Contents

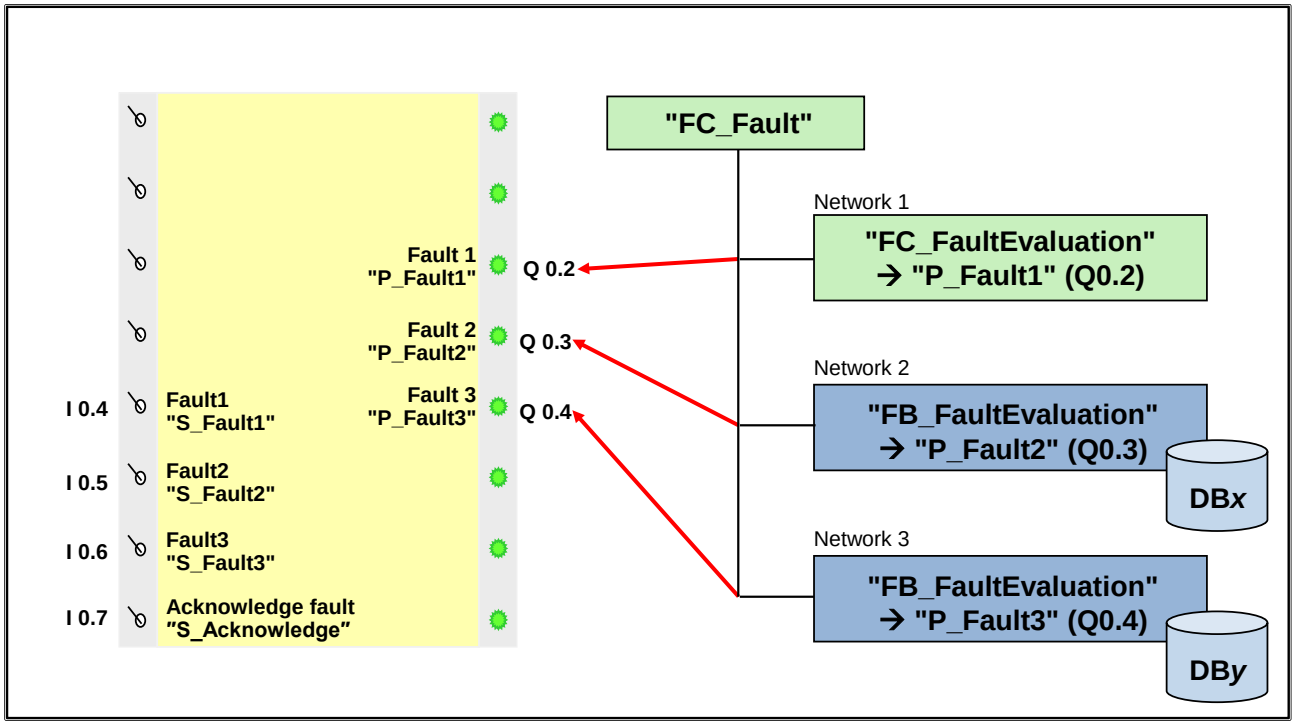
9

9. Functions and Function Blocks

At the end of the chapter the participant will ...

- **... be familiar with the purpose of parameter-assignable blocks**
- **… be familiar with the declaration section of a block**
- **… be familiar with the purpose of temporary variables**
- **... be familiar with the purpose of static variables**
- **… know what a structured programming is**
- **... be able to program parameter-assignable functions and function blocks and their calls**

9.1. Task Description: Fault Evaluation with Parameter-assignable Blocks



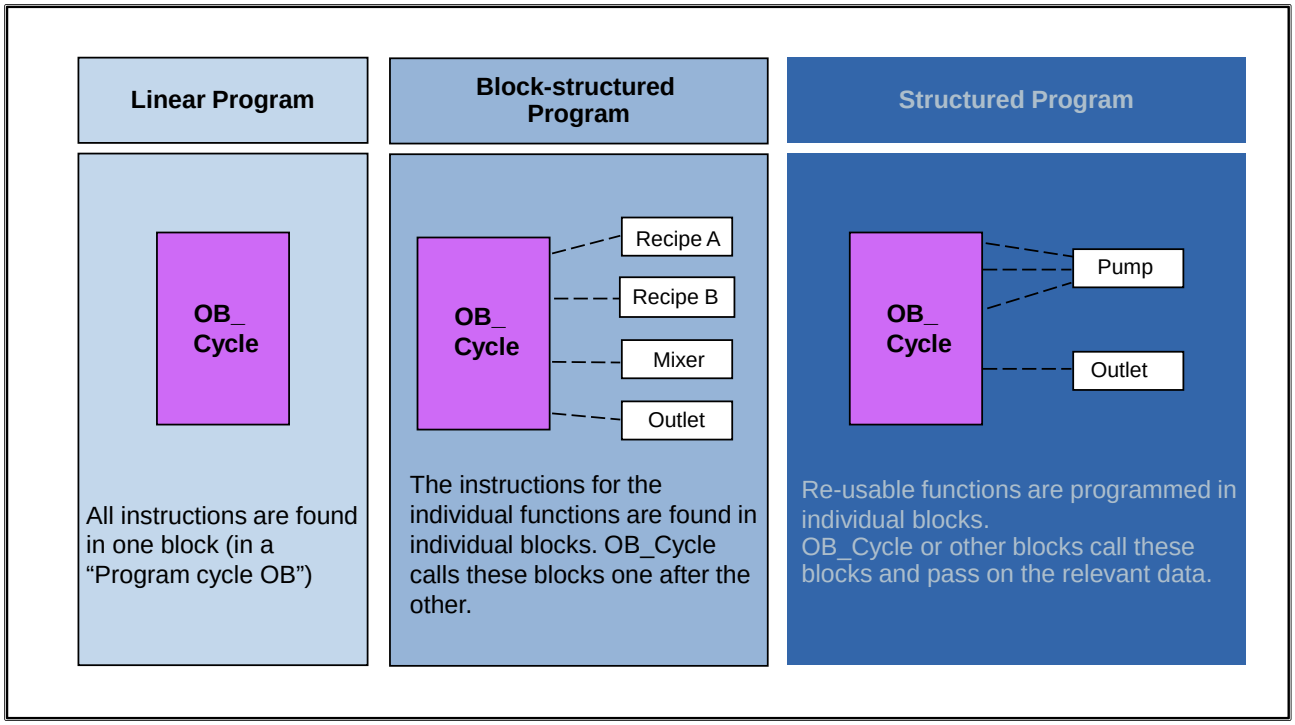
Task Description

Independent of the conveyor model functions so far, 3 different faults are to be evaluated as follows.

If a fault is triggered at the simulator inputs "S_Fault1" (I 0.4) to "S_Fault3" (I 0.6), the associated simulator LEDs "P_Fault1" (Q0.2) to "P_Fault3" (Q0.4) begin to flash.

A group acknowledgement for all faults takes place using the simulator input "S_Acknowledge" (I 0.7). If the fault still exists after acknowledgement, the LED changes to constant light; if the fault no longer exists, the LED goes dark.

9.2. Structured Programming



Linear Program

The entire program is found in one continuous program block (Program cycle OB) which is automatically called by the system. This model resembles a hard-wired relay control that was replaced by an automation system (programmable logic controller). The CPU processes the individual instructions one after the other.

Block-structured Program

The program is divided into blocks, whereby every block only contains the program for solving a partial task. Further structuring through networks is possible within a block. You can generate network templates for networks of the same type. Normally, a cyclically called Organization block contains instructions which call the other blocks in a defined sequence.

Structured Program

A structured program contains parameter-assignable blocks that are set up in such a way that they can be used universally. When a parameter-assignable block is called, it is passed current parameters (for example, the specific addresses of inputs and outputs as well as parameter values).

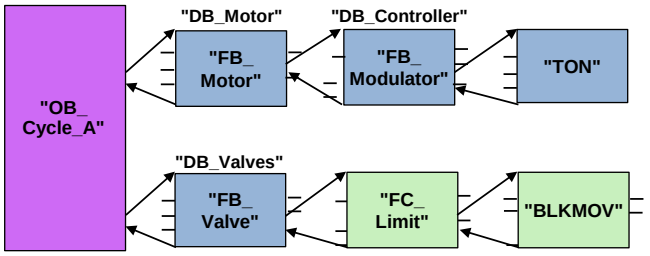
Example:

- A "pump block" contains instructions for the control of a pump.
- The program blocks, which are responsible for the control of special pumps, call the "pump block" and provide it with information about which pump is to be controlled with which parameters.

9.2.1. Modular and Re-usable Blocks

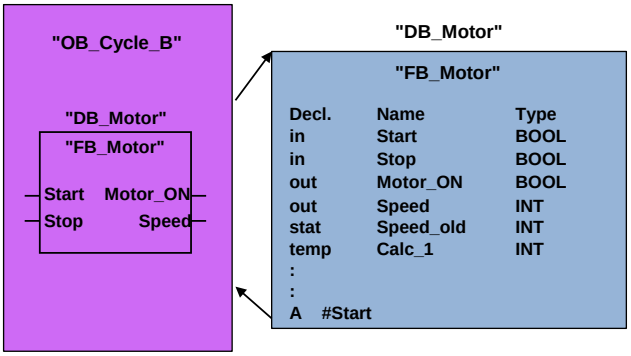
Modularization of the entire task:

- Partial tasks are solved in separate programs or blocks
- Parameter assignment enables flexible usage
  - Example: Drilling cycle with changeable depth



Re-usability of blocks:

- Blocks can be called as often as is required
- Restrictions:
  - No access to global operands
  - Communication only via the parameter list



Modularization of the Entire Task

Abstraction is the basis for solving complex problems, in which we concentrate on the fundamental aspects of a program in every abstraction level and ignore all the details that are not essential. Abstraction helps us to divide complex tasks into partial tasks which can then be solved on their own.

Parameter-assignable (Re-usable) Blocks

STEP 7 supports this concept of modularization with its block model. The partial tasks that result from the division of the entire task are assigned to blocks in which the necessary algorithms and data for solving the partial problems are stored. STEP 7 blocks such as functions (FC) and function blocks (FB) can be assigned parameters so that the concepts of structured programming can be implemented with them. This means:

- Blocks for solving partial tasks implement their own data management with the help of local variables.
- Blocks communicate with the "outside world", that is, with the sensors and actuators of the process control or with other blocks of the user program, exclusively through their block parameters. No access to global operands such as inputs, outputs, memory bits or variables in DBs can be made from within the statement section of blocks.

Advantages

- The blocks for the partial tasks can be created and tested independent of one another.
- Blocks can be called as often as is required in different locations with different parameter sets, that is, they can be reused.
- "Re-usable" blocks for special tasks can be delivered in pre-designed libraries.

9.2.2. Local and Global Operands

Global Operands (valid in the entire program)	**Local Operands** (only valid in one block)
• **PII / PIQ** • **I / O peripherals** • **Memory bits** • **Variables in DBs** (Chapter Data Blocks) • **S5-Timers and Counters** (not forS7-1200) • **Constants**	**Formal Parameters** (Input, Output, InOut) • interface for data exchange between calling and called block • temporary storage in the L-stack for FCs or storage in the IDB for FBs • can be used in FCs / FBs
	Temporary Variables (Temp) • are overwritten after the block is executed • temporary storage in the local data stack (L-stack) • can be used in OBs / FCs / FBs
	Static Variables (Static) • retain their value after the block is executed • permanent storage in instance data block (IDB) • can only be declared in FBs
	Constants (Constant) • read-only as well as only symbolic access • no memory usage • can be used in OBs / FCs / FBs

Global Operands

Global operands are valid throughout the entire S7 program. Accordingly, every code (logic) block (OB, FC, FB) can access these operands.

Global operands include inputs, outputs, memory bits, SIMATIC timers, SIMATIC counters, constants and variables which are declared in global data blocks (Chapter: Data Blocks).

Local Operands

Local operands are only valid in the block in which they were declared in the declaration part. Accordingly, only this block can access them.

- Formal Parameters

  Formal parameters form the interface between the calling and the called block (FC, FB). They are used to realize a data exchange between the calling and the called block.

- Temporary Variables

  Temporary variables can be declared in every code (logic) block (OB, FC, FB) and are managed in the local data stack of the CPU. Accordingly, they only retain their values while the block is being executed. For that reason, it is important that in the current cycle, a write access must have taken place on the temporary variable in the block before a read access can take place. They are, for example, unsuitable as auxiliary variables for edge evaluations or to store quantities. They are, in fact, used to store intermediate results, such as, for complex calculations or format conversions.

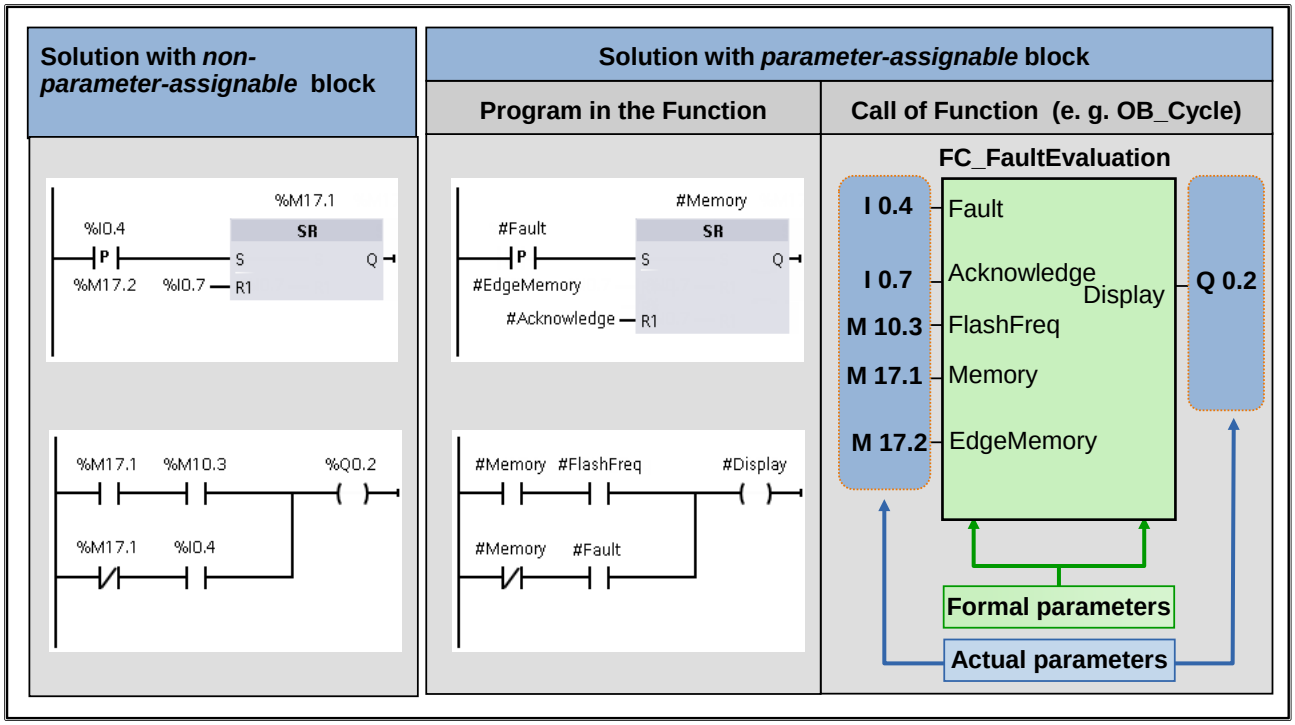
- Static Variables

  Static variables can only be declared in FBs and are stored in the associated instance data block. Accordingly, these variables retain their value even after the FB is executed.

- Constants

  Constants are fixed values which have a read-only access and which do not take up any memory space.

9.3. Solution with Parameter-assignable Block



Application

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

- The program only has to be created once, which significantly reduces programming effort.
- The block is only stored in the user memory once, which significantly reduces the amount of memory used.
- The block or the functionality implemented with the block can be called as often as you like, each time with different operands. For this, the formal parameters (input, output, or in/out parameters) are supplied with different actual parameters every time they are called.

Program Execution

When the block is executed, the formal parameters are replaced with the actual parameters passed during the call.

If, as in the example, during the call of the block, the memory byte M17.1 is passed as the actual parameter for the formal parameter *#Memory*, then, at runtime, the memory byte M17.1 is set or reset and its signal status is scanned etc.

Parameter-assignability

You can program FC or FB blocks as parameter-assignable. You cannot program organization blocks as parameter-assignable since they are called by the operating system and so the call cannot be programmed and also no actual parameters can be specified.

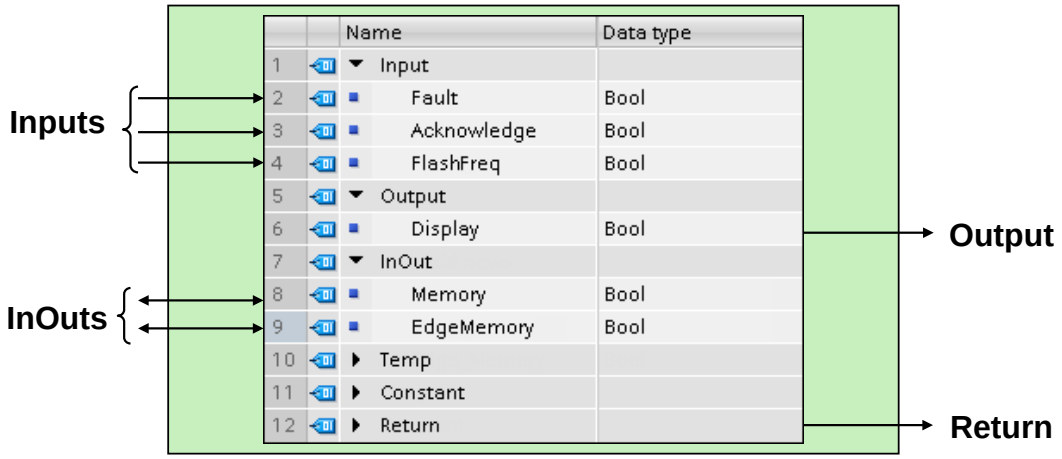
Our Example

Even if the function is required repeatedly in the system, you only have to program "FC_FaultEvaluation" once as parameter-assignable.

"FC_FaultEvaluation" is then called several times for the different fault evaluations and is assigned a different actual operand each time.

9.3.1. Declaration of Formal Parameters

Type of Parameter	Declaration	Use	Graphic Display
Input parameter	Input	Read only	To the left of the block box
Output parameter	Output	Write only	To the right of the block box
In/Out parameter	InOut	Read / Write	To the left of the block box



Formal Operands

Before you can create the program in the parameter-assignable block, you have to define the formal parameters in the declaration part.

Type of Parameter

In the table in the picture, you can see the three possible types of parameters and their use. Please note that formal operands that have a reading and a writing access have to be declared as 'In/Out' parameters.

Interface

The **Input**, **Output** and **InOut** parameters as well as the **Return** parameter form the interface of a block. The **Return** parameter is an additional Output parameter and, defined according to IEC 61131-3, the **Return value** of the function. The Return parameter only exists for FCs. If it has the data type VOID, it is not used and also does not appear as a parameter when the function is called.

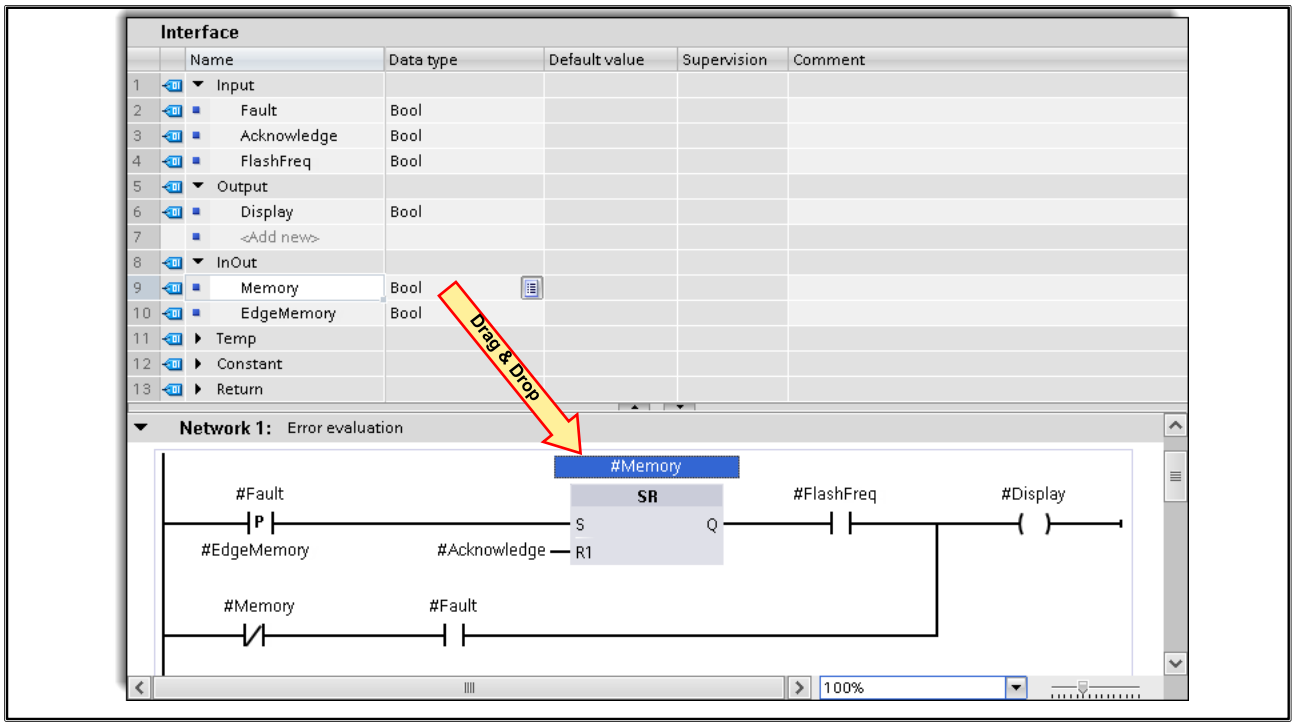
The variables **Temp** and **Constant** are – even though they are listed in the declaration section of the interface – not components of the block interface, since they do not become visible when the block is called.

Example:

The picture shows the declaration section, that is, the interface of a block. Since the formal parameters #Memory and #EdgeMemory are to be accessed both reading and writing (see next page), they are declared as InOut parameters.

Caution!
The declared formal parameters (Input, Output, InOut and Return) of a block are its interface to the "outside". That is, 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 of the modified block have to be updated or corrected in all calling blocks.

9.3.2. Editing a Parameter-assignable Block



Notes

It doesn‘t matter whether the names of the formal parameters are written with capital or small letters. The "#" character in front of the name is automatically inserted by the PG. The character is used to indicate that the parameter is a local operand that was defined in the variable (tag) declaration table of this block.

It is possible, that when you write the program in KOP or FUP, that the name is not completely displayed in one line. This depends on how you have customized the settings in the Program Editor:

Options→Settings→PLC programming→LAD/FBD→Operand field→Maximum width

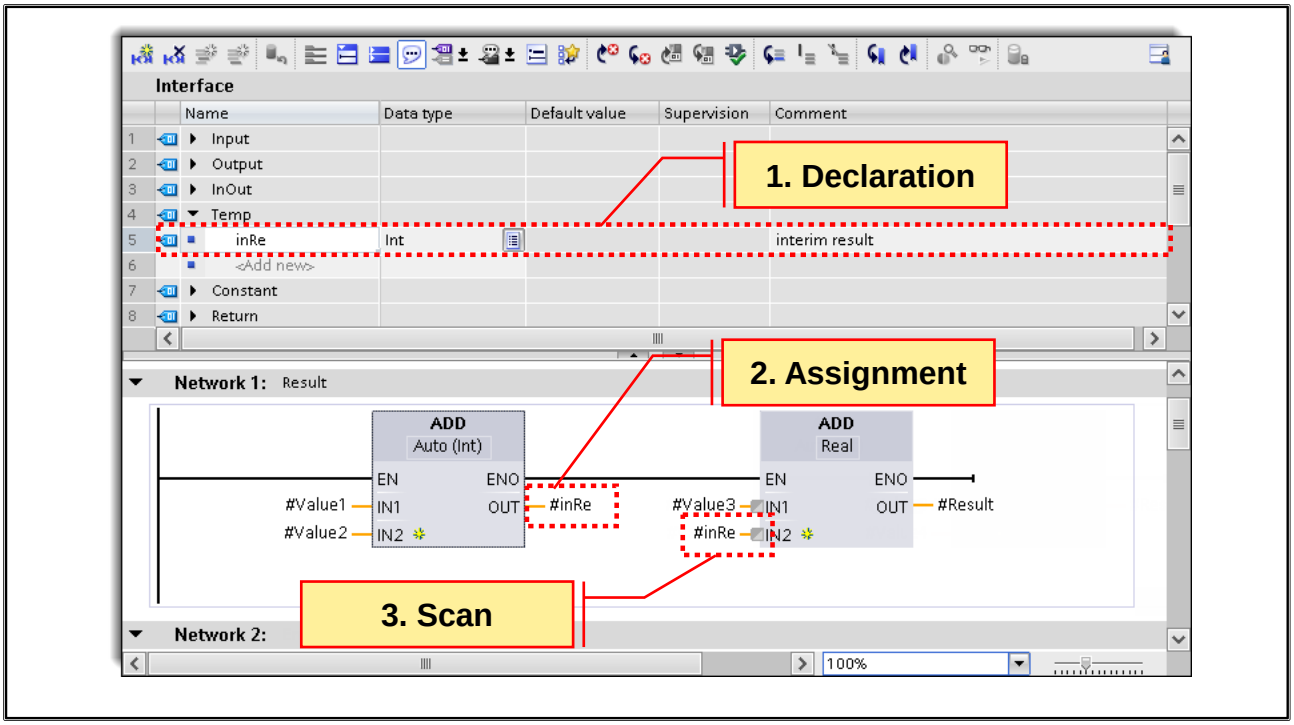
Symbols

1. If you use a symbolic name when you edit a block, the Editor first of all searches through the interface of the block. If the symbolic name is there, the symbol with # in front of it is accepted in the program as a local operand.
2. If a symbol cannot be found as a local operand, the Editor searches through the PLC tags for the global symbol. If the symbol is found there, the symbol is placed in quotation marks and is accepted in the program as a global operand.
3. If you specified the same symbolic name globally (in the PLC tags) as well as locally (in the variable (tag) declaration table) the Editor will always insert the local operand.
   If, however, you want to work with the global symbol, you must select the relevant operand when you make the entry, place the symbol name in quotation marks or change it later on.

Drag & Drop

Just as with global operands (for example, from the PLC tags) local operands can be dragged into the program part of the Editor from the block interface using drag & drop and placed in the desired position there.

9.4. Local, Temporary Variables



Declaration

The variables are also defined in the declaration part of the block. The name of the variable and the data type must be specified.

Access

With optimized blocks, all temporary variables are initialized with 0 at the beginning of block execution.

With not-optimized blocks, all temporary variables have an undefined value at the beginning of block execution. When working with temporary variables, you must therefore make sure that the variable is first of all assigned a defined value before it is scanned.

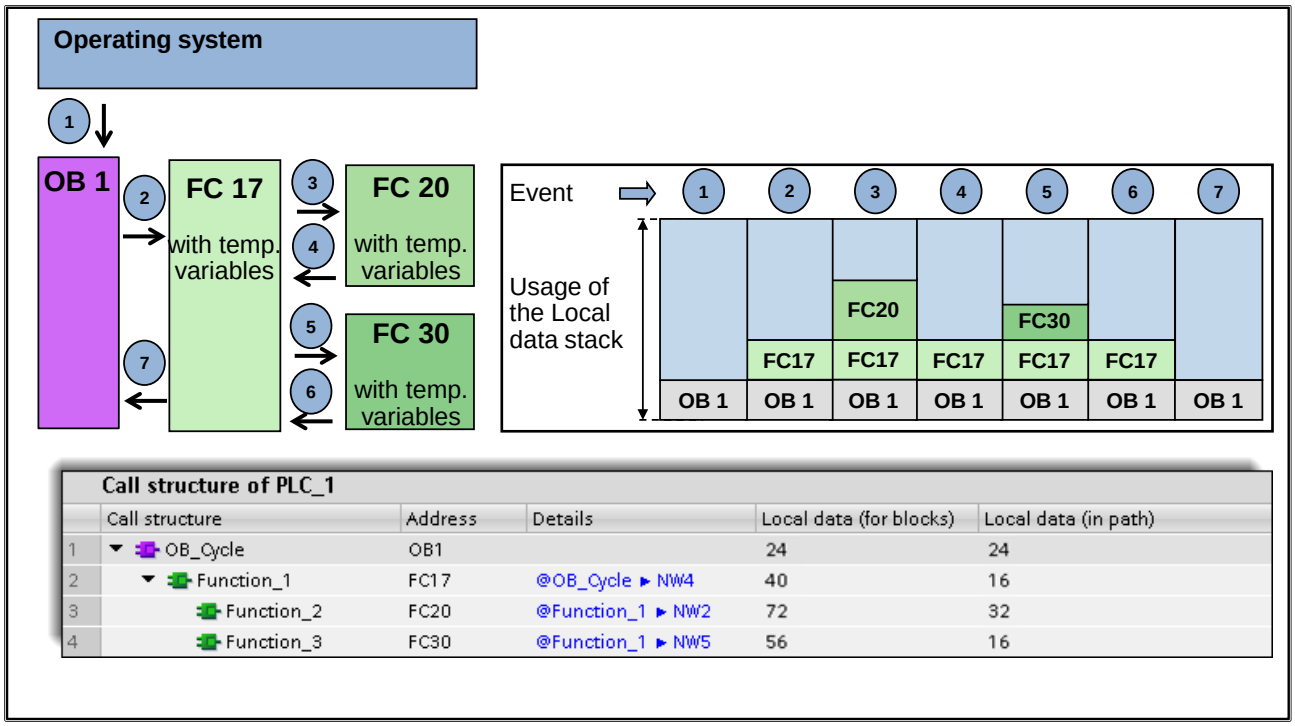
In the example, the result of the Addition is assigned to the temporary variable *#inRe* before it is then scanned during the Multiplication. (The arithmetic operations are dealt with in the chapter "Digital Operations".)

Note

Operands that begin with the # special character are local operands (parameters or local variables) that must be declared in the declaration part of the block. Local operands are only valid and usable in the block in which they were declared.

The Program Editor automatically inserts the # character.

9.4.1. Local Data Stack



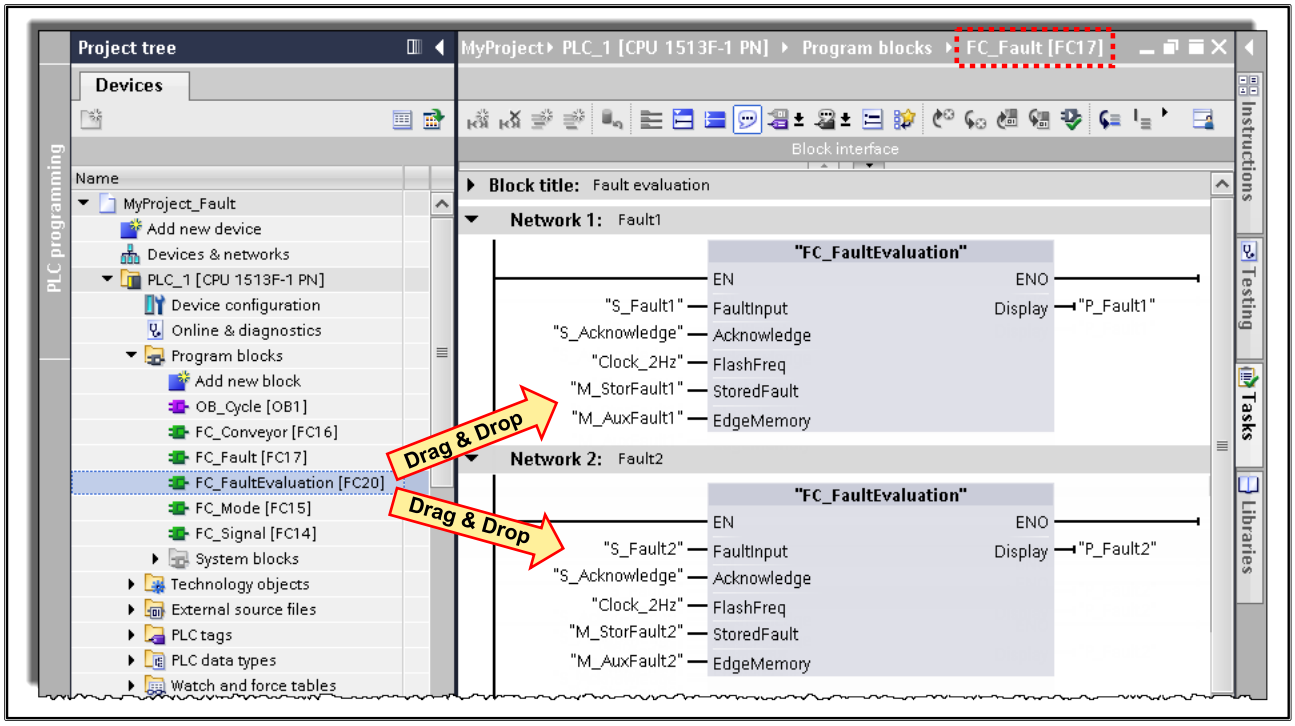
Total Usage of the Local Data Stack (L-Stack)

For every program execution level or priority class (such as, OB 1 with all blocks that are called in it), a separate local data stack is reserved. That is, a segment of defined size is reserved on the L stack of the CPU (allocation or reservation of memory space).

The local variables/operands of OB 1 as well as the local, temporary variables of the blocks (FCs and FBs) called in or by OB 1 are stored in this local data stack.

You can use the "*Reference Data*" tool to display the "*Program Structure*" to see to what extent an S7 program puts a burden on the local data stack. (The reference data and where they are displayed is dealt with in the chapter "Troubleshooting".)

9.5. Calling a Parameter-assignable Block



Block Call

A block can be called by dragging it from the "Program blocks" folder & dropping (inserting) it in the statement (code) part of the calling block.

Note

When a parameter-assignable function (FC) is called, an actual parameter must be passed for every formal parameter.

Exception:

In the graphic programming languages LAD and FBD, the assignment of the EN and ENO parameters, which are automatically added by the Editor, is optional.

Parameter Assignment

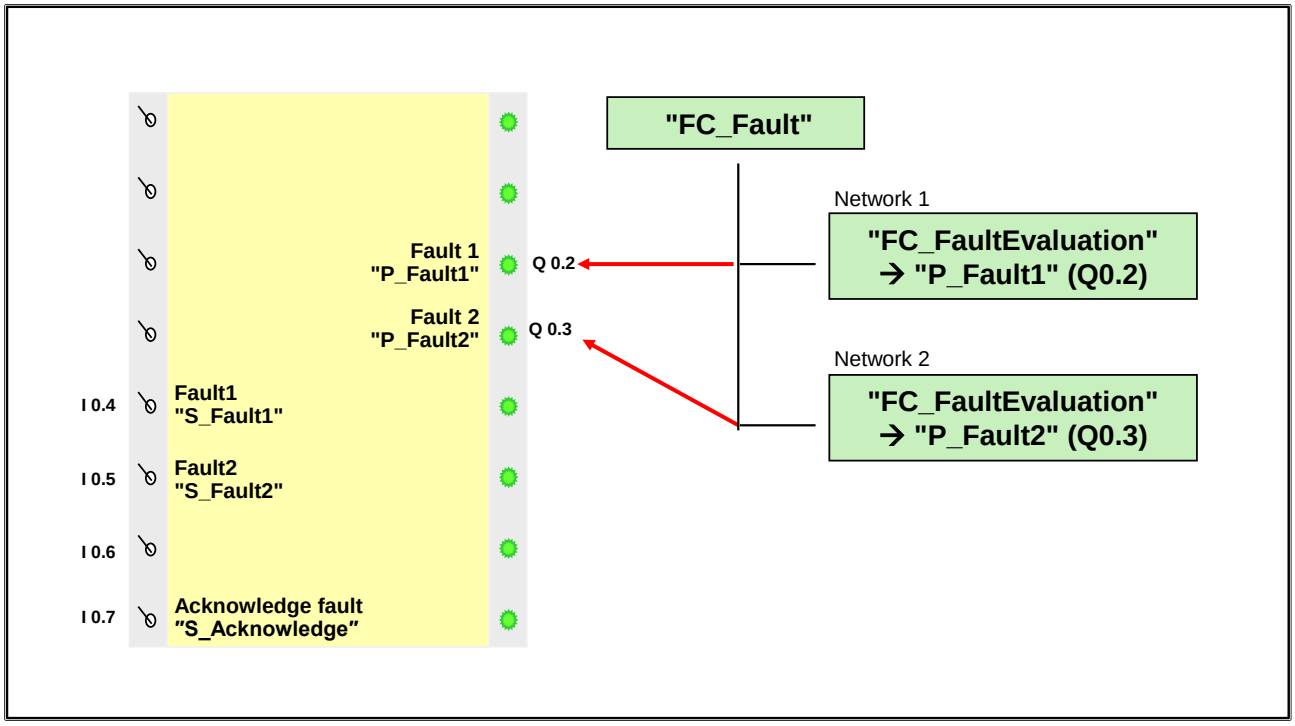
All global and local operands whose data type corresponds to the formal parameters of the called block can be passed as actual parameters.

The actual parameters can be passed with an absolute address or with a symbolic name - as declared in the PLC tags or in the declaration part of the calling block.

Passing On of Parameters

Basically, a "passing on of parameters" is also possible. That is, formal parameters of the calling block are passed on as actual parameters to the called block. For parameters of complex data types (see chapter "Data Blocks") this is however only possible with limitations.

9.6. Task Description: Fault Evaluation by means of a Function (FC)



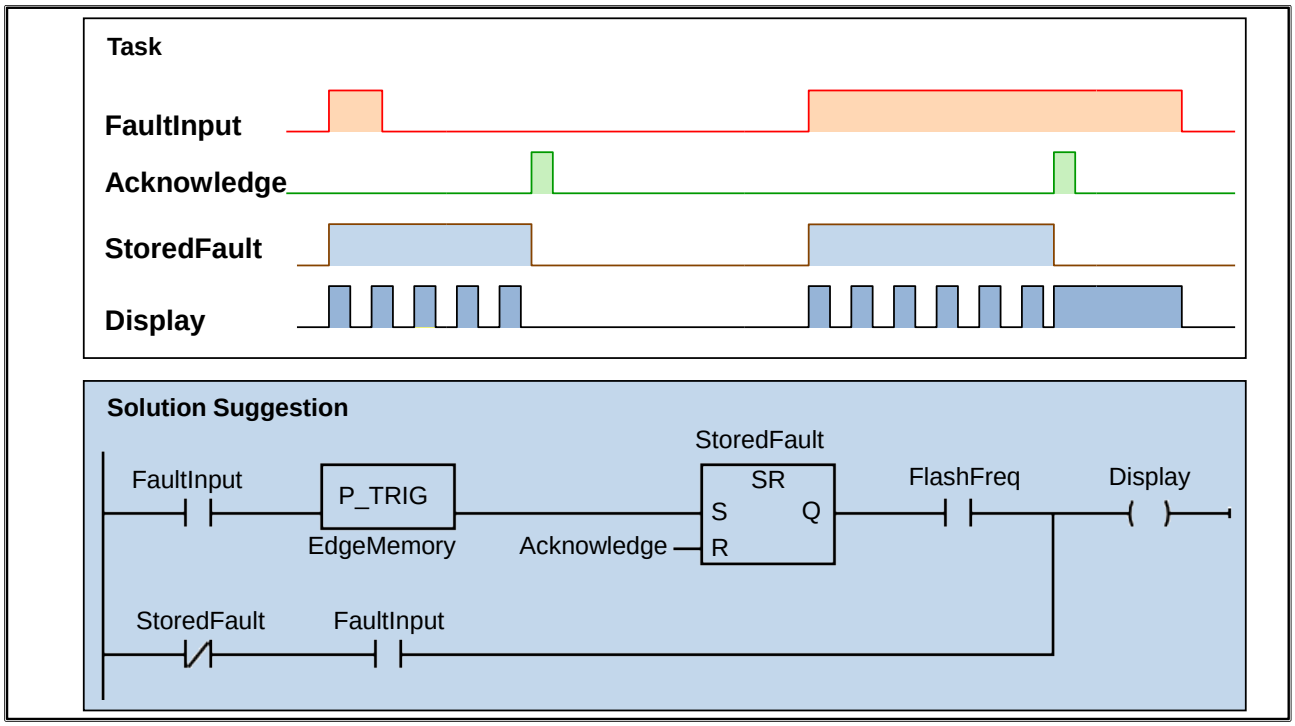
Task Description

If a fault is triggered at the inputs I 0.4 "S_Fault1" or I 0.5 "S_Fault2", the associated LED Q0.2 "P_Fault1" or Q0.3 "P_Fault2" begins to flash.

The input I 0.7 "S_Acknowledge" is a group acknowledgement for all faults. If the fault still exists after acknowledgement, the LED changes to constant light; if the fault no longer exists, the LED goes dark.

First, an "FC_Fault" is to be created. Then, the required function is to be programmed in the parameter-assignable "FC_FaultEvaluation" which is to be called twice in "FC_Fault" for the evaluation of the two faults.

9.6.1. Fault Evaluation



Task

Faults that occur are to be displayed by an indicator light on the operator console. When there is a signal change from 0 → 1 at the input, the output shows a 2Hz flashing light.

After the fault is acknowledged but still exists, the output (light) switches to a constant light. When the fault no longer exists, the light at the output goes dark.

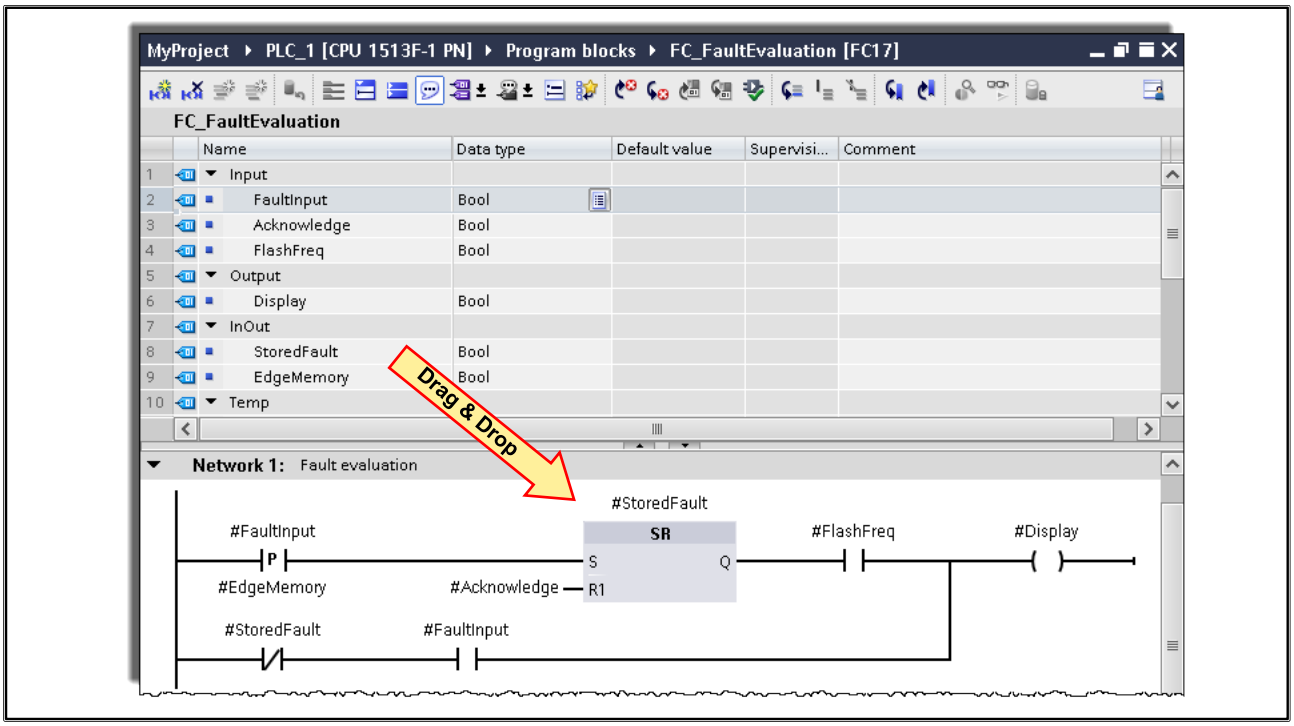
Solution Suggestion

An edge evaluation of the fault (**FaultInput**) is required since the message buffer (**StoredFault**) would otherwise immediately be set again after an acknowledgement (**Acknowledge**) and a still existing fault, thus making the display (**Display**) flash once more.

When an acknowledgement (**Acknowledge**) has not yet occurred, that is, the message buffer (**StoredFault**) still exists, the upper AND logic operation with the linked flash frequency (**FlashFreq**) causes the display (**Display**) to flash.

When acknowledgement has already occurred (**Acknowledge**) and therefore the message buffer (**StoredFault**) no longer exists, but the fault input (**FaultInput**) still exists, the lower AND logic operation causes a constant light at the display (**Display**).

9.6.2. Exercise 1: Creating the "FC_FaultEvaluation" Function



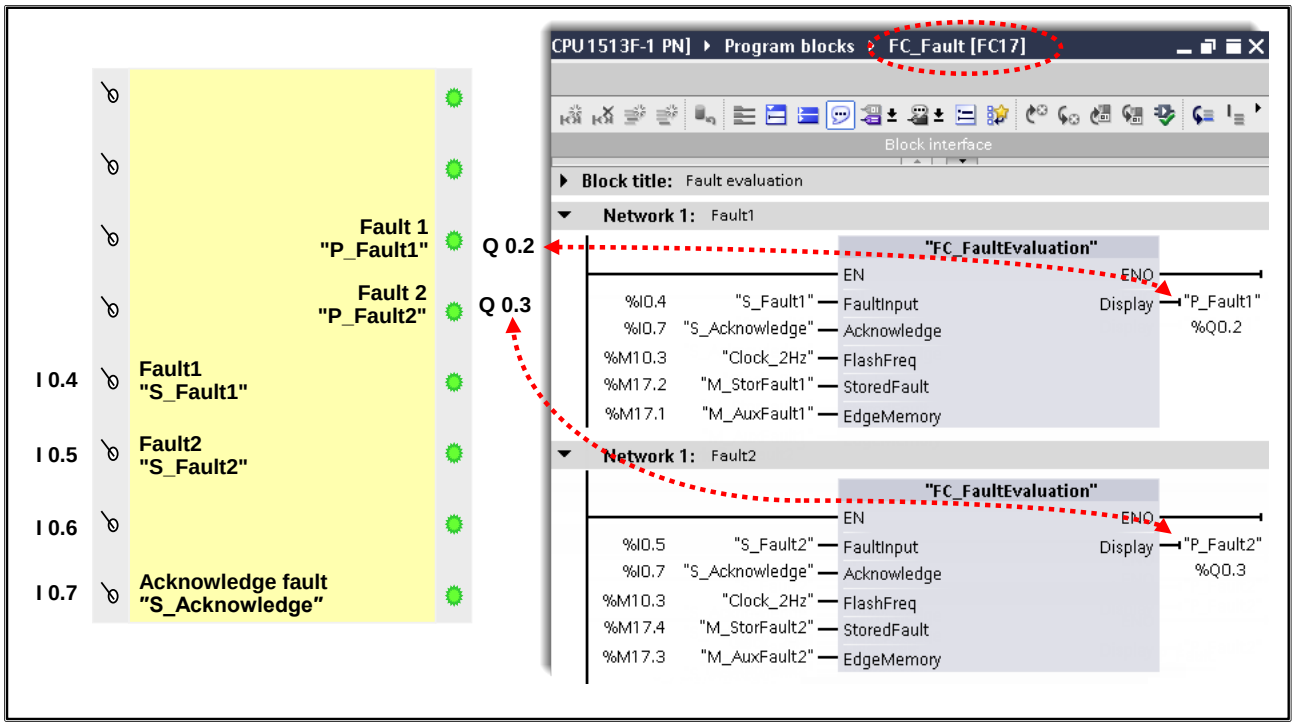
Task

You are to create the program for the fault evaluation in the parameter-assignable "FC_FaultEvaluation".

What to Do

1. Insert the "FC_FaultEvaluation" block in the "Program blocks" folder.
2. Declare the formal parameters as shown in the picture.
3. Create the program as shown in the picture.
4. Save the block.

9.6.3. Exercise 2: Calling and Parameterizing "FC_FaultEvaluation"



Task

You are to create the new block "FC_Fault" which will process the fault handling and fault evaluation in later exercises.

In the new blocks, 2 faults from the process (signals of the two simulator switches) are to be evaluated. For this, the previously programmed "FC_FaultEvaluation" must be called twice.

What to Do

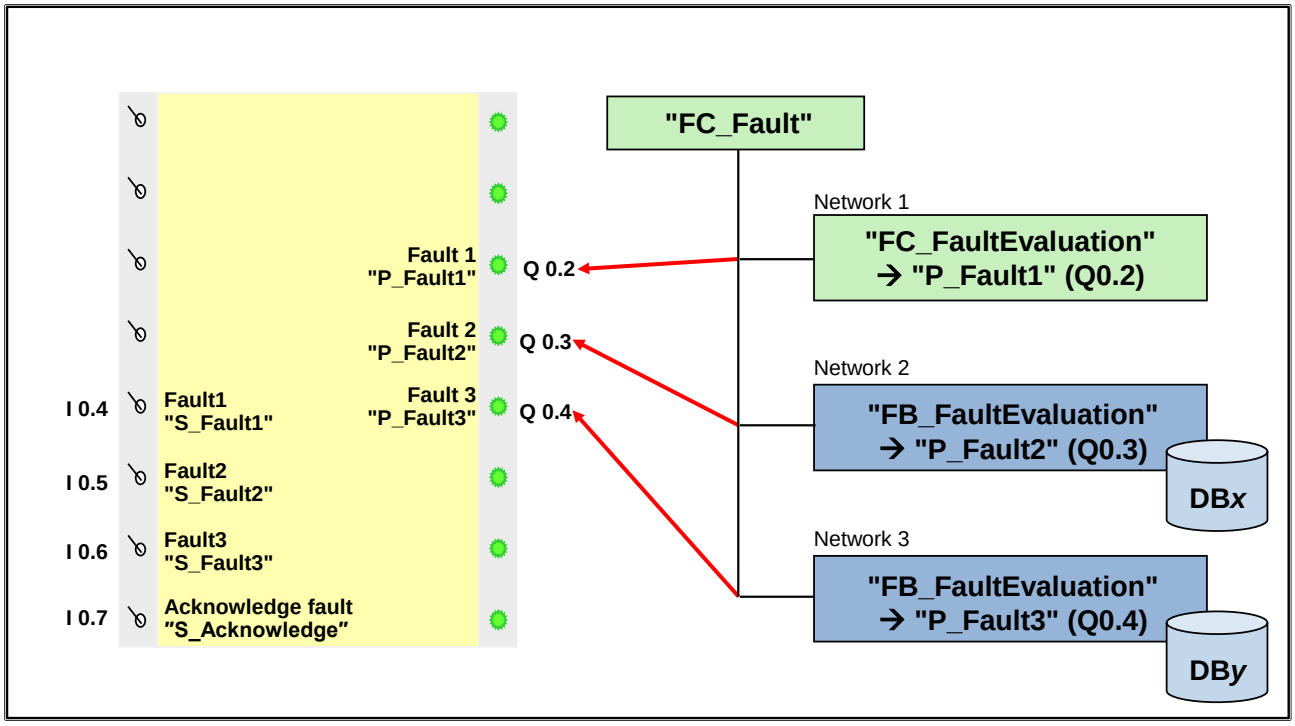
1. Create the new "FC_Fault" block.
2. In "FC_Fault", program the two calls of the previously created "FC_FaultEvaluation" block as shown in the picture.
3. Call "FC_Fault" in "OB_Cycle".
4. Save the change and transfer the program into the CPU.
5. Check your program to see whether it fulfills the described functions for fault evaluation.

Note

The MB 10 memory byte was already parameterized as a clock memory byte in the device configuration.

The "Clock_2Hz" (M10.3) memory bit has a flashing frequency of 2Hz and was already created as a PLC tag.

9.7. Task Description: Fault Evaluation by means of a Function Block (FB)

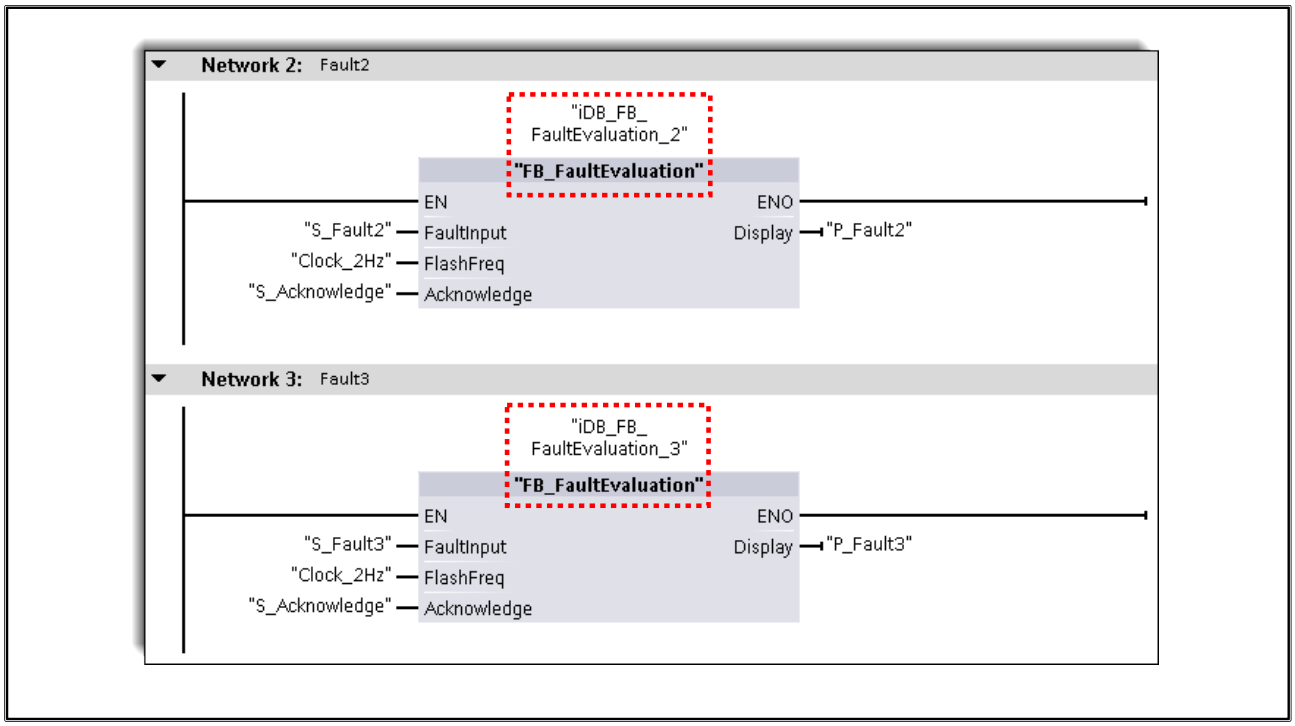


Task Description

The previously described fault evaluation is now to be implemented with an FB instead of an FC. This offers the advantage that for the internally required edge evaluation of the fault and as stored fault, the FB doesn't have to be passed any global operands from outside since local, static variables can be used.

Fault 1 is to continue to be evaluated by the already existing "FC_FaultEvaluation". The evaluation of Fault 2 and 3 is to be carried out by the "FB_FaultEvaluation" which is now to be created.

9.7.1. Instantiating Function Blocks



Special Features

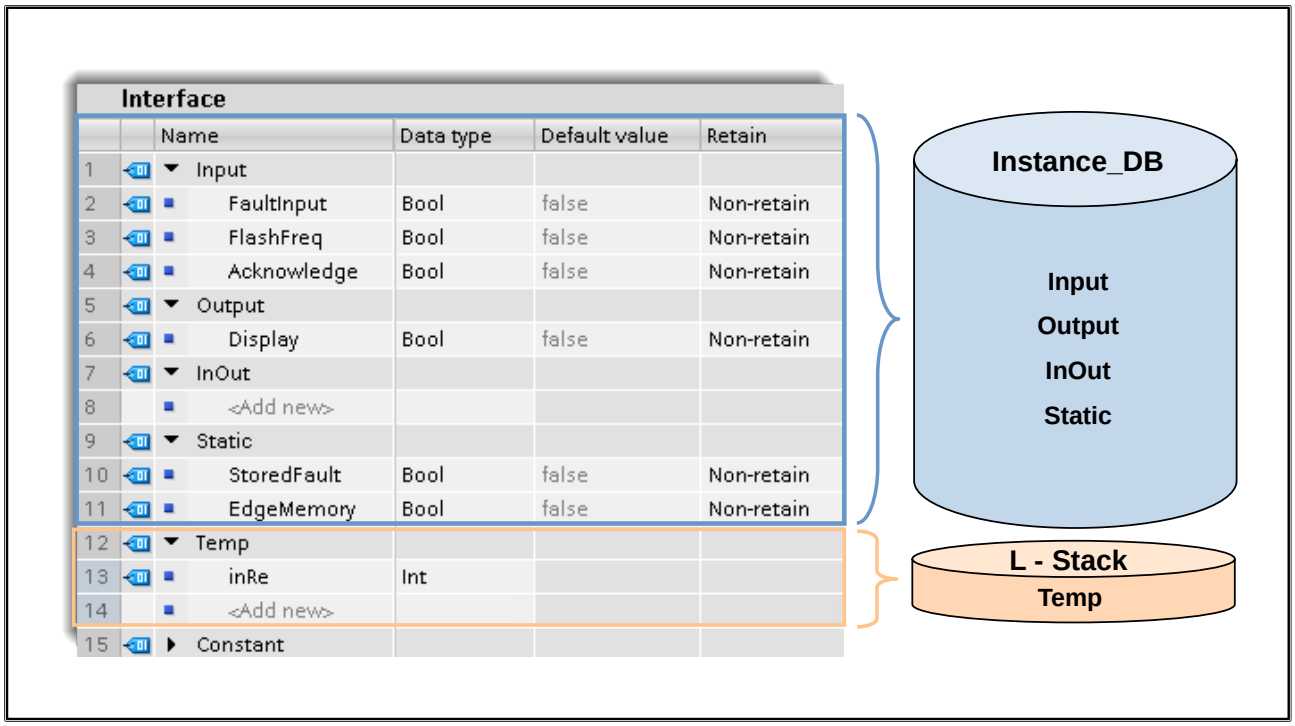
Unlike functions (FCs), function blocks (FBs) have a (recall) memory. That means that a local data block is assigned to the function block. This data block is known as an instance data block. When you call an FB, you must also specify the Instance-DB which is then automatically used as an instance for this FB call.

An instance DB is used to save static variables, among other things. These local variables can only be used in the FB, in whose declaration table they were declared. When the block is exited, they are retained.

FB Advantages

- For the FC programming, the user must search for free memory areas and maintain them himself. The static variables of an FB, on the other hand, are maintained by the STEP 7 software.
- The known danger of memory bit double assignments in FC programming is avoided with the use of static variables.
- Instead of the InOut formal parameters "StoredFault" and "EdgeMemory" of the "FC_FaultEvaluation", static variables are used in the "FB_FaultEvaluation". This makes the block call simpler since the two formal parameters are dropped.

9.7.2. FB - Declaration Section



Parameters

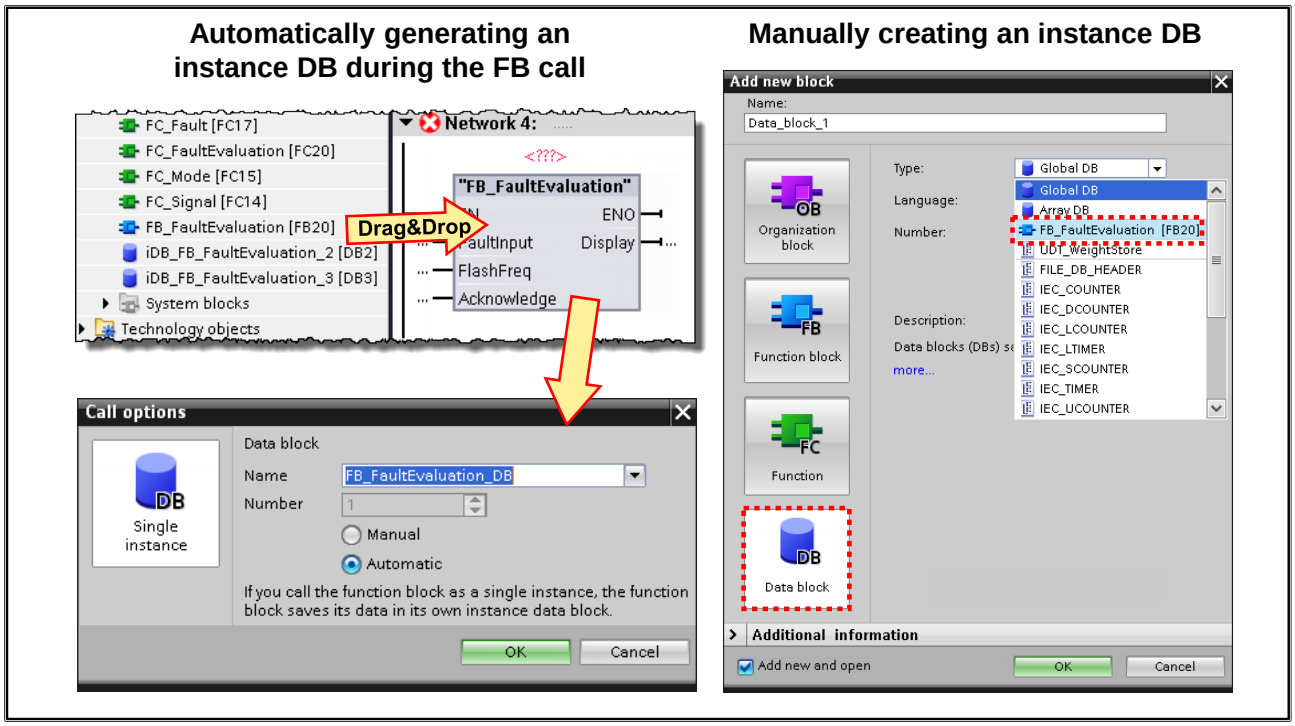
When the function block is called, the values of the actual parameters are stored in the instance data block. If no actual parameter was assigned to one of these formal parameters in a block call, then the last value stored in the instance DB for this parameter is used in the program execution. One exception is InOut parameters whose data types are not elementary. These must be assigned since the values of the actual parameters are not stored in the instance DB but rather the information about the storage location of the actual parameter.

Just as for a function, different actual parameters can be passed for each FB call. When the function block is exited, the data is retained in the instance data block.

Static Variables

Unlike functions, function blocks additionally have "static variables" (Static). These variables form the memory of the FB. They are not stored in the L-Stack but also in their own instance data block.

9.7.3. Generating Instance Data Blocks



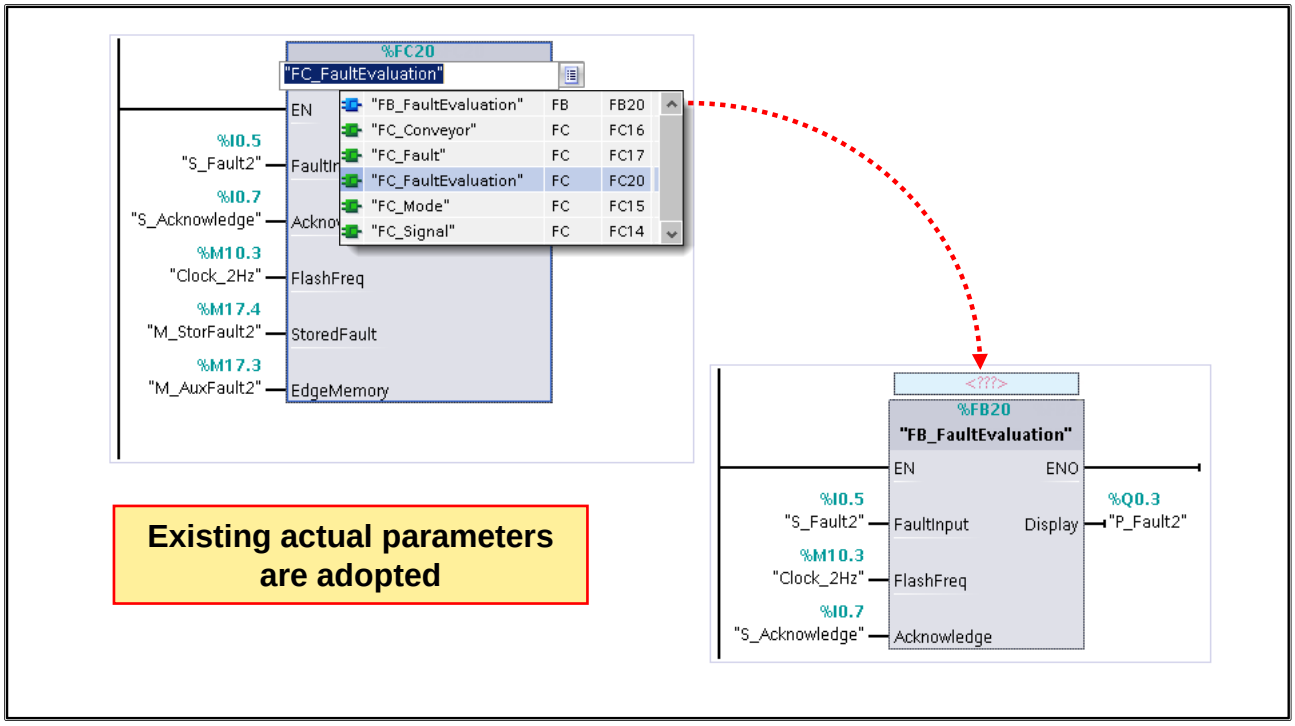
Generating

There are two ways of generating an instance data block:

- Create a new block (data block) and select "Function block XY" as Type.
- For an FB call, the user specifies with which instance DB the FB is to work.
  A dialog automatically opens in which the symbolic name and, if desired, a manual number of the instance DB can be preset.
  Already existing instance data blocks can also be selected here.

Caution!
If you modify the FB (by adding additional parameters or static variables), you must then also generate the instance DB again.

9.8. Changing the Block Call

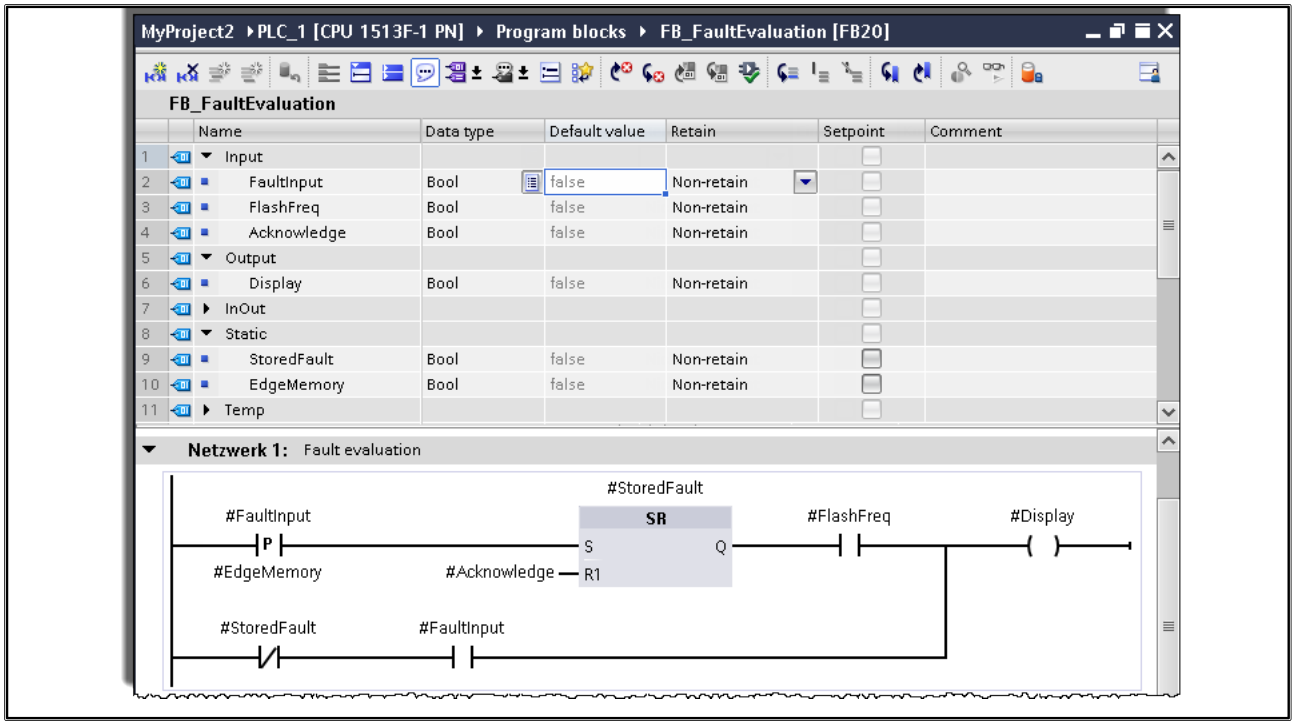


In order to replace the call of a block with another block call, a selection list of all FCs and FBs can be opened at the calling point by double-clicking on the name of the already called block.

Advantage:

If both blocks have the same formal parameters, then they retain their actual parameters and all formal parameters do not have to be supplied with new actual parameters.

9.9. Exercise 3: Creating the Function Block "FB_FaultEvaluation"



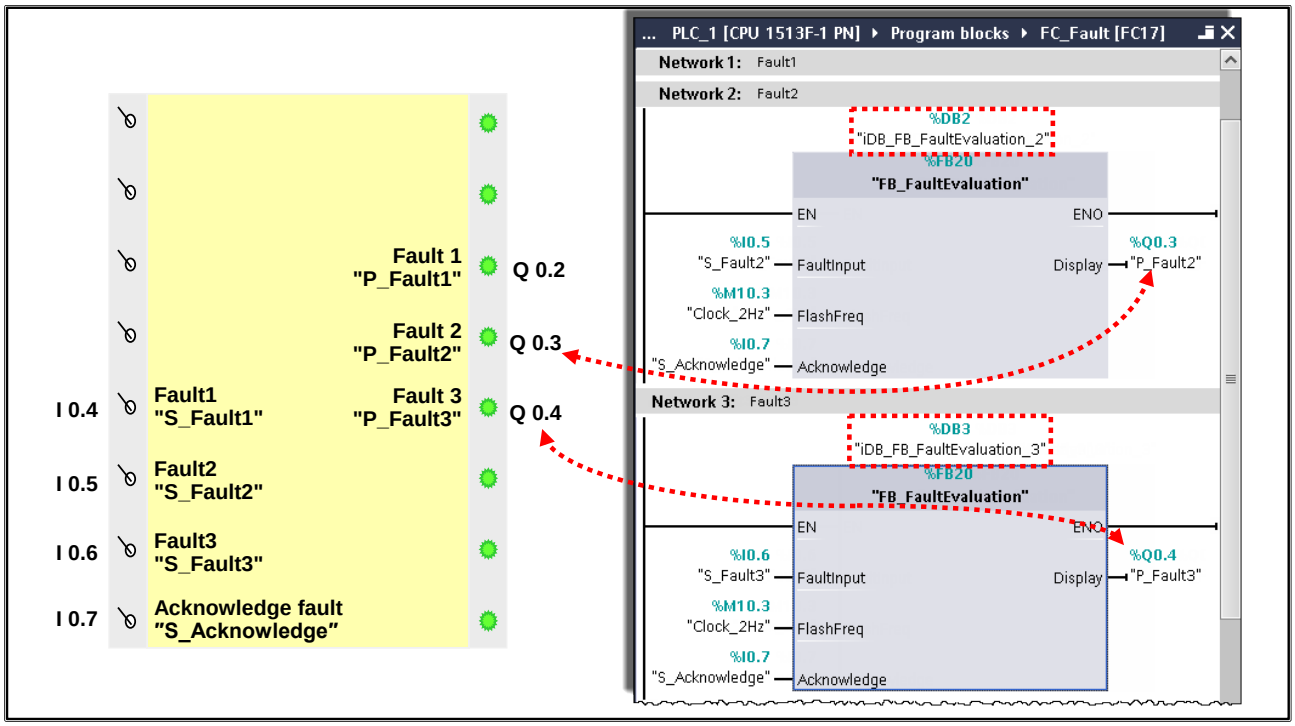
Task

You are to create the new "FB_FaultEvaluation" block for the subsequent evaluation of Fault 2 and 3.

What to Do

1. Insert the new "FB_FaultEvaluation" block.
2. Declare the formal parameters and the static variables of the block as shown in the picture. For this, you can copy the required variables from the already programmed "FC_FaultEvaluation".
3. Program "FB_FaultEvaluation". For this, you can copy the required program parts from the already programmed "FC_FaultEvaluation".
4. Save the block and download it into the CPU.

9.9.1. Exercise 4: Calling and Parameterizing "FB_FaultEvaluation"



Task

The evaluation of the old Fault 2 (programmed up until now through the call of "FC_FaultEvaluation") and the evaluation of the new Fault 3 is to be implemented with the newly created "FB_FaultEvaluation".

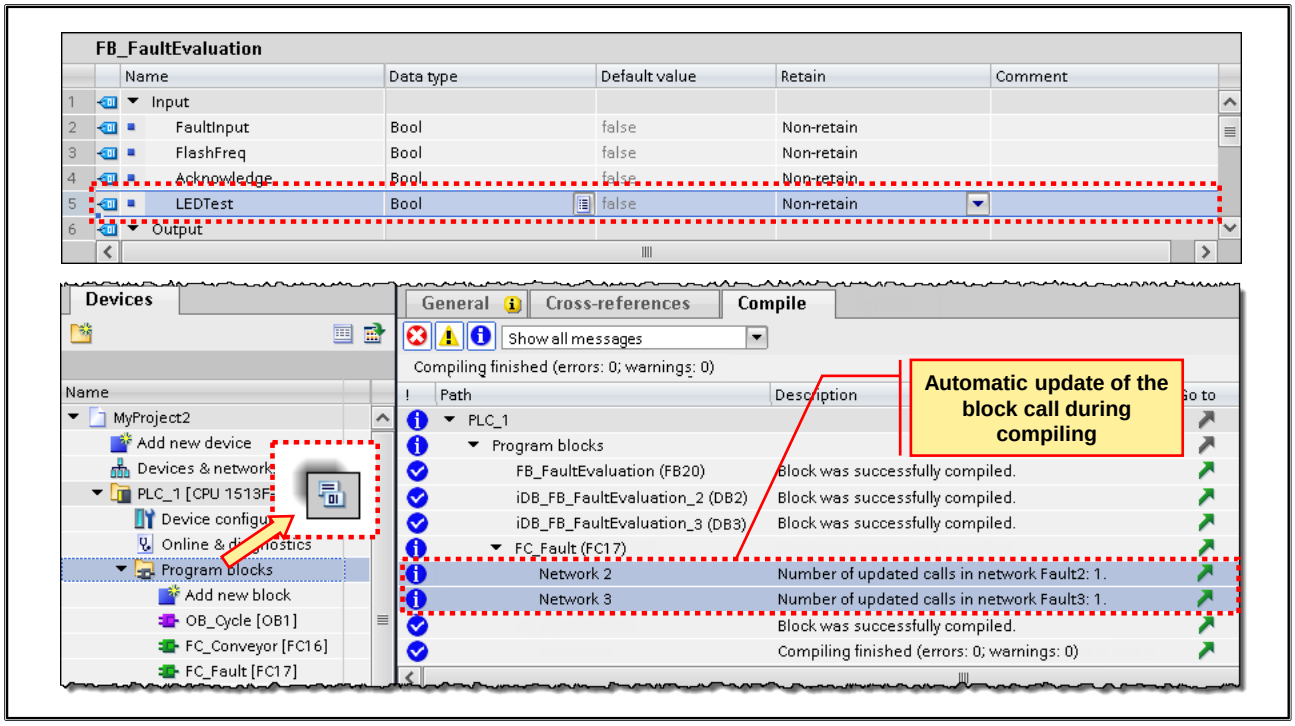
For this, the parameter-assignable block "FB_FaultEvaluation" must be called twice in "FC_Fault", each time with a different instance data block.

What to Do

1. In "FC_Fault", replace the second call of "FC_FaultEvaluation" with the call of "FB_FaultEvaluation".
2. Generate the instance data block "iDB_FB_FaultEvaluation_2" and specify it as the instance for the programmed call of "FB_FaultEvaluation".

   You can generate a new instance via the context menu item of the call (right-click on the Block call > "Create instance") or you create a new block (data block of the type "FB_FaultEvaluation") and insert it using drag & drop.
3. Program the second call of "FB_FaultEvaluation" - as shown in the picture - in a new network and let the Editor generate the instance "iDB_FB_FaultEvaluation_3".
4. Save the modified "FC_Fault".
5. Download the entire program into the CPU and check the program function.

9.10. Adding Block Parameters Later On



Problem

If you have to adjust or supplement the interfaces or the code of individual blocks during or after program creation, it can lead to time stamp conflicts. Time stamp conflicts can, in turn, lead to block inconsistencies between calling and called blocks or reference blocks and thus to a high degree of correction effort.

If block parameters are **added** later on to a block already called in the program, you also have to update the calls of the block in other blocks.

- Automatic Update

  Time stamp conflicts are also detected when the entire user program is compiled and in case of added parameters, affected block calls are automatically updated.

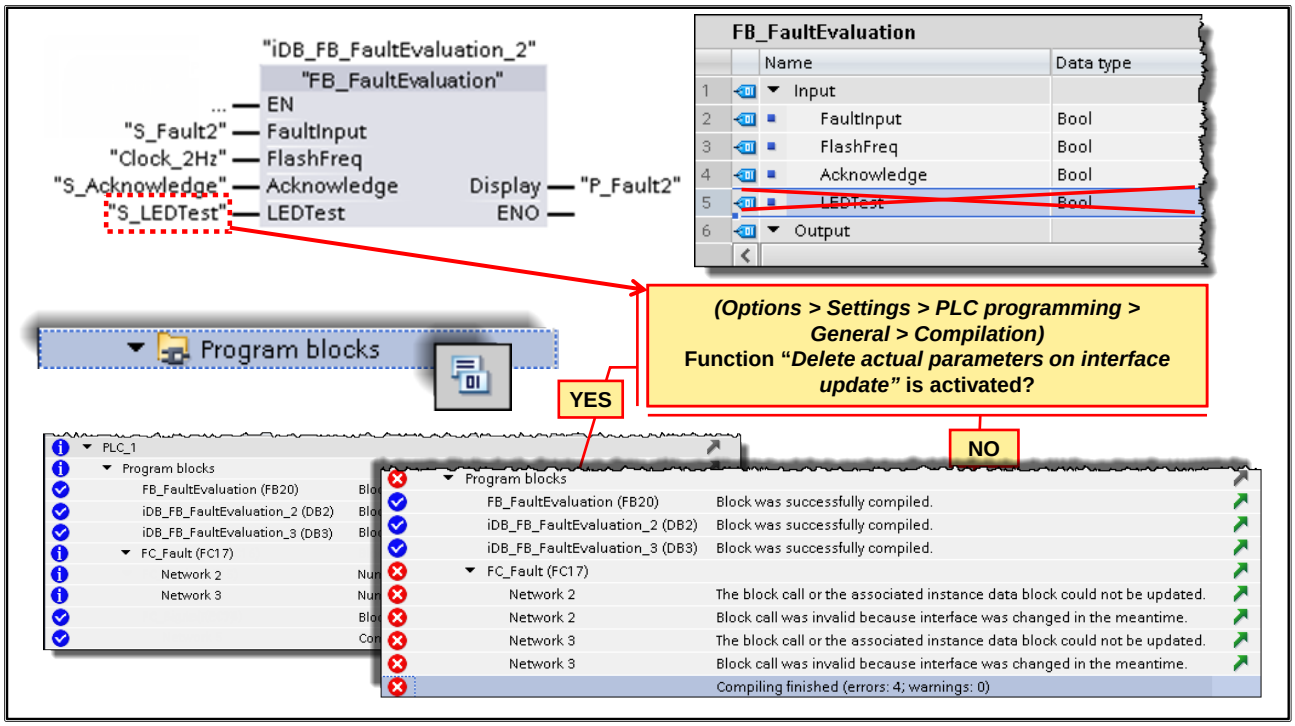
  For functions, the added formal parameter must still be assigned before downloading into the CPU, since this is a "Must Assign".

  For function blocks, the default value from the associated instance DB is used when the formal parameter is not assigned ("Can Assign").

- Manual Update

  See 9.10.2 Manually Updating a Block Call

9.10.1. Removing Block Parameters Later On



If block parameters are **deleted (removed)** later on from a block already called in the program, you also have to update the calls of the block in the calling blocks.

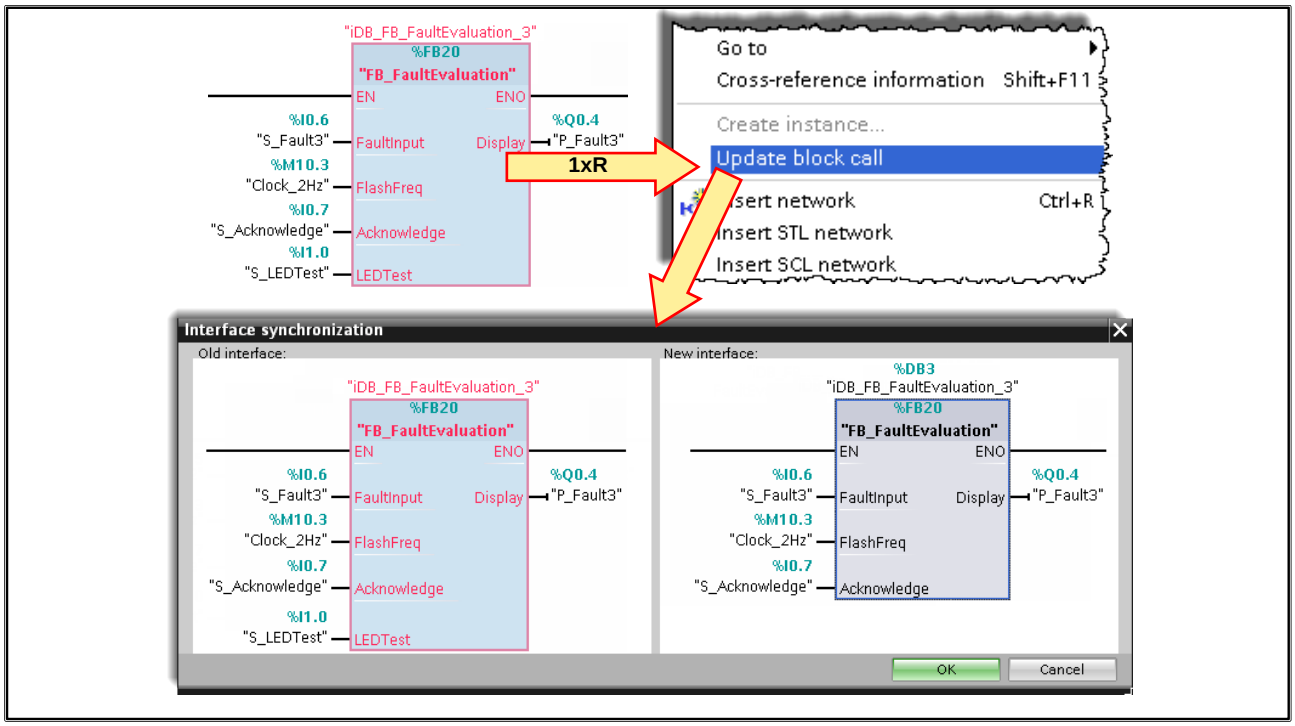
- Automatic Update

  Attention/Caution: If the deleted formal parameters have already been assigned with actual parameters, then this automatic update only occurs if the function "Delete actual parameters on interface update" is activated under Options > Settings > PLC-programming > General > Compilation.

- Manual Update

  See 9.10.2 Manually Updating a Block Call

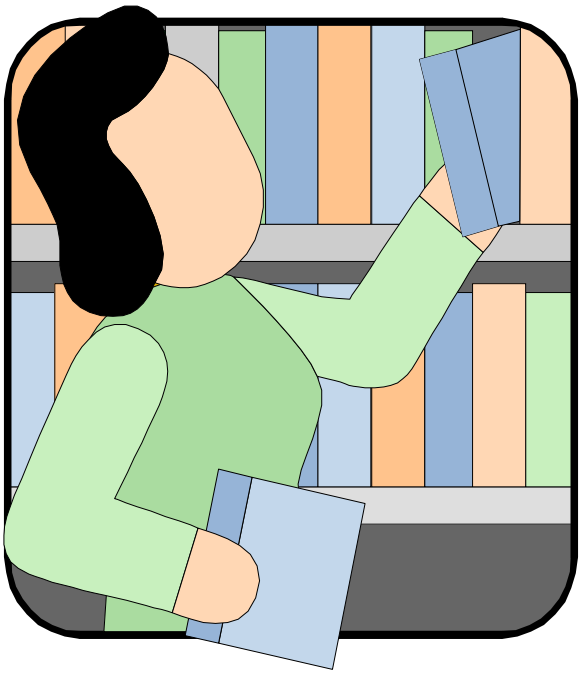
9.10.2. Manually Updating a Block Call



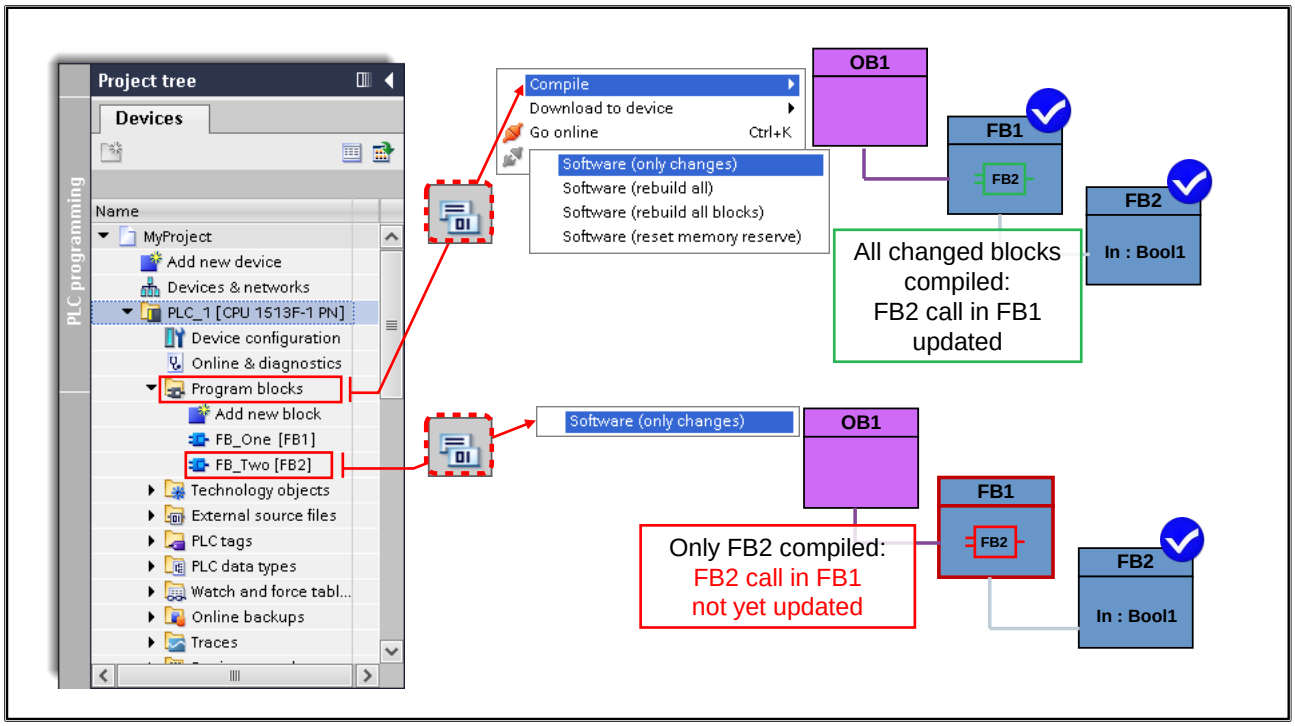
Manual Update

In the open, calling block, the inconsistent calls of a block are highlighted in red. By right-clicking the inconsistent call, the function "Update" can be selected in the context menu. A window then appears in which the old (faulty) and the new block call (in the picture without the parameter "LEDTest") are displayed. For function blocks, the associated instance DB is subsequently regenerated.

9.11. Additional Information



9.11.1. Compiling Individual / All Changed Blocks



Compile → Software (Only Changes):

If one individual block is selected in the Project tree or if a compilation is triggered through the button shown to the left when a block is open, only this single block is compiled (if it was changed). The disadvantage of compiling one individual block is that interface conflicts in the calling blocks caused by interface changes are not corrected.

If several blocks are selected or a block group, only those blocks modified since the last compilation are compiled (delta compilation).

If the "Program blocks" folder is selected, the delta compilation for the entire program is carried out.

Compile → Software (Rebuild All Blocks)/ or Rebuild All):

All, even those blocks not modified since the last compilation, are compiled.

Compile → Reset Memory Reserve:

Note: What a memory reserve is, is dealt with in Programming 2.
With the compilation process, the memory reserve of data blocks is also reset that is, that variables that were created later on in the course of data block expansions are removed from the memory reserve and integrated in the regular part of the data block.

Compilation Results

The status of the compilation is hierarchically displayed in the Inspector window "Info -> Compile". If errors occurred during compilation, you can jump directly to the error location by double-clicking on the error entry.

9.11.2. Global and Local Tags

	Global Tags	Local Tags
Validity range	• Valid throughout the entire CPU, i.e. all blocks have access • The name of the tag must be unique within the entire CPU	• Are only valid in the block in which they have been declared (defined) • The name of the tag must be unique within the block
Operands	• Inputs • Outputs • Memory bits • Tags in data blocks • SIMATIC Timers / Counters	• Temporary variables (in all code (logic) blocks) • Static variables (only in function blocks)
Location of declaration	• PLC tag table • Global data blocks (Chap. 11)	• Declaration part of the block
Presentation	• Presented in quotation marks Example: "Max"	• Presented preceded by # Example: #Max

Validity Range of Tags

Tags that are declared in the PLC tag table or in a global data block can be addressed by all CPU program blocks. For that reason, these tags are called global tags.

Tags and parameters that are declared in the declaration part of a code (logic) block are local operands; they can only be used in the statement part of the same block.