

```
#define ST 0
#define STP 10
#define VALUE 33

// ST Logic Block
TagA := VALUE;
```

The screen shot below gives an example of how these preprocessor commands can be used:

```
#define ERR_TABLE 5
#ifdef ERR_TABLE THEN
    #define ALARMS_EMPTY 0      Local Define Area
    #define ALARMS_FULL 1
    #define ALARMS_MAX 2
    #define ALARMS_OVER 3
    #define ALARMS_FAULTY 4
#else
    #define GENERAL_ERR 0
#endif

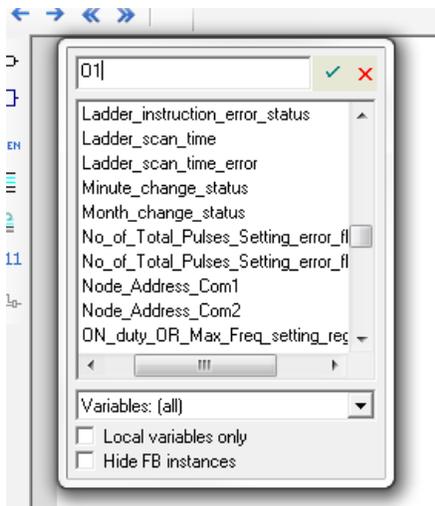
// end-of-line comments will comment the entire line
O1 := I1 OR I2; // comment
O2 := I3 OR I4; // comment      Normal Work Area
```

Quick Select Menu Options

In the ST Editor, the Quick Select Menu has the options detailed in the sections that follow.

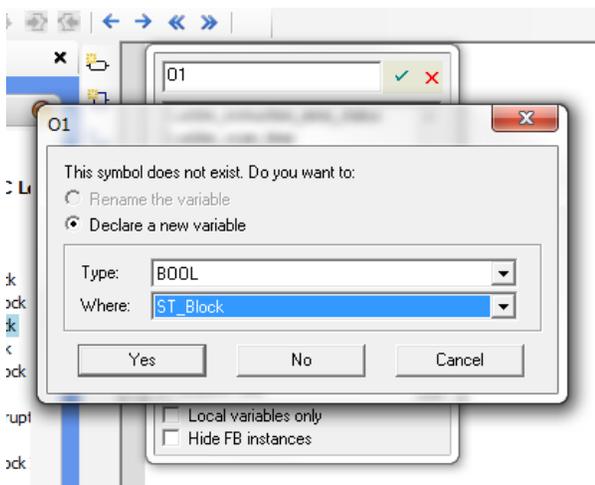
Insert Variable

This option allows you to use the standard variable selection window to insert a variable in the program instead of simply typing in the tag name in the editor. This can be helpful in remembering what tags are available.



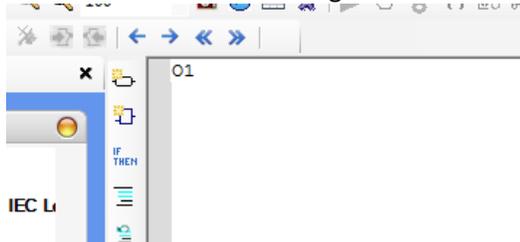
By default, all of the System Tags, global tags, and Local tags for the selected logic block are listed. You can select a tag from the list or create a new one by typing a new tag name, then clicking the Accept  button. *Note: if you only want to see local tags for the selected logic block, check the Local variables only box.*

In this example, we will add a local variable O1. If the tag is new, clicking the Accept  button displays a new dialog:



When creating a new variable, you must determine the data type (i.e. BOOL, INT, STRING, etc.) and visibility of the variable. If Global is selected, the variable can be used in any of the logic blocks you create. If you select Local (by selecting the name of the logic block you are in), then the tag variable can only be used and seen by the logic block. Selecting RETAIN stores the tag in non-volatile memory of the device so that the value is retained after power is cycled.

Click Yes to return to the logic block work area and place the tag:



The tag is automatically added to the tag database.

Tag No	Tag Name	Data Type	Attribute	Tag Address	Pc
125	ST_Block/O1	BOOL (L)	Read Write	-	-
124	Initialize__Port_Com2	BOOL	Read Write	S00093	-
123	Initialize__Port_Com1	BOOL	Read Write	S00092	-
122	Node_Address_Com2	WORD	Read Write	SW0248	-
121	Stop_Bits_Com2	WORD	Read Write	SW0247	-

In ST Programming, you may prefer to write the code statements, then assign variables afterwards.

For example, let's add a statement to the variable O1 that we already created by typing:

```
O1 := I1 AND I2;
```

If we try to compile, we will get errors because the new variables, I1 and I2 are not yet added to the Tag library:

```
Compiler V14.5.7.0
>> Complex variables stored in a separate segment
Loading application symbols...
Main
LD_Block
ST_Block
ST_Block: (1): I1: Unknown identifier
Error(s) detected
```

You can add a new variable that has already been written into the code by double-clicking the variable (to highlight the entire name), then clicking the Insert Variable button:

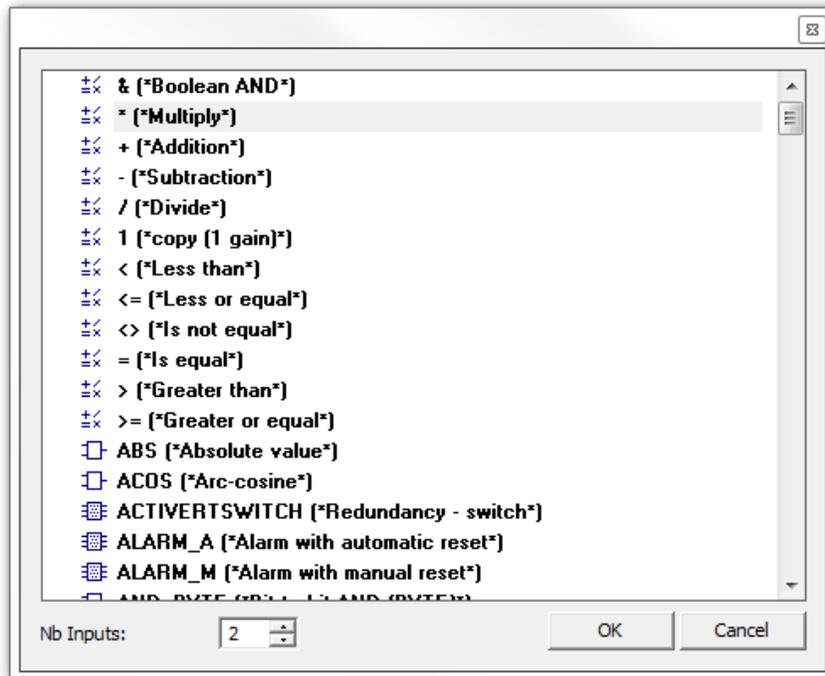
```
O1 := I1 AND I2;
```

Once a variable has been added to the Tag library, you do not have to use the Insert Variable button again to reference the tag- simply type the name of the tag into the code.

Insert FB

This option can be used to insert a function call or function block call. Or to select an operator to use.

Click to add a function block to the logic code work area. The Function Block dialog box appears:



Note the icons to the left of the functions in this window. These indicate the type of object being selected.

- 
 this icon indicates an operator or a simple function with 1 input and one output. Examples are:

```

Bool3 := Bool1 & Bool2; // Boolean AND

Real3 := Real1 / Real2; // Division

Bool4 := Int1 <= Int2; // Check if Int1 is Less Than or Equal to Int2

Real4 := ANY_TO_REAL( Dint1 ); // Convert DINT to REAL

Bool5 := NOT( Bool5 ); // Change state of BOOL

String3 := CONCAT( String1, String2 ); // Combine two strings

```

- 
 this indicates more complex function call that may have several inputs of different data types and one output. To get the output, you must use a variable with the correct data type with the assignment operator. Examples are:

```

Int2 := ABS( Int1 );

Word3 := AND_MASK( Word1, Word2 );

Dint2 := ASCII( String1, Dint1 );

Real1 := SCALELIN( inREAL, iMinREAL, iMaxREAL, oMinREAL, oMaxREAL );

```

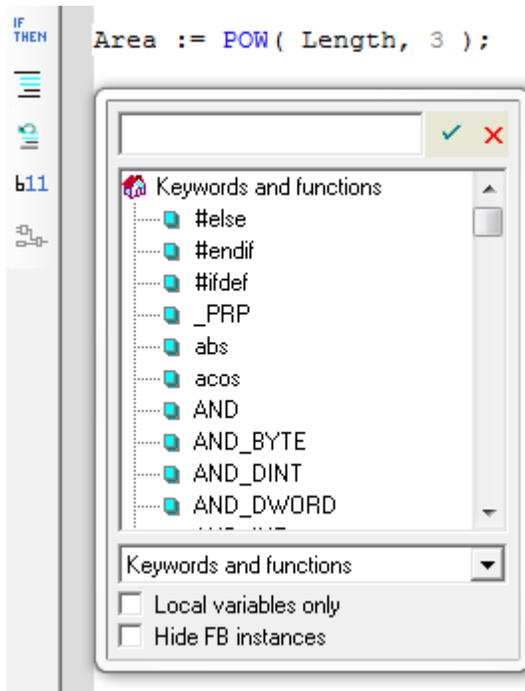
- 
 this icon indicates a user defined subroutine. These functions are used the same way as built in functions 
- 
 this icon indicates a built in function block. In this case selecting the function block will simply place the name of the function block in the text editor. If you want to use a function block this will not work. Instead you should enter the name of a function block instance as described in the section on [Function Block Calls](#).

-  this icon indicates a User Defined Function Block. In this case selecting the function block will simply place the name of the function block in the text editor. If you want to use a function block this will not work. Instead you should enter the name of a function block instance as described in the section on [Function Block Calls](#).

Select the Function that you wish to use. With some functions, you can change the number of inputs using the Nb Inputs option. Because ST programming is text-based, all functions are displayed in the work area as either a mathematical operator (i.e. +, -, *, >, :=) or as a function name with a description of required input parameters, (ex. AND_MASK(IN(*ANY*), MSK(*ANY*))). You must complete the function block by adding the necessary input and output parameters.

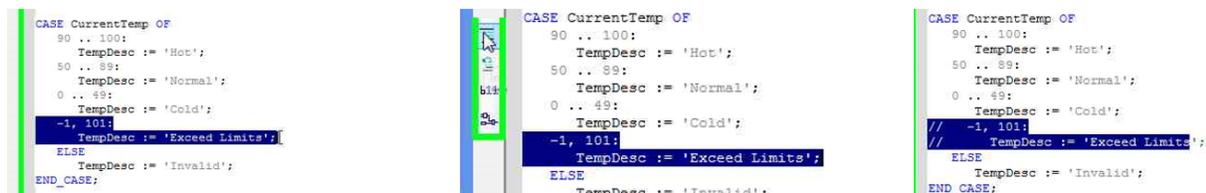
List Key Words

Lists the available keywords. This includes the list of operators, keywords, preprocessor statements, and built in functions. The selected keyword can be pasted in the project by pressing the checkbox or pressing <enter> after a keyword is selected.



Insert Comment

This is a handy way to comment out blocks of a program at a time.



1- Highlight code

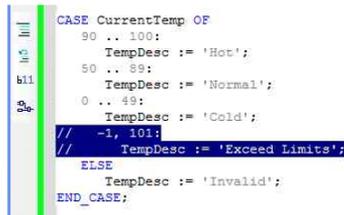
2- Click Insert Comment

3. Comment marks added.

Commenting out a block of code is a common way to trouble shoot a logic block by turning off a chunk of code.

Remove Comment

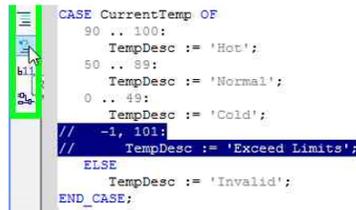
This reverses the insert comment action:



```

CASE CurrentTemp OF
  90 .. 100:
    TempDesc := 'Hot';
  50 .. 89:
    TempDesc := 'Normal';
  0 .. 49:
    TempDesc := 'Cold';
  // -1, 101:
  // TempDesc := 'Exceed Limits';
ELSE
  TempDesc := 'Invalid';
END_CASE;
  
```

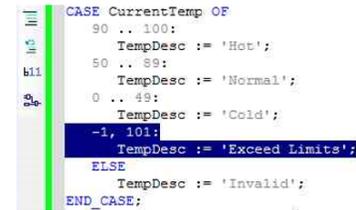
1. Highlight commented code



```

CASE CurrentTemp OF
  90 .. 100:
    TempDesc := 'Hot';
  50 .. 89:
    TempDesc := 'Normal';
  0 .. 49:
    TempDesc := 'Cold';
  // -1, 101:
  // TempDesc := 'Exceed Limits';
ELSE
  TempDesc := 'Invalid';
END_CASE;
  
```

2. Click Remove Comment



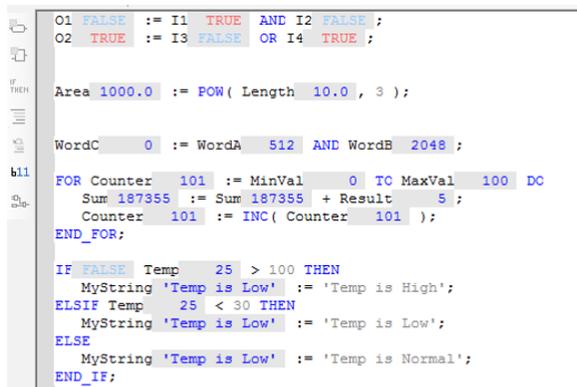
```

CASE CurrentTemp OF
  90 .. 100:
    TempDesc := 'Hot';
  50 .. 89:
    TempDesc := 'Normal';
  0 .. 49:
    TempDesc := 'Cold';
  -1, 101:
    TempDesc := 'Exceed Limits';
ELSE
  TempDesc := 'Invalid';
END_CASE;
  
```

3. Comment marks removed.

Show Value in Text

This button is used when testing the logic using Simulation Mode (click **Mode > Start Simulation > Logic Only**). It shows the value of each variable:



```

O1 FALSE := I1 TRUE AND I2 FALSE ;
O2 TRUE := I3 FALSE OR I4 TRUE ;

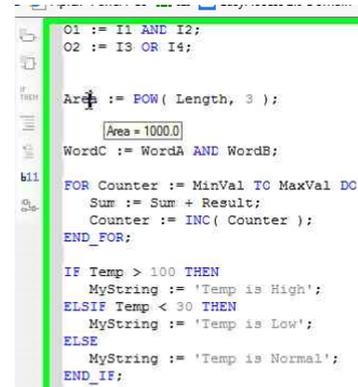
Area 1000.0 := POW( Length 10.0 , 3 ) ;

WordC 0 := WordA 512 AND WordB 2048 ;

FOR Counter 101 := MinVal 0 TO MaxVal 100 DC
  Sum 187355 := Sum 187355 + Result 5 ;
  Counter 101 := INC( Counter 101 ) ;
END_FOR;

IF FALSE Temp 25 > 100 THEN
  MyString 'Temp is Low' := 'Temp is High';
ELSIF Temp 25 < 30 THEN
  MyString 'Temp is Low' := 'Temp is Low';
ELSE
  MyString 'Temp is Low' := 'Temp is Normal';
END_IF;
  
```

- Press the  button to display values in the variables during simulation.



```

O1 := I1 AND I2;
O2 := I3 OR I4;

Area := POW( Length, 3 );
Area = 1000.0

WordC := WordA AND WordB;

FOR Counter := MinVal TO MaxVal DC
  Sum := Sum + Result;
  Counter := INC( Counter );
END_FOR;

IF Temp > 100 THEN
  MyString := 'Temp is High';
ELSIF Temp < 30 THEN
  MyString := 'Temp is Low';
ELSE
  MyString := 'Temp is Normal';
END_IF;
  
```

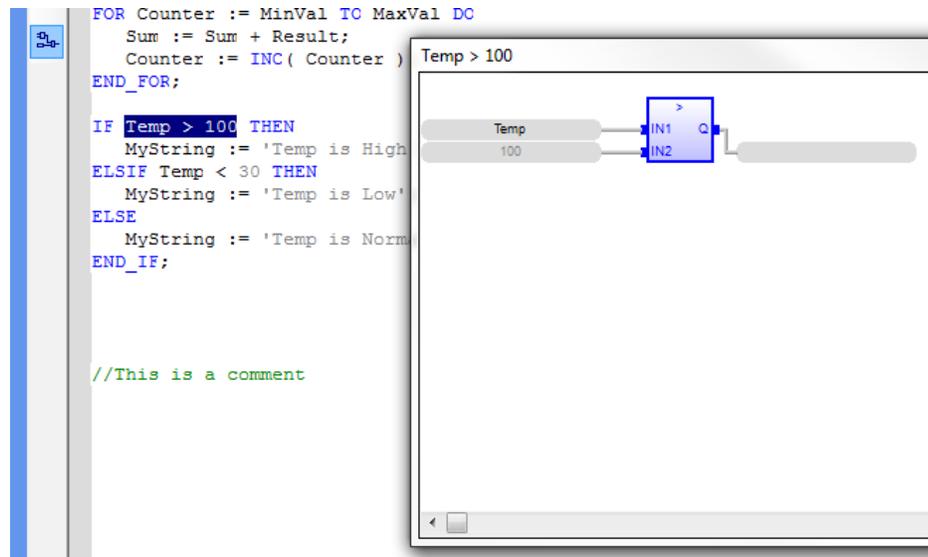
- Press the  button again to go back to normal mode.

In normal mode, you can see a particular variable value by clicking the variable then keeping the cursor on the variable until a popup display appears (see picture above).

Show Expression

When this button is pressed while an expression is highlighted in the code, it will display a popup window that shows the expression in FBD format.

For example, clicking the  button while the expression 'Temp > 100' is highlighted displays:



For programmers who are more experienced with the graphic format of a Function Block, this may help visualize the expression.

Functions, Subroutines and Function Blocks

In addition to using ST keywords to create the flow of statements detailed in the previous section, the primary way to control a program's execution path is by using functions or subroutines. A function is a predefined block of code that is invoked with a function call. When a function call occurs in an ST logic block, the processor jumps to the code in that function and executes it until it reaches the end of the function or a **RETURN** statement. The basic syntax of a function call is:

```
[return value := ] function_name([parameter1], [parameter2], [ > ]);
```

Functions have one (optional) return value, and can have several (optional) input parameters. Some examples of function calls are:

```

// custom subroutine
// No inputs or return value
user_subroutine();

// built in function with input and return value
RealNumber := any_to_real(10);

// UDFB with 1 input and 1 output
Temp1F := C2F(Temp1C);

// UDFB with multiple inputs
fbScaleInit(100, 0, 212, 32, C2F);

```

```

// Built in FB with multiple inputs
// and multiple outputs
MyPIDInstance (auto,
                Pv,
                Sp,
                xout_manu,
                Kp,
                Ti,
                Td,
                Ts,
                xmin,
                xmax,
                i_sel,
                int_hold,
                i_itl_on,
                i_itlval,
                deadb_err,
                ffd
);
x_out := MyPIDInstance.Xout;
er := MyPIDInstance.ER;
x_out_p := MyPIDInstance.Xout_P;
x_out_i := MyPIDInstance.Xout_I;
x_out_d := MyPIDInstance.Xout_D;
x_out_hlm := MyPIDInstance.Xout_HLM;
x_out_llm := MyPIDInstance.Xout_LLM;

```

Functions vs. Function Blocks

In MAPware-7000 IEC programming, there are two basic types of functions that can be called from ST programs.

- Ordinary functions – These can be built in functions or user created subroutines
- Function Blocks – These can be built in function blocks or user defined function blocks

The fundamental difference between the two is that function blocks have instances, while ordinary functions do not. Because of this difference, calls to ordinary functions are somewhat different than calls to function blocks.

Ordinary Function Calls

Ordinary function can have multiple inputs but only one output. To call an ordinary function you use the function name, as it appears in the Instruction List for built in functions, or in the Subroutine folder for user created subroutines. If there are input parameters, you can pass in tags, or literal values in a comma separated list within the parentheses following the function name. If there is a return value. You can use the assignment operator before the function call to assign the return value to a local tag:

```

// custom subroutine
TagB := MySubroutine(TagA);

// built in function with input and return value
RealNumber := any_to_real(10);

```

Because of the limit of one output, or return value, for ST functions, if you are defining a subroutine that you want to call from an ST program make sure to only define one output parameter. Calling a subroutine that has more than one output will cause a compile error.

Function Block Calls

Since Function Blocks have instances with their own set of data, when you want to call a function block you make the call using the name of the function block instance as it appears in the Function Block Instance folder. Do not use the name of the function block; listed in the UDFB folder for user defined function blocks, or in the Instruction List for built in function blocks.

For those familiar with object oriented programming in other programming languages, a function block in MAPware-7000 can be thought of as an object that has multiple properties, but only one method. Using the function block instance name as a function call invokes the method.

If the function block has only one output parameter you can use the assignment operator to assign the output value as with ordinary function calls.

```
// UDFB with 1 input and 1 output
Temp1F := C2F(Temp1C);
```

If there are multiple output this will cause a compile error. The outputs of the function block can still be accessed however, using the following syntax:

< Tag > := <Function Block Instance Name>.<property name>

For example a call to the built in PID function block will look like this:

```
// Built in FB with multiple inputs
// and multiple outputs
MyPIDInstance(auto,
               Pv,
               Sp,
               xout_manu,
               Kp,
               Ti,
               Id,
               Ts,
               xmin,
               xmax,
               i_sel,
               int_hold,
               i_itl_on,
               i_itlval,
               deadb_err,
               ffd
);
x_out := MyPIDInstance.Xout;
er := MyPIDInstance.ER;
x_out_p := MyPIDInstance.Xout_P;
x_out_i := MyPIDInstance.Xout_I;
x_out_d := MyPIDInstance.Xout_D;
x_out_hlm := MyPIDInstance.Xout_HLM;
x_out_llm := MyPIDInstance.Xout_LLM;
```

The input parameters are passed in using the parameter list. The results are accessed by assigning the output properties to standard tags.

Using this '.' operator, it is possible to use properties of a function block instance as variables in ST programs, or as parameters in other function calls.

```
// custom subroutine  
TagB := MySubroutine(C2F.Output);
```

And function block instances themselves can be used as parameters, provided the function block or function is defined to accept them:

```
// UDFB with multiple inputs  
fbScaleInit(100, 0, 212, 32, C2F);
```

Chapter 7 – Instruction List (IL)

Overview

Instruction List (IL) programming is the most basic of the five IEC 61131-3 programming languages. The format consists of a series of simple text statements. Each statement performs only one function.

Most instructions in IL are carried out using the accumulator (also known as current value). Instructions like Load (LD) and Addition (ADD) put a value into the accumulator according to the instruction. Other instructions like Store (ST) or Jump with Carry (JMPC) perform some action according to the value in the accumulator. Other instructions like Call Function Block (CAL) or Jump to Label (JMP) perform some action regardless of the current value in the accumulator.

Adding Logic to the Block

ST Instructions supported

When creating an IL logic block, you must put all IL commands in between the BEGIN_IL and END_IL sections. Above and below this block, you are allowed to use ST instructions.

```
O1 := I1 AND I2; // ST instructions may be placed in an IL logic block above this point.

BEGIN_IL

(* enter your IL program here *)
LD( Int1 (* in ST: Int5 := (Int1 + (Int2 * Int3)-Int4); *)
ADD( Int2
MUL Int3 )
SUB Int4 )
ST Int5

LD Word1 // HI
MAKEDWORD Word2 // LO
ST DWord1 // Q

END_IL

O1 := I1 AND I2; // ST instructions may be placed in an IL logic block below this point.
```

Instruction List (IL) Instructions

The table below shows the Instructions that can be used on the Instruction List editor:

Instruction	Definition	Description	Example
LD	Loads the operand into the accumulator. The operand can be a tag variable of any data type or a constant value. Note: When using the LD instruction with the ST instruction, the Data Types must match.		LD Bool1 LD Int1 LD Real1 LD String1 LD 45 (ST must be DINT) LD 345.87 (ST must be REAL) LD 'Hello' (ST must be STRING) LD TRUE (ST must be BOOL)
LDN	Loads the negated value of the operand into the accumulator. The operand must be data type BOOL.		LDN Bool2 {if Bool2 is TRUE, then acc. is FALSE}

Instruction	Definition	Description	Example
AND (&)	Boolean ANDs the value stored in the accumulator with the tag variable and stores the result in the accumulator.	B1 B2 & B3 0 0 0 0 1 0 1 0 0 1 1 1	LD Bool1 AND Bool2 ST Bool3
ANDN	Boolean ANDs the value stored in the accumulator with the negated value in the tag variable and stores the result in the accumulator.	— B1 B2 & B3 0 0 0 0 1 0 1 0 1 1 1 0	LD Bool1 ANDN Bool2 ST Bool3
OR	Boolean ORs the value stored in the accumulator with the tag variable and stores the result in the accumulator.	B1 B2 B3 0 0 0 0 1 1 1 0 1 1 1 1	LD Bool1 OR Bool2 ST Bool3
ORN	Boolean ORs the value stored in the accumulator with the negated value in the tag variable and stores the result in the accumulator.	— B1 B2 B3 0 0 1 0 1 0 1 0 1 1 1 1	LD Bool1 ORN Bool2 ST Bool3
XOR	Boolean exclusive ORs the value stored in the accumulator with the tag variable and stores the result in the accumulator.	B1 B2 ⊕ B3 0 0 0 0 1 1 1 0 1 1 1 0	LD Bool1 XOR Bool2 ST Bool3
XORN	Boolean exclusive ORs the value stored in the accumulator with the negated value in the tag variable and stores the result in the accumulator.	— B1 B2 ⊕ B3 0 0 1 0 1 0 1 0 0 1 1 1	LD Bool1 XORN Bool2 ST Bool3
ADD	Adds the value stored in the accumulator with the value in the tag variable and stores in the accumulator. Note: the current accumulator value and tag variable must be the same data type.	ST Equivalent: Dint3 := Dint1 + Dint2;	LD Dint1 ADD Dint2 ST Dint3
SUB	Subtracts the value in the tag variable from the value stored in the accumulator and stores in the accumulator. Note: the current accumulator value and tag variable must be the same data type.	ST Equivalent: Dint3 := Dint1 - Dint2;	LD Dint1 SUB Dint2 ST Dint3
MUL	Multiplies the value stored in the accumulator with the value in the tag variable and stores in the accumulator. Note: the current accumulator value and tag variable must be the same data type.	ST Equivalent: Dint3 := Dint1 * Dint2;	LD Dint1 MUL Dint2 ST Dint3
DIV	Divides the value in the accumulator by the value stored in the tag variable and stores in the accumulator. Note: the current accumulator value and tag variable must be the same data type.	ST Equivalent: Dint3 := Dint1 / Dint2;	LD Dint1 DIV Dint2 ST Dint3
GT	Compares the value in the accumulator with the value in the tag variable. If the accum. value is greater than the tag variable, then a value of 1	ST Equivalent: Bool1 := Int1 > Int2;	LD Int1 GT Int2 ST Bool1

Instruction	Definition	Description	Example
	(true) is placed into the accumulator. If the accum. value is less or equal to the tag variable, then a value of 0 (false) is placed into the accumulator. Note: the current accumulator value and tag variable must be the same data type.		
GE	Compares the value in the accumulator with the value in the tag variable. If the accum. value is equal or greater than the tag variable, then a value of 1 (true) is placed into the accumulator. If the accum. value is less than the tag variable, then a value of 0 (false) is placed into the accumulator. Note: the current accumulator value and tag variable must be the same data type.	ST Equivalent: Bool1 := Int1 >= Int2;	LD Int1 GE Int2 ST Bool1
LT	Compares the value in the accumulator with the value in the tag variable. If the accum. value is less than the tag variable, then a value of 1 (true) is placed into the accumulator. If the accum. value is greater or equal to the tag variable, then a value of 0 (false) is placed into the accumulator. Note: the current accumulator value and tag variable must be the same data type.	ST Equivalent: Bool1 := Int1 < Int2;	LD Int1 LT Int2 ST Bool1
LE	Compares the value in the accumulator with the value in the tag variable. If the accum. value is equal or less than the tag variable, then a value of 1 (true) is placed into the accumulator. If the accum. value is greater than the tag variable, then a value of 0 (false) is placed into the accumulator. Note: the current accumulator value and tag variable must be the same data type.	ST Equivalent: Bool1 := Int1 <= Int2;	LD Int1 LE Int2 ST Bool1
EQ	Compares the value in the accumulator with the value in the tag variable. If the accum. value is equal to the tag variable, then a value of 1 (true) is placed into the accumulator. If the accum. value is not equal to the tag variable, then a value of 0 (false) is placed into the accumulator. Note: the current accumulator value and tag variable must be the same data type.	ST Equivalent: Bool1 := Int1 = Int2;	LD Int1 EQ Int2 ST Bool1
NE	Compares the value in the accumulator with the value in the tag variable. If the accum. value is not equal to the tag variable, then a value of 1 (true) is placed into the accumulator. If the accum. value is equal to the tag variable, then a value of 0 (false) is placed into the accumulator. Note: the current accumulator value and tag variable must be the same data type.	ST Equivalent: Bool1 := Int1 <> Int2;	LD Int1 NE Int2 ST Bool1
Function Call	Use a function from the Keywords IF THEN List. Note: not all functions are supported in IL programming.	ST Equivalent: Real3 := POW(Real1, Real2);	LD Real1 POW Real2 ST Real3
Parenthesis	Determines the order in which mathematical operators (ex. +, -, *, /, <, AND, NOT, etc.) are processed in a complex instruction.	ST Equivalent: Int5 := (Int1 + (Int2* Int3) - Int4);	LD(Int1 ADD(Int2 MUL Int3) SUB Int4)

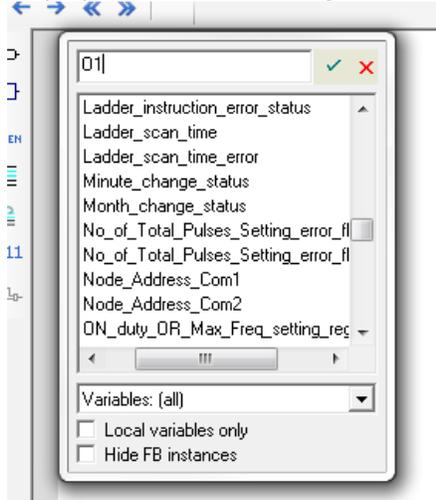
Instruction	Definition	Description	Example
ST	Loads the contents of the accumulator into the operand. The operand can be a tag variable of any data but, generally, must match the data currently stored in the accumulator.		ST Int5 ST Bool1 ST Int1 ST Real1 ST String1
STN	Loads the negated contents of the accumulator into the operand. The operand can be a tag variable of any data but, generally, must match the data currently stored in the accumulator.	Ex. If accumulator = 0, then STN Bool1 would put 1 in Bool1.	STN Bool1 STN Int1 STN Real1 STN String1
JMP	Jumps to the specified label. Contents of accumulator are ignored.	<pre> Start: LD Sint1 // Sub if Sint1 < 0, Add if Sint1 = 0, M > 0 JMPC MultiplyVars LD Sint1 < 0 JMPC SubtractVars // jump to SubtractVars if Bool1 is T AddVars: LD Int1 // otherwise, add the two variables ADD Int2 ST Int3 JMP TheEnd SubtractVars: LD Int1 // subtract Int2 from Int1 SUB Int2 ST Int3 JMP TheEnd MultiplyVars: LD Int1 // multiplies Int1 and Int2 MUL Int2 ST Int3 TheEnd: </pre>	
JMPC	Jumps to the specified label, if the accumulator is TRUE.		
JMPNC or JMPCN	Jumps to the specified label, if the accumulator is FALSE.		
Label:	Places a label in front of the instruction.		
S	Sets the operand to TRUE, if the accumulator is TRUE.	Note: operand remains TRUE after accumulator transitions from TRUE to FALSE.	<pre> LD Bool1 FALSE S Bool2 FALSE ↓ LD Bool1 TRUE S Bool2 TRUE ↓ LD Bool1 FALSE S Bool2 TRUE </pre>
R	Sets the operand to FALSE, if the accumulator is TRUE.	Note: operand remains FALSE after accumulator transitions from TRUE to FALSE.	<pre> LD Bool1 FALSE R Bool2 FALSE ↓ LD Bool1 TRUE R Bool2 FALSE ↓ LD Bool1 FALSE R Bool2 FALSE </pre>
CAL	Calls a function block (locate in an FBD logic block). Contents of accumulator are ignored.	<pre> IL_Block CAL MyTimer(StartTmr, TmrPreset) (* call MyTimer located in another logic bloc LD TmrElapsed // Let's see if the Timer is done yet JMPC TimerDone // Yep, so let's go to TimerDone routine JMP TheEnd // Nope, so let's wait and do something els TimerDone: // Timer is done so time to take cookies out c LD TRUE ST CookiesReady TheEnd: END_IL </pre>	
CALC	Calls a function block, if the accumulator is TRUE.		
CALNC or CALCN	Calls a function block, if the accumulator is FALSE.		
		<p>The screenshot shows a function block named 'MyTimer' in an FBD logic block. It has two input terminals: 'IN' (TON) and 'PT' (TmrPreset). It has two output terminals: 'Q' (TmrElapsed) and 'ET' (ElapsedTime).</p>	

Quick Select Menu Options

The sections that follow detail the options available in the Quick Select Menu of the IL editor.

Insert Variable

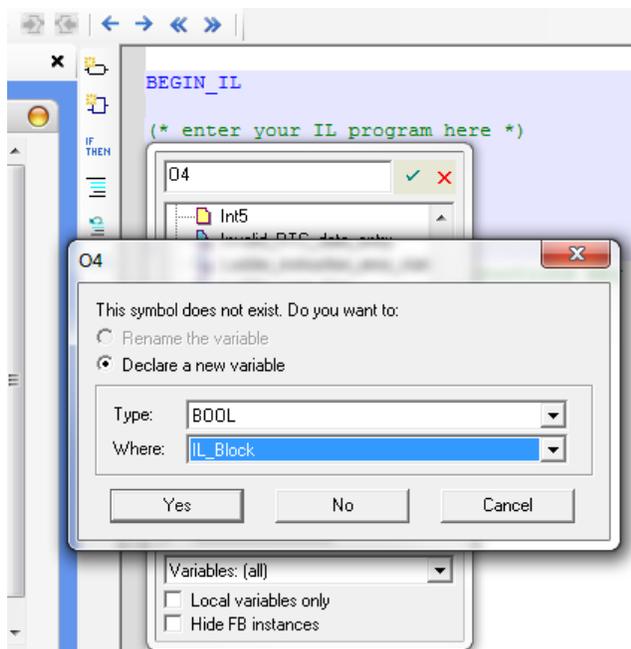
Displays the Variable Dialog box:



The Variable Dialog box is used to select/create a tag variable and place it onto the IL work area.

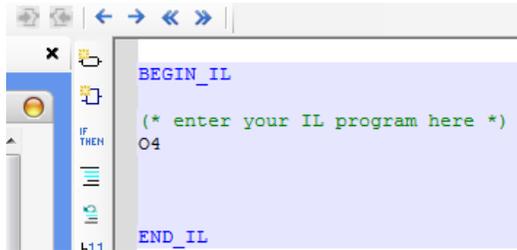
By default, all of the System Tags, global tags, and Local tags for the selected logic block are listed. You can select a tag from the list or create a new one by typing a new tag name, then clicking the Accept  button. *Note: if you only want to see local tags for the selected logic block, check the Local variables only box.*

In this example, we will add a local Boolean variable O4. If the tag is new, clicking the Accept  button displays a new dialog:



When creating a new variable, you must determine the data type (i.e. BOOL, INT, STRING, etc.) and visibility of the variable. If Global is selected, the variable can be used in any of the logic blocks you create. If you select Local (by selecting the name of the logic block you are in), then the tag variable can only be used and seen by the logic block. Selecting RETAIN stores the tag in non-volatile memory of the MLC so that the value is retained after power is recycled on the MLC.

Click Yes to return to the logic block work area:



The variable is placed into the IL code area and is also stored in the Tags library.

Tag No	Tag Name	Data Type	Attribute	Tag Address	Por
20	IL_Block/Int3	INT (L)	Read Write	-	-
27	IL_Block/Int4	INT (L)	Read Write	-	-
28	IL_Block/Int5	INT (L)	Read Write	-	-
2	IL_Block/O1	BOOL (L)	Read Write	-	-
4	IL_Block/O2	BOOL (L)	Read Write	-	-
9	IL_Block/O3	BOOL (L)	Read Write	-	-
44	IL_Block/O4	BOOL (L)	Read Write	-	-
10	IL_Block/Real1	REAL (L)	Read Write	-	-

In IL Programming, you may prefer to write the code statements, then assign variables afterwards.

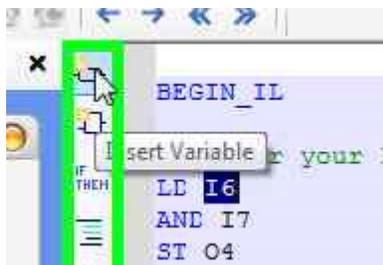
For example, let's add a statement to the variable O1 that we already created by typing:

```
BEGIN_IL
(* enter your IL program here *)
LD I6
AND I7
ST O4
```

If we try to compile, we will get errors because the new variables, I6 and I7 are not yet added to the Tag library:

```
IL_Block
IL_Block: (5): I6: Operand expected after instruction
IL_Block: (6): Action expected (ST, S, R, JMPC, CALC...) after expression loading
IL_Block: (6): AND: Bad or unexpected statement
IL_Block: (11): #endif: "=" expected after assigned variable
Error(s) detected
```

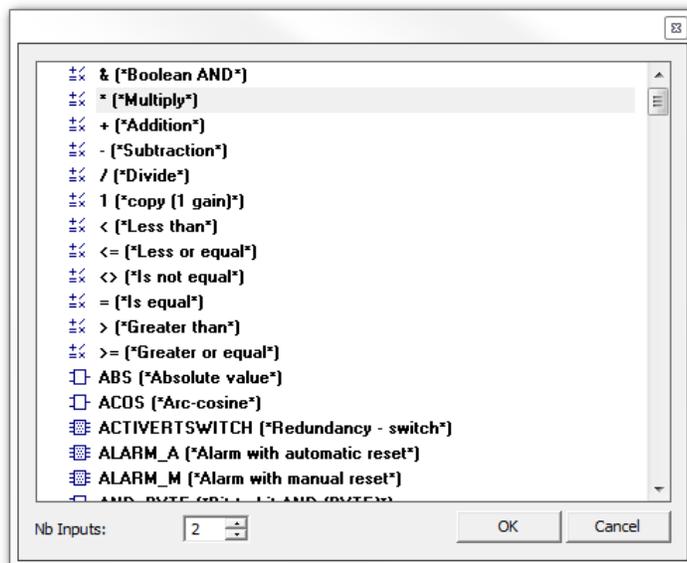
You can add a new variable that has already been written into the code by double-clicking the variable (to highlight the entire name), then clicking the Insert Variable button:



Once a variable has been added to the Tag library, you do not have to use the Insert Variable button again to reference the tag- simply type the name of the tag into the code.

Insert (FB) Function Block

Click to add a function block (see Block Name section on right side of work area) to the logic code work area. The Function Block dialog box appears:

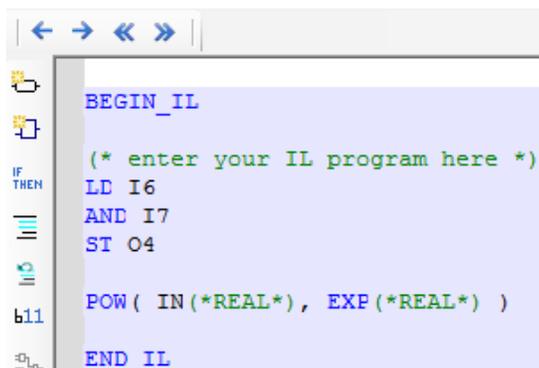


Select the Function Block that you wish to use. With some functions, you can change the number of inputs using the Nb Inputs option. Since IL programming is limited, not all functions are supported.

Suppose we want to calculate the volume of various cubes using the power (pow) function. For a cube, this would simply be the length of one of the sides to the power of 3:

Area := POW(length, 3);

With IL Programming, you could click the Insert FB button, then click the POW function block:



The MAPware-7000 software indicates that the POW function requires two input parameters: IN and EXP. Both parameters must be Data Type Real and the result must be stored in a variable of Data Type Real. So we will create three variables for this function: Area, Length, and Cube- all configured as REAL.

With IL programming, you cannot load more than one input value into the accumulator at one time. When using a function that requires two inputs in IL programming, you generally load the first variable into the accumulator, then bring in the second variable when invoking the function:

```
// POW( IN(*REAL*), EXP(*REAL*) )
LD Length
POW Cube
```

To finish, you would use another variable to store the result:

```

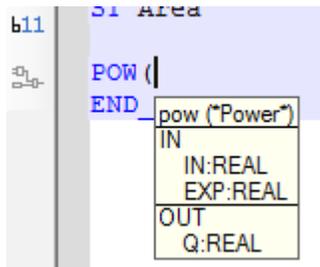
11 // POW( IN(*REAL*), EXP(*REAL*) )
12 LD Length
13 POW Cube
14 ST Area

```

Note: To determine what data type is required for any outputs of a function block, double-click the function block in the Block List to display the help file for that function block.

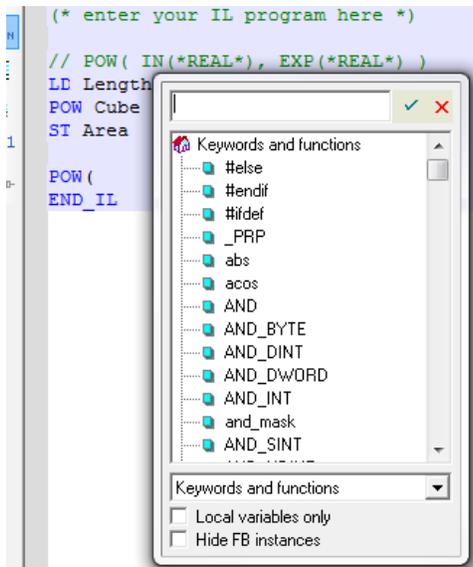
As with entering a variable, when using IL programming, it is acceptable to type in the statement- including all variables and the function. After pressing the ENTER key, MAPware-7000 checks all variables and displays the Variable Dialog Box if any variables are new.

When typing the function block directly into the work area, you will also see a popup box that displays the proper format for the selected function block:



List Key Words

The selection tool displays the Variable Dialog box with the Keywords and Functions option selected:



Keywords are words that have a predefined meaning in the ST programming language. Keywords are not supported in the IL programming language. Functions such as AND, POW, or any_to_uint are function blocks. They usually require at least one input parameter and one output parameter.

Insert Comment

This key places double-slash marks at the beginning of any highlighted text or line of code (see ST Programming [section](#) for more details).

Remove Comment 

Removes double-slash marks at the beginning of any highlighted text or code. (See ST Programming [section](#) for more details).

Show Value in Text 

This button is used when testing the logic using Simulation Mode (click **Mode > Start Simulation > Logic Only**). It shows the value of each variable (see ST Programming [section](#) for more details).

Show Expression 

When this button is pressed while an expression is highlighted in the code, it will display a popup window that shows the expression in FBD format. This tool is not very useful when programming using IL, it is more appropriate when using ST programming language (see ST Programming [section](#) for more details).

Chapter 8 – Sequential Function Chart (SFC)

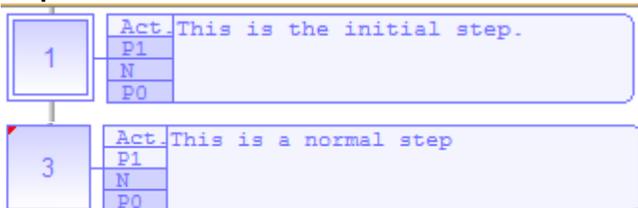
Overview

Sequential Function Chart (SFC) is a graphical logic editor. The SFC editor models machine operation in the form of a flow chart or state diagram. The programmer defines what happens in each step or state, the conditions to transition from one step to the next, and how steps are connected to one another. Graphical steps are used to represent stable states, and transitions describe the conditions and events that lead to a change of state. Machines that perform a sequence of repetitive operations over time are well suited to be programmed using SFC.

The workbench fully supports SFC programming with several hierarchical levels of charts: i.e. a chart that controls another chart. Each SFC program may have one or more "child programs". Child programs are written in SFC and are started (launched) or stopped (killed) in the actions of the parent program. A child program may also have children. The number of hierarchy levels should not exceed 19. Working with a hierarchy of SFC charts is an easy and powerful way to manage complex sequences and save run time performance.

The basic building blocks of a SFC logic block are:

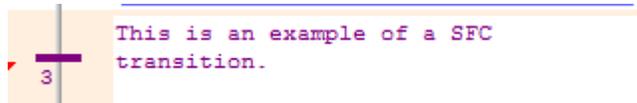
- **Steps**



Steps define the actions carried out by the program. The programmer can define actions that take place:

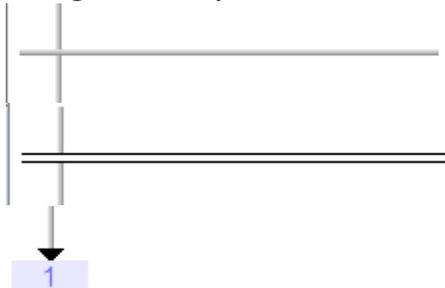
- When a step is entered
- While in the active step
- When the active step is exited

- **Transitions**



Transitions define the conditions that need to be met in order to move from the current step to the next. The transitions act as gates. Until the transition condition is met, the step connected before (above) the transition remains the active step.

- **Divergences/Jumps**



Divergences and jumps determine how steps and transitions connect together. The selection of a divergence type can allow:

- One step to connect to multiple transitions. This allows selection between several steps: the transition condition that is met first determines which of the step will become active next.
- One transition to execute multiple steps in parallel (i.e. at the same time).
- A series of steps, or network of steps to be repeated (using a jump)
- A series of steps to converge into a single step.

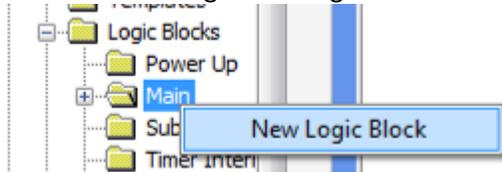
The task of the SFC programmer is to connect these building blocks together into a network modeling the functionality of the process being controlled.

Adding Logic to the Block

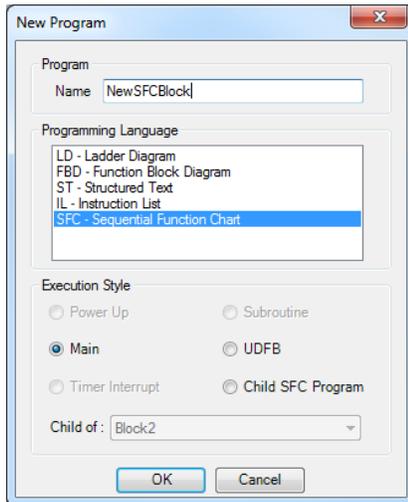
Building a Simple SFC Diagram

For those new to SFC programming this section will cover the minimum steps needed to create a very simple SFC diagram in the editor.

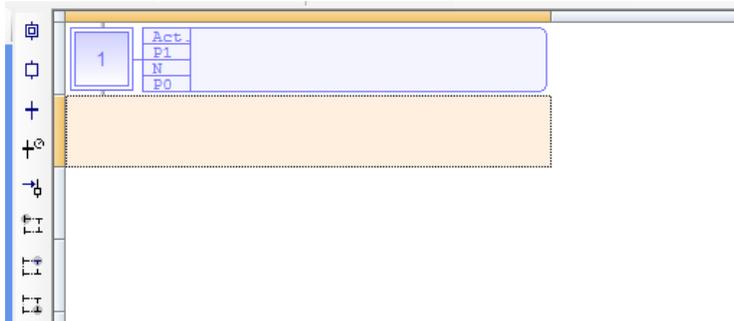
To create a new SFC logic block right-click the Main folder within the Logic Blocks folder of the Project Tree.



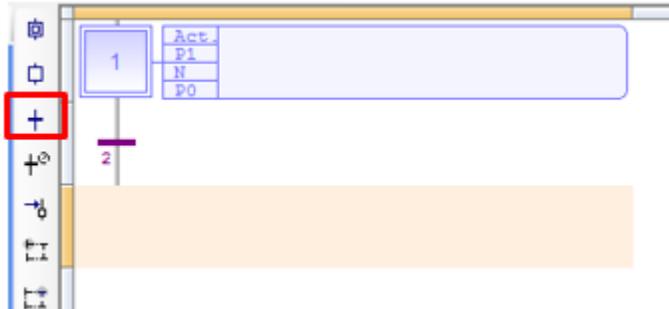
Enter a name for the logic block and select **SFC – Sequential Function Chart** for the programming language, then click **OK** to create the new block.



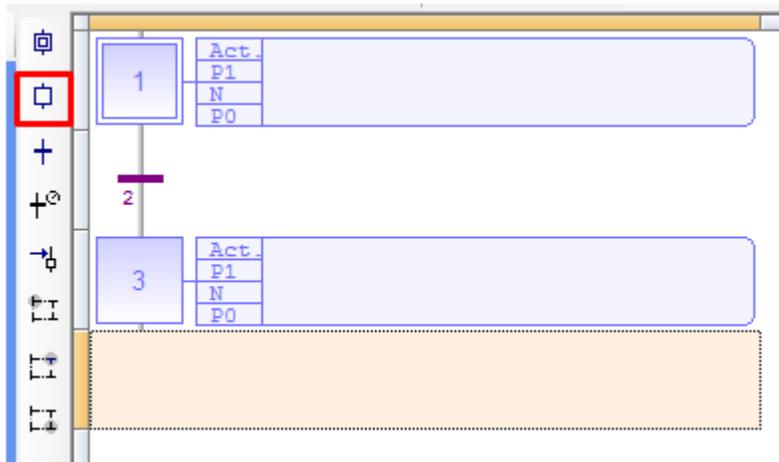
The entry point to any SFC diagram is the initial step. MAPware-7000 will create and place the initial step in the top left corner when a new SFC logic block is created:



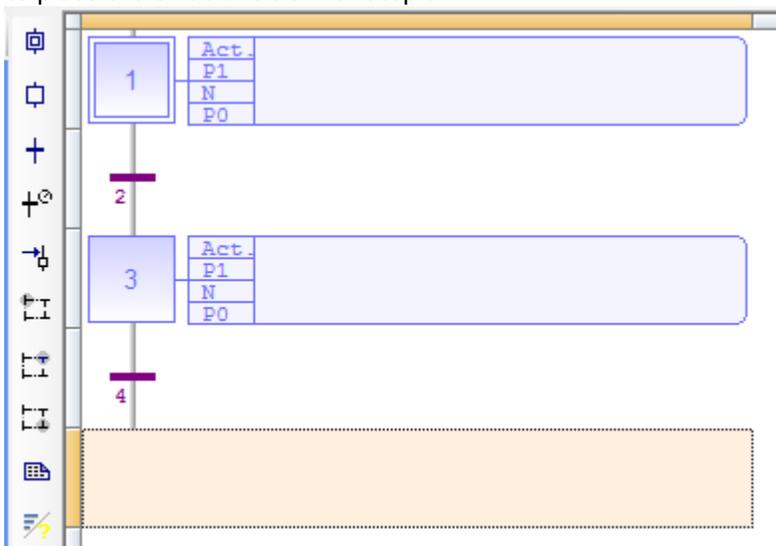
The rule for exiting a step is that it must connect to either a transition or a divergence; a step cannot connect directly to another step. To keep this chart simple, we will use a transition. Click the editor square below the initial step to select it. The selected square will be highlighted in a light orange color as shown in the screen shots. Then click the Transition icon in the Quick Select menu: **+**. This will place a transition in the selected square.



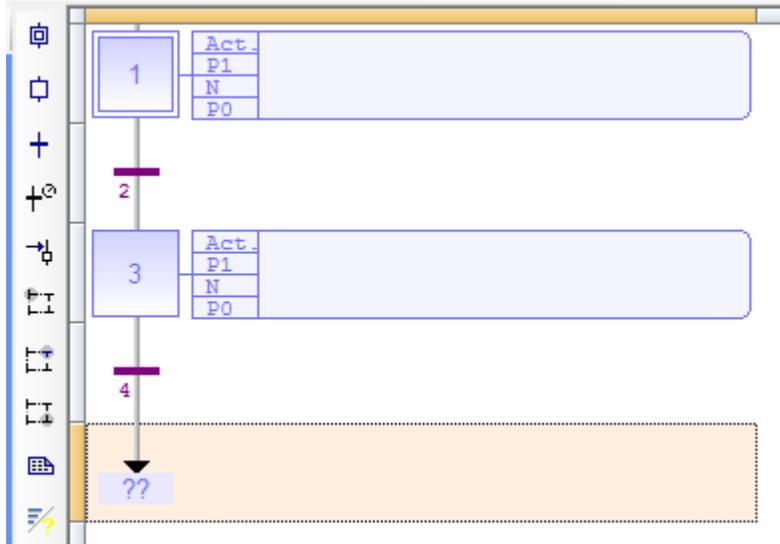
Now that we have a transition, we can add another step. The editor should automatically select the square below the transition as shown in the screen shot above. With this square selected, click the Step icon (**□**) from the quick select menu to add a step to the chart.



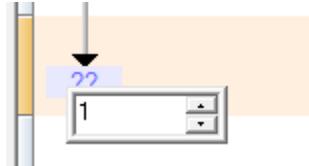
Note that steps and transitions are designated with a number by the editor. Click the transition icon (**+**) to place the exit transition for Step 3.



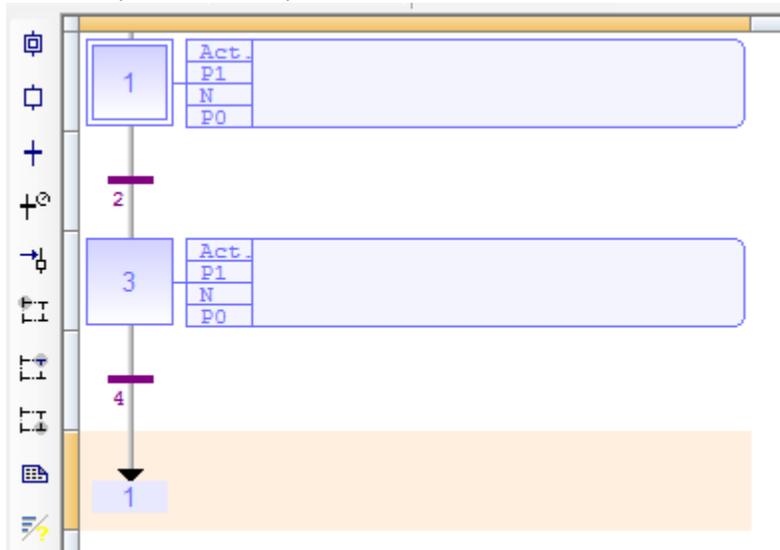
Finally, complete the chart with a Jump back to the initial step. Click the Jump icon  to place the Jump below Transition 4.



The Jump is placed with ?? for the destination. To tell the Jump where to jump to, double-click the question marks and enter '1' to direct the jump back to the initial step.



That completes the simple SFC.



The active step will simply alternate between Step 1 and Step 3 in an endless loop. Of course, the steps do nothing and the transitions are always true. So, while this logic block will work, it is not particularly useful. Use the sections below to configure the steps and transitions.

Steps

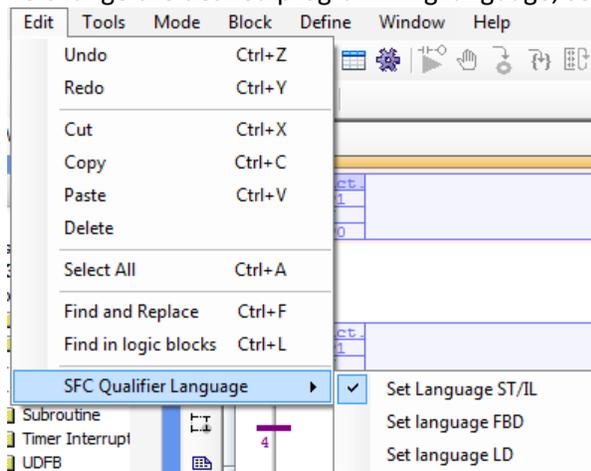
A step represents a stable state and defines what a program does in a given state. Each Step contains four different action types that can be configured independently:

1. **Action:** This is a list of actions that operate as long as the step is active. Actions can be simple boolean or SFC actions that consist of assigning a boolean variable or controlling a child SFC program.
2. **P1:** This is an action block that is executed once when the program is activated. Action blocks can be programmed in any of other IEC languages (FBD, LD, ST, or IL).
3. **N:** This is an action block that operates continuously while the step is active. Action blocks can be programmed in any of other IEC languages (FBD, LD, ST, or IL).
4. **P0:** This is an action block that operates once when the step is deactivated (i.e. the transition condition below the step becomes true). Action blocks can be programmed in any of other IEC languages (FBD, LD, ST, or IL).

Within the SFC program, you can test the step activity by specifying its name ("GS" plus the step number) followed by ".X". Example: *GS3.X* Is TRUE if step 3 is active (expression has the BOOL data type).

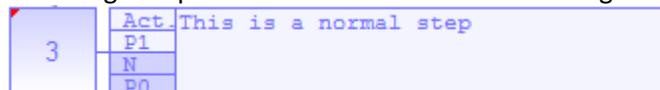
You can also test the active time of a step by specifying the step name followed by ".T". This is the time elapsed since the activation of the step. When the step is de-activated, this time will remain unchanged. It is reset to 0 the next time the step becomes active. Example: *GS3.T* is the time elapsed since step 3 became activate (expression has the TIME data type).

The programming language for the **P1**, **N**, and **P0** action blocks defaults to Structured Text/Instruction List. To change the desired programming language, select **Edit > SFC Qualifier Language** from the main menu.



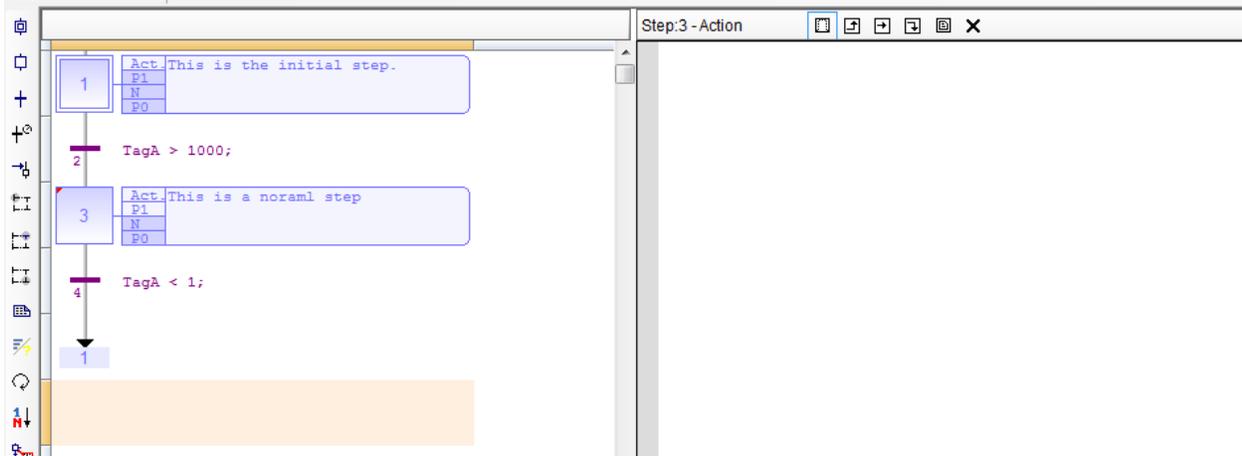
In addition, each step has a **Notes** section containing a simple text editor that allows notes to be entered describing what the step does.

You can get a quick idea of which actions are configured for a step from the step's graphic in the SFC editor:



Actions that have been configured are shown in a darker color, while un-configured actions are lighter. Notes entered in the step's **Note** editor appear in the label to the right of the list of actions. Step 3 above has logic in both the N and P0 action types.

To open the editor for a given step, simply double-click the step. The SFC editor window will split into two panes.



On the left, the SFC diagram is still visible. On the right, the editor for the step is displayed. Zooming in on the top of the editor pane shows the following:

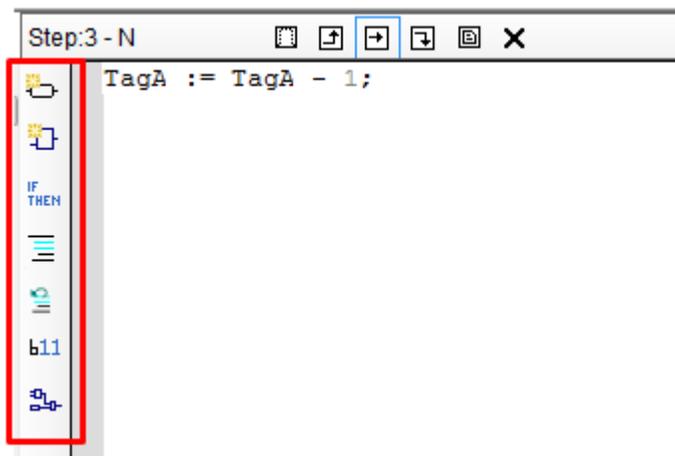


In the top left of this editor pane, the step number is shown. The open editor window is listed immediately after. In this case it shows the step is number 3, and the Action editor is open.

To the right of this there are six clickable icons. These icons function as follows:

- Open the Action editor for the action
- Open the P1 action block editor (P1 = executed once when the step is activated)
- Open the N action block editor (N = executed while the step is active)
- Open the PO action block editor (PO = executed once when the set is de-activated)
- Open the Notes text editor to type in a comment describing what the step is supposed to do
- Close the editor window

Click the N action icon () Note how the Quick Select Menu for the ST Editor is displayed on the left edge of the editor pane:

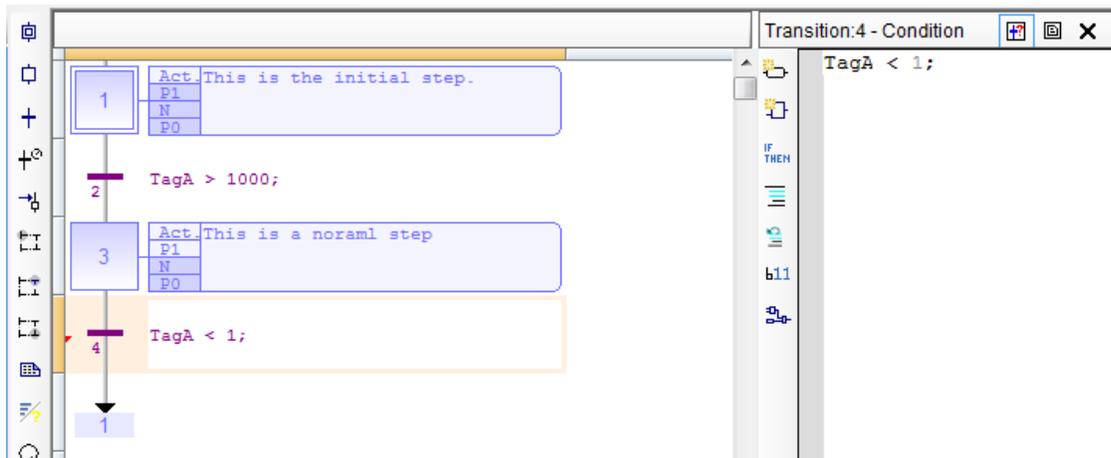


This is because this window defaults to using the ST Editor. The Quick Select Menu and Instruction List can be used to build statements just as if this were a regular ST logic block. To change the programming language for the individual step, select **Edit > SFC Qualifier Language** from the main menu.

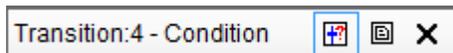
Refer to the previous chapters on the different editor languages to configure the P1, N and P0 actions.

Transitions

Transitions represent a condition that changes the program activity from one step to the next. Each transition must be completed with a condition that evaluates to either TRUE or FALSE to indicate if the transition can be crossed. The condition is a boolean expression that can be programmed either in ST or LD language. Similar to a step, the editor for a transition is opened by double-clicking the transition graphic in the SFC diagram.



The transition editor only has a Condition Editor for the transition condition and a text editor to enter notes for the transition.



To enter the transition condition editor click the condition editor icon (🔍). This will open a ST editor by default as shown above. To change to the LD editor, select it from the **Edit > SFC Qualifier Language** menu.

The transition condition is a very simple program. If there are effects that need to occur when one step is exited and another is entered, the P0 and P1 step action blocks are the proper places to implement that logic. Transition conditions should simply control when a move to the next step occurs.

The Transition condition must consist of a single ST statement or rung of LD code that evaluates to a single Boolean value.

Valid Structured Text Transition Condition Expressions

Because only a single statement is used, the ';' character is optional when using Structured Text. The expression can be composed of multiple expressions, but only one statement is allowed.

Expressions that evaluate to Boolean values generally are expressions that use comparison operators or Boolean logic operators. The table below lists the ST operators that are valid to use for transition conditions:

Operator	Name	Description
func()	Function call	Values are assigned to parameters. Function is executed. Expression evaluated to returned value.
NOT	Complement	Returns the opposite logical value of the Boolean operator on the RHS.
<	Less Than	True when LHS is less than the RHS, false otherwise.

Operator	Name	Description
>	Greater Than	True when the LHS is greater than the RHS, false otherwise
<=	Less than or equal	True when the LHS is less than OR equal to the RHS, false otherwise
>=	Greater than or equal	True when the LHS is greater than OR equal to the RHS, False otherwise
=	Test for equality	True if RHS has the same value as the LHS, False otherwise.
<>	Test for inequality	True if the RHS does not have the same value as the LHS, False otherwise
&, AND	Boolean AND operation	True if LHS AND RHS are true, false otherwise
XOR	Exclusive OR	True if the LHS is true while the RHS is false, or the LHS is false while the RHS is true. False if both LHS and RHS are true. False if both LHS and RHS are false.
OR	Boolean OR	False if both LHS AND RHS are false, true otherwise.

The expression used for the transition expression must evaluate to a Boolean value, however, the expression can be composed of operands that are themselves expressions. In this case, the expressions making up the operands are not limited to using the above operators. For example, the expressions below are all valid transition expressions:

```
TagA = 22
TagA < (TagB - TagC)
TagA > (22 - 4)
(TagA - TagB) * TagC / TagD >= 22;
(A and B) or (TagA - TagB < 10)
// Returns are bool
MySubroutine();
```

Using Function Calls as Transition Conditions

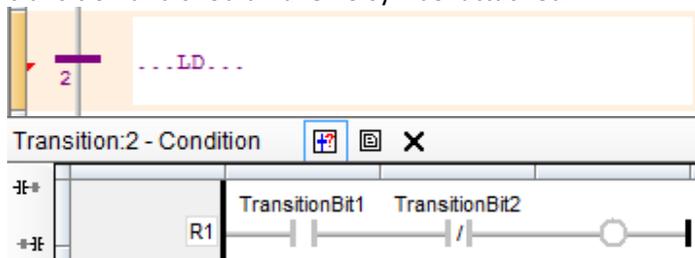
Note that a function call is a valid transition condition expression. The requirement is simply that the function call evaluate to a Boolean value. That is, the function must return a single Boolean value. This is appropriate for complicated calculations that can't be done in a single statement.

A subroutine can be defined, and the subroutine can contain any number of statements, or be constructed using LD or FBD as required. The only requirement is that the subroutine must return a single Boolean value.

When calling the subroutine, do not assign the return value to a tag. The assignment operator is not allowed in transition conditions.

Using Ladder Logic for Transition Conditions

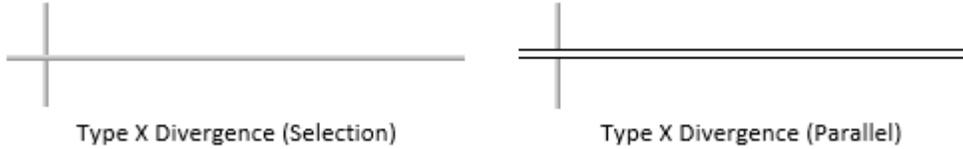
In LD language, the condition is represented by a single rung. The coil at the end of the rung represents the transition and should have no symbol attached.



Divergences

Divergences are what allow SFC programs to branch. Using divergences, one step can lead to multiple steps, a program can select between steps, and multiple steps can converge back to one step. The Quick Select Menu contains options for three types of divergences that are used for different purposes:

- 
Type X divergence - Used to initiate or terminate a SFC branch:



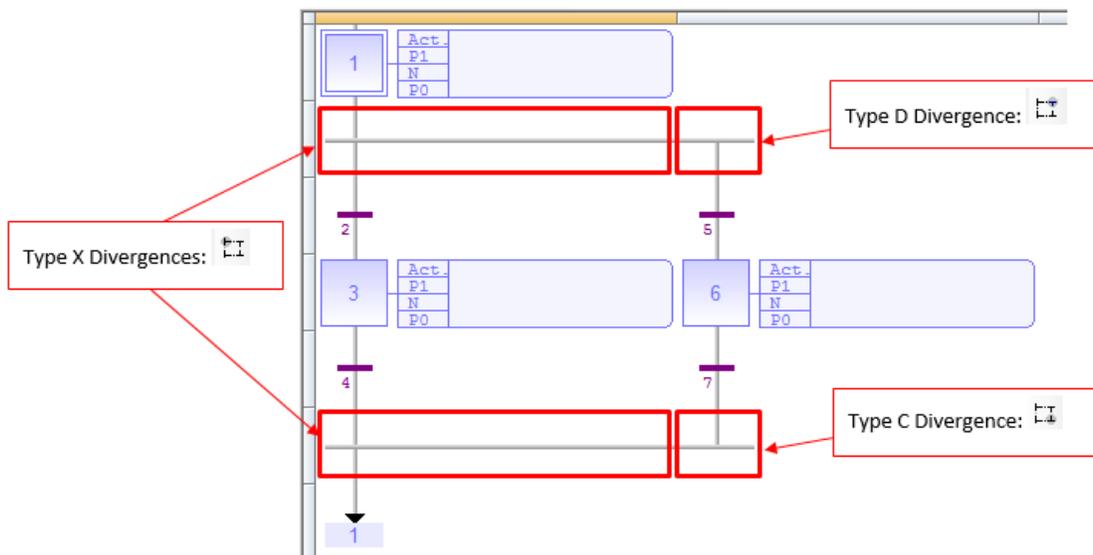
- 
Type D divergence – Used when initiating a SFC branch to provide a connection point to the right of a Type X divergence and above either a transition (in the case of a selection branch), or above a step (in the case of a parallel branch).



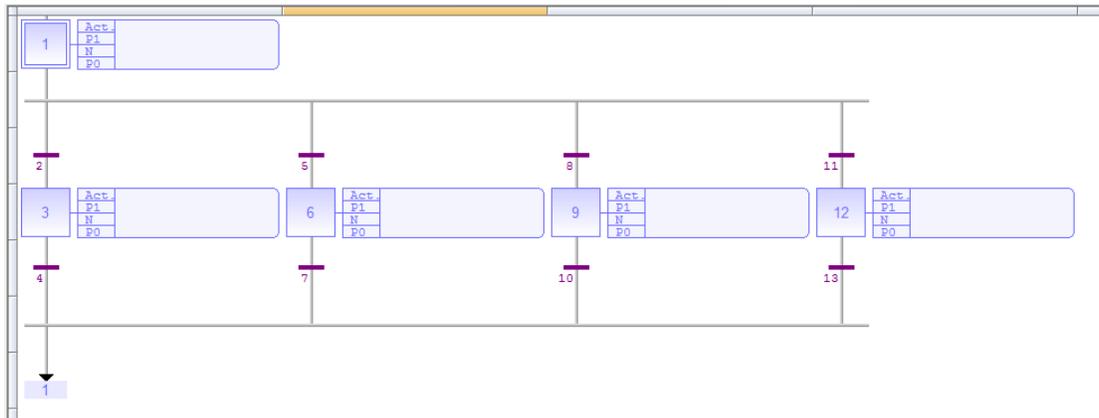
- 
Type C divergence – Used when terminating a SFC branch to provide a connection point to the right of a Type X divergence and below a step or a transition.



The SFC snippet below shows where the different divergence types are used in a typical branch:



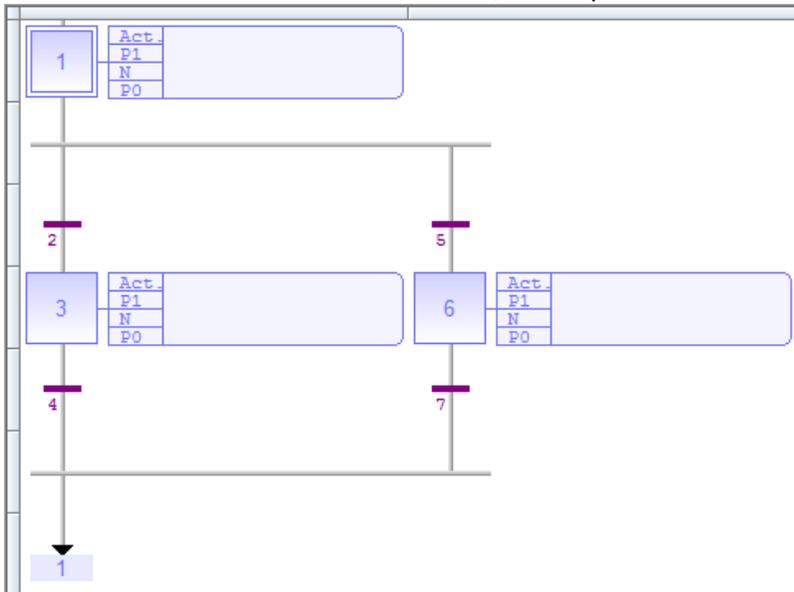
Multiple D and C type divergences can be connected to the right of an X type divergence in order to add more vertical branches to the chart:



There are two types of branches possible in SFC diagrams:

Selection Branches

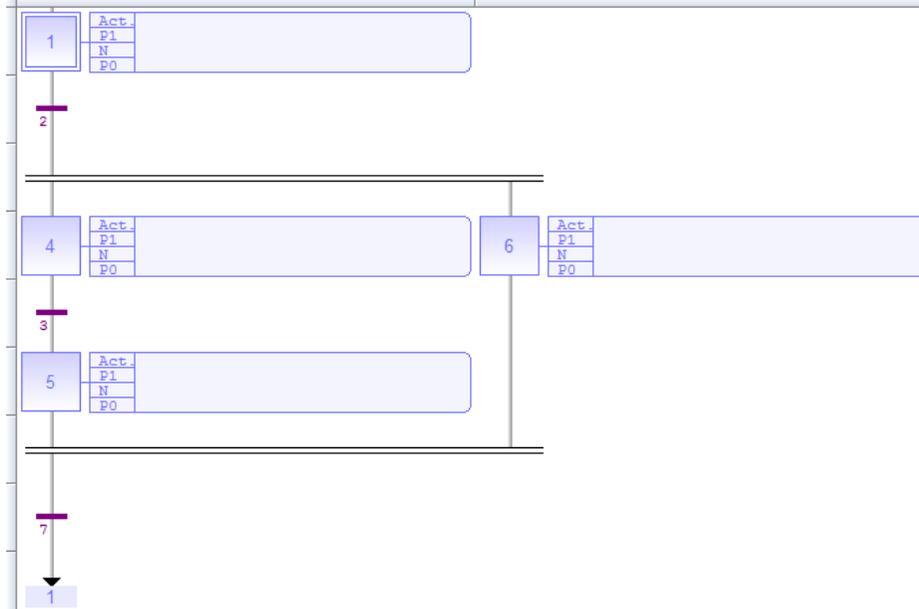
— In a selection branch, one of several alternative steps is executed depending on which transition condition becomes true first. Transition conditions are evaluated from left to right. If multiple transition conditions become true at the same time, the step furthest left will be activated.



Parallel Branches

— In a parallel branch, all of the steps connected to the branch become active simultaneously (in the example below, steps 4 and 6). That is, they are executed at the same time. This is similar to having several logic blocks in the main folder. Each block is executed once per PLC scan. Similarly, each step connected to

a parallel branch will be executed each PLC scan, provided the parallel branch is active.



Processing of parallel branches may take different timing according to each branch execution. The transition after the convergence (transition 7) is crossed when all the steps connected before the convergence line (last step of each branch, 5 and 6) are active. The transition indicates a synchronization of all parallel branches.

If needed, a branch may be finished with an empty step (with no action). It represents the state where the branch "waits" for the other ones to be completed.

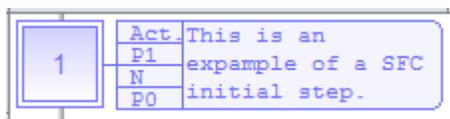
To switch from one type of branch to another, use the Swap Item Style quick select menu item . Notice how a single branch can only have one style. The entire branch will change type when the Swap Item Style icon is pressed.

Quick Select Menu Options

In the SFC menu the Quick Select Menu has the options described in the following sections.

Initial Step <I>

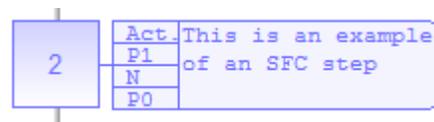
This item places an initial step in the SFC Editor.



By default when a SFC block is created, it will have an initial step in the top left square. SFC diagrams must have at least one initial step. The initial step will be the active step when the SFC block is first evaluated.

Step <S>

Insert a step.

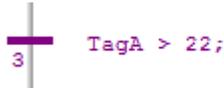


Select the square in the editor where the step is to be placed. Then click this icon to place the step. Steps must be placed after a transition (or a divergence connected to a transition), and steps must be followed by transitions (or a divergence connected to a transition).

See the [Configuring Steps](#) section for information on configuring a step once it is placed.

Transition  **< T >**

Place a transition.

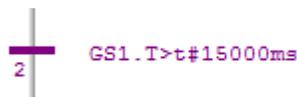


Select the square in the editor where the transition is to be placed then click this icon to place a transition. Transitions must be placed between steps or divergences/jumps that connect to steps. A transition cannot connect to another transition.

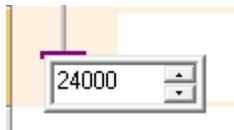
See the [Configuring Transitions](#) section for information on configuring the transition once it is placed.

Set Timer on Transition 

Use this option to place a timer transition.



Select the location in the editor where the timer transition is to be placed, then click this icon. A text box will appear where the preset time (in milliseconds) can be entered. This is the time that the step above the transition will be active.



A timer will automatically be created for the timer transition. The timer begins counting up once the step directly above the transition becomes active. When the timer counter (GSX.T) becomes larger than the preset the transition condition becomes true. The transition condition is automatically formatted to accomplish this.

Jump  **< J >**

Place a Jump.

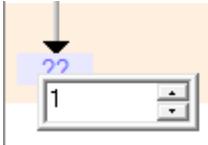


A Jump points to a step without actually being connected to it. Since a Jump connects to a step it must be added directly after a transition. Click the square in the editor where the jump is to be placed, then click this icon to place the jump.

When first placed, the jump will appear with question marks for the destination.



Double-click the question marks to select the destination step.



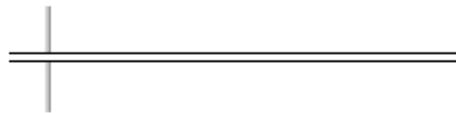
Once configured, the destination can be changed by double-clicking the destination step number again.

Divergence (X) < X >

Place an X type divergence.



Type X Divergence (Selection)



Type X Divergence (Parallel)

To place an X type divergence click the square in the editor where the divergence is to be placed, then click this icon.

Use this type of divergence to start a SFC branch, or to converge several branches. Use the Swap Item Style



quick select option to switch between Selection and Parallel branch styles.

See the [Divergences](#) section above for more information.

Divergence (D) < D >

Place a D type divergence



Type D Divergence (Selection)



Type D Divergence (Parallel)

A D type divergence is used to add a vertical branch to the right of an X type divergence. Multiple D type divergences can be added to create multiple vertical branches.

To place a D type divergence select the block of the SFC editor where the divergence is to be placed then click this icon.

Use the swap item style quick select menu option  to switch between selection and parallel branch styles.

See the [Divergences](#) section above for more information.

Divergence (C) < C >

Place a C type divergence



Type C Divergence (Selection)



Type C divergence (Parallel)

Despite the name a C type divergence is used to converge a series of vertical SFC branches. The C type divergence should be placed to the right of an X type divergence. One C type divergence should be used for each vertical branch that is to be converged.

To place a C type divergence select the block of the SFC editor where the divergence is to be placed then click this icon.

Use the swap item style quick select menu option  to switch between selection and parallel branch styles.

See the [Divergences](#) section above for more information.

Insert Comment

A comment is simply a text box placed in the SFC diagram that is used to annotate the chart.

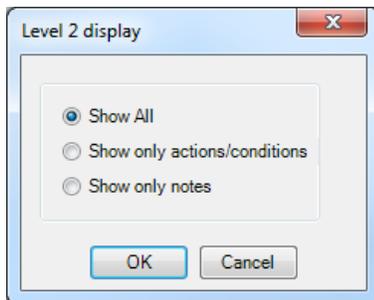


To place a comment select the editor square where the comment is to be placed, then click the Insert Comment icon.

To enter text for the comment Double-click in the comment box and type the text in the editor window.

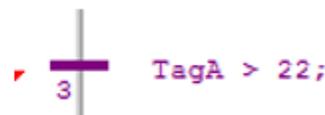
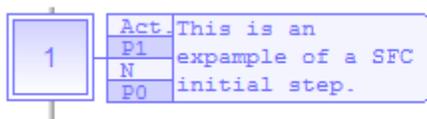
Change Display

This controls how steps and transitions are shown in the SFC editor.

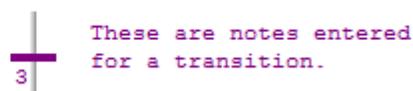
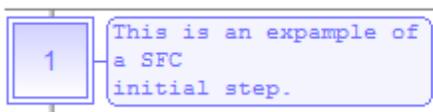


Options are:

- **Show All** – This is the default configuration. For steps, a list of all action types are shown with ones that are used highlighted in a darker shade. The notes are shown to the left of the list of actions. For transitions the condition is shown.



- **Action Only** – This option shows only the Action configured in the step, and the condition configured for a transition. Notes are not shown.
- **Show Only Notes** – Only shows notes. The action list is hidden in steps. And the notes are shown for transitions.

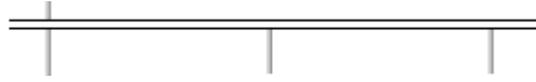


Swap Item Style

Use this option to toggle the style between a selection branch.



And a parallel branch



To use the option select the X Type divergence that terminates the branch, then click this icon.

Note that the entire branch will change when this option is used.

For more information on the difference between selection and parallel style branches refer to the [Parallel vs. Selection Branches](#) section above.

Renumber

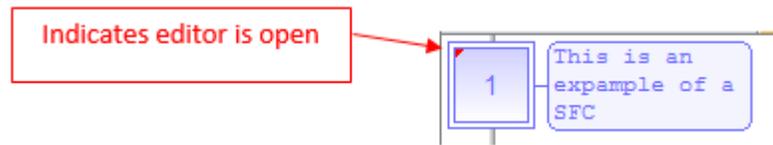
This option will automatically re-assign numbers to all of the elements in the SFC. Simply click this option to auto number the elements.

Note: To renumber an element the editor for that element must be closed.

Elements can be manually assigned numbers using the Edit Reference option below.

Edit Reference

Used to change the reference number of an SFC element. Before using this option, the editor for the element must be closed. A red indicator in the top left corner of the element indicates that the editor for that element is open:



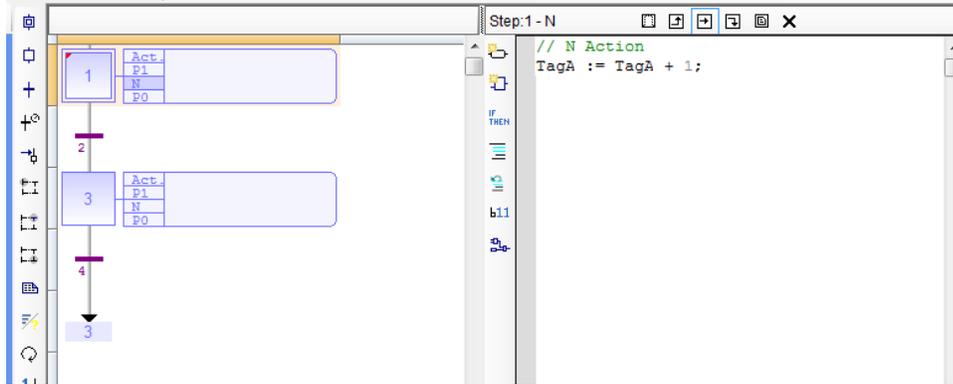
Select the element to change and click the Edit Reference icon. A text box appears where the new reference number can be entered.



Show One Qualifier in Level 2

This option is used to change the view of the editor window for editing step actions. This is the default setting. In this mode the SFC editor is split into two panes. The left pane shows the SFC diagram, and the

right displays an editor for an action:

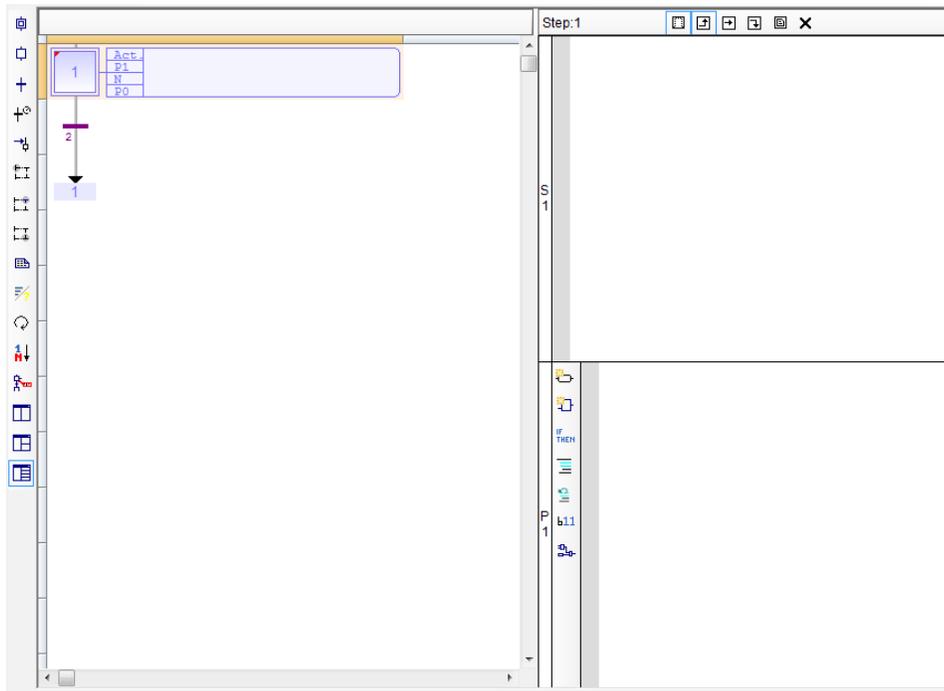


The user must select the action they want to edit using the icons at the top of the editor pane:



Show/Hide Split Window in Level 2

This option is used to change how the editor window for a step is displayed. In this mode, when a step action editor is opened the SFC editor window is split into three sections. The left pane of the editor window continues to display the SFC diagram. The right section is reserved for the action editor. The editor can itself be divided into two sections. When first opened the editor pane still only shows one editor, however the user can click another action icon to split the right pane in two:



When two actions types are open for editing the icons corresponding to those editors are highlighted:

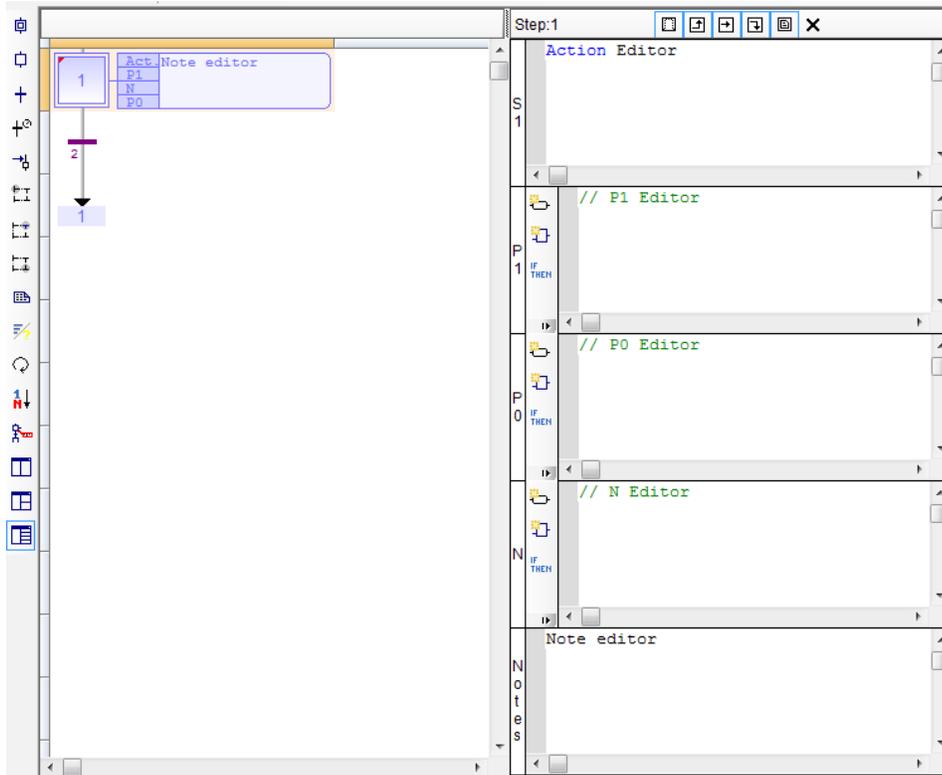


And a bar along the left edge of the editor pane indicates which action is being edited.

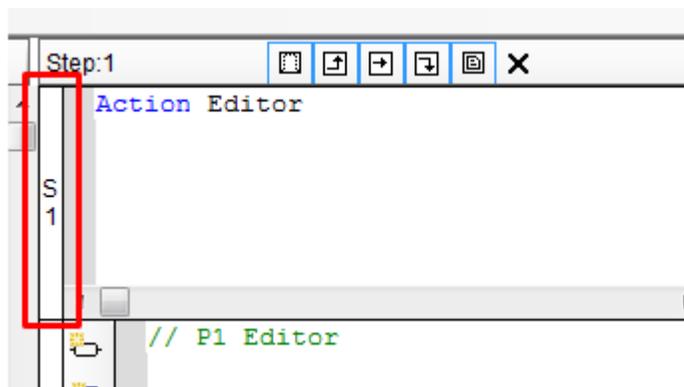
Clicking an open icon that is open will close the editor pane for that action.

Show All Qualifiers in Level 2

This option is used to change the view of the action editor for steps and transitions. This option is similar to the Show/Hide Split Window in Level 2 option, but the editor window can contain up to five splits so that all action types can be edited at the same time:



Using this option the SFC diagram is still displayed on the left and the editor(s) are displayed on the left. As the user clicks on the action icons in the editor pane the editor is split as needed. Open editor icons are highlighted, and a bar on the left edge indicates the action that is being edited.



When the icon for an open action is clicked on, the split for that action is closed.

Quick Select Menu Keyboard Short Cuts

In the SFC editor, many of the items in the quick select menu are bound to keyboard shortcuts. Pressing the shortcut key is equivalent to clicking the corresponding quick select menu option. Using the keyboard short cuts can make editing SFC charts much faster.

The headings of the Quick Select Menu Options sections below list the quick select key in brackets (e.g., < I > means 'I' is the quick select key to place an Initial Step ).

Quick select shortcut keys are not case sensitive.

The following chart lists all of the quick select key bindings for quick reference.

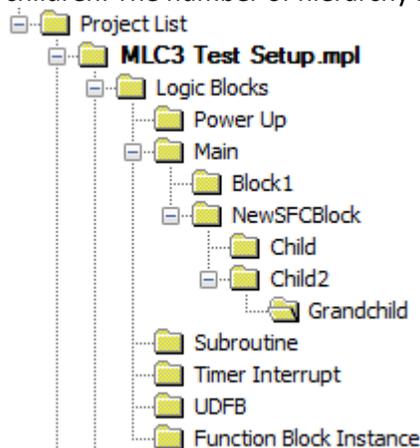
Key	Quick Select Option	Icon
I	Initial Step	
S	Step	
T	Transition	
J	Jump	
X	Divergence (X)	
D	Divergence (D)	
C	Divergence (C)	

In addition to these quick select keys, the following hints help to edit SFC diagrams quickly using the keyboard:

- The arrow keys can be used to select a particular location on the editor.
- Using the arrow keys with <shift> held down will select a series of editor blocks.
- Cut, Copy and paste function can be done using < CTL > + < X >, < CTL > + < C >, < CTL > + < V >, respectively.
- The delete key can be used to remove items.

Child SFC Programs

Each SFC program may have one or more "child programs". Child programs are written in SFC and are started (launched) or stopped (killed) in the actions of the parent program. A child program may also have children. The number of hierarchy levels should not exceed 19.

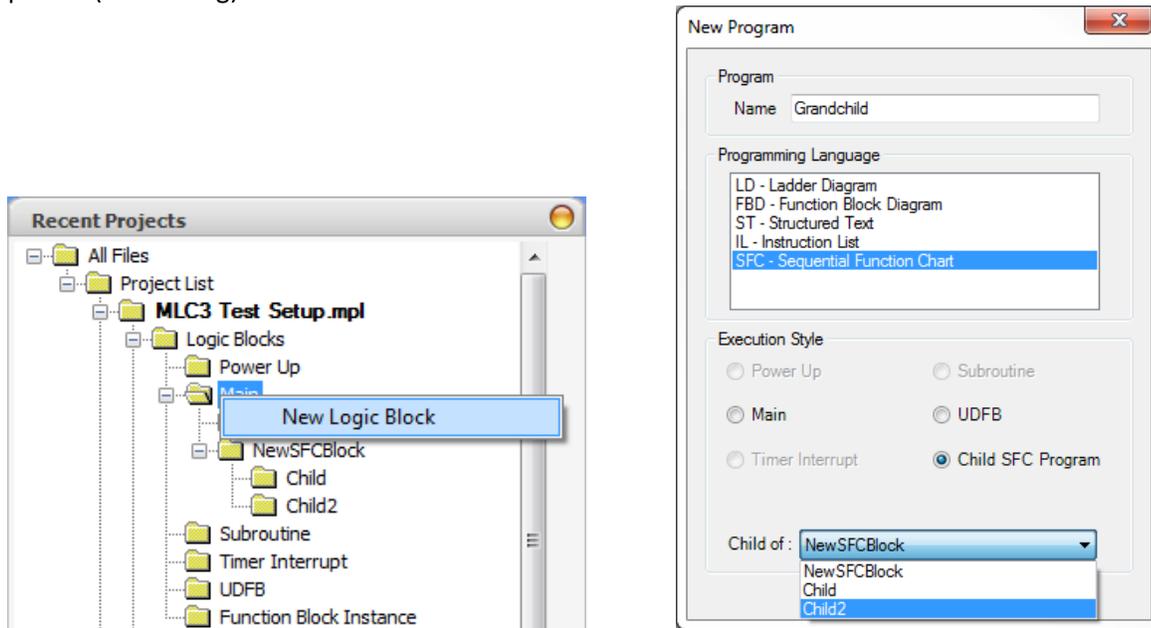


When a child program is stopped, its children are also implicitly stopped. When a child program is started, it must explicitly in its actions start its children.

A child program is controlled (started or stopped) from the action blocks of its parent program. Designing a child program is a simple way to program an action block in SFC language.

Using child programs is very useful for designing a complex process and separate operations due to different aspects of the process. For instance, it is common to manage the execution modes in a parent program and to handle details of the process operations in child programs.

To create a Child SFC Program, right-click the **Main** folder in the **Project Information Window** and select **New Logic Block**. Select **SFC - Sequential Function Chart** for the **Programming Language**, **Child SFC Program** for the **Execution Style**, and use the **Child of** dropdown to select an existing SFC logic block as the parent (controlling) block.



Controlling an SFC Child Program

Controlling a child program is achieved by specifying the name of the child program in a step in its parent program. Below are possible qualifiers that can be used in the Action block section for handling a child program. In the following table, *ChildProg* represents the name of the child program:

Qualifier	Description
ChildProg (N) ;	Starts the child program when the step is activated and stops (kills) it when the step is de-activated.
ChildProg (S) ;	Starts the child program when the step is activated. (The child program continues running until it is explicitly stopped elsewhere.)
ChildProg (R) ;	Stops (kills) the child program when the step is activated. (All active steps of the child program are deactivated)

Alternatively, you can use the following statements in a P1, N, or P0 action block programmed in ST language. In the following table, *prog* represents the name of the child program:

Statement	Description
GSTART (prog) ;	Starts the child program when the step is activated. (Initial steps of the child program are activated)
GKILL (prog) ;	Stops (kills) the child program when the step is activated. (All active steps of the child program are deactivated)
GFREEZE (prog) ;	Suspends the execution of a child program.
GRST (prog) ;	Restarts a program suspended by a GFREEZE command.

You can also use the "GSTATUS" function in expressions. This function returns the current state of a child SFC program:

Statement	Description
GSTATUS (prog);	Returns the current state of a child SFC program: 0: program is inactive 1: program is active 2: program is suspended

Note: When a child program is started by its parent program, it keeps the inactive status until it is executed (further in the scan cycle). If you start a child program in a SFC chart, GSTATUS will return 1 (active) on the next scan cycle.

SFC Best Practices

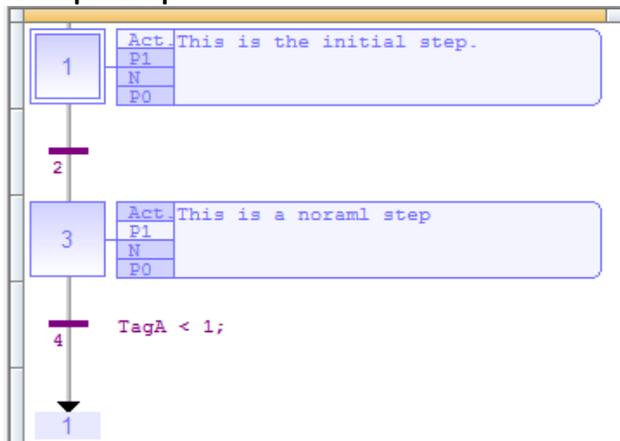
The general principle of an SFC is that the activated Step(s) follow a well-defined path through the chart. Given the state of the system, which governs the transitions, it should be clear what step(s) will be active at any given time. The rules below will help avoid compile errors and logical errors:

1. A step must connect to a transition and a transition must connect to a step.
2. A parallel branch should have a transition before the branch point, and have one transition for the entire branch.
3. A selection branch should have transitions after the branch point and should have one transition for each step attached to the branch.
4. Any branch should have a corresponding convergence of the same type.
5. Jump instructions should not be used to exit parallel branches.
6. Generally, keep SFC diagrams simple. Use function calls etc. to construct complex operations out of simple building blocks.

SFC Design Patterns

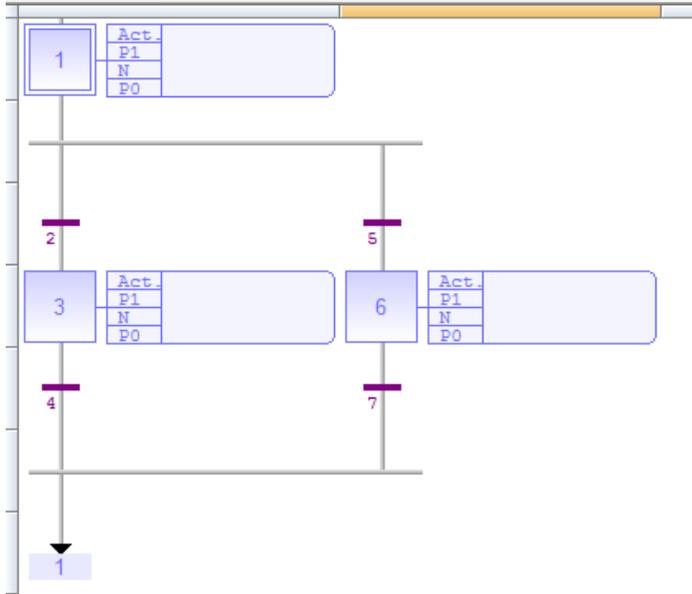
Using the rules from above there are some common design patterns that are useful in creating SFC logic.

- **A Simple Loop**



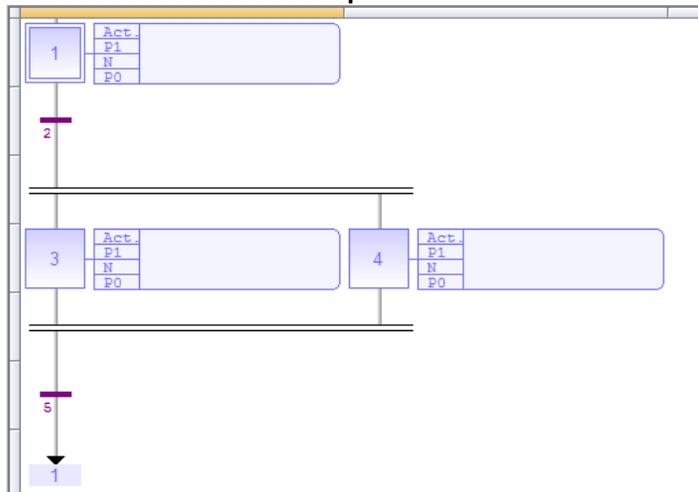
Loops are created using a series of steps and transitions, which end with a jump back to the step and the beginning of the loop.

- **Selection**



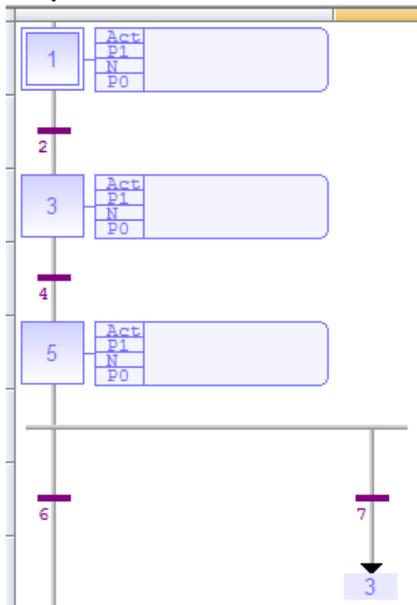
Selection allows the program to select between several divergent paths.

- **Parallel or Simultaneous Steps**



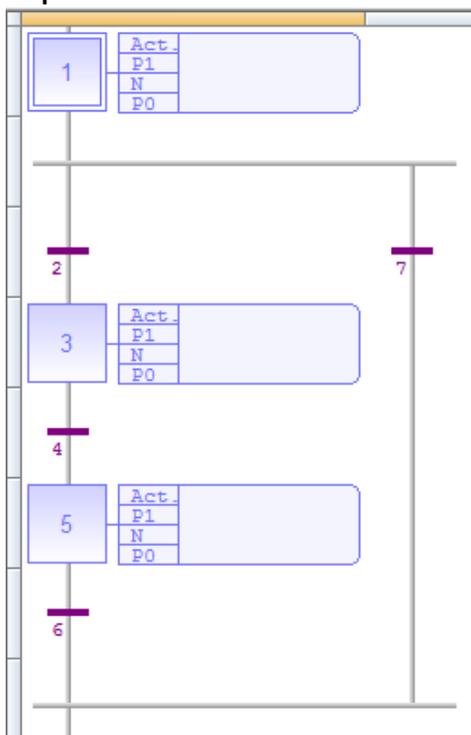
Allows multiple actions to take place at the same time.

- **Loop**



This pattern can be used to emulate a For or a While loop.

- **Skip**



This sequence is used to short circuit a series of steps if a particular condition is met. It functions similar to an IF statement.

By combining these building blocks, an arbitrarily complex SFC can be constructed.

Your Industrial Control Solutions Source

www.maplesystems.com



AW1010-1045 Rev 04