

# Chapter 14 OOP

## Introduction

What is OOP? OOP is short for Object Oriented Programming and implies an object is the focus of the program. This chapter discusses the use of objects in PLC programming and their use for making programs that are more readable. Software engineering in general has looked for programming tools that allow more robust and reliable computer programs.

Earlier programming discussions of “top-down” computer programming and “structured programming” have evolved into the use of objects that encapsulate an idea and stand alone in the program as that evolved concept. The overall idea of any programming endeavour has been to break the problem down into component parts and solve each part as it pertains to the whole and then combine the parts into a unified overall program. While the idea of a data structure is important, the focus of a top-down program or structured program would focus on the structure or flow rather than the task.

## OOP

The idea of pluggable entities or “lego” modules implies that modules can be created that can be plugged into one another and perform as a whole. This entails the idea of information that may be needed only inside the “lego” program and thus hidden from the outside world, in short, the idea of encapsulation. This leads to the idea of objects and object oriented programming or OOP. In OOP, the data is protected since it can be manipulated only inside the shell or protected program called the object. The logic is protected since it is only executed inside the object and the details of the program are known inside the object as to how the program and data interact.

OOP protects the data. It is easier to write in a general sense since the module can block out other programming considerations and the data can be stored inside the OOP as opposed to elsewhere. Data can, obviously, be stored anywhere the programmer wants but the protection of the OOP gives a security that wasn't available in earlier programming methods. And the programmer and the end user can focus on the object. This gives a great advantage to maintaining the program since the focus is placed where it was originally intended, on the object itself. And if some part changes, the programmer is reminded of the entity that is being affected and that changes should be studied as a whole for the object and not for just a part of the object. In the case of maintainability of control programs for the factory floor, the diagnostic part should be modified when the control portion is modified. If encapsulated, the programmer is reminded of both together since both should be encapsulated inside the object.

The programming language C refers to an object as a class. Other languages have similar names. In Siemens S7-Basic, the object is referred to as a Function (FC) or Function Block (FB). Allen-Bradley's RSLogix 5000 also has introduced a function with version 18 called the AOI or add-on instruction. While it will not be featured in this chapter, its use is similar to the function or function block described here.

To program an FC or FB, first identify the object or objects involved. Identify the messages or

signals the object needs to respond to and the outputs that result from these messages or signals. The FC or FB, while considered a class, may also be considered a template for the program. And the idea of FB's or FC's calling other FB's or FC's is a powerful concept and creates the idea of sub classes or sub-sub classes. The FC or FB can also be re-used again and again in the same or other programs for the same or different clients. This gives the idea of a class or OOP a huge advantage over conventional programming as we build programs over the years or from job to job.

Arguments for the benefits of reusability include:

- Reliability
- Efficiency of programming effort
- Saving time in program development
- Decreased maintenance effort
- Resulting cost savings
- Consistency of programs

Some additional terms of OOP include:

- Encapsulation – combining of programs and data to manipulate outcome in an object
- Inheritance – building a hierarch of objects, each with inheritance of the parent object
- Polymorphism – allowing one set of actions to share an object with another set of actions

FCs can be locked by the creator. This helps to preserve and protect the code and can actually help to simplify the overall program by firmly defining a functionality that is unchanged from one instance of code to another. STEP7 enables the user to create a storage location for custom functions called a Library. Several frequently used standardized and system functions are provided to the user in several libraries included with STEP7. The user can create a custom library and add items as needed, supporting programming standardization across projects.

The S7 architecture also supports the structuring of user-defined data storage locations, called data blocks, and reusable data templates called PLC Data Types.

The S7-1200 and 1500 controllers use programming elements that comply with IEC 61131-3 standard. At the core of the programming structure are code and data containers, known collectively as “Blocks”. The programmable logic controller provides various types of blocks in which the user program and the related data can be stored. Depending on the requirements of the process, the program can be structured in different blocks.

**Organization blocks:** (OB's) form the interface between the operating system and the user program. The entire program can be stored in OB1 that is cyclically called by the operating system (linear program) or the program can be divided and stored in several blocks (structured program).

**Function:** (FC's) contains a partial functionality of the program. It is possible to program functions so that they can be assigned parameters. As a result, functions are also suited for

programming recurring, complex partial functionalities such as calculations. System functions (SFC) are parameter-assignable functions integrated in the CPU's operating system. Both their number and their functionality are fixed. More information can be found in the Online Help.

**Function Block:** (FB's) offer the same possibilities as functions. In addition, function blocks have their own memory area in the form of instance data blocks. As a result, function blocks are suited for programming frequently recurring, complex functionalities such as closed-loop control tasks. System function blocks (SFB) are parameter-assignable functions integrated in the CPU's operating system. Both their number and their functionality are fixed.

**Data Blocks:** (DB's) are data areas of the user program in which user data is managed in a structured manner.

**Permissible Operations:** You can use the entire operation set in all blocks (FB, FC and OB). We will now start to explore these basic "program structuring elements" of S7 beginning with the FC, or Function. A Function is defined by the IEC 61131 standard as a code container that does not retain internal values from one scan to the next. Functions in the S7 PLC behave in this fashion, and act as a container for user developed program code. A function may have a set of local variables defined for use within the function. Typically, when "called" in the main program, a function will have new "values" (or actual parameters) loaded into the local variable (called formal parameters) for use during execution of the function. Once the "results" are calculated and function execution finishes, the resulting "output value(s)" get returned to the main program.

Before you can create the program for the parameter-assignable FC, you have to define the formal parameters in the declaration table. It is up to the programmer to select the declaration type for each formal parameter.

- The 'IN' declaration type should be assigned only to declaration types that will be read for instructions in the subroutine.
- Use the 'OUT' declaration type for parameters that will be written to within the function.
- Use the "IN\_OUT" for formal parameters that need to have a reading access and a writing access, such as a bit passed into the block that is used in the block for an edge operation.
- TEMP variables are intended to be used for holding interim calculation values or other values that are only required when the block is executing. TEMP variables exist in the local data stack while the block is executing and are overwritten when the block exits. The TEMP variables are - even though they are listed under "Interface" - not components of the block interface, since they do not become visible when the block is called and that no actual parameters have to be passed for the declared TEMP variables in the calling block.

The interface of a block forms the IN, OUT, and IN\_OUT parameters. The RETURN parameter is a defined, additional OUT parameter that has a specific name according to IEC 61131-3. This parameter only exists in FCs in the interface. The declared formal parameters of a block are its interface to the "outside" meaning they are "visible" or relevant to other blocks that call this block. If the interface of a block is changed by deleting or adding formal parameters later on, then the calls have to be updated.

When an FC is added to a project, the FC is accessible via the Project Browser. When the FC is to be executed must be determined. This is defined by which OB in which the FC is to be called. For example, if the FC is to be executed every scan, it is placed in OB1. To call an FC in OB1, drag and drop the FC from the project browser onto a network.

### Blocks Types

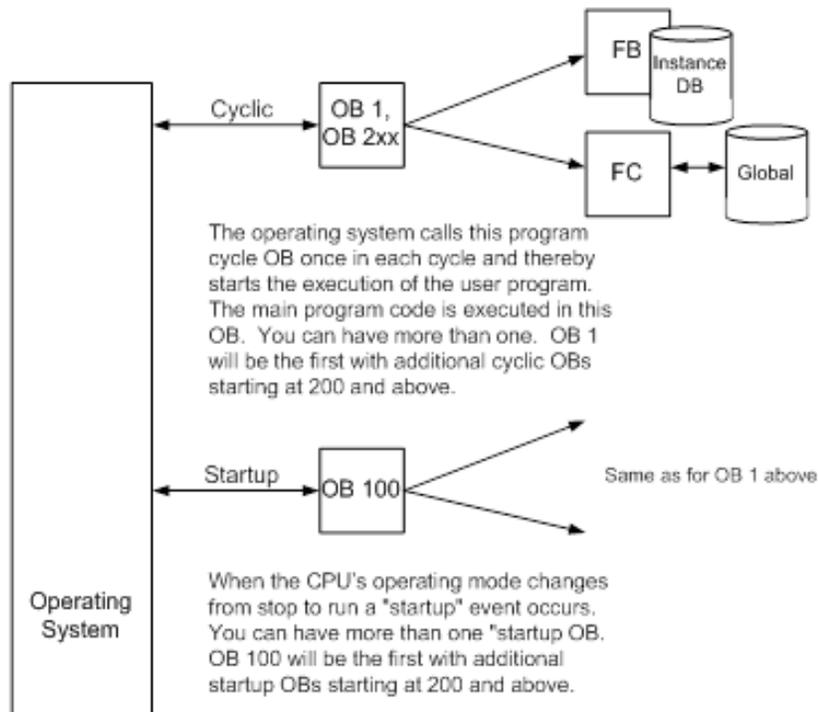
This is primarily a Siemens concept although Allen-Bradley has also introduced the idea with their function blocks in later versions of RSLogix 5000.

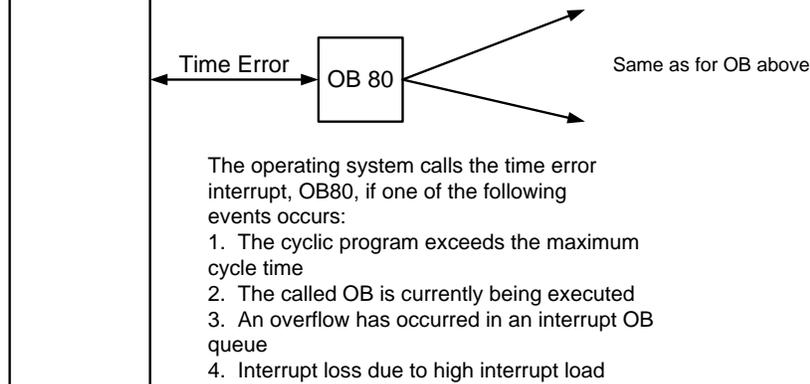
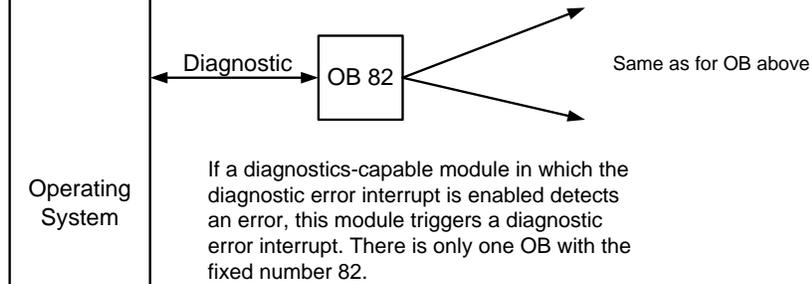
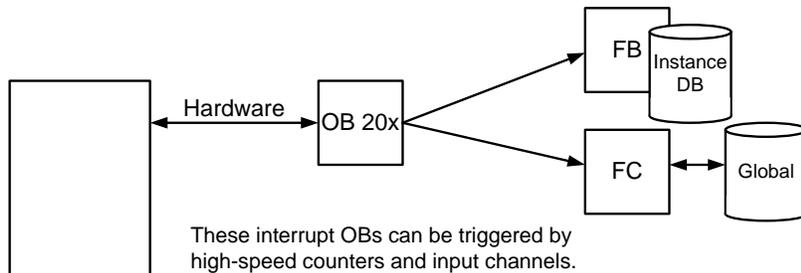
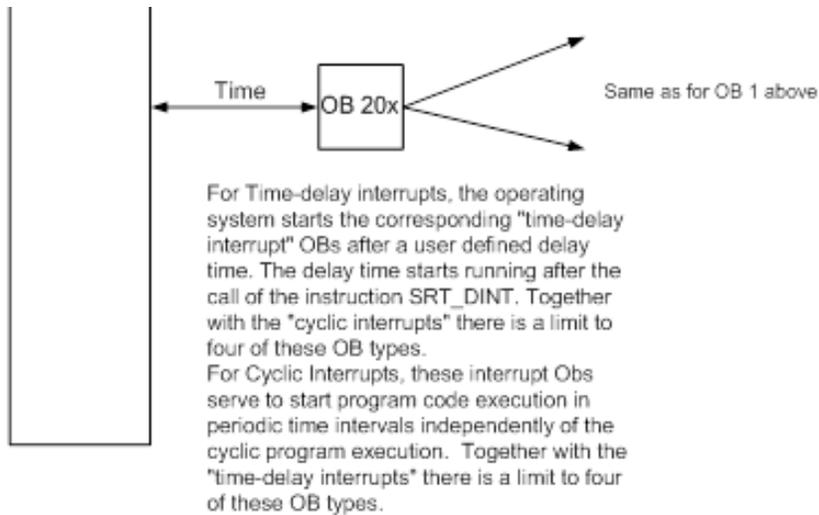
### Global DBs

A data block (DB) is a data area in the user program containing user data. Global data blocks store data that can be used by all other blocks. The structure of the global data blocks is user defined.

### Several Types of Blocks in STEP 7 Basic

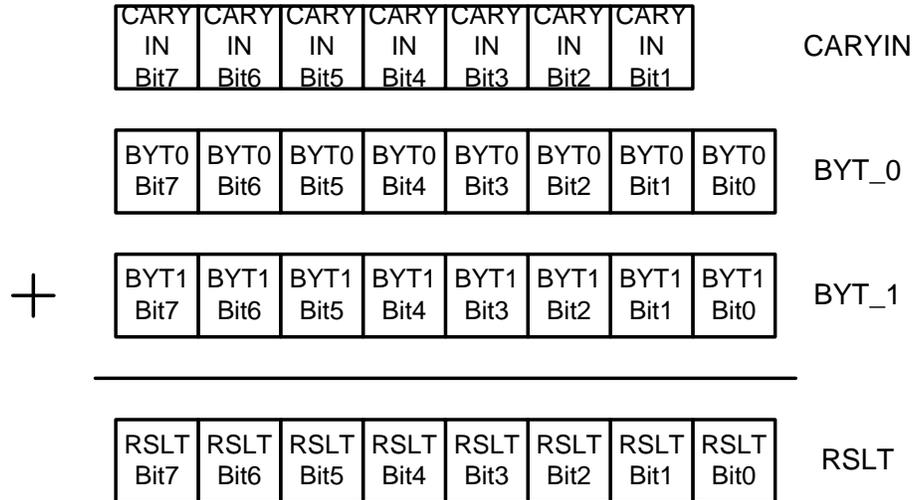
Interaction between the operating system and the various block types is pictured below:  
“



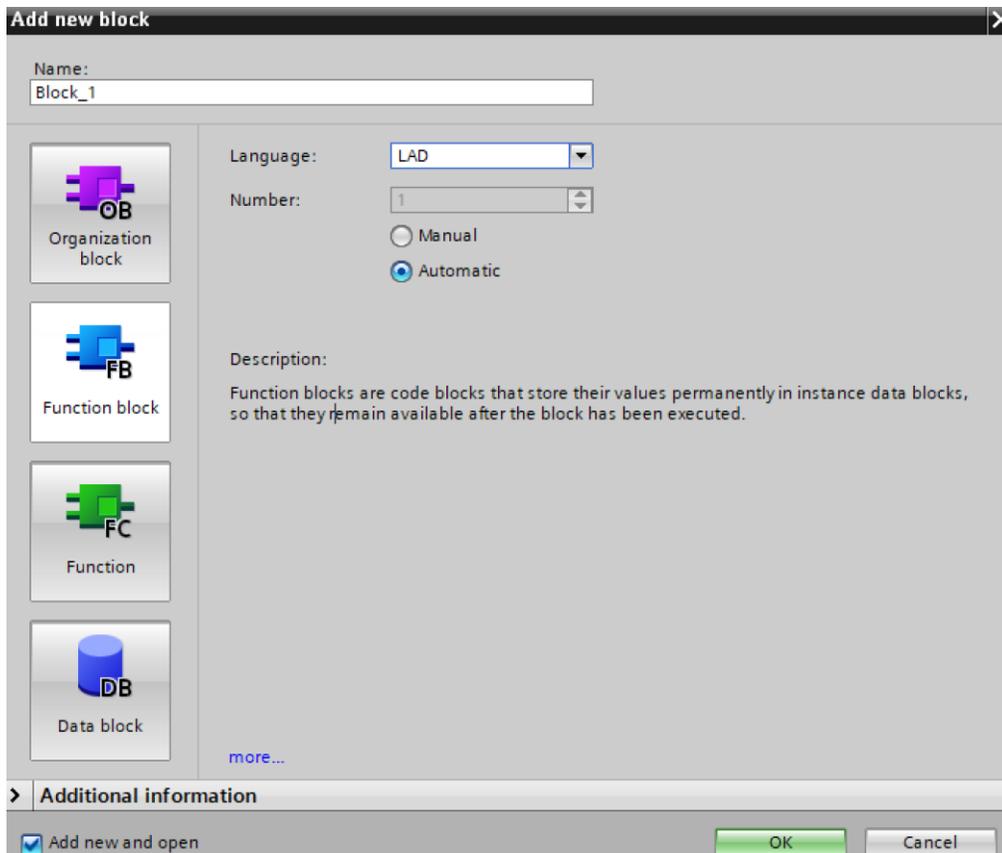


”

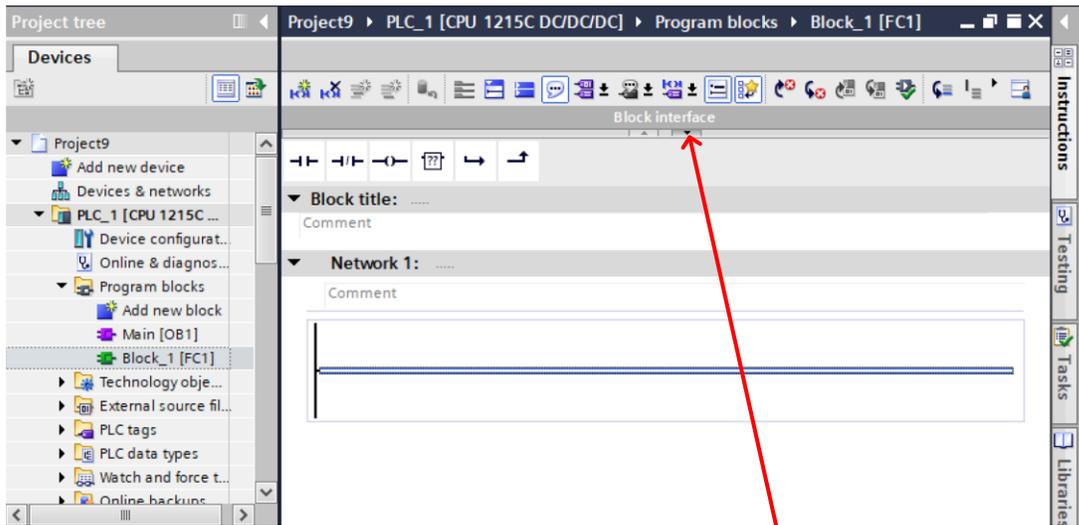
We will begin with a Function and start the project at the end of the chapter. Remember the directions from Ch. 8 – no instructions other than contacts and coils could be used to add two 16-bit numbers. The following show an 8-bit version of the same problem:



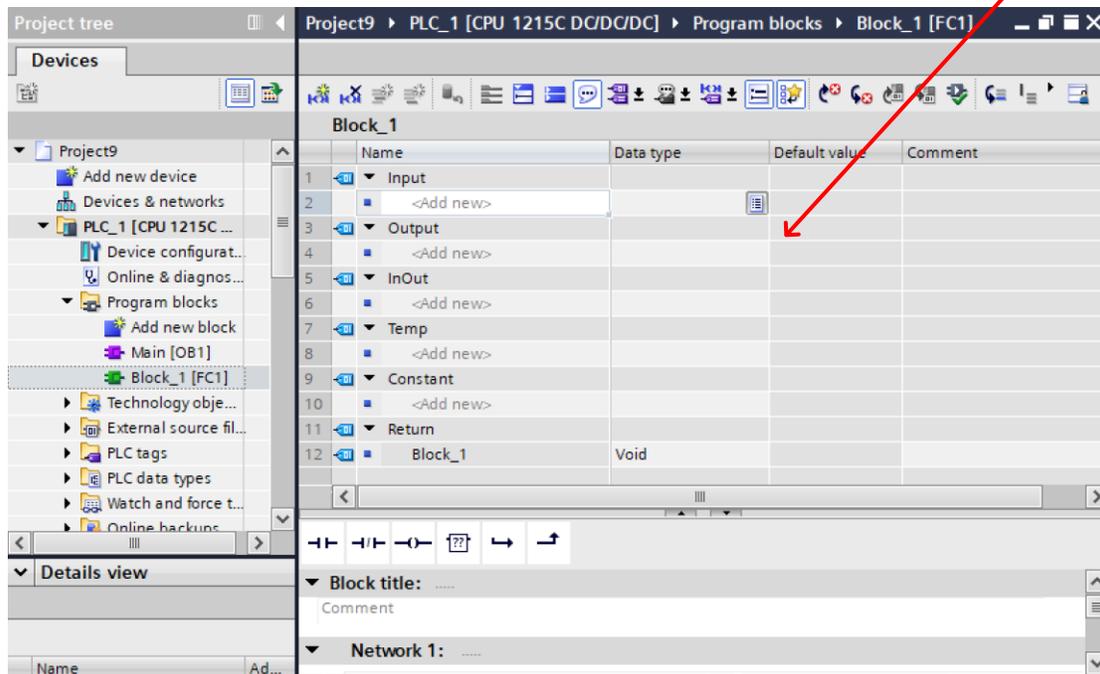
For the application, we need a FC rather than an FB since there is nothing to remember from function to function. Something comes in. Something goes out. Nothing is set in the function we need to remember. Choose Add new block, then Function and we will choose LAD (Ladder) because of the need for only contacts and coils:



Notice at left, we are now in the Function “Block\_1” and we have a new network to start.



Click here to view Variables



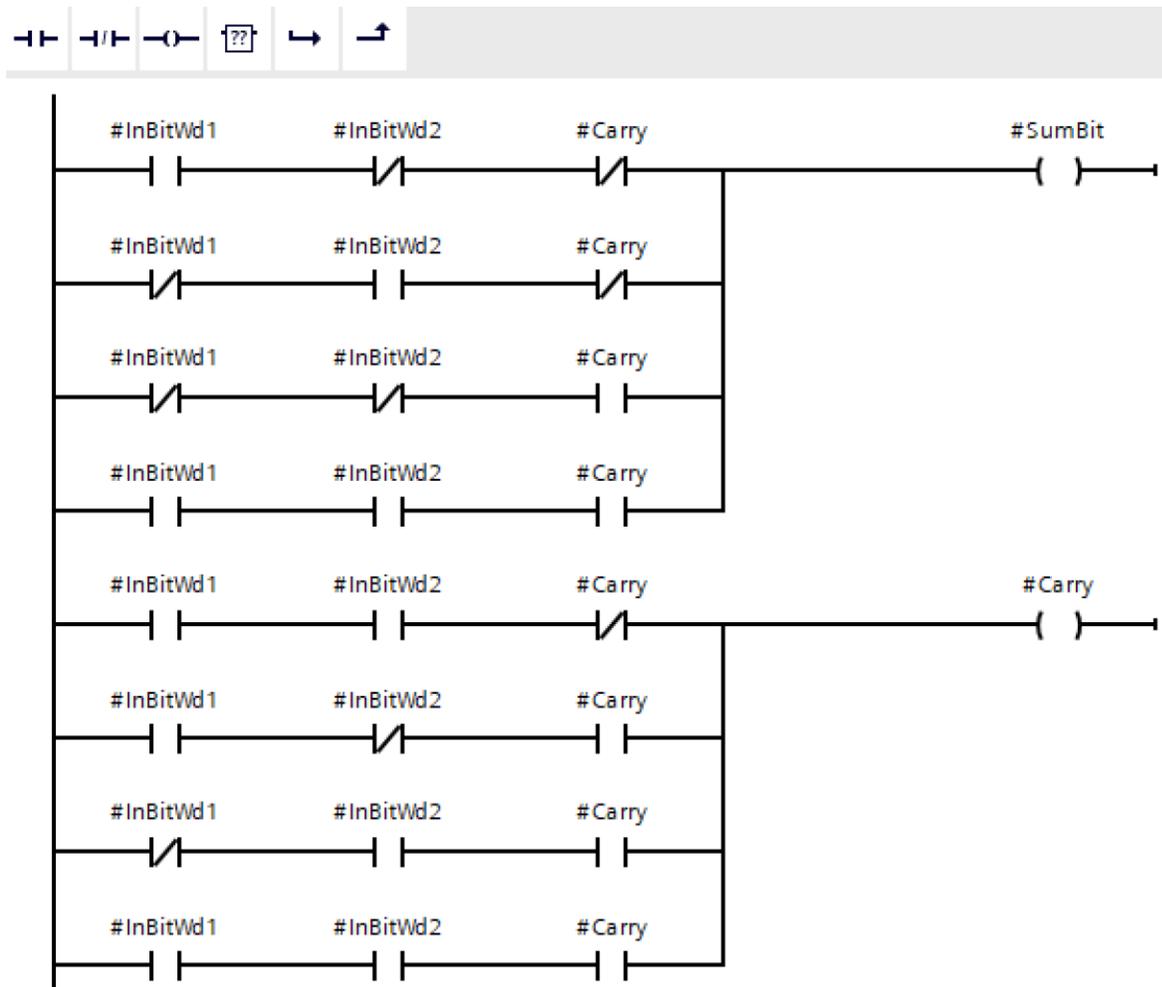
These variables are the ones inside the Function and are, in our case, Bool. We give pseudo-names to these variables and begin to program the program inside the Function:

Input – InBitWd1  
           InBitWd2  
 Output - SumBit  
 In/Out - Carry

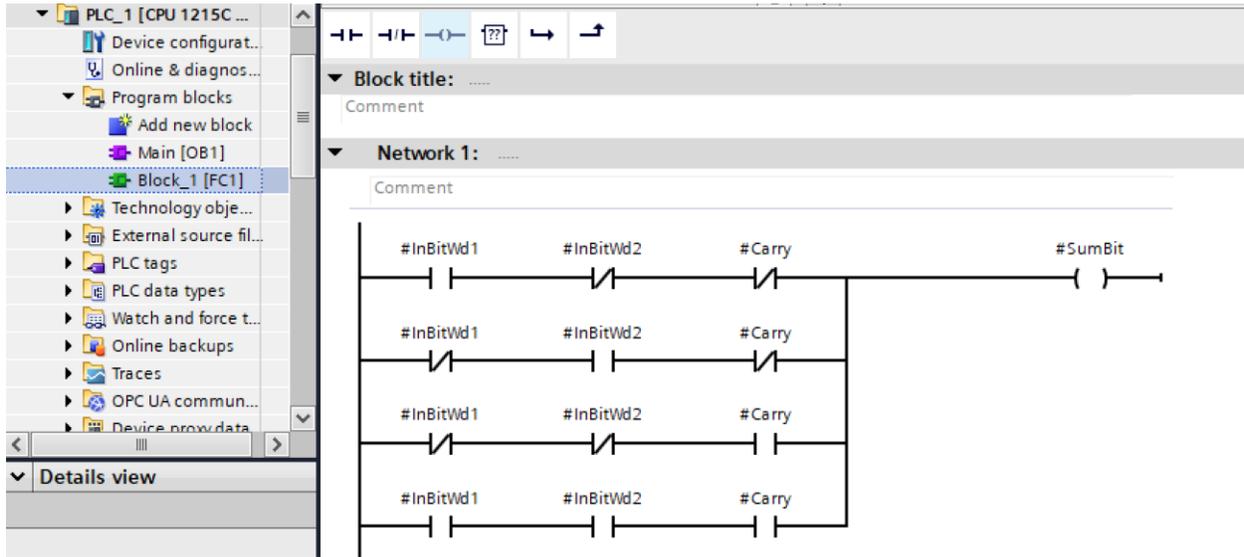
As can be seen later, the Input, Output and InOut variables are the ones that are visible inside the Function when used in OB1.

Block_1				
	Name	Data type	Default value	Comment
1	▼ Input			
2	■ InBitWd1	Bool		
3	■ InBitWd2	Bool		
4	■ <Add new>			
5	▼ Output			
6	■ SumBit	Bool		
7	■ <Add new>			
8	▼ InOut			
9	■ Carry	Bool		
10	■ <Add new>			
11	▼ Temp			
12	■ <Add new>			

Using these pseudo-variables, we enter the program inside the Function:



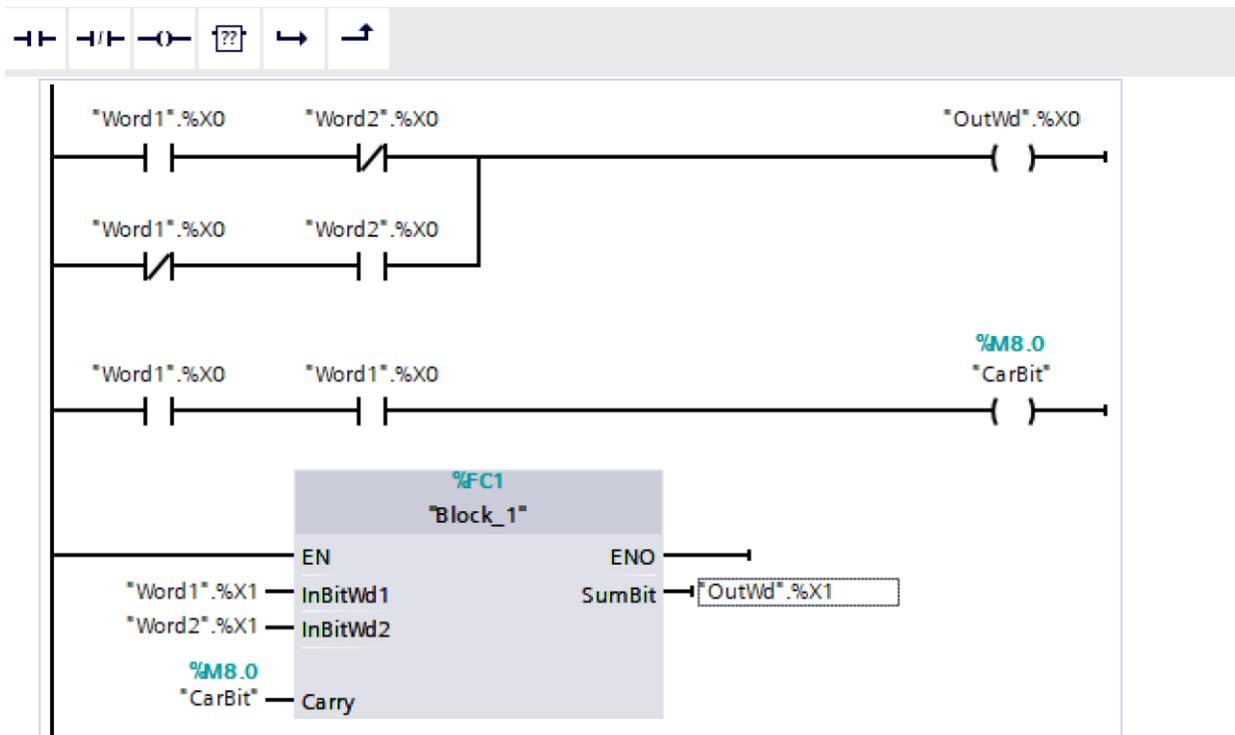
After building the logic, right click on Block\_1 in the project tree and compile the function:



Now, we are ready to incorporate the FC in the main program OB. We first add the words we want to add together:

PLC tags	Name	Tag table	Data type	Address	Retain	Acces...	Writa...	Visibl...	C...
1	Word1	Default tag table	Int	%MW2	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
2	Word2	Default tag table	Int	%MW4	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
3	OutWd	Default tag table	Int	%MW6	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
4	CarBit	Default tag table	Bool	%M8.0	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
5	<Add new>				<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

After the words to be added are defined, we start programming in OB1. First, we add the logic for bit 0. This is a half-adder and we need only build this logic once. Then we drag the block Block\_1 from the left and add it to the logic. We finish the process by adding the word.bit addresses for bit 1 to this function. We now are ready to add the bits 2-15 with function block for each and we are complete. This is left as an exercise.



**A review of the Variables inside the FC and FBs defined by Siemens:**

Before you can create the program for the parameter-assignable block, you have to define the block parameters in the Interface table. The block interface allows local tags to be created and managed.

The tags are subdivided into two groups shown by the table below:

Block parameters that form the block interface when it is called in the program			
Type	Section	Function	Available in
Input parameters	Input	Parameters whose values are read by the block	Functions, function blocks and some types of organization blocks
Output parameters	Output	Parameters whose values are written by the block	Functions and function blocks
InOut Parameters	InOut	Parameters whose values are read by the block when it is called, and whose values are written again by the block after execution	Functions and function blocks
Local data that are used for storage of intermediate results			
Type	Section	Function	Available in

Temporary local data	Temp	Tags that are used to store temporary intermediate results. Temporary local data are retained for only one cycle	Functions, function blocks and organization blocks
Static local data	Static	Tags that are used for storage of static intermediate results in the instance data block. Static data is retained until overwritten, which may be after several cycles	Function blocks only

**A review of when Temporary Tags are active:**

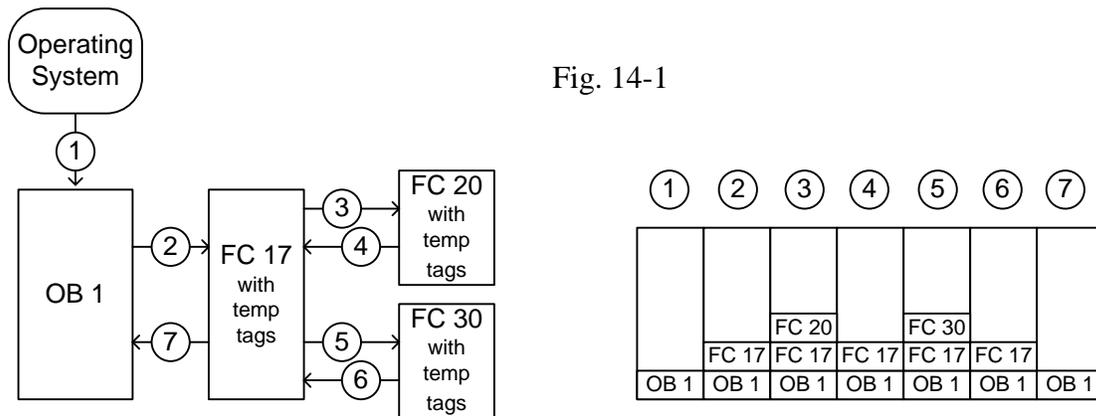


Fig. 14-1

**Creating an FB Block**

“FB – Function block Code blocks that store their values permanently in instance data blocks, so that they remain available even after the block has been executed.

All In-, Out-, InOut- parameters are stored in the instance DB – the instance DB is the memory of the FB.

**Definition**

Function blocks are code blocks that store their values permanently in instance data blocks, so that they remain available even after the block has been executed. They save their input, output and in/out parameters permanently in the instance data blocks. Consequently, the parameters are still available after the block execution. Therefore they are also referred to as blocks "with memory".

**Block Interface**

The block interface for an FB looks similar to that of an FC. There are two groups of Block interface tags:

1. Block parameters that form the block interface when it is called in the program.
  - Input, Output, and In/Out parameters are a part of this group
2. Local data that are used for storage of intermediate results
  - Temporary local data and Static local data are part of this group

**Static Local Data**      An instance DB is used to save static local data. These static local data can only be used in the FB, in whose block interface table they are declared. When the block is exited, they are retained.

**Parameters**            When the function block is called, the values of the actual parameters are stored in the instance data block.

If no actual parameter is assigned to an interface parameter in a block call, then the last value stored in the instance DB for this parameter is used in the program execution.

You can specify different actual parameters with every FB call. When the function block is exited, the data in the data block is retained. To keep the data unique for each instance of a call it is required to assign a different instance DB each time a call instruction to an FB is written in code.

You can program parameter-assignable blocks for frequently recurring program code. This has the following advantages:

1. The program only has to be created once, which significantly reduces programming time.
2. The block is only stored in the user memory once, which significantly reduces the amount of memory used.
3. The FB can be called as often as you like, each time with a different address assignment. For this, the interface parameters (input, output, or in/out parameters) are supplied with different actual operands every time called.

### **Multi-instance data block**

**Definition**            Multi-instances enable a called function block to store its data in the instance data block of the calling function block. This allows you to concentrate the instance data in one instance data block and thus make better use of the number of instance data blocks available.”

So, where do most programmers write the majority of their code when using Siemens? That is a good question but one that should consider the use of FB's as the main area for large control programs. Why? We have the ability to use static global variables in this area and also the ability to have all the variable types including arrays present. In the classroom environment, it is not necessary to consider this because programs here probably are rather small. When they grow, however, use the FB as a primary area for large programming efforts.

Later, in the Lab Text, we will see what the Festo Programs give in the way of FB's. They are very large and complex.

## **Summary**

This chapter introduces the student to the important concept of functions and function blocks from Siemens. It is not a complete study of this subject. In the appendix below is found a more thorough discussion as well as a discussion of optimization techniques using functions and function blocks. Data types are discussed as well here, a subject that has been discussed in earlier chapters but not to the extent as found in this appendix. The movement toward functions and function blocks from Siemens shows a steady move to convince the programming engineer that their organizational advantages are to be considered.

The appendix also explains more thoroughly global versus local data. Also, the subject of retentive data and how it is saved is shown. An example of newer programming techniques such as interspersing SCL in a network in a LAD program is shown as well. This provides very powerful design in program building.

Personally, I have not had many opportunities to use functions and function blocks in real-world applications. Most programs written by myself have been unique to the point that the organization of functions and function blocks gave little or no advantage. Even so, they should be understood because of their organizational advantages and time saved in constructing programs that may be used more than once.



## Accessing Bits in Words (Siemens)

### Examples

In the PLC tag table, "DW" is a declared tag of type DWord. The examples show bit, byte, and word slice access:

	LAD	FBD	SCL
<b>Bit access</b>			<pre>IF "DW".x11 THEN ... END_IF;</pre>
<b>Byte access</b>			<pre>IF "DW".b2 = "DW".b3 THEN ... END_IF;</pre>
<b>Word access</b>			<pre>out := "DW".w0 AND "DW".w1;</pre>

## Accessing a tag with an AT overlay

The AT tag overlay allows you to access an already-declared tag of a standard access block with an overlaid declaration of a different data type. You can, for example, address the individual bits of a tag of a Byte, Word, or DWord data type with an Array of Bool. To overlay a parameter, declare an additional parameter directly after the parameter that is to be overlaid and select the data type "AT". The editor creates the overlay, and you can then choose the data type, struct, or array that you wish to use for the overlay.

### Example

This example shows the input parameters of a standard-access FB. The byte tag B1 is overlaid with an array of Booleans:

B1	Byte
AT	AT "B1" Array [0..7] of Bool
AT[0]	Bool
AT[1]	Bool
AT[2]	Bool
AT[3]	Bool
AT[4]	Bool
AT[5]	Bool
AT[6]	Bool
AT[7]	Bool

Table 4-6 Overlay of a byte with a Boolean array

7	6	5	4	3	2	1	0
AT[0]	AT[1]	AT[2]	AT[3]	AT[4]	AT[5]	AT[6]	AT[7]

Another example is a DWord tag overlaid with a Struct:

DW1	DWord
DW1_Struct	AT "DW1" Struct
S1	Word
S2	Byte
S3	Byte

The overlay types can be addressed directly in the program logic:

LAD	FBD	SCL
		<pre>IF #AT [1] THEN ... END_IF ;</pre>
		<pre>IF (#DW1_Struct.S1 = W#16#000C) THEN ... END_IF ;</pre>
		<pre>out1 := #DW1_Struct.S2 ;</pre>

### Rules

- Overlaying of tags is only possible in FB and FC blocks with standard access.
- You can overlay parameters for all block types and all declaration sections.
- An overlaid parameter can be used like any other block parameter.
- You cannot overlay parameters of type VARIANT
- The size of the overlaying parameter must be less than or equal to the size of the overlaid parameter.
- The overlaying variable must be declared immediately after the variable that it overlays and identified with the keyword “AT”.

**Complete the lab using a function instead of coding each network as separate logic.**

### Binary Subtraction:

To perform binary subtraction, the easiest method is to find the 2’s complement of the second number and then add the two numbers together.

The best method of finding the 2’s complement requires the use of a memory bit. The rule requires that bits from the original number be copied to the 2’s complement number starting at the right-most bit. The rule applies until a “1” is encountered. The first “1” is copied but a memory bit is set after which the bits are “flipped”. Try this rule. It works and may be employed using ladder logic and a Latch bit to quickly find the 2’s complement of a number. The logic for finding the 2’s complement of a number in ladder logic is begun in Fig. 8-37. Again, logic must be added to complete the function using rungs similar to rungs 4 and 5 of this figure but using bits 2 through 15.

Again, code the logic using a function.

## Lab 14.2

Implement the following:

### Linking PLC UDT Tags to HMI Faceplates and Pop-ups

<https://www.dmcinfo.com/latest-thinking/blog/id/9136/linking-plc-udt-tags-to-hmi-faceplates-and-pop-ups-in-tia-portal-v13-sp1>

## Lab 14.3      Repeat Lab 13.2C1 using an FB and UDT – Whack-a-Mole

**Lab 13.2C**      Add a table of results including whether the player hit the light while the light was on and how long the response was delayed from when the light first turned on. Results for each hit are to be saved sequentially in the table.

**Lab 13.2C1**      Implement 13.2C above with a UDT output table. Save sequential hit data for later display or analysis.

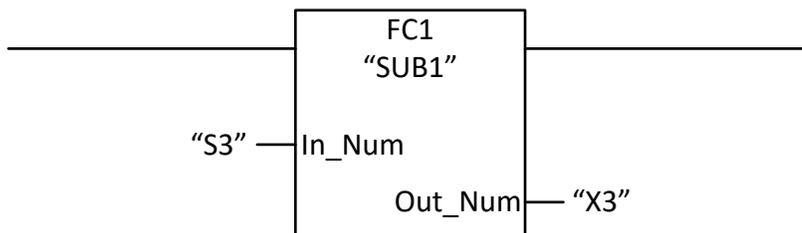
## Problems

- Three types of parameters for interface of a function are:
  - a
  - b
  - c
- Local data is of two types. They are:
  - a
  - b
- Data blocks are either Single \_\_\_\_ or multi \_\_\_\_\_. What is the deciding factor which to use?
- List some program blocks that are standard.
- Describe a function or function block that would have the title “Engine” and have two types of engines that could be called – Diesel or Gasoline.
- What is the A-B process for adding a function?

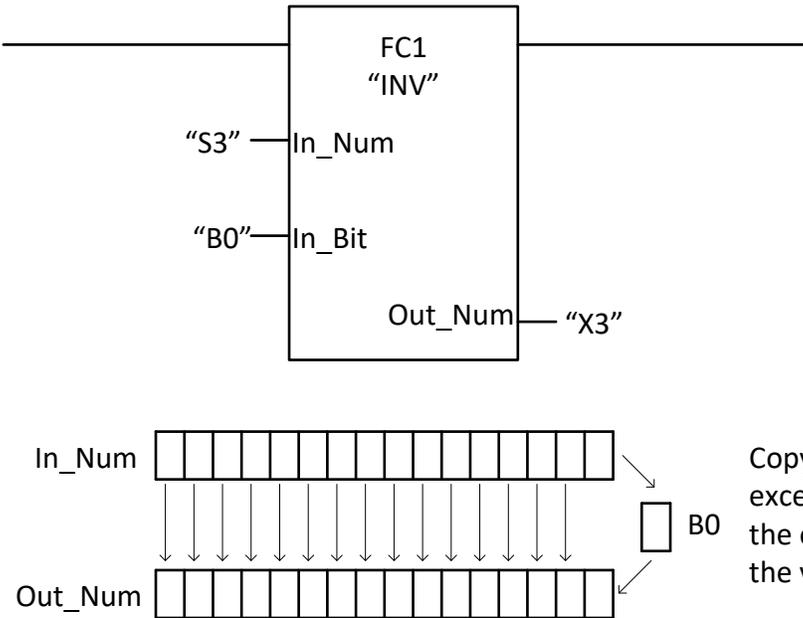
For the next three programs, only use skeletal statements but enough to get the idea:

- In the Kitchen:

In the kitchen are several needs for automation including cooking breakfast. In the breakfast shelf are several kinds of cereal including oatmeal, cream-of-wheat and grits. Each requires the microwave and a cooking time. Each requires an amount to be weighed on a scale. When the weight is achieved, the bowl is placed in the microwave for a time period. Write a program using FC’s or FB’s to achieve cooking of the breakfast cereal.
- Three numbers are to be added and the result displayed. Use a FC or FB to accomplish this.
- In OB1, there is a FC1 accessed that **subtracts 1** from a number. Build the function block SUB1 to complete the operation. Show all **tables** and **logic**:

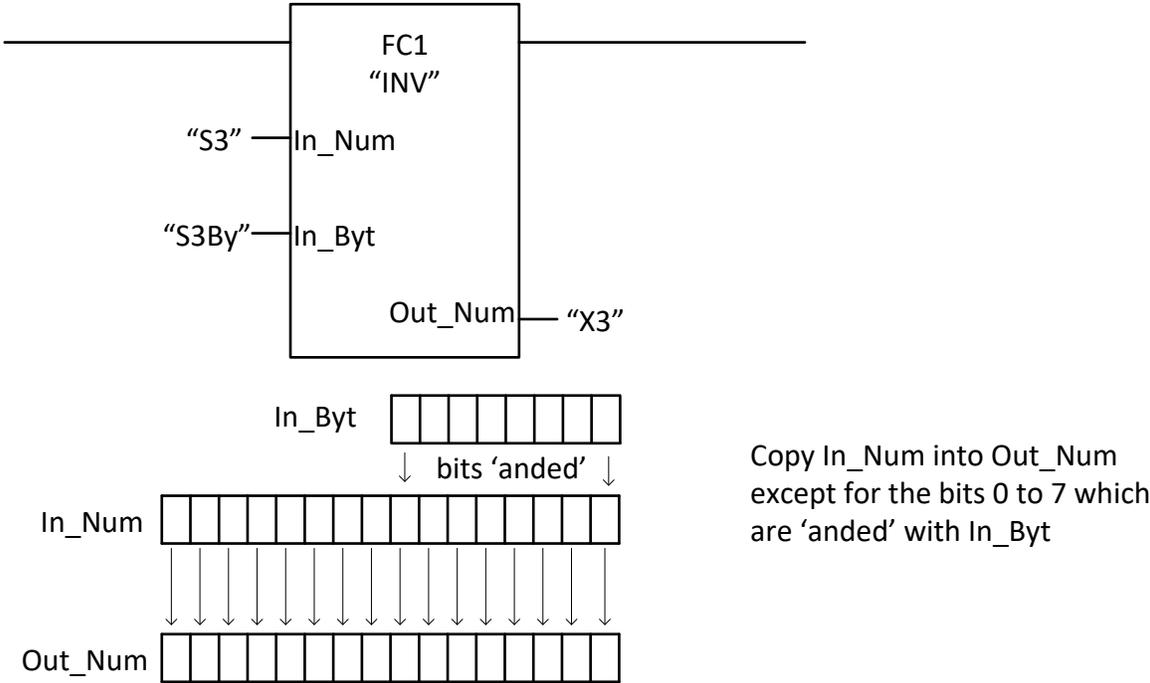


10. In OB1, there is a FC1 accessed that does the following:



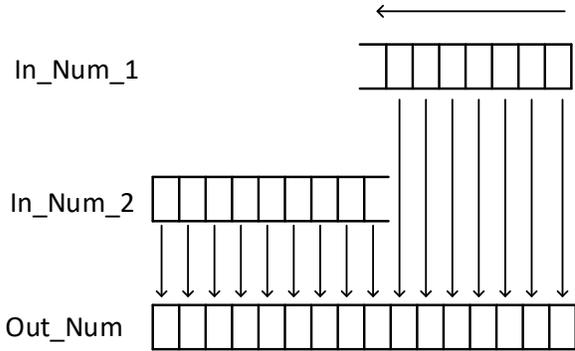
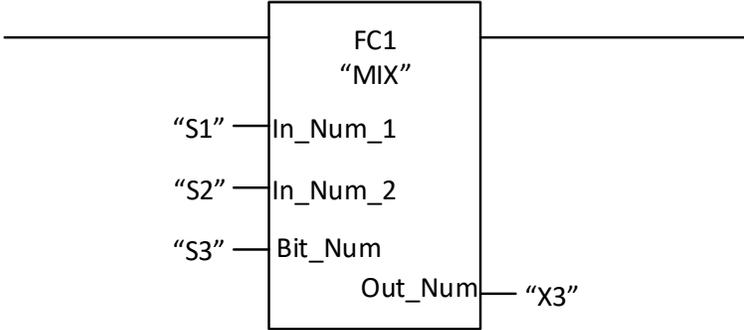
Build the function block "INV" to complete the operation. Show all tables and logic:

11. In Siemens' OB1, there is a FC1 accessed that does the following:



Build the function "INV" to complete the operation. Show all tables and logic inside the Function (FC):

12. In Siemens' OB1, there is a FC1 accessed that does the following:



Bit\_Num determines how many bits are moved from In\_Num\_1. Remaining bits are moved from In\_Num\_2. For example, if Bit\_Num=7, the number of bits moved from In\_Num\_1 is 7 as shown at left.

Build the function "MIX" to complete the operation. Show all tables and logic inside the Function (FC1):

## Appendix 1

<http://www.youtube.com/watch?v=aUllkF4al30&feature=relmfu>

Siemens SIMATIC S7-1200 Part 2 - Re-Usable Libraries

See how easy it is to implement reusable Libraries in Step 7 Basic Software eliminating time consuming coding of repeat functions. This is part two of a four part series showcasing the time and cost saving benefits of the new S7-1200 and its Step 7 Basic development software. For more information see:

<http://www.usa.siemens.com/s7-1200>

<http://www.youtube.com/watch?v=L2NLcAQhiSg&feature=relmfu>

Siemens SIMATIC S7-1200 Part 4 - Project-wide Cross Referencing Made Easy

See how easy it is to troubleshoot the complete Controller and HMI software project together for both SIMATIC Basic HMI panels and S7-1200 Controllers. This is part four of a four part series showcasing the time and cost saving benefits of the new S7-1200 and its Step 7 Basic development software. For more information see:

<http://www.usa.siemens.com/s7-1200>

## Appendix 2

The following pages are from the Siemens Text:

**Programming Guideline for S7-1200/1500 Entry ID: 81318674, V1.6, 12/2018**

They summarize changes and upgrades to the Siemens Portal Language from Version 14 – TIA and later. Covered below is a review of the discussions above concerning FB's and FC's as well as general organization using them. Advantages of this type of programming are discussed as well. Speed of execution is an important part of this discussion.

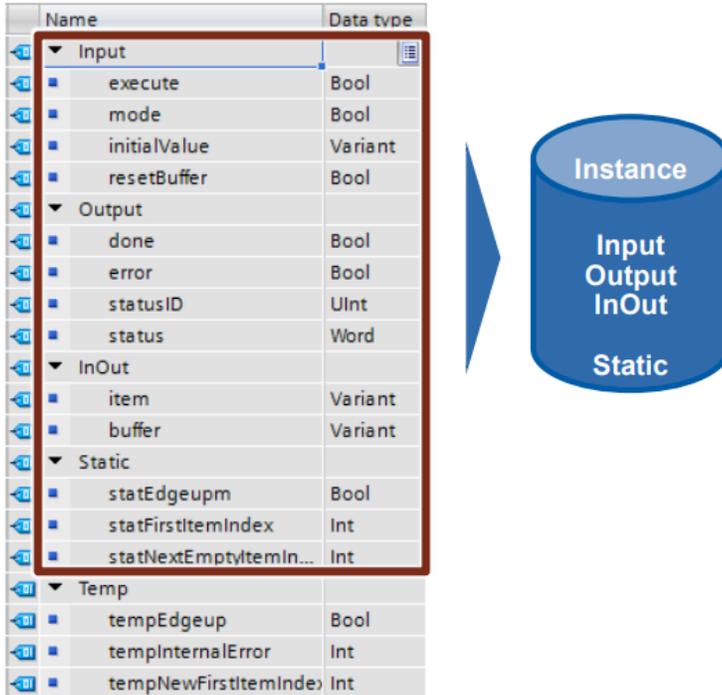
“

### 3.2.4 Instances

The call of a function block is called instance. The data with which the instance is working is saved in an instance DB.

Instance DBs are always created according to the specifications in the FB interface and can therefore not be changed in the instance DB.

Figure 3-8: Structure of the interfaces of an FB



The instance DB consists of a permanent memory with the interfaces input, output, InOut and static. Temporary tags are stored in a volatile memory (L stack). The L stack is always only valid for the current processing. I.e. temporary tags have to be initialized in each cycle.

#### Properties

- Instance DBs are always assigned to a FB.
- Instance DBs do not have to be created manually in the TIA Portal and are created automatically when calling an FB.
- The structure of the instance DB is specified in the appropriate FB and can only be changed there.

#### Recommendation

- Program it in a way so that the data of the instance DB can only be changed by the appropriate FB. This is how you can guarantee that the block can be used universally in all kinds of projects.

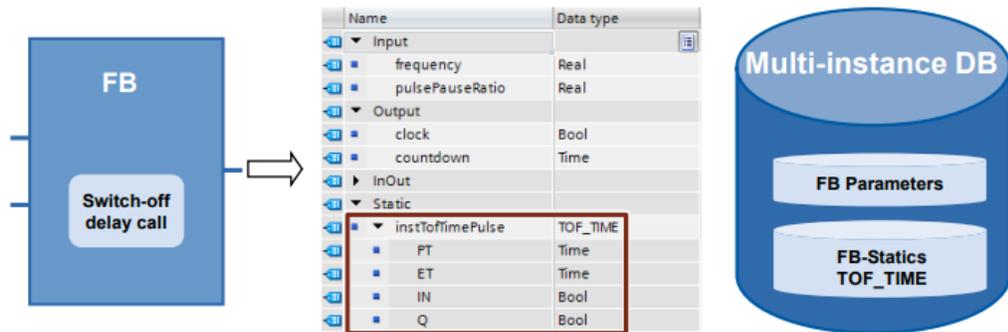
For more information, please refer to chapter [3.4.1 Block interfaces as data exchange](#).

### 3.2.5 Multi-instances

With multi-instances called function blocks can store their data in the instance data block of the called function block. This means, if another function block is called in a function block, it saves its data in the instance DB of the higher-level FBs. The functionality of the called block is thus maintained even when the calling block is transferred.

The following figure shows an FB that uses another FB ("IEC Timer"). All data is saved in a multi instance DB. It is thus possible to create a block with an independent time behavior, for example, a clock generator.

Figure 3-9: Multi-instances



#### Advantages

- Reusability
- Multiple calls are possible
- Clearer program with fewer instance DBs
- Simple copying of programs
- Good options for structuring during programming

#### Properties

- Multi-instances are memory areas within instance DBs.

#### Recommendation

Use multi-instances in order to ...

- reduce the number of instance DBs.
- create reusable and clear user programs.
- program local functions, for example, timer, counter, edge evaluation.

#### Example

If you require the time and counter function, use the "IEC Timer" blocks and the "IEC Counter" blocks instead of the absolutely addressed SIMATIC Timer. If possible, also always use multi-instances here. Thus, the number of blocks in the user program is kept low.

### 3.2.6 Transferring instance as parameters (V14)

Instances of called blocks can be defined as InOut parameters.

#### Advantages

- It is possible to create standardized functions whose dynamic instances are transferred.
- Only when calling the block it is specified what instance is used.

#### Example

Figure 3-11: Transferring instance as parameter

The image shows two parts of the software interface. The top part is the 'Call options' dialog box, and the bottom part is the 'FunctionBlock' editor.

**Call options dialog box:**

- Parameter instance:** This option is selected and highlighted with a red box. It shows a database icon with two arrows pointing to it.
- Name in the interface:** The text 'instIECCounter1' is entered in the input field.
- Text:** 'If you call the function block as a parameter instance, the function block saves its data in the instance you specify as block parameter and not in the instance of the called block. This gives you the option of defining the instance for this FB call during runtime.'
- Buttons:** 'OK' and 'Cancel' buttons are at the bottom right.

**FunctionBlock editor:**

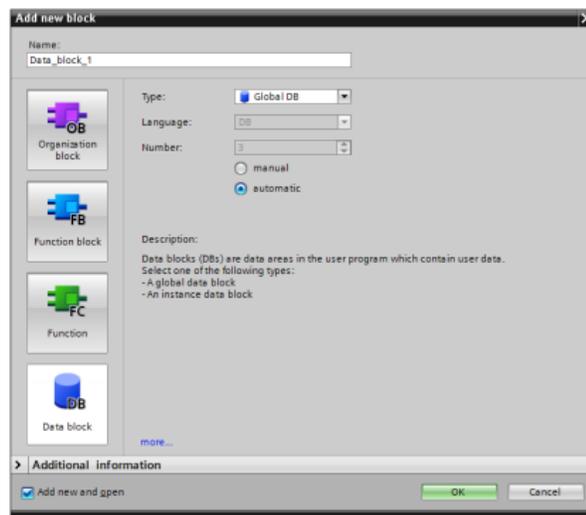
- Table:** A table with columns 'Name' and 'Data type'.

Name	Data type
1 Input	
2 Output	
3 InOut	
4 instIECCounter	CTU_INT
6 Static	
7 statCount	Bool
8 statValue	Int
9 <Add new>	
10 Temp	
- Code:** Below the table, there is a code editor showing the declaration of the parameter instance:

```
#instIECCounter (CU:=#statCount,  
                PV:=#statValue);
```
- Block Call:** At the bottom, a block call for 'FunctionBlock' is shown. The 'counterParam' parameter is set to 'instIECCounter' from a dropdown menu, which is highlighted with a red box.

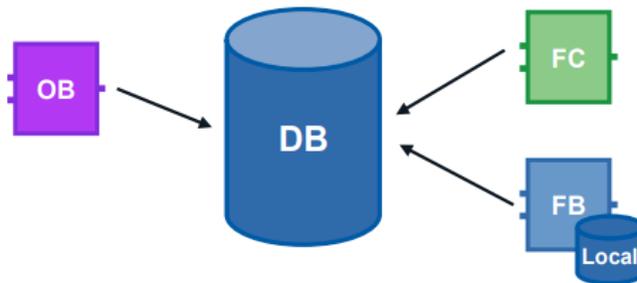
### 3.2.7 Global data blocks (DB)

Figure 3-12: "Add new block" dialog (DB)



Variable data is located in data blocks that are available to the entire user program.

Figure 3-13: Global DB as central data memory



#### Advantages

- Well-structured memory area
- High access speed

#### Properties

- All blocks in the user program can access global DBs.
- The structure of the global DBs can be arbitrarily made up of all data types.
- Global DBs are either created via the program editor or according to a previously created "user-defined PLC data type" (see chapter [3.6.4 STRUCT data type and PLC data types](#)).
- A maximum of 256 structured tags (ARRAY, STRUCT) can be defined. This does not apply to tags that are derived from a PLC-data type.

#### Recommendation

- Use the global DBs when data is used in different program parts or blocks.

**Note**

More information can be found in the following entry:

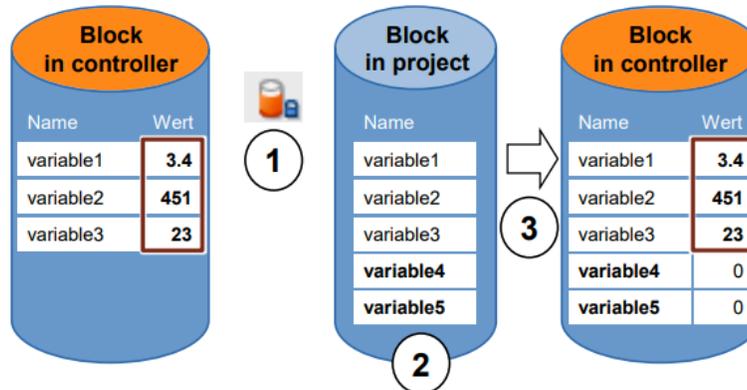
How is the declaration table for global data blocks structured in STEP 7 (TIA Portal)?

<https://support.industry.siemens.com/cs/ww/en/view/68015630>

### 3.2.8 Downloading without reinitialisation

In order to change user programs that already run in a controller, S7-1200 (firmware V4.0) and S7-1500 controllers offer the option to expand the interfaces of optimized function or data blocks during operation. You can load the changed blocks without setting the controller to STOP and without influencing the actual values of already loaded tags.

Figure 3-14: Load without reinitialization



Execute the following steps whilst the controller is in RUN mode.

1. Enable "Downloading without reinitialisation"
2. Insert newly defined tags in existing block
3. Load block into controller

#### Advantages

- Reloading of newly defined tags without interrupting the running process. The controller stays in "RUN" mode.

#### Properties

- Downloading without reinitialization is only possible for optimized blocks.
- The newly defined tags are initialized. The existing tags keep their current value.
- A block with reserve requires more memory space in the controller.
- The memory reserve depends on the work memory of the controller; however, it is max. 2 MB.
- It is assumed that a memory reserve has been defined for block.
- By default, the memory reserve is set to 100 byte.
- The memory reserve is defined individually for every block.
- The blocks can be variably expanded.

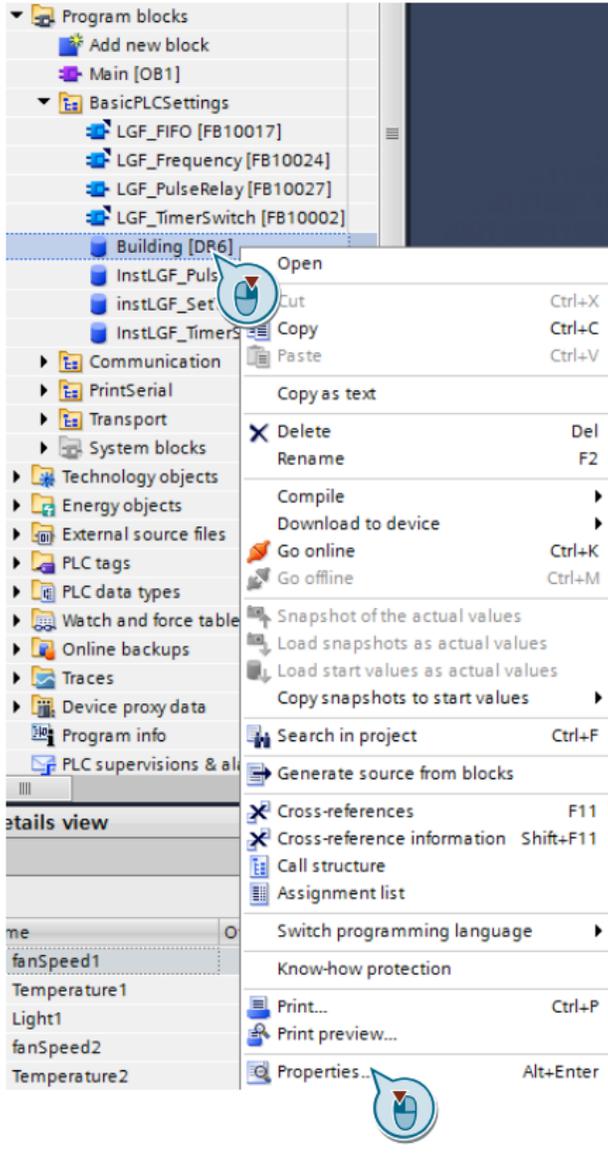
### Recommendation

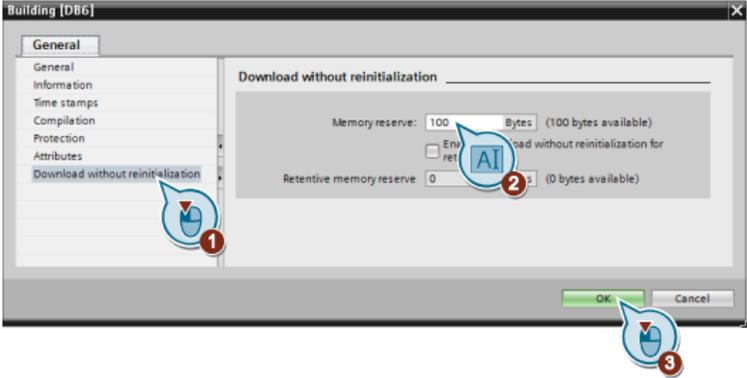
- Define a memory reserve for blocks that are to be expanded during commissioning (e.g. test blocks). The commissioning process is not disturbed by a download since the actual values of the existing tags remain.

### Example: Setting memory reserve on the block

The following table describes how you can set the memory reserve for the downloading without reinitializing.

Table 3-3: Setting memory reserve

Step	Instruction
1.	<p data-bbox="493 594 1312 625">Right-click any optimized block in the project tree and select "Properties".</p> 

Step	Instruction
2.	 <p>Click "Download without reinitialization".</p> <ol style="list-style-type: none"> <li>Enter the desired memory reserve for "Memory reserve".</li> <li>Confirm with "OK".</li> </ol>

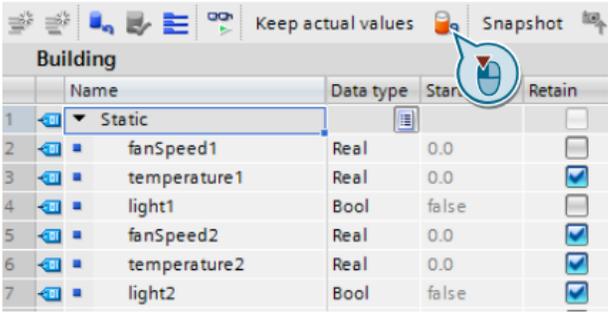
**Note** You can also set a default value for the size of the memory reserve for new blocks in the TIA portal.

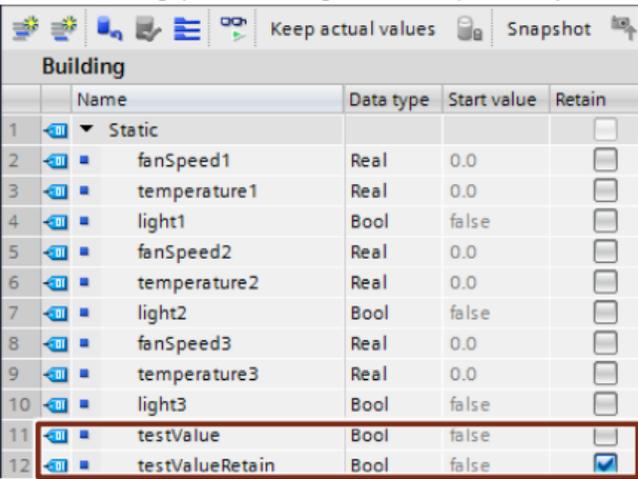
In the menu bar, navigate to "Options – Settings" and then to "PLC programming – General – Download without reinitialization".

**Example: Downloading without reinitialisation**

The following example displays how to download without reinitialization.

Table 3-4 Load without reinitialization

Step	Instruction
1.	Prerequisite: a memory reserve has to be set (see above)
2.	Open, e.g. an optimized global DB.
3.	<p>Click the "Activate memory reserve" button and confirm the dialog with "OK".</p> 

Step	Instruction																																																																						
4.	<p>Add a new tag (retentive tags are also possible).</p>  <table border="1" data-bbox="451 310 1089 789"> <thead> <tr> <th colspan="5">Building</th> </tr> <tr> <th></th> <th>Name</th> <th>Data type</th> <th>Start value</th> <th>Retain</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>Static</td> <td></td> <td></td> <td><input type="checkbox"/></td> </tr> <tr> <td>2</td> <td>fanSpeed1</td> <td>Real</td> <td>0.0</td> <td><input type="checkbox"/></td> </tr> <tr> <td>3</td> <td>temperature1</td> <td>Real</td> <td>0.0</td> <td><input type="checkbox"/></td> </tr> <tr> <td>4</td> <td>light1</td> <td>Bool</td> <td>false</td> <td><input type="checkbox"/></td> </tr> <tr> <td>5</td> <td>fanSpeed2</td> <td>Real</td> <td>0.0</td> <td><input type="checkbox"/></td> </tr> <tr> <td>6</td> <td>temperature2</td> <td>Real</td> <td>0.0</td> <td><input type="checkbox"/></td> </tr> <tr> <td>7</td> <td>light2</td> <td>Bool</td> <td>false</td> <td><input type="checkbox"/></td> </tr> <tr> <td>8</td> <td>fanSpeed3</td> <td>Real</td> <td>0.0</td> <td><input type="checkbox"/></td> </tr> <tr> <td>9</td> <td>temperature3</td> <td>Real</td> <td>0.0</td> <td><input type="checkbox"/></td> </tr> <tr> <td>10</td> <td>light3</td> <td>Bool</td> <td>false</td> <td><input type="checkbox"/></td> </tr> <tr> <td>11</td> <td>testValue</td> <td>Bool</td> <td>false</td> <td><input type="checkbox"/></td> </tr> <tr> <td>12</td> <td>testValueRetain</td> <td>Bool</td> <td>false</td> <td><input checked="" type="checkbox"/></td> </tr> </tbody> </table>	Building						Name	Data type	Start value	Retain	1	Static			<input type="checkbox"/>	2	fanSpeed1	Real	0.0	<input type="checkbox"/>	3	temperature1	Real	0.0	<input type="checkbox"/>	4	light1	Bool	false	<input type="checkbox"/>	5	fanSpeed2	Real	0.0	<input type="checkbox"/>	6	temperature2	Real	0.0	<input type="checkbox"/>	7	light2	Bool	false	<input type="checkbox"/>	8	fanSpeed3	Real	0.0	<input type="checkbox"/>	9	temperature3	Real	0.0	<input type="checkbox"/>	10	light3	Bool	false	<input type="checkbox"/>	11	testValue	Bool	false	<input type="checkbox"/>	12	testValueRetain	Bool	false	<input checked="" type="checkbox"/>
Building																																																																							
	Name	Data type	Start value	Retain																																																																			
1	Static			<input type="checkbox"/>																																																																			
2	fanSpeed1	Real	0.0	<input type="checkbox"/>																																																																			
3	temperature1	Real	0.0	<input type="checkbox"/>																																																																			
4	light1	Bool	false	<input type="checkbox"/>																																																																			
5	fanSpeed2	Real	0.0	<input type="checkbox"/>																																																																			
6	temperature2	Real	0.0	<input type="checkbox"/>																																																																			
7	light2	Bool	false	<input type="checkbox"/>																																																																			
8	fanSpeed3	Real	0.0	<input type="checkbox"/>																																																																			
9	temperature3	Real	0.0	<input type="checkbox"/>																																																																			
10	light3	Bool	false	<input type="checkbox"/>																																																																			
11	testValue	Bool	false	<input type="checkbox"/>																																																																			
12	testValueRetain	Bool	false	<input checked="" type="checkbox"/>																																																																			
5.	Download the block to the controller.																																																																						
6.	<p>Result:</p> <ul style="list-style-type: none"> <li>Actual values of the block remain</li> </ul>																																																																						

**Note**

Further information can be found in the online help of the TIA Portal under "Loading block extensions without reinitialization".

For further information, refer to the following entry:

How is the declaration table for global data blocks structured in STEP 7-1500 (TIA Portal)?

<https://support.industry.siemens.com/cs/ww/en/view/68015630>

### 3.2.9 Reusability of blocks

The block concept offers you a number of options to program in a structured and effective way.

#### Advantages

- Blocks can be used universally in any location of the user program.
- Blocks can be used universally in different projects.
- When every block receives an independent task, a clear and well-structured user program is automatically created.
- There are clearly fewer sources of errors.
- Simple error diagnostic possible.

#### Recommendation

If you want to reuse the block, please note the following recommendations:

- Always look at blocks as encapsulated functions. I.e. each block represents a completed partial task within the entire user program.
- Use several cyclic Main OBs to group the plant parts.
- Always execute a data exchange between the blocks via its interfaces and not via its instances (chapter [3.4.1 Block interfaces as data exchange](#)).
- Do not use project-specific data and avoid the following block contents:
  - Access to global DBs and use of single-instance DBs
  - Access to tags
  - Access to global constants
- Reusable blocks have the same requirements as know-how-protected blocks in libraries. This is why you have to check the blocks for reusability based on the "Multiple instance capability" block property. Compile the block before the check.

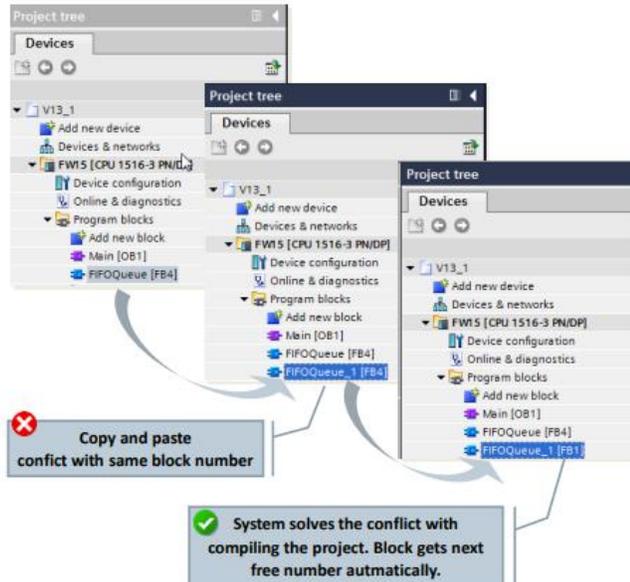
Figure 3-15: Block attributes



### 3.2.10 Auto numbering of blocks

For internal processing, required block numbers are automatically assigned by the system (setting in the block properties).

Figure 3-16: Auto numbering of blocks



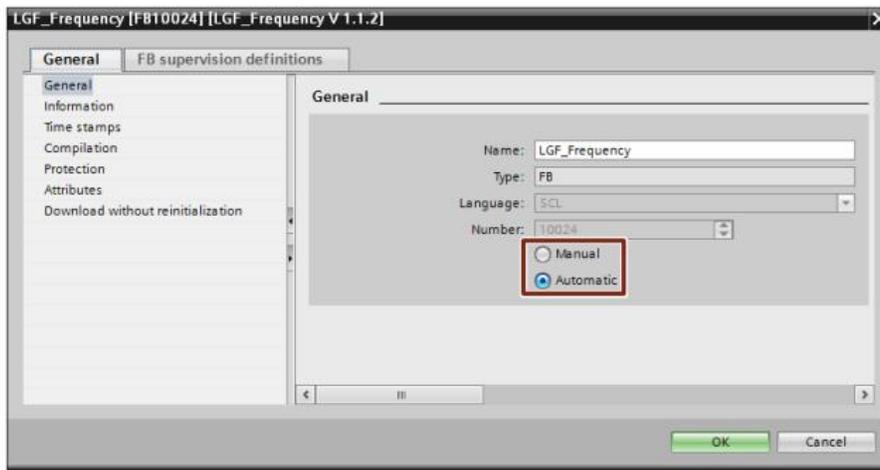
#### Advantages

- Conflicting block numbers, e.g. as a result of copying, automatically deletes the TIA Portal during compilation.

#### Recommendation

- Leave the existing setting "Automatic" unchanged.

Figure 3-17: Setting in the block



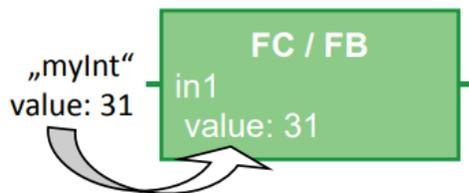
### 3.3 Block interface types

FBs and FCs have three different interface types: In, InOut and Out. Via these interface types the blocks are provided with parameters. The parameters are processed and output again in the block. InOut parameters serve for the transfer of data to the called block as well as the return of results. There are two different options for the parameter transfer of data.

#### 3.3.1 Call-by-value

When calling the block, the value of the actual parameter is copied onto the formal parameter of the block. For this, an additional memory in the called block is provided.

Figure 3-18: Transfer of the value



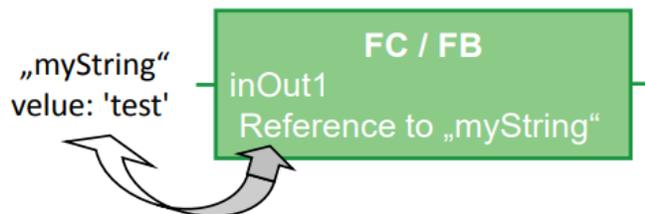
#### Properties

- Each block displays the same behavior as the transferred parameters
- Values are copied when calling the block

#### 3.3.2 Call-by-reference

When calling the block, a reference is transferred to the address of the actual parameter. For this, no additional memory is required.

Figure 3-19: Referencing the actual parameter (pointer to data storage of the parameter)



#### Properties

- Each block displays the same behavior as the referenced parameters.
- Actual parameters are referenced when the block is called, i.e. with the access, the values of the actual parameter are directly read or written.

#### Recommendation

- Generally use the InOut interface type for structured tags (e.g. of the ARRAY, STRUCT, STRING, type...) in order to avoid enlarging the required data memory unnecessarily.

### 3.3.3 Overview for transfer of parameters

The following table gives a summarized overview of how S7-1200/1500 block parameters with elementary or structured data types are transferred.

Table 3-5: Overview for transfer of parameters

Block type / formal parameter		Elementary data type	Structured data type
FC	Input	Copy	<b>Reference</b>
	Output	Copy	<b>Reference</b>
	InOut	Copy	<b>Reference</b>
FB	Input	Copy	Copy
	Output	Copy	Copy
	InOut	Copy	<b>Reference</b>

#### Note

When optimized data with the property "non-optimized access" is transferred when calling the block, it is generally transferred as copy. When the block contains many structured parameters this can quickly lead to the temporary storage area (local data stack) of the block to overflow.

This can be avoided by setting the same access type for both blocks (chapter [2.6.5 Parameter transfer between blocks with optimized and non-optimized access](#)).

## 3.4 Memory concept

For STEP 7 there is generally the difference between the global and local memory area. The global memory area is available for each block in the user program. The local memory area is only available within the respective block.

### 3.4.1 Block interfaces as data exchange

If you are encapsulating the functions and program the data exchange between the blocks only via the interfaces, you will clearly have advantages.

#### Advantages

- Program can be made up modularly from ready blocks with partial tasks.
- Program is easy to expand and maintain.
- Program code is easier to read and test since there are no hidden cross accesses.

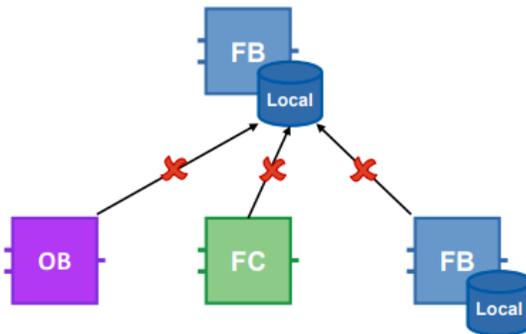
#### Recommendation

- If possible, only use local tags. Thus, you can use the blocks universally and in a modular fashion.

### 3.4 Memory concept

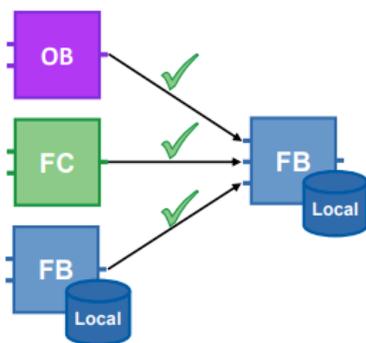
- Use the data exchange via the block interfaces (In, Out, InOut), his guarantees the reusability of the blocks.
- Only use the instance data blocks as local memory for the respective function block. Other blocks should not be written into instance data blocks.

Figure 3-20: Avoiding access to instance data blocks



If only the block interfaces are used for the data exchange it can be ensured that all blocks can be used independent from each other.

Figure 3-21: Block interfaces for data exchange



#### 3.4.2 Global memory

Memories are called global when they can be accessed from any location of the user program. There are hardware-dependent memories (for example, bit memory, times, counters, etc.) and global DBs. For hardware-dependent memory areas there is the danger that the program may not be portable to any controller because the areas there may already be used. This is why you should use global DBs instead of hardware-dependent memory areas.

##### Advantages

- User programs can be used universally and independent from the hardware.
- The user program can be modularly configured without having to divide bit memory areas for different users.
- Optimized global DBs are clearly more powerful than the bit memory address area that is not optimized for reasons of compatibility.

## 3.4 Memory concept

### Recommendation

- Do not use any bit memory and use global DBs instead.
- Avoid hardware-dependent memory, such as, for example, clock memory or counter. Use the IEC counter and timer in connection with multi-instances instead (chapter [3.2.5 Multi-instances](#)). The IEC timers can be found in "Instructions – Basic Instructions – Timer operations".

Figure 3-22: IEC timers

Timer operations	
TP	Generate pulse
TON	Generate on-delay
TOF	Generate off-delay
TONR	Time accumulator
[TP]-	Start pulse timer
[TON]-	Start on-delay timer
[TOF]-	Start off-delay timer
[TONR]-	Time accumulator
[RT]-	Reset timer
[PT]-	Load time duration

### 3.4.3 Local memory

- Static tags
- Temporary tags

### Recommendation

- Use the static tags if the values are required in the next cycle.
- Use the temporary tags as intermediate memory in the current cycle. The access time for temporary tags is shorter than for static ones.
- If an Input/Output tags is accessed very frequently, use a temporary tag as intermediate memory to save runtime.

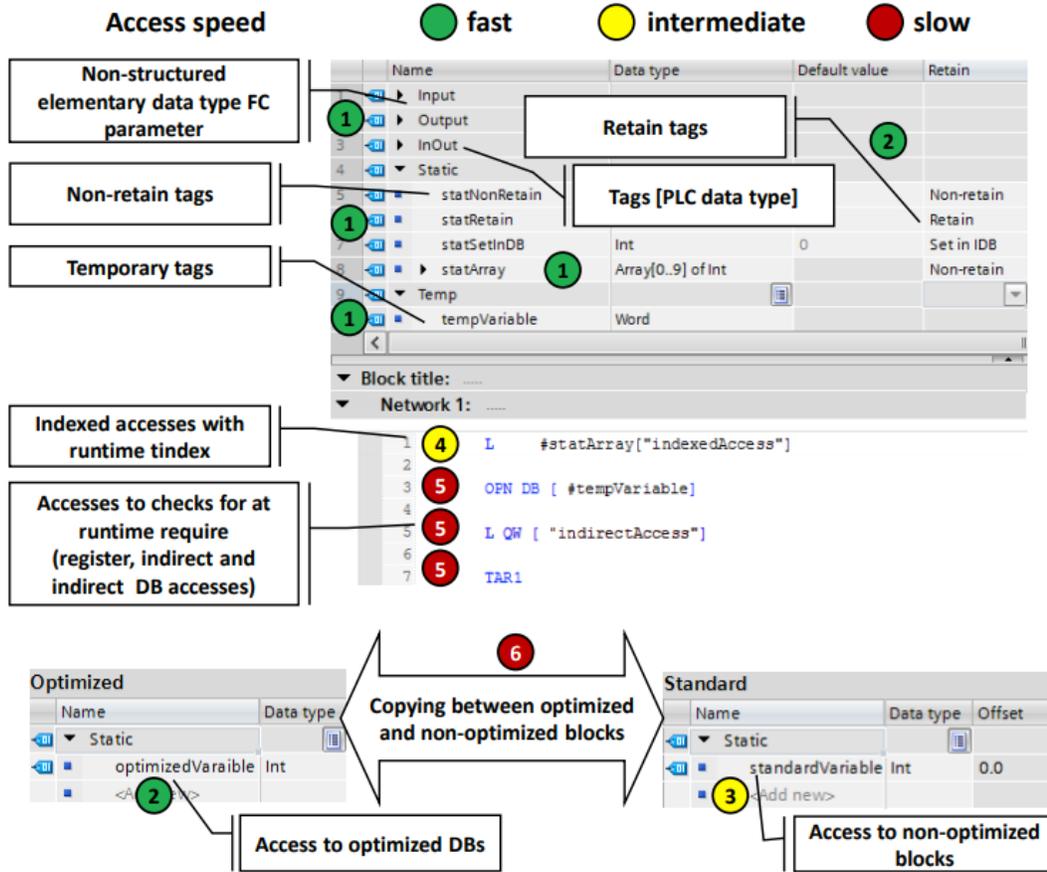
### Note

Optimized blocks: Temporary tags are initialized in each block call with the default value (S7-1500 / S7-1200 firmware V4 or higher).  
Non-optimized blocks: Temporary tags are undefined for each call of the block.

### 3.4.4 Access speed of memory areas

STEP 7 offers different options of memory access. For system-related reasons there are faster and slower accesses to different memory areas.

Figure 3-23: Different memory access



#### Fastest access in the S7-1200/1500 in descending order

1. Optimized blocks: Temporary tags, parameters of an FC and FB, non-retentive static tags, tags [PLC data type]
2. Optimized blocks whose access for compiling is known:
  - Retentive FB tags
  - Optimized global DBs
3. Access to non-optimized blocks
4. Indexed accesses with index that was calculated at runtime (e.g. Motor [i])
5. Accesses that require checks at runtime
  - Accesses to DBs that are created at runtime or which were opened indirectly (e.g. OPN DB[i])
  - Register access or indirect memory access
6. Copying of structures between optimized and non-optimized blocks (apart from Array of Bytes)

### 3.5 Retentivity

In the event of a failure of the power supply, the controller copies the retentive data with its buffer energy from the controller's work memory to a non-volatile memory. After restarting the controller, the program processing is resumed with the retentive data. Depending on the controller, the data volume for retentivity has different sizes.

Table 3-6: Retentive memory for S7-1200/1500

Controller	Usable retentive memory for bit memory, times, counters, DBs and technology objects
CPU 1211C,1212C, 1214C, 1215C, 1217C	10 kByte
CPU 1511-1 PN	88 kByte
CPU 1513-1 PN	88 kByte
CPU 1515-2 PN, CPU 1516-3 PN/DP	472 kByte
CPU 1518-4 PN/DP	768 kByte

Table 3-7: Differences of S7-1200 and S7-1500

S7-1200	S7-1500
Retentivity can <b>only</b> be set for bit memory	Retentivity can be set for bit memory, times and counters

#### Advantages

- Retentive data maintains its value when the controller goes to STOP and back to RUN or in the event of power failure and a restart of the controller.

#### Properties

For elementary tags of an optimized DB the retentivity can be set separately. Non-optimized data blocks can only be defined completely retentive or non-retentive.

Retentive data can be deleted with the actions "memory reset" or "Reset to factory settings" via:

- Operating switch on the controller (MRES)
- Display of the controller
- Online via STEP 7 (TIA Portal)

#### Recommendation

- Do not use the setting "Set in IDB". Always set the retentive data in the function block and not in the instance data block.  
The "Set in IDB" setting increases the processing time of the program sequence. Always either select "Non-retain" or "Retain" for the interfaces in the FB.

### 3.5 Retentivity

Figure 3-24: Program editor (Functions block interfaces)

▼ Static			
▶ instToffTimePulse	TOF_...		Non-retain
▶ instToffTimePause	TOF_TIME		Non-retain
statFrequency	Real	0.0	Non-retain
statTimePeriod	Time	T#0ms	Non-retain
statTimePulse	Time	T#0ms	Non-retain

Figure 3-25: Program editor (data block)

Building					
Name	Data type	Start value	Retain	Accessible f...	
▼ Static			<input type="checkbox"/>		<input type="checkbox"/>
fanSpeed1	Real	0.0	<input type="checkbox"/>		<input checked="" type="checkbox"/>
temperature1	Real	0.0	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>
light1	Bool	false	<input type="checkbox"/>		<input checked="" type="checkbox"/>
fanSpeed2	Real	0.0	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>
temperature2	Real	0.0	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>
light2	Bool	false	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>
fanSpeed3	Real	0.0	<input type="checkbox"/>		<input checked="" type="checkbox"/>
temperature3	Real	0.0	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>
light3	Bool	false	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>

#### Example: Retentive PLC tags

The setting of the retentive data is performed in the tables of the PLC tags, function blocks and data blocks.

Figure 3-26: Setting of the retentive tags in the table of PLC tags

PLC tags

Name	Tag table	Data type	Address
1 activateLeft	Default tag table	Bool	%I0.0
2 activateRight	Default tag table	Bool	%I0.1
3			
4			
5			
6			

**Retain memory**

Number of memory bytes starting at MB0: 0

Number of SIMATIC timers starting at T0: 0

Number of SIMATIC counters starting at C0: 0

Currently available retain memory (bytes): 90784

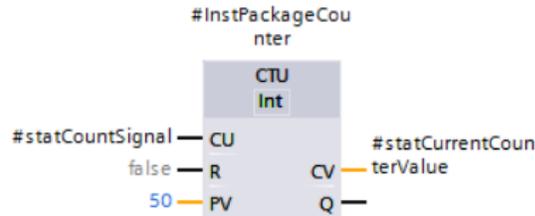
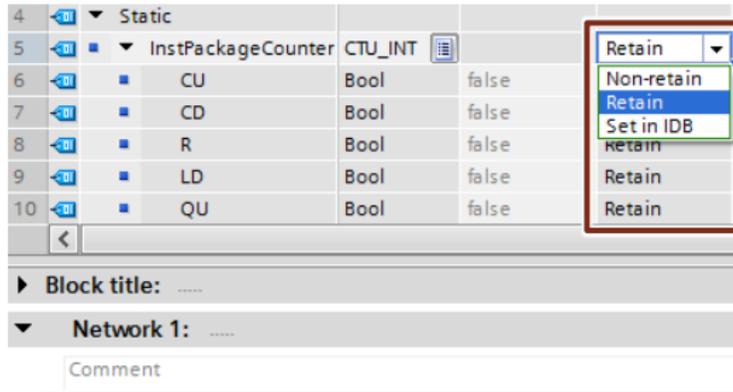
OK Cancel

Retentivity can be set from address 0 onward!  
e.g. from MB0, T0 or C0

**Example: Retentive counter**

You can also declare instances of functions (timer, counter, etc.) retentive. As already described in chapter [3.2.5 Multi-instances](#), you should always program such functions as multi-instance.

Figure 3-27: Retentive counter as multi-instance



**Note**

If the retentive memory on the PLC is not sufficient, it is possible to store data in the form of data blocks that are only located in the load memory of the PLC. The following entry is described by taking the example of an S7-1200. This programming also works for S7-1500.

More information can be found in the following entries:

How do you configure data blocks in STEP 7 (TIA Portal) with the "Only store in load memory" attribute for a S7-1200?

<https://support.industry.siemens.com/cs/ww/en/view/53034113>

Using Recipe Functions for persistent Data with SIMATIC S7-1200 and S7-1500

<https://support.industry.siemens.com/cs/ww/en/view/109479727>

## 3.6 Symbolic addressing

### 3.6.1 Symbolic instead of absolute addressing

The TIA Portal is optimized for symbolic programming. This results in many advantages. Due to symbolic addressing, you can program without having to pay attention to the internal data storage. The controller handles where the best possible storage is for the data. You can therefore completely concentrate on the solution for your application task.

#### Advantages

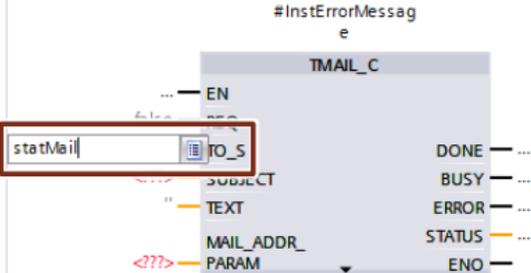
- Easier to read programs through symbolic tag names
- Automatic update of tag names at all usage locations in the user program
- Memory storage of the program data does not have to be manually managed (absolute addressing)
- Powerful data access
- No manual optimization for performance or program size reasons required
- Auto-complete for fast symbol input
- Fewer program errors due to type safety (validity of data types is checked for all accesses)

#### Recommendation

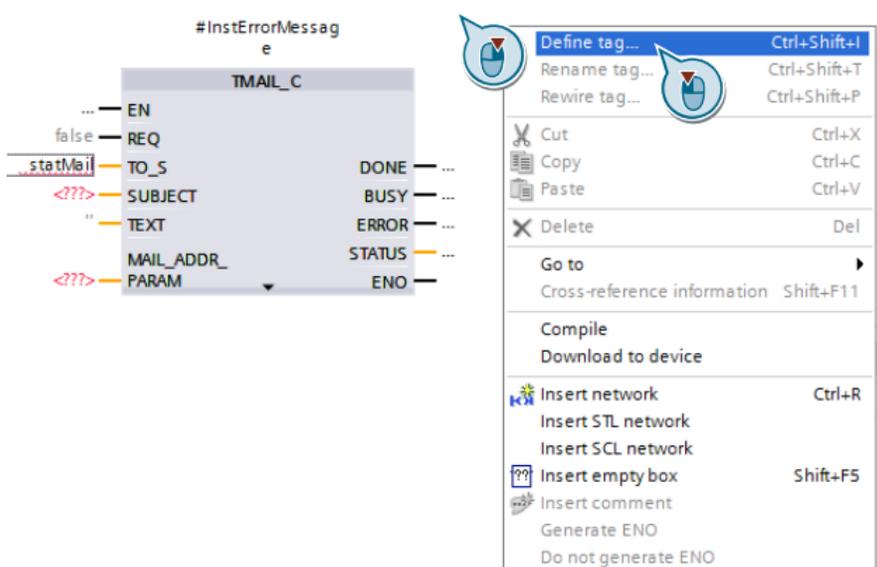
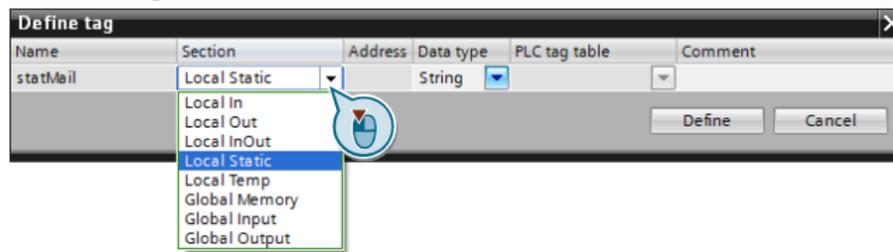
- "Don't worry about the storage of the data"
- "Think" symbolically. Enter the "descriptive" name for each function, tag or data, such as, for example, Pump\_boiler\_1, heater\_room\_4, etc. Thus a created program can be simply read, without requiring many comments.
- Give all the tags used a direct symbolic name and define them afterwards with a right-click.

#### Example

Table 3-8: Example for creating symbolic tags

Step	Instruction
1.	Open the program editor and open any block.
2.	Enter a symbolic name directly at the input of an instruction. 

### 3.6 Symbolic addressing

Step	Instruction
3.	<p>Right-click next to the block and select "Define tag..." in the context menu.</p> 
4.	<p>Define the tag.</p> 

There is an elegant method to save time, if you want to define several tags in a network. First of all, assign all tag names. Then define all tags at the same time with the dialog of step 4.

**Note**

More information can be found in the following entry:

What are the advantages of using symbolic addressing for S7-1500 in STEP 7 (TIA Portal)?

<https://support.industry.siemens.com/cs/ww/en/view/67598995>

### 3.6.2 ARRAY data type and indirect field accesses

The ARRAY data type represents a data structure which is made up of several elements of a data type. The ARRAY data type is suitable, for example, for the storage of recipes, material tracking in a queue, cyclic process acquisition, protocols, etc.

Figure 3-28: ARRAY with 10 elements of the Integer (INT) data type

Name	Data type
statArray	Array[0..9] of Int
statArray[0]	Int
statArray[1]	Int
statArray[2]	Int
statArray[3]	Int
statArray[4]	Int
statArray[5]	Int
statArray[6]	Int
statArray[7]	Int
statArray[8]	Int
statArray[9]	Int

You can indirectly access individual elements in the ARRAY with an index (`array["index"]`).

Figure 3-29: Indirect field access

KOP / FUP:



SCL:

```
1 #statField := #statArray[#statIndex];
```

#### Advantages

- Easy access through ARRAY index
- No complicated pointer creation required
- Fast creation and expansion possible
- Useable in all programming languages

#### Properties

- Structured data type
- Data structure made of fixed number of elements of the same data type
- ARRAYS can also be created multi-dimensional
- Possible indirect access with runtime tag with dynamic index calculation at runtime

## Recommendation

- Use ARRAY for indexed accesses instead of pointer (e.g. ANY pointer). This makes it easier to read the program since an ARRAY is more meaningful with a symbolic name than a pointer in a memory area.
- As run tag use the DINT data type as temporary tag for highest performance.
- Use the "MOVE\_BLK" instruction to copy parts of an ARRAY into another one.
- Use the "GET\_ERR\_ID" instruction to catch access errors within the Array.

## Note

More information can be found in the following entries:

How do you implement an array access with an S7-1500 with variable index?  
<https://support.industry.siemens.com/cs/ww/en/view/67598676>

How do you address securely and indirectly in STEP 7 (TIA Portal)?  
<https://support.industry.siemens.com/cs/ww/en/view/97552147>

In STEP 7 (TIA Portal), how do you transfer S7-1500 data between two tags of the data types "Array of Bool" and "Word"?  
<https://support.industry.siemens.com/cs/ww/en/view/108999241>

### 3.6.3 Formal parameter Array [\*] (V14 or higher)

With the formal parameter Array [\*], arrays with variable length can be transferred to functions and function blocks.

With the instructions "LOWER\_BOUND" and "UPPER\_BOUND" the array limits can be determined.

#### Advantages

- Blocks that can process the flexible arrays with different lengths
- Optimum readability due to fully-symbolic programming
- No pointer programming for arrays of different lengths necessary anymore

#### Example

Figure 3-30: Initializing different arrays

**Main**

Name	Data type
1 Input	
2 Temp	
3 tempArray1	Array[0..125] of Real
4 tempArray2	Array[10..80] of Real
5 Constant	

Block title: "Main Program Sweep (Cycle)"

Network 1: Array initialization

```

graph TD
    subgraph Network1 [Network 1: Array initialization]
        direction TB
        I1[Input] --- EN1[EN]
        I1 --- quantityArray1[quantityArray]
        EN1 --- ENO1[ENO]
        subgraph InitArray1 [InitArray]
            EN1 --- quantityArray1
        end
        I2[Input] --- EN2[EN]
        I2 --- quantityArray2[quantityArray]
        EN2 --- ENO2[ENO]
        subgraph InitArray2 [InitArray]
            EN2 --- quantityArray2
        end
    end
  
```

**InitArray**

Name	Data type	Default value
1 Input		
2 Output		
3 InOut		
4 quantityArray	Array[*] of Real	
5 Temp		
6 tempLower	DInt	
7 tempUpper	DInt	
8 count	DInt	

```

IF... CASE... FOR... WHILE... (*...*) REGION
OF... TO DO... DO... DO...
1 #tempLower := LOWER_BOUND(ARR := #quantityArray, DIM := 1);
2 #tempUpper := UPPER_BOUND(ARR := #quantityArray, DIM := 1);
3
4 FOR #count := #tempLower TO #tempUpper DO
5   #quantityArray[#count] := 0.0;
6 END_FOR;
  
```

### 3.6.4 STRUCT data type and PLC data types

The STRUCT data type represents a data structure which is made up of elements of different data types. The declaration of a structure is performed in the respective block.

Figure 3-31: Structure with elements with different data types

Name	Data type	Default value
statEngineData	Struct	
power	Struct	
maxpower	Int	1000
cosPhi	Real	0.89
<Add new>		
outputValues	Struct	
voltage	Real	0.0
current	Real	0.0
frequency	Real	0.0
<Add new>		

In comparison to structures, PLC data types are defined across the controller in the TIA Portal and can be centrally changed. All usage locations are automatically updated.

PLC data types are declared in the "PLC data types" folder in the project navigation before being used.

Figure 3-32: PLC data types

typeEngineData			
Name	Data type	Default value	
1 power	Struct		
2 maxpower	Int	1000	
3 cosPhi	Real	0.89	
4 outputValues	Struct		
5 voltage	Real	0.0	
6 current	Real	0.0	
7 frequency	Real	0.0	
8 <Add new>			

#### Advantages

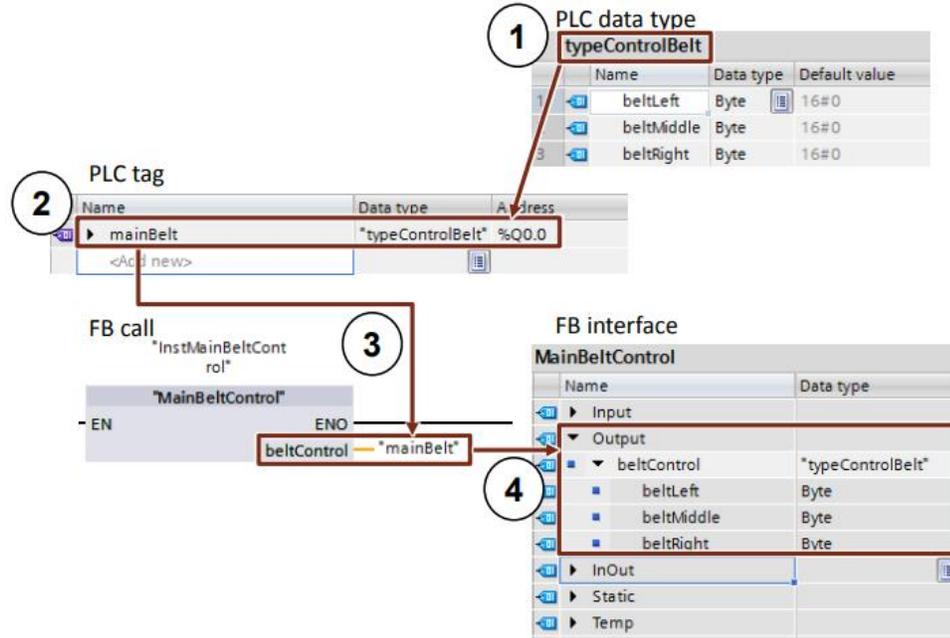
- A change in a PLC data type is automatically updated in all usage locations in the user program.
- Simple data exchange via block interfaces between several blocks
- In PLC data types STRING tags with defined length can be declared (e.g., String[20]). As of TIA V14 a global constant can also be used for the length (e.g., String[LENGTH]).  
If a STRING tag is declared without defined length, the tag has the maximum length of 255 characters.



### 3.6.5 Access to I/O areas with PLC data types

With S7-1500 controllers, you can create PLC data types and use them for structured and symbolic access to inputs and outputs.

Figure 3-35: Access to I/O areas with PLC data types



7. PLC data type with all required data
8. PLC tag of the type of the created PLC data type and start address of the I/O data area (%Ix.0 or %Qx.0, e.g., %I0.0, %Q12.0, ...)
9. Transfer of the PLC tag as actual parameter to the function block
10. Output of the function block is of the type of the created PLC data type

#### Advantages

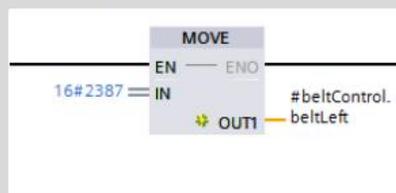
- High programming efficiency
- Easy multiple usability thanks to PLC data types

#### Recommendation

- Use PLC data types for access to I/O areas, for example, to symbolically receive and send drive telegrams.

#### Note

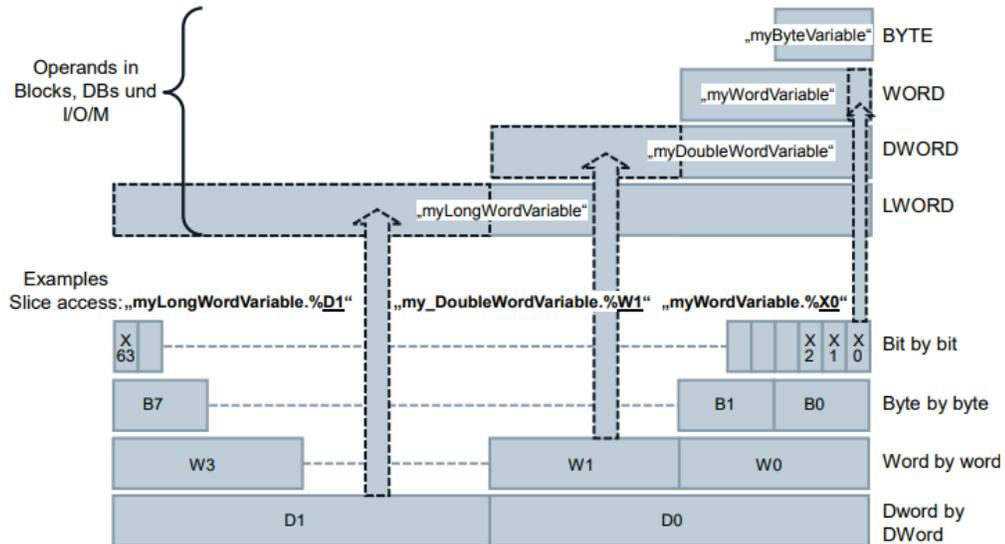
Individual elements of a PLC data type of a tag can also be directly accessed in the user program:



### 3.6.6 Slice access

For S7-1200/1500 controllers, you can access the memory area of tags of the Byte, Word, DWord or LWord data type. The division of a memory area (e.g. byte or word) into a smaller memory area (e.g. Bool) is also called slice. The figure below displays the symbolic bit, byte and word accesses to the operands.

Figure 3-36: Symbolic bit, byte, word, DWord slice access



#### Advantages

- High programming efficiency
- No additional definition in the tag declaration required
- Easy access (e.g. control bits)

#### Recommendation

- Use the slice access via AT construct rather than accessing certain data areas in operands.

#### Note

More information can be found in the following entry:

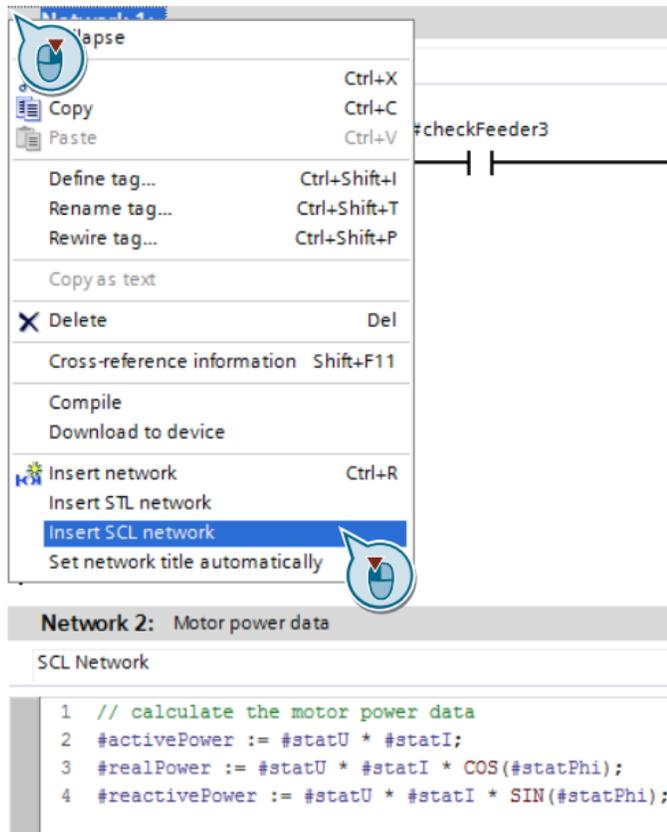
How in STEP 7 (TIA Portal) can you access the unstructured data types bit-by-bit, byte-by-byte or word-by-word and symbolically?

<https://support.industry.siemens.com/cs/ww/en/view/57374718>

### 3.6.7 SCL networks in LAD and FBD (V14 and higher)

With SCL networks you can make calculations in LAD and FBD that can only be programmed with considerable effort in LAD and FBD instructions.

Figure 3-37: Inserting SCL network



#### Advantages

- Time saving through efficient programming
- Clear code, thanks to symbolic programming

#### Properties

- Supports all SCL instructions
- Supports comments

#### Recommendation

- Use the SCL networks in LAD and FBD for mathematical calculations instead of instructions, such as ADD, SUBB etc.

“



This work is licensed under a Creative Commons Attribution 4.0 International License.