SIEMENS

Information and Training
Automation and Drives

SIMATIC S7

Programming 2

Course ST-7PRO2

AL: N ECCN: N

Export Regulations

AL Number of European resp. German export list.
Goods with labels not equal to "N" are subject to export authorization.

ECCN Number of US export list (Export Control Classification Number). Goods with labels not equal to "N" are subject to re-export authorization for export to certain countries.

Indication Goods labeled with "AL not equal to N" (here: technical documentations) are subject to European or German export authorization when being exported out of the EU.
Goods labeled with "ECCN equal to N" (here: technical documentations) are subject to US re-export authorization.
Even without a label, or with label "AL:N" or "ECCN:N", authorization may be required due to the final whereabouts and purpose for which the goods are to be used.
Decisive are the export labels stated on order acknowledgements, delivery notes and invoices.

Name: _____

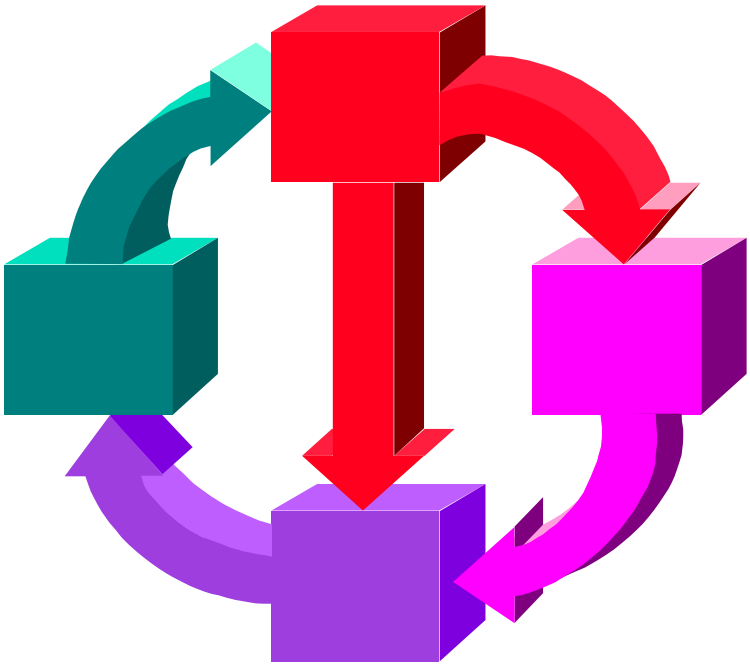
Course: from _____ to _____

Instructor:

Infoline Tel: 01805 23 56 11
Fax: 01805 23 56 12
Internet: http://www.ad.siemens.de/training

ID-No.: E80850-K07-X-A5-3
Version A5.3 (for STEP7 Version 5.1x)

Status Bit-Dependent Instructions

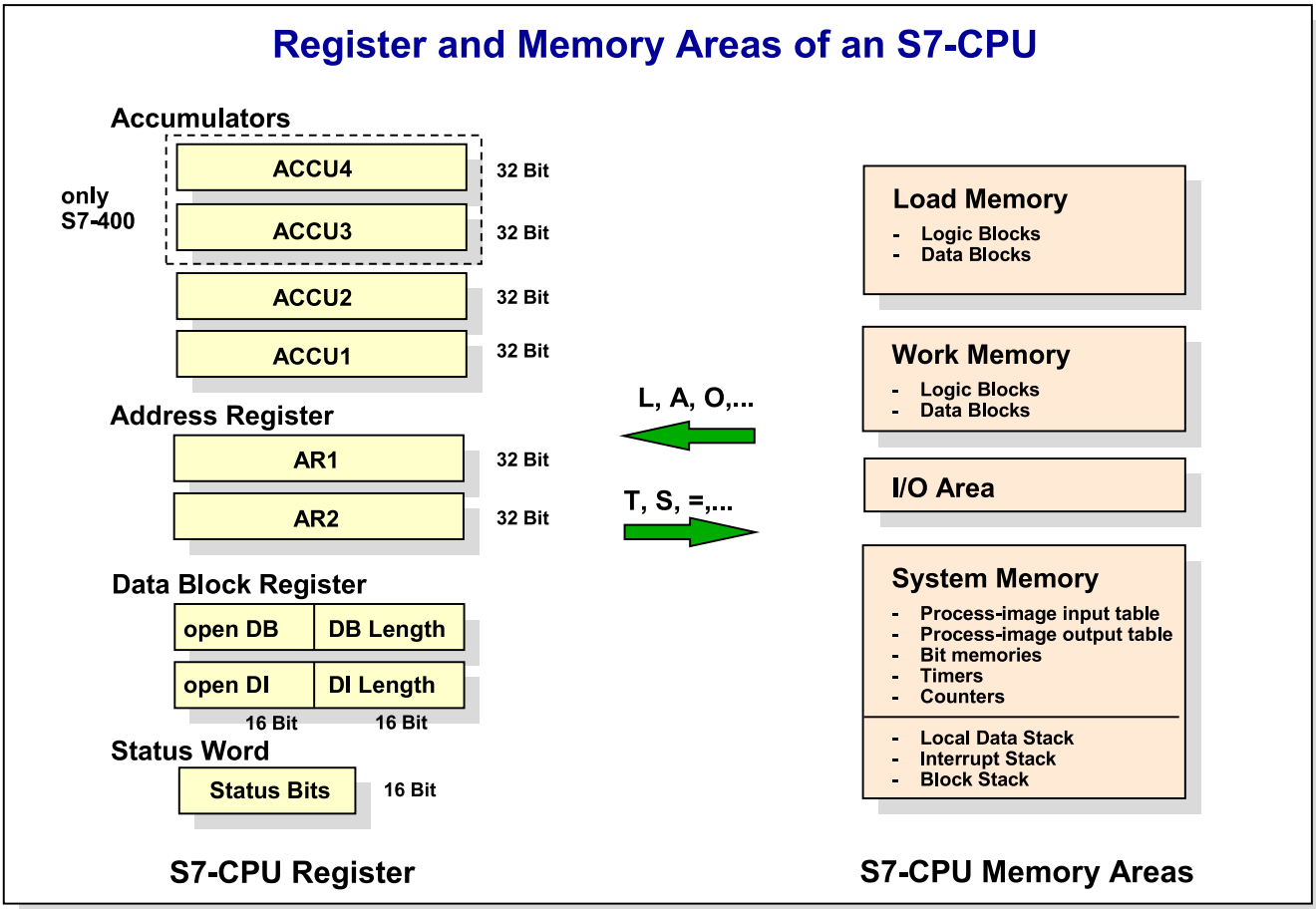


SIMATIC S7

Date: 30.06.2011
File: PRO2_01E.1

SITRAIN Training for Automation and Drives

Contents **Page**



Date: 30.06.2011
File: PRO2_01E.2

CPU Register

The registers of a CPU are used to address or to process data. The data can, with the help of relevant commands (L, T,...), be exchanged between the CPU memory areas and the registers:

- Accumulators: Two (with S7-300) or four (S7-400) accumulators are used for arithmetic, comparison or other byte, word and double word instructions.
- Address register: Two address registers are used as pointers for the register indirect addressing of the memory.
- Data block register: The data block registers contain the numbers of the opened (active) data blocks. Thus it is possible that two DBs are open simultaneously; one DB with the help of the DB register, the other as instance DB with the DI register.

  When a DB is opened, its length (in bytes) is automatically loaded in the associated DB length register.
- Status Word: Contains various bits, that reflect the result or the status of individual instructions within the program execution.

Memory Areas

The memory of the S7-CPU can be divided into four areas:

- The load memory is used to store the user program without symbolic address assignment or comment. The load memory can be RAM or FLASH EPROM memory.
- The work memory (integrated RAM) is used to store the relevant portion of the S7 program needed for program execution. Program execution takes place exclusively from the work memory.
- The I/O area permits a direct access of the inputs and outputs of the connected signal modules.
- The system memory (RAM) contains areas, such as the process-image input and output tables, bit memories, timers and counters. In addition, it contains the local data stack, block stack and interrupt stack.

Status Word Structure

Meaning of the bits in the status word

Bit	Assignment	Value	Meaning
0	/FC	2^0	First check bit
1	RLO	2^1	Result of logic operation
2	STA	2^2	Status
3	OR	2^3	Or
4	OS	2^4	Stored overflow
5	OV	2^5	Overflow
6	CC 0	2^6	Condition code
7	CC 1	2^7	Condition code
8	BR	2^8	Binary result
9...15	Unassigned	2^9 ..2^{10}	

Status Word

The individual bits of the status word give information about the result or the status of instructions as well as errors that have arisen.

You can integrate the status bit's signal state directly into your program using binary logic operations and thus control the program flow.

First Check

The status word's bit 0 is called the first check bit. The signal state "0" in the /FC bit indicates that with the next logic instruction, a new logic string starts in your program. The diagonal stroke in front of the abbreviation FC indicates that the /FC bit is negated.

Result of Logic Operation

The status word's bit 1 is the RLO bit (RLO= "Result of Logic Operation"). It is used as temporary memory in binary logic operations.
An instruction in a string of logic instructions checks, for example, the signal state of a contact and gates the result of check (status bit) according to the rules of Boolean logic with the RLO bit. The result of logic operation is in turn stored in the RLO bit.

Status Bit

The status bit (bit 2) saves the value of a bit that is addressed. The status bit always shows, for scanning (A, AN, O,...) or write instructions (=, S, R,), the status of the bit that is addressed (for write instructions, the status of the address after execution of the instruction).

OR Bit

The OR bit is required when you perform an AND before OR logic operation with the instruction O. The OR bit indicates that a previously performed AND logic operation has delivered the value "1", whereby the result of the OR logic operation is already determined as being "1".

OV Bit

The OV bit (overflow) displays an error in a math instruction or a comparison instruction. The bit is set according to the result of the performed math or comparison instruction.

Checking the Status Bits

Checking in STL

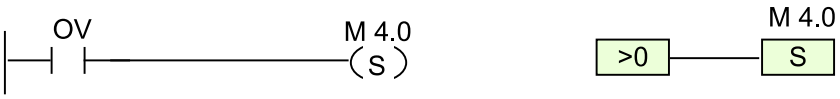
- A OV Scan overflow display
- A OS Scan stored overflow
- A BR Scan BR flag

Checking the Condition Code (CC0, CC1)

- A == 0 Result equal to 0
- A > 0 Result greater than 0
- A <> 0 Result not equal to 0
- A =< 0 Result less than or equal to 0

  etc.
- A UO Instruction is unordered

Checking in LAD and in FBD



SIMATIC S7

Date: 30.06.2011
File: PRO2_01E.4

SITRAIN Training for Automation and Drives

OS Bit

The OS bit (overflow stored) is set together with the OV bit. The OS bit remains set after a renewed math instruction, that is, it is not changed by the result of the next math instruction.

Thus you have the opportunity, even in a later location in your program, to evaluate a number area overflow or an instruction with an invalid REAL number.

The OS bit is only reset with the commands: JOS (Jump, if OS = 1), block calls and block end.

CC1 and CC0

The bits CC1 and CC0 (condition codes) give information on the following results:

- result of a math instruction
- or comparison instruction.
- a word logic instruction, or
- about the bit shifted out in a shift function.

The CC1 and CC0 condition codes can be checked using the following instructions.

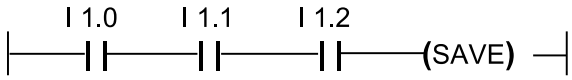
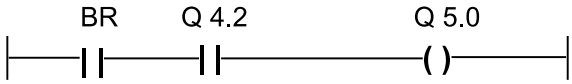
CC1	CC0	Check fulfilled, if:	
0	0	A ==0	Result = 0 (ACCU2 = ACCU1)
1	0	A >0	Result > 0 (ACCU2 > ACCU1)
0	1	A <0	Result < 0 (ACCU2 < ACCU1)
1	1	A UO	unordered instruction (e.g. division by 0).

In addition, jump functions exist that evaluate the digital displays and thus permit a suitable program branching.

LAD/FBD

You can find the checks for the LAD or FBD programming languages in the Catalog under Status Bits.

Instructions with Status Bits

Instruction	Meaning	Example
• SET	Set RLO to "1"	SET //RLO-1 = M 0.1
• CLR	Set RLO to "0"	CLR //RLO-0
• NOT	Invert RLO	O Manual mode; O Automatic mode; NOT; = Operating modes = M0.0
• SAVE	Save RLO in binary result	
• A BR	Check binary result	

L STW/T STW

It is also possible to load the entire status word and to save it for a later check (scan).

- L STW Load status word
- T MW 114 Save in memory word 114

With the T STW instruction, the status word can, for example, be loaded with a previously saved status word. The bits 0, 2, 3, 9..15 are not affected by this instruction.

Change RLO

With STEP7 there are a number of instructions that affect the RLO.

With SET you set the result of logic operation to "1", with CLR to "0". Parallel to this, the status bit STA is also set to "1" or to "0". Both instructions are executed independent of conditions.

SET and CLR also reset the status bits OR and /FC, that is, a new check string begins afterwards.

The instruction NOT negates the result of logic operation.

BR Bit

The BR bit represents an internal bit memory, in which the RLO can be saved before an RLO changing instruction. This is so that the RLO is then available for the resumption of the interrupted bit string.

If you write a function block or a function and want to call it in LAD, you have to manage the BR bit. The BR bit corresponds to the enable output (ENO) for the LAD box.

Setting and Resetting BR

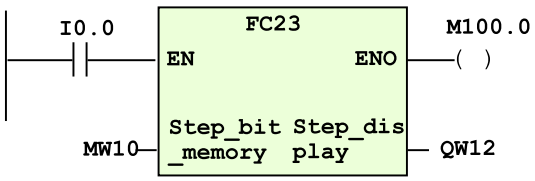
With SAVE you save the RLO in the binary result (register). SAVE transmits the signal state from the RLO to the status bit BR.

SAVE is executed independent of any conditions and does not affect any further status bits.

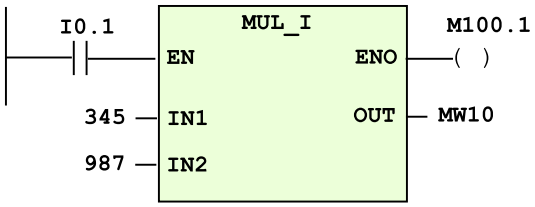
BR Bit and ENO in a Block Call or Complex Functions

LAD

Network 1: Cyclic Program



Network 2: ???



STL

```
Network 1: Cyclic Program
        A     I   0.0
        JNB   _001
        CALL  FC  23
         Step_bit_memory  :=MW10
         Step_display     :=QW12
_001:   A     BR
        =     M   100.0
```

```
Network 2: ???
        A     I   0.1
        JNB   _002
        L     345
        L     987
        *I
        T     MW  10
        AN    OV
        SAVE
        CLR
_002:   A     BR
        =     M   100.1
```


EN = Enable input

You can modify a call with the help of the enable input EN. It exists at every Box for the call of a block or a complex function in Ladder Diagram (corresponds to the conditional call in STEP 5).

- If EN is not activated, (that is, the signal state is "0"), then the Box does not execute its function. The enable output ENO is also not activated accordingly.
- If EN is activated (that is, the signal state is "1"), then the function in the box is executed.

ENO = Enable output

With the help of the enable output ENO, the block that is called or the complex function can signal whether the processing was executed error-free or not.

You can use the BR-bit of the status word to save errors. The BR-bit is only changed by the user program, not by the system.

If an error occurs during execution, you can "save" this error state by resetting the BR-bit. After executing a Box in LAD/FBD, the state of the BR-bit is then copied into the "output parameter" ENO.

A uniform mechanism for passing the error status is thus available in STEP 7. In this way, for example, a block that is called can inform the calling block if the processing was executed error-free or not.

Note

The EN parameter is not a true input parameter. If it is assigned, then two instructions with a conditional jump to a label behind the Box execution are automatically generated.

By the same token, ENO is not a true output parameter. If ENO is assigned, then two instructions for copying the BR-bit into the current output parameter are automatically generated.

The value of the BR-bit affects also the display of the processed blocks in the LAD/STL/FBD- editor when the debug -> monitor function is active.

A called block with the BR-bit set to 1 is displayed with status fulfilled, otherwise it is displayed with status not fulfilled.

Jump Functions Dependent on Status Bits

- JU Label[1)] — Jump unconditional
- JC Label[1)] — Jump if "RLO" bit =1
- JCN Label[1)] — Jump if "RLO" bit = 0
- JCB Label[1)] — Jump if "RLO" bit = 1 and save RLO
- JNB Label[1)] — Jump if "RLO" bit = 0 and save RLO
- JBI Label[1)] — Jump if "BR" bit = 1
- JNBI Label[1)] — Jump if "BR" bit = 0
- JO Label[1)] — Jump if "OV" bit in status word =1
- JOS Label[1)] — Jump if "OS" bit in status word =1

[1)] Label can consist of 4 alpha-numeric characters

SIMATIC S7

Date: 30.06.2011
File: PRO2_01E.7

SITRAIN Training for Automation and Drives

Jump Functions

With jump functions you can interrupt the linear processing of the program and continue at another location in the block. A program branching can be executed independent of conditions or, only then when specific conditions are fulfilled.

Jump Unconditional

The JU jump function is always executed, that is, independent of any conditions. JU interrupts the linear processing of the program and resumes it at the jump label. JU does not affect the status bits, neither at the jump nor at the destination.

Jump Functions with RLO and BR

A program branching can take place dependent on the states of the RLO and BR bits. In addition, it is possible during the RLO bit check to save this in the BR bit at the same time.

The RLO-dependent jump functions (JC, JCN) set, not only for fulfilled but also for unfulfilled conditions, the status bits STA and RLO to "1" and the bits OR and /FC to "0".

The RLO saving jump functions (JCB, JNB) save the state of the RLO bit in the BR bit. The remaining bits, STA, RLO, OR and /FC, are handled in the same manner as those jump functions that do not save the RLO.

The jump functions (JBI, JNBI) that are dependent on the BR bit set the status bit STA to “1” and the OR and /FC bits to "0", not only for fulfilled but also for unfulfilled conditions. The RLO and BR bits remain unchanged

Jump Functions Dependent on Condition Codes

- JZ Label[1)] — Jump if in the status word bit "CC 1"=0 and "CC 0"=0 (Result = 0)
- JN Label[1)] — Jump if in the status word bit "CC 1" is not equal to "CC 0" (Result <> 0)
- JP Label[1)] — Jump if in the status word bit "CC 1"=1 and "CC 0"=0 (Result > 0)
- JM Label[1)] — Jump if in the status word bit "CC 1"=0 and "CC 0"=1 (REsult < 0)
- JPZ Label[1)] — Combines the jumps JZ and JP (Result >= 0)
- JMZ Label[1)] — Combines the jumps JM and JZ (Result <= 0)
- JUO Label[1)] — Jump if: invalid Real number or division by zero

[1)] Label can consist of 4 alpha-numeric characters

Jump Functions with OV and OS

The jumps JO and JOS are executed if an overflow has occurred. In a string calculation with several successively executed instructions, the evaluation of the OV bit must take place after every mathematical function. A mathematical instruction, whose result lies in the allowed numbers range, and that follows an overflow, resets the OV bit.

In order to evaluate a possible numbers range overflow at the end of a string calculation, you have to check the OS bit. The OS bit is only reset with block call and block end as well as with the JOS jump.

The remaining bits in the status word are not changed with the jump functions JO and JOS.

Jump Functions with CC0 and CC1

A program function can take place dependent on the status bits CC0 and CC1. With it you can check if the result of a calculation is positive, zero, or negative, for example .

The jump functions dependent on the status bits CC0 and CC1 do not change any status bits. The result of logic operation is “taken along” with a jump and can thus be used for further logic operations in the user program (no change to /FC).

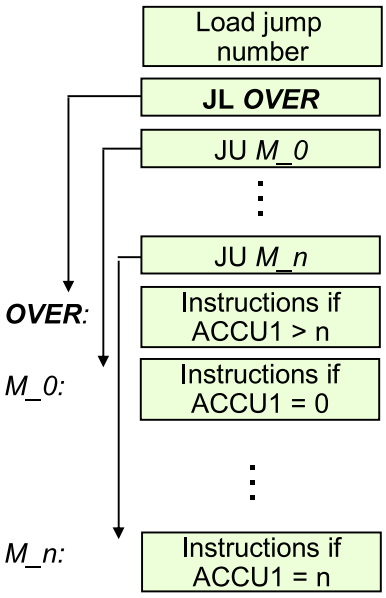
Example

Subtraction of two integers with subsequent evaluation :

```
      L MW2
      L MW8
      -I
      JZ  ZERO      // Jump occurs if result equals "0"
                    // Instructions, if result is not equal to "0"
ZERO: .             // Instructions for the reaction when result equals "0"
      .
      .
      .
```

Programming Jump Distributors

Mode of Operation



Example: Jump distributor with 3 entries

```
        L Recipe no// Load recipe no.
        JL OVER     // Jump distributor with 3 entries
        JU Rec0     // Jump to recipe no. 0 (ACCU1 = 0)
        JU Rec1     // Jump to recipe no. 1 (ACCU1 = 1)
        JU Rec2     // Jump to recipe no. 2 (ACCU1 = 2)
OVER:   JU ERROR // Jump if ACCU1 > 2
        ...
Rec0:   L DBW4
        ...
Rec1:   L DBW6
        ...
Rec2:   L DBW8
        ...
ERROR:  CLR
        ...
```

SIMATIC S7

Date: 30.06.2011
File: PRO2_01E.9

SITRAIN Training for Automation and Drives

Jump Distributor

The JL jump distributor permits the targeted jumping to a program section in the block dependent on a jump number. The JL instruction works together with a list of JU jump functions.

This list follows immediately after JL and can include a maximum of 256 entries. With JL there is a jump label that points to the end of the list, that is, to the first instruction after the list.

Only JU instructions can be placed between the JL <Jump label> and the <jump label>: <instruction>. If "0" is located in the ACCU1-L-L, the first jump instruction is executed, with "1" the second, etc. If the number is larger than the length of the list, JL branches to the end of the list.

The JL instruction is executed regardless of any conditions and for its part does not change the status bits.

Note

Jumps can be made forwards as well as backwards. Jumps can only be executed within one block. That is, the jump instruction and the jump destination must be within the same block.

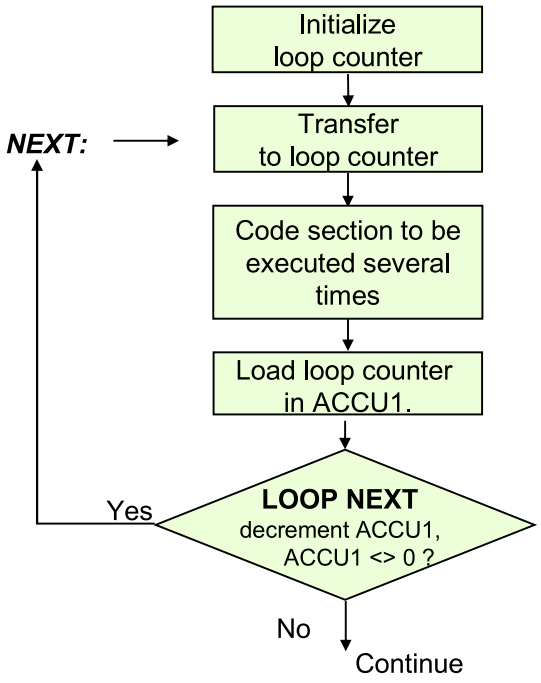
The jump destination can only exist once within this block. The maximum jump width lies between -32768 or +32767 words of program code. The actual maximum number of statements you can jump over depends on the mix of the statements used in your program (one-, two-, or three word statements).

The length of the jump label is limited to four alpha-numeric characters, whereby the first character must be a letter. Differentiation is made between capital and small letters in the jump labels.

An instruction must always be placed after the jump label - separated by a ":" .

Programming Loop Instructions

Mode of Operation



Example: Loop instruction with 4 scans

```
      L  +4          // Load loop counter
NEXT: T  MW10        // Beginning of the loop
      .              // Code
      .              // Code
      .              // Code
      L  MW10        // Load loop counter again
      LOOP NEXT      // Loop counter -1 and
                     // jump to jump label
                     // NEXT if Accu 1> 0
```

SIMATIC S7

Date: 30.06.2011
File: PRO2_01E.10

SITRAIN Training for Automation and Drives

Loop Instruction

The loop instruction LOOP simplifies the programming of program loops.

For programming a loop instruction, the desired number of loop scans to be executed are loaded in ACCU1-L. LOOP interprets the right word of Accumulator 1 as unsigned 16-bit number in the range from 0 to 65535.

With every execution of the LOOP instruction, the value in ACCU1-L is decremented by one. Subsequently, the value is compared to zero. If the value is unequal to zero, a jump takes place to the jump label designated in the LOOP instruction. If the value is equal to zero no jump takes place, and instead, the immediately following instruction is executed.

Note

The loop counter must not be initialized with 0, because this would cause the loop to be executed 65535 times

Instructions for Block End

- **BE** Block End
- **BEU** Block End Unconditional (within a block)
- **BEC** Block End Conditional (dependent on the RLO)

——(RET) in the LAD programming language

——[RET] in the FBD programming language

Block End Functions

You can end the processing of blocks with BEC, dependent on the result of logic operation or with BEU or BE independent of conditions.

BE

The BE instruction ends program execution in the current program block. BE is always the last instruction of a block. It is automatically generated by the PG when a block is saved. Thus, it does not have to be entered separately. This BE is not visible.

The operating system branches back to the calling block and resumes program execution with the first instruction after the program call. The current reserved local data area is once again enabled.

BEU

The BEU instruction ends program execution in the current program block just like BE.

Unlike the BE instruction, you can program BEU repeatedly within a block. The program section following BEU is only then processed if it is jumped to with a jump function.

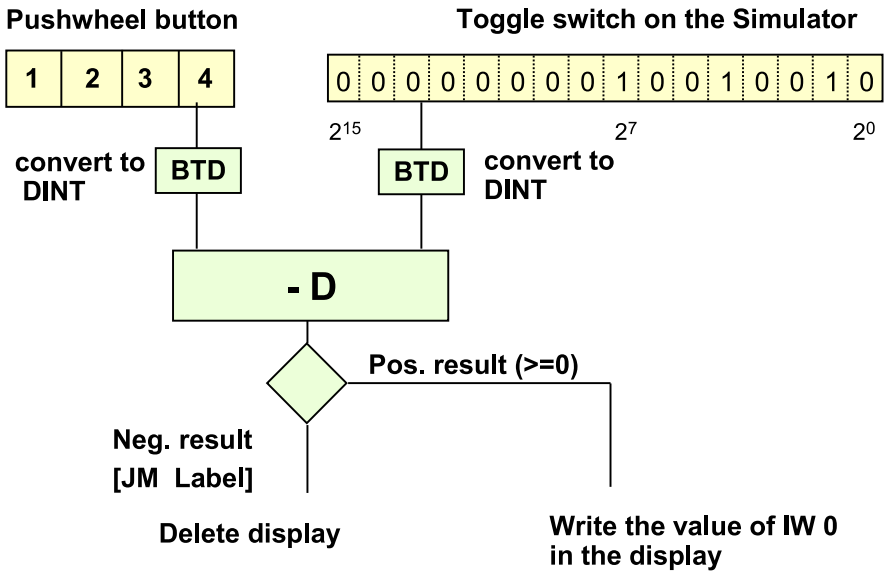
BEC

The block is ended dependent on the value of the RLO bit. If RLO = 1, program execution is ended in the current block and resumes in the calling block with the first instruction after the program call.

The current reserved local data area is once again enabled.

If RLO=0, the BEC instruction is not executed. The CPU sets the RLO to "1" and processes the instruction following BEC.

Exercise 1.1: Jump After a Subtraction



Decade switch: S7-300: IW4 (IW2, 32-Bit Mod.) S7-400: IW 30

Toggle switch: S7-300: IW0 (IW0) S7-400: IW28

Display: S7-300: QW12 (QW6, 32-Bit Mod.) S7-400: QW38

Date: 30.06.2011
File: PRO2_01E.12

Overview

With jump functions, the linear processing of a program can be interrupted and continued at another location. Such a jump can, in particular, be made dependent on conditions (results).

Goal of the Exercise

Programming a jump function that is executed depending on the result of a subtraction.

Task

Create a project PRO2 and then an S7-program folder with the name EXERCISE and generate an FC 11 with the following functionality:

1. Load the input words of the decade switch and the toggle switch as BCD coded values in the Accumulators.
2. Subsequently carry out a conversion of the values to DINT. For the conversion use the command BTD (BCD_TO_DINT). This command makes sure that the values read are interpreted as four-digit positive decimal numbers.
3. Then subtract the "value" of the toggle switch from the "value" of the decade switch.
4. Carry out, dependent on the result, the following actions:

   Result < 0: Delete the display on the simulator, that is, transfer "0" to the display.

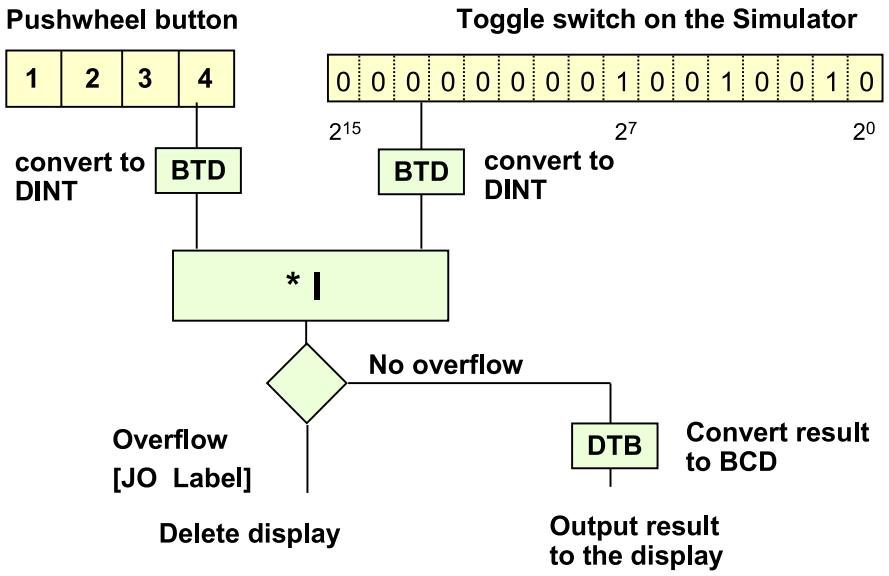
   Result >= 0: Output the BCD coded value of the decade switch to the display.

   Notes: Use the jump command "JM [*Label*]" for the case distinction.

   For masking out conversion errors during digit setting, program OB121 with an instruction: NOP 0.
5. Call FC11 in OB1 and download the blocks (OB1, OB121 and FC11) in the S7-CPU.
6. Test your program.

.

Exercise 1.2: Jump After a Multiplication



Decade switch: S7-300: IW4 (IW2, 32-Bit Mod.) S7-400: IW30

Toggle switch: S7-300: IW0 (IW0) S7-400: IW28

Display: S7-300: QW12 (QW6, 32-Bit Mod.) S7-400: QW38

Goal of the Exercise Programming a jump function that is executed depending on the result of a multiplication.

Task

Generate an FC 12 with the following functionality:

1. Load the input words of the decade switch and the toggle switch as BCD coded values (without sign) in the Accumulators.
2. Subsequently carry out a conversion of the values to DINT. For the conversion use the command BTD (BCD_TO_DINT). This command makes sure that the values read are interpreted as four-digit positive decimal numbers.
3. Then carry out a 16 bit multiplication.
4. Check your calculation results for "Overflow" and carry out the following actions:

   Overflow: Delete the display

   No Overflow: Carry out a conversion of the result in the corresponding positive BCD number and output the result (at least the four least significant digits) to the display.

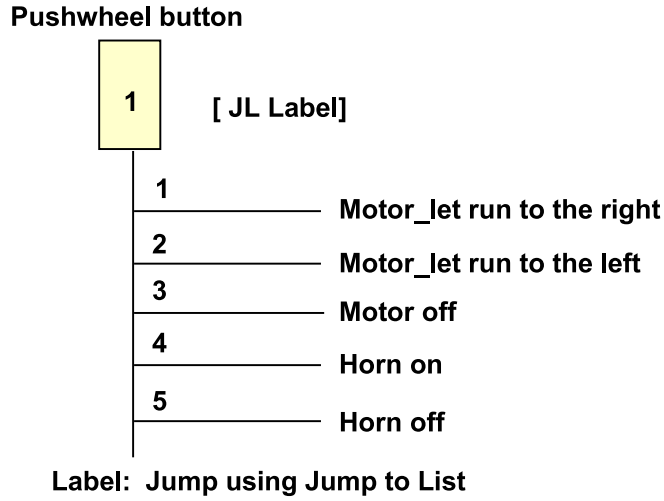
   Notes: Use the jump command "JO [*Label*]" for the testing for "Overflow"

   For masking out conversion errors during digit setting, program OB121 with an instruction: NOP 0.
5. Call FC12 in OB1 and download the program (OB1,OB121 and FC12) in the S7-CPU.
6. Test your program.

.

Exercise 1.3: Programming a Jump Distributor

Function:



Addresses:	S7-300 (16-Bit)	S7-300 (32-Bit)	S7-400
Motor_right:	Q20.5	Q8.5	Q40.5
Motor_left:	Q20.6	Q8.6	Q40.6
Horn:	Q20.7	Q8.7	Q40.7

SIMATIC S7

Date: 30.06.2011
File: PRO2_01E.14
SITRAIN Training for Automation and Drives

Goal of the Exercise You become familiar with the use of Jump To Lists.

Task

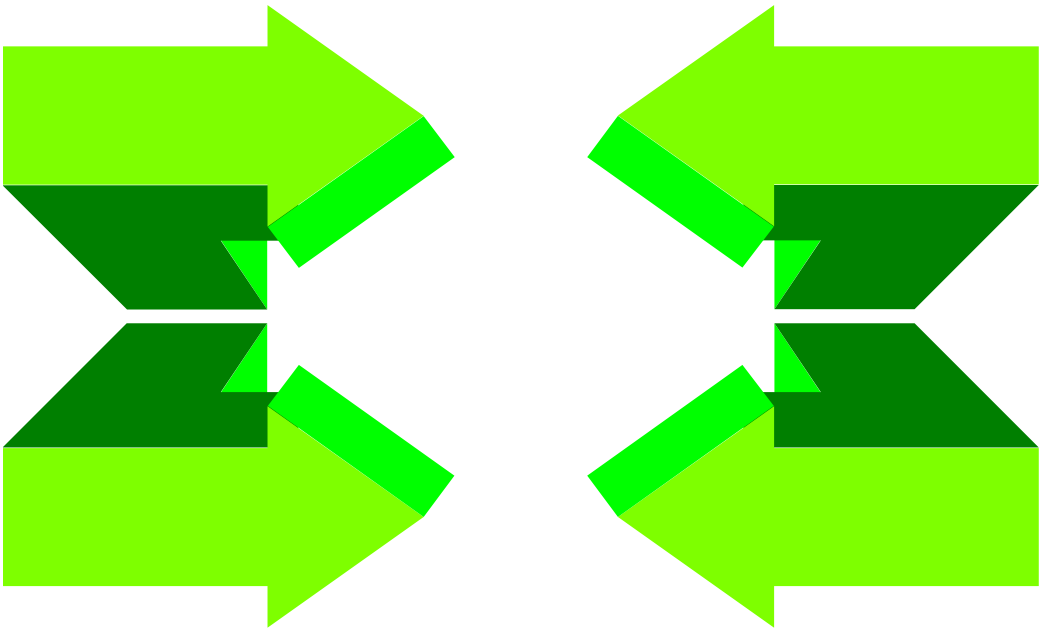
Generate an FC 13 with the following functionality:

- A number from 1 to 5 can be passed using the input parameter #Select from the INT data type
- Depending on the number passed, the following actions are carried out:
  1: The conveyor belt moves in the direction of the final assembly
  2: The conveyor belt moves in the opposite direction.
  3: The conveyor belt stops.
  4: The horn is switched on.
  5: The horn is switched off.
- All other numbers are interpreted as errors, that is, the horn and the conveyor belt are stopped and the "output parameter" ENO is set to FALSE.

What to Do

1. Generate an FC13 with the above described functionality.
   In the implementation of the Jump To List note that only absolute jumps can be used.
2. Call FC13 in OB1 dependent on I 0.0.
   Gate the input parameter #Select with the inputs of the decade switch and the “output parameter" ENO with Q8.0.
3. Download the FC13 and the OB1 and test your program.

Accumulator Functions



SIMATIC S7

Date: 30.06.2011
File: PRO2_02E.1

SITRAIN Training for Automation and Drives

Contents **Page**

Overview of the Accumulator Functions

Instructions, that affect several Accumulators

- TAK: Toggle (swap) the contents of ACCU1 and ACCU2
- PUSH: Shifting the ACCU contents "upward"
- POP: Shifting the ACCU contents "downward"
- ENT: Shifting the ACCU contents "upward", without ACCU1
- LEAVE: Shifting the ACCU contents "downward", without ACCU2
- Arithmetic Instructions and Word Logic Instructions

Instructions, that only affect ACCU1

- INC: Increment the contents of ACCU1-LL
- DEC: Decrement the contents of ACCU1-LL
- CAW: Reverse the order of the bytes in ACCU1-L
- CAD: Reverse the order of the bytes in ACCU1
- INVI, INVD: Forming the ones complement
- NEGI, NEGD, NEGR: Forming the twos complement (Negation)
- SLW, SLD, SRW, SRD, SSI, SSD: Shift the contents of ACCU1 word-by-word or doubleword-by-doubleword to the left or the right
- RLD, RRD: Rotate the contents of ACCU1 to the left or to the right
- RLDA, RRDA: Rotate the contents of ACCU1 to the left or to the right via the condition code bit CC1

Date: 30.06.2011
File: PRO2_02E.2

Overview

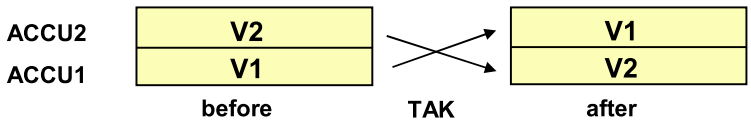
The accumulator functions transmit values between the accumulators or exchange bytes in ACCU1. The execution of pure accumulator functions is independent of the result of logic operation or of the status bits.

Equally, neither the result of logic operation nor the status bits are affected by the execution.

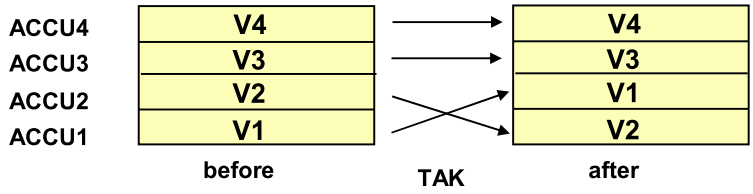
The accumulator functions permit the optimum runtime programming of automation tasks.

The Instruction TAK (Toggle ACCU1 and ACCU2)

S7-300:



S7-400:



TAK

TAK (Toggle ACCU1 with ACCU2) exchanges the contents of ACCU1 with the contents of ACCU2. The instruction is executed without regard to, and without affecting, the status bits. The contents of ACCU3 and ACCU4 remain unchanged for CPUs with four ACCUs. (for S7-400).

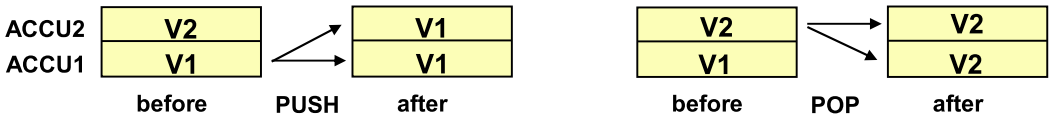
Example

Subtract the smaller value from the larger value:

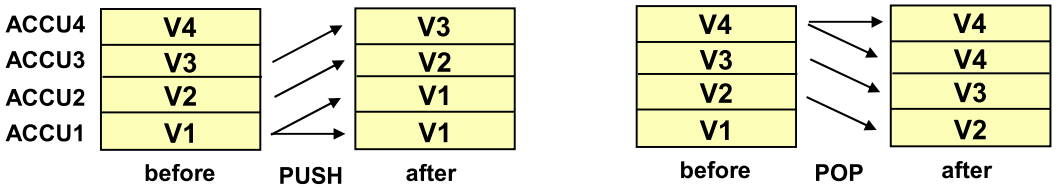
```
        L   MW10    // Load contents of MW10 into ACCU1-L.
        L   MW12    // Load contents of ACCU1-L into ACCU2-L. Load contents
                    // of MW12 into ACCU1-L.
        >I          // Check if ACCU2-L (MW10) is greater than ACCU1-L
                    // (MW12).
        JC NEXT     // Jump to NEXT jump label if ACCU2 (MW10)
                    // is greater than ACCU1 (MW12).
        TAK         // Swap the contents of ACCU1 and ACCU2.
NEXT:-I             // Subtract the contents of ACCU1-L from the contents
                    // of ACCU2-L.
        T   MW14    // Transfer the result (= greater value minus
                    // smaller value) to MW14
```

The Instructions PUSH and POP

S7-300:



S7-400:



SIMATIC S7

Date: 30.06.2011
File: PRO2_02E.4

SITRAIN Training for Automation and Drives

PUSH

The PUSH instruction shifts the contents of the accumulators, each into the next higher accumulator. PUSH is normally used in order to duplicate the value of ACCU1, without losing the original contents of ACCU2 or ACCU3 (only for S7-400).

- PUSH (S7-300): The PUSH instruction copies the entire contents of ACCU1 to ACCU2. ACCU1 remains unchanged.
- PUSH (S7-400): The PUSH instruction copies the contents of ACCU3 to ACCU4, the contents of ACCU2 to ACCU3, and the contents of ACCU1 to ACCU2. ACCU1 remains unchanged

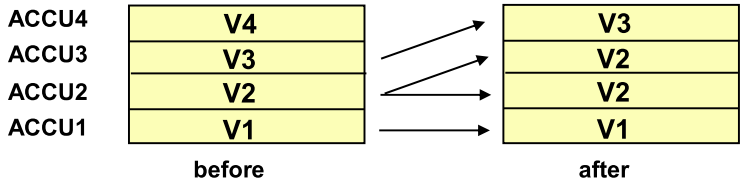
POP

The POP instruction shifts the values found in accumulators 2 to 4 into the underlying accumulators. This instruction is normally executed after transfer instructions, when the contents of ACCU1 are no longer required and processing is to continue with the values saved in the upper accumulators.

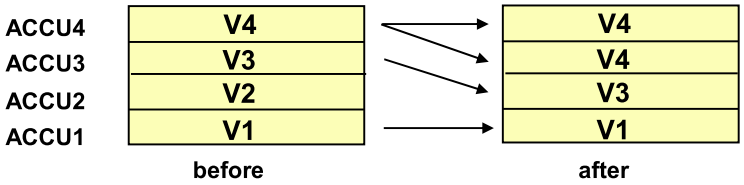
- POP (S7-300): The POP instruction copies the entire contents of ACCU2 to ACCU1. ACCU2 remains unchanged.
- POP (S7-400): The POP instruction copies the contents of ACCU2 to ACCU1, the contents of ACCU3 to ACCU2, and the contents of ACCU4 to ACCU3. ACCU4 remains unchanged.

The Instructions ENT and LEAVE (only S7-400)

ENT:



LEAVE:



ENT

The ENT instruction(ENTER ACCU Stack)) shifts the contents of accumulators 2 and 3 each into the next higher accumulator. The contents of accumulators 1 and 2 remain unchanged.

ENT in conjunction with an immediately following load function:

- ENT
  L ...

has as a result that during loading, the contents of accumulators 1 to 3 are shifted "upwards" (similar to PUSH) and the loaded value in ACCU1 remains.

The ENT instruction is executed without regard to, and without affecting, the status bits.

LEAVE

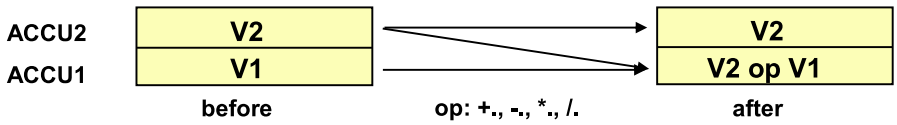
The LEAVE instruction shifts the contents of accumulators 3 and 4 each into the underlying accumulator. The contents of accumulators 4 and 1 remain unchanged.

The arithmetic functions contain the functionality of LEAVE. With LEAVE you can emulate the same functionality in other digital logic operations (a word logic instruction, for example) as well.

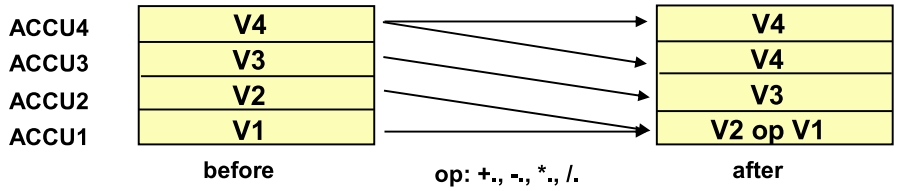
LEAVE programmed after a digital logic operation, fetches the contents of accumulators 3 and 4 into the accumulators 2 and 3. The result of the digital logic operation remains unchanged in accumulator 1.

Arithmetic Instructions

S7-300:



S7-400:



SIMATIC S7

Date: 30.06.2011
File: PRO2_02E.6

SITRAIN Training for Automation and Drives

Arithmetic Instructions

The arithmetic instructions combine two digital values found in accumulators 1 and 2, according to the fundamental operations of arithmetic. The result of the calculation is found in ACCU1.

The status bits CC0, CC1 and OV and OS give information about the result or the intermediate result of the calculation.

S7-300

With the S7-300 CPUs, the contents of ACCU2 remain unchanged with the execution of an arithmetic function.

S7-400

With the S7-400 CPUs, the contents of ACCU2 are overwritten by the contents of ACCU3. The contents of ACCU4 are transferred to ACCU3.

Example

The following program segment produces different results, depending on whether the code is run on an S7-300 CPU or on an S7-400 CPU:

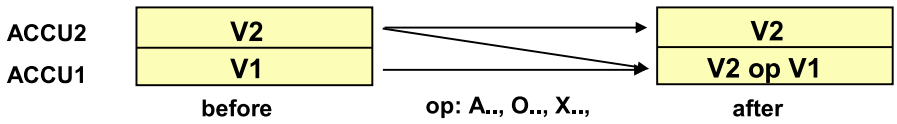
```
L     0      // load integer number 0 in ACCU1
L     5      // load integer number 5 in ACCU1, 0 in ACCU2
PUSH         // shift 5 (ACCU1) to ACCU2;      (S7-400: ACCU2 -> ACCU3)
*I           // multiply ACCU1 with ACCU2;     (S7-400: ACCU3 -> ACCU2)
*I           // multiply ACCU1 with ACCU2;     (S7-400: ACCU3 -> ACCU2)
```

Result:

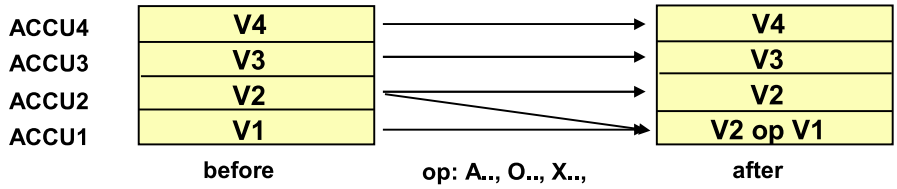
S7-300: ACCU1 = 125
S7-400: ACCU1 = 0

Word Logic Instructions

S7-300:



S7-400:



SIMATIC S7

Date: 30.06.2011
File: PRO2_02E.7

SITRAIN Training for Automation and Drives

Word Logic Instructions

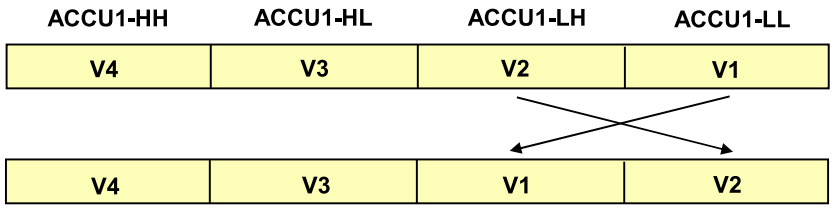
The word logic instructions combine bit by bit the values found in ACCU1 with a constant or with the contents of ACCU2 and store the result in ACCU1.

The contents of the remaining ACCUs (ACCU2 for S7-300, or ACCU2, ACCU3 and ACCU4 for S7-400) remain unchanged. The logic operation can be performed word or double-word-wise.

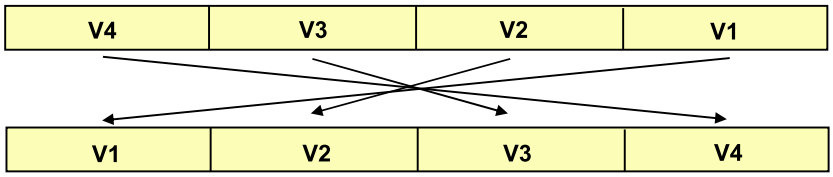
The instructions AND, OR or Exclusive OR are available as word logic instructions.

Change Instructions for ACCU1

CAW:



CAD:



SIMATIC S7

Date: 30.06.2011
File: PRO2_02E.8

SITRAIN Training for Automation and Drives

CAW

With the CAW instruction, the bytes in the right data word of ACCU1 are exchanged. That is, the contents of ACCU1-LH is transferred to ACCU1-LL and vice versa.

With this instruction it is possible to convert a 16-bit numbers format (INT and WORD) in the SIMATIC programming language to the numbers format of the INTEL programming language (data transfer to PCs).

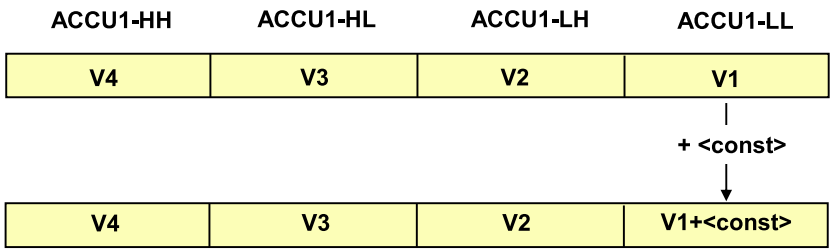
CAD

With the CAD instruction, the bytes in ACCU1 are exchanged. That is, the contents of ACCU1-HH are transferred to ACCU1-LL and vice versa or the contents of ACCU1-HL to ACCU1-LH and vice versa.

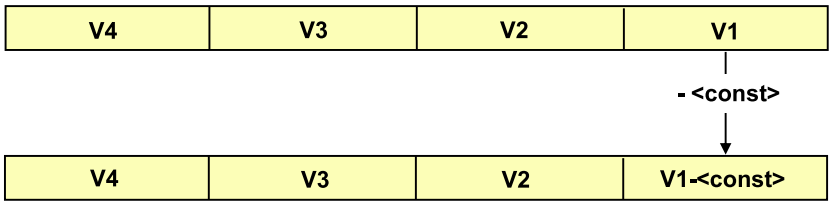
With this instruction it is possible to convert a 32-bit numbers format (DINT, DWORD and REAL) in the SIMATIC programming language to the numbers format of the INTEL programming language (data transfer to PCs).

Increment Instructions for ACCU1

INC <const>:



DEC <const>:



SIMATIC S7

Date: 30.06.2011
File: PRO2_02E.9

SITRAIN Training for Automation and Drives

INC

The INC instruction <8-bit integer> adds the 8-bit integer to the contents of ACCU1-LL and saves the result in ACCU1-LL.

ACCU1-LH, ACCU1-H and ACCU2, or ACCU3 and ACCU4 remain unchanged.

DEC

The DEC instruction <8-bit integer> subtracts the 8-bit integer from the contents of ACCU1-LL and saves the result in ACCU1-LL.

ACCU1-LH, ACCU1-H and ACCU2, or ACCU3 and ACCU4 remain unchanged.

Notes

The INC and DEC instructions are so-called "Low Level Instructions". That is, in case of an overflow, the processor does not set any overflow bits in the status word.

Instead of the INC instruction, you can also use the following instructions for the INT- or DINT- addition, for example:

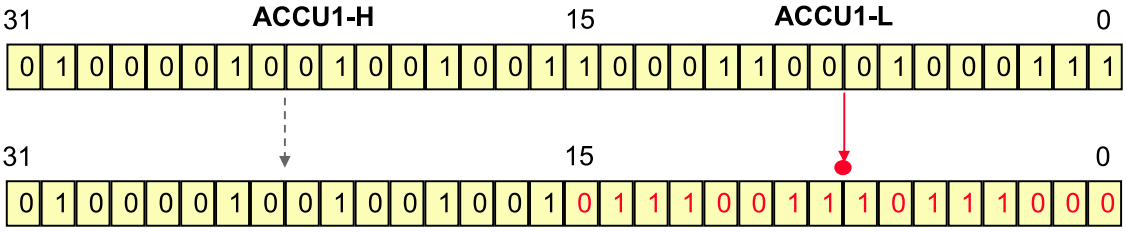
+ <const> (adds a 16-bit <const> to ACCU1)
+ L#<const> (adds a 32-bit <const> to ACCU1)

Instead of the DEC instruction, you can use the following instructions for the INT- or DINT- subtraction:

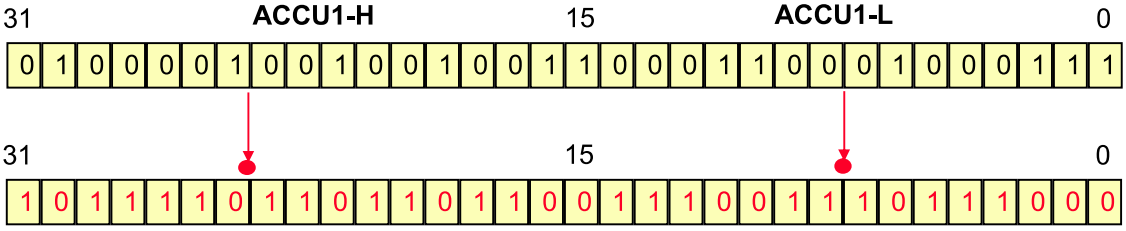
+ - <const> (subtracts a 16-bit <const> from the contents of ACCU1)
+ L# - <const> (subtracts a 32-bit <const> from the contents of ACCU1)

Forming the Ones Complement

INVI (Ones Complement of ACCU1-L):



INVD (Ones Complement of ACCU1):



SIMATIC S7

Date: 30.06.2011
File: PRO2_02E.10

SITRAIN Training for Automation and Drives

INVI The INVI instruction negates bit by bit the value (bits 0 to 15) found in the right word of Accumulator 1. It replaces the zeros with ones and the ones with zeros. The contents of the left word (bits 16 to 31) remain unchanged.

The INVI instruction does not set any status bits.

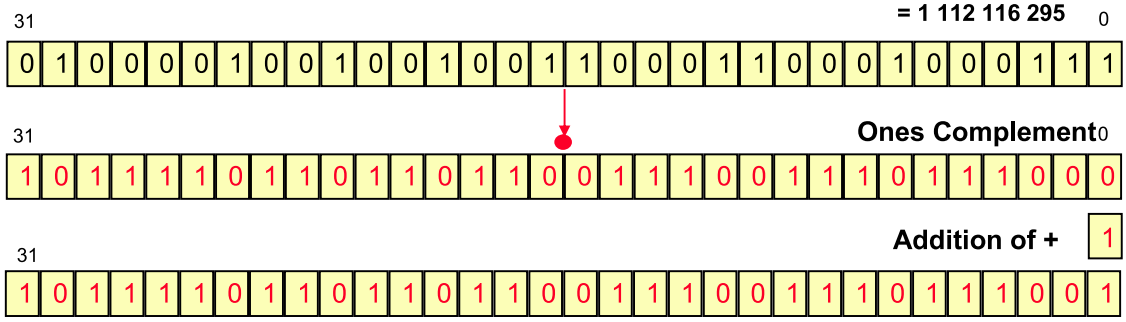
INVD The INVD instruction negates bit by bit the value (bits 0 to 31) found in Accumulator 1. It replaces the zeros with ones and the ones with zeros.

The INVD instruction does not set any status bits.

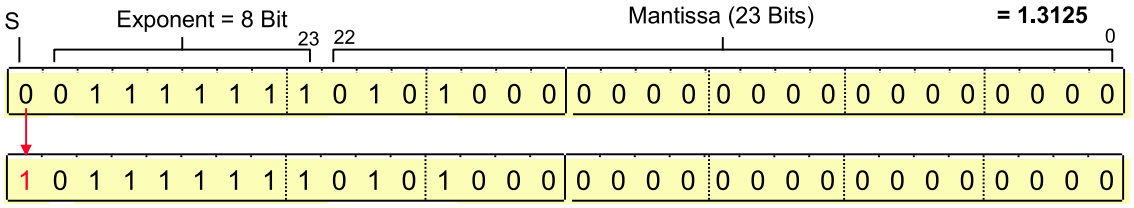
Negation of Numbers (Twos Complement)

NEGI (Negation of INT numbers)

NEGD (Negation of DINT numbers):



NEGR (Negation of REAL numbers):



NEGI

The NEGI instruction interprets the value (bits 0 to 15) found in the right word of ACCU1 as an INT number and by forming a twos complement reverses the sign.

This instruction is equivalent to multiplication by “-1”. The left word of ACCU1 (bits 16 to 31) remains unchanged.

The status bits CC1, CC0, OS and OV are set as a function of the result of the operation.

NEGD

The NEGD instruction interprets the value found in ACCU1 as a REAL number and multiplies it by ‘-1’.

Forming the twos complement can also be achieved by forming a ones complement and subsequently adding 1 (with a binary number this is equivalent to a subtraction).

The status bits CC1, CC0, OS and OV are set as a function of the result of the operation.

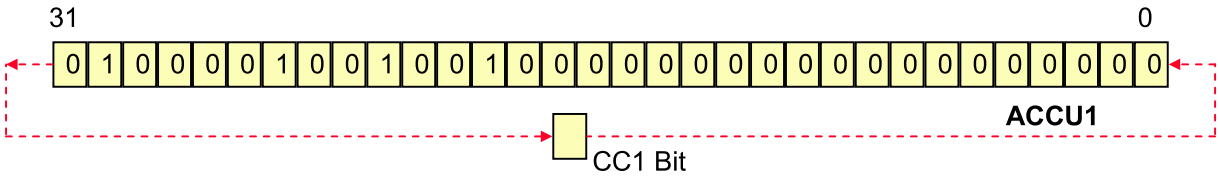
NEGR

The NEGR instruction interprets the value found in ACCU1 as a REAL number (32 bit, IEEE-FP) and multiplies this number by ‘-1’. The instruction inverts the state of bit 31 in ACCU1 (sign of the mantissa).

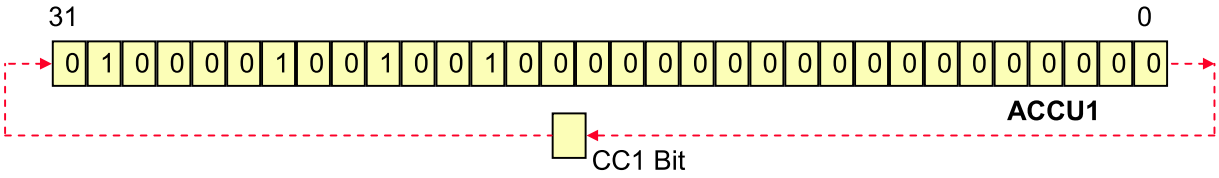
The NEGR instruction does not change any status bits.

32 Bit Rotate Instructions via CC1 Bit

RLDA (Rotate left via status bit CC1):



RRDA (Rotate right via status bit CC1):



RLDA

The RLDA shift function shifts the entire contents of ACCU1 by 1 bit to the left. The bit position (bit 0) that becomes free during shifting is overwritten with the value of status bit CC1. The status bit CC1 then gets the value of the pushed out bit (bit 31);

The status bits CC0 and OV are reset to "0".

- Example:

  ACCU1: 0100 0100 1100 0100
  CC1: 1

  RLDA

  ACCU1: 1000 1001 1000 1001
  CC1: 0

RRDA

The RRDA shift function shifts the entire contents of ACCU1 by 1 bit to the right. The bit position (bit 31) that becomes free during shifting is overwritten with the value of status bit CC1. The status bit CC1 gets the value of the pushed out bit (bit 0);

The status bits CC0 and OV are reset to "0".

- Example:

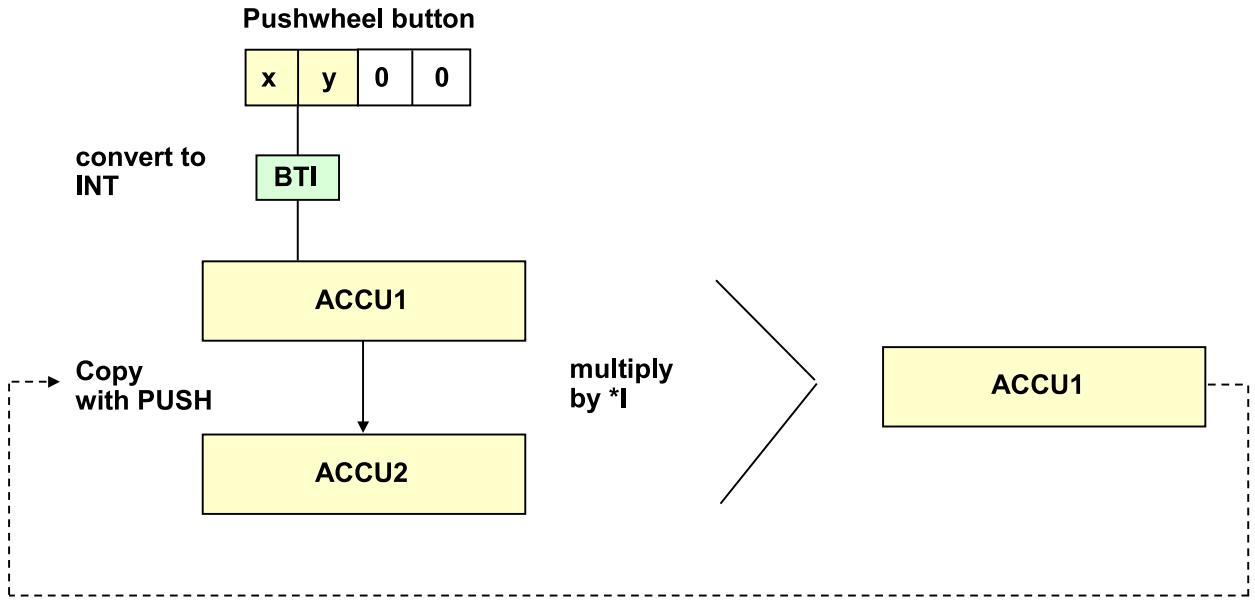
  ACCU1: 0100 0100 1100 0100
  CC1: 1

  RRDA

  ACCU1: 1010 0010 0110 0010
  CC1: 0

Exercise 2.1: Calculation of Exponents

Example: Forming the 6th power of an integer through successive use of PUSH and *I



Goal of the Exercise

To become familiar with the ACCU change functions by means of the calculation of powers of an Integer.

Task

Create an FC21 with the following functionality:.

- Read-in the pushwheel button's left byte and convert the BCD value into an Integer value (BTI).
- Form the 6th power from the read-in value.

What to Do

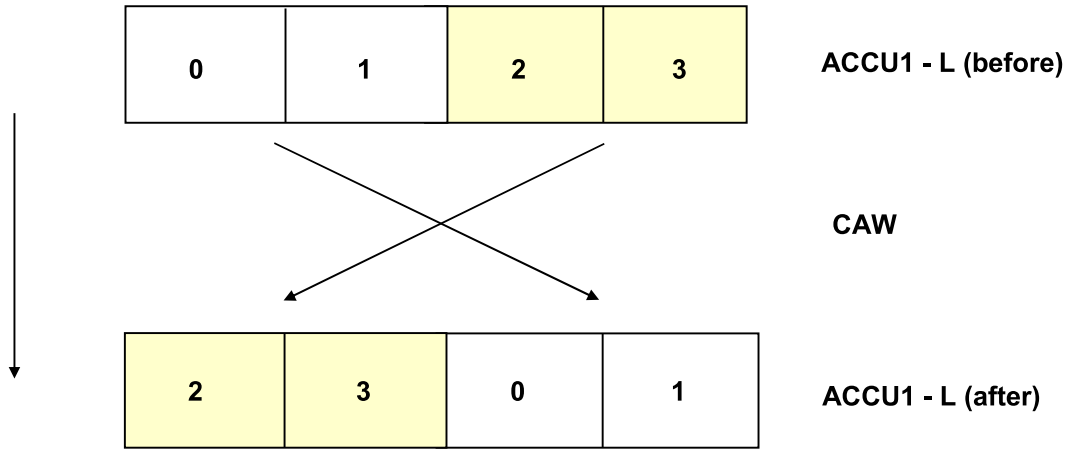
1. Copy the contents of ACCU1 into ACCU2 with the help of the PUSH instruction.
2. Multiply ACCU1 by ACCU2 (forming the square).
3. Copy the contents of ACCU1 into ACCU2 with the help of the PUSH instruction.
4. etc., etc., etc.

   Careful: How must you carry out the fourth step, so that FC21 delivers the correct result on an S7-300-CPU as well as on an S7-400-CPU?
5. Display the result on the digital display.
6. Call FC21 in OB1 and download the program in the S7-CPU.
7. Test the program.

Note

So that the read-in value does not become too large, you should only use the right decade.

Exercise 2.2 : Data Exchange in ACCU1



SIMATIC S7

Date: 30.06.2011
File: PRO2_02E.14

SITRAIN Training for Automation and Drives

Goal of the Exercise

You are to become familiar with the instruction for swapping bytes within ACCU1.

Application: Converting the number representation on a SIMATIC controller into the representation on a PC using INTEL-CPU (80486, Pentium,...).

These conversions must always be carried out when number values are exchanged between a SIMATIC controller and a PC.

Task

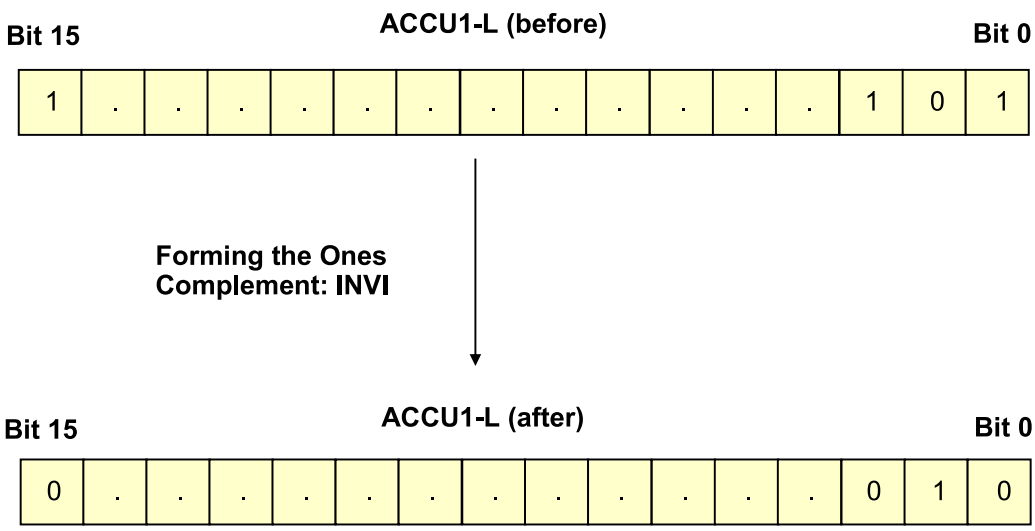
Create an FC22 with the following functionality:

- Load the pushwheel button's value in ACCU1.
- In ACCU1 - L swap the two bytes with the help of the CAW instruction.
- Display the contents of ACCU1 on the digital display.

What to Do

1. Create the FC22.
2. Call the FC22 in OB1.
3. Download the program into the S7-CPU.
4. Test the program.

Exercise 2.3 : Forming Complements



SIMATIC S7

Date: 30.06.2011
File: PRO2_02E.15

SITRAIN Training for Automation and Drives

Goal of the Exercise

You are to become familiar with the instruction for forming the complement in the SIMATIC S7.

Application: Converting a "negative" into a "positive" logic

Such conversions are always undertaken when 0-active signals are read-in, however the user program continues to work with positive logic.

Task

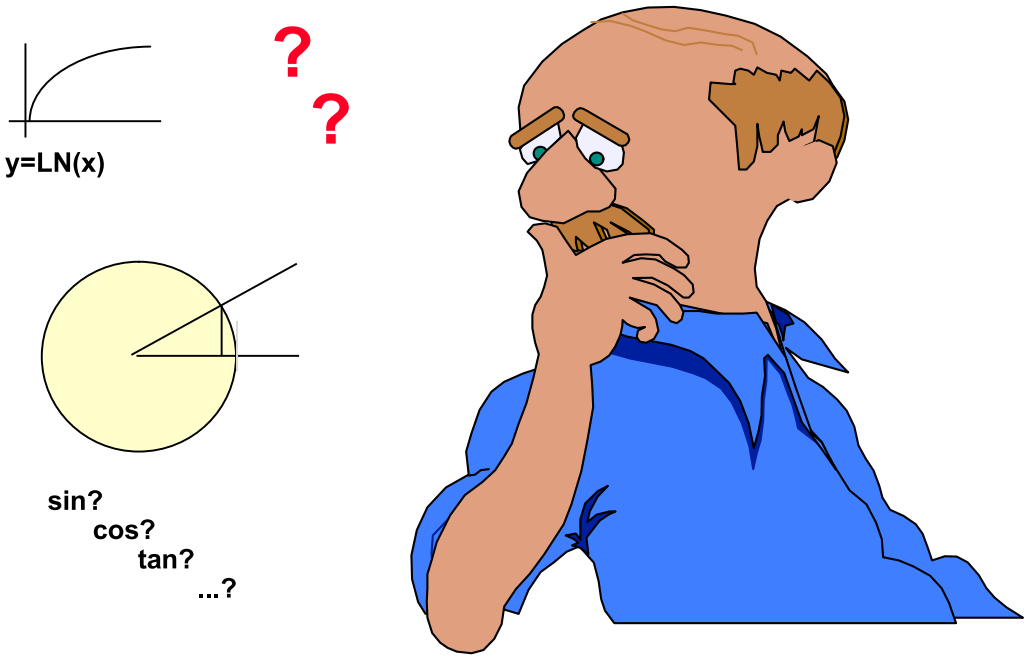
Create an FC23 with the following functionality:.

- Load the value of the toggle switch's word in ACCU1.
- Form the Ones Complement.
- Output the result to the simulator's LEDs.

What to Do

1. Create the FC23
2. Call the FC23 in OB1
3. Download the program into the S7-CPU.
4. Test the program with the help of the variable status and binary preset option.

Instructions with REAL Numbers

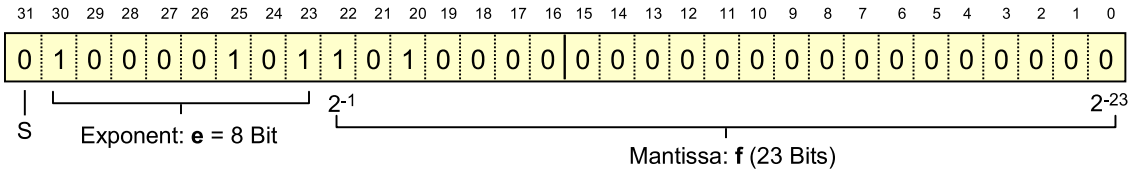


Date: 30.06.2011
File: PRO2_03E.1

Contents **Page**

REAL Number Representation in SIMATIC S7

- **Representation format of a REAL number (IEEE FP 32 bit binary format):**



- **Representation of a normalized REAL number:**

S x (1.f) x 2 (e-127)

S = Sign bit, (0 corresponds to +, 1 corresponds to -)
f = 23 bit Mantissa with MSB = 2^{-1} and LSB = 2^{-23}
e = binary integer exponent (0 < e < 255)

- **Example:**

S = 0
e = 1000 0101 = 133
f = 1010 0000... = 0.5 + 0.125

R = +1.625 x 2 (133-127) = 1.625 x 64 = **104.0**

- **Value range of normalized REAL numbers:**

- 3.402 823 x 10^{+38} ... -1.175 494 x 10^{-38}, 0, 1.175 494 x 10^{-38} ... 3.402 823 x 10^{+38}

Date: 30.06.2011
File: PRO2_03E.2

REAL Number

REAL (floating point) numbers enable the implementation of complex, mathematical calculations for process control and closed-loop process control.

A REAL data type variable consists internally of three components: the sign, the 8-bit exponent of Base 2 and the 23-bit Mantissa.

The sign can take the values "0" (positive) or "1" (negative). The exponent is increased by one constant (Bias, +127) and stored, so that it has a value range of 0 to 255.

The Mantissa represents the fractional part. The integer part of the Mantissa is not stored, since it is always either 1 (for normalized floating point) or 0 (for denormalized floating point).

Range Limits

Designation	Value e	Mantissa f	Value	CC1	CC0	OV	OS
Not float.pt.no.	255	<>0	[qNaN]	1	1	1	1
Overflow	255	0	>(2-2^{-23}) 2^{127} <(-2+ 2^{-23}) 2^{127}	1 0	0 1	1 1	1 1
Normalized No.	1.. 254	any	(1.f) 2^{e-127} (-1.f) 2^{e-127}	1 0	0 1	0 0	- -
Denormaliz. No.	0	<>0	(0.f) 2^{-126} (- 0.f) 2^{-126}	0 0	0 0	1 1	1 1
Zero	0	0	+0	0	0	0	-

Note

The CPUs calculate with the full accuracy of the floating-point numbers. The display on the PG can deviate from the exact representation, due to round up errors in conversion. REAL numbers are rounded up to the sixth decimal place

Basic Instructions with REAL Numbers

- **REAL Addition:**

```
L     MD10    // Load 1st. REAL number
L     MD20    // Load 2nd. REAL number
+R            // Addition of REAL numbers (MD10 + MD20)
T     MD30    // Transfer result into MD30
```

- **REAL Subtraction:**

```
L     MD10    // Load 1st. REAL number
L     MD20    // Load 2nd. REAL number
-R            // Subtraction of REAL numbers (MD10 - MD20)
T     MD30    // Transfer result into MD30
```

- **REAL Multiplication:**

```
L     MD10    // Load 1st. REAL number
L     MD20    // Load 2nd. REAL number
*R            // Multiplication of REAL numbers (MD10 * MD20)
T     MD30    // Transfer result into MD30
```

- **REAL Division:**

```
L     MD10    // Load 1st. REAL number
L     MD20    // Load 2nd. REAL number
/R            // Division of REAL numbers (MD10 / MD20)
T     MD30    // Transfer result into MD30
```


Overview

The functions +R, -R, *R, /R interpret the values found in ACCU1 and ACCU2 as REAL data type numbers. They carry out the programmed logic operation (+R, -R, *R and /R) and save the result in ACCU1.

After the calculation is carried out, the status bits CC0 and CC1 indicate whether the result is negative (CC1=0, CC0=1), zero (CC1=0; CC0=0) or positive (CC1=1, CC0=0).

The status bits OV and OS signal the leaving of the permitted numbers range.

Unauthorized REAL Numbers

With an unauthorized calculation, that is, when one of the two input values is an invalid REAL number, then the result in ACCU1 is also an invalid REAL number.
Invalid REAL numbers are also stored as a result in ACCU1 if you try to process unauthorized values with the following instructions:

Addition: Addition of + infinite and - infinite.

Subtraction: Subtraction of + infinite and + infinite or - infinite and - infinite

Multiplication: Multiplication of 0 by infinite

Division: Division of infinite by infinite or 0 by 0.

The result of the division of valid REAL numbers by 0 is, depending on the number's sign, + infinite or - infinite.

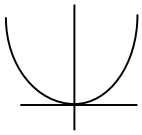
Note

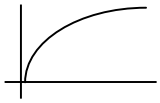
The hexadecimal number DW#16#FFFF FFFF represents an invalid REAL number, for example.

Extended Mathematical Functions

- **Mathematical Functions:**

SQR	Form the square
SQRT	Calculate the square root
EXP	Exponential function for Base e
LN	Natural Logarithm (e=2.718282)

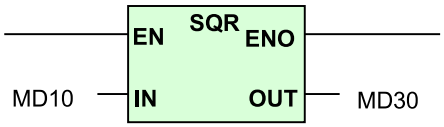




- **Example:**

```
L     MD10    // Load REAL number
SQR           // Calculate the square
T     MD30    // Transfer result into MD30
```

(STL)



(LAD)

Overview

The mathematical functions take the number in ACCU1 as the input value for the function to be carried out and store the result in ACCU1.

Mathematical functions only change the contents of ACCU1. The contents of ACCU2, or ACCU3 and AKKU4 for S7-400, remain unchanged.

Depending on the function result, the mathematical functions set the status bits CC0, CC1, OV and OS.

If there is an invalid REAL number in ACCU1 before the function is carried out, then the mathematical function returns an invalid REAL number and sets the status bits accordingly.

SQR

The SQR function squares the contents of ACCU1.

SQRT

The SQRT function calculates the square root from the value in ACCU1. If there is a value smaller than zero in ACCU1, SQRT sets the status bits CC0, CC1, OV and OS to "1" and returns an invalid REAL number.

If -0 (minus zero) is in ACCU1, -0 is also returned.

EXP

The EXP function calculates the power from the Base e (=2.71828) and the value (e^{ACCU1}) found in ACCU1.

LN

The LN function calculates the natural logarithm to Base e from the number found in ACCU1. If there is a value smaller than or equal to zero in ACCU1, LN sets the status bits CC0, CC1, OV and OS to "1" and returns an invalid REAL number.

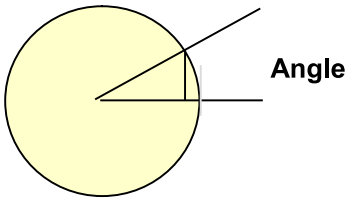
The natural logarithm is the reverse function to the exponential function:

If: $y = e^x$
then: $x = \ln y$

Trigonometric Functions and Their Inverse Functions

- **Trigonometric functions:**

SIN	Sine
COS	Cosine
TAN	Tangent



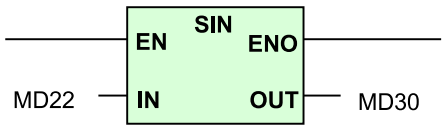
- **Arc functions:**

ASIN	Arc sine
ACOS	Arc cosine
ATAN	Arc tangent

- **Example:**

```
L     MD10     // Load REAL number
SIN            // Calculate the sine
T     MD30     // Transfer result into MD30
```

(STL)



(LAD)

Trigonometric Functions

The trigonometric functions expect an angle in radian measure as a REAL number in ACCU1. For the angle input (0^0 ... 360^0), you must, if necessary, perform a conversion to degree measure (0 ... 2 pi, with pi=3.141593).

During function execution, for values smaller than 0 or larger than 2pi, a multiple of 2pi is automatically added or subtracted until the value lies between 0 and 2pi (automatic modulo-2pi calculation).

Arc Functions

The arc functions are the inverse functions of the respective trigonometric functions. They expect a REAL number in a specific value range in ACCU1 and return an angle in radian measure:

Function	Permitted definition range	Value range
ASIN	-1 to +1	- pi/2 to + pi/2
ACOS	-1 to +1	0 to pi
ATAN	entire range	- pi/2 to + pi/2

With an overrange of the permitted definition range, the arc functions return an invalid REAL number and set the status bits CC0, CC1, OV and OS to "1".

Other Instructions with REAL Numbers

- **Conversion instructions from REAL to DINT:**

RND+	with rounding off to the next larger DINT number
RND-	with rounding off to the next smaller DINT number
RND	with rounding off to the next integer
TRUNC	integer component

- **Conversion instructions from DINT to REAL:**

DTR	conversion with rounding off

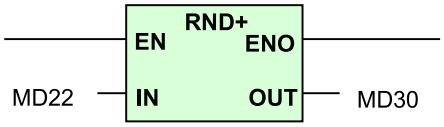
- **Other instructions from REAL to REAL:**

ABS	forming the absolute amount
NEGR	negation of a REAL number

- **Example:**

```
L       MD10    // Load REAL number
RND+            // Convert to next larger DINT number
T       MD30    // Transfer result into MD30
```

(STL)



(LAD)

Overview

The conversion functions convert the data type of the value found in ACCU1 into another data type and store the result in ACCU1. The contents of the other ACCUs remain unchanged.

If, in one of the instructions (RND+, RND-, RND or TRUNC), the value found in ACCU1 is larger or smaller than that for a number in the DINT format's permissible range or it doesn't correspond to a number in the REAL format, the instruction sets the status bits OV and OS to "1". A conversion then does not take place.

RND+

The RND+ instruction converts the contents of ACCU1 as REAL number into an integer (DINT), that is larger than or equal to the number to be converted.

RND-

The RND- instruction converts the contents of ACCU1 as REAL number into an integer (DINT), that is smaller than or equal to the number to be converted.

RND

The RND instruction converts the contents of ACCU1 as REAL number into the next possible integer (DINT). If the result lies exactly between an even and an uneven number, the even number is returned.

TRUNC

The TRUNC instruction returns the integer component of the number to be converted; the fraction component is cut off.

DTR

The DTR instruction converts a number from the DINT format into the REAL number format. Since a large number in the DINT format is more exact than a number in the REAL format, it is possible that during conversion a rounding takes place to the next valid number.

ABS

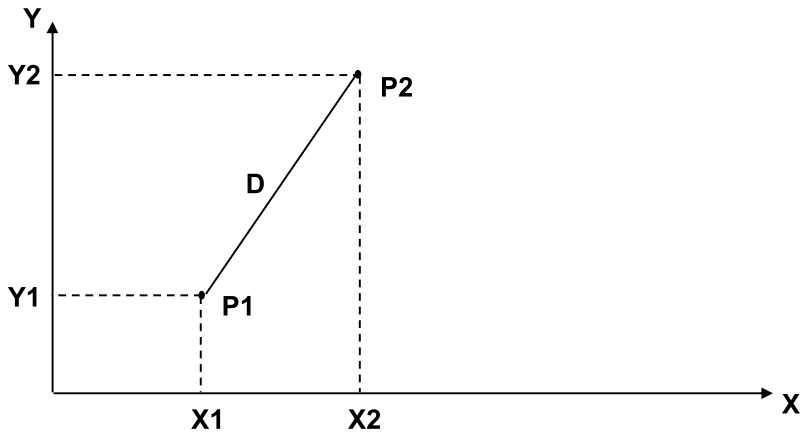
The ABS instruction forms the absolute value from the REAL number found in ACCU1, that is, the sign (bit 31) is set to "0" (even for an invalid REAL number).

NEGR

The NEGR instruction negates the REAL number in ACCU1. That is, the sign (bit 31) is inverted (even for an invalid REAL number). The instructions DTR, ABS and NEGR do not affect the status bits.

Exercsie 3.1: Calculating the Distance

Example: Calculating the distance D between two points in a right-angled coordinate system



Function: FC31 with $D=\sqrt{(X2 - X1)^2 + (Y2 - Y1)^2}$

Goal of the Exercise

The application of mathematical functions for calculating the distance between two points.

Task

Create an FC31 with the following functionality:

- FC31 expects the coordinates (X1, Y1) or (X2, Y2) of the two points P1 and P2 in the input parameters.
- FC31 returns the distance of both points in the output parameter RET_VAL.
- FC31 is to be installable in the S7-300 system as well as the S7-400 system. It may not use global CPU addresses for any possible saving of intermediate results.

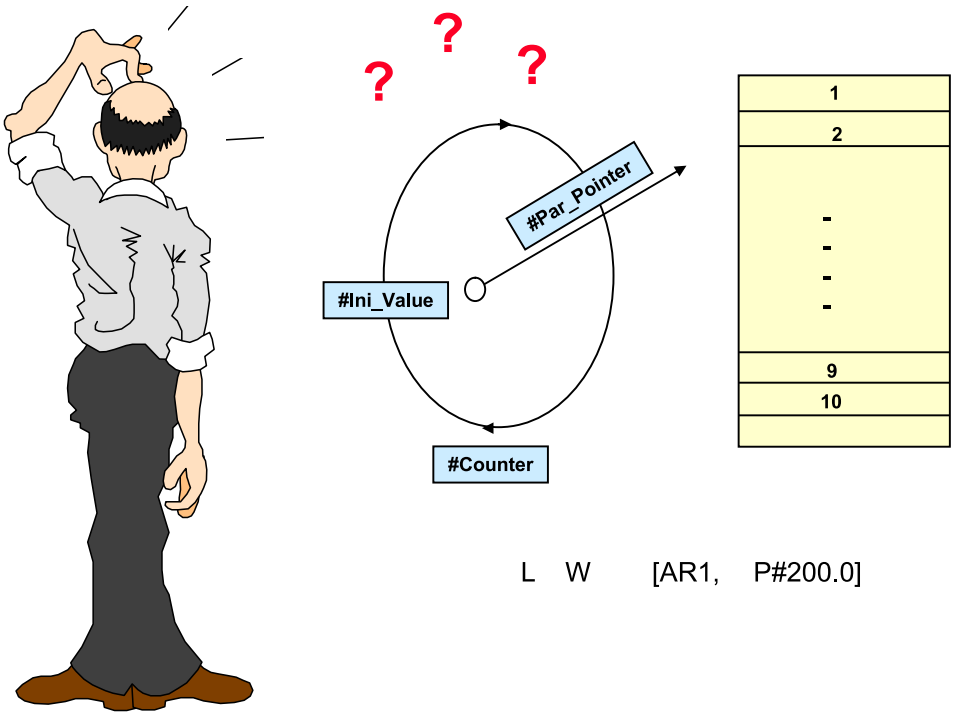
What to Do

1. Create an FC31 with the above functionality
2. Call FC31 in OB1 and connect the input and output parameters as follows:
   X1 = MD0, Y1 = MD4
   X2 = MD8, Y2 = MD12
   RET_VAL = MD16
3. Download the program into the S7-CPU.
4. Test FC31 with the help of "Monitor/Modify Variable".

Additional Task

Create a run-time optimal version of FC31 for S7-400 that can manage without the use of temporary variables.

Indirect Addressing and Address Register Instructions

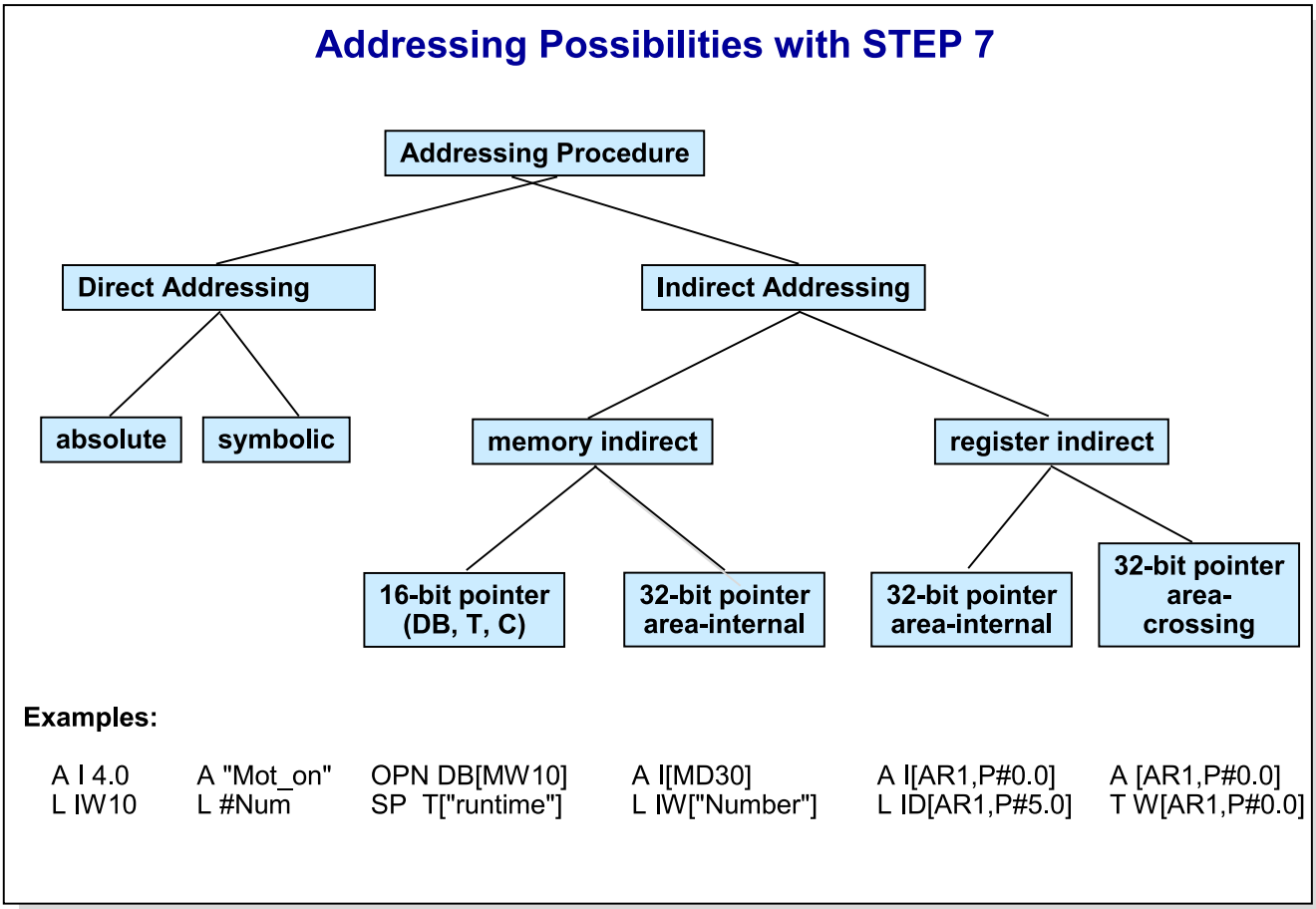


SIMATIC S7

Date: 30.06.2011
File: PRO2_04E.1

SITRAIN Training for Automation and Drives

Contents **Page**



Direct Addressing

With direct addressing, the memory location is coded in the instruction. That is, the address identifier specifies the value's address which the instruction is to process.

Symbolic Addressing

In a control program, addresses can be addressed absolute (I 1.0, for example) or symbolic ("start signal" for example). The symbolic address uses names instead of the absolute addresses.

A program is easier to read when meaningful names are used. With symbolic addressing differentiation is made between local symbols (in the declaration part of a block) and global symbols (symbol table).

Indirect Addressing

With indirect addressing you are able to address address identifiers whose address is only determined at program runtime. With indirect addressing, program parts, for example, can be scanned repeatedly (loop programming), whereby the addresses used are assigned different addresses with every scan.

With indirect addressing, differentiation is made between:

- memory indirect addressing: A pointer to the addressed address is found in a user memory's memory cell (MD30 for example).

  With memory indirect addressing, the variables, in the memory in which the pointer of the addressed address identifier is stored, can also be assigned symbolic names.
- register indirect addressing: The pointer to the addressed address is loaded in one of the two address registers (AR1 or AR2) of the S7 processor before being accessed.

Caution

Since, with indirect addressing, the addresses are only calculated at run-time, there exists the danger that unintentionally, memory areas are overwritten and thus an unexpected reaction from the PLC occurs.

Direct Addressing of Variables

Address	Memory Location (for example)	Additional Access Widths	Meaning
I	37.4	Byte, word, double word	Inputs
Q	27.7	Byte, word, double word	Outputs
PIB	655	Byte, word, double word	Peripheral inputs
PQB	653	Byte, word, double word	Peripheral outputs
M	55.0	Byte, word, double word	Bit memories
T	114	--	Timers
C	13	--	Counters
DBX	2001.6	Byte (DBB), word (DBW), double word (DBD)	Data addressed via DB register
DIX	406.1	Byte (DIB), word (DIW), double word (DID)	Data addressed via DI register
L	88.5	Byte (LB), word (LW), double word (LD)	Local data stack

Date: 30.06.2011
File: PRO2_04E.3

Direct Addressing of Variables

With the help of direct addressing, simple (elementary) variables, that is, variables up to a length of a maximum of 4 bytes, can be addressed. Simple variables consist of:

- an address identifier (for example: "IB" for input byte)
- an exact address (memory location) (byte or bit address) within the memory area, that is determined by the address identifier.

Addresses or simple variables can also be addressed via global, symbolic names (symbol table).

Peripherals

Unlike S5, it is now necessary to make a distinction between inputs and outputs when accessing peripherals. However, it is still only possible to have a read-only access (L PIW) of peripheral inputs and a write-only access (T PQW) of peripheral outputs.

Local Data

With STEP 7 it is also possible to have an absolute access of the local data stack's own blocks, for example:

- A L 12.6 (scan local data bit with address 12.6 for signal state = 1)
- L LW 12 (Load local data word in ACCU1)

DBX/DIX

You can also access directly simple variables within data blocks:

- A DBX 12.6 (Scan data bit with address 12.6 of a DB for signal state = 1, DB must be opened beforehand).
- L DB5.DBW10 (Load DW10 in DB5)

Complex Variables

You can access local variables that have a complex data type, such as structures or arrays, symbolically.

Absolute access is only possible with components of complex variables, that for their part are elementary data types.

Direct Addressing of Address Identifiers in DBs

Open data blocks	Load and transfer in data blocks	
OPN DB 19		
OPN "Values"		
	L DBB 1	Load data byte 1
	L DBW 2	Load data word 2 (byte 2/3)
	L 5	Load number 5
	T DBW 4	Transfer into word 4
OPN DI 20		
	L 'A'	Load ASCII character A
	L DIB28	Load data byte 28
	==I	Compare
	A DBX 0.0	Scan bit 0 from byte 0
Combined instruction (contains OPN DB..)	L DB19.DBW4	Load data word 4 from DB 19
	L "Values".Number_1	Symbolic access of variable Number_1. DB19 has the symbol name "Values"
	A DB10.DBX4.7	Scan bit 7 from byte 4 of DB 10

Overview

The CPU makes two data block registers available for data address processing. The numbers of the data blocks currently open are found in these registers.

Before you can access a data block, you must first open it using one of the two data block registers.

You can open a data block using the following instructions:

- OPN DBx or OPN DIx

or with the help of the combined addressing of the DB address identifier, such as:

- L DBx.DBWy (L DIx.DIWy is not possible!)

In this case the DB number x is also loaded in the DB register.

Addressing

Data blocks are organized byte-by-byte in STEP7. For the direct access of addresses with the length BIT, BYTE, WORD or DWORD, the byte address (as for I/Q/M) is given in each case.

Symbolic Access

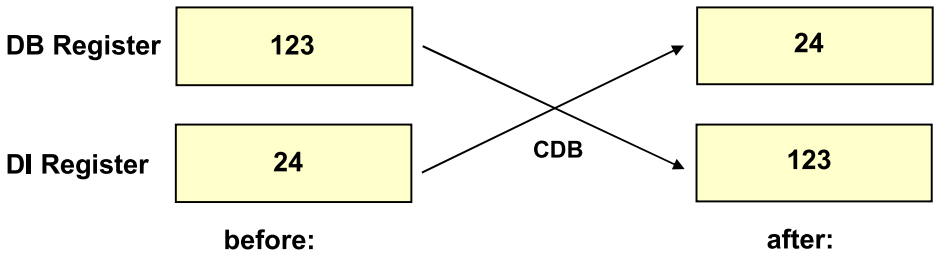
For a symbolic access you enter the data block's symbol name in the symbol list. You assign symbolic names to the data block's individual variables using the DB Editor.

Now, a complete symbolic access of a data element is possible with the instruction L "Values"..Number_1. With this, DB19 is opened ("Values" is the symbol name of DB 19) and DW 2 is loaded (Number_1 is the symbol name of DW2).

Analyzing DB Information in the Program

Instructions with DB Registers:

- **CDB: Exchange DB Registers**



- **Load DB Register in ACCU1**
  - L DBNO (load opened DB number in ACCU1)
  - L DINO (load opened DI number in ACCU1)
- **Load length of data blocks**
  - L DBLG (load the length/bytes of the opened DB in ACCU1)
  - L DILG (load the length/bytes of the opened DI in ACCU1)

DB, DI Registers

These registers contain the current valid numbers of the opened data blocks. Two data blocks can be open simultaneously in a call level.

STL prefers to use the first DB register for accessing shared (global) DBs and the second DB register for accessing instance DBs. These registers are also called DB register or DI register for this reason .

The CPU treats these registers equally. Every data block can be opened using one of these two registers (even via both simultaneously).

CDB

CDB (Exchange DB registers) exchanges the contents of the DB and DI registers. The DB register's contents are transferred into the DI register and vice versa. This instruction affects neither the contents of ACCU1 nor the status bits.

L DBLG, L DILG:

These instructions read out the data length in number of bytes of the currently opened data blocks. With the help of this information, the user program can test if a DB has the necessary length before the DB is accessed.

L DBNO, L DINO:

These instructions read out the numbers of the currently opened data blocks

Memory Indirect Addressing

- **16-bit Pointer in Word Format (Addressing of DBs, T, C)**

```
L 11
T MW 60

OPN DB[MW 60]   ⇒   OPN DB 11
```

- **32-bit Pointer in Double Word Format (Addressing of I, Q, M, ...)**

```
L   P#24.0
T   MD 50

L   I     W     [MD50]   ⇒   L   IW 24
      ↑     ↑       ↑
    Area  Access  Address
          width
```

SIMATIC S7

Date: 30.06.2011
File: PRO2_04E.6

SITRAIN Training for Automation and Drives

Overview

With memory indirect addressing, the address of the variable that is to be accessed is found in an address (memory location).

Program statements that use memory indirect addressing contain:

- an instruction (for example: OPN, A, L, etc.)
- an address identifier (DB, C, T, I, QW, MD, etc.)
- and a [variable], that must be enclosed within square brackets. This variable contains the address (pointer) of the operand which the instruction accesses.

Depending on the address identifier used, the instruction will interpret data stored in the specified [variables], as either a word or double word pointer.

Instructions with 16-bit Pointers

You use the 16-bit pointer for addressing timers, counters or blocks (DB, FC, FB).

All timer and counter instructions can be addressed using indirect addressing. To address timers, counters or blocks use area identifiers of the form T, C, DB, DI, FB, FC. The address (memory location) of the addressed operand is stored in a word.

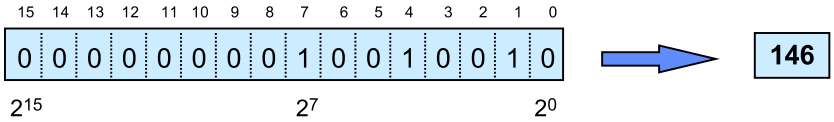
A data block can be opened using the DB as well as using the DI register. If a zero is found in the pointer when you indirectly open a data block (DB, DI), then the DB/DI register is loaded with the value "0". An error is not triggered when you load with "0".

The call of logic blocks can be indirectly addressed with the help of the instructions UC or CC (not CALL). The blocks, however, may not contain any block parameters or static variables.

This pointer in word format is interpreted as an integer number (0 ... 65 535). It refers to the number of a timer (T), a counter (C), a data block (DB, DI) or a logic block (FC, FB).

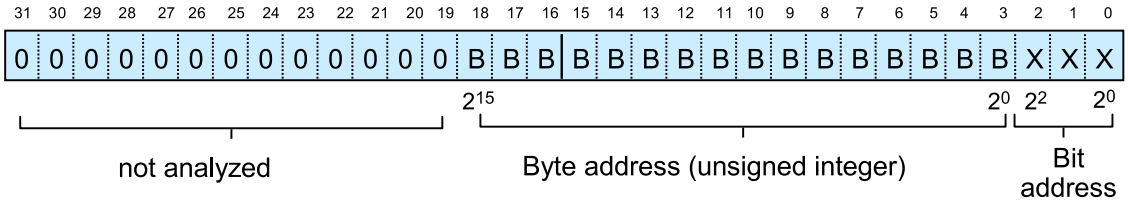
Structure of Pointers with Memory Indirect Addressing

- **Structure of a 16-bit Pointer:**



Interpretation as unsigned integer between 0 ... 65 535

- **Structure of a 32-bit Pointer (area-internal):**



- **Loading of 32-bit Pointer Constants (area-internal):**

L P#25.3 (P = Pointer, Byte address= 25, Bit address: 3)

SIMATIC S7

Date: 30.06.2011
File: PRO2_04E.7

SITRAIN Training for Automation and Drives

Instructions with 32-bit Pointers

The following addresses can be accessed with the help of memory indirect addressing using 32-bit pointers:

- Bits that are addressed by bit logic operations.
  I, Q, M, L, DIX or DBX can be used as address identifiers
- Bytes, words and double words that are addressed by load or transfer instructions. IB, IW, ID, DBB, DBW, DBD, DIB, DIW, DID, PIB, PIW, PID, can be used as address identifiers.

The address of the addressed operand is interpreted as 32-bit pointer. In this double word, the least significant bits (bit 0 to bit 2) are interpreted as bit address, the next 16 bits (bit 3 to bit 18) are interpreted as byte address of the addressed operand. Bits 19 to 31 are not analyzed by memory indirect addressing.

Note

If you want to access an address by means of memory indirect addressing using load or transfer instructions, you must make sure that the bit address of the pointer is “0”.

If this is not the case, a runtime error is triggered by the CPU during execution.

Loading of 32-bit Pointer Constants

32-bit pointer constants can be loaded into ACCU1 with the help of the following syntax:

L P#<Byte address>.<Bit address>

Storage Locations for Pointers

16-bit and 32-bit pointers for memory indirect addressing must be stored in one of the following areas:

- M - Bit memory
- L - Local data
- D - Data block (DB or DI)

Special Features of Memory Indirect Addressing

Address Areas for Storing 16-bit and 32-bit Pointers:

- Bit Memories (addressed absolute or symbolic,
  for example: *OPN DB[MW30]*, *OPN DI["Motor_1"]*, etc.
  A I[MD30], *T QD["Speed_1"]*, etc.)
- Local Data Stack (addressed absolute or symbolic,
  for example: *OPN DB[LW10]*, *OPN DI[#DB_NO*], etc.
  A I[LD10], *T QD[#Par_Pointer]*, etc.)
- Global (Shared) Data Block (can only be addressed absolute, DB must be opened beforehand,
  for example: *OPN DB[DBW0]* (overwrites DB register !!!), OPN *DI[DBW22]*, etc.
  A I[DBD10], *T QD[DBD22]*, etc.)
- Instance Data Block (can only be addressed absolute, DI must be opened beforehand,
  for example: *OPN DB[DIW20]*, *OPN DI[DIW0]* (overwrites DI register !!!), etc.
  A I[DID10], *T QD[DID22]*, etc.)

Characteristics in the Passing of Pointers to FBs and FCs

- Pointers passed in parameters cannot be used directly for memory indirect addressing.
- Pointers passed for memory indirect addressing must be copied into temporary variables before accessing.

Address Areas for Pointers

With memory indirect addressing, the address (memory location) is found in a 16-bit or a 32-bit address. This address can be found in one of the following areas:

- Bit memory: as absolute addressed operand or as variable symbolically addressed via the symbol table.
- Local data stack: as absolute addressed operand or as temporary variable declared in the block's declaration section.
- Global (Shared) data block: as absolute addressed operand. When global (shared) DBs are used as the storage location for pointers, it must be taken into consideration, that the "correct" data block has been opened using the DB register (OPN DBn, for example) before accessing.
- Instance data block: as absolute addressed operand. When instance data are used, the following points are to be observed:

  OBs and functions: Within functions or OBs, a pointer that is stored in an instance data block, can be used exactly as if it were stored in a global (shared) DB. It must merely be remembered that instead of the DB register, the DI register is now used.

  FBs: Within FBs, instance data, that is parameters or static variables, cannot generally be used symbolically for memory indirect addressing.

  Absolute access of local data within an FB is, in principle, possible using the "address" entered in the declaration section. However, when the FB is used as multi instance, it must be observed that this address is not the absolute address specified in the instance DB but actually the address relative to AR2.

Note

When you pass pointers for memory indirect addressing to blocks or want to keep the value permanently in static variables, then you must copy the pointer value from the parameter or static variable into a temporary variable and then complete the access using this temporary variable.

.

Example of Indirect Addressing

```
FC30: Example of indirect addressing

Network 1:    Open DB with indirect addressing

      L    #dbnumber            // Copy DB number in MW100
      T    MW   100             //
      OPN  DB[MW  100]          // Open DB


Network 2:    Loop for deleting

      L    P#18.0               // Store end address (DBW18) as Pointer
      T    MD   40              // in MD 40;
      L    10                   // Preset loop counter to 10
next: T    MB   50              // and transfer into MB 50;
      L    0                    // Load initialization value
      T    DBW[MD  40]          // and transfer into the DB;
      L    MD   40              // Load Pointer
      L    P#2.0                // and decrease by 2 bytes
      -D                        // and then transfer back
      T    MD   40              // to MD 40;
      L    MB   50              // Load loop counter
      LOOP next                 // decrease and if necessary jump;
```


Description

This example shows a function that initializes the inputs of a data block with the value "0". The DB number is passed to the function in an input parameter.

The data block that is addressed is first of all opened in network 1. For that, the block number that is passed (input parameter: #dbnumber) is copied into a memory word (MW100) and then the DB is opened using this memory word.

In network 2, the DB's first 10 data words are set to "0" via a loop. The loop uses the LOOP instruction, whereby the loop counter is stored in MB50.

The transferring of the value "0" into the DB's individual data words takes place with the help of memory indirect addressing via MD 40.

Before entering the loop, a pointer with the address of the last data word (DBW 18) is loaded into MD 40. With every loop scan, the access address in MD40 is decreased by P#2.0, since the values are transferred word by word not byte by byte into the DB.

Notes

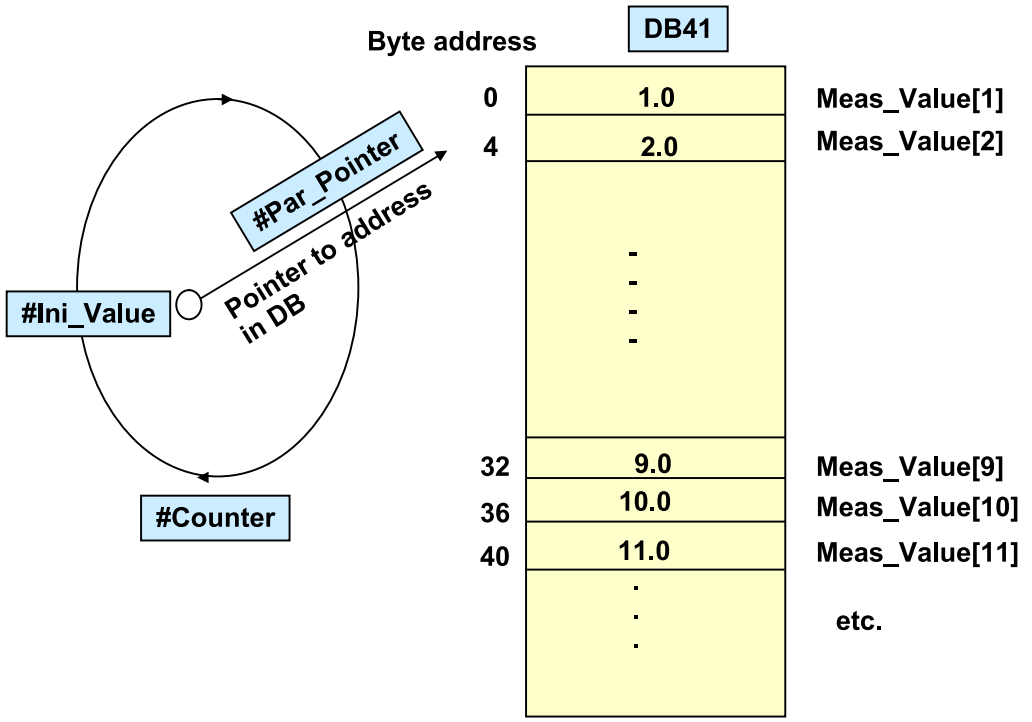
In order to keep the example short, the checking of the data block number was not done.

In practice, it would also make sense, to design the initial address and the length of the area preset with "0" as 'parameterizable' and to check before opening the DB, if the DB even exists with the necessary length.

An additional weakness in the above example is that all memory indirect accesses are made via pointers in the bit memory address area.

What would be the better alternative here? Why?

Exercise 4.1: Loop Programming with Indirect Addressing



Goal of the Exercise

You are to become familiar with the use of memory indirect addressing in a loop by doing a practical example.

Task

Memory indirect addressing is used for the programming of a loop. With it, 100 successive memory cells are described in ascending order with the values 1.0 to 100.0.

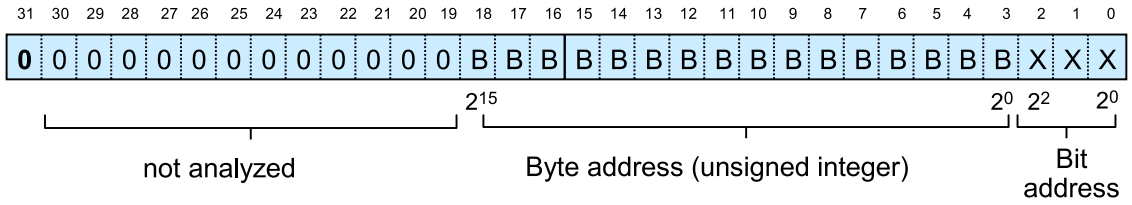
1. Create an FC41 and a DB41.
2. In the declaration part of DB41, define a variable *#Meas_Value* of the ARRAY[1..100] type with the component type REAL.
3. In the declaration part of FC41, define an input parameter *#DB_Num* of the WORD type and four temporary variables *#L_Counter* of the INT type, *#Ini_Value* of the REAL type, *#I_DB_Num* of the WORD type as well as *#Par_Pointer* of the DWORD type.
4. Within FC41, first of all open the data block whose number was passed using *#DB_Num*. Use the temporary variable *#I_DB_NUM* for this.
5. Then preset the fields *#Meas_Value*[1] to *#Meas_Value*[100] in DB41 in ascending order with the numbers 1.0 to 100.0.

   Use the loop programming for this (Instruction: LOOP):

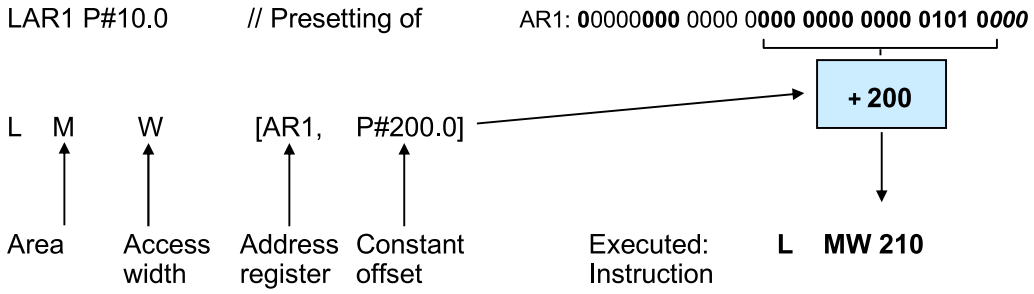
   - Save the counter for the loop scan in the variable *#L_Counter* and the initialization values for the individual components of *Meas_Value*[..] in the variable *#Ini_Value*.
   - Use memory indirect addressing for addressing the individual components of *#Meas_Value*[..] . Save the address for accessing in the variable *#Par_Pointer*.
6. Call FC41 in OB1 and assign parameters to the input parameter *#DB_Num* accordingly. Then download the blocks to the CPU and test your program.

Area-Internal, Register Indirect Addressing

- **Area-internal Pointer in AR 1 or AR2:**



- **Command Syntax:**



Overview

With register indirect, area-internal addressing, the address (memory location) of the operand which is to be accessed is found in one of the two address registers (AR1, AR2).

The address registers' contents are, in this case, an area-internal 32-bit pointer with the same configuration and the same meaning as with memory indirect addressing.

Syntax

With register indirect, area-internal addressing, the entire instruction consists of:

- an instruction (for example: A, L, T, etc.)
- an address identifier (I, MB, QD, etc.), that is the combination of an area identifier (I, Q, M, DB, DI, etc.) and an identifier for the access width (B=Byte, W=WORD, D=DWORD).
- and the declaration of an address register, that along with a constant offset must be enclosed within square brackets. This offset is added to the contents of the specified address register before the instruction is executed.

  The address register's contents and the offset have the format of area-internal pointers consisting of byte address and bit address.

  The offset's declaration (even P#0.0) in the command syntax is imperative.

Notes

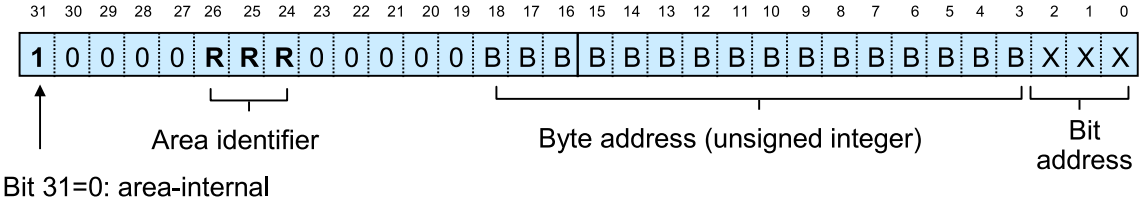
- With indirect addressed byte, word or double word addresses, the entire offset must have the bit address "0", otherwise a runtime error is triggered by the CPU during execution of the instruction.
- If the address register AR1 or AR2 specified in the register indirect, area-internal addressing contains an area-crossing pointer (see next page), then the pointer's area identifier is not evaluated during execution of the instruction.

  The area identifier in the address identifier is valid.

.

Area-Crossing, Register Indirect Addressing

- **Area-crossing Pointer in AR 1 or AR2:**

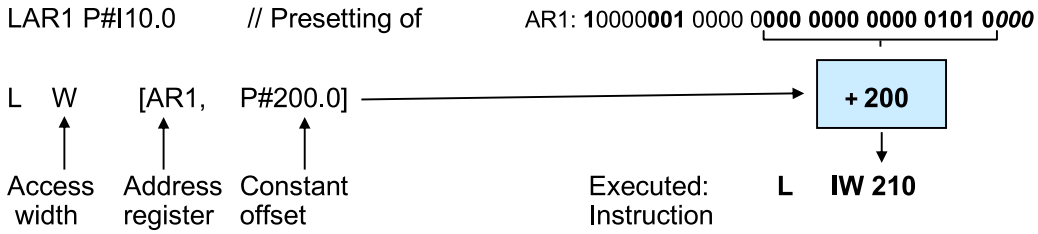


Bit 31=0: area-internal
Bit 31=1: area-crossing

- **Area Identifiers:**

000	I/O	001	Inputs (PII)
010	Outputs (PIQ)	011	Bit memories
100	Data in DB Register	101	Data in DB Register 2 (DI)
110	own local data	111	LD of calling block

- **Command Syntax:**



Date: 30.06.2011
File: PRO2_04E.12

Overview

With register indirect, area-crossing addressing, the area identifier (I, Q, M, etc.) and the address (memory location) (byte.bit address) of the operand which is to be accessed is found as area-crossing pointer in one of the two address registers (AR1, AR2).

Syntax

With register indirect, area-crossing addressing, the entire instruction consists of:

- an instruction (for example: A, L, T, etc.)
- an identifier for the access width (B=BYTE, W=WORD, D=DWORD).
- and the declaration of an address register, that along with a constant offset must be enclosed within square brackets.

  The address register's contents must be an area-crossing pointer with area identifier and byte.bit address in this case.

  The offset has the format of an area-internal pointer consisting of byte and bit address that is added to the pointer's byte.bit address in the declared address register before the instruction is executed.

  The offset's declaration (even P#0.0) in the command syntax is imperative.

Notes

- With indirect addressed byte, word or double word addresses, the entire offset must have the bit address "0", otherwise a runtime error is triggered by the CPU during execution of the instruction.
- For CPUs that have older firmware versions, access to their own local data (identifier: 110) is not possible using indirect area-crossing addressing.

  In these cases, the runtime error "unknown area identifier" is triggered. Access to their own local data is then only possible with area-internal addressing.

Instructions for Loading Address Registers

Loading the Address Register

- LARn (n =1 or 2): Load contents of ACCU1 in ARn
- LARn <Address> Load contents of <Address> in ARn
- LARn P#<Address> Load address of <Address> in ARn

<Address>:

- Processor register: AR1, AR2 (e.g. *LAR1 AR2* and *LAR2 AR1*)
- 32-bit variables as of: MDn, LDn, DBDn, DIDn (e.g. *L DBD5*, etc.)
- symbol. 32-bit variables: (global and local) 32-bit global variables (e.g. *LAR1 "Index"*, etc.) and TEMP variables of OBs, FBs and FCs (for example, *LAR1 #Address*, etc.)

P#<Address>

- Pointer to absolute boolean addresses: En.m, An.m, Mn.m, Ln.m, DBXn.m, DIXn.m (for example, *LAR1 P#M5.3*, *LAR2 P#I3.6*, etc.)
- Pointer to local, symb. addresses: OB: TEMP variables (e.g.: *LAR1 P##Par_Pointer*, etc.) FB: IN-, OUT-, INOUT-, STAT- and TEMP- variabl. FC: TEMP variables (*LAR1 P##Loop*, etc.)

SIMATIC S7

Date: 30.06.2011
File: PRO2_04E.13

SITRAIN Training for Automation and Drives

Loading Operands

With the help of load functions, it is possible to initialize the address registers with new values.

The load function LARn (n=1, 2) loads the pointer into the address register ARn. ACCU1 or ARn, or, a double word from the address areas bit memory, temporary local data, global (shared) data and instance data can be used as the source. Access can take place absolute or symbolic.

If you do not specify an address, the contents of ACCU1 are automatically loaded into the address register ARn. The contents of the register that is loaded or the contents of the double word must correspond to the format of an area pointer.

Loading Pointers

Direct pointers (addresses) can naturally also be loaded in addresses in the address registers.

With the help of the instruction:

- L P#<area identifier>n.m

you can load an area-crossing pointer directly into the specified address register.

An area-crossing pointer to a variable or to a parameter (*#Address)* can be loaded into one of the two address registers with the following instruction, for example

- LARn P##*Address* (n=1, 2)

This access is possible to all TEMP variables of OBs, FBs and FCs, as well as to IN, OUT, INOUT and STAT variables of FBs.

Note

If you want to load a pointer to an IN, OUT and INOUT parameter (*#Param*) of an FC into the address register, it is not possible to do so in a direct way. An intermediate step must be taken:

- L P##*Param* (Load pointer to parameter *#Param* in ACCU1)
  LARn (Load contents from ACCU1 in ARn)

Other Instructions with Address Registers

Transferring from an Address Register

- TARn (n =1 or 2): Transfer contents from ARn to ACCU1
- TARn <Address> Transfer contents from ARn to <Address>

<Address>:

- Processor register: AR2 (e.g. *TAR1 AR2*)
- 32-bit variables absolute: MDn, LDn, DBDn, DIDn (e.g. *TAR2 MD5*, etc.)
- symbol. 32-bit variables: (global and local) 32-bit global variables (e.g. *TAR1 "Index"*, etc.) and TEMP variables of OBs, FBs and FCs (e.g. TAR1 #Address, etc.)

Toggle (Swap) Address Registers

- TAR Swap contents of address register AR1 and AR2

Adding to Address Register

- +ARn Add ACCU1-L to ARn
- +ARn P#n.m Add area-internal pointer P#n.m to AR1 or AR2

SIMATIC S7

Date: 30.06.2011
File: PRO2_04E.14

SITRAIN Training for Automation and Drives

Transferring from Address Registers

The TARn instruction transfers the complete area pointer from the address register ARn. The other address register or a double word from the bit memory, temporary local data, global (shared) data and instance data address areas can be specified as destination.

If no address is specified, TARn transfers the address register's contents into ACCU1. The previous contents of ACCU1 are shifted into ACCU2; the contents of ACCU2 are lost.

The contents of ACCU3 and ACCU4 (S7-400) remain unchanged.

Swapping Address Registers

The TAR instruction swaps the contents of address register AR1 and AR2.

Adding to Address Registers

A value can be added to the address registers in order to increase the address of an address with every loop scan in a program loop, for example. The value can be specified as either a constant (area-internal pointer) with the instruction or as the content of the right word in ACCU1-L .

The +AR1 and +AR2 instructions interpret the value found in ACCU1 as number in the INT format, expand it to 24 bits with its correct sign and add it to the address register's contents. In this way a pointer can also be made smaller. An overrange or an underrange of the maximum area of the byte address (0 to 65 535) has no further impact; the upper bits are simply "cut off".

The +ARn P#n.m instruction adds an area-internal pointer to the specified address register. The area pointer P#n.m can have a maximum size of P#4095.7.

None of the addressed instructions above or on the previous page change the bits in the status word.

Special Features of Register Indirect Addressing

Internal Use of AR1 by STL/LAD/FBD Editor

- With accesses to parameters in FCs, the **AR1 register** and the **DB register** are overwritten if the parameters are of the complex data type (ARRAY, STRUCT, DATE_AND_TIME) .
- With accesses to INOUT parameters of FBs, the **AR1 register** and the **DB register** are overwritten if the INOUT parameter is of the complex data type (ARRAY, STRUCT, DATE_AND_TIME)

No access to the local parameters may take place between the loading of the address register and the register indirect access of the desired variable

Internal Use of AR2 by STL/LAD/FBD Editor

- The **AR2 register** and the **DI register** are used as basis address register for the addressing of all parameters and STAT variables within an **FB.**

If **AR2** or **DI** are overwritten by the user within an FB, no access to the FB‘s own parameters or STAT variables can take place after that. That is, not without restoring both registers.

- No restriction within FCs with reference to the AR2 register and DI register

SIMATIC S7

Date: 30.06.2011
File: PRO2_04E.15

SITRAIN Training for Automation and Drives

Address Register AR1

The STEP7 Editor uses the address register AR1 to access complex block parameters. Within functions, the registers AR1 and DB are overwritten when all block parameters of the "ARRAY" or "STRUCT" type are symbolically accessed.

As well, the registers AR1 and DB are overwritten when in/out parameters of the "ARRAY" or "STRUCT" type are accessed within an FB.

Symbolic accesses to an FB’s or FC’s temporary variable overwrite neither the AR1 nor the DB register.

Address Register AR2

The STEP7 Editor uses the area-internal register indirect addressing for the symbolic access to instance data, that is, of all parameters and of an FB’s static variables. The DI register contains the number of the instance data block and the AR2 register the respective address offset of the instance data area within the instance data block.

No access to the instance data can take place after these DI and AR2 registers are overwritten, if the contents of both these registers are not restored. If you want to use the AR2 or DI register within an FB for your own purposes, then the following procedure is recommended :

1. Save the contents of DI and AR2 in variables of the DWORD type:

```
TAR2 #AR2_REG    // Save AR2 in temporary variable #AR2_REG
L DINO           // Load contents of DI in ACCU1
T #DI_REG        // Save in temporary variable #DI_REG
```

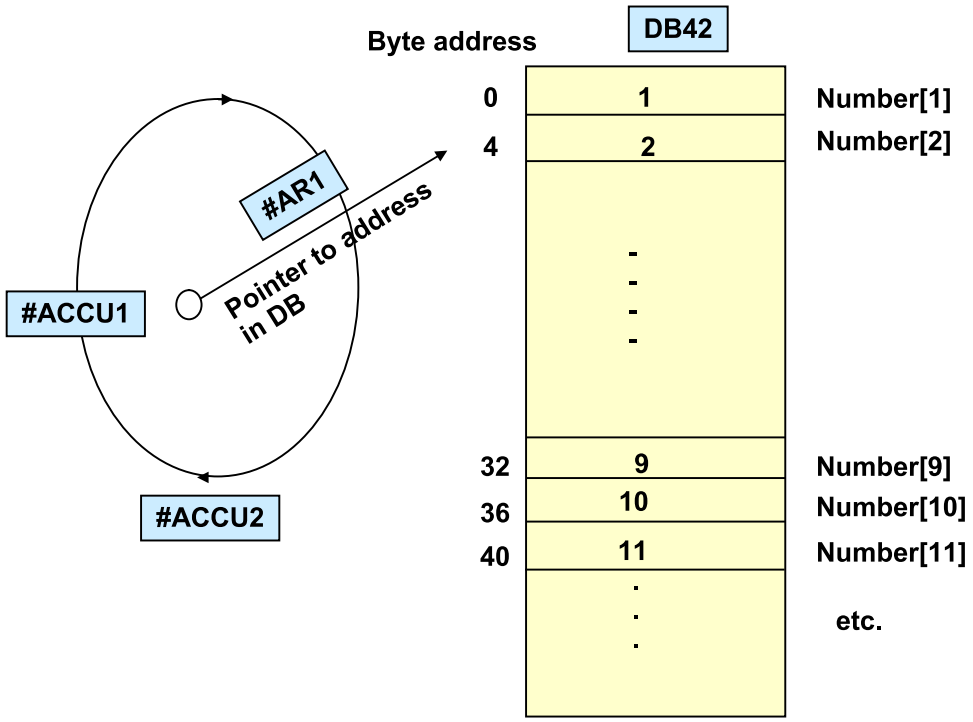
2. Use the DI and AR2 register for your own purposes. No access to the FB’s parameters or static variables can take place during this segment.
3. Restore the original contents of the DI and the AR2 register:

```
LAR2 #AR2_REG    // Load AR2 with contents of #AR2_REG
OPN DI[#DI_REG]  // Restore DI register
```

The FB’s parameters and static variables can once again be accessed symbolically.

.

Exercise 4.2: Loop Programming with Register Indirect Addressing



SIMATIC S7

Date: 30.06.2011
File: PRO2_04E.16

SITRAIN Training for Automation and Drives

Goal of the Exercise

You become familiar with the use of register indirect addressing in a loop by doing a practical example.

Task

Register indirect addressing is used for the programming of a loop. With it, the values 1 to 100 are written in 100 successive memory cells.

Create a time optimal solution (without additional temporary variables) of exercise 4.1. Save the values for the loop counter and the initialization value in the accumulators.

For addressing the components of #*Number*[..], use the address register AR1 (area-internal, register indirect addressing).

What to Do

1. Create an FC42 and a DB42.
2. In the declaration part of DB42, define a variable #*Number* of the type ARRAY[1..100] with the component type DINT.
3. In the declaration part of FC42, define an input parameter #*DB_Num* of the type WORD and a temporary variable #*I_DB_Num* of the type WORD.
4. Within FC42, first of all open the data block, whose number was passed using #*DB_Num*. Use the temporary variable #*I_DB_Num* for this.
5. Then preset the fields #*Number[1]* to #*Number[100]* in DB42 in ascending order with the numbers 1.0 to 100.0.
   - Use the loop programming for this (Instruction: LOOP):
   - Use register indirect addressing with AR1 for addressing the individual components of #*Number*[..] .
6. Call FC42 in OB1 and assign parameters to the input parameter #*DB_Num* accordingly. Then download the blocks to the CPU and test your program.

Block Parameters of the POINTER and ANY Data Type

FC...

Address	Decl.	Name	Type
0.0	in	DB_Num	WORD
2.0	in	Area_Pointer	DWORD
6.0	in	Length	WORD
. . .			

```
L     #DB_Num
T     #I_DB_Num
OPN   DB[I_DB_Num]
.
L     #Area_Pointer
LAR1
L     #Length
...
```

FC...

Address	Decl.	Name	Type
0.0	in	Measure_1	POINTER
. . .			
8.0	in	Measure_2	ANY
. . .			

```
L     P##Measure_1
LAR1
L     W[AR1,P#0.0]
T     #I_DB_Num
OPN   DB[I_DB_Num]
L     D[AR1,P#2.0]
LAR1
....
```

Passing the individual information of an actual address (as in STEP 5):

- DB number
- initial address
- area length, etc.

Drawbacks:

- the programmerer must do the splitting up
- symbolic name of the actual address cannot be used during the call
- several parameters are necessary for one address (area)
- is not corrected in the consistency check

Passing using parameters of the POINTER or ANY type (STEP 7):

- the programmerer can use the symbolic name of the actual address in the call
- splitting up into individual information is done by the LAD/FBD/STL Editor
- only one parameter is required for one address
- is corrected in the consistency check

SIMATIC S7

Date: 30.06.2011
File: PRO2_04E.17

SITRAIN Training for Automation and Drives

Pointer Types in STEP7

Besides the pointer types described in the previous section (16-bit, 32-bit area-internal and 32-bit area-crossing), STEP7 recognizes two additional pointer types, the:

- 48-bit pointer ("POINTER " data type)
- 80-bit pointer ("ANY" data type)

The 16 and 32-bit pointer types can be loaded directly into the accumulator or in an address register and thus can be used for indirect addressing within a block.

The POINTER and ANY pointer types (larger than 32 bit) cannot be loaded directly into registers and used for indirect addressing within a block. They are used exclusively for the complete addressing of actual parameters in the passing to formal parameters of called blocks.

For example, you can declare a parameter of the POINTER or ANY data type within a block and during the block call assign the parameter with the address of an actual parameter.

POINTER

The use of POINTER data type parameters are only of minor importance to the user. It is mainly used by the STL/LAD/FBD Editor, to pass an actual parameter of the complex data type, such as ARRAY, STRUCT, and DT, to the called FB or FC.

Since the STL/LAD/FBD Editor immediately checks the correctness of the data type and the length with the assignment of the actual parameter, it suffices to merely pass the complete initial address of the actual parameter internally.

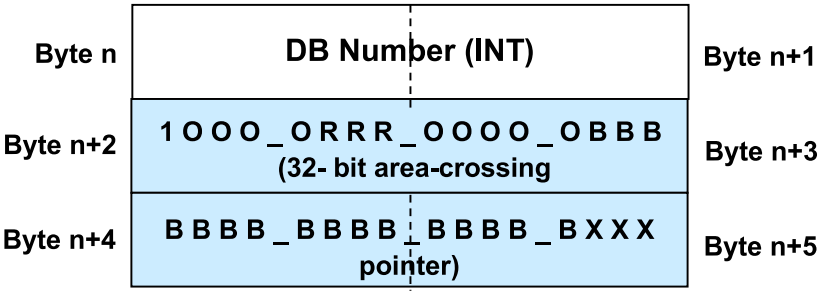
Within a called block, you can then access the actual parameters register indirect using this POINTER.

ANY

The ANY pointer type is mainly used by STEP7 for assigning the parameters of system functions (SFCs) and system function blocks (SFBs). Parameters of the ANY data type can also be used by the user, to generate powerful blocks.

Structure and Assignment of the POINTER Data Type

Structure of the Data Type: POINTER



Assignment of a Parameter of the POINTER Type

- **Pointer display**
  P#DBn.DBX x.y with: n= DB number, x= byte number, y= bit number
  P#DIn .DIX x.y (e.g.: P#DB5.DBX3.4, P#DI2.DIX10.0, etc.)
  P#Zx.y with: Z= area, e.g.: P, I, Q, M and L
  (e.g.: P#I5.3, P#M10.0, etc.)
- **Declaration of address:**
  MD30 (in this case, DB number, area identifier and
  #Motor_on bit address are automatically entered in the
  "Motor_1".speed POINTER)

SIMATIC S7

Date: 30.06.2011
File: PRO2_04E.18
SITRAIN Training for Automation and Drives

POINTER Data Type

In addition to the area-crossing pointer, a parameter of the POINTER data type also contains a data block number as positive unsigned integer (value area: 0 to
65 535). It enters the data block's number when the area-crossing pointer references the global data or instance data address areas.

In all other cases, when the addressed address is found in other areas (P, I, Q, M, L), "0" is entered in the first two bytes of the "POINTER".

Parameter Assignment

If, during a block call (FC or FB), a parameter of the POINTER data type must be assigned, this can take place using the pointer display or by declaring the address.

Pointer Display

In this case, a pointer (P#...) must be entered on the first bit of the address, such as:

- P#DB10.DBX2.0 //Data bit 2.0 in DB10, area identifier DB
- P#I5.3 //I 5.3, DB number = 0, area identifier I

Declaration of Address

In this case, the declaration of address is sufficient (without P#...). The address can be entered absolute, that is, using DB number, address identifier and associated byte or bit address, such as:

- DB5.DBW10 //Bit 10.0, DB number = 5, area identifier DB

or symbolic

- #Motor_on, "Motor_1".speed

In both cases, the STL/LAD/FBD Editor defines the associated DB number, the area identifier and byte.bit address of the address and enters these in the "POINTER".

Configuration of the ANY Data Type

- **ANY Pointer for Data Types**

Byte n	16#10	Data type
Byte n+2	Replication factor	
Byte n+4	DB number	
Byte n+6	1 O O O _ O R R R _ O O O O _ O B B B	
Byte n+8	B B B B _ B B B B _ B B B B _ B X X X	

Data Type	Identifier
VOID	00
BOOL	01
BYTE	02
CHAR	03
WORD	04
INT	05
DWORD	06
DINT	07
REAL	08
DATE	09
TOD	0A
TIME	0B
S5TIME	0C
DT	0E
STRING	13

- **ANY Pointer for Parameter Types**

Byte n	16#10	Parameter type
Byte n+2	16#0001	
Byte n+4	16#0000	
Byte n+6	16#0000	
Byte n+8	Number of the timer, counter or block	

Parameter T.	Identifier
BLOCK_FB	17
BLOCK_FC	18
BLOCK_DB	19
BLOCK_SDB	1A
COUNTER	1C
TIMER	1D

SIMATIC S7

Date: 30.06.2011
File: PRO2_04E.19

SITRAIN Training for Automation and Drives

ANY Data Type

In addition to the area-crossing pointer and the DB number, the ANY pointer contains an identifier for the data type and a replication factor. With it, it is possible to identify not only an individual address but also a complete data area.

There are two versions of the ANY pointer:

- for variables with data types: The ANY pointer then contains a syntax-ID 16#10 for STL, an identifier for the data type, a replication factor, the DB number and an area-crossing pointer.
- for variables with parameter types: In this case, the ANY pointer consists merely of the syntax-ID 16#10 for STL, an identifier for the parameter type and a 16-bit unsigned number in byte n+8 and byte n+9, that reflects the block number. The bytes n+4, ..., n+7 are filled with “0”.

Declaration of ANY Pointers

Variables of the ANY data type generally can be declared as IN, OUT and INOUT parameters in FCs and FBs.

The declaration can also be used as a temporary variable within FBs. With the help of this temporary variable it is possible to create an ANY pointer that is changeable at runtime and to pass this to a called block (see: Indirect Parameter Assignment of the ANY Type).

Area Identifier (RRR):

000	I/O	001	Inputs (PII)
010	Outputs (PIQ)	011	Bit memory
100	Data w.ref.to DB register	101	Data w.ref.to DI register
110	Own local data	111	LD of caller

Assignment of Parameters of the ANY Data Type

Pointer Display:

- **P#[Data block.]Bit address Type Number**

P#DB10.DBX12.0 REAL 20	Pointer to an area in DB10, beginning with Byte 12, consisting of 20 addresses of data type REAL (ARRAY[1..20] OF REAL)
P#I 10.0 BOOL 8	Pointer to a field of 8 bits in IB10

Declaration of Address:

- **absolute:**

DB5.DBD10	Data type: DWORD, Replication factor: 1 DB number: 5, Pointer: P#DB5.DBX10.0
IW32	Type: WORD, WF: 1, DB Nr.: 0, Pointer: P#I 32.0
T35	Type: TIMER, Nr.: 35

- **symbolic:**

#Motor_1.speed "Pump".Start	with elementary data types, the compiler establishes the correct data type, replication factor 1 and pointer

Note

with symbolic assignment (ARRAY, STRUCT, STRING, UDT), the data type identifier 02 (BYTE) and the area length in bytes is merely established by the compiler and entered in the ANY pointer

Assignment	A parameter of the "ANY" data type can be assigned using the pointer display as well as by direct declaration of address (variables).
Pointer Display	With assignment using the pointer display (for example: P#DB5.DBX10.0 INT 8) the STL/LAD/FBD Editor sets up an ANY pointer that corresponds in type and in number with the declarations. Assignment in the pointer display always makes sense when a data area is to be addressed for which no variable was defined or for example, no suitable variable (ARRAY or STRUCT, for example) can be defined (P, PII, PIQ, M, for example). In addition, the absolute pointer display must be used when the correct information about the replication factor and the data type is required within the called block (for example, ARRAY[1..8] OF REAL).
Address Display	A parameter of the "ANY" type can also be directly assigned with the address to which the ANY pointer is to point. This declaration can take place absolute or using the symbolic variable name. With the declaration of an absolute address, the STL/LAD/FBD Editor automatically determines the associated data type (BOOL, BYTE, WORD, DWORD,), a replication factor of "1", the DB number as well as the area-crossing pointer to the first bit of the address and enters these values in the pointer structure. Likewise, the STL/LAD/FBD Editor determines the correct information using the address when the declaration takes place using the symbol name and the variable that is entered is of the elementary data type.
Note	If a variable is of the complex data type (for example, ARRAY[1..8] OF REAL), then the STL/LAD/FBD Editor merely enters information in bytes about the area occupied by the variable (that is,: replication factor: 32, data type: BYTE).

Indirect Parameter Assignment of the ANY Type

Assignment by means of temp. actual parameters of the ANY data type

- **declare temporary variable of the ANY data type in the calling block**

  e.g.: `temp  aux_pointer ANY`

- **fill temporary ANY variable with pointer information**

  e.g.:

```
LAR1  P##aux_pointer        // Load address of aux. pointer
L     B#16#10               // Load identifier 10
T     LB [AR1,P#0.0]        // and transfer to Offset 0
L     ...
...
```

- **Assign block parameter of the ANY type (`target field`) with variable `auxiliary pointer`**

  e.g.:

```
CALL  FC    111
   Targetfield:=#aux_pointer
```

Advantage

- **dynamic reassignment of parameters of an ANY pointer at runtime**

SIMATIC S7

Date: 30.06.2011
File: PRO2_04E.21

SITRAIN Training for Automation and Drives

Indirect Assignment

The calling block can also assign an FC or FB parameter of the ANY data type with a temporary variable of the ANY data type. This temporary variable has been stored in the local data stack of the calling block.

In this case, the STL Editor passes no pointer to the temporary variable (in the local data stack), but assumes that this temporary ANY variable already contains the pointer to the actually desired variable.

In this case, the Editor passes the ANY pointer contained in the temporary variable to the called FC or the called FB.

Advantage

This way, you have the possibility of setting up an ANY pointer to an ANY parameter which you can change at runtime. The variable ANY pointer can be very useful, especially in connection with the system functions, such as SFC 20 (BLKMOV) or SFC 21 (FILL).

Evaluating a Passed ANY Pointer

Address	Declaration	Name	Type	Initial Value	Comment
0.0	in	Par_Pointer	ANY		
	out				
	in_out				
0.0	temp	Data_type	BYTE		
2.0	temp	WF	WORD		
4.0	temp	DB_Nr	WORD		
6.0	temp	Area_Pointer	DWORD		

```
Network 1: Establishment of data type, replication factor, DB-Nr and Be-
Pointer

      L    P##Par_Pointer       // Load address of #Pointer in ACCU1
      LAR1                      // and from there load in AR1;
      L    B [AR1,P#1.0]        // Establish data type from pointer
      T    #Data_type           // and load in temporary variable;
      L    W [AR1,P#2.0]        // Establish replication factor
      T    WF                   // and load in temporary variable;
      L    W [AR1,P#4.0]        // Establish DB number
      T    #DB_Nr               // and load in temporary variable;
      L    D [AR1,P#6.0]        // Establish area pointer
      T    #Area_Pointer        // and load in temporary variable;
```


Overview

Within the called block (FB or FC), the information that is in the pointer (POINTER or ANY type) that is passed can be read out and evaluated with the help of register indirect addressing.

Procedure

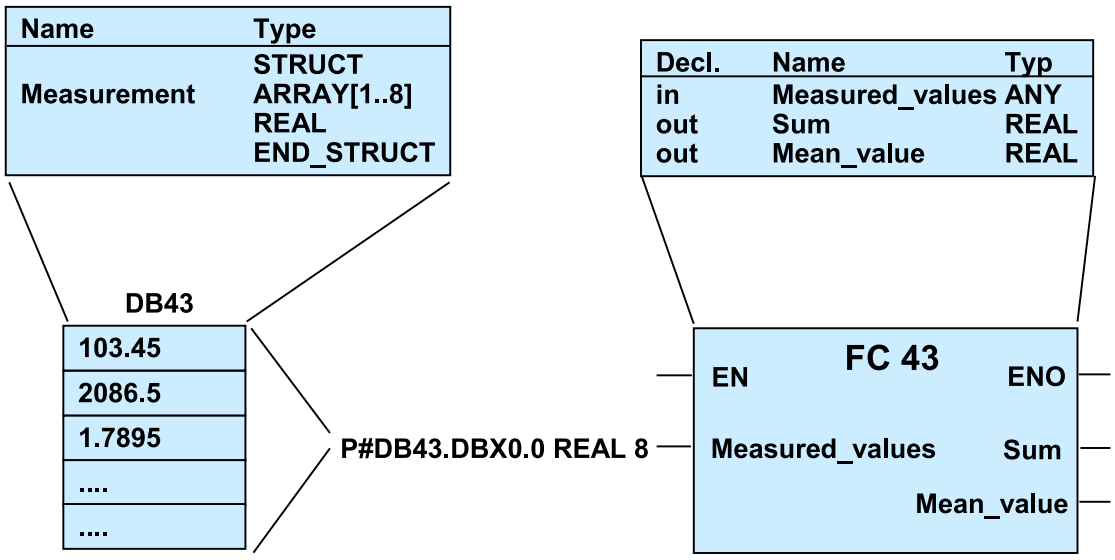
The evaluation of the information passed in an “ANY" pointer is listed in the following steps. The steps orient themselves to the above example, where an input parameter (type "ANY") with the name *#Par_Pointer* and several temporary variables for the temporary storage of information were declared.

1. First of all, an area-crossing pointer is established to the “ANY” pointer that is passed and loaded into the address register AR1. This takes place with the instruction:
   - LAR1 P##Par_Pointer //in FBs or
   - L P##Par_Pointer //in FCs, the address must first of all be loaded in
     LAR1 //ACCU1 and from there in the AR1 register

   The "ANY" pointer that is passed is, in the case of an FB, stored in the instance DB (this is automatically opened) or with an FC, in the local data stack of the caller.
2. Using register indirect addressing, the information in the "ANY" pointer that is passed can now be read out and, for example, can be temporarily stored in temporary variables of the block for further processing.
   - L B[AR1,P#1.0] //reads the identifier of the data type of the actual // parameter in ACCU1
   - L W[AR1,P#2.0] //reads the replication factor in ACCU1
   - L W[AR1,P#4.0] //reads the DB number of the DB in which the actual // parameter is stored in ACCU1, or "0" when the //actual parameter is stored in P, PII, PIQ, M, L
   - L D[AR1,P#6.0] //reads the area-crossing pointer that points to the //actual parameter into ACCU1

   The information stored in the ANY pointer about the actual parameters must then be further processed according to the task.

Exercise 4.3: Function for Calculating Sum and Mean Value



Overview

Generic FCs or FBs can be created with the help of the "ANY" data type. Generic FCs or FBs are not committed to specific data types. They can “adapt” themselves at runtime to the data types or field lengths passed to them.

Goal

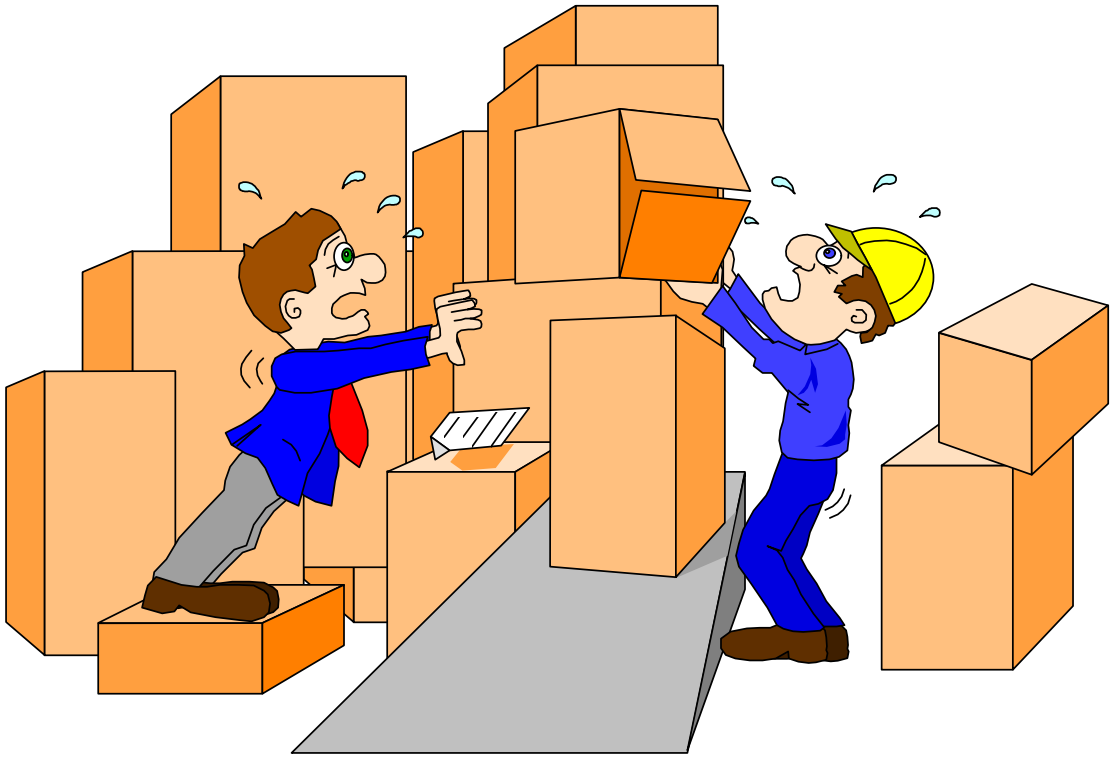
Create an FC43 with the following functionality:

- The function expects a field (array) of REAL values in the input parameter *Measured_values* (type "ANY").
- The function delivers the summed up values of the field (array) elements that are passed in the output parameter #*Sum* (type: REAL) and the mean value of all field elements in the output parameter #*Mean_value* (type: REAL).
- If another data type is passed, FC 43 ends in error (that is, BR bit=0, invalid Real number for #*Sum* and #*Mean_value*).

What to Do

1. Generate the FC43 and declare the above listed input and output parameters. Also declare corresponding temporary variables for temporary storage of information about replication factor, DB number and the actual parameter’s area pointer.
2. To begin with, read out the data type identifier from the "ANY" pointer that is passed, and exit the FC43 accordingly, if the actual parameter’s data type is not REAL.
3. In a loop (LOOP instruction), program the sum of all field (array) elements. Calculate sum and mean value and assign the results to the corresponding output parameters.
4. Create a DB43. Declare a variable *Measurement* of the type ARRAY[1..8] in DB43 and enter appropriate values in the data view.
5. Program the call of FC43 in OB1. Assign the input parameter in the pointer display. Assign addresses in the memory area to the output parameters.
6. Download the participating blocks into the CPU and test the result.

STEP 7 Data Types and Variables



SIMATIC S7

Date: 30.06.2011
File: PRO2_05E.1

SITRAIN Training for Automation and Drives

Contents **Page**

Meaning of Data Types and Variables

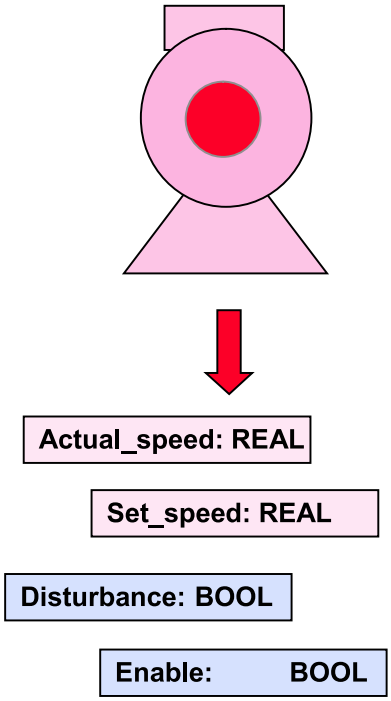
Data types characterize the basic properties of data

- continuous area: e.g. actual speed
- "yes/no" property: e.g. disturbance

The data type establishes:

- the possible range
  (INT: -32 768 ... +32 767, etc.)
- the allowed instructions
  (arithmetic instructions: +, -, etc.)
- data types abstract from the underlying representation of the bits in the memory

Variables permit you to save and later continue to process values



Overview

The modern computer systems were developed to simplify and to speed up complicated and time consuming calculations. The capability to be able to process large amounts of information, to store and to make it accessible again plays an important roll for most applications.

The information available to the controller consists of a selected amount of data about the "real world". The data represent an abstract of reality because incidental and unimportant properties of the objects concerned are not taken into account for this specific problem.

Data Types

It is often quite difficult to decide how data are to be represented. The choice is quite often also restricted by the available possibilities. On the one hand, the properties of the objects that are described by the data must be correctly reflected, on the other hand, the instructions that are necessary for process management must also be able to be carried out with the data.

The data type determines which values are accepted by data and which instructions can be carried out with these values.

The data type uniquely defines:

- the possible range
- the allowed instructions

Data types abstract also from the underlying representation (format) of the individual bits as they are finally stored in the computer memory.

Meaning of Variables

Next to commands, variables are the most important elements of a programming system. Their task is to save values in a program so that they can be further processed at a later time. The value of a variable can be saved "anywhere" in the memory of a PLC.

Properties and Declarations of Variables

The following properties are determined by the declaration of a variable:

- symbolic name
- data type
- validity range

Variables can be declared:

- in the global symbol table (elementary data type)
- in the declaration table of a global data block (all data types)
- in the declaration table of a logic block (OB, FB and FC)

Meas_point: ARRAY[1..10]

Meas_point[1]: Real
Meas_point[2]: Real
Meas_point[3]: Real
...
Meas_point[10]: Real

"Conventional" Variables

In conventional PLC programming, PLC memory addresses are accessed directly by specifying the memory area (e.g.: M=Bit memories, I=Inputs, etc.), the access width (e.g.: B=byte, W=word, etc.) and by specifying the byte/(bit) address. These memory areas addressed using addresses can be used within a program for different purposes, for example, as integer (DINT for example), as floating point number (REAL for example) or simply as a collection of individual signals (WORD for example).

Up until now it was up to the programmer to remember the format and the application purpose of the individual memory locations. Faulty programs could easily result because incorrect memory addresses or the incorrect format was unintentionally used in the instructions.

Declaring Variables

Earlier PLC systems (STEP 5 for example) permitted the use of symbols to make programs easier to read. STEP 7 goes one step further and uses variables instead of PLC addresses and symbols.

By explicitly declaring a variable, the following properties are fixed:

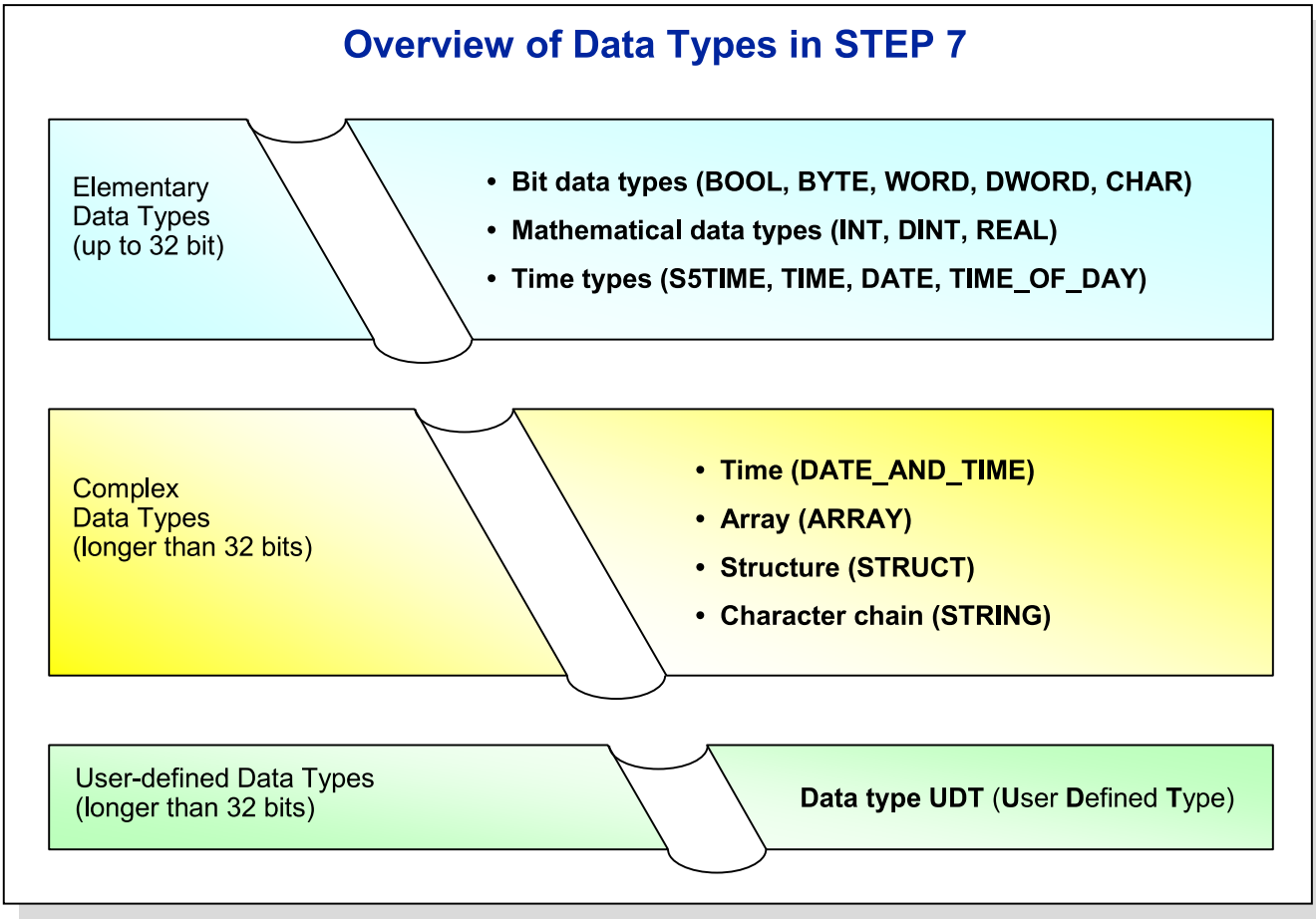
- Symbolic name of the variable
- Data type of the variable
- Validity range

If variables are declared, the program editor can then use data type information to check the admissibility of instructions such as the parameter assignment in a block call, for example.

Validity Range

Variables that are declared in the global symbol table or in a global data block can be addressed by all blocks of the program folder. These variables are called global variables for that reason.

Variables and parameters that are declared in the declaration section of a logic block are called local; they can only be used within the instruction section of the same block.



Overview

The solution of automation tasks using computer systems is based on algorithms that process the data collected by sensors in order to output new values to the actuators. Programs are basically forms of algorithms that rely on specific data or data structure representations.

Elementary Data Types

The elementary data types make up the "atoms" of every programming system. The choice of elementary data types of a programming system says a lot about the intended application area.

In STEP 7, the elementary data types are predefined in accordance with IEC 61131-3. The data types are chosen in such a way that in addition to the innate tasks of a PLC such as binary and analog signal processing, a simple signalling system as well as time-stamp management can be operated as well.

With elementary data types, the data type determines the amount of memory space that a variable requires. Elementary data types are never more than 32 bits long in STEP 7 and can be loaded into the accumulators in full and processed with STEP 7 instructions.

Complex Data Types

The basic idea of data structuring is the differentiation between elementary and higher structures. The former are the atoms from which the complex data types are configured.

In STEP 7, complex data types can only be used in conjunction with variables declared in global DBs or in the local data stack. Complex data types cannot be completely loaded into an accumulator and processed.

User-defined Data Types

Complex data types do not have their own identifiers for the data type and as a result cannot be repeatedly used for parameter or variable declarations.

With the help of user-defined data types (UDT), unique, structured data types can be created that can then be used as often as required for other variable or parameter declarations.

Elementary Data Types in STEP 7

Keyword	Length (in bits)	Example of a constant of this type
BOOL	1	1 or 0
BYTE	8	B#16#A9
WORD	16	W#16#12AF
DWORD	32	DW#16#ADAC1EF5
CHAR	8	'w'
INT	16	123
DINT	32	65539 or L#-1
REAL	32	1.2 or 34.5E-12
S5TIME	16	S5T#5s_200ms
TIME	32	T#2D_1H_3M_45S_12MS
DATE	16	D#1999-06-14
TIME-OF-DAY	32	TOD#12:23:45.12

Date: 30.06.2011
File: PRO2_05E.5

BOOL, BYTE, WORD DWORD, CHAR

Variables of the BOOL data type consist of one bit, variables of data types BYTE, WORD, DWORD are sequences of 8, 16 and 32 bits respectively. The individual bits are not evaluated in these data types.

Special forms of these data types are the BCD numbers and the count value used in conjunction with the count function, as well as the data type CHAR, which represents a character in ASCII code.

INT, DINT, REAL

Variables of these data types represent numbers which can be used in mathematical operations.

S5TIME

Variables of the S5TIME data type are required for specifying timer values in timer functions (S5 Timer functions). You specify the time in hours, minutes, seconds or milliseconds. You can enter the timer values with an underline (1h_4m) or without an underline (1h4m).
Functions FC 33 and FC40 from the *IEC Function Blocks* library convert S5TIME to TIME format and TIME to S5TIME format.

TIME

A variable of the TIME data type takes up a doubleword. This variable is used for specifying timer values in IEC timer functions, for example.

The contents of the variable are interpreted as a DINT number in milliseconds and can be either positive or negative (e.g.: T#-1s=L#-1 000, T#24d20h31m23s647msw = L#214748647).

DATE

A variable of the DATE data type is stored in a word in the form of an unsigned integer. The contents of the variable represent the number of days since 01.01.1990 (e.g.: D#2168-12-31 = W#16#FF62).

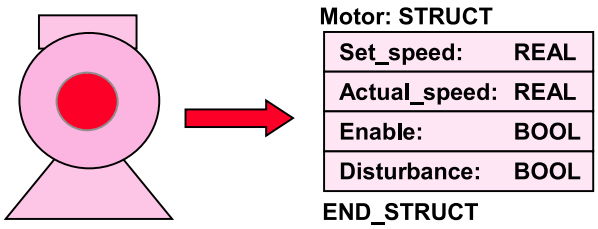
TIME_OF_DAY

A variable of the TIME_OF_DAY data type takes up a doubleword. It contains the number of milliseconds since the beginning of the day (0:00 o'clock) in the form of an unsigned integer. (e.g.: TOD#23:59:59.999 = DW#16#0526_5B77).

Importance of Complex Data Types

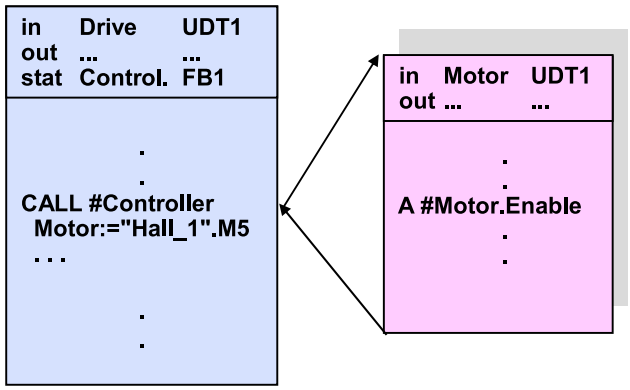
"Better" Structuring of Data:

- adapted to the task
- with "correct" data type



Compact Form of Data Passing in a Block Call:

- "many" data items can be passed in one parameter
- makes structured programming possible
- blocks "communicate" only via the parameter bar
- re-usable software



SIMATIC S7

Date: 30.06.2011
File: PRO2_05E.6
SITRAIN Training for Automation and Drives

Complex Data Types

Complex data types (arrays and structures) result from the grouping of elementary or complex data types.

Complex data types are useful for organizing complex data. In this way, the programmer can generate data types that suit the particular task. He can combine elementary, logically related information units to a new "unit" with its own name.

A typical example for a structure is the data record for a drive. The drive is described as a record of attributes (properties, states), such as, *#Set_speed*, *#Actual_speed*, *#Enable* and *#Disturbance*. Some of these attributes could in turn be structures.

A *#Disturbance* could, for example, be made up of individual components (bits) that for their part give you more exact information about the cause of the disturbance.

Structured Programming

Complex data, in particular, can be passed in a block call as a unit. That is, in one parameter to the called block.

In this way, a multitude of elementary information units can be transferred between the calling and the called block in an elegant and compact manner.

Software Re-usability

This type of data transfer makes structured programming possible and guarantees a high degree of re-usability of software created once.

The entire task to be automated is divided into individual blocks. In the statement section of the called block, no accessing of global addresses, such as, bit memories or variables in global DBs is carried out. Information processing is carried out exclusively with parameters, in which the relevant process data are passed.

The results of processing are returned in parameters to the calling block as well.

Complex Data Types in STEP 7

Keyword	Length (in bits)	Example	
DATE_AND_TIME (Date and Time)	64	DT#99-06-14-12:14:55.0	
STRING (Character string with max. 254 characters)	8 * (number of characters +2)	´This is a string´ ´SIEMENS´	
ARRAY (Group of elements of the same data type)	user defined	Meas_vals: ARRAY[1..20] INT	
STRUCT (Structure, Group of elements of different data types)	user defined	Motor: STRUCT Speed : INT Current : REAL END_STRUCT	
UDT (**U**ser **D**efined Data **T**ype = "Template" consisting of elementary and/or complex data types)	user defined	**UDT as block** STRUCT Speed : INT Current : REAL END_STRUCT	**UDT as array element** Drive: ARRAY[1..4] UDT1

Arrays and Structures

With the help of ARRAYs (Field), several objects of the same type can be combined into one data type. An ARRAY is a data type that consists of a fixed number of elements of a specific data type. An index is assigned to every element of an ARRAY. The index is used to access the element.

An example of an ARRAY is a series of measurements that consists of a fixed number of individual measured values.

In the same way that an ARRAY allows the combining of same-type elements into a higher unit, the data type STRUCT (structure) enables the union of elements of different data types.

Complex data types are predefined. The DATE_AND_TIME data type has a length of 64 bits. The lengths of the data types ARRAY, STRUCT and STRING are defined by the user.

User-defined Data Type

With the help of user-defined data types (UDT), you can define special data types (structures) that can then be used as often as you like in the declaration of parameters and variables.

The data structure is stored in a UDT block (UDT1 ... UDT65535) and can be used - like a "template" - in the declaration of the data type of a variable or a parameter in OBs, FCs, FBs and DBs.

With the help of UDTs, you can save typing time when the same structure is required several times.

Example: You require the same structure 10 times in a data block. First, you define the structure and save it as UDT 1, for example.
In the DB, you define a variable "Drive" as an array with 10 elements of the type UDT1:

Drive: array[1..10]
UDT 1

Thus, you have created 10 data ranges with the structure defined in UDT 1 without additional "typing".

Parameter Types in STEP 7

Keyword	Length (in bits)	Example
TIMER	16	Contact time: TIMER . SI #Contact_time
COUNTER	16	NoCompParts: COUNTER . LC #No_Comp_Parts
BLOCK_FB BLOCK_FC BLOCK_DB BLOCK_SDB	16	Recall: BLOCK_FB . UC #Recall
Pointer	48	Measure: POINTER . L P##Measure .
ANY	80	Measured Values: ANY . L P##Meas_Values .

Parameter Types

In addition to the elementary and complex data types, you can define parameter types for the parameters of FCs and FBs. With this formal parameter you can then carry out the same instructions as with the actual addresses.

These formal parameters must then be supplied with associated actual parameters during a block call.

TIMER and COUNTER

This parameter type defines a formal parameter of the TIMER or COUNTER type.

BLOCK_xx

With the help of the BLOCK_FB or Block_FC parameter types, program blocks can be passed as parameters to called blocks. However, only those blocks (FBs,FCs) that themselves have no control over parameters or static variables (BLOCK_FB) can be passed.

The formal logic blocks can only be called using the instructions UC or CC (not CALL) within the called block.

There are no restrictions for the passing of data blocks (DB, SDB) and for the associated instructions (for example, OPN …).

POINTER

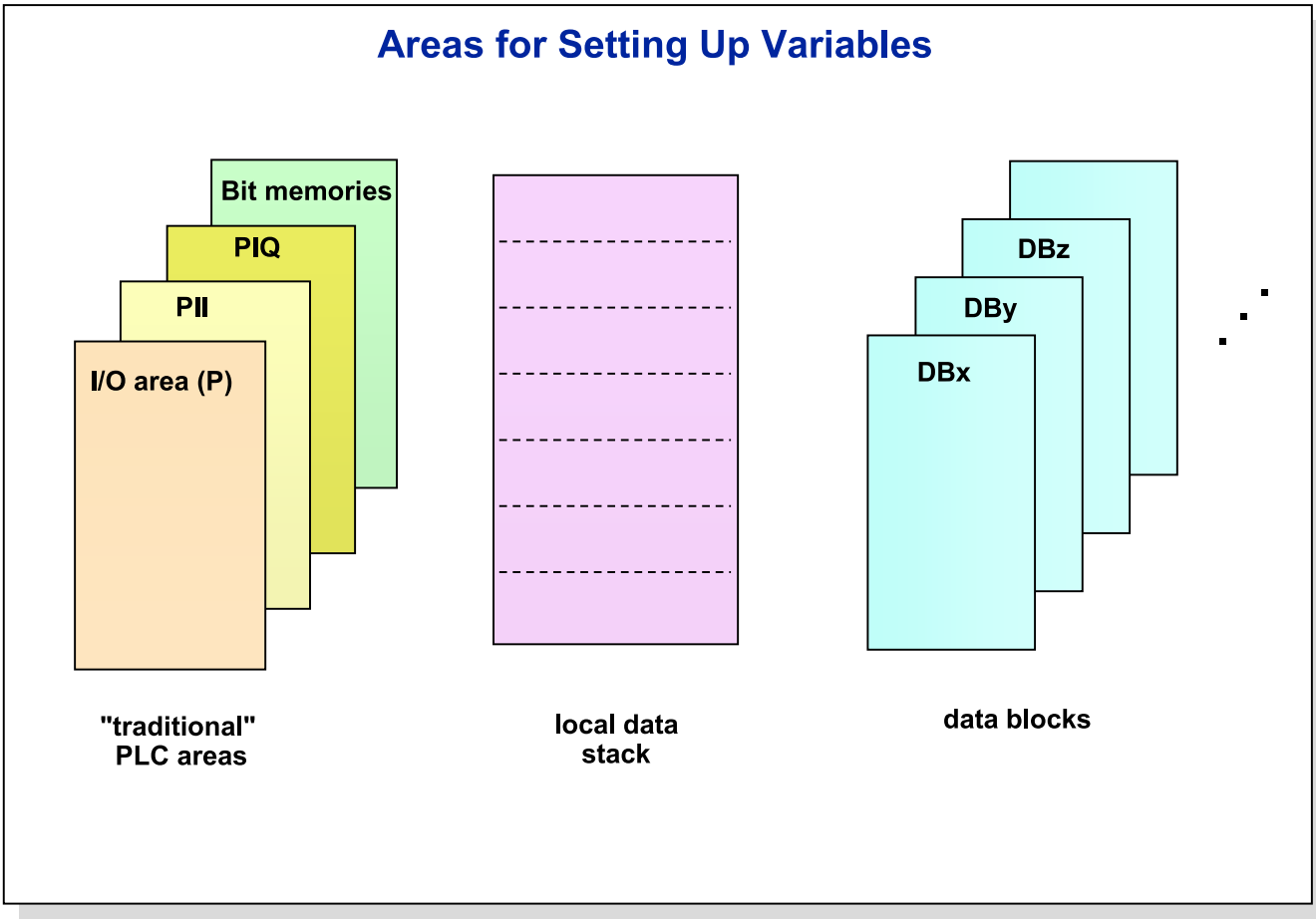
The POINTER data type is used when *any* data type can be the data type of the actual parameter. The POINTER contains the complete initial address (DB number, data area, byte address and bit address) of the actual parameter.

You can assign a formal parameter of the POINTER type by giving the address of the actual parameter: for example P#M50.0.

ANY

The ANY data type is used when any data type can be the data type of the actual parameter. In addition to the complete initial address, information about the data type and the length is also passed in an ANY pointer.

P#M10.0 Byte 10 (Field of 10 components of the BYTE data type beginning with MB 10).



Overview

Apart from program blocks, a user program also consists of data containing information about process states, signals, etc., which are then processed according to the instructions in the user program.

Variables can have a permanent memory location in the process image, bit memory address area or in data blocks or they can be set up dynamically at runtime in the L Stack.

PII, PIQ, Bit Memory I/O

Elementary variables can be declared in the global symbol table of a program folder. In addition to the symbolic name of the variable, you must also give a memory area consisting of area identifier and length, as well as a data type (e.g. FullCrate MW 10 INT).

Unlike the corresponding symbol table in STEP 5 (Assignment List), the program editor permits not only the use of the symbolic name instead of the absolute address. It also monitors the correct use of the variable when parameters are assigned in block calls (type proofing).

Variables that were declared in the global symbol table are global. All blocks in the program folder can access them.

Local Data Stack

The local data stack (L stack) is an area for storing:

- temporary variables of a logic block, including OB start information
- actual parameters to be passed when calling functions
- intermediate logic results in LAD programs

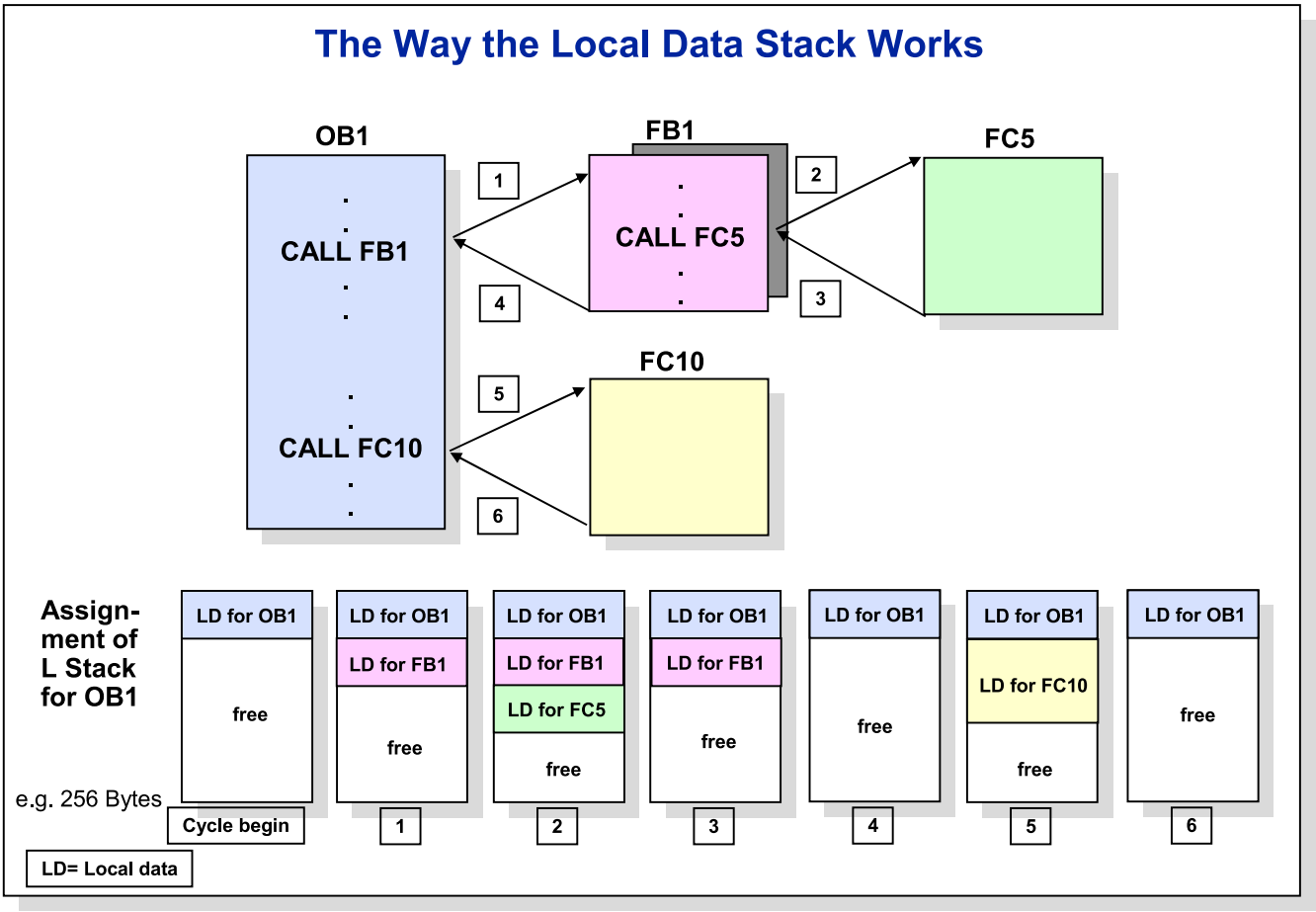
Areas in the L Stack for variables are set up dynamically at the runtime of the block and are enabled once again after execution of the block.

Data Blocks

Data blocks are blocks used by the logic blocks of the user program for storing data.

Unlike the temporary variables, the contents of variables in DBs are not overwritten when execution of the block is completed.

The Way the Local Data Stack Works



Local Data Stack

Every priority class, that is, every OB is assigned its own L-Stack for the OB's temporary variables and for the temporary variables of the blocks called by the OB.

Before a block (OB, FB or FC) is processed, the system reserves dynamic memory in the L-Stack for the temporary variables declared in the block's declaration part. The memory is enabled after BE (block end).

Sequence

The above slide shows a typical sequence of a cyclical OB1 execution. Before the OB1 is processed, the operating system reserves (allocates) memory space for the OB1's temporary variables. Besides the temporary variables declared by the user, a 20 byte area is also reserved and initialized for the start information.

1

Before the FB1's execution, the operating system reserves memory for the FB1's temporary variables. The respective memory area directly follows the memory for the OB1's temporary variables.

2

Before the FC5's execution, the operating system reserves memory for the FB5's temporary variables. The respective memory area directly follows the memory for the FC1's temporary variables.

3

After completion of the FC5, the associated memory is enabled once again.

4

After completion of the FB1, the associated memory is enabled once again.

5

Now, the operating system reserves memory for the FC10. This area directly follows the memory for the OB1's temporary variables.

That way, the exact memory space is used that was previously also used by FB1 and FC5. That is, the original temporary variables of FB1 and FC5 are now overwritten.

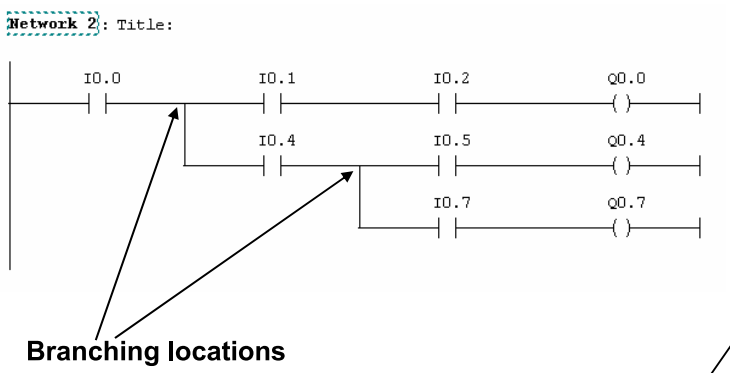
Advantage

The management of temporary memory is performed by the operating system and does not have to be organized by the user (programming errors).

If the respective priority class is interrupted by an OB with another priority class, the local data does not have to be saved. The various OBs are assigned their own local data stacks in each case.

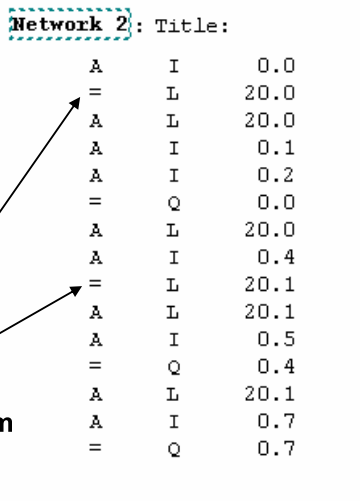
Example: Replacement for Scratchpad Memory

Branching in LAD



Branching locations

STL representation



Help variables from Local Data Stack

SIMATIC S7

Date: 30.06.2011
File: PRO2_05E.11

SITRAIN Training for Automation and Drives

Branching in the LAD Editor

The above example shows the representation of branching, as it can now be programmed by the user with the help of the LAD Editor of STEP 7 by inserting additional coils (Q 0.7, for example).

Scratchpad Memory and Connectors

With STEP 5, the programming of a branching was not directly possible. The user must insert a help variable, as a rule a memory bit (connector), at the location of the branching point as output of the network.

In the next segments - one for every additional branch - this help variable is then used in each case as input and scanned.

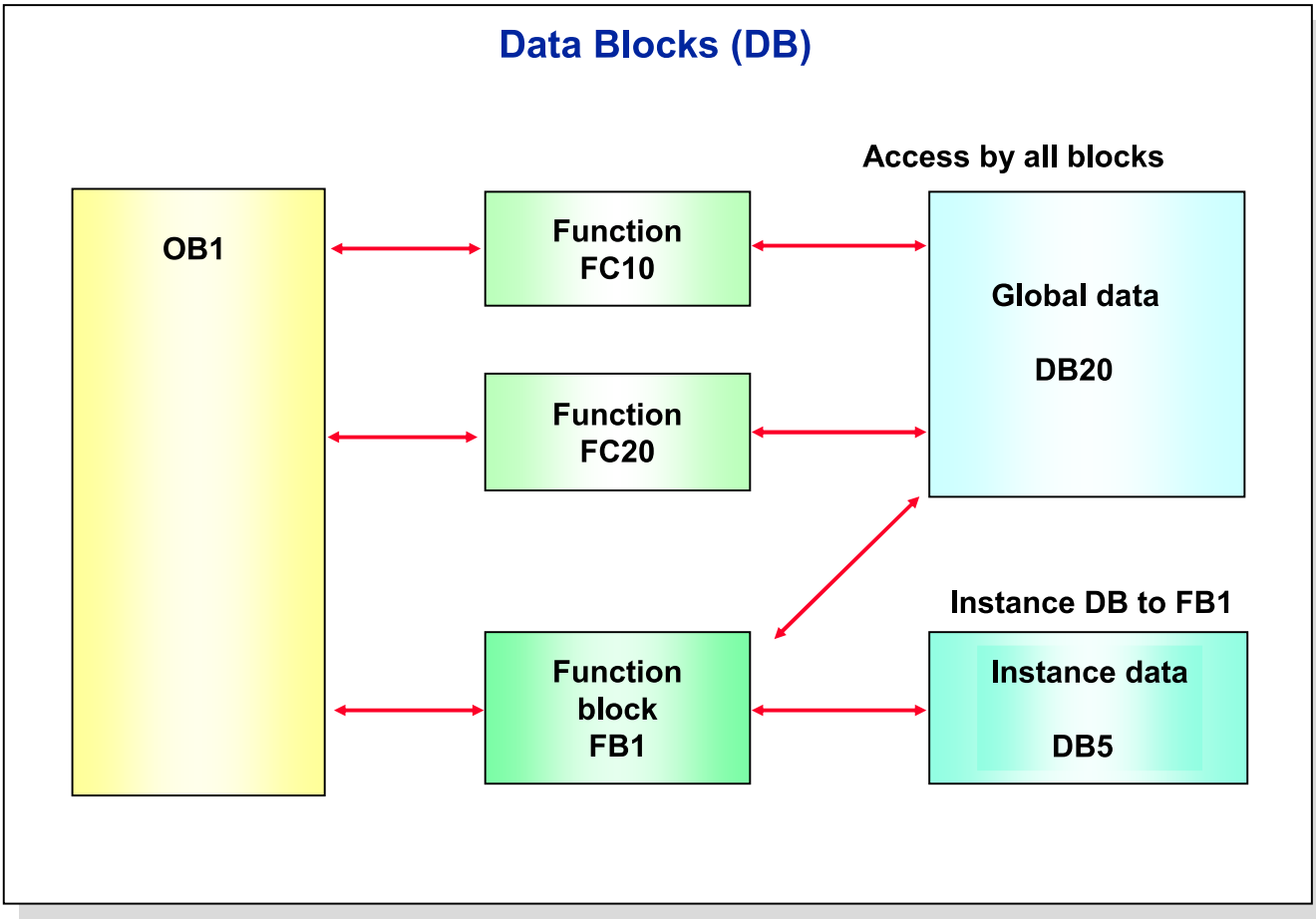
LAD Editor

With the help of the LAD Editor, it is possible to program such branchings directly with STEP 7. Internally, a help variable - a bit of the local data stack - is also used at the branching location by the LAD Editor.

The use of memory in the local data stack guarantees in this case, that the two help variables (L 20.0 and L 20.1) cannot be overwritten by blocks called in the meantime.

Temporary Variables

The user can also declare temporary variables in the declaration part and access the variables absolutely or symbolically, that is, via the specified identifier name.



Overview

Data blocks are used to store user data. Just like logic blocks, data blocks occupy space in the user memory. Variable data (for example, numerical values), with which the user program works, are found in the data blocks.

The user program can access the data of a data block via bit, byte, word or double word instructions. Access can take place symbolically or absolutely.

Application Area

Data blocks can, dependent on their contents, be installed by the user in different ways. Distinction is made between:

- Global (Shared) data blocks: They contain information which can be accessed by all logic blocks of the user program.
- Instance data blocks: They are always assigned to one FB. The data of this DB is only to be processed by the associated FB.

Creation of DBs

Global DBs are either created using the DB Editor or in accordance with a previously set up "user-defined data type".

Instance data blocks are generated when an FB block is called.

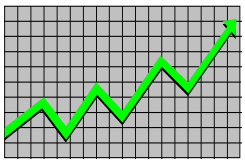
Register

The CPU has two data block registers, the DB register and the DI register. That way two data blocks can be open at the same time.

The Data Type: ARRAY

ARRAY (Field):

- Group of components of the same data type
- Declaration:
  - one dimensional:
    Fieldname: ARRAY[*minIndex*..*maxindex*] OF *data type;*
  - multi-dimensional:
    Fieldname: ARRAY[*mindex1*..*maxindex1*,*mindex2*..*maxindex2*,...] OF *data type;*
    Index: Data type INT (-32768...32767)



Meas_value: ARRAY[1..10]

Meas_value[1]:	Real
Meas_value[2]:	Real
Meas_value[3]:	Real
⋮	
Meas_value[10]:	Real

Examples:

- Declaration of a variable:
  - one dimensional: *Meas_value*: ARRAY[*1*..*10*] OF *REAL;*
  - multi-dimensional: *Position*: ARRAY[*1*..*5*,*2*..*8*,...] OF *INT;*

- Access to a variable:
  - L #Meas_value[5] // Load the 5th. element of the ARRAY‘s // Meas_value in ACCU1
  - T #Result[10,5]

SIMATIC S7

Date: 30.06.2011
File: PRO2_05E.13

SITRAIN Training for Automation and Drives

Overview

The ARRAY data type represents a field with a fixed number of components (elements) of the same data type. A field (=ARRAY) can have up to 6 dimensions (number of indexes). The following restrictions apply for the components’ data types of an array:

- elementary (no restriction)
- complex (DATE_AND_TIME, STRUCT, UDT)
- no parameter types
- no FBs (multi-instance model)

ARRAYs cannot be nested. The limits of the min. or max index range is determined by the value range of INT, that is, from -32768 to 32767.

Access

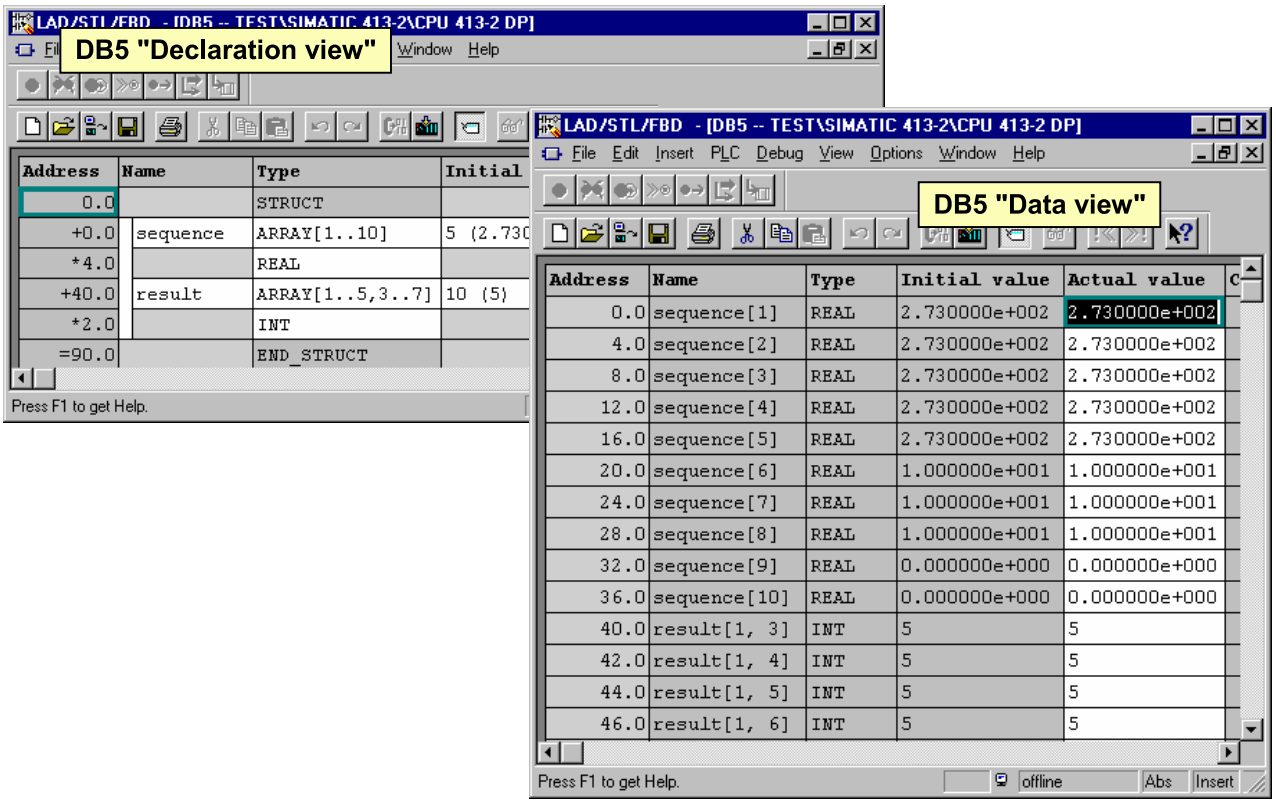
STL instructions can be used to access array components of the elementary data types. An array component is addressed with the array name and an index in square brackets.

The index must be a fixed value, that is, a constant term. Variable indexing at runtime is not possible in STL.

Note

Variable indexing of individual array elements is only possible in the S7-SCL language. Variable access can only be implemented in STL with the help of memory or register indirect addressing.

Declaration and Initialization of ARRAYs



Overview

In the above example, two variables of the ARRAY data type are declared in DB5 with the help of the data block editor. The setting up of a new variable is only possible in the "Declaration view" of the DB (*View -> Declaration View*):

- sequence: ARRAY[1..10] OF REAL
- result: ARRAY[1..5,3..7] OF INT

Initialization of ARRAYs

The individual array components can be pre-assigned values in the declaration (not with FC parameters, in/out parameters of FBs or temporary variables). The initialization value's data type must be compatible with the component's data type.
The initialization values are entered in the column "Initial Value", separated by a comma. If several successive components with the same value are to be initialized, a replication factor can be used to do so. The replication factor is placed in front of the initialization value that is to be entered in round brackets.

Examples

5 (1.23467E+002) //the next 5 components are initialized with the value //123.467

5 (7,2,3) //the next 15 elements are alternately assigned with the values 7, 2 and 3

The result of the initialization can be checked or changed in the "Data View" (*View -> Data View*). If the number of initialization values is smaller than the number of components, only the first are pre-assigned and the rest are initialized with "0".

Acceptance of Initialization Values

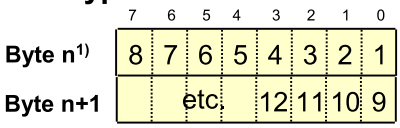
If new initialization values are entered in the declaration view of a DB, these changes only become effective (valid as actual values) after, in the data view, the menu option *Edit -> Initialize Data Block* has been carried out.

The initialization values of ARRAYs in the declaration of input and output parameters in FBs are accepted as actual values in instance DBs when these are generated.

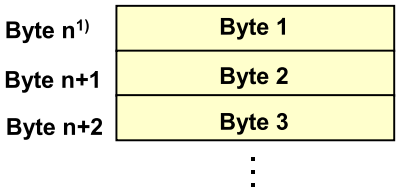
Storage of ARRAY Variables in the Memory

one-dim. arrays

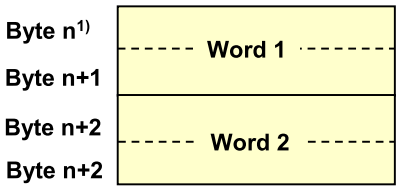
- **Data type BOOL**



- **Data type BYTE, CHAR**



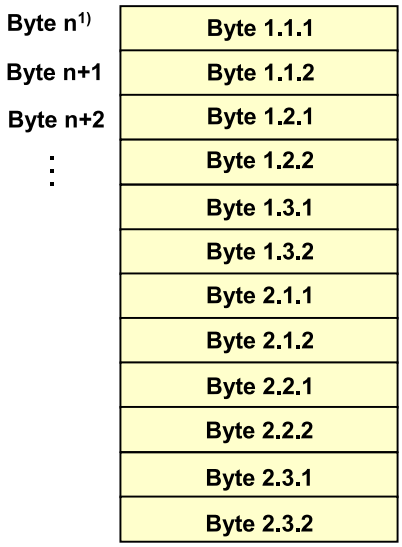
- **Data type WORD, DWORD,...**



1) n = even

multi-dim. arrays

- **Example.**
  ARRAY[1..2,1..3,1..2] OF BYTE



SIMATIC S7

Date: 30.06.2011
File: PRO2_05E.15

SITRAIN Training for Automation and Drives

Overview

Exact knowledge of the storage of ARRAY variables in the memory is then necessary when, during runtime, the individual components are to be accessed using memory or register indirect addressing.

Storage of Variables

An ARRAY variable always begins at a word limit, that is, at a byte with an even address. An ARRAY variable occupies the memory until the next word limit.
Components with BOOL data type begin in the least significant bit address, components with BYTE and CHAR data type at an even byte address. The individual components are listed in sequence.

In multi-dimensional arrays, the components are stored line by line beginning with the first dimension. A new dimension always begins in the next byte with bit and byte components, with components of other data types always in the next word.

Note

The addresses of the individual ARRAY components in a DB are displayed in "Data View" in the column "Address".

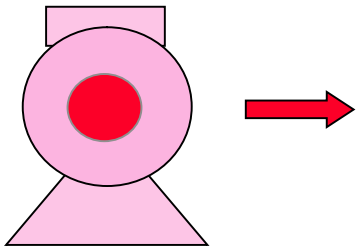
.

The Data Type: STRUCT

STRUCT (Structure):

- Group of components of different data types
- Declaration:

StructName: STRUCT
Comp1Name: *data type;*
Comp2Name: *data type;*
...
END_STRUCT



Motor: STRUCT

Set_Speed:	REAL
Actual_Speed:	REAL
Enable:	BOOL
Disturbance:	BOOL

END_STRUCT

Example:

- Declaration of a variable:

```
• MotorControl    : STRUCT
  ON              : BOOL;
  OFF             : BOOL;
  SetSpeed        : INT;
  ActualSpeed     : INT;
END_STRUCT;
```

Access to the variable

```
S  #MotorControl.ON
L  #MotorControl.ActualSpeed
T  #MotorControl.SetSpeed
                ...
```

SIMATIC S7

Date: 30.06.2011
File: PRO2_05E.16

SITRAIN Training for Automation and Drives

Overview

The STRUCT (Structure) data type represents a specific number of components that can each have different data types. A structure can have up to 8 nesting levels.

A structure can be declared in the declaration part of a logic block, in a global DB or in a user defined data type (UDT).

The following restrictions apply for the components' data types of a structure :

- elementary (no restrictions)
- complex (DATE_AND_TIME, ARRAY, STRUCT, UDT)
- no parameter types
- no FBs (multi-instance model)

Access to Components

STL instructions can be used to access components (elementary data types) of a structure. A component of the structure is addressed using:

- *StructureName.ComponentName*

A period must be inserted between *StructureName* and *ComponentName* as a separator.

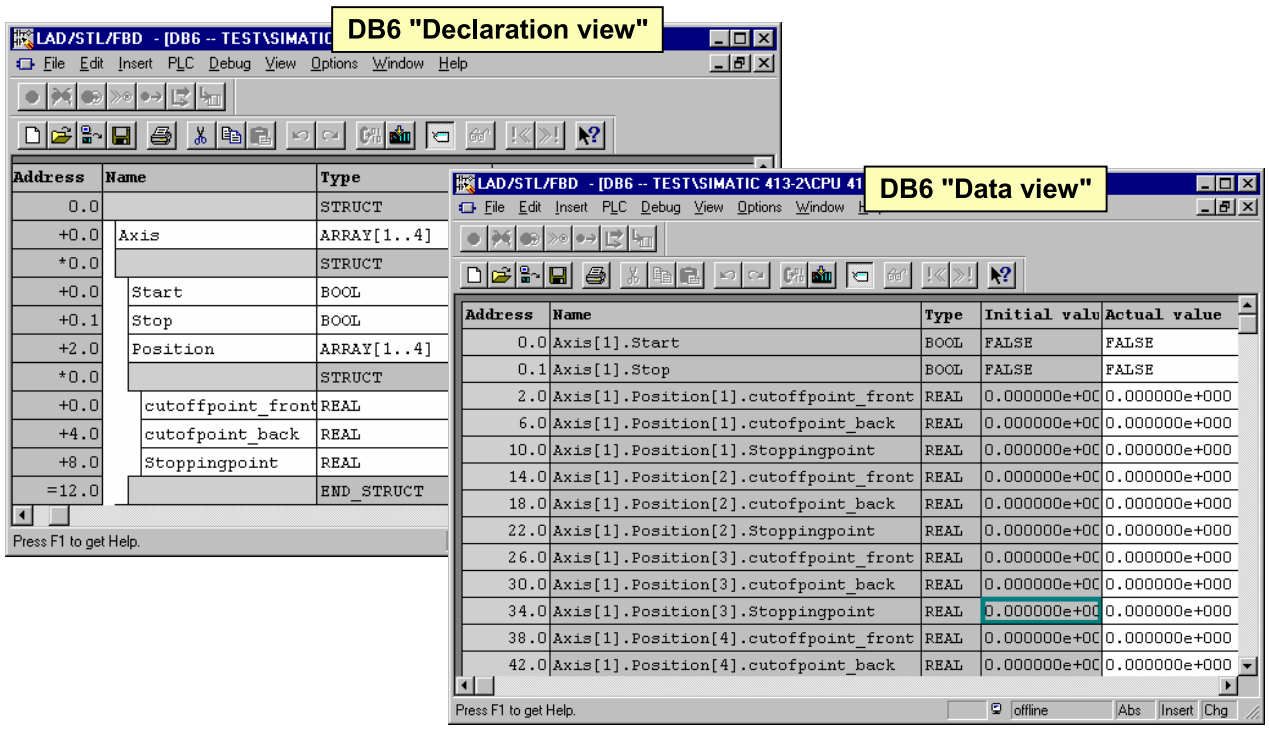
If the nesting depth of the structure is greater, that is, components of the structure are in turn structures, then access of the lowest components of the structure is possible using the "name path", such as:

- *StructureName.ComponentName.SubcomponentName. ...*

A period must be inserted between the names of components and subcomponents in each case.

Declaration of STRUCTs

Example: Declaration of Array - Structure - Array



Overview

In the above example, a single dimensional ARRAY [1..4] with components of the STRUCT type is declared within DB6 ("Hall_1") with the incremental data block editor.

The structure in turn consists of three components of which the first two, that is, "START" and "STOP" have the BOOL data type. The third component has the complex data type ARRAY[1..10].

The components of this ARRAY[1..10] data type are in turn of the STRUCT type with the REAL components "Cutoffpoint_front", "Cutoffpoint_ back" and "Stoppingpoint" .

Access

The individual components can be addressed as follows, for example:

- L "Hall_1".Axis[3].Position[7].Cutoffpoint_back
- S "Hall".Axis[2].START, etc.

Initialization of STRUCTs

The individual structure components can be pre-assigned values in the declaration (column “Initial Value”). The following parameters or variables cannot be initialized:

- input, output and in/out parameters for FCs
- in/out parameters in FBs
- local data in OBs, FBs and FCs

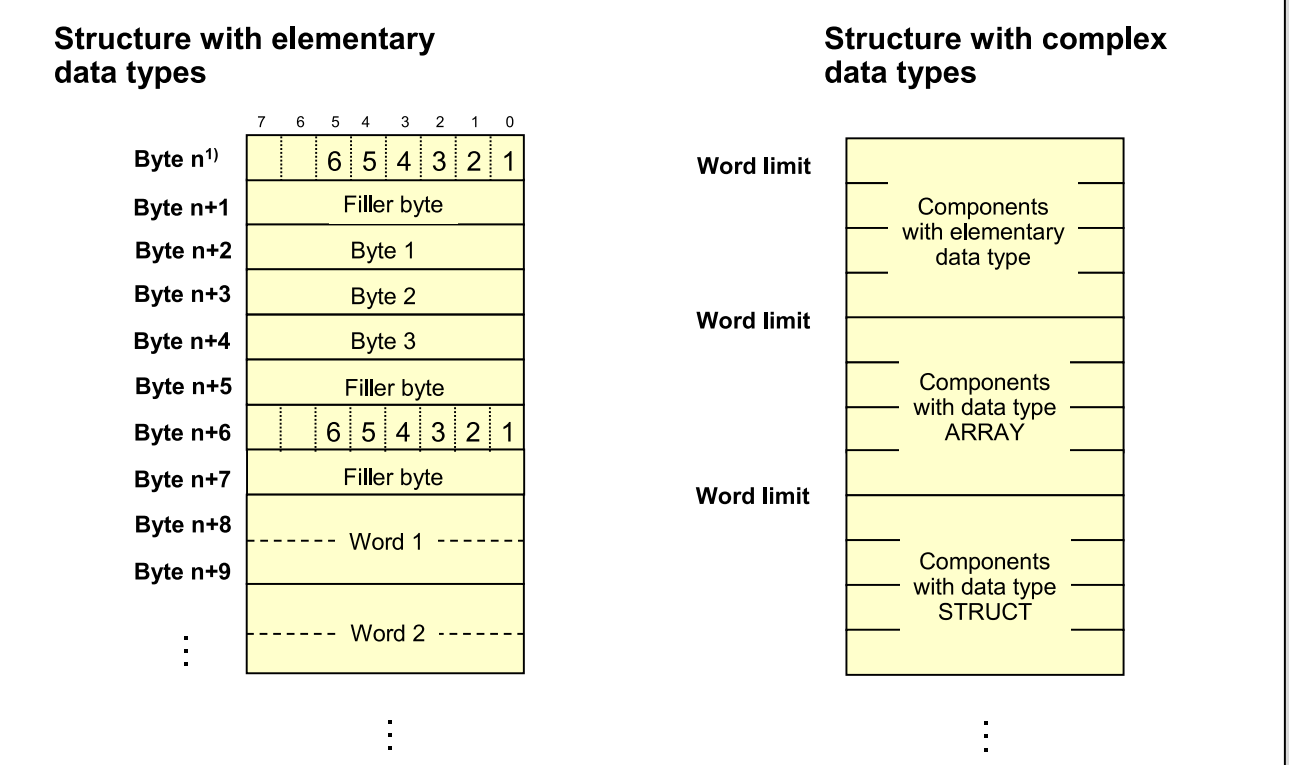
The initialization value’s data type must be compatible with the component’s data type.

Acceptance of Initialization Values

If new initialization values are entered in a DB’s declaration view, these changes only become effective (valid as actual values) after the menu option *Edit -> Initialize Data Block* has been carried out.

The initialization values of STRUCTs in the declaration of input and output parameters in FBs are accepted in the instance DB when it is generated.

Storage of STRUCT Variables in the Memory



Overview

Exact knowledge of the storage of ARRAY variables in the memory is then necessary when, during runtime, the individual components are to be accessed using memory or register indirect addressing.

Storage of Variables

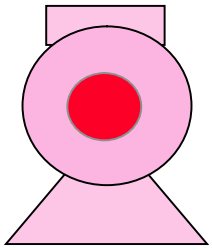
A STRUCT variable always begins at a word limit, that is, at a byte with an even address. Subsequently, the individual components are then located in the sequence of their declaration in the memory. A STRUCT variable occupies the memory until the next word limit.

Components with BOOL data type begin at an even byte in the least significant bit, components with BYTE and CHAR data type at an even byte address.
Components with other data types always begin at a word limit.

User-defined Data Types: UDT

UDT user-defined data types:

- **create a template for later use in declarations**
- **globally valid for all blocks of the program folder**



Example:

- **Definition of a new data type (Structure):**

```
UDT1    STRUCT
    SetSpeed      : REAL;
    ActualSpeed   : REAL;
    Enable        : BOOL;
    Disturbance   : BOOL;
  END_STRUCT;
```

...

UDT1: STRUCT

Set_Speed:	**REAL**
Actual_Speed:	**REAL**
Enable:	**BOOL**
Disturbance:	**BOOL**

END_STRUCT

- **Declaration of variables:**
  - Motor_1: UDT1;
    Motor_2: UDT1;
- **Access to variables:**
  - L #Motor_1.ActualSpeed

SIMATIC S7

Date: 30.06.2011
File: PRO2_05E.19

SITRAIN Training for Automation and Drives

Overview

When a data structure repeats itself frequently in a user program or when a data structure is to be given its own name, then STEP7 permits own, user defined data types (UDT= User Defined Data Type) to be defined (like *typedef* in the high level language "C").

A task to be solved can be programmed more efficiently through the use of application-related data types. Users like manufacturers can then draft and use data types adapted for their problem.

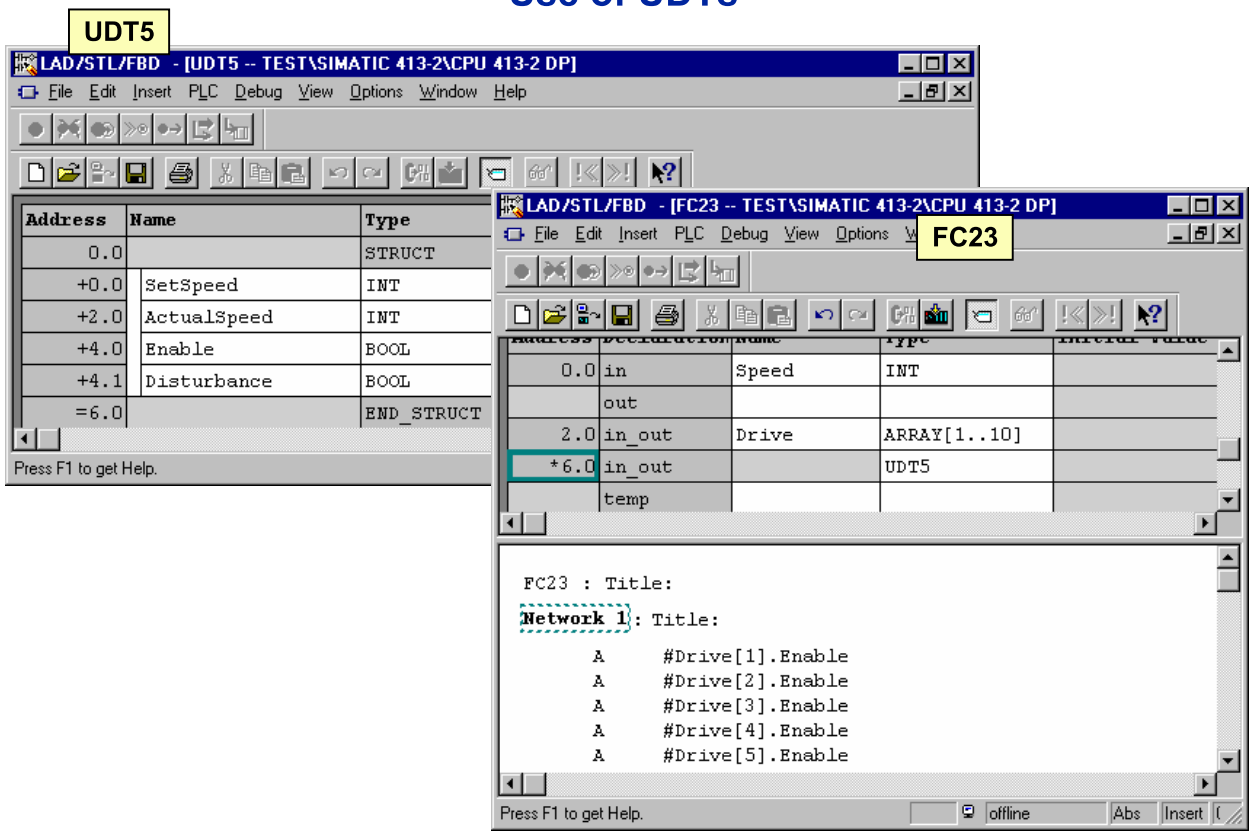
Creation of UDTs

UDTs are created with the DB Editor or with the text editor and then stored in the block folder as a block (UDT1 ... UDT65535).

A symbolic name can then be assigned to this UDT or the related data structure in the global symbol table. A globally valid "template" is created through a UDT that can then used as often as is desired in the declaration of new variables or for the creation of global DBs.

.

Use of UDTs



Overview

In the above example, the UDT5 is created from 4 components (SetSpeed, ActualSpeed, Enable, Disturbance) for a drives structure and then inserted in FC23 in the declaration of in/out parameters.

A one-dimensional ARRAY with 10 components of the UDT5 data type is declared in FC23.

Initial Values for UDTs

User defined data types are pre-assigned and then used in the user program just like structures. The structure of a UDT is the same as that for a STRUCT. The declaration of variables, that can be processed by the user program, has not yet taken place with the creation of a UDT. The UDT is a "template" that you can use as often as you like for the declaration of new variables.

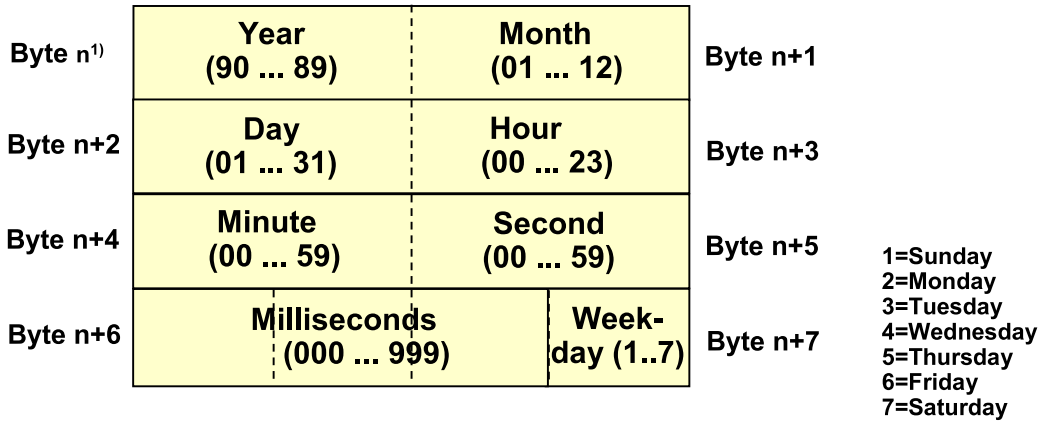
Just as with a structure, you also have the possibility of establishing initial values in the UDT. If the UDT is then used for the declaration of a variable, the contents of these variables are initialized with the initial values of the UDT (not for parameters in FCs, for in/out parameters of FBs and temporary variables).

Creation of DBs

A UDT can also be used as a pattern for the creation (Dialog: *New Data Block*) of a global data block. In this case, a DB is set up with the same structure and with the initial values of the respective UDTs.

The Data Type: DATE_AND_TIME

Structure:



- **All values are saved in the BCD format**
- **Presetting of variables:**
  DT#Year-Month-Day-Hours:Minutes:Seconds.[Milliseconds]
  Example: DT#1998-03-21-17:23:00:00
- **Processing through IEC-Library functions**

1) n = even

Overview

The DATE_AND_TIME data type represents an instant, consisting of the date and the time-of-day. The abbreviation DT can also be used instead of DATE_AND_TIME.

DATE_AND_TIME or DT are keywords and can therefore also be written in lower case.

Presetting

A variable can be preset with an initial value in the declaration (not as block parameter on an FC, as in/out parameter on an FB or as temporary variable).

The presetting must be of the type:

- DT#Year-Month-Day-Hours:Minutes:Seconds.Milliseconds

Specifying the milliseconds can be dropped.

Processing

Variables of the DATE_AND_TIME data type can be processed either with the help of absolute accesses to the individual components or the corresponding IEC-Library functions.

Note

The current time-of-day of the CPU's real-time clock can be read out with SFC1 (READ_CLK). The time is returned by SFC1 as output parameter of the type DATE_AND_TIME.

Functions for Processing DT Variables

IEC-Library in Standard Library V3.x

- **FC1 (AD_DT_TM):** The function FC 1 adds a time duration (format TIME) to an instant (format DT) and returns a new instant (format DT) as a result.
- **FC34 (SB_DT_DT):** The function FC 34 subtracts two instants (format DT) and returns a time duration (format TIME) as a result.
- **FC35 (SB_DT_TM):** The function FC 35 subtracts a time duration (format TIME) from an instant (format DT) and returns a new instant (format DT) as a result.
- **FC3 (D_TOD_DT):** The function FC 3 combines the date formats DATE and TIME_OF_DAY (TOD) and converts these formats into the date format DATE_AND_TIME (DT).
- **FC6 (DT_DATE):** The function FC 6 extracts the data format DATE from the format DATE_AND_TIME.
- **FC7 (DT_DAY):** The function FC 7 extracts the weekday from the format DATE_AND_TIME.
- **FC8 (DT_TOD):** The function FC 8 extracts the date format TIME_OF_DAY from the format DATE_AND_TIME.
- **Comparison functions for DT#Variables:** FC9 (EQ_DT), FC12 (GE_DT), FC14 (GT_DT), FC18 (LE_DT), FC23 (LT_DT), FC28 (NE_DT)

Date: 30.06.2011
File: PRO2_05E.22

Overview

With the installation of STEP7, the Standard Library V3.x library with sub-library IEC Converting Blocks is also installed, which contains functions for the processing of IEC data types.

Functions for processing the variables of the DATE_AND_TIME type are also in this library.

Notes

When using FC1, FC3 and FC34 the following points must be observed:

FC1, FC35

The instant (Parameter T) must lie in the range between DT#1990-01-01-00:00:00.000 and DT#2089-12-31-23:59:59.999. The function does not perform a check of the input parameter.

If the addition or subtraction result does not lie in the range specified above, the result is limited to the respective value and the binary result BR is set to “0”.

FC34

The instants must lie in the range between DT#1990-01-01-00:00:00.000 and DT#2089-12-31-23:59:59.999. The function does not perform an input check. If the first instant (parameter T1) is greater (younger) than the second (parameter T2), the result is positive. If the first instant is less (older) than the second, the result is negative.

If the subtraction result lies outside of the TIME number range, the result is limited to the respective value and the binary result BR is set to “0”.

FC3, FC6, FC7, FC8

These function values do not report back any errors. The user himself is responsible that the input parameters are supplied with valid values.

Comparison Functions

The comparison functions also do not perform any error evaluation. The respective comparison function signals in the return value RET_VAL if the comparison is fulfilled (RET_VAL=TRUE) or not (RET_VAL=FALSE).

The Data Type: STRING

Variables of the STRING (character string) type:

- **Data type STRING represents a character string of up to 254 characters**
- **Application: Handling of message texts**
- **Declaration:**
  - *StringName: STRING[maxNo]:* *'Initializationtext'*
    (String variable for up to *maxNo* characters, *maxNo*: 0... 254)
  - *StringName:* STRING: *'Initializationtext'*
    (String variable for up to 254 characters)

Examples:

- **Declaration of variables:**
  - Fault signal : STRING 'Motor failure_4'
    (Variable *Fault signal* initialized with above text)
  - Warning : STRING[50] ' '
    ("empty" variable *Warning,* can accept up to 50 characters)
- **Processing:**
  - elementary accesses:
    L #Fault signal[5] (loads the 5th character from *Fault signal*)
  - Processing by means of FCs of the IEC library

SIMATIC S7

Date: 30.06.2011
File: PRO2_05E.23

SITRAIN Training for Automation and Drives

Overview

The String data type is used for storing character strings (for example message texts). In this way, a simple “(message) word processing system” can be implemented in an S7-CPU. The STRING data type represents a character string of up to 254 characters.

The number specified in square brackets in the declaration (1..254) gives the maximum number of characters that can be saved in the STRING variable. If this information is left out, the STL/LAD/FBD Editor assumes a length of 254 characters.

Access to STRING Variables

The individual characters of a STRING variable can be accessed with the help of elementary STL instructions, such as:

- L *StringName[5]* (loads the 5th. character that is stored in the variable)

The actual processing of STRING variables (message texts) is possible using FCs of the IEC-Library.

Initialization

In the declaration, the variables of the STRING data type can be pre-assigned with start texts (not as block parameters to an FC, as in/out parameters of an FB or as temporary variable).

Initialization takes place in ASCII coded characters that are framed with single quotation marks. If special characters for controlling the term are to be included, then a dollar character ($) must be placed in front.

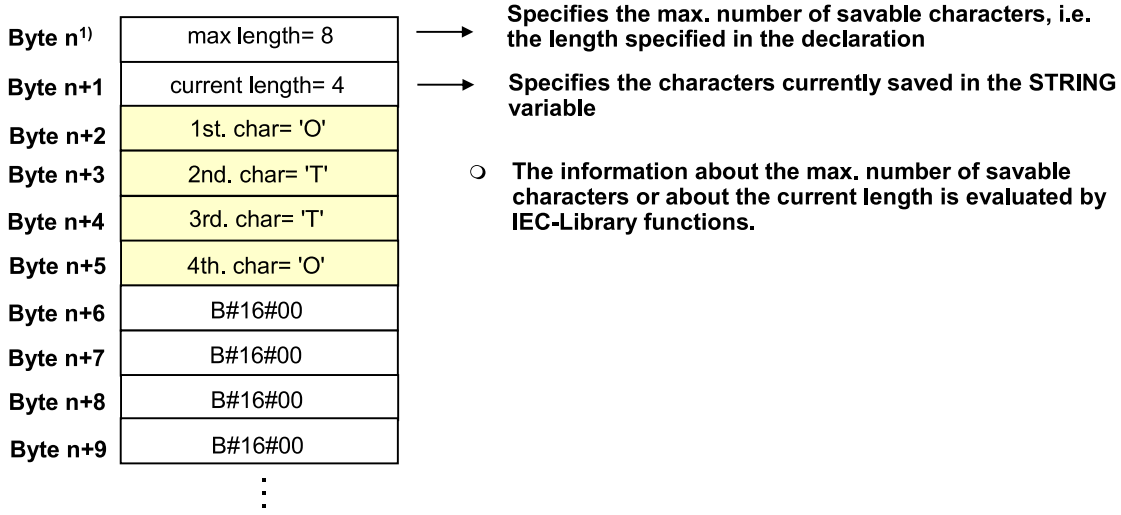
Usable special characters are:

- $$ simple dollar character
- $L, $l line feed (LF)
- $P, $p page feed
- $R, $r carriage return
- $T, $t tabulator

Storage of STRING Variables in the Memory

Example:

- **Declaration with initialization**
  - Given name: STRING[8]: 'OTTO'
- **Storage of STRING variable "Given name"**



1) n = even

SIMATIC S7

Date: 30.06.2011
File: PRO2_05E.24

SITRAIN Training for Automation and Drives

Overview

A variable of the STRING data type is maximum 256 bytes long, whereby up to 254 "net data", that is, characters can be accepted.

Storage of Variables

STRING variables always begin at a word limit, that is, at a byte with even address.

In setting up the variable, its maximum length is entered into the first byte of the variable according to the variable declaration. Likewise, in the pre-assignment or in the processing, the currently used length, that is the length of the saved character string is entered in the second byte with the help of IEC-Library functions.

Both information is required by the IEC-Library functions in the processing of STRING variables.

Subsequently, the characters follow in ASCII-Format. Unoccupied characters in the STRING variable are filled with B#16#00 in the initialization.

Parameter Passing

Variables of the STRING data type can be passed just like ARRAY or STRUCT variables to block parameters with the same data type, that is, same STRING length.

A passing to FC or FB parameters of the POINTER or ANY type is also possible.

.

Functions for Processing STRING Variables (Selection)

IEC-Library in Standard Library

- **FC2 (CONCAT):** The function FC2 combines two STRING variables into a character string.
- **FC4 (DELETE):** The function FC 4 deletes L characters as of the Pth character in a character string
- **FC11 (FINF):** The function FC 11 delivers the position of the second character string within the first character string.
- **FC17 (INSERT):** The function FC 17 inserts the character string at parameter IN2 into the character string at parameter IN1 after the Pth character.
- **FC20 (LEFT):** The function FC 20 delivers the first L characters of a char. string.
- **FC21 (LEN):** The function FC 21 outputs the current length of a character string (number of valid characters).
- **FC26 (MID):** The function FC 26 delivers the middle section of a character string
- **FC31 (REPLACE):** The function FC 31 replaces L characters of the first character string (IN1) as of the Pth character (included) with the second char. string (IN2).
- **FC32 (RIGHT):** The function FC 32 delivers the last L characters of a char. string.
- **Comparison functions for STRING variables:** FC10 (EQ_STRING), FC13 (GE_STRING), FC15 (GT_STRING), FC19 (LE_STRING), FC24 (LT_STRING), FC29 (NE_STRING)

Overview

With the installation of STEP7, the Standard Library V3.x library with sub-library IEC Converting Blocks is also installed, which contains functions for the processing of IEC data types.

Notes

The functions, in general, perform error evaluation with the help of details about maximum length or actual length used. If the functions recognize an error, then, generally, the BR bit is set to “0”.

A detailed description of the individual functions can be found in the Online help for IEC-Library.

Comparison Functions

The comparison functions perform a lexicographic comparison of the character strings. The characters are, beginning from the left, compared with their ASCII coding (is 'a' larger than 'A' and 'A' smaller than 'B', for example).

The first differing character determines the comparison result. If the left portion of the longer character string is identical to the shorter character string, then the longer character string is considered to be larger.

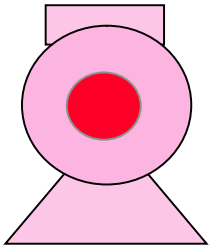
The functions do not signal any errors. The respective comparison function signals in the return value RET_VAL if the comparison if fulfilled (RET_VAL=TRUE) or not (RET_VAL=FALSE).

Conversion Functions

The following functions continue to exist, with which a conversion of process sizes in the INT, DINT and REAL format into the STRING data type and vice versa is possible.

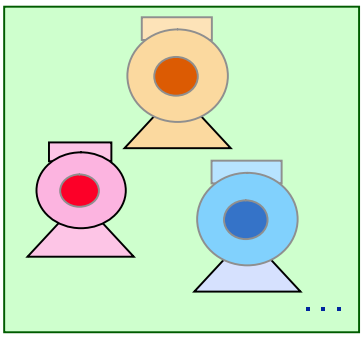
•	FC5:	DI_STRING	FC37:	STRING_DI
•	FC16:	I_STRING	FC38:	STRING_I
•	FC30:	R_STRING	FC39:	STRING_R

Exercise 5.1: Use of Complex Data Types



UDT99 "Motor"

Address	Name	Type	Initial Value
0.0		STRUCT	
+0.0	SetSpeed	REAL	0.000000e+000
+4.0	ActualSpeed	REAL	0.000000e+000
+8.0	SetActDiffMax	REAL	5.000000e-002
+12.0	Enable	BOOL	FALSE
+12.1	Disturbance	BOOL	TRUE
=14.0		END_STRUCT	



Hall_1

DB51 "Conv_area_Motors"

Address	Name	Type	Initial Value
0.0		STRUCT	
+0.0	ConvArea_1_Motor	ARRAY[1..20]	
*14.0		UDT99	
+280.0	ConvArea_2_Motor	ARRAY[1..20]	
*14.0		UDT99	
=560.0		END_STRUCT	

Goal of the Exercise: To become familiar with the complex data type STRUCT, as well as its handling with the help of UDTs.

Task

In the conveyor areas of a milling operation, 20 motors of the same type are each active. The management of the displayed motor data takes place in each case in one ARRAY per conveyor area for this reason.

On the other hand, the data structure for each individual motor is identical, so that it is stored once in a UDT.

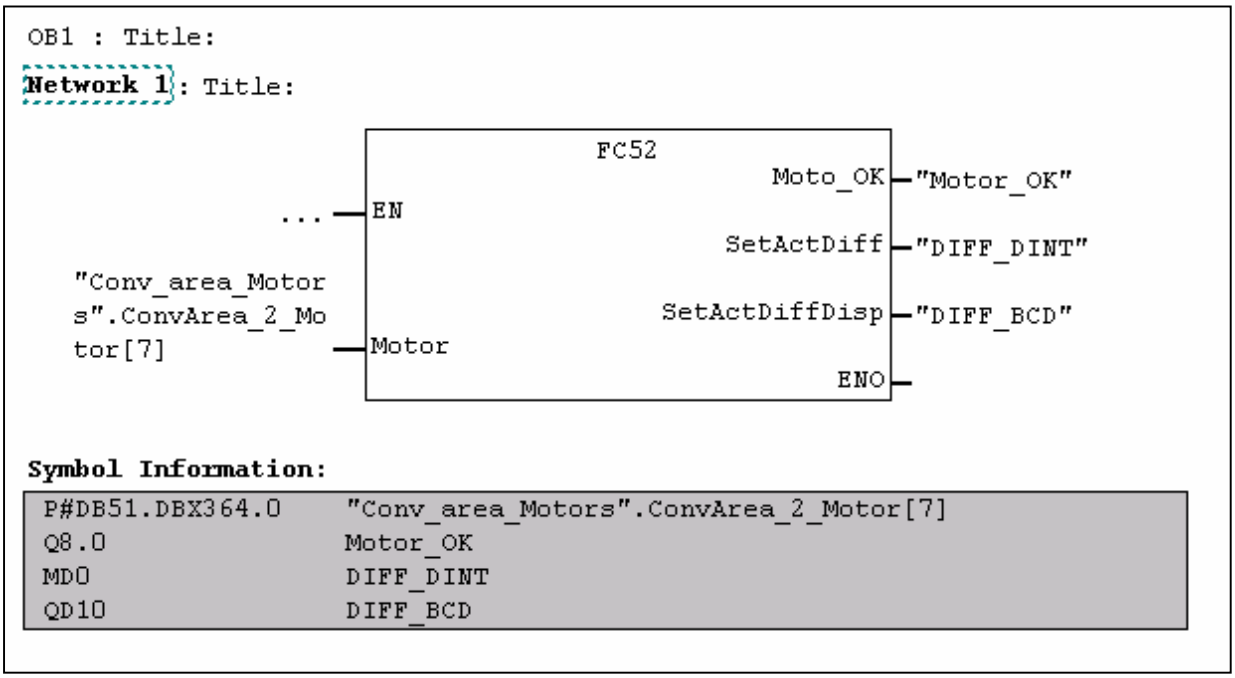
The data record of a motor consists of the following information:

- SetSpeed (REAL): Speed specified by control room
- ActualSpeed (REAL): Measured Actual speed
- SetActDiffMax (REAL): The maximum percent deviation between Set and actual speed specified by quality assurance
- Enable (BOOL): Enable signal specified by control room
- Disturbance (BOOL): OK signal returned to the control room.

What to Do

1. First of all, create a UDT99 "Motor" with the desired structure.
2. Initialize the entries of the UDT99 as follows:
   - SetSpeed: 0.0
   - ActualSpeed: 0.0
   - SetActDiffMax: 0.05 (equals 5% deviation)
   - Enable: FALSE
   - Disturbance: TRUE
3. Then create a DB51 "Conv_Area_Motors". Within DB51 declare two variables *ConvArea_1_Motor* and *ConvArea_2_Motor* of the type ARRAY[1..20] and component type UDT99.
4. Check the contents of DB51 in the data view.

Exercise 5.2: Accessing Complex Data Type



Goal of the Exercise: To become familiar with accessing parameters and variables of the complex data type, as well as its declaration using UDT.

Task

The operating mode of the individual motors in the manoeuvring areas is to be monitored with the function FC52.

The FC52 has the following properties:

- FC52 expects the data record of any motor in the input parameter *#Motor* (UDT99).
- FC52 delivers, in the output parameter *#Motor_OK* (BOOL), the value TRUE, if the bit *Disturbance* is not set and the percent deviation between *SetSpeed* and *ActualSpeed* is less than the *SetActDiffMax* in the motor data record that is passed.
- In addition, FC52 makes available the difference between *SetSpeed* and *ActualSpeed* as a DINT number or as a BCD coded number in the output parameters *#SetActDiff* (DINT) and *SetActDiffDisp* (DWORD).
- In case of an area overrange or an area underrange in the conversion to DINT or DWORD of the REAL numbers passed, FC52 sets the BR bit to "0".

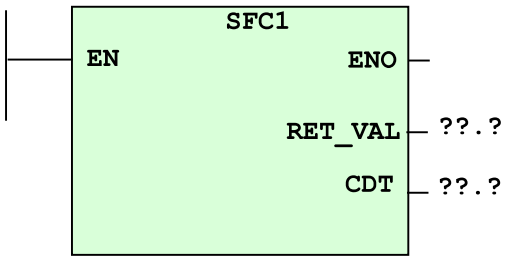
What to Do

1. Create an FC52 with the desired properties
2. Call FC52 in OB1. Supply the input parameter #Motor with the data record of the 7th. motor from ConvArea_2.

   Read out the output signal *#Motor_OK* at output Q8.0.

   Also supply the output parameters *#SetActDiff* and *#SetActDiffDisp* with the actual parameters MD0 and QD10 (digital display)
3. Download the affected blocks to the CPU.
4. Test FC52 with the help of the function "Monitor/Modify Variable", by specifying different values in the respective data record.

Additional Exercise 5.3: Reading the Time-of-Day with SFC 1 (READ_CLK)



Parameter

Parameter	Declaration	Data Type	Memory Area	Description
CDT	OUTPUT	DATE_AND_TIME (DT)	D, L	Output of the current time-of-day and the current date
RET_VAL	OUTPUT	INT	I, Q, M, D, L	Return value of SFC

Goal of the Exercise: To become familiar with the complex data type DATE_AND_TIME.

Task

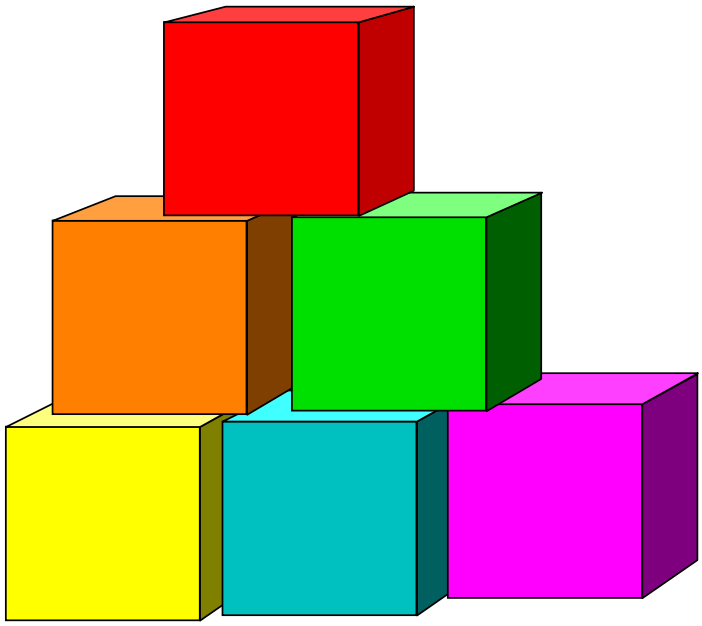
Create an FC53 with the following functionality:

- The FC53 reads out the actual CPU time-of-day with the help of the system function SFC1.
- The hours and minutes are displayed on the digital display.

What to Do

1. Create an FC53 block with the above functionality.
2. Call FC53 in OB1.
3. In the SIMATIC Manager, check, with the help of the menu option *PLC -> Set Time and Date,* if the CPU clock is correctly set.
4. Download the FC53 and the OB1 into the CPU.
5. Test the program.

Block Calls and Multi-instance Model



SIMATIC S7

Date: 30.06.2011
File: PRO2_06E.1

SITRAIN Training for Automation and Drives

Contents **Page**

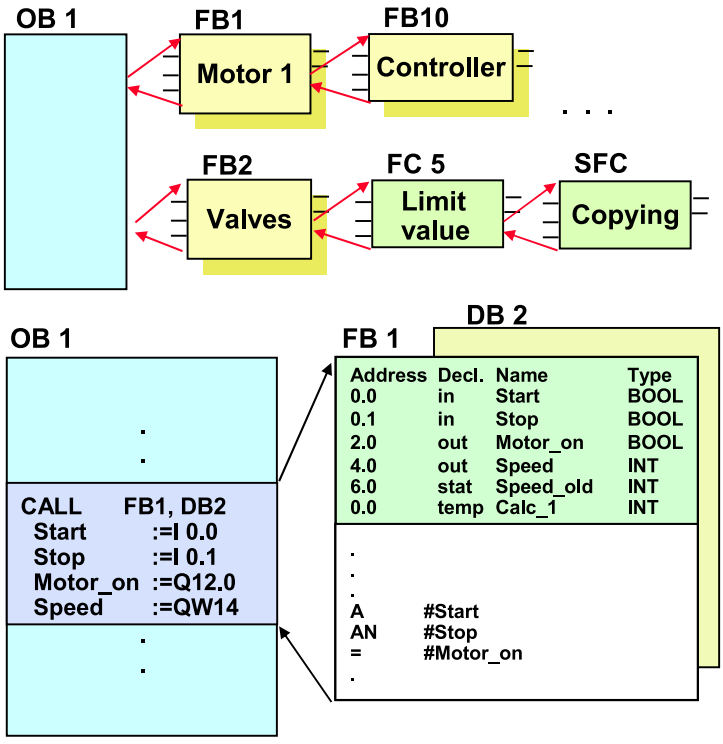
Blocks for Structured Programming

Modularization of the Entire Task:

- **Partial tasks are solved in their own blocks**
- **Parameter assignment enables flexible usage**
  - Example: Drilling cycle with parameter-assignable depth

Re-useability of Blocks:

- **Blocks can be called as often as is required**
- **Restrictions:**
  - no access to global addresses
  - communication only via the parameter list



Modularization of Tasks

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.

Structured Programming

STEP7 supports this concept of modularization with its block model. The partial tasks that result from the division of the entire task are assigned blocks in which the necessary algorithms and data for solving the partial problems are stored.

STEP7 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 addresses such as inputs, outputs, bit memories or variables in DBs can be made from within the instruction section of blocks.

Advantages

Structured programming has the following advantages:

- The blocks for the partial tasks can be created and tested independent of one another.
- With the help of parameters, blocks can be designed so that they are flexible. That way, for example, a drilling cycle can be created that has the coordinates and the depth of the drilling hole passed on to it by means of parameters.
- Blocks can be called as often as is required in different locations with different parameter data records, that is, they can be reused.
- "Re-useable" blocks for special tasks can be delivered in predesigned libraries.

Block Overview in STEP 7

Type of Block	Properties
Organization block **(OB)**	- user interface - graduated priorities (0 to 27) - specific start information in the local data stack
Function block **(FB)**	- parameter-assignable (parameters can be assigned in a call) - with (recall) memory (static variables)
Function **(FC)**	- parameter-assignable (parameters must be assigned in the call) - basically without memory (only temporary variables)
Data block **(DB)**	- structured local data storage (Instance DB) - structured global data storage (valid throughout the entire program)
System function block **(SFB)**	- FB (with memory) stored in the CPU's operating system and callable by the user
System function **(SFC)**	- function (without memory) stored in the CPU's operating system and callable by the user
System data block **(SDB)**	- data block for configuration data and parameters

Blocks in STEP 7

Blocks are, by their function, their structure or their application, limited parts of the user program. The blocks in STEP 7 can - in keeping with their contents - be divided into two classes :

- User Blocks: User blocks include Organization Blocks (OB), Function Blocks (FB), Functions (FC) and Data Blocks (DB).
  The programming person stores the program instructions for data processing or process control in the blocks (OB, FB and FC).
  In the data blocks (DB), the programming person can save data that occur during program execution and then reuse them at a later time.
  User blocks are created in a programming device and are downloaded from there into the CPU.
- System Blocks: System blocks include System Function Blocks (SFB), System Functions (SFC) as well as the System Data Blocks (SDB).
  SFBs and SFCs are used to solve frequently required PLC standard tasks. They are integrated in the CPU's operating system.
  SDBs contain parameter assignment data that are evaluated exclusively by the CPU. SDBs are not created or written by the user program, but by tools such as HW-CONFIG or NETPRO.
  SDBs are created by these tools during loading of the parameter assignment data - invisible to the user - and downloaded into the CPU. Downloading is only possible in the STOP mode.

Downloading Blocks Later On

In addition to the advantages of structured programming, the STEP 7 block concept also provides the following advantage:

- User blocks (OB, FB, FC and DB) in STEP 7 can be modified and downloaded into the CPU during runtime.
  That way, software parts of the system can be upgraded during running operation or (software) errors that occur can be eliminated, for example.

Parameter Declaration in Functions and Function Blocks

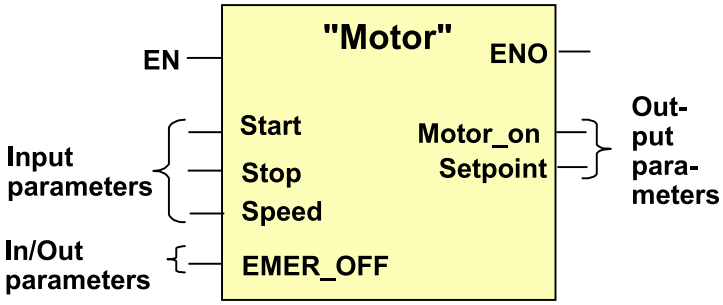
Parameters are the channels for transferring information:

- **Input parameters: Caller -> Block**
- **Output parameters: Block -> Caller**
- **In/Out parameters: Caller <-> Block**

Add.	Decl.	Name	Type	Initial Value	...
0.0	in	Start	BOOL	FALSE	
0.1	in	Stop	BOOL	TRUE	
2.0	in	Speed	INT	0	
2.0	out	Motor_on	BOOL	FALSE	
8.0	out	Setpoint	INT	0	
10.0	inout	EMER_OFF	BOOL	FALSE	
...	stat	...	...	...	
...	temp	..	...	...	

Parameters form the "shell" of a block:

- **Parameters are in the code section like "local" variables**
- **Parameters can have any data type**
  - Data type check during call
  - Exception: POINTER, ANY
- **Call interface is language-independent**
  - PLC languages can be mixed



SIMATIC S7

Date: 30.06.2011
File: PRO2_06E.4

SITRAIN Training for Automation and Drives

Overview

Parameters are used as channels for transferring information between the calling block and the called block. The symbolic names, data types and, if necessary, initial values of parameters are established in the declaration table.

Declaration Types

The type of parameter indicates the direction of the data transfer:

in — Input parameters (FB, FC): They are used to pass information from the calling block to the called block. Only a read-only access to the input parameters is possible within the called block.

out — Output parameters (FB, FC): They are used to pass information (results) from the called block back to the calling block.

in_out — In/Out parameters (FB, FC): In/Out parameters are used to transfer information in both directions. A read and write access is possible to in/out parameters.

Name and Type

Just like local variables, parameters have a symbolic name and a type (data or parameter type). Parameters can be used in the code section of a block in the same way as local variables of the same data type.

For this reason, parameters are also called formal parameters within a block.

Block Calls

In a block call, the formal parameters of a block (FC) must and/or (FB) can be assigned appropriate actual parameters.
To avoid a misinterpretation (regarding the data type) or erroneous use of the actual parameters that are passed, the PLC Editor checks that the actual address created is exactly the same type as the formal parameter when the block is called (Exception: POINTER and ANY).

The type check and the parameter passing mechanism is language independent. This guarantees that blocks that were created with different PLC Editors (STL, LAD, FBD, SCL, etc.) can call each other.

Function Properties

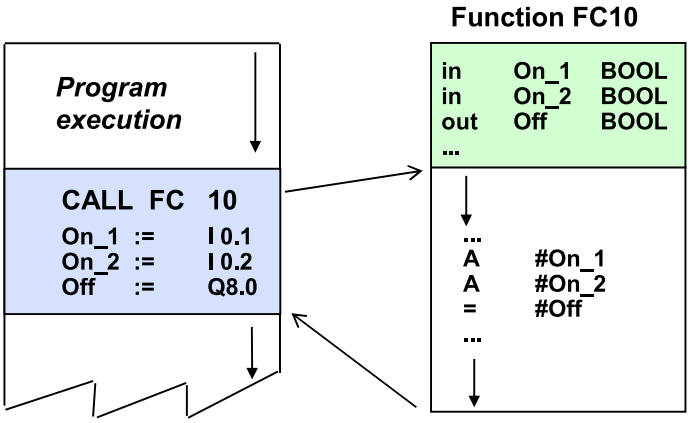
Parameter-assignable blocks:

- as many input, output, and in/out parameters as required
- without memory, i.e. only temporary variables

IEC 61131-3 conform:

- as many input parameters as required
- only one output parameter RET_VAL
- no access to global variables and absolute addresses
- with the same input parameters they deliver the identical result

Expand the instruction set of the processor



SIMATIC S7

Date: 30.06.2011
File: PRO2_06E.5

SITRAIN Training for Automation and Drives

Overview

Functions represent parameter-assignable blocks without memory. In STEP 7 they can have as many input parameters, output parameters and in/out parameters as is required.

Functions have no memory; no separate, permanent data area for storing results exists. Temporary results that occur during function execution can only be stored in the temporary variables of the respective local data stack.

Functions expand the instruction set of the processor.

Application

Functions are primarily used when function values are to be returned to the calling blocks. (for example, mathematical functions, single control with binary logic operation).

IEC 61131 Conforming Functions

If IEC 61131-3 conforming functions are to be created, then the following rules must be observed:

- Functions can have as many input parameters as is required. They can, however, only return one result to the output parameter RET_VAL.
- Global variables can neither be read nor written within functions.
- Absolute addresses can neither be read nor written within functions.
- No instances of function blocks can be called within functions.

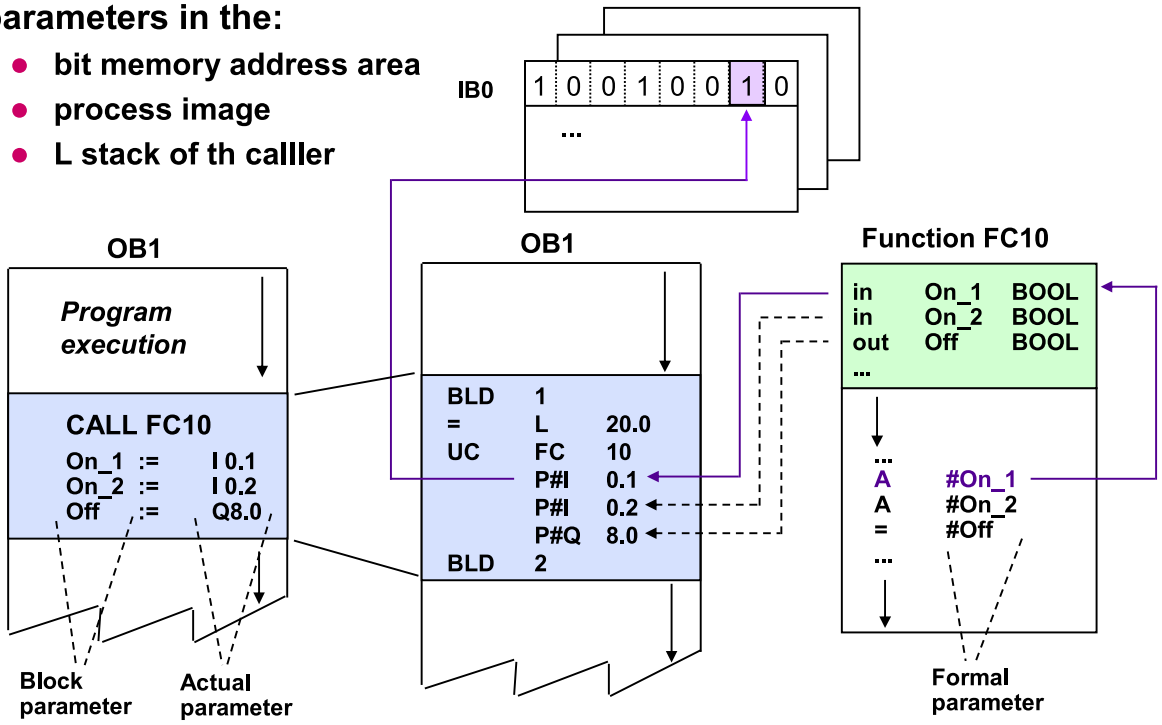
Because of the missing "memory", the returned result of a norm-conforming function is solely dependent on the values of the input parameter. For identical values of the input parameter, a function also returns the identical result.

It is therefore up to the programming person to create norm-conforming functions or to do the block programming and structuring in STEP 7 as it is in STEP 5.

Passing Mechanism for Elementary Data Types

Elementary actual parameters in the:

- bit memory address area
- process image
- L stack of th calller



FC Parameter

Data for processing can be handed over to a called function. This data passing takes place exclusively via the parameter list that pops up after the CALL. The names and data types of the block parameters that pop up are defined in the declaration part of the FC.

Input (read-only), output (write-only) and in/out parameters (reading and writing) can be declared.

The number of parameters is unrestricted (memory space!), the names can contain a maximum 24 characters. In addition, the parameters can be provided with a detailed comment. If the block doesn't have any parameters, then the parameter list is dropped in the FC call.

Passing Mechanism

With a CALL, the STL/LAD/FBD Editor first of all calculates area-crossing pointers from the actual parameters given in the parameter list and stores these immediately after the FC call instruction.

If an access to a formal parameter (for example: *A On_1*) now takes place within the FC, the CPU determines the FC call instruction from the return address stored in the B-Stack. From the pertinent parameter list, the FC then determines the area-crossing pointer to the actual parameters which belong to the formal parameters. Access to the actual parameters then takes place via this pointer.

This passing mechanism corresponds to a "Call by Reference". If access to a formal parameter takes place within an FC, then access to the corresponding actual parameter results from this.

This passing mechanism via pointer entails that:

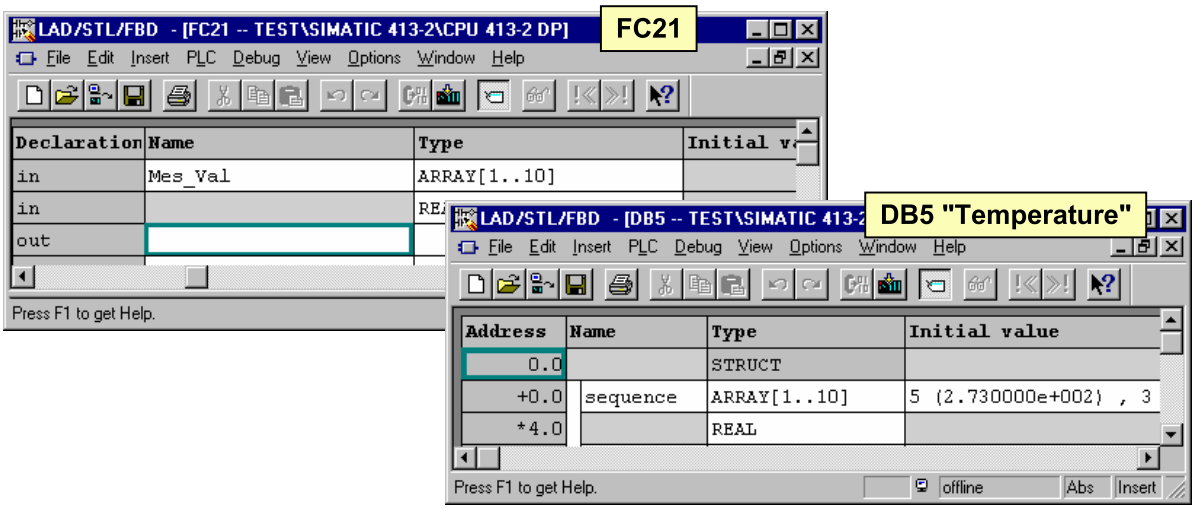
- all block parameters must be assigned in an FC call.
- block parameters cannot be initialized in the declaration.

Notes

If a block parameter is assigned with an actual parameter within a DB or if complex data types are passed, then parameter passing becomes more complex. (see Appendix).

Function Call with Complex Data Types

Example: Passing an ARRAY to a Function



Assignment of the parameters is only possible symbolically

```
Network 1: Mes_Val is declared as an array in FC21

      CALL   FC     21
       Mes_Val:="Temperature".sequence
```


Overview

Parameters of the complex data type (ARRAY and STRUCT) offer a clear and efficient way for transferring larger amounts of related data between the calling and the called block and thus accommodating the concept of "Structured Programming".

An array or a structure can be passed to a called function as a complete variable.

Assigning the Parameters

For the pass, a parameter of the same data type as the actual parameter to be passed must be declared in the called function. The assignment of such a parameter (data type: ARRAY, STRUCT, DATE_AND_TIME and STRING) can only take place symbolically.
Since variables of the complex data type can only be set up in data blocks or in local data stacks, the actual parameter must either be located in a data block (global or instance DB) or in the local data stack of the calling block.

After the STL/LAD/FBD Editor has checked the compatibility of the data types of actual parameter and block parameter in the parameter passing to an FC, only a POINTER with the DB number and area-crossing pointer to the actual parameter is passed to the called FC.

This POINTER is set up in the L Stack of the calling block (V area) through the CALL macro. This POINTER is then of great importance for the programming person in particular, when the passed parameter has to be accessed indirectly (see appendix).

Notes

- The number of occupied local data can be determined by selecting the menu options *View -> Block Properties*.
- Components of ARRAYs or STRUCTs can also be passed to a block parameter if the block parameter and the ARRAY or STRUCT components are of the same data type.

Characteristics for Calling Functions

CALL Instruction

- **Instruction is Macro**
  - Register contents may be overwritten, even DB registers
  - Caution in the interpretation of B stack contents
  - After the call, another DB may be opened
  - Processing time for CALL depends on the number and memory location of the actual parameters
- **CALL instruction makes sure that the block parameters are correctly supplied with current data**
- **Example:**
  - CALL FC10
On_1 := I 0.1
On_2 := I 0.2
Off := Q8.0

Call instruction UC and CC

- **RLO independent (UC) or RLO dependent block call**
  - Examples: UC FC20 or CC FC20
- **only usable, when the FC has no parameters**

SIMATIC S7

Date: 30.06.2011
File: PRO2_06E.8

SITRAIN Training for Automation and Drives

CALL Instruction

The (Macro) CALL instruction should be used for calling blocks (FCs, SFCs, FBs and SFBs).

In an FC call, a direct information exchange between calling block and called function is only possible via CALL. The CALL makes sure that the formal block parameters are correctly supplied.

At any rate, several characteristics must also be taken into consideration, that result from the fact that the CALL is implemented through a macro that for its part consists of several STL instructions.

If a formal parameter is assigned with addresses that are found in a DB, then parameter passing takes place with the help of the DB register (see appendix). From this results:

- within the called FC, it is possible that the DB that is opened is not the DB that was opened before the CALL.
- if the CPU goes into STOP during processing of the called FC, then the value displayed in the B-Stack -> DB-Register is that which the STL Editor used to overwrite the DB register in the parameter assignment.
- if after processing, a jump is made back into the calling block, it is possible that the DB is no longer open that was opened before the CALL.

UC, CC Instruction

Blocks can also be called with the instructions UC or CC. The UC call instruction is an absolute instruction, that is, UC always calls the block independent of conditions (e.g.: UC FC20).

The CC call instruction is a conditional instruction, that is, CC only calls the block when the RLO is equal to "1". If the RLO is equal to "0", then CC does not call the block and sets the RLO to "1". Subsequently, the instruction following the CC call is processed.

Important

UC and CC can only be used when no parameters were declared in the called FC.

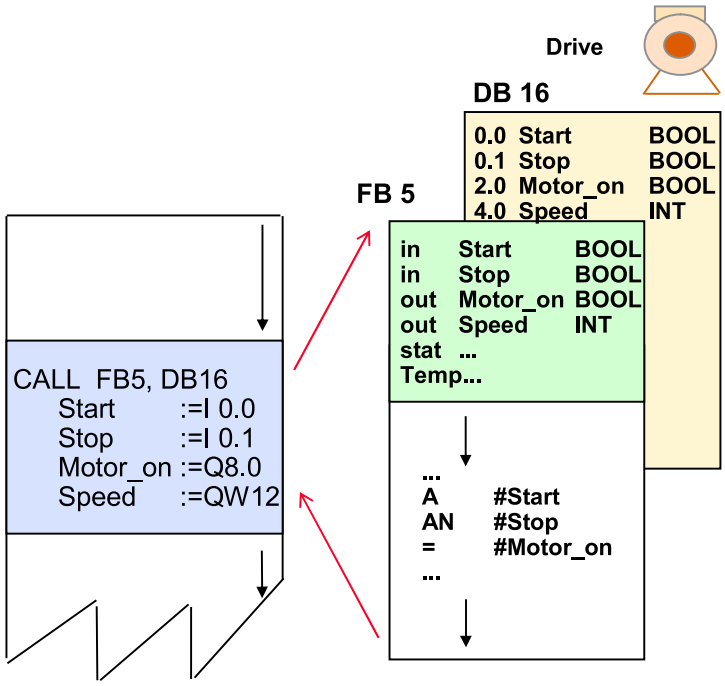
Function Block Properties

Parameter-assignable blocks:

- **IEC 61131-3 conforming**
- **as many input, output and in/out parameters as required**
- **with memory, that is, not only temporary but also static variables**
- **Call with own data area (instantiating)**
- **"Data encapsulation"**

Application:

- **Timer and counter functions**
- **Controlling process units with internal states**
  - boilers
  - drives, valves, etc.



Overview

Function blocks (FB) are blocks of the user program and represent logic blocks with memory according to the IEC Standard 61131-3. They can be called by OBs, FBs and FCs.

Function blocks can have as many input, output and in/out parameters as is required as well as static and temporary variables.

Unlike FCs, FBs are instantiated, that is, an FB is assigned its own private data area in which the FB can "remember" process states from call to call, for example. In the simplest form, this private data area is its own DB, the so-called instance DB.

"Memory"

The programming person has the opportunity to declare static variables in the declaration section of a function block. The function block can "remember" information from call to call in these variables.

The ability of a function block to "remember" information over several calls is the essential difference to functions.

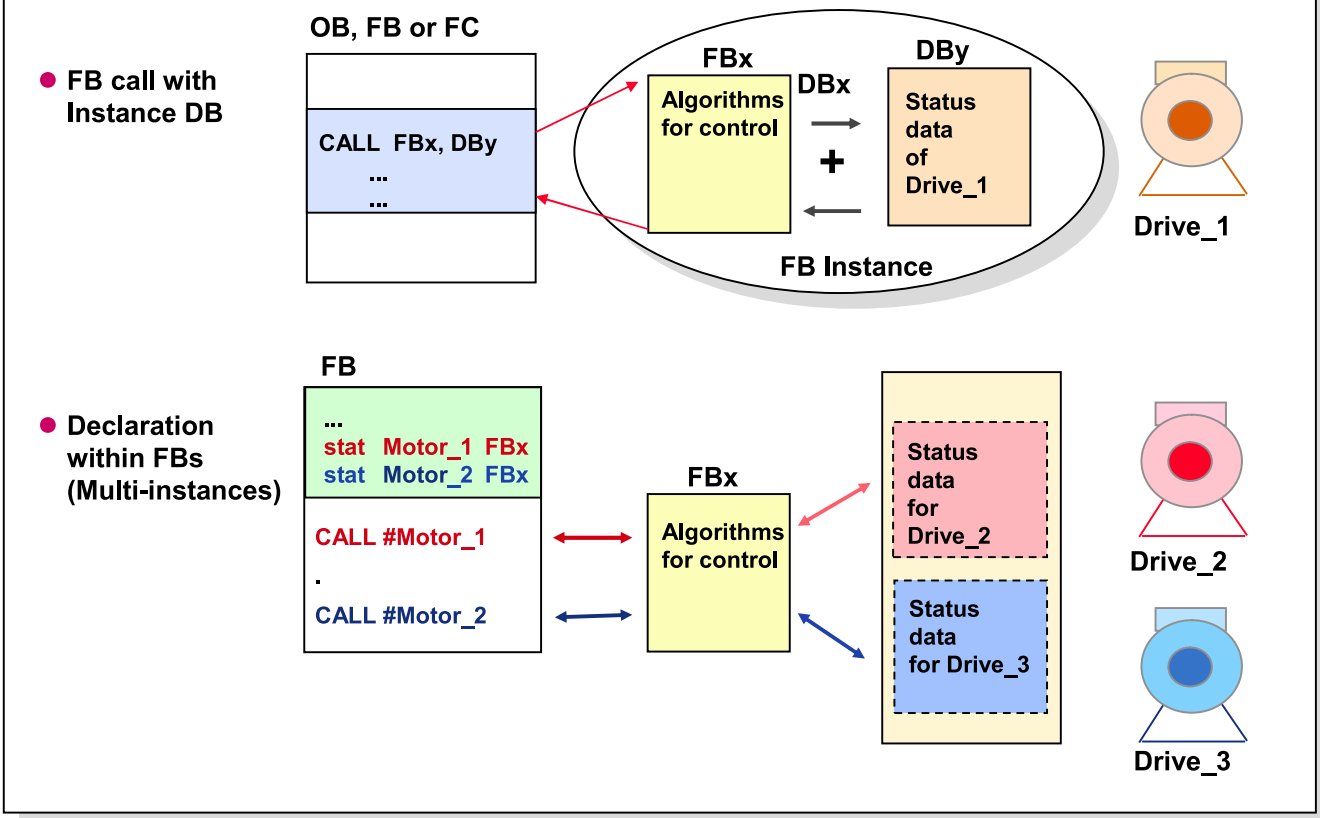
Application

With the help of this "memory", a function block can implement counter and timer functions or control process units, such as processing stations, drives, boilers etc., for example.

In particular, function blocks are well suited for controlling all those process units whose performance depends not only on outside influences but also on internal states, such as processing step, speed, temperature etc.

When controlling such units, the internal status data of the process unit are then copied to the static variables of the function block.

Instance Formation of Function Blocks



What is an Instance?

The concept of instantiating function blocks has great importance and makes up the essential distinctive criterion to the FCs. The setting up of variables within a high level language such as "C" under declaration of variable name and data type in the declaration is called “instantiate” or “instance forming”.

Just like variables, function blocks are also "instantiated". Only through this ‘own’ data area, in which the block parameter values as well as the static variables are stored, does an FB become an executable unit (FB-instance).

The control of a physical process unit, such as a drive or a boiler then takes place with the help of an FB instance, that is, a function block with an assigned data area. The relevant data for this process unit are then stored in this data area.

Instantiating

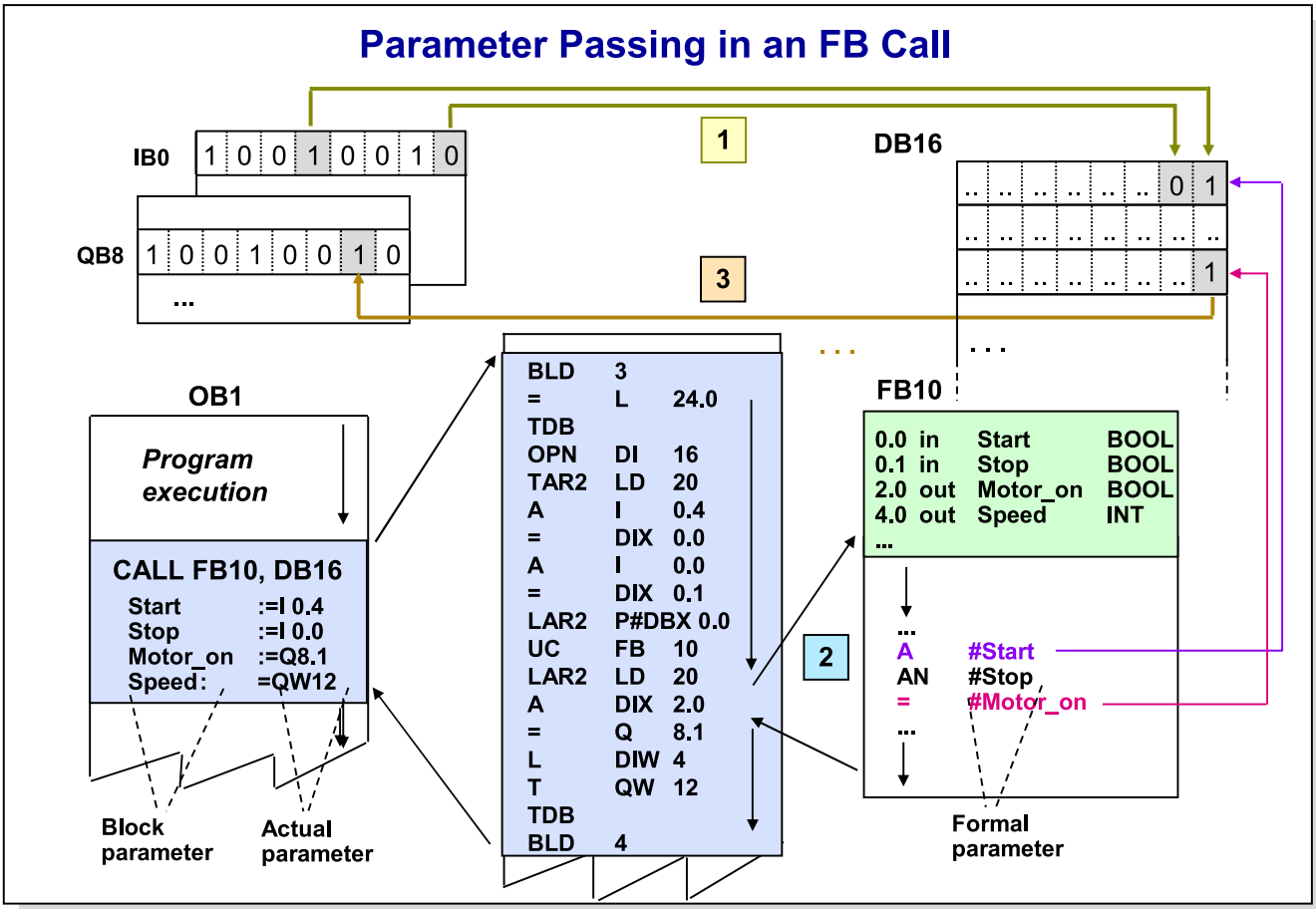
The creation of an FB instance, that is, the assignment of its own memory area in an FB call, can be made in two ways in STEP 7:

- through the explicit declaration of a so-called instance data block when a function block is called.
- through the explicit declaration of instances of a function block within a higher-level function block (multi-instance model).
  STEP 7 then makes sure that the data area required for the instance is set up within the data area of the higher-level FB.

Advantages

The instance concept of STEP 7 has the following advantages:

- In the call of FBs, no measures for the saving and administration of local data are necessary except for the assignment of instance DBs.
- A function block can be used several times due to the instance concept. If, for example, several drives of the same type are to be controlled, then this takes place using several instances of an FB. The status data of the individual drives are stored in the static variables of the FB.



Parameters in an FB Call

Data for processing can be handed over to a called FB-instance. This data passing can take place using the parameter list that pops up after the CALL. The type (input, output, or in/out parameter), the names and data types of the parameter are defined in the declaration part of the FC.

Unlike an FC call, the input and output parameters as well as the in/out parameters of the elementary data type do not have to be assigned with actual parameters in the call of an FB. The reason for this lies in the mechanism of how the actual parameters are passed to the called FB.

Passing Mechanism

If an instance DB is created for an FB, the block editor automatically reserves memory for the block parameters (input, output, and in/out parameter) and for the static variables declared in the declaration part of the FB. The addresses of the parameters and static variables in the instance DB are exactly those byte or bit addresses given by the Editor that are found in the first column of the FB declaration section.

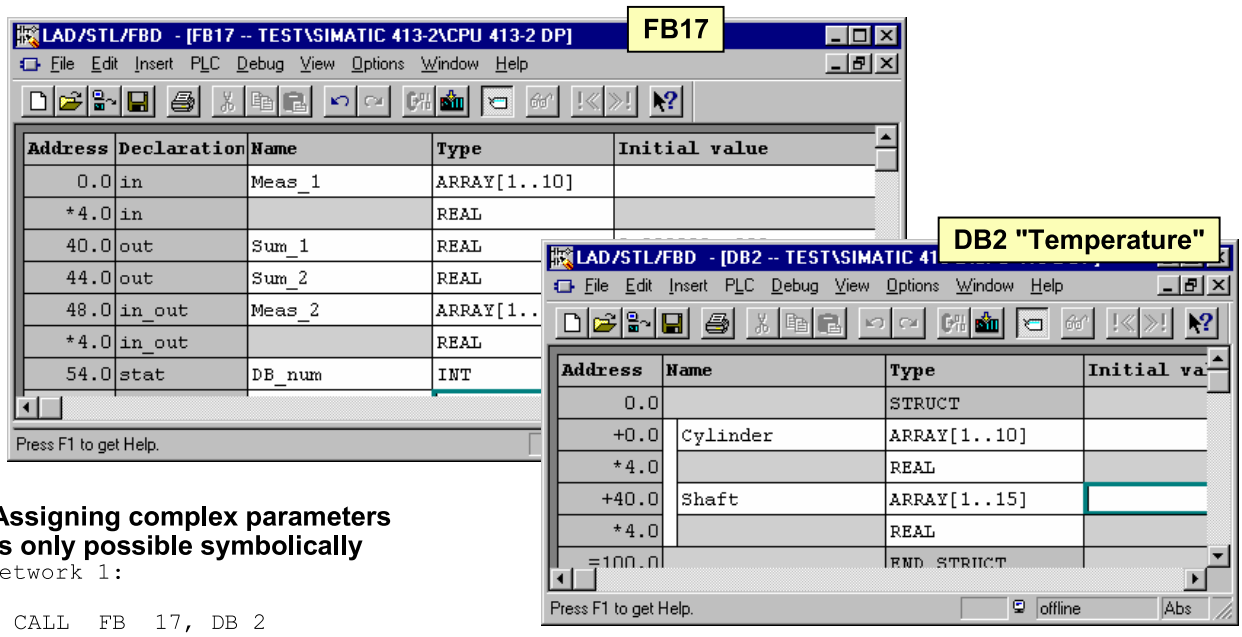
In an FB-instance call using the CALL macro, the instance-DB is opened using the DI register and the values of the current input and in/out parameters are copied into the instance-DB, before the actual FB processing.

The switch to the FB processing then takes place. If an access to the formal parameters now takes place within the FB processing, then this results in an access to the addresses belonging to the instance-DB. This access takes place internally using register indirect addressing with regard to the DI register and the AR2 register.

After FB processing, the values of the formal output and in/out parameters are copied into the actual parameters specified in the CALL. Only after that, does the process continue with the next instruction after the CALL

FB Call with Complex Data Types

Example: Passing an ARRAY to a Function Block



Assigning complex parameters is only possible symbolically

```
Network 1:

  CALL   FB   17, DB 2
   Meas_1     :="Temperature".Cylinder
   Sum_1      :=MD20
   Sum_2      :=MD30
   Meas_2     :="Temperature".Shaft
```


Complex Data Types

Just as with functions, addresses of the complex data type (ARRAY, STRUCT, DATE_AND_TIME and STRING) can be passed completely to a called function block.

For the pass, a parameter of the same data type as the actual parameter to be passed must be declared in the called function block.

The assignment of such a parameter is only possible symbolically.

Input and Output Parameters

For input and output parameters of the complex data type, corresponding areas for the values of the actual parameters are set up in the instance DB. In the call of the FB, the actual parameters of the input parameter are then copied into the instance DB using SFC 20 (BLKMOV) ("Passing by Value"), before the actual switch to the instruction section of the FB.

In the same manner, the values of the output parameter are copied from the instance DB back into the actual parameter after the FB has been processed.

As a result, a not insignificant amount of copying (processing time) can occur in the assignment of input and output parameters. This amount of copying is bypassed with in/out parameters.

In/Out Parameters

No "Passing by Value" occurs with in/out parameters of the complex data type. 6 bytes are merely reserved for every in/out parameter in the instance data area. A POINTER to the actual parameters is entered in these bytes ("Passing by Reference").

Notes

- Input and output parameters of the complex data type can be initialized in the declaration section of an FB, however not in/out parameters.
- Input and output parameters of the complex data type do not have to be assigned in an FB call, in/out parameters have to be assigned.
- The memory or register-indirect access to input/output parameters or in/out parameters of the complex data type is configured differently to that of elementary parameters (see Appendix).

Characteristics for Calling Function Blocks

Parameter Passing "by value" (Copying the Value):

- **Assignment of the FB parameter in a CALL:**
  - FB parameters do not have to be assigned
  - Assignment and "de-" assignment can take place from "outside" e.g.: direct from the Operator Panel
  - Exception: in/out parameters of the complex data type (STRUCT, ARRAY, STRING and DATE_AND_TIME)
- **Initialization:**
  - FB parameters can be initialized in the declaration
  - Exception: in/out parameters of the complex data type (STRUCT, ARRAY, STRING and DATE_AND_TIME)
- **Access to formal parameters takes place internally using DI and AR2 registers**
  - If the DI or AR2 register is overwritten, access to the instance data is no longer possible
- **Additional call instruction UC and CC**
  - Examples: UC FB20 or CC FB20
  - only usable if the FB has no instance data (param. + static variables)

Assignment of Block Parameters

Block parameters do not have to be assigned in an FB call. In this case, no values are copied into or out of the instance-DB. The parameters in the instance-DB keep the values that were also saved in the last call.

Exception: In/out parameters of the complex data type must be assigned in the parameter list.

Parameter Access from "Outside"

Access to the parameters within an instance DB can be made in the same manner as with the addresses of global DBs. Block parameters can thereby also be assigned or “de-“ assigned from “outside”.

This is then especially useful when, for example, only individual components of complex data types have to be assigned or “de-“ assigned or parameters are directly linked with input/output fields on OPs.

Exception: In/out parameters of the complex data type cannot be assigned or “de-“ assigned from “outside”.

Initialization

Block parameters and static variables can be initialized in the FB declaration. If an instance DB is then created, the initial values specified in the declaration are entered in the instance DB.

Exception: In/out parameters of the complex data type cannot be initialized.

Note

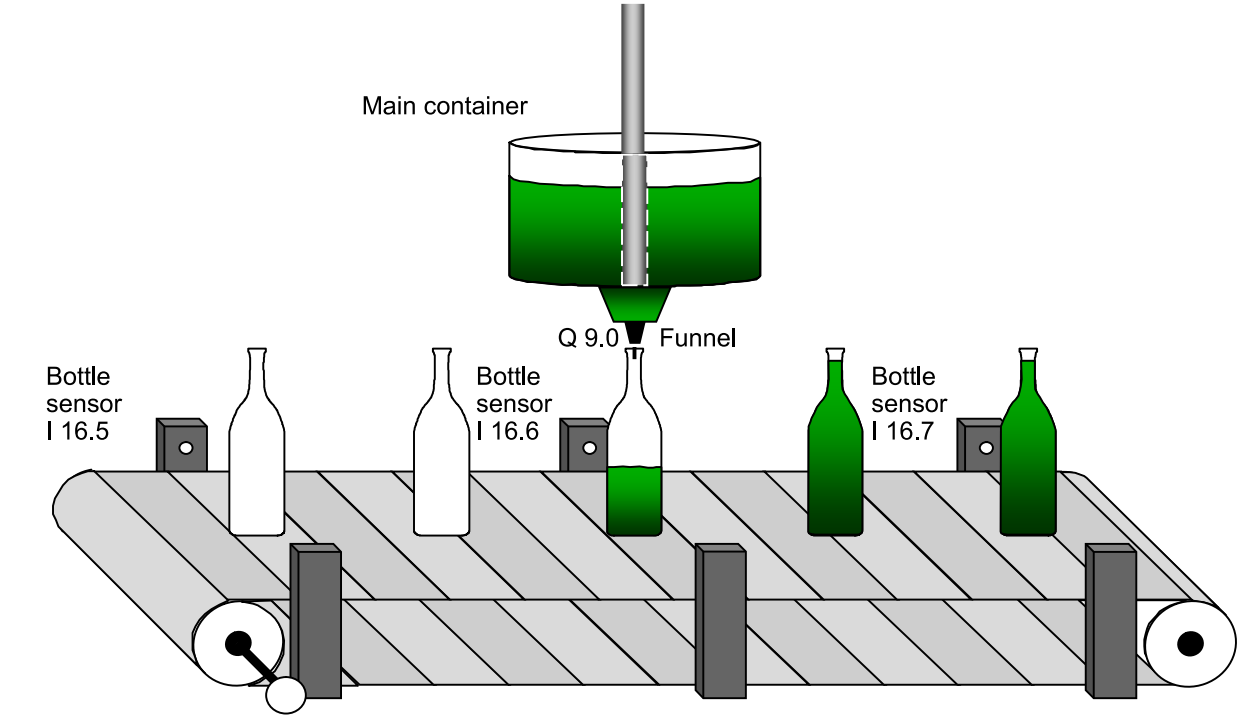
If the DI register or the AR2 register is overwritten within the FB processing, then access to the instance data (input, output, in/out parameter and static variables) can no longer be made within the FB..

UC, CC Instruction

Blocks can also be called with the RLO independent instruction UC or with the RLO dependent instruction CC.

UC and CC can only be used when the called FB has no instance data. That is, neither block parameters nor static variables were declared in the declaration section.

Exercise 6: The Conveyor Model as Bottling Plant



SIMATIC S7

Date: 30.06.2011
File: PRO2_06E.14

SITRAIN Training for Automation and Drives

Task

A bottling plant is to be automated in the following task:

The plant can thus be operated in the two operating modes "Manual" and "Automatic". In addition, a corresponding logic for switching on and switching off the system exists.

- Switching On and Off: The plant can be switched on using the input I 0.0 (momentary-contact switch with NO functions). The plant is switched off again using the input I 0.1 (switch with NC functions). When the plant is switched on, the LED at output Q 8.1 is lit up.
- Choosing the operating mode: When the plant is switched on, an operating mode can be chosen

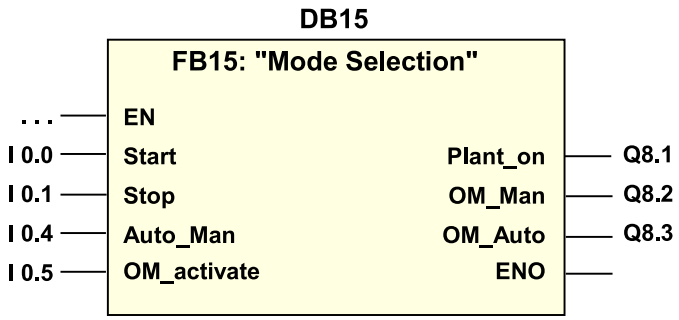
  Manual mode is preset with I 0.4 = 0 and automatic mode is preset with I 0.4 = 1. The selected mode is adopted with a pulse at input I 0.5. The selected mode is indicated (Manual = Q 8.2, Automatic = Q 8.3).

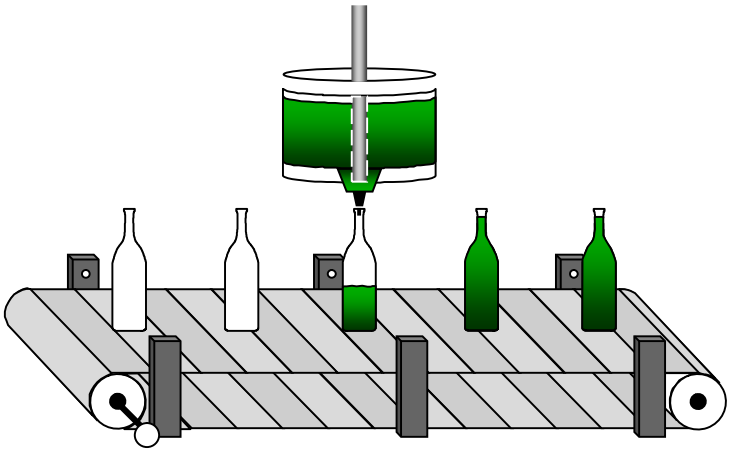
  When the mode is changed or the plant is off, the mode previously selected is deselected.
- Conveyor control in manual mode: In manual mode, the conveyor can be moved forwards with the momentary-contact switch I 0.2 (Q 20.5) and backwards with I 0.3 (Q 20.6).
- Conveyor control in automatic mode: In the automatic mode, the drives motor (Q 20.5) switches on and remains on until the off switch (I 0.1)is depressed and it is stopped or until the sensor (I 16.6) detects a bottle in the filling position.
- Filling a bottle: When a bottle is under the filling station (I 16.6=1), filling begins. The filling procedure is simulated for a period of 3 seconds and is indicated at the output Q9.0.
- Counting bottles: Another sensor is used to record the filled bottles. Bottle sensor I 16.7 records the full bottles. The full bottles are counted from the time the plant is switched on and are displayed at digital display QW 12.

Exercise 6.1a: Bottling Plant - Mode Section

Plant ON/OFF
I 0.0: Start (NO, mom.-cont. switch)
I 0.1: Stop (NC)
Q8.1: Plant_on

Manual/Automatic mode
I 0.4: Automatic/Manual
I 0.5: Adopt mode
Q8.2: Manual mode selected
A8.3: Automatic mode selected





SIMATIC S7

Date: 30.06.2011
File: PRO2_06E.15

SITRAIN Training for Automation and Drives

Note

The controlling of automation objects (drives, conveyor belts, etc.) takes place using function blocks.

So as not to infringe on the principles of structured programming, you should not directly access global addresses such as inputs, outputs, bit memories etc. within a function block. Every information transfer with signals of the process or with other blocks of the user program must take place using the parameters of the block.

Only in the call of a block in the highest level, that is, in the associated organization block, can you assign the process input signals or output signals directly to the block parameters.

Controlling the Mode: FB15

You will find a symbol table in the program folder Chap_06_1 of the project "Pro2_e_x53" for exercise 6.1.

First of all, create an FB15 "Mode Selection", in which the complete logic for ON/OFF and the selection of the plant mode is implemented.

FB15 "Mode" has the following input and output parameters:

#Start (in, BOOL): The plant is switched on using *#Start* (1-active).

#Stop (in, BOOL): The plant is switched off using *#Stop* (0-active). The input parameter *#Stop* has priority over *#Start*.

#Plant_on (out, BOOL): When the plant is switched on # *Plant_on* is set to "1".

You can select a mode when the plant is switched on.

#Auto_Man (in, BOOL): *#Auto_Man*=0 selects manual mode
#Auto_Man=1 selects automatic mode

#OM_activate (in, BOOL): The selected mode is adopted when there is a pulse at *#OM_activate*.

The active mode is indicated by FB15 in the following output parameters:

#OM_Man (out, BOOL): manual mode is active

#OM_Auto (out, BOOL): automatic mode is active

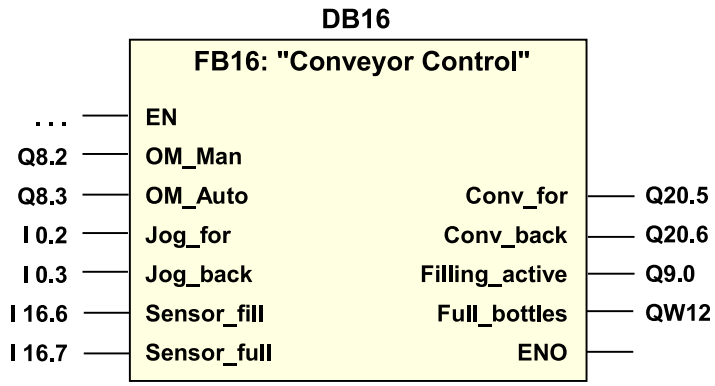
Exercise 6.1b: Bottling Plant - Conveyor

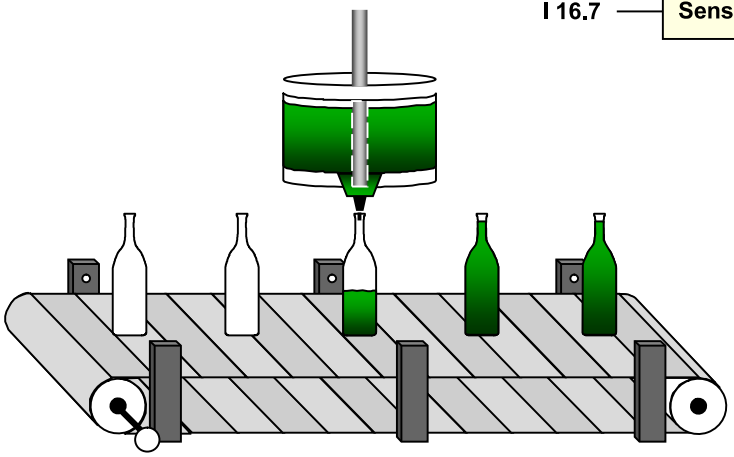
Manual mode

I 0.5:	Jog forward
I 0.6:	Jog backward
Q20.5:	Conveyor forward
Q20.6:	Conveyor backward

Automatic mode

I 16.6:	Sensor: Filling location
I 16.7:	Sensor: Count bottles
Q9.0:	Filling active
QW12::	Display of full bottles





SIMATIC S7

Date: 30.06.2011
File: PRO2_06E.16

SITRAIN Training for Automation and Drives

Controlling the Conveyor: FB16

Create an FB16 "Conveyor Control", containing the complete logic for the control of the conveyor in manual and automatic mode.

FB16 has the following input and output parameters:

#OM_Man (in, BOOL): Conveyor is operated in the manual mode.

#OM_Auto (in, BOOL): Conveyor is operated in the automatic mode.

#Jog_for (in, BOOL): The conveyor can be moved forwards in the manual mode using this input. This input is of no importance in the automatic mode.

#Jog_back (in, BOOL): The conveyor can be moved backwards in the manual mode using this input. This input is of no importance in the automatic mode.

#Sensor_fill (in, BOOL): Indicates that an empty bottle has reached the filling position.

#Sensor_full (in, BOOL): Indicates that yet another full bottle has passed the barrier for counting full bottles.

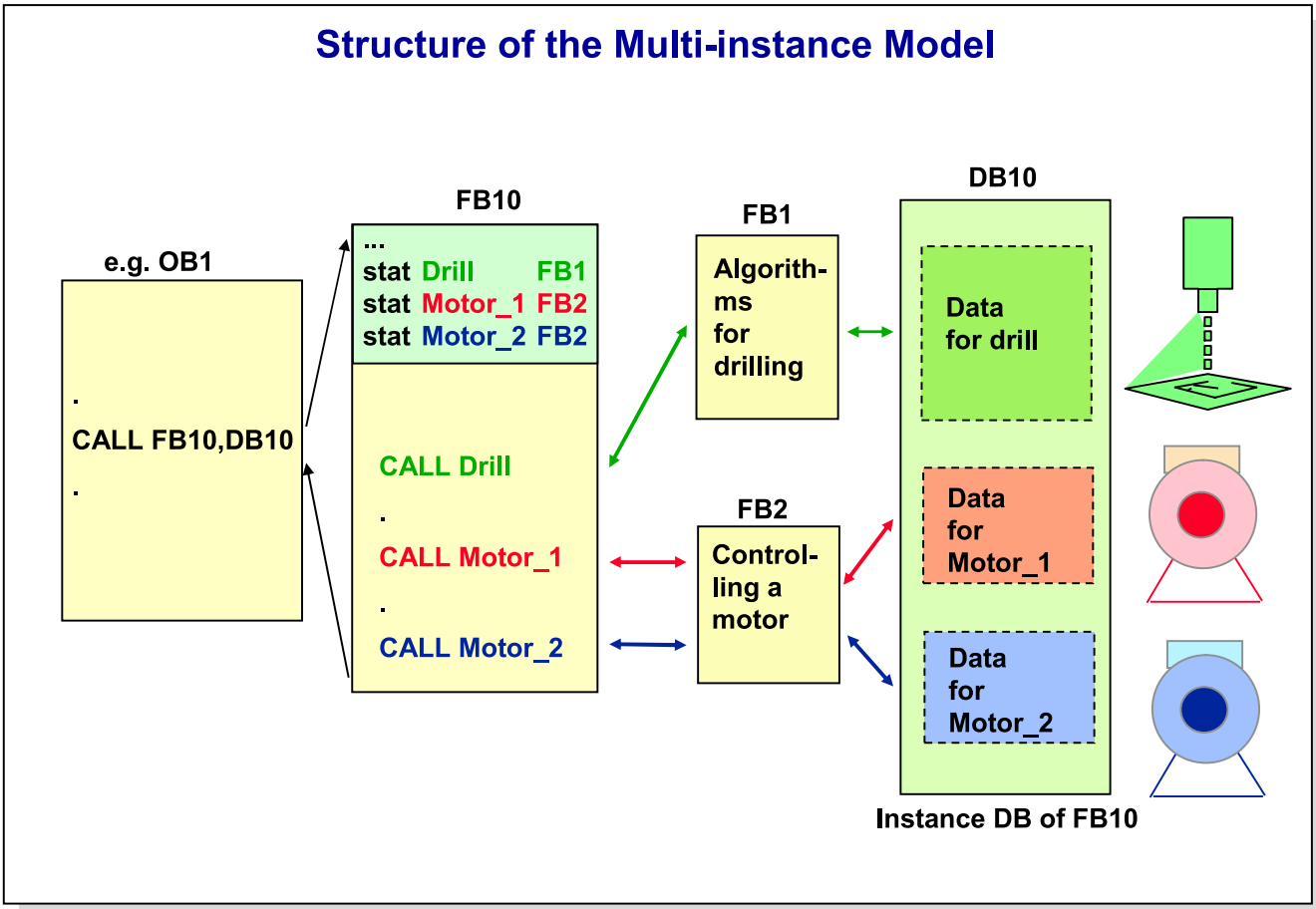
#Conv_for (out, BOOL): Delivers the control signal for operating the conveyor forward.

#Conv_back (out, BOOL):Delivers the control signal for operating the conveyor backward.

#Filling_active (out, BOOL): Indicates that filling is currently active.

#Full_bottles (out, WORD): Gives the number of filled bottles since the last switch on in BCD code.

Call both of these blocks along with the associated instance data blocks DB15 and DB16 in OB1 and assign the parameters of the FBs with the signals for the operator panel (simulator) or with the signals for the process (conveyor) as given above in the two pictures.



Multi-instance Model

In addition to instantiation of function blocks by specifying an instance DB in an FB call, STEP 7 also supports the explicit declaration of FB instances within a higher-level function block.

For this, instances of the called function blocks are declared with data type FB1 or FB2 using symbolic identifiers (Drill, Motor_1 and Motor_2). This takes place in the declaration section of the calling FB 10 function block in the section "*static variable*". Within the higher-level function block, the individual instances are then called using their symbolic identifier. The higher-level FB10 function block must however be called with its own instance DB (DB10).

In the creation of the higher-level instance DB, STEP 7 makes sure that the data areas required for the individual instances are set up in the data area of the higher-level FB10.

In the call of the individual instances using the symbolic names, the CALL macro makes sure that the AR2 register is set to the beginning of the data area assigned to the instance so that the parameters and local variables of the instance are also accessed during the processing of the called function block.

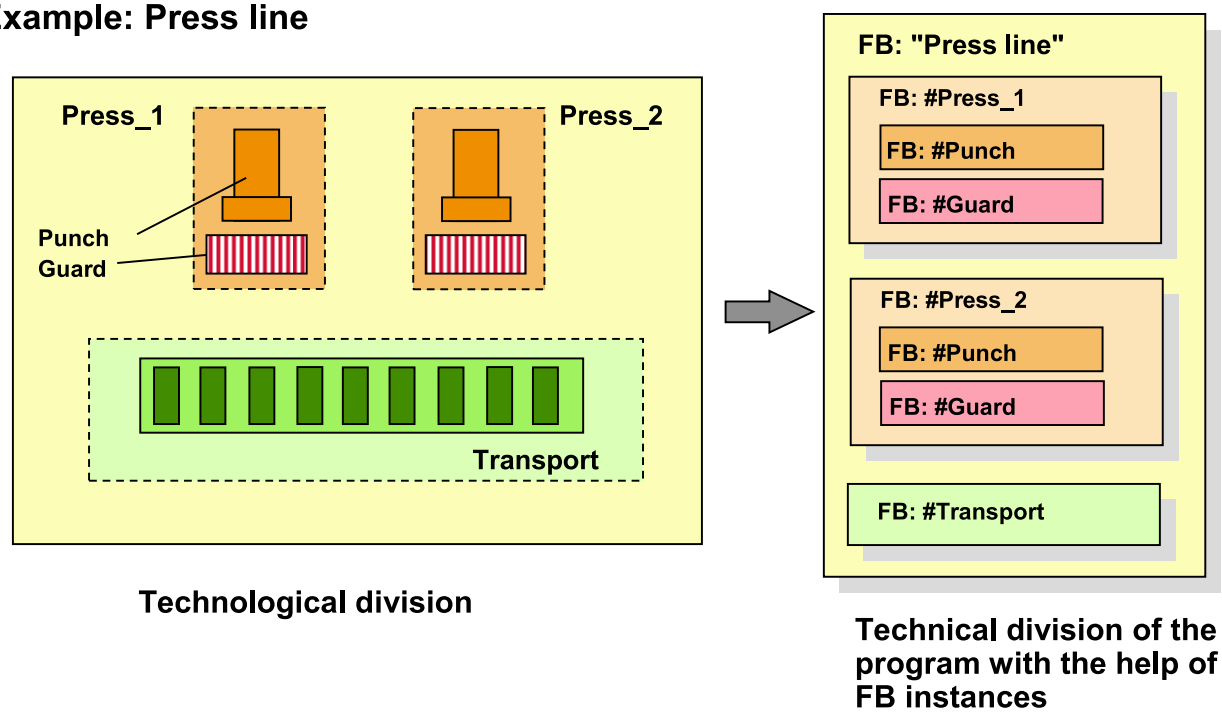
Benefits

The use of the multi-instance model has the following benefits:

- The individual instances do not require their own data block every time. Within a call hierarchy of function blocks, only one instance DB is "wasted" in the call of the "outer" function block.
- The multi-instance model "welds" a function block and an instance data area into one object (FB instance), that can also be handled as one unit. The programming person does not have to take care of the management (creation, addressing) of the individual instance data areas. He must simply provide an instance DB for the "outer" FB.
- The multi-instance model supports an object-oriented programming style.

Object-oriented Programming using Multi-instances

Example: Press line



Technological division

Technical division of the program with the help of FB instances

Process Units

Process units are physical objects in the process, such as units of an assembly line (transport belt, processing stations) or a complete machine or parts of a machine (the complete press or the punch or the guard of a press, for example).

Process units are used as logical identification criterion. They are, as a rule, of a hierarchical design. That way, process units can, in turn, contain sub units (e.g. the unit "Press" contains the units "Punch" and "Guard"). Process units can in this way be configured from smaller sub units. (Aggregation).

Object-oriented Programming Style

You can implement an object-oriented programming style with the help of function blocks. The technical description of a process unit or process sub unit program is made with an FB instance. The division of the user program into units is achieved by declaring lower-level FB instances within a higher-level FB.

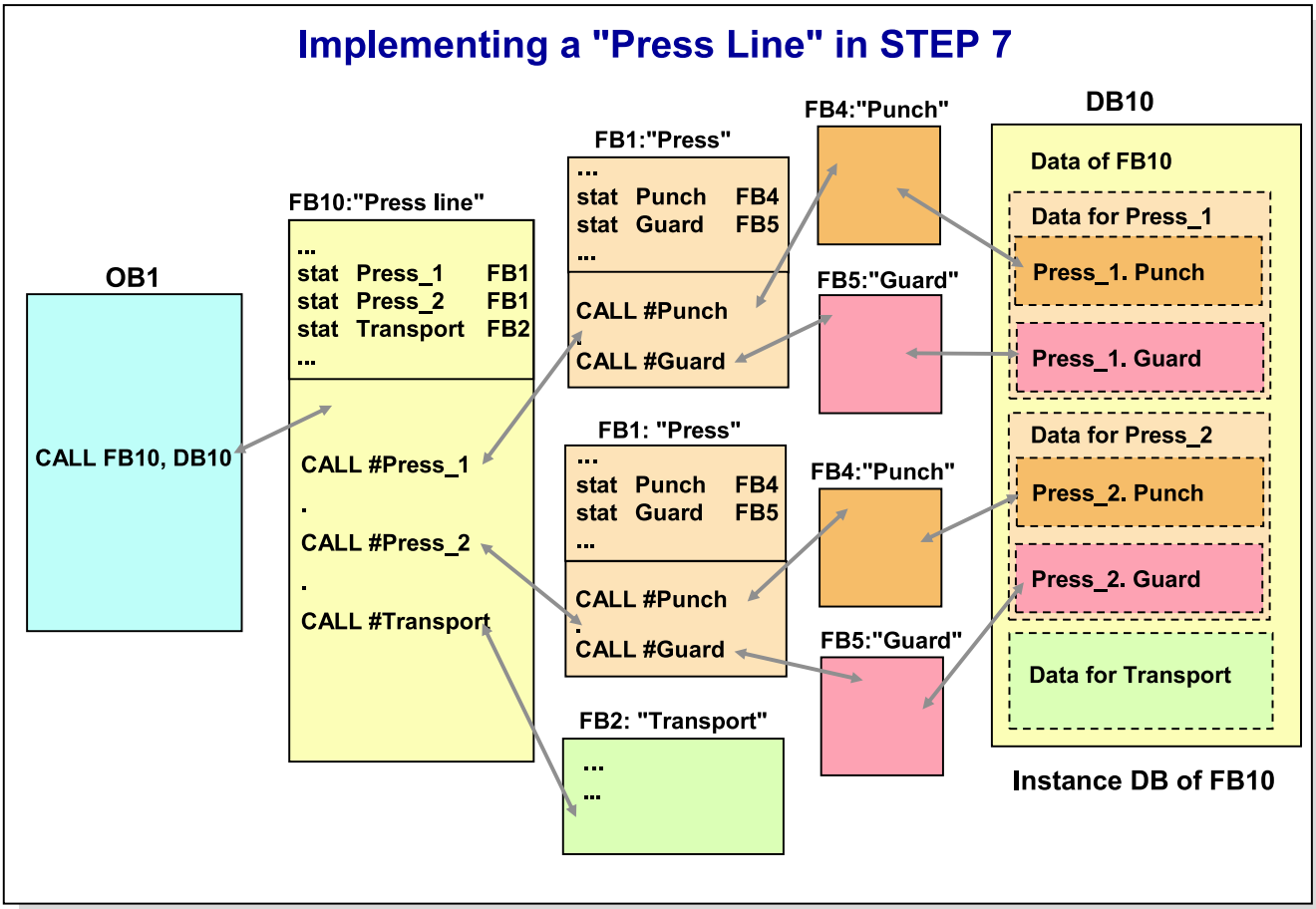
In this way, the same division into process units is achieved in the user program as in the existing system or machine. (Concept of object-oriented programming using aggregation).

Re-usability of Software

This hierarchical concept offers a high measure of re-usability of one-time generated software and thus offers a great savings potential in the creation, modification and upkeep of user programs:

- Whenever a manufacturer creates a process subunit (valve, motor, etc.), he also delivers an FB for controlling this process subunit.
- Whenever such a physical process unit is built into the next larger unit, an FB instance of the unit is also declared in the FB of the higher-level unit.

FBs are the basic components of a control program. In the planning stages of a program, they have the same task as the integrated circuits (ICs) in printed circuit board manufacturing. The structure of user programs consists of pre-fabricated FBs that must simply be interconnected.



Multi-instance Model

When a multi-instance model is used, the instance DB contains the data for several function blocks of a call hierarchy. For this, the instances of the called function blocks are declared using symbolic identifiers in the *stat. Var.* section of the declaration section of the calling FB.

The FB instance for the "furthest out" process unit (here: FB10 "Press line") must be called absolutely or symbolically along with the specification of its own instance DB (here: DB10).

Declarations

In the section *stat. Var* of the declaration part of FB10 ("Press line"), two instances (variables) of the data type FB1 ("Press") with the names *#Press_1* and *#Press_2* are declared, as well as an instance of the data type FB2 ("Transport") with the name *#Transport*.

In the declaration part of FB1 both an instance of FB4 ("Punch") with the name *#Punch* and an instance of FB5 ("Guard") with the name *#Guard* are declared.

In the instruction section of FB1 ("Press"), the respective FB instances are then called using the symbolic names *#Punch* and *#Guard* that were declared in the declaration section.

Note

The declaration of an instance in the declaration part of a function block works only if the FB, of which an instance is being declared, already exists.

When designing such a call hierarchy, those FBs that are to be called last in the chain have to be created first.

Multi-instance DB

The multi-instance DB has the same structure as the declaration parts of the function blocks concerned. If an instance is called in the instruction section, then it automatically accesses the data in the corresponding section of the instance DB (DB10).

Properties of the Multi-instance Model

Benefits of the Multi-instance Model:

- **Only one DB is required for several instances**
- **No additional management is necessary in the setting up of "private" data areas for the respective instances**
- **The multi-instance model makes an "object-oriented programming style" possible (re-usability by means of "Aggregation")**
- **Maximum nesting depth of 8**

Prerequisites for the FBs:

- **Direct (I, Q) access to the process signals within the FB is not possible**
- **Access to process signals or communication with other process units can only take place using FB parameters**
- **The FB can only remember process states in its static variables, not in global DBs or bit memories**

Note:

- **Instance data can also be accessed from "outside" e.g. in OB1: L "Press line".Press_2.Punch.<VarName>**

Benefits of the Multi-instance Model

With the multi-instance model you can store the respective data sections of several instances of one and the same call hierarchy in one single DB. That way only one DB is required for several instances.

With the multi-instance model no measures for the administration of local FB data are necessary except for the assignment of a mutual instance DB.

The multi-instance model supports the concept of object oriented programming. Code and data that are needed for the controlling of process units are summed up in FBs.

If a process unit consists of hierarchical sub units then exactly this structure can be reflected in the user program by means of the multi-instance model. The control program can be designed with FB instances the same way as the machine may consist of components.

STEP7 supports a nesting depths of 8 with the multi-instance model.

Prerequisites for Multi-instances

In order to use an FB as multi-instance without problems, the following points must be adhered to:

- For the purpose of process control, no direct access to global addresses of the CPU (such as inputs and outputs) are allowed. Each access to global inputs and outputs violates the re-usability.
- Communication with the process or with other program sections (FBs) must only be done using FB parameters.

  Only after integration of the FB into a higher-level unit, is the “assignment” of the FB via the parameter list carried out with the FB call.
- Statuses or other information about the unit to be controlled must be “remembered” by the FB in it’s own static variables

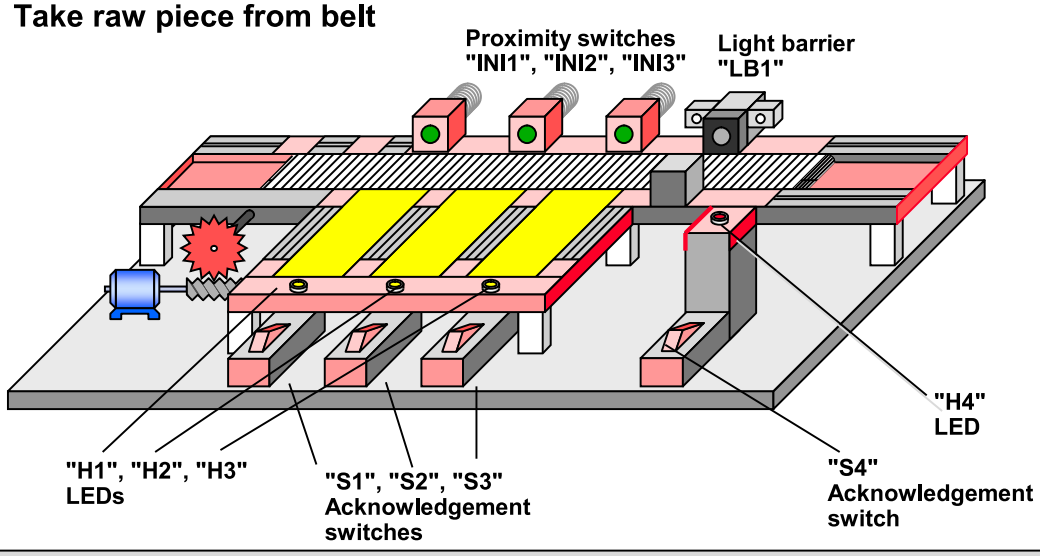
Exercise 6.2: The Conveyor Model as Assembly Line

Process sequence for the work station

- Processing the piece
- Processing finished
- Place piece on belt
- Wait for raw piece
- Take raw piece from belt

Process sequence for the transport belt

- Wait for finished piece
- Transport to final assembly
- Final assembly, insert raw piece
- Transport to station



SIMATIC S7

Date: 30.06.2011
File: PRO2_06E.21

SITRAIN Training for Automation and Drives

Goal

By means of an assembly line, the principle of solving a task is to be practiced using FB programming. A separate FB is used in each case for the control of work station 1 and the transport belt. The FB for the work station is to be multi-instance capable.

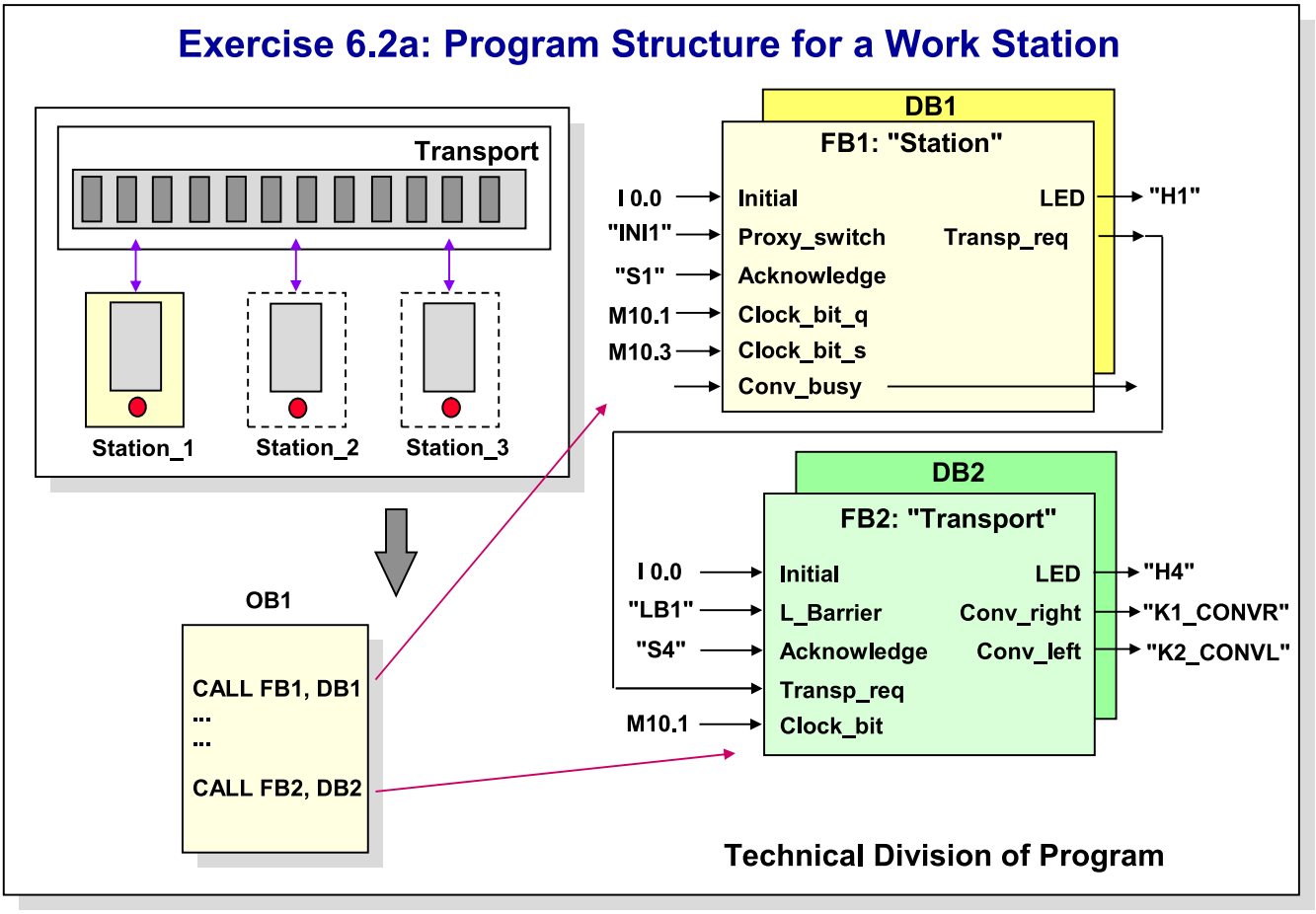
In the next exercise, the functionality of the assembly line has to be expanded to work stations 2 and 3 by means of the multi-instance model.

How the Conveyor Model Works

For the exercises for FB programming, the conveyor model is to work as an assembly line with the following functionality (for now, only one work station):

1. The system is in the initial state, that is, work station 1 has one piece that is currently being processed. This is indicated with a lit "H1" LED at Location 1.

   The transport belt, that is, "INI1" in particular and the final assembly, that is, "LB1" are not occupied. The conveyor motor is switched off.
2. After the piece has been finished, the operator acknowledges this with the acknowledgement switch "S1". The "H1" LED flashes.
3. The operator places the finished piece on the "empty" belt in front of proximity switch "INI1". The "H1" LED goes off.
4. The belt then transports the finished piece to the final assembly. LED "H4" flashes during transport. When the final assembly is reached, the "H4" LED lights continuously.
5. The operator at the final assembly takes the finished piece from the belt and places a new raw piece on the belt. He then acknowledges this with the "S4" switch.
6. The belt transports the new raw piece back to work station 1. LED "H4" flashes during transport. When the proximity switch "INI1" is reached, the work station LED "H1" begins to flash.
7. The operator can take the raw piece from the belt and place it on work station 1 and begin to process once again. The "H4" LED is lit continuously once more. The work process begins again with Step 1.



Task

In the first step, only one work station in the assembly line is to be implemented for the time being. The control of the entire system is therefore divided into two process units:

- Station: Control of the first work station, implemented by FB1 with the global symbolic name "Station" (global symbol table)
- Transport: Control of the transport belt, implemented by FB2 with the global symbolic name "Transport"

For the control of the entire assembly line, both FBs each with their own instance DB are then called in OB1.

What to Do

For exercise 6.2, you will find the relevant FB1 and FB2 function blocks as well as the corresponding symbol table in the program folder Chap_06_2 of the project "Pro2_e_x53".

1. Open the project "Pro2_e_x53" and insert the hardware configuration of your training station.
2. For the CPU, assign parameters for a clock bit in MB10 and then download the parameter assignment data of the hardware configuration into the CPU of the training station.
3. Call FB1 with the instance DB DB1 in OB1 and assign the associated process signals to the interface parameters.

   Assign input parameter *#Clock_bit_q* with M10.1 and *#Clock_bit_s* with M10.3 .
4. In OB1, program the FB2 call (instance DB DB2) after the FB1 call.

   Assign the input and output parameters according to the description of FB1 and FB2.
5. Download the blocks to the CPU and test the functioning of your program.

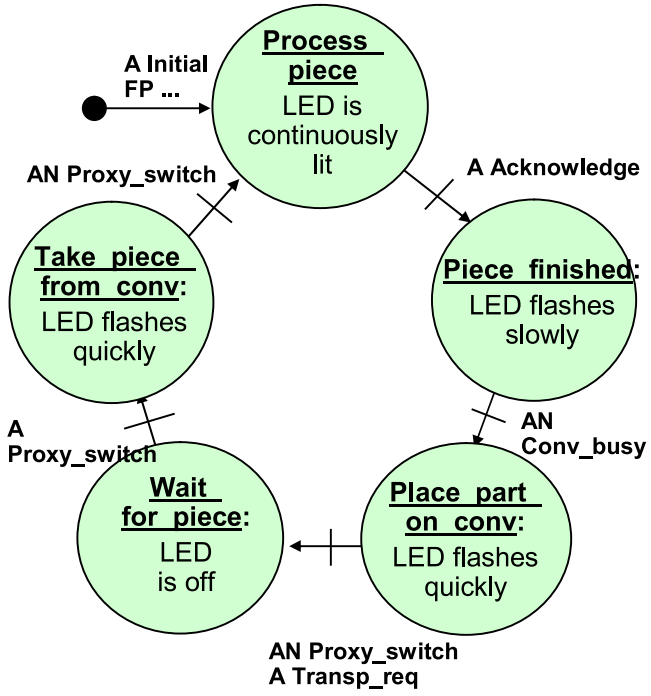
How the FB1 "Station" Works

- **Declarations in FB1:**

FB1: "Station"

	IN Parameters:	Data type:
→	Initial	BOOL
→	Proxy_switch	BOOL
→	Acknowledge	BOOL
→	Clock_bit_q	BOOL
→	Clock_bit_s	BOOL
	OUT Parameters:	
←	LED	BOOL
←	Transp_req	BOOL
	I/O Parameter:	
↔	Conv_busy	BOOL
	Stat. Var.:	
	State STRUCT	
	Process_piece	BOOL
	Piece_finished	BOOL
	Place_part_on_conv	BOOL
	Wait_for_piece	BOOL
	Take_piece_from_conv	BOOL
	END_STRUCT	

- **State model:**



Initialization — With an impulse at input parameter *#Initial* , FB1 can be initialized with the state *#Process_piece*. Assign *#Initial* with I 0.0.

#Process_piece — The piece is processed in this state. The LED "H1" is continuously lit to indicate processing.

A transition to the state *#Piece_finished* occurs when the operator acknowledges the completion of the piece with the switch "S1" .

#Piece_finished — In this state, the LED flashes with the frequency from *#Clock_bit_s* (slower flashing frequency). The operator waits for the transport belt to be enabled (Important for the expansion to 3 stations!).

If the conv is empty (*#Conv_busy*=0), it is immediately defined as occupied (*#Conv_busy*=1) and goes into the state *#Place_piece_on_Conv*.

#Place_piece_ on_conv — In this state - the LED flashes with the frequency from *#Clock_bit_q* (faster flashing frequency) - the operator can place the piece on the conveyor. With the signal *#Proxy_switch*=1, the signal *#Transp_req*=1 is also set, thus triggering movement of the conveyor towards final assembly.

A transition to the state *#Wait_for_piece* occurs when the piece leaves the proximity switch (*#Proxy_switch*=0).

#Wait_for_piece — The operator waits for the arrival of a new raw piece in this state; the LED in front of the station is off.

With the arrival of a new raw piece (*#Proxy_switch*=1), the transition to the state *#Take_part_from_conv* takes place.

#Take_piece_ from_conv — In this state - the LED flashes with *#Clock_bit_q* (faster flashing frequency) - the piece can be taken from the conveyor.

A transition to the state *#Process_piece* takes place when the piece is accepted (*#Proxy_switch*=0).

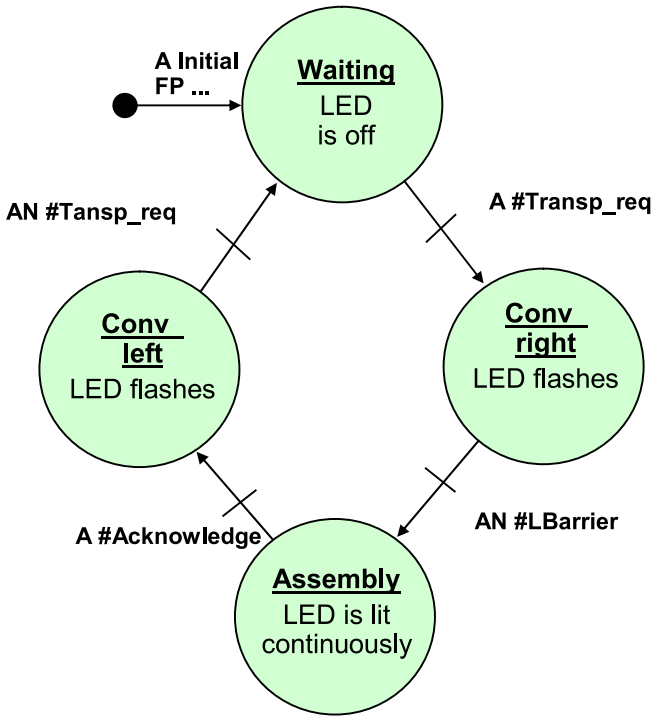
How the FB2 "Transport" Works

- **Interface of FB2:**

FB2: "Transport"

IN Parameters:		**Data type:**
→	**Initial**	**BOOL**
→	**L_Barrier**	**BOOL**
→	**Acknowledge**	**BOOL**
→	**Transp_req**	**BOOL**
→	**Clock_bit**	**BOOL**
OUT Parameters:		
←	**LED**	**BOOL**
←	**Conv_right**	**BOOL**
←	**Conv_left**	**BOOL**
Stat. Var.:		
State	**STRUCT**	
	Waiting	**BOOL**
	Conv_right	**BOOL**
	Assembly	**BOOL**
	Conv_left	**BOOL**
	END_STRUCT	

- **State model:**



Initialization

The FB2 can be initialized with the state *#Waiting* via the input signal *#Initial*. Assign the input parameter *#Initial* with I 0.0.

#Waiting

In this state, the transport belt waits for a finished piece that is placed on the conveyor by one of the stations. As long as the transport belt is in the state *#Waiting*, it is stopped and the "H4" LED is switched off.

With the State 1 of the signal *#Transport_req,* a transition to the state *#Conv_right* takes place.

#Conv_right

In this state, the piece is transported in the direction of final assembly. As long as the conveyor is moving, the "H4" LED flashes with the frequency (M10.1) given by the input parameter.

Final assembly is reached, that is, a switch to the state *#Assembly* takes place, when the finished piece passes the light barrier "LB1".

#Assembly

In this state, the operator exchanges the finished piece with a new raw piece. The "H4" LED is lit continuously in this state. The operator signals the completion of this task with the switch "S4".

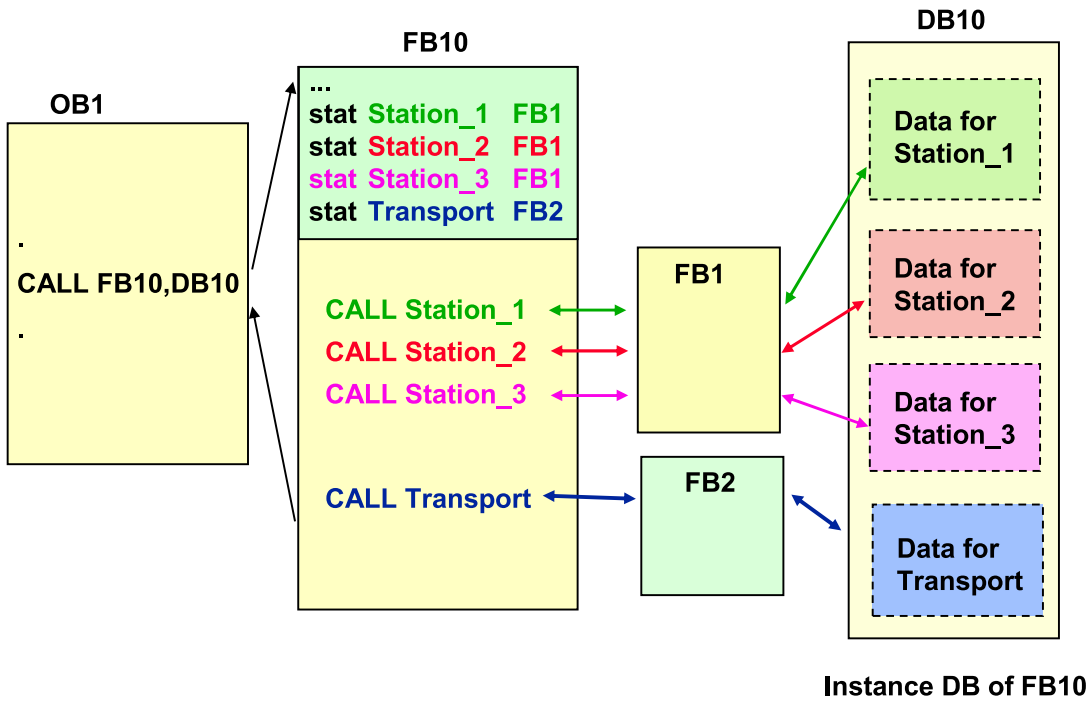
This signal also leads to the transition to the state *#Conv_left*.

#Conv_left

In this state, the raw piece is transported in the direction of the work station. As long as the conveyor is moving, the "H4" LED flashes with the frequency given by the input parameter *#Clock_bit*.

Transport is stopped when the input signal *#Transp_req* is reset. A transition to the state *#Waiting* also takes place.

Exercise 6.2b: Expansion to 3 Stations



SIMATIC S7

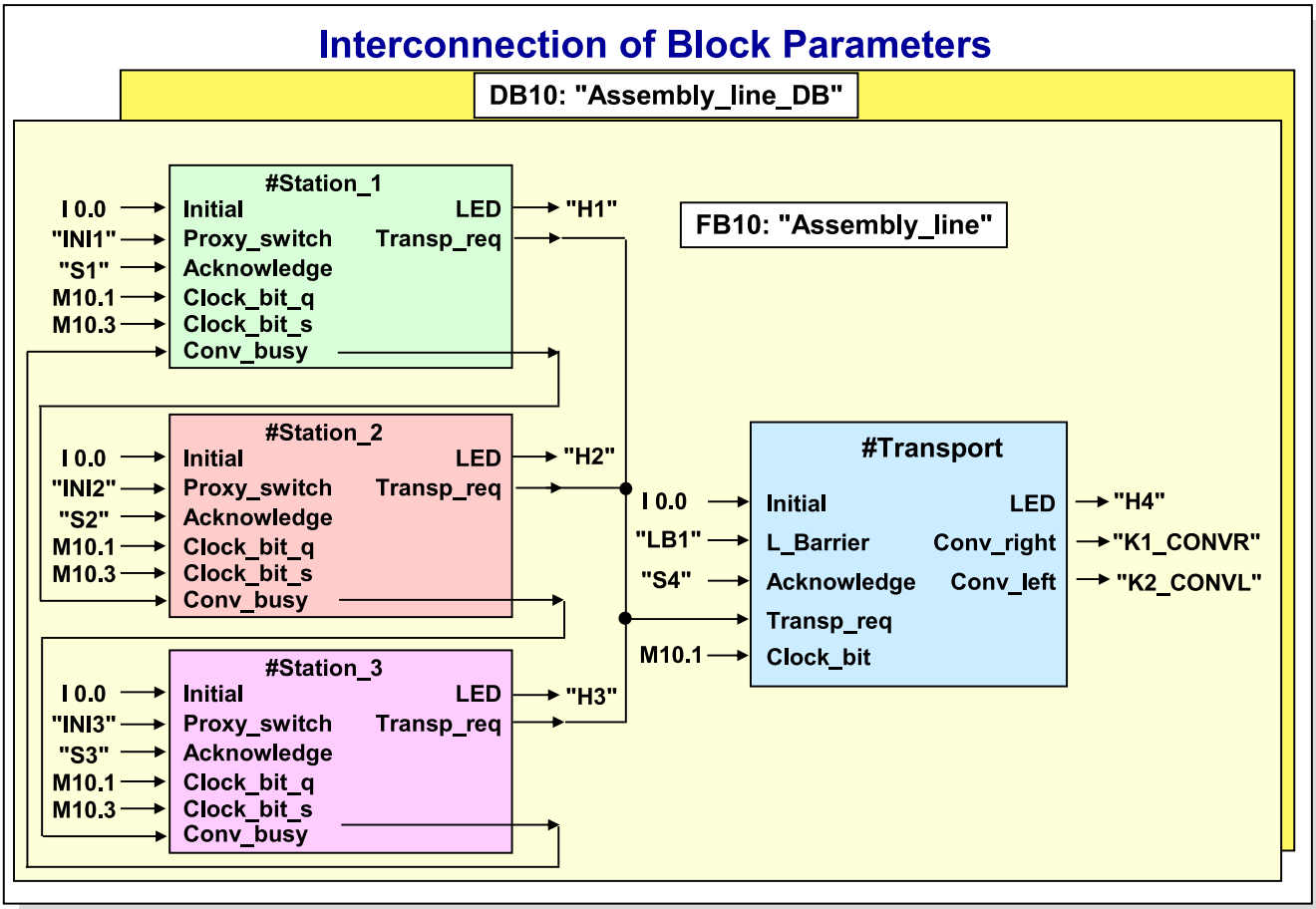
Date: 30.06.2011
File: PRO2_06E.25

SITRAIN Training for Automation and Drives

Program Structure

In the second part of this exercise, the full functionality of the conveyor model for all three work stations is to be achieved. For this, the control of the entire conveyor model (3 stations and one transport belt) is to be moved into one single FB (FB10).

Within the FB10, the control of the three work stations is implemented as separate instances of FB1 and the control of the transport belt is implemented as instance of FB2.



SIMATIC S7

Date: 30.06.2011
File: PRO2_06E.26

SITRAIN Training for Automation and Drives

What to Do

1. First of all create an FB10. In the section *stat. Var.*, declare the three instances of FB1 with the names: *#Station_1*, *#Station_2* and *#Station_3* and one instance of FB2 with the name: *#Transport*.
2. In FB10, first of all, call in sequence *#Station_1*, *#Station_2*, *#Station_3* and *#Transport* and interconnect the block parameters according to the schematic above.

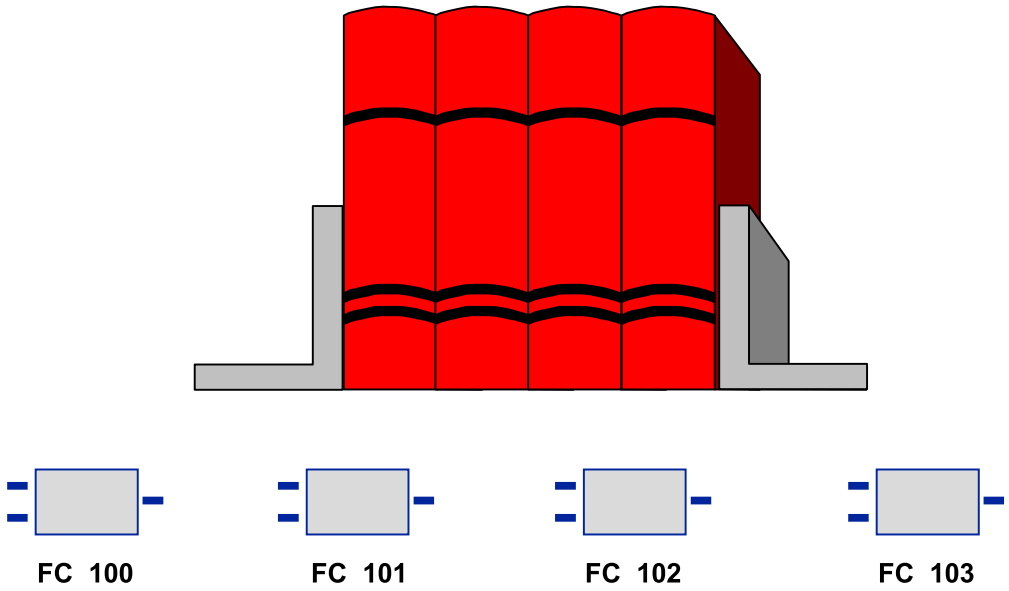
   Note the interconnection of the in/out parameter *#Conv_busy*. How can this be implemented? Can temporary or static auxiliary variables be used here?

   Also pay attention to the interconnection of the output parameter *#Transp_req* (logical OR) to the input parameter *#Transp_req* of the belt control. How can such an interconnection be implemented?
3. Create an explicit DB10 with associated FB10. Edit the DB10 and check its structure in the declaration and data view of the DB Editor.
4. Call the FB10 with instance DB DB10 in OB1.
5. Download the participating blocks to the CPU and test the result.

Questions

- What are the advantages and disadvantages of such an approach?
- How would the control have to be expanded, so that an "empty" assembly line can also be "filled" or a "filled" line can also be "emptied".

Using Libraries



SIMATIC S7

Date: 30.06.2011
File: PRO2_07E.1

SITRAIN Training for Automation and Drives

Contents **Page**

Interesting Facts about Libraries

Purpose:

- **Storage of re-usable program components**
- **Direct transferring to the CPU and testing is not possible**

Library Configuration:

- **A library can contain several program folders**
- **A library cannot contain any "Hardware"**
- **Every program folder contains:**
  - the folders "Blocks", “Sources", "Symbols"
  - the folder "Charts" (only for the option software: S7-CFC)

Use of Libraries:

- **With the SIMATIC Manager:**
  - libraries can be set up (but not with the same names as projects)
  - blocks and sources can be copied between libraries and projects
  - libraries can be archived
- **With the LAD/FBD/STL Editor**
  - network templates can be stored in libraries as sources and can then always be reused

Overview

Libraries are used to store re-usable program components for the SIMATIC S7/M7. The program components can be copied from existing projects into a library or they can be generated directly in the library independent of projects.

The same functionality as for projects is available for the generation of S7-Programs in a library - with the exception of Testing -.

Configuration

Just like projects, libraries are configured in a hierarchical manner:

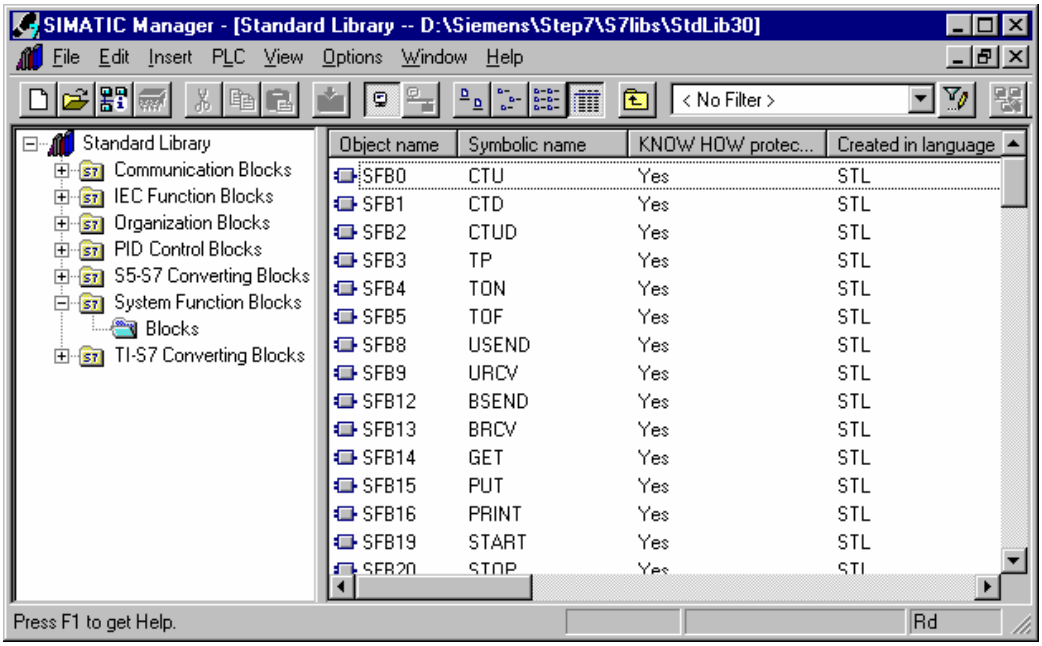
- libraries can contain S7-Programs.
- an S7-Program can contain exactly one *Blocks* folder, one *Sources* folder, one *Charts* folder as well as one *Symbols* (symbol table) object.
- The *Blocks* folder contains the blocks that can be loaded into the S7-CPU. The variable tables (VATs) and user defined data types contained in them are not loaded into the CPU.
- The *Sources* folder contains the sources for the programs generated in the diverse programming languages.
- The *Charts* folder contains the CFC-Charts (only for the option software S7-CFC)

When you insert a new S7-Program, a *Blocks* and a *Sources* folder as well as a *Symbols* object are automatically set up in it.

Use of Libraries

Blocks that are to be used again and again can be stored in libraries. From there, they can be copied into the relevant user program and be called by other blocks.

Configuration and Contents of the Standard Library



Date: 30.06.2011
File: PRO2_07E.3

Introduction

Two standard libraries are automatically installed on the hard disk with the installation of the STEP7 software:

- the standard libraries stdlibs(V2) for Version 2 and
- Standard Library for Version 3 and above.

Opening a Library

To open a library, use either the menu option: *File -> Open* or the associated icon in the toolbar.

A further dialog then pops up in which you can select the desired project or the desired library.

Standard Library

The Standard Library contains the following S7-Program folders:

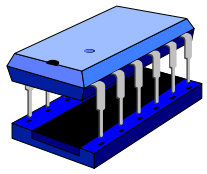
- Communication Blocks: contains the functions for connecting distributed I/O when using an S7-300 PROFIBUS CP
- IEC Function Blocks: contains blocks for IEC functions e.g. for handling the data types DATE_AND_TIME and STRING (see Chap. 5).
- Organization Blocks: contains all OBs with the symbolic identifiers for the start information
- PID Control Blocks: with function blocks for PID control
- S5-S7 Converting Blocks: with the standard blocks that are required in the converting of S5-Programs to S7
- System Function Blocks: contains all system functions (SFCs and SFBs) of S7-300/400.
- TI-S7 Converting Blocks: with generally usable standard functions, for example, scaling of analog values, etc.

Notes

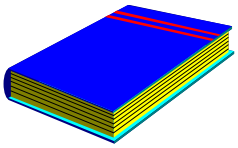
To an extent, additional libraries are created during the installation of option packages.

A description of the S7 Libraries PID and TI - S7 Converting Blocks is located under:*Taskbar ->SIMATIC ->S7 manuals ->PID Control, Standard Functions 2.*

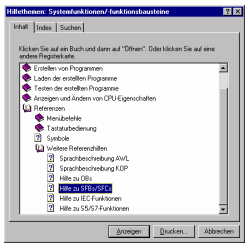
Interesting Facts about System Functions



The system functions (SFCs and SFBs) are stored in the CPU‘s operating system



System Software Reference Manual for S7-300/400 System Functions and Standard Functions



Extensive Online help available in the STEP 7 software

Date: 30.06.2011
File: PRO2_07E.4

Introduction

Functionality, that cannot be implemented with STEP 7 instructions (for example DB creation, communication with other PLCs, etc.) can be implemented in STEP7 with the help of system functions (SFCs) or system function blocks (SFBs).

SFCs and SFBs are blocks that are stored in the CPU’s operating system instead of in the user memory. For that reason, the actual instruction part is not transmitted but only the SFC’s or SFB’s declaration part during the reading out of an SFC or SFB from the CPU.

With the help of the STL/LAD/FBD Editor, the "block" that is read out can be opened and the declaration part displayed. A reverse transmission of SFCs and SFBs into the CPU is, however, not possible.

In the user program, the SFBs and SFCs can however be called just like FBs or FCs using the CALL instruction. With SFBs, a user DB must also be specified as instance DB for the SFB for that reason.

Which SFBs and SFCs are available depends individually on the PLC system (S7-300 or S7-400) used and on the CPU installed. The blocks have, however, regardless of whether they are called in an S7-300 or S7-400, the same numbers, the same functionality and the same call interface.

Manual

An extensive description of the system functions can be found in the manual:

- The System Software Reference Manual for S7-300/400, System Functions and Standard Functions.

Online Help

There is also an extensive description of the system functions in the STEP 7 software. Call the help menu in the program editor and select the entry:

- Help Topics -> Calling Reference Helps (LAD, FBD, STL, Blocks ...) -> Jumps to Language Descriptions and Help on Blocks and System Attributes -> Help on SFBs/SFCs.

Overview of the System Functions (Part 1)

Function Group	Function	Block	S7-300	S7-400
Copy and Block Functions	Blockmove	SFC 20	X	X
	Preset field	SFC 21	X	X
	Generate DB	SFC 22	X	X
	Delete DB	SFC 23	-	X
	Test DB	SFC 24	X	X
	Compress	SFC 25	-	X
	Substitute value in Accu 1	SFC 44	X[1)]	X
Program Control	Multicomputing interrupt	SFC 35	-	X[2)]
	Trigger cycle time	SFC 43	X	X
	Stop state	SFC 46	X	X
	Delay (Wait)	SFC 47	X[1)]	X
Handling the Clock	Set clock time	SFC 0	X	X
	Read clock time	SFC 1	X	X
	Synchronize	SFC 48	-	X
Operating Hours Counter	Set the counter	SFC 2	X[1)]	X
	Start and stop	SFC 3	X[1)]	X
	Read out	SFC 4	X[1)]	X
	Read system time	SFC 64	X	X

1) not for the CPU 312IFM 2) only for innovated CPUs

Date: 30.06.2011
File: PRO2_07E.5

Copy Functions and Block Functions

- SFC 20 copies the contents of a memory area (source) into another memory area (destination)
- SFC 21 fills a memory area (destination field) with the contents of a specified memory area (source field)
- SFC 22 creates a DB without preset values in the work memory
- SFC 23 deletes a DB in the work memory and possibly in the load memory.
- SFC 24 determines if a DB is present in the work memory (with length).
- SFC 25 compresses the memory. When blocks are corrected, gaps occur in the memory that are removed during compressing.
- SFC 44 (call in OB 122) saves a substitute value for a faulty input module in the Accu. Usage in OB121 is also possible.

Program Control

- SFC 35 triggers, with multicomputing, the synchronized start of OB 60 on all CPUs
- SFC 43 starts CPU scan cycle monitoring anew
- SFC 46 brings the CPU to the Stop state
- SFC 47 implements wait times in the user program up to 32767 µs

Handling the Clock

- SFC 0 sets the date and time-of-day for the real-time clock on the CPU
- SFC 1 reads the current CPU date and time-of-day
- SFC 48 synchronizes all slave clocks present on a bus segment. The CPU with the call must be assigned parameters as the master clock

Operating Hours Counter

The CPU has a specific number of operating hours counters with which you can record the operating duration of the operating equipment, for example.

- SFC 2 sets the operating hours counter to a specified value
- SFC 3 starts and stops the operating hours counter
- SFC 4 reads the current number of operating hours and the status
- SFC 64 reads the CPU's system time. The system time is a free-running time counter that counts up every 10 ms (S7-300) or 1 ms (S7-400).

Overview of the System Functions (Part 2)

Function Group	Function	Block	S7-300	S7-400
Transfter data records	Write dynamic parameters Write defined parameters Assign parameters to modules Write data record Read data record	SFC 55 SFC 56 SFC 57 SFC 58 SFC 59	X X X X X	X X X X X
Time interrupts	Set Cancel Activate Scan	SFC 28 SFC 29 SFC 30 SFC 31	X[1)] X[1)] X[1)] X[1)]	X X X X
Delay interrupt	Start Cancel Scan	SFC 32 SFC 33 SFC 34	X[1)] X[1)] X[1)]	X X X
Synchronous errors	Mask errors Demask errors Read status register	SFC 36 SFC 37 SFC 38	X X X	X X X
Interrupt errors and Asynchronous errors	Cancel new interrupts Enable new interrupts Delay new interrupts Enable higher priority interrupts	SFC 39 SFC 40 SFC 41 SFC 42	X X X X	X X X X

1) not for the CPU 312IFM

Transfer Data Records

There is a system data area with parameter and diagnostics data for the parameter-assignable modules. This area contains data records from 0 to 255 that can be read or written.

- SFC 55 transfers the dynamic parameters to the addressed module. The parameters in the SDB on the CPU are not overwritten.
- SFC 56 transfers the parameters (data record RECNUM) to the module.
- SFC 57 transfers all data records from the SDB to the module.
- SFC 58 transfers the RECORD data record to the module.
- SFC 59 reads out the RECORD data record from the module.

Time Interrupts

The blocks are used for time-of-day controlled processing (OB 10 to 17). You can define the starting point either with the STEP 7 software or with the following system functions.

- SFC 28 sets the start date and time-of-day of a time-of-day controlled OB.
- SFC 29 deletes the start date and time-of-day of an OB (OB 10 to OB 17).
- SFC 30 activates the specified time interrupt OB.
- SFC 31 queries the status of a time interrupt OB.

Delay Interrupt

- SFC 32 starts in a delayed manner a delay interrupt (OB 20 to 27).
- SFC 33 cancels a delay interrupt.
- SFC 34 queries the state of a delay interrupt.

Synchronous Errors

- SFC 36 masks a synchronous error. That is, a faulty instruction does not lead to the call of the associated synchronous error OB.
- SFC 37 demasks the synchronous error
- SFC 38 reads out the Error Register.

Interrupt and Asynchronous Errors

- SFC 39 disables the processing of interrupt and asynchronous error events.
- SFC 40 enables the processing of interrupt and asynchronous errors.
- SFC 41 delays the processing of interrupt and asynchronous errors.
- SFC 42 enables once more the processing of delayed interrupt and asynchronous errors

Overview of the System Functions (Part 3)

Function Group	Function	Block	S7-300	S7-400
System Diagnostics	Read start info. Read partial system status list Write diagnostics buffer	SFC 6 SFC 51 SFC 52	- X X	X X X
Process image, I/O area	Update PII inputs Update PIQ outputs Set bit field in I/O Reset bit field in I/O	SFC 26 SFC 27 SFC 79 SFC 80	- - - -	X X X X
Addressing of Modules	Determine logical address Determine slot Determine all logical addresses	SFC 5 SFC 49 SFC 50	- X X	X X X
Distributed I/O	Trigger hardware interrupt Synchronize DP Slaves Read diagnostic interrupt Read user data Write user data	SFC 7 SFC 11 SFC 13 SFC 14 SFC 15	1) 1) 1) 1) 1)	1) 1) 1) 1)
Global Data Communication	Send GD package Receive GD package	SFC 60 SFC 61	- -	X X

1) only for CPU with DP interface, for example CPU 315-2 DP

System Diagnostics

- SFC 6 reads out the start information of the OB called last and that of the start-up OB
- SFC 51 reads out a part of the system status list. The list contains: system data, diagnostics state data, diagnostics data, and the diagnostics buffer.
- SFC 52 writes a user entry in the diagnostics buffer

Process Image, I/O Area

- SFC 26 updates the entire or a partial process-image input table
- SFC 27 transfers the entire or a partial process-image to the output modules
- SFC 79/ 80 are used to set and reset bit fields in the I/O area in conjunction with the Master Control Relay function

Addressing of Modules

- SFC 5 provides the logical address for a geographical address
- SFC 49 determines the geographical address from the logical address
- SFC 50 provides all the logical addresses of a module

Distributed I/O

- SFC 7 triggers a hardware interrupt at the DP master. The SFC 7 is called in the user program of an intelligent slave (CPU 315-2DP)
- SFC 11 synchronizes one or several groups of DP slaves
- SFC 13 reads the diagnostics data of a DP slave
- SFC 14 reads consistent data from a DP slave.
- SFC 15 writes consistent data to a DP slave

Global Data Communication

The global data are transferred cyclically (such as every 8th cycle) without using the SFC. With the help of the SFC 60 and 61 system function, sending and receiving data packages in the user program can be triggered.

- SFC 60 sends a global data package
- SFC 61 receives a global data package

Overview of the System Functions (Part 4)

Function Group	Function	Block	S7-300	S7-400
Data Exchange using SFB, configured connection	Query state	SFC 62	-	X
	Uncoordinated send	SFB 8	-	X
	Uncoordinated receive	SFB 9	-	X
	Send block	SFB 12	-	X
	receive block	SFB 13	-	X
	Read data from remote CPU	SFB 14	-	X
	Write data to remote CPU	SFB 15	-	X
	Send to printer	SFB 16	-	X
	Carry out complete restart	SFB 19	-	X
	Stop state	SFB 20	-	X
	Carry out restart	SFB 21	-	X
	Query device status	SFB 22	-	X
	Receive device status	SFB 23	-	X
Data Exchange using SFC, non-configured connection	Send data externally	SFC 65	1)	1)
	Recevie data externally	SFC 66	1)	1)
	Read data externally	SFC 67	1)	1)
	Write data externally	SFC 68	1)	1)
	Cancel connection externally	SFC 69	1)	1)
	Read data internally	SFC 72	1)	1)
	Write data internally	SFC 73	1)	1)
	Cancel connection internally	SFC 74	1)	1)

1) only for innovated CPUs

Data Exchange using SFBs

The SFBs are used to exchange data and to manage programs using configured connections. Depending on whether SFB calls are necessary at only one communications partner or at both, reference is made to one-sided or two-sided communication. SFBs exist only in the S7-400 operating system.

- SFC 62 determines the state of a local SFB instance and the state of the associated connection
- SFB 8 sends data to a remote partner without coordination
- SFB 9 is the counterpart to the SFB 8
- SFB 12 sends data (up to 64 KByte) to the remote partner with an ack.
- SFB 13 receives data from the remote partner with an acknowledgement
- SFB 14 reads data from a remote CPU (one-sided communication)
- SFB 15 writes data to a remote CPU (one-sided communication)
- SFB 16 sends data with formatting to a remote printer
- SFB 19 triggers a complete restart at the remote partner
- SFB 20 transfers the remote partner to the STOP state.
- SFB 21 carries out a restart at a remote partner
- SFB 22 provides the device status (operating state, error information) of the remote partner.
- SFB 23 receives the device status of a remote partner.

Data Exchange using SFCs

This communication - also referred to as basic communication - is implemented with S7- 300 as well as with S7-400. In comparison to SFB communication the following differences emerge:

- no connection configuration necessary
- no instance data blocks required
- maximum user data length 76 bytes
- dynamic connection configuration
- communication via MPI or K bus

Overview of the System Functions (Part 5)

Function Group	Function	Block	S7-300	S7-400
Integrated Closed-loop Control	Continuous control Step control Pulse shaping	SFB 41 SFB 42 SFB 43	3) 3) 3)	- - -
Plastics Technology	Call assembler block	SFC 63	1)	-
Integrated Functions	High speed counter Frequency meter A/B counter Positioning	SFB 29 SFB 30 SFB 38 SFB 39	2) 2) 3) 3)	- - - -
IEC Timer and IEC Counter	Pulse On delay Off delay Count up Count down Count up/down	SFB 3 SFB 4 SFB 5 SFB 0 SFB 1 SFB 2	X X X X X X	X X X X X X
Block Referenced Messages	Message without acknowledgement Message with acknowledgement Message with 8 wild cards Message without wild cards Send archive data Disable archive data Enable messages	SFB 36 SFB 33 SFB 35 SFB 34 SFB 37 SFC 10 SFC 9	- - - - - - -	X X X X X X X

1) only for CPU 614 2) only for CPU 312 IFM 3) only for CPU 314IFM

Integrated Closed-Loop Control

These blocks will be integrated in a later CPU version.

Plastics Technology

For the CPU 614 (S7-300), individual blocks can be created in the “C” language. The SFC 63 system function is used to call such blocks.

Integrated Functions

These blocks exist only for the CPU 312 IFM (S7-300).
You will find a description in the *Integrated Functions* manual.

- SFB 29 counts pulses on the integrated CPU inputs.
- SFB 30 is used to measure frequencies using the integrated inputs.

IEC Timer and Counter

This makes timer and counter functions available that correspond to the IEC 61131-3 standard. The remaining timer and counter functions are implemented as for SIMATIC S5, due to compatibility reasons.

The IEC timers and counters differ in a larger value range for the timer and counter values.

Block-Referenced Messages

These blocks are used to implement message systems for HMI systems, such as process control systems.

The messages are generated in the S7-CPU with this procedure and the respective messages including process variables are sent to the logged-on display devices.

A central acknowledgement concept is used. That is, when you acknowledge a message on the display device, a response is sent to the originating CPU. The information is distributed to all logged-on users from the CPU.

The messages are triggered for an edge change on the signal input.

Note

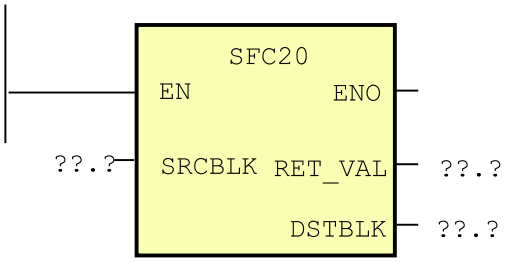
Block related messages are also possible with SFC18 "Interrupt_S" and SFC17 "Interrupt_SQ". This enables graphics-capable OPs to be able to process such messages (enabling of Interrupt_S messages with PROTOOL/ProAgent).

Call of System Functions and System Function Blocks

System Functions:

```
CALL  SFC   20
 SRCBLK :=
 RET_VAL:=
 DSTBLK :=
```

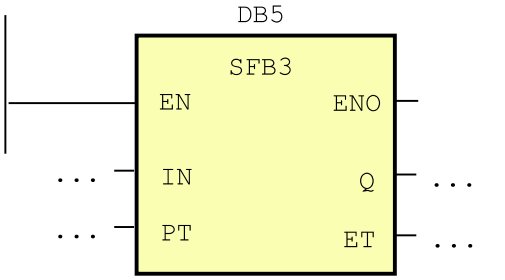
Call in STL



Call in LAD

System Function Blocks:

```
CALL  SFB   3,DB5
 IN:=
 PT:=
 Q  :=
 ET:=
```



System Function Blocks

An SFB system function block is a function block that is integrated in the operating system of an S7-CPU. As a result, SFBs are not downloaded into the CPU as part of the user program.

Just like FBs, SFBs are blocks “with memory“. They must be instantiated in the user program.

System Functions

A system function is a function that is integrated in the operating system of the S7-CPU. SFCs can be called from the user program just like FCs.

Just like FCs, SFCs are blocks “without a memory“.

Call

When a system function is called, the system function is automatically copied into the relevant user program.

In addition, all system functions are stored in the standard library Standard Library, S7-Program System Function Blocks. You can also copy the SFCs and SFBs into the user program from this library.

A completed symbol table (with English designations) exists in the library. The symbols of the blocks used are automatically copied into the user program’s symbol table from there.

Analyzing an Error Message

The scan of the BR-bit (binary result) returns RLO=0 for processing with fault and RLO=1 for fault-free processing.

- **Scan of BR for STL with A BR**
- **Scan in LAD/FBD using the output parameter ENO**

Most of the system functions return an error code with the following configuration in the output parameter RET_VAL (INT) :

- **RET_VAL=W#16#8 X Y Z**

Error class, error's individual number (SFC specific) or event number (general)

X>0: general error, X= no. of faulty parameter
X=0: SFC specific error occurred

"8" signals: error occurred

- **Example:**
  - W#16#8081 is an SFC specific error code.
  - W#16#823A is a general error code; the error was caused by parameter no. 2.

Error Information

A processed SFC shows you in the user program, whether the CPU could successfully execute the SFC function or not. You receive corresponding error information in two ways:

- in the BR bit of the status word and
- in the output parameter RET_VAL (return value).

Note

You should always proceed as follows before evaluating the SFC specific output parameter:

- first of all evaluate the BR bit of the status word.
- subsequently check the output parameter RET_VAL.

If a faulty processing of the SFC is signalized through the BR bit or a general error code is found in RET_VAL, you may not evaluate the SFC specific output parameter.

General Errors

The general error code indicates errors that can occur with all system functions. A general error code consists of the following two numbers:

- a parameter number between 1 and 127, whereby 1 indicates the first parameter, 2 the second parameter etc., of the SFC called.
- an event number between 0 and 127. The event number indicates a synchronous error.

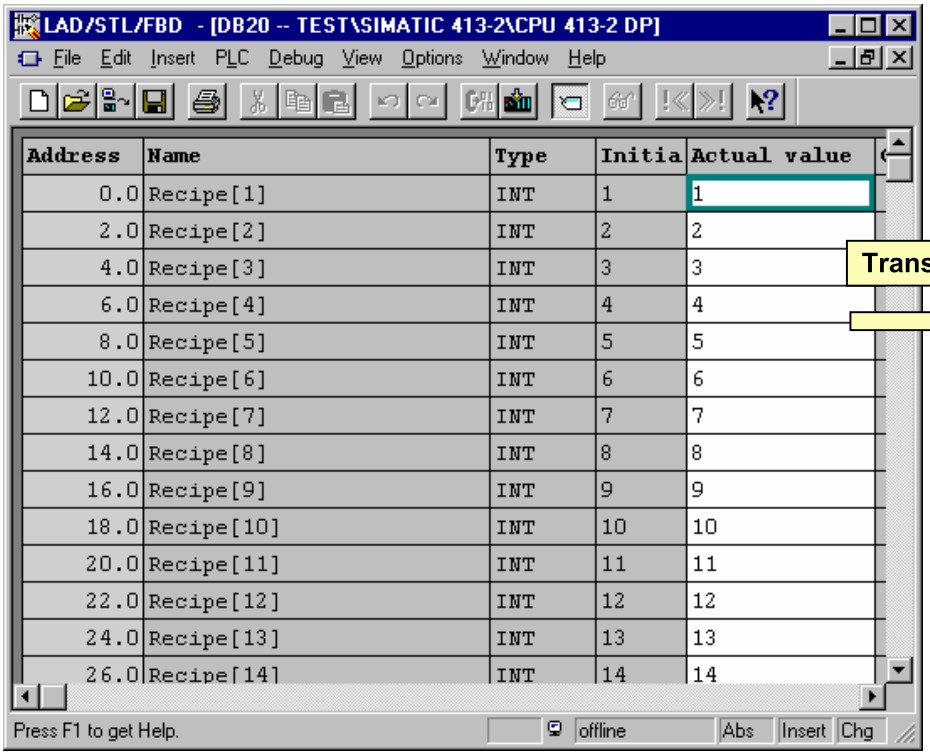
An extensive description of the general error codes can be found in the manual: "System Functions and Standard Functions" or in the Online Help.

Specific Errors

Several system functions (SFCs) have a return value that provides a specific error code. This error code indicates that an error that belongs to a specific system function occurred during the processing of the function.

A description of the specific error code can be found in the Online Help for the system functions.

Exercise 7.1: Generating a DB with an "UNLINKED" Attribute



Goal of the Exercise

You are to generate a data block with the "UNLINKED" attribute.

Task

Since the work memory only has a limited (usually too small) size, several data blocks with various recipe values are to be stored only in the load memory for recipe management.

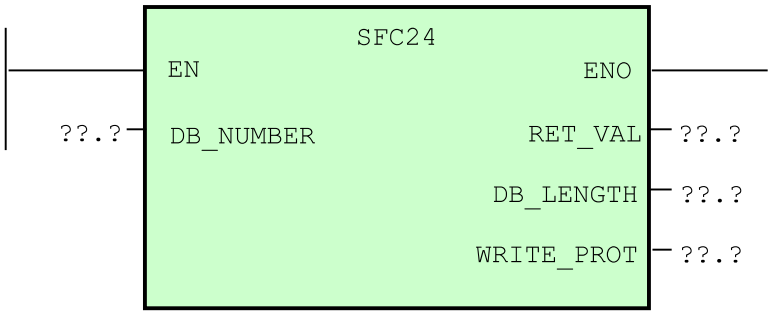
Only a work DB, in which the current recipe is stored, is present in the work memory. For a recipe change, the required values are copied from the load memory to the work memory.

With the help of the "UNLINKED" attribute you make sure that the data blocks are only saved in the load memory during a transfer from the PG to the CPU and that they are not automatically copied into the work memory.

What to Do

1. Insert a DB20.
2. Declare a variable "Recipe" of the ARRAY[1..20] type with an "INT" component type in DB20.
3. With the help of the menu option *View -> Data View,* switch to data view and initialize the individual field components in ascending sequence.
4. Select the block properties and set the "UNLINKED" attribute.
5. Transfer the DB 20 data block into the CPU.
6. What happens when you access the DB 20 with the instruction L DB20.DBW0 in the user program, for example?

Exercise 7.2: Testing the Data Block



Parameter	Declaration	Data Type	Memory Area	Description
DB_NUMBER	INPUT	WORD	I, Q, M, D, L, Const.	Number of the DB to be checked
RET_VAL	OUTPUT	INT	I, Q, M, D, L	Error information
DB_LENGTH	OUTPUT	WORD	I, Q, M, D, L	No. of data bytes, that the selected DB has
WRITE_PROT	OUTPUT	BOOL	I, Q, M, D, L	Information about the write protection ID of the selected DB (1 means write protected)

Goal of the Exercise

With the help of SFC 24 you can determine whether a specific data block exists in the work memory or not.

Task

With the help of SFC 24, create an FC 72 that determines if a DB exists in the work memory, in the load memory, or if it does not exist at all in the CPU:

- The FC 72 expects the number of the block to be tested in the input parameter #DB_NUM (WORD).
- The FC 72 returns the desired information in its return value #RET_VAL (INT) to the calling block:
  - 1: DB exists in the load memory
  - 0: DB exists in the work memory
  - -1: DB does not exist

Note

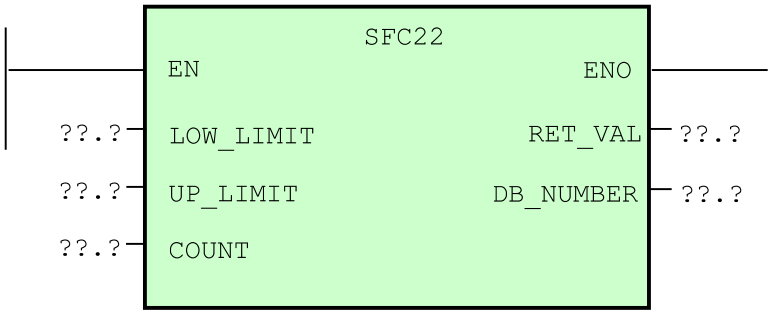
The SFC 24's #RET_VAL output parameter returns the following system specific error identifiers:

- w#16# 0000 no error occurred
- w#16# 80A1 incorrect number at parameter DB_NUMBER (0 or > max. DB number)
- w#16# 80B1 the DB does not exist in the CPU
- w#16# 80B2 the DB was generated with the keyword UNLINKED (is found only in the load memory)

What to Do

1. Create an FC 72 block.
2. Create an OB1, that with the help of FC 72 checks, whether the DB 20 exists or not. Display the returned information in the Simulator's display.
3. Download the blocks into the CPU and test your program.

Exercise 7.3: Generating a DB (SFC 22)



Parameter	Declaration	Data Type	Memory Area	Description
LOW_LIMIT	INPUT	WORD	I, Q, M, D, L, Const.	Smallest DB number
UP_LIMIT	INPUT	WORD	I, Q, M, D, L, Const.	Largest DB number
COUNT	INPUT	WORD	I, Q, M, D, L, Const.	No. of data bytes; an even number must be specified here
RET_VAL	OUTPUT	INT	I, Q, M, D, L	Return value of the SFC
DB_NUMBER	OUTPUT	WORD	I, Q, M, D, L	Number of the created DB, lies between LOW_LIMIT and UP_LIMIT

Goal of the Exercise

You are to become familiar with the setting up of a new DB per program.

Task

In start-up OB100, a DB 10 is to be generated in the work memory. Later, the recipe values are to be copied from the load memory into this DB.

What to Do

1. Create the OB 100.
2. Create the DB 10 with a length of 20 data words in the OB100. Use the SFC 22 for this (see above). Store the parameter #RET_VAL in MW 0 and the parameter #DB_NUMBER in the Simulator's segment display.
3. Download the OB 100 into the CPU and test your program.

Note

In copying between the load memory and the work memory, you must note that access to the "slow" load memory requires considerably more time than the access to the "fast" work memory.

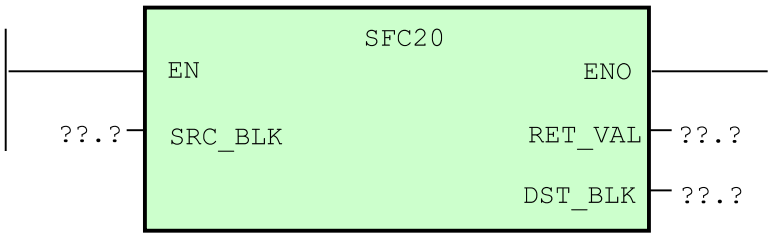
If larger amounts are copied within OB1, the cycle time, among other things, must be retriggered.

Error Identifiers

The SFC 22 system function provides the following error messages using the parameter #RET_VAL:

- W#16# 0000 no error
- W#16# 8091 nesting depth exceeded
- W#16# 8092 compressing is currently active
- W#16# 80A1 incorrect DB number
- W#16# 80A2 incorrect length
- W#16# 80B1 no DB number available (DB already exists)
- W#16# 80B2 not sufficient memory
- W#16# 80B3 not sufficient continuous memory (compressing required)

Exercise 7.4: Copying a DB from the Load Memory into the Work Memory (SFC 20)



Parameter	Declaration	Data Type	Memory Area	Description
SRC_BLK	INPUT	ANY	I, Q, M, D, L	Memory area to be copied (=source field). The source field can also be present in a non-sequence relevant data block in the load memory (DB, that was compiled with the keyword UNLINKED)
RET_VAL	OUTPUT	INT	I, Q, M, D, L	Return value of the SFC
DST_BLK	OUTPUT	ANY	I, Q, M, D, L	Memory area in which copying is to take place (destination field)

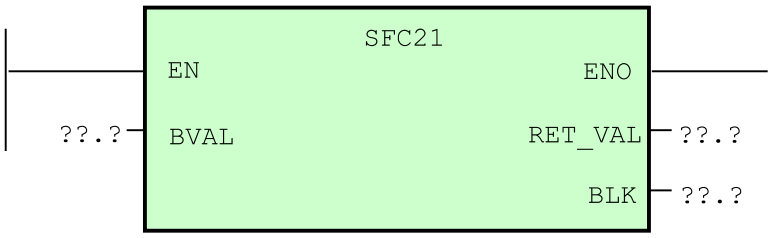
Goal of the Exercise You are to become familiar with the system function SFC 20 (BLKMOV).

Task The recipe values (DBW0-DBW38) are to be copied from data block DB 20 into DB10 (DBW0-DBW38) in the work memory. Copying takes place once after an edge at input I 0.0 .

What to Do

1. Create an OB1 that copies the recipe values from DB 20 into DB 10 with the help of SFC20 (BLKMOV) at an edge at input I 0.0.
2. Transfer the return value #RET_VAL to the digital display of the simulator
3. Download your user program to the CPU and test the program.

Additional Exercise 7.5: Initializing a DB (SFC 21)



Parameter	Declaration	Data Type	Memory Area	Description
BVAL	INPUT	ANY	I, Q, M, D, L	Preset value
RET_VAL	OUTPUT	INT	I, Q, M, D, L	Return value of the SFC
BLK	OUTPUT	ANY	I, Q, M, D, L	Destination area, that is initialized with the contents of BVAL

Goal of the Exercise: Becoming familiar with the use of system functions.

Task

Create an FC 75 with which data blocks can be initialized. The FC 75 has the following functionality:

- The FC 75 expects the following input parameters:
  - #DB_NUM (WORD): Number of the DB to be initialized
  - #INI (BYTE): Byte pattern with which all of the DB's memory cells are to be preset.
- The FC 75 first of all determines if the desired DB exists in the work memory. If it exists, then its length is also determined.

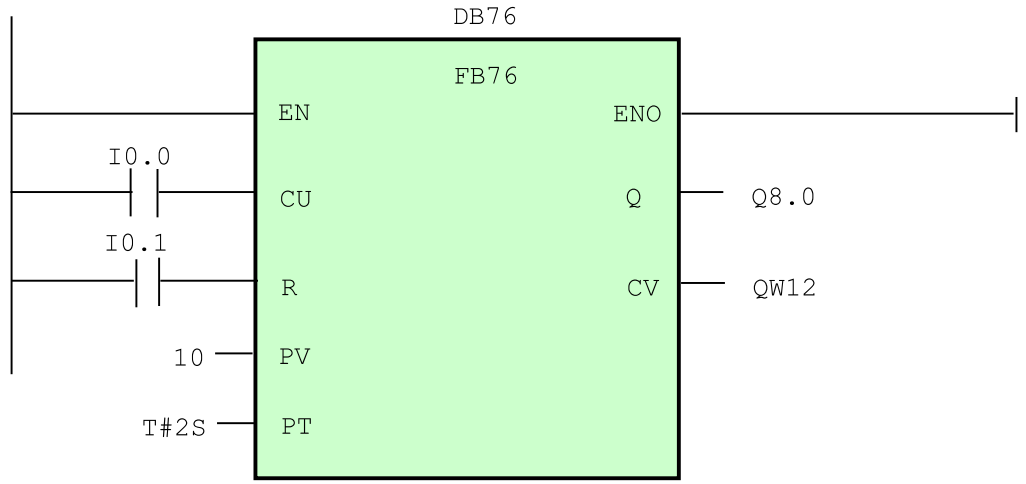
  S7-300: The necessary SFC24 does not exist in S7-300 training units. In this case, only the length of the DB that exists in the work memory is to be determined.

  Subsequently the FC 75 initializes the block with the byte that is passed.
- The FC 75 signals in its #RET_VAL (BOOL)
  - TRUE: DB was successfully initialized.
  - FALSE: DB was not initialized, that is, DB does not exist in the work memory

What to Do

1. Create the FC 75.
2. Integrate the FC 75 in the OB1 so that the DB 10 is initialized with "0" at an edge at input I 1.1.
3. Download your program into the CPU and test your program.

Additional Exercise 7.7: Counter Block with "Contact Debouncing" Function



Task

Create a 16-bit count block (count up counter) FB76 "CU" with the following properties:

- The counter is increased by 1 with a rising edge, when the signal level is subsequently longer than the time PT at input CU.
- Otherwise, the count block has the same characteristics as the IEC conforming counter SFB0 "CTU".
- Output Q indicates whether the current count value is greater than or equal to the preset value PV.

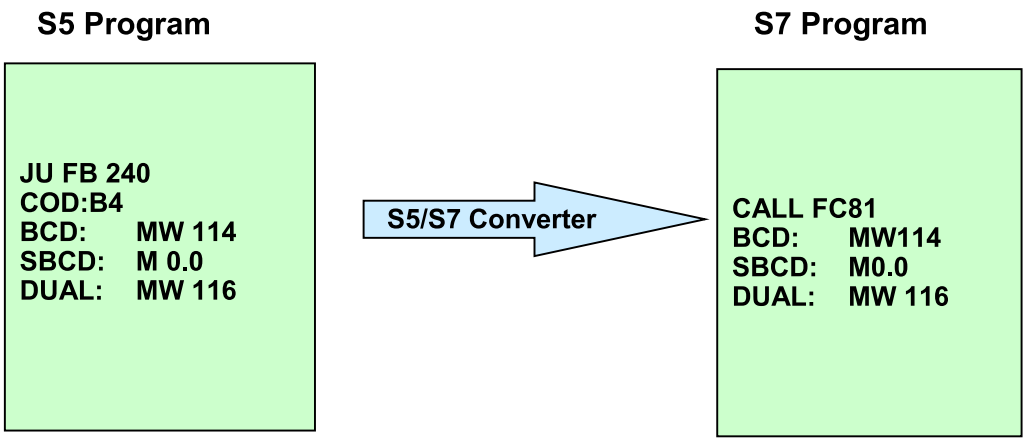
Parameters

Parameter	Declaration	Data Type	Description
CU	INPUT	BOOL	Count input (Count up)
R	INPUT	BOOL	Reset input, dominates vis-à-vis CU.
PV	INPUT	INT	Preset value.
PT to	INPUT	TIME	Time period that the signal level has be in the 1-state after a positive edge, so that the counter is increased by 1.
Q	OUTPUT	BOOL	Status of counter: Q has the value: 1, if CU >PV, 0 otherwise
CV	OUTPUT	INT	Current value

What to Do

1. Create an FB76 with the desired properties. Use the system function blocks SFB0 and SFB4 for implementation.
2. Call the count block FB76 with instance DB76 in OB1. Assign the block parameters with the following actual parameters:
   - CU = I 0.0 - R = I 0.1
   - PV = I W4 - PT = T#1000MS
   - Q = Q8.0 - CV = QW12 (digital display on simulator)
3. Download the blocks to the CPU and test the program.

The Library: S5-S7 Converting Blocks



SIMATIC S7

Date: 30.06.2011
File: PRO2_07E.18

SITRAIN Training for Automation and Drives

Introduction

This library contains S7 standard blocks needed for the conversion of S5 programs. This means, for example, that if an FB 240 standard block was present in the S5 program, the FC 81 block in the library replaces the FB 240 standard block.

Since the converter only transmits the FC 81 block call, you must copy the called block from the library into your S7 program.

Library Contents

The library blocks are divided into the following functions:

- Floating point arithmetic, such as adding and subtracting
- Signal functions, such as first-up signal with a double flashing frequency
- Integrated functions, such as code converter BCD --> Dual
- Basic logic functions, such as LIFO

Manual

The blocks are described in extensive detail in the “Converting from STEP 5 Programs” manual.

Online Help

You can also find an extensive description in the STEP 7 software. To do so call the STEP7 help with:

- Help -> Contents -> Calling Reference Helps (LAD, FBD, STL, Blocks, etc.) -> Jumps to Language Descriptions and Help on Block and System Attributes -> Help on S5/S7 Standard Functions.

Note

So-called scratchpad flags are also used for these blocks, as was typical for SIMATIC S5.

The Library: TI-S7 Converting Blocks (Part 1)

Block	Symbol	Description
FC 80	TONR	Start time as a retentive ON delay
FC 81	IBLKMOV	Transfer data area indirectly
FC 82	RSET	Reset bit memory area or I/O area
FC 83	SET	Set bit memory area or I/O area
FC 84	ATT	Enter value in the table
FC 85	FIFO	Output the first value in the table
FC 86	TBL_FIND	Search for the value in the table
FC 87	LIFO	Output the last value in the table
FC 88	TBL	Execute the table operation
FC 89	TBL_WRD	Copy value from the table
FC 90	WSR	Save the data in the shift register
FC 91	WRD_TBL	Logically combine the value with the table element and save it
FC 92	SHRB	Shift the bit to the shift register
FC 93	SEG	Generate bit pattern for digital display
FC 94	ATH	Convert the ASCII character string to a hexadecimal number
FC 95	HTA	Convert a hexadecimal number to an ASCII character string
FC 96	ENCO	Set the specified bit in the word
FC 97	DECO	Read the bit number of the least significant bit
FC 98	BCDCPL	Generate the ten‘s complement
FC 99	BITSUM	Count the number of set bits

FC 80 The FC80 function starts the "time" as latching ON delay (TONR). The FC80 accumulates the time value until the current time value of the run out time (#ET) is the same as the preset time value (#PV) or it exceeds this.

FC 81 With the transfer data range indirectly function, (IBLKMOV), you can transfer a data range consisting of bytes, words, integers (16 bit), double words, or integers (32 bit) from a source to a destination.

The #S_DATA and #D_DATA "POINTERs" point, for their part, to structures of the "POINTER" data type that determine the start of the source area and the destination area. The area length to be copied is determined via separate parameters.

FC 82/83 Sets the signal state of the bits in a specified area to ”1” (FC 83) or to "0" (FC 82), if the MCR bit is “1.” If the MCR bit is ”0,” the signal state of the bits in the area is not changed.

FC 84-FC92 This deals with table functions to implement FIFO functions, for example. The values are to be input in the word format and the length is adjustable.

FC 93-FC 99 This group makes various conversion functions available.

The Library: TI-S7 Converting Blocks (Part 2)

Block	Symbol	Description
FC 100	RSETI	Reset output area immediately
FC 101	SETI	Set output area immediately
FC 102	DEV	Standard deviation
FC 103	CDT	Correlated data tables
FC 104	TBL_TBL	Table logic operation
FC 105	SCALE	Scale value
FC 106	UNSCALE	Unscale value
FB 80	LEAD_LAG	Lead/Lag algorithm
FB 81	DCAT	Discrete control interrupt
FB 82	MCAT	Motor control interrupt
FB 83	IMC	Index matrix comparison
FB 84	SMC	Matrix scanner
FB 85	DRUM	DRUM (sequence processor)
FB 86	PACK	Collect/distribute table data

FC 100-FC 101 The (RSETI) function resets the signal state of the bits in a specified range of bytes to ”0” or to "1" for FC 101, if the MCR bit is ”1.” If the MCR bit is ”0,” the signal state of the bytes in the range is not changed.

FC 102 The standard deviation (DEV) function calculates the standard deviation from a group of values stored in a table (TBL). The result is stored in OUT. The standard deviation is calculated according to the following formula :

$$\text{Standard deviation} = \sqrt{\frac{(N \times SqSum) - Sum^2}{N \times (N - 1)}}$$

with:

- #Sum = Sum of the values in TBL N = number of values in TBL
- #SqSum = Sum of all values in TBL squared

FC 103 The “correlated data tables” (CDT) function compares an input value (#IN) with an already existing table with input values (#IN_TBL) and looks for the first value that is larger than or equal to the input value

With the help of the index of the located value, the value is then copied to the respective output value (#OUT) in the table of output values (#OUT_TBL).

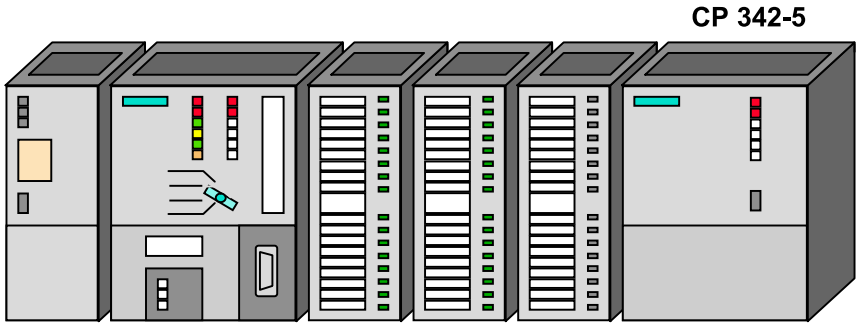
FC 104-FC 105 Is used to scale analog values from an analog input or to an analog output.

FB 80- FB 86 Refer to the electronic manual.

The Library: Communication Blocks

Block	Symbol	Description
~~FC 1~~	~~DP_SEND~~	~~Send data to PROFIBUS-CP~~
FC 2	DP_RECV	Receive data from PROFIBUS-CP
FC 3	DP_DIAG	Load diagnostic data of a station
FC 4	DP_CTRL	Send control task to the CP

Exclusively in the configuration:
S7-300 CPU with CP 342-5



Overview

The library functions FC1, FC2, FC3 and FC4 are used exclusively in the following configuration:

- S7-300 CPU with external PROFIBUS CP 342-5

In all other cases, that is, for S7-300 with integrated PROFIBUS-DP interface and for the entire S7-400 system, the respective functionality is implemented using the standard load and transfer commands (L ... , T...) or using SFC14 (DPRD_DAT), SFC15 (DPWR_DAT), SFC11 (DPSYC_FR) and SFC13 (DPNRM_DG).

FC1

The DP_SEND block passes the data of a specified DP output area to the PROFIBUS-CP for passing on to the distributed I/O.

FC2

The DP_RECV block accepts the process data of the distributed I/O as well as a status information in a specified DP input range.

FC3

The FC block DP_DIAG is used for requesting diagnostic information. Differentiation is made between the following types of tasks:

- request DP station list;
- request DP_diagnostic list;
- request DP single diagnosis;
- read input / output data of a DP slave acyclic;
- read DP operating mode.

FC4

The FC block DP_CTR passes control tasks to the PROFIBUS-CP. Differentiation is made between the following types of tasks:

- Global Control acyclic / cyclic;
- delete older diagnosis;
- set current DP operating mode;
- set DP operating mode for PLC/CP stop;
- read input/output data cyclically;
- set processing mode of DP slave.

The Library: PID Control Blocks

Block	Symbol	Description
FB 41	CONT_C	Continuous PID control block
FB 42	CONT_S	PI controller with binary manipulated value output
FB 43	PULSEGEN	Pulse generator for PID controller with pulse output

FB 41

The SFB "CONT_C" (continuous controller) is used on SIMATIC S7 programmable logic controllers to control technical processes with continuous input and output variables. During parameter assignment, you can activate or deactivate subfunctions of the PID controller to adapt the controller to the process.

You can use the controller as a PID fixed setpoint controller or in multi-loop controls as a cascade, blending or ratio controller. The functions of the controller are based on the PID control algorithm of the sampling controller with an analog output signal, if necessary extended by including a pulse generator stage to generate pulse duration modulated output signals for two or three step controllers with proportional actuators.

FB42

The SFB "CONT_S" (step controller) is used on SIMATIC S7 programmable logic controllers to control technical processes with digital manipulated value output signals for integrating actuators. During parameter assignment, you can activate or deactivate subfunctions of the PI step controller to adapt the controller to the process.

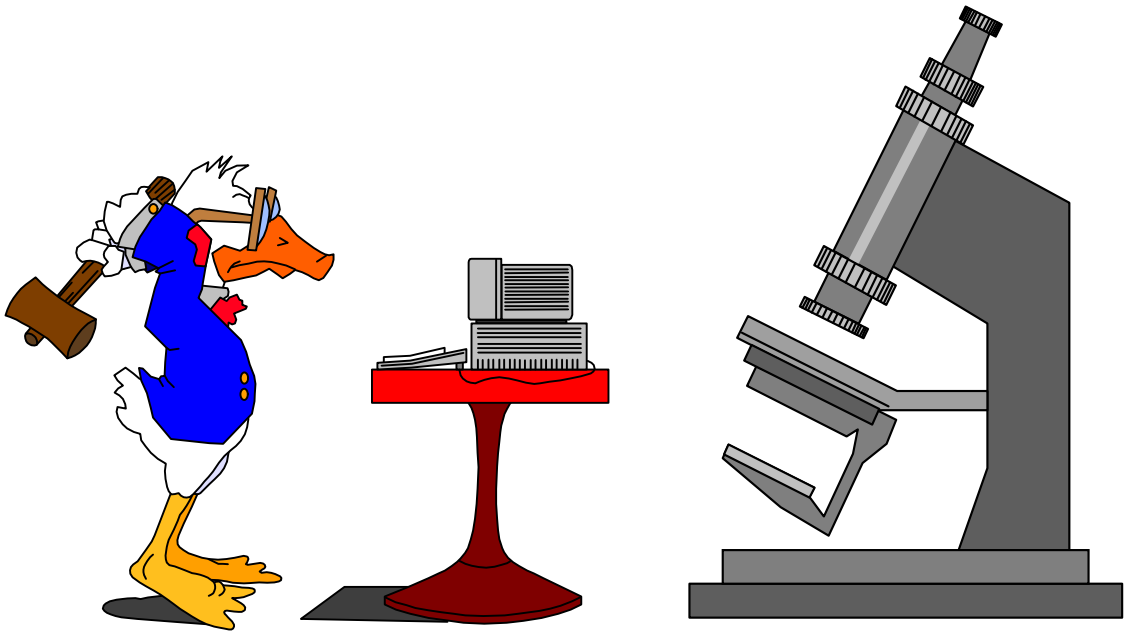
You can use the controller as a PI fixed setpoint controller or in secondary control loops in cascade, blending or ratio controllers, however not as the primary controller. The functions of the controller are based on the PI control algorithm of the sampling controller supplemented by the functions for generating the binary output signal from the analog actuating signal.

FB43

The SFB43 "PULSEGEN" (pulse generator) is used to structure a PID controller with pulse output for proportional actuators.

Using SFB "PULSEGEN", PID two or three step controllers with pulse duration modulation can be configured. The function is normally used in conjunction with the continuous controller "CONT_C".

Handling Synchronous and Asynchronous Errors



SIMATIC S7

Date: 30.06.2011
File: PRO2_08E.1

SITRAIN Training for Automation and Drives

Contents **Page**

Handling Asynchronous Errors

Asynchronous errors are not assigned to a particular program position, i.e. they appear asynchronous to the program processing.

Type of Error	Example	Error OB
Time Error	Max. scan time exceeded	**OB 80**
Power Supply Error	Failure of the backup battery	**OB 81** [2]
Diagnostic Interrupt	Wirebreak at the input of a diagnostics-capable module	**OB 82**
Remove/Insert-module Interrupt	Removing a signal module in S7-400 while in running mode	**OB 83** [1]
CPU Hardware Error	Faulty signal level on the MPI interface	**OB 84** [1]
Program Sequence Error	Error in the updating of the process image (defective module)	**OB 85**
Rack Failure	Power supply failure in the distributed rack	**OB 86** [1]
Communications Error	Incorrect message identifier	**OB 87**

[1] **only with S7-400**

[2] **no Stop w/o Error OB**

Date: 30.06.2011
File: PRO2_08E.2

Introduction The slide lists the asynchronous error events. These errors are not assigned to a certain program position.

Time Error The scan cycle monitoring time has a default setting of 150 ms. The system recognizes a time error if the duration of the cycle is longer than 150 ms. If the error occurs twice in the same cycle, the CPU goes into the Stop state.

Power Supply Error Occurs with the failure of or a missing backup battery and in addition with S7-400 with the failure of the 24 V supply in the central or expansion device. Unlike the other types of errors, the CPU, without an existing error OB, remains in the Run state and a red error LED lights up on the CPU.

Diagnostic Interrupt Diagnostic capable modules like analog modules for example, can trigger a diagnostic interrupt in the case of an error. The modules must be assigned parameters in such a way that the diagnostic interrupt is enabled.

Remove/Insert Interrupt Is triggered by inserting or removing modules in the S7-400 PLC system. In inserting the modules, the operating system checks if the correct module type was inserted. This function makes it possible to remove and insert modules during the program cycle.

CPU-HW Error In the S7-400 errors are recognized on the MPI interface at the K-Bus or at the interface module for distributed I/O.

Program Sequence Error Results from I/O access errors in updating the process image or for example, from a missing OB for a parameterized time-of-day interrupt.

Rack Failure Is recognized when a rack, a subnet in networked PLC systems, or a station of a distributed I/O fails.

Communications Error An incorrect message identifier exists in the receiving of global data in the S7-300, or the data block is too short for storing the status information. In the S7-400 there are further causes, for example, the sending of synchronization messages is not possible.

Handling the Error Organization Blocks

- **In order to suppress the CPU Stop when there is an error, transfer an empty error organization block**
- **You can program the desired response in the Error OB and, if required, request the Stop state with the system function SFC 46 after the execution of the Error OB**
- **An additional error identifier is stored in the start information in the error organization block, that can be evaluated in the program**
- **A description of the error organization blocks can be found in the Online help or in the System and Standard Function manual**
- **The transmission of error OBs into a CPU that does not support these OBs is rejected with an error message**

Start Information

Temporary variables are defined in the declaration part for every organization block,. The operating system stores the start information in these variables. The operating system stores additional information in the start information as to why the block is called.
As an example you can see the start info from OB 81.

Name	Type	Initial Value	Comment
OB81_EV_CLASS	BYTE		16#39, Event class 3, Entering ev
OB81_FLT_ID	BYTE		16#XX, Fault identifcation code
OB81_PRIORITY	BYTE		26/28 (Priority of 1 is lowest)
OB81_OB_NUMBR	BYTE		81 (Organization block 81, OB81)
OB81_RESERVED_1	BYTE		Reserved for system
OB81_RESERVED_2	BYTE		Reserved for system
OB81_MDL_ADDR	INT		Reserved for system
OB81_RESERVED_3	BYTE		Reserved for system
OB81_RESERVED_4	BYTE		Reserved for system
OB81_RESERVED_5	BYTE		Reserved for system
OB81_RESERVED_6	BYTE		Reserved for system
OB81_DATE_TIME	DATE_AND_TIME		Date and time OB81 started

The variable OB81_FLT_ID has the following meaning:

- B#16#21: At least one backup battery of the central rack is empty BATTF)
- B#16#22: Backup voltage in central rack is missing (BAF).
- B#16#23: Failure of the 24V supply in central rack /eliminated.
- B#16#31: At least one backup battery of an expansion rack is empty
- B#16#32: Backup voltage in one of the expansion racks is missing
- B#16#33: Failure of the 24V supply in an expansion rack

Example of an Asynchronous Error OB

```
OB81:  Error OB: Power supply failure

Network 1:  Battery failure, coming event

      L     #OB81_FLT_ID            // Load error identifier
      L     B#16#22                 // Identifier: Battery failure in CR
      ==I
      =     M     81.1              // Set auxiliary memory marker
      L     #OB81_EV_CLASS          // Identifier: incoming, outgoing
      L     B#16#39                 // Identifier: incoming event
      ==I
      =     M     81.2              // Aux. mem. marker incoming event
      A     M     81.1              // Battery failure and
      A     M     81.2              // incoming event
      S     M     81.0              // Set aux. mem. marker for error
                                    // display


Network 2:  Reset auxiliary memory marker, when battery O.K.

      L     #OB81_EV_CLASS          // Identifier: incoming, outgoing
      L     B#16#38                 // Identifier: outgoing
      ==I
      R     M     81.0              // Reset auxiliary memory marker
```


Task

A battery failure should result in an error display on the operating console. After changing the battery the display should automatically go off.

Description

In power supply errors, for example a battery failure, the error organization block is called once by the operating system. After the error is eliminated the OB 81 is called once more.

In the program example, the #OB81_FLT_ID variable is evaluated, in order to determine if there was a battery failure. In this case the variable contains the value 22H. The comparison is thereby fulfilled and the bit memory M 81.1 is triggered.

The error display is to be started when the battery has failed (incoming event) and cleared after the error has been eliminated (outgoing event).

The following identifiers are in the variable #OB81_EV_CLASS:

- B#16#39 incoming event
- B#16#38 outgoing event.

The setting and resetting of the auxiliary memory marker M 81.0 is achieved through the evaluation of these variables.

In the cyclical program, the auxiliary memory marker M81.0 can be linked to a flash clock memory and be assigned an output. The output then flashes as long as the battery is empty or removed.

Handling Synchronous Errors

- **Sychronous errors are assigned directly to a position in the user program**
- **Errors in arithmetic instructions (overflow, invalid REAL number)**

→ **Setting Status Bits**

Errors in the processing of STL instructions (synchronous error)

→ **Call of the synchronous error OB**

Type of Error	Example	Error OB
Programming error	Called block does not exist in the CPU	**OB 121**
Access error	Direct access to a defective or not existing module	**OB 122**

Synchronous Errors

The CPU's operating system generates a synchronous fault when an error occurs in immediate connection with the program processing.

OB121 is called when there is a programming error. OB122 is called when there is an access error. If a synchronous error OB is not loaded into the CPU, the CPU switches into the STOP mode when a synchronous fault occurs.

A synchronous error OB has the same priority as the block in which the error occurred. For that reason, the registers of the interrupted block can be accessed in the synchronous error OB and that is why the program in the synchronous error OB can also return the registers (if necessary with changed contents) to the interrupted block.

Masking Synchronous Errors

S7 has the following SFCs, with which you can mask and demask OB121 start events while your program is being processed:

- SFC36 (MSK_FLT): masks specific error codes
- SFC37 (DMSK_FLT): demasks the error codes that were masked by SFC36
- SFC38 (READ_ERR): reads the Error Register

Start Information for Programming Error OB121

Variable Name	Data Type	Description, Assignment
OB121_EV_CLASS	BYTE	B#16#25=Call programming error OB121
OB121_SW_FLT	BYTE	Error code (see text)
OB121_PRIORITY	BYTE	Priority class in which the error occurred
OB121_OB_NUMBR	BYTE	OB Number (B#16#79)
OB121_BLK_TYPE	BYTE	Type of interrupted block (only S7-400) OB: B#16#88, DB: B#16#8A, FB: B#16#8E, FC: B#16#8C
OB121_RESERVED_1	BYTE	Addition to error code (see text)
OB121_FLT_REG	WORD	OB121: Error source
OB121_BLK_NUM	WORD	Number of blocks in which the error occurred
OB121_PRG_ADDR	WORD	Error address in the error causing block (only S7-400)
OB121_DATE_TIME	DT	Recording point in time of programming error

Error Codes (#OB121_SW_FLT)

B#16#21: BCD conversion error. The variable #OB121_FLT_REG contains an identifier for the register concerned (W#16#0000: ACCU 1).

B#16#22: Range length error during reading
B#16#23: Range length error during writing
B#16#28: Indirect reading access of BYTE, WORD or DWORD with bit address not equal to 0 (Misalignment during reading).
B#16#29: Indirect writing access of BYTE, WORD or DWORD with bit address not equal to 0 (Misalignment during writing).

In this case, #OB121_FLT_REG contains the faulty byte address and #OB121_RESERVED_1 contains the type of access and the memory area:

Bit 7 to 4 (access type):	Bit 3 to 0 (memory area)	
0: Bit access	0: I/O area	4: Global DB
1: Byte access	1: PII	5: Instance DB
2: Word access	2: PIQ	6: Own local data
3: Double word access	3: Bit memory	7: Local data of caller

B#16#24: Range error during reading
B#16#25: Range error during writing

#OB121_FLT_REG contains the identifier B#16#86: own local data area.

B#16#26: Error with Timer number (invalid number in #OB121_FLT_REG)
B#16#27: Error with Counter number (invalid number in #OB121_FLT_REG)

B#16#30: Write access of write-protected global DB (No. in #OB121_FLT_REG)
B#16#31: Write access of write-prot. instance DB (No. in #OB121_FLT_REG)
B#16#32: Number error in access of global DB (No. in #OB121_FLT_REG)
B#16#33: Number error in access of instance DB (No. in #OB121_FLT_REG)

B#16#34: Number error in FC call (No. in #OB121_FLT_REG)
B#16#35: Number error in FB call (No. in #OB121_FLT_REG)
B#16#3A: Access of not loaded DB (No. in #OB121_FLT_REG)
B#16#3C: Access of not loaded FC (No. in #OB121_FLT_REG)
B#16#3D: Access of not loaded SFC (No. in #OB121_FLT_REG)
B#16#3E: Access of not loaded FB (No. in #OB121_FLT_REG)
B#16#3F: Access of not loaded SFB (No. in #OB121_FLT_REG)

Start Information for the Access Error OB122

Variable Name	Data Type	Description, Assignment
OB122_EV_CLASS	BYTE	B#16#29=Call access error
OB122_SW_FLT	BYTE	Error code (Possible values: B#16#42, B#16#43, B#16#44, B#16#45)
OB122_PRIORITY	BYTE	Priority class in which the error occurred
OB122_OB_NUMBR	BYTE	OB Number (B#16#80)
OB122_BLK_TYPE	BYTE	Type of interrupted block (only S7-400) OB: B#16#88, DB: B#16#8A, FB: B#16#8E, FC: B#16#8C
OB122_MEM_AREA	BYTE	Addition to error code (see text)
OB122_FLT_REG	WORD	OB122: Address identifier where the error occurred.
OB122_BLK_NUM	WORD	Number of block in which the error occurred
OB122_PRG_ADDR	WORD	Error address in the error causing block (only S7-400)
OB122_DATE_TIME	DT	Recording point in time of programming error

Error Codes The variable #OB122_SW_FLT has the following meaning:

B#16#42 S7-300: I/O access error, reading
S7-400: First reading access after an error occurs

B#16#43: S7-300: I/O access error, writing
S7-400: First writing access after an error occurs

B#16#44: Only for S7-400: Error at the n-th (n>1) reading access after an error occurs.

B#16#45: Only for S7-400: Error at the n-th (n>1) writing access after an error occurs.

OB122_MEM_AREA The variable #OB122_MEM_AREA contains information about the type of access and the memory area:

Bit 7 to 4 type of access:

0: Bit access
1: Byte access
2: Word access
3: Double word access

Bit 3 to 0 memory area:

0: I/O area
1: Process-image input table
2: Process-image output table

Masking Synchronous Errors

Drawbacks of the Synchronous Error OB:

- **Code for the process management and for the error handling is distributed among at least two blocks**
- **Problems with subsequent changes or with maintenance**

Better:

- **Code for process management and for error handling is in the same block**

Masking of Synchronous Faults:

- **Before the "critical" instructions:**
  SFC 36 MSK_FLT: mask synchronous faults
  (OB12x-inhibit call)
- **Execute "critical" instructions**
- **Evaluate if an error occurred**
  SFC 38 READ_ERR: read Error Register
- **OB12x-enable call once again:**
  SFC 37 DMSK_FLT: demask synchronous faults

Drawbacks of Synchronous Error OBs

The handling of synchronous error events by means of synchronous error OBs has numerous drawbacks:

- For a qualified error handling, a corresponding error evaluation in the synchronous error OB must be performed for every block with instructions that can trigger a synchronous error.

  Within the synchronous error OB considerable work must therefore be carried out in order to localize the error in the user program and then to react accordingly.
- Every change in an existing block entails corresponding changes in the synchronous error OB.
- Blocks cannot be integrated in a user program without corresponding consideration in the synchronous error OB.

Alternatives to Synchronous Error OBs

S7 offers, with the help of the function "Masking Synchronous Errors", a mechanism that allows the code for the process management and for the associated error handling to be installed in the same block.

This takes place, for example, in the following steps:

1. Before the execution of "critical" instructions (for example, opening a DB, or access to a DB of unknown length), the corresponding synchronous errors can be masked by means of SFC 36 (MSK_FLT).

   If an instruction then fails, no synchronous error OB is called.
2. After execution of the "critical" instructions, you can check by means of SFC 38 (READ_ERR), whether or not errors occurred within the critical section and react accordingly.
3. Upon conclusion of the activities, the previously masked synchronous faults can then be demasked and thus the call of the synchronous error OB is enabled once again.

SFC 36 for Masking Synchronous Faults

Parameter	Declaration	Data Type	Memory Area	Description
PRGFLT_SET_MASK	INPUT	DWORD	I, Q, M, D, L, Const.	new (additional) programming fault filter
ACCFLT_SET_MASK	INPUT	BYTE	I, Q, M, D, L, Const.	new additional access fault filter
RET_VAL	OUTPUT	INT	I, Q, M, D, L	Return value of SFC, W#16#0001: the new filter overlaps with the existing filter
PRGFLT_MASKED	OUTPUT	DWORD	I, Q, M, D, L	complete programming fault filter
ACCFLT_MASKED	OUTPUT	DWORD	I, Q, M, D, L	complete access fault filter

Masking Synchronous Errors

With the SFC 36 (MSK_FLT), you inhibit the call of the synchronous error OB using fault filters. With signal state "1" you identify in the fault filters for which synchronous errors the OBs are not to be called (the synchronous faults are “masked”).

The specified masking is undertaken in addition to the masking stored in the operating system (OR logic operation of the filter bits). SFC36 signals in the return value if, for the masking specified at the input parameters, a masking already existed (W#16#0001) for at least one bit.

The SFC36 delivers in its output parameters all currently masked events with signal state "1".

CPU Reaction

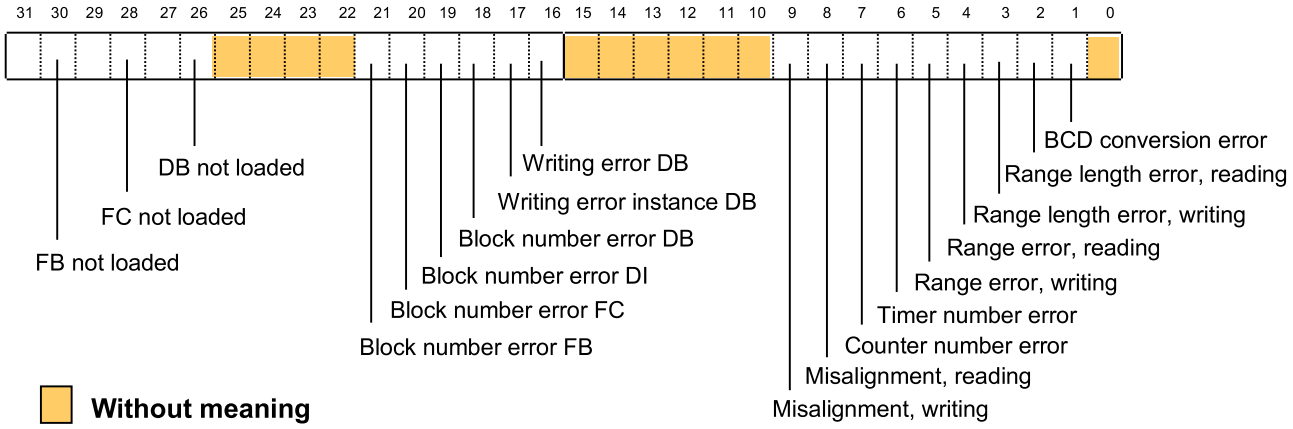
When a programming or access error is masked, the CPU reacts as follows to errors of this type:

1. The error OB is not called for programming or access errors.
2. The error event is entered in the Error Register. The Error Register can be read with the help of SFC38 (READ_ERR).
3. The operating system enters the synchronous fault in the diagnostic buffer independent of the masking.

Masking Scope

The masking is only valid for the priority class in which the SFC 36 was called. If you, for example, inhibit the call of a synchronous error OB in the main program, the synchronous error OB will still be called if the error occurs in an interrupt program.

Structure of the Programming Fault Filter



Note: The corresponding bits of the output parameter PRGFLT_MASKED are set as follows:

Value = 1: Error is masked.

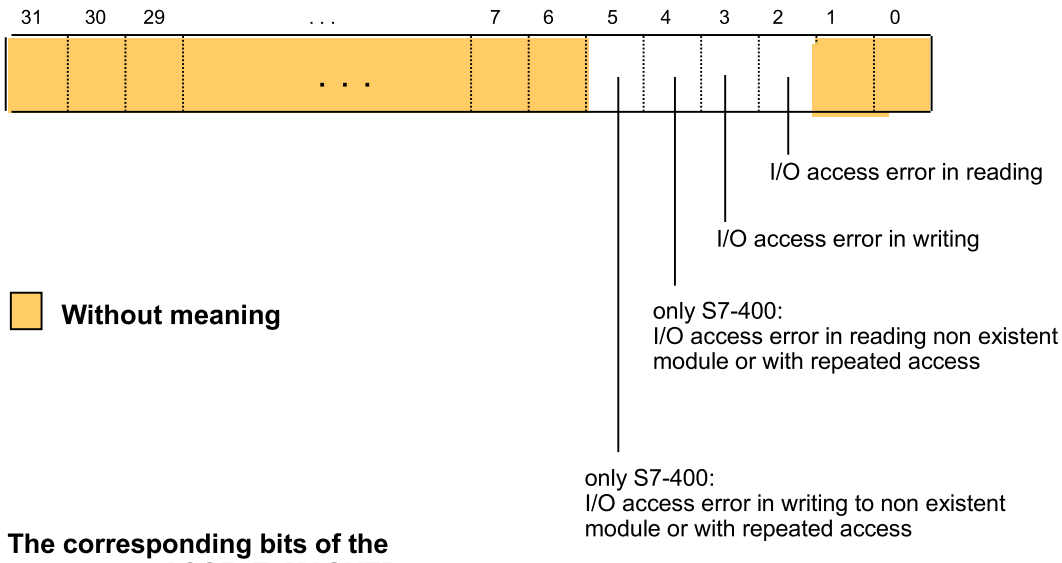
Value = 0: Error is not masked.

Programming Fault Filter

You control the system function for the synchronous error handling with the fault filters. In the programming fault filter there is a bit for every possible programming fault. In specifying the fault filters, you set the synchronous error bits which you want to mask, demask or check.

The fault filters sent by the system functions indicate with signal state “1” the synchronous errors that are still masked or that occurred.

Structure of the Access Fault Filter



Note: The corresponding bits of the output parameter ACCFLT_MASKED are set as follows:

Value = 1: Error is masked.

Value = 0: Error is not masked.

Not relevant bits have the value "1".

SIMATIC S7

Date: 30.06.2011
File: PRO2_08E.11

SITRAIN Training for Automation and Drives

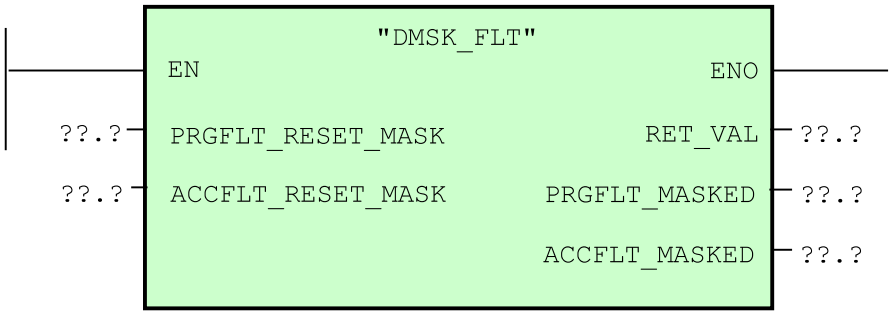
Access Fault Filter

The S7-400 CPUs distinguish between two types of I/O access error. Access to a non existent module and faulty access of a module entered as existing.

If a module fails during operation, a time-out (QVZ) occurs after a short time when the module is accessed. At the same time, this module is entered as “non existent”, so that with every additional access an I/O access error (PZF) is then signaled.

The CPU also signals an I/O access error when a non existent module is accessed, be it directly via the I/O area or indirectly via the process image.

SFC 37 for Demasking Synchronous Faults



Parameter	Declaration	Data Type	Memory Area	Description
PRGFLT_RESET_MASK	INPUT	DWORD	I, Q, M, D, L, Const.	Programming fault filter for resetting
ACCFLT_RESET_MASK	INPUT	BYTE	I, Q, M, D, L, Const.	Access fault filter for resetting
RET_VAL	OUTPUT	INT	I, Q, M, D, L	Return value of SFC, W#16#0001: the new filter contains bits that are not set in the stored filter
PRGFLT_MASKED	OUTPUT	DWORD	I, Q, M, D, L	still masked programming errors
ACCFLT_MASKED	OUTPUT	DWORD	I, Q, M, D, L	still masked access errors

Demasking Synchronous Faults

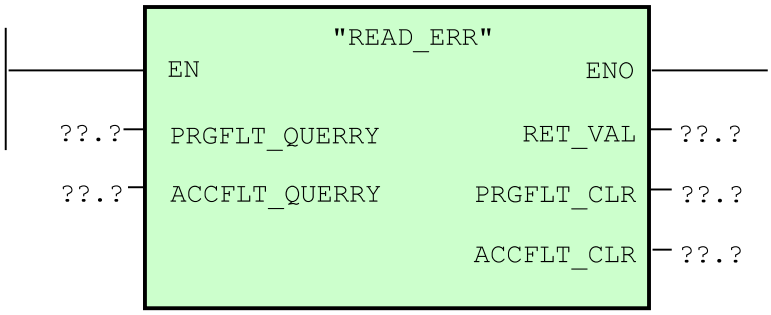
The SFC37 (DMSK_FLT) system function uses the fault filters to enable the call of the synchronous error OBs once again. With signal state "1" you identify in the fault filters for which synchronous errors the OBs are once again to be called (the synchronous faults are “demasked”).
The corresponding entries of the specified demasking, that are in the Error Register, are deleted.

In the return value, the SFC37 signals with W#16#0001 if for the specified demasking at the input parameters, no (stored) masking existed for at least one bit.

The SFC37 delivers in its output parameters all currently masked events with signal state "1".

If a demasked synchronous fault occurs, the corresponding OB is called once again and the event is entered in the Error Register. Enabling is valid for the current priority class.

SFC 38 for Reading Out the Error Register



Parameter	Declaration	Data Type	Memory Area	Description
PRGFLT_QUERRY	INPUT	DWORD	I, Q, M, D, L, Const.	Programming fault filter for checking
ACCFLT_QUERRY	INPUT	BYTE	I, Q, M, D, L, Const.	Access fault filter for checking
RET_VAL	OUTPUT	INT	I, Q, M, D, L	Return value of SFC, W#16#0001: the check filter contains bits that are not set (in the stored filter)
PRGFLT_CLR	OUTPUT	DWORD	I, Q, M, D, L	Programming fault filter with error messages
ACCFLT_CLR	OUTPUT	DWORD	I, Q, M, D, L	Access fault filter with error messages

Reading the Error Register

The SFC38 (READ_ERR) system function reads out the Error Register. With signal state "1" you identify in the fault filters for which synchronous errors you want to read the entries.

In the return value, the SFC38 signals with W#16#0001 if for the specified selection at the input parameters, no (stored) masking exists for at least one bit.

The SFC38 returns the selected events with signal state "1" to the output parameters, when they occurred and deletes these events in the Error Register with the scan. A set bit means that the associated masked synchronous error occurred at least once.

The synchronous faults that occurred in the current priority class are signaled.

Example: Data Block Testing

```
Network 1: Masking, Testing, Demasking
// Mask "DB does not exist"
        CALL   SFC    36(
          PRGFLT_SET_MASK      := DW#16#4000000,  // Identifier: DB does not exist
          ACCFLT_SET_MASK      := DW#16#0,        // no masking for access errors
          RET_VAL              := #SFC36Error,
          PRGFLT_MASKED        := #Prog36Mask,
          ACCFLT_MASKED        := #Acc36Mask);

// Test call
        OPN DB[DB_NO];

// Check programming error
        CALL   SFC    38(
          PRGFLT_QUERRY        := DW#16#4000000,  // Identifier: DB does not exist
          ACCFLT_QUERRY        := DW#16#0,        // no masking for access errors
          RET_VAL              := #SFC38Error,
          PRGFLT_MASKED        := #Prog38Mask,
          ACCFLT_MASKED        := #Acc38Mask);

// Evaluate result
        L      #Prog38Mask
        L      DW#16#4000000
        ==D
        =      #DB_NOT_THERE                    // Set auxiliary variable DB not there

// Demask "DB does not exist"
CALL    SFC    37(
          PRGFLT_RESET_MASK    := DW#16#4000000,  // Identifier: DB does not exist
          ACCFLT_RESET_MASK    := DW#16#0,        // no masking for access errors
          RET_VAL              := #SFC37Error,
          PRGFLT_MASKED        := #Prog37Mask,
          ACCFLT_MASKED        := #Acc37Mask);
```


Example

This example shows the procedure for masking a possible synchronous fault in the opening of a DB.

1. In the 1st. step, the "critical" instruction OPN DB... is masked with the help of SFC 36 (MSK_FLT).
2. After that, the instruction OPN DB[DB_NO] is carried out. If the DB is not in the work memory of the CPU, then OB121 is not called in this case.
3. With the help of SFC38 (READ_ERR), the Error Register is read out and checked whether the instruction for opening the DB failed or not.

   In case of an error, the local variable #DB_NOT_THERE is set to "1" so that an evaluation can be made later on.
4. In the end, the masked synchronous fault is demasked once again with the help of SFC37 (DMSK_FLT), thus re-establishing the the original state.

Exercise 8.1: Error Handling in FC81

Decl.	Name	Type
in	Meas_values	ANY
out	RET_VAL	INT
out	Sum	REAL
out	Mean_value	REAL

Cause	Error Code
Everything O. K.	0
Data type<>REAL	-1
DB does not exist	-2
Range length error	-4

Error Code

Overview

In Exercise 4.3 you created an FC43 that determines the sum and the mean value from an ARRAY of REAL numbers. Up until now, only an elementary error handling (checking the data type) has been performed within this FC.

Error handling is now to be expanded in such a way that the new FC81 is ”crash-safe”. That is, no synchronous fault is to be triggered even with incorrect parameter assignment.

Moreover, the FC81 gives, in the additional output parameter #RET_VAL, information about the type of error.

Goal

First of all copy FC43 into FC81 and integrate the following error handling:

- If a data type unequal to REAL is passed, then FC81 is exited with Error Code -1.
- If an invalid DB number is passed (e.g. number outside of the allowed range or DB does not exist), then FC81 is exited with Error Code -2.
- If within a loop there is an access to a not existing address (range or range length error), then FC81 is exited with Error Code -4.
- In all error cases, FC81 sets the BR-Bit to zero and returns an invalid REAL number in the output parameters #Sum and #Mean_Value

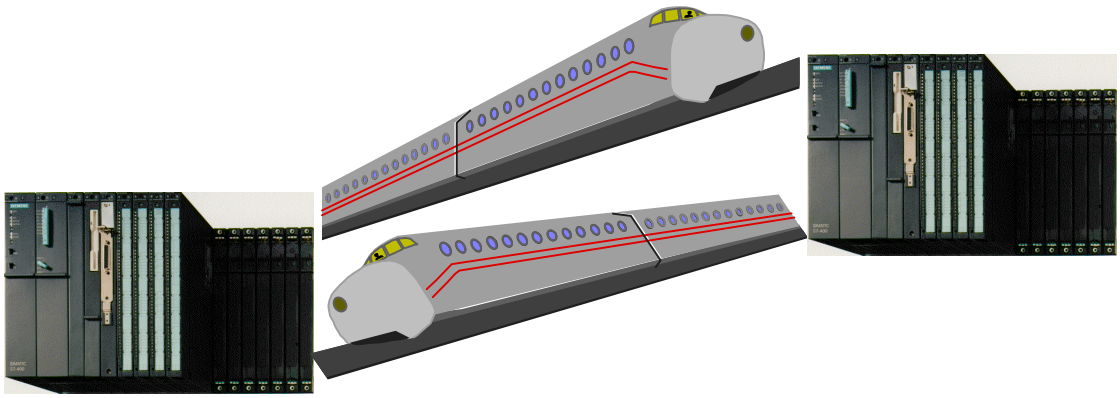
What to Do

1. Supplement FC81 with the output parameter #RET_VAL (Error Code).
2. In FC81 perform the corresponding error handling.
3. Program the FC81 call in OB1.
4. Download the participating blocks into the CPU and test the result.

Question?

How can you make the FC81 "crash"?

Basic and Expanded S7 Communication



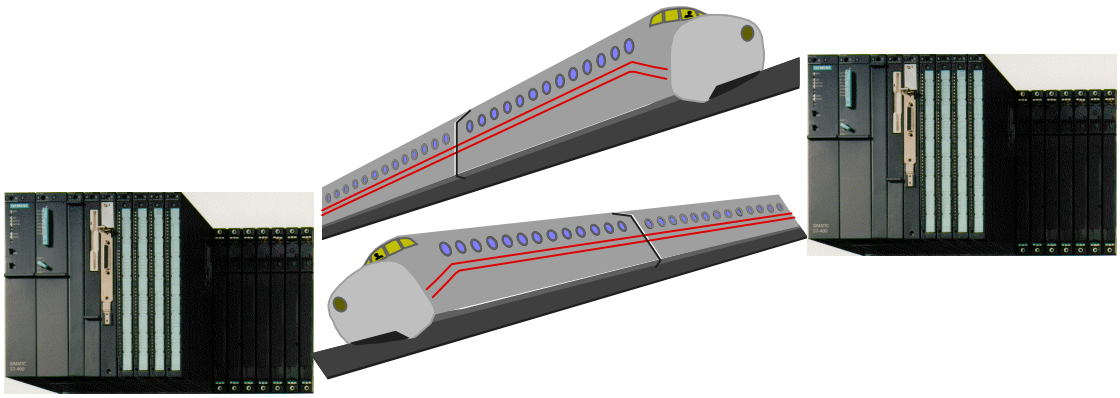
SIMATIC S7

Date: 30.06.2011
File: PRO2_09E.1

SITRAIN Training for Automation and Drives

Contents **Page**

Basic and Expanded S7 Communication

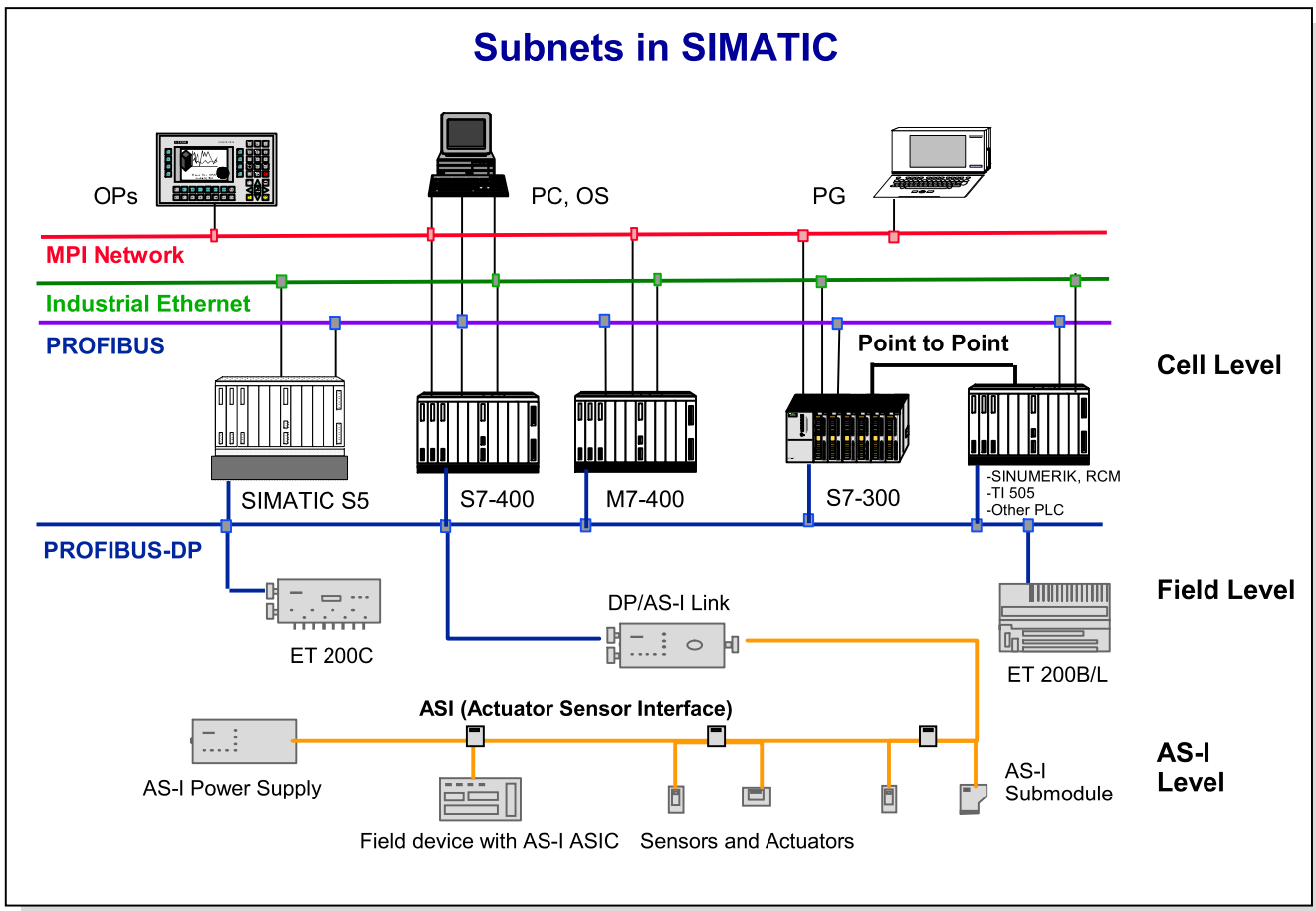


SIMATIC S7

Date: 30.06.2011
File: PRO2_09E.2

SITRAIN Training for Automation and Drives

Contents **Page**



Overview

SIEMENS offers the following subnets, depending on the different requirements for the communication tasks in the cell area (non-time critical) or in the field level (time critical).

MPI

The MPI subnet is designed for tasks in the cell area. The MPI is the multi-point capable interface in SIMATIC S7.

It is designed as the PG interface, that is, for the connection of PGs (commissioning and test) and OPs (operator interface). Beyond that, the MPI subnet can also be used to network a few CPUs.

Industrial Ethernet

Industrial Ethernet is the network for the management level and the cell level in the open, manufacturer independent communication system of SIMATIC.

Industrial Ethernet is designed for the non-time critical transmission of larger amounts of data and offers the possibility of connecting to location crossing networks via Gateways.

PROFIBUS

PROFIBUS is the network for the cell and field area in the open, manufacturer independent communication system of SIMATIC. There are two versions each with their own characteristics:

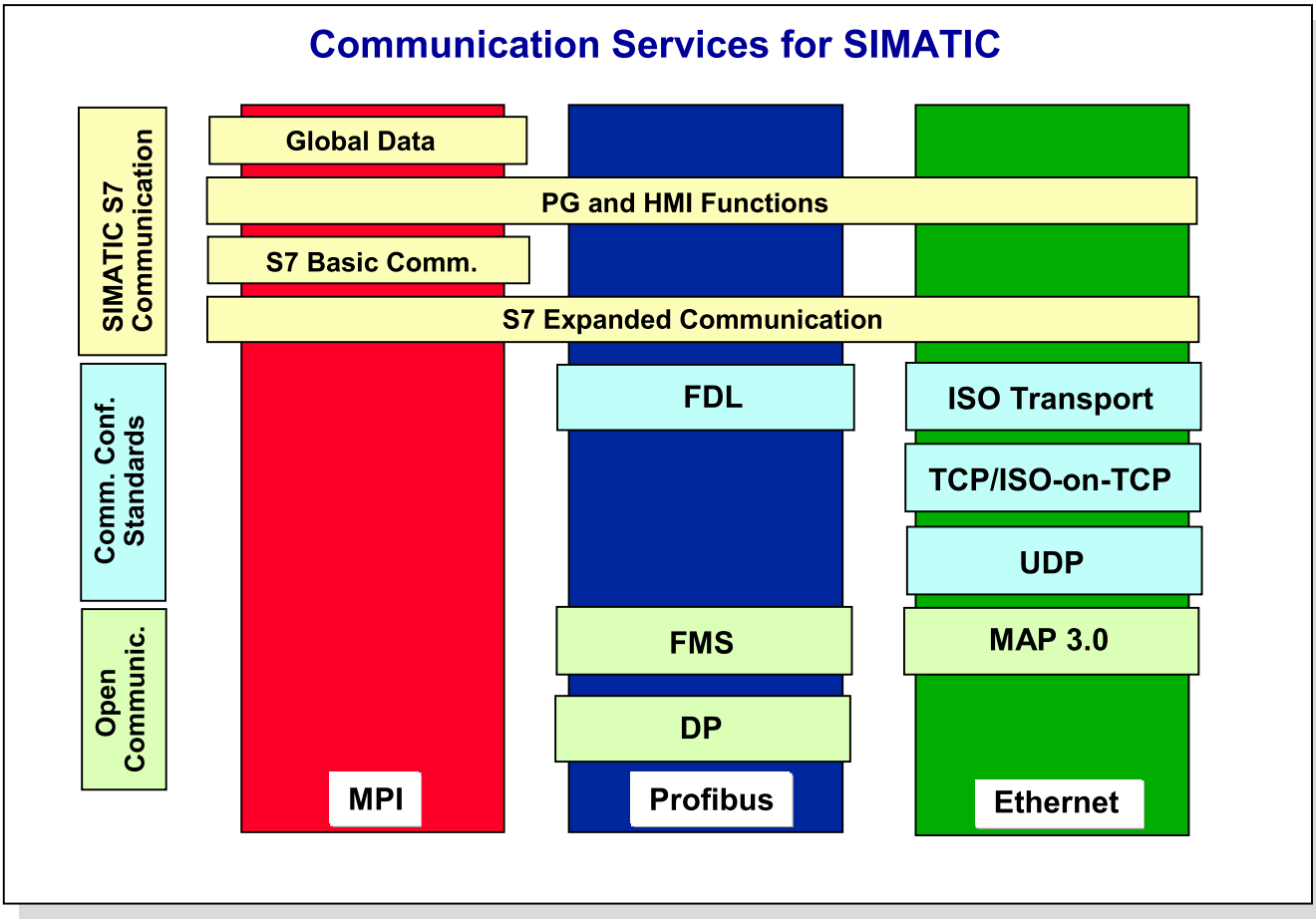
- in the cell area as PROFIBUS for the non-time critical communication between equal, intelligent nodes.
- as field bus PROFIBUS DP for time critical, cyclical data exchange between intelligent masters and field devices.

PtP-Connection

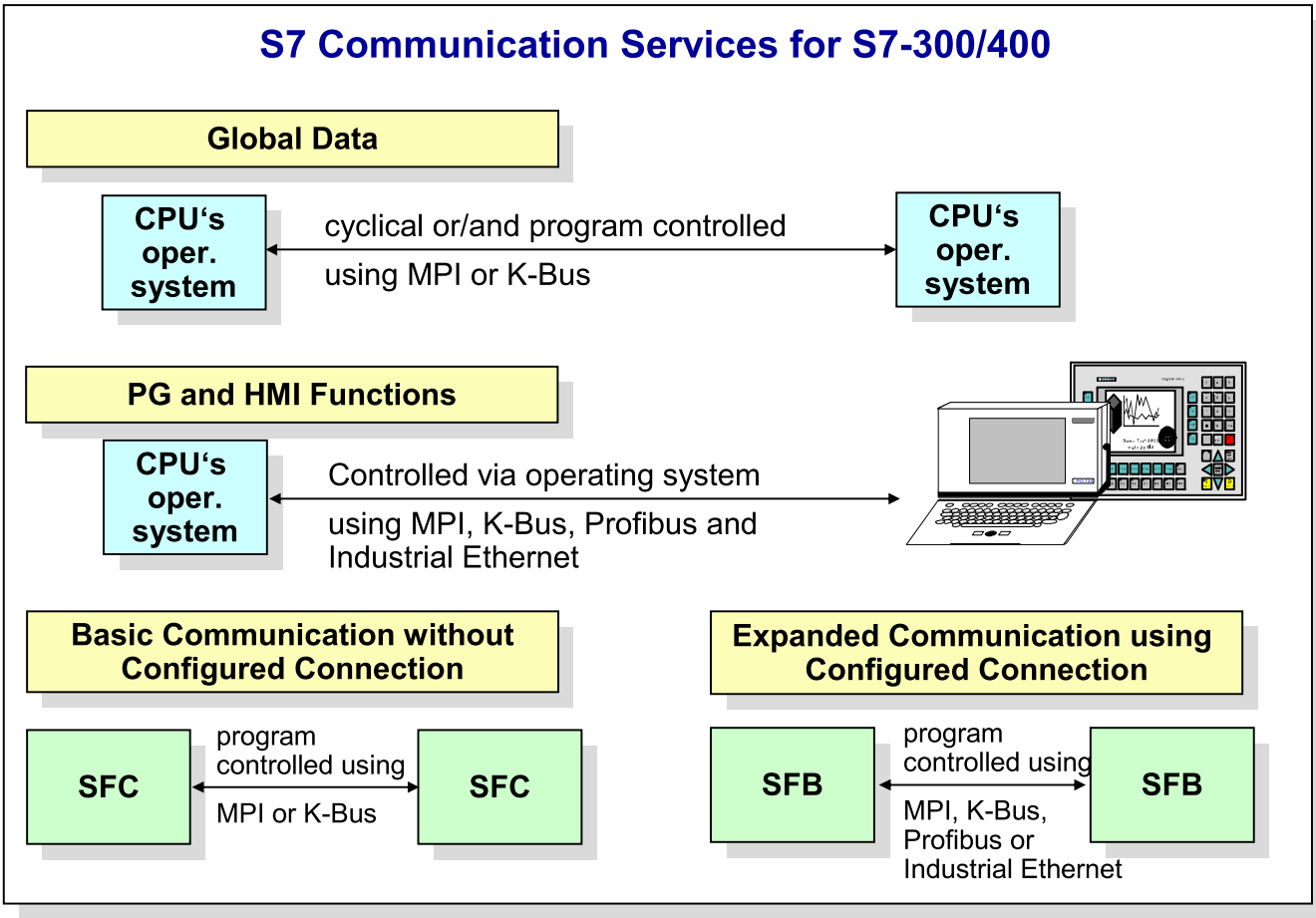
Point-to-Point connections are mainly used for the non-time critical data exchange between two stations or for the connection of devices, such as OPs, printers, bar code readers, magnetic card readers etc. to a station.

AS-Interface

The Actuator-Sensor-Interface is a subnet for the lowest process level in PLC systems. With its help, binary sensors and actuators can be networked.



Services	A communications service describes communications functions with defined performance characteristics, such as exchanging data, controlling devices, monitoring devices, and loading programs.
Global Data	GD (Global Data in the network) for cyclical data exchange of small amounts of data (in S7-400 additionally event-driven).
S7 Communication	These communication utilities are optimized for the communication of S7 PLCs, PGs/PCs and OP/TDs in the SIMATIC S7 connection. • PG functions; a PG can be connected without configured connection. • HMI functions; an OP can be connected without a configured connection. • Basic communication is implemented with SFCs that are contained in the CPU's operating system. (SFC communication runs without configured connection). • Expanded communication takes place via configured connections with the help of SFBs (S7-400 Client/Server; S7- 300 only Server).
FDL (SDA)	For the safe data transfer of average amounts of data between SIMATIC S7 and S5. Corresponds to Layer 2 *Fieldbus Data Link (FDL)* for Profibus.
ISO-Transport TCP, ISO-on-TCP UDP	Is used for the safe data transfer of average to larger amounts of data from SIMATIC S7 to PCs or non-Siemens systems using Industrial Ethernet networks. The FDL, ISO, TCP , ISO-on-TCP and UDP utilities are made available on the CPU using the functions AG-SEND/RECV or AG-LSEND/LRECV.
FMS	*Fieldbus Message Specification (FMS)* makes the object-oriented communication between intelligent partners as well as field devices possible. Utilities supported by *FMS* (variables, domain services, etc.) are specified in EN 50170 Vol. 2.
MAP	Originally developed by the American automobile company General Motors, this protocol is for object-oriented communication between PLC systems (MAP= *Manufacturer Automation Protocol*).
DP	The DP (Distributed I/O) protocol is specially optimized for the time-critical, data-oriented communication from intelligent control units (DP Masters) to field devices (EN 50170 Vol. 3).



SIMATIC S7

Date: 30.06.2011
File: PRO2_09E.5

SITRAIN Training for Automation and Drives

Global Data

This communication makes it possible for data to be exchanged cyclically between CPUs using the MPI interface and without a program. The data exchange takes place at the cycle control point, together with the updating of the process image.

PG and HMI Function

System services such as PG and HMI functions are based in the final analysis on the expanded S7-Communication. Prerequisite for the connection of a PG or an HMI device to an S7-300/400 system is the availability of a free connection on the connection partner (S7-CPU, M7-CPU, M7-FM, etc.).

Basic Communication

With these communication services, data for all S7-300/400 CPUs can be transferred by means of the MPI subnet or within a station via the K bus. System functions (SFCs), such as X_SEND on the send side, and X_RCV on the receiver side, are called in the user program.
The amount of user data that can be transferred in one call is a maximum of 76 bytes.

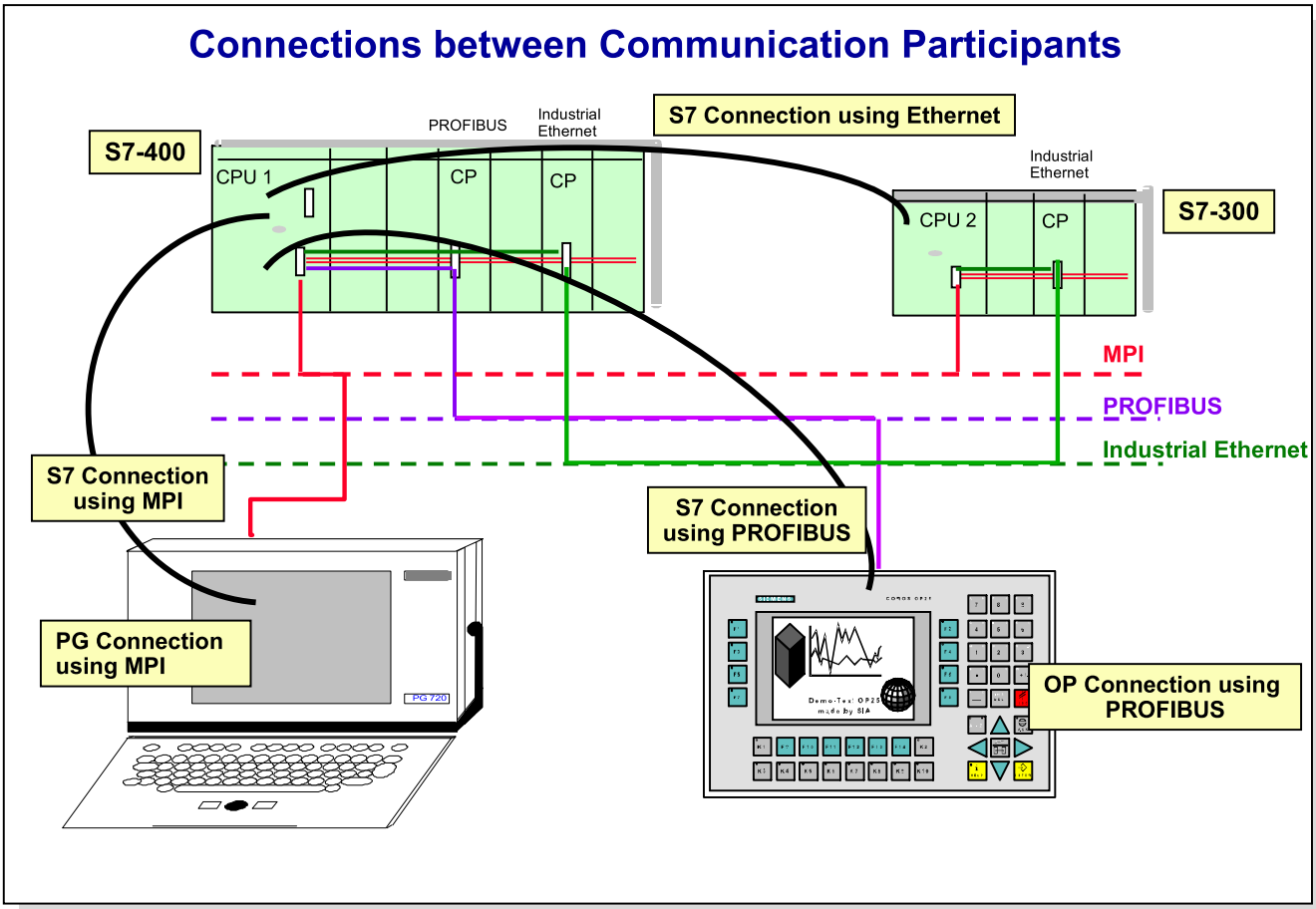
A connection to the communications partner is actively configured when the system functions are called and disconnected after the transmission. A configured connection is not necessary for this.

Expanded Communication

You can use these communication services for all S7-400 CPUs. Data up to a maximum of 64KBytes can be transferred by means of various subnets (MPI, K-Bus, Profibus, and Industrial Ethernet).

System function blocks (SFBs) are used as the programming interface. These SFBs are only integrated in the operating system of the S7-400 CPUs. They do not exist in S7-300.

Besides the functions for data transmission, this communications service also contains control functions such as START and STOP of the partner PLC. Communication takes place via configured connections (connection table). These connections are configured during the station's power up and permanently continue to exist.



Connections

A connection is a logical assignment of two communication partners for carrying out communication services. The connection is linked directly with a communication service.

Each connection has an end position on each of the CPUs in question which contains the necessary information for addressing the communication partner as well as additional attributes for the connection configuration.

Connections can occupy one or several connection resources on the participating communication capable modules (CPUs, CPs, FMs) per end position.

In order to guarantee an orderly connection configuration, connections must be active at one end position and passive at the other end position. Otherwise, the connection cannot be established.

Application

Depending on the chosen communication functions, either configured (expanded communication) or non-configured (basic communication) connections are used.

Configured Connections

This type of connection is configured with STEP 7. The connection end position is assigned a local ID that, among other things, identifies its own address information and that of the communication partner.

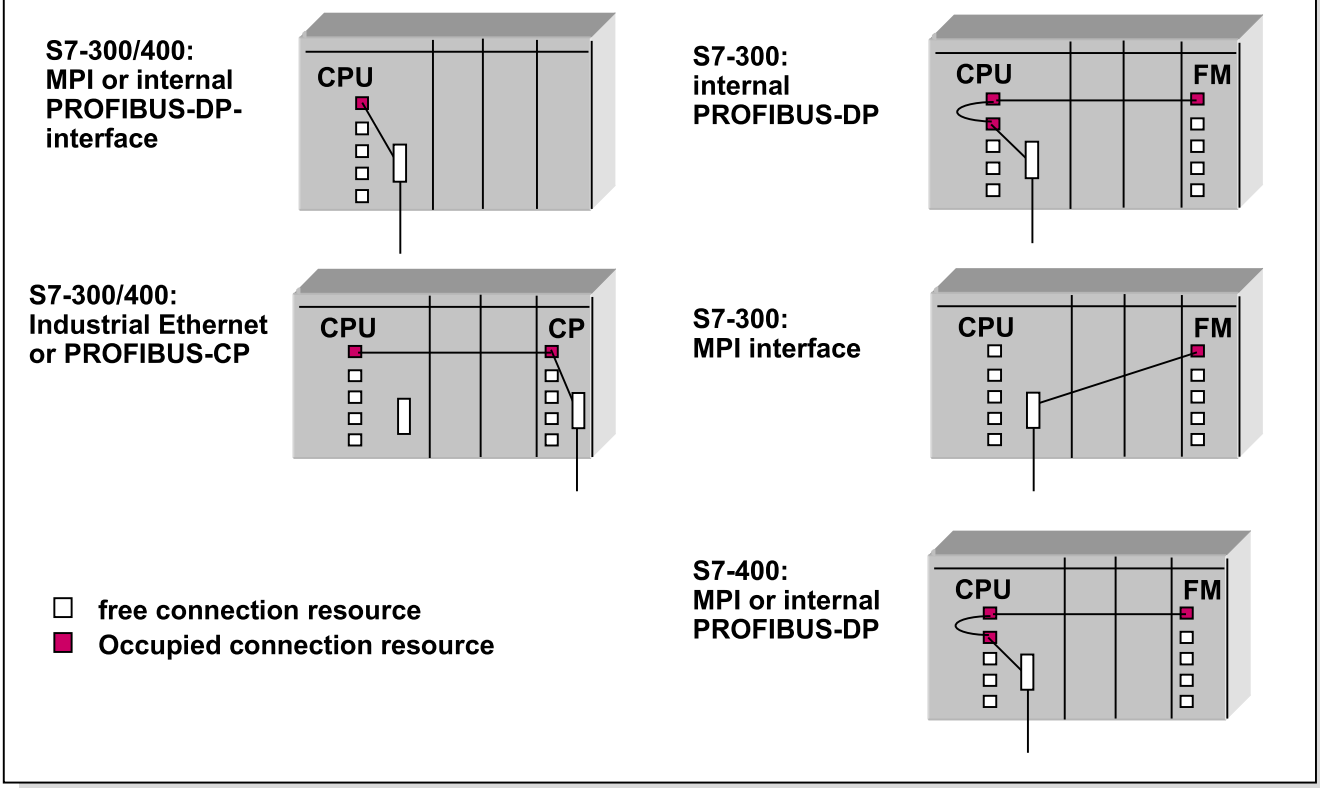
Communication functions that are initiated by a SIMATIC-OP or PC also require configured connections.

Configured connections are configured by the active nodes during power up and remain configured during the entire operating time.

Non-configured Connections

These connections are configured when the communication function is called and are disconnected after the data transmission is completed, if required.

Assignment of Connection Resources for S7 Communication



Overview

At the participating stations, connection resources for the end position or for the transition position (e.g. CP) are required for every connection. The number of connection resources depends on the CPU/CP.

If all connection resources of a communication partner are occupied, no new connection can be established.

S7Functions to CPUs

For the S7 functions via the integrated MPI-/PROFIBUS-DP interface, one connection resource is occupied on the CPU for the end position per S7 connection.

For the S7 functions via an external CP interface, one connection resource each is occupied on the CPU (for the end position) and on the CP (transition position) per S7 connection.

S7Functions to FMs

For the S7 functions to a function module (FM) via the internal MPI-/PROFIBUS-DP interface, two connection resources (for two transition positions) are occupied on the S7-400-CPU per S7 connection and on the FM one connection resource each (for the end position) is occupied.

This is also valid for every additional CPU (multi processor operation) within the same station, whereby the additional CPUs are connected indirectly via K-Bus with an MPI subnet.

PG/OPs

Every PG or OP/TD connection requires a connection resource on the SIMATIC S7/M7-CPU. By default, a connection resource for the connection each of a PG and an OP/TS is reserved for this in every S7/M7-CPU.

An available connection resource is required for every additional PG/OP connection. If several PG/OPs are connected, the number of available connection resources for S7 functions is reduced.

Characteristic Data of S7-CPU Communication

S7- 300	Connection Resources	Reserved by default for connections to PG	OP	Basic comm.
CPU 312 IFM	6	1	1	2
CPU 313	8	1	1	4
CPU 314 /314 IFM	12	1	1	8
CPU 315/315 -2 DP	12	1	1	8
CPU 316	12	1	1	8
CPU 318 -2 DP	30	1	1	4

S7- 400	Connection Resources	Preassigned reservaton for connections to PG	OP	Basic comm.
CPU 412-1/412-2	16	1	1	-
CPU 413-1/413-2	16	1	1	-
CPU 414-1/ -2 / -3	32	1	1	-
CPU 416-1/ -2 / -3	64	1	1	-
CPU 417-4	64	1	1	-

Connections

Every communication connection requires a connection resource on the S7-CPU as a management element for the duration of the communication connection. When communication services log on, the connection resources are occupied according to the sequence of the log on.

S7-300

So that the assignment of the connection resources does not just depend on the sequence of the log on of various communication services, connection resources can be reserved in the S7-300 for the following communication services:

- PG and OP communication
- S7 basic communication

The necessary setting is made in the CPU‘s hardware parameter assignment in the tab: Communication.

At least one connection resource each is preassigned (reserved) for the PG/OP communication. A lesser value is not possible.

Other communication services such as S7 communication with PUT/GET functions cannot occupy this connection resource, even if they make their connections first. Instead, the connection resources that were not specially reserved for a service and that are still available are occupied.

S7-400

In the CPUs of the S7-400, one connection resource each is preassigned (reserved) for the connection to a PG or OP.

SFC Communication: Overview

- **Data exchange using the MPI subnet or within a station**
- **No connection configuration necessary in comparison to SFB communication**
- **The connection to the partner is dynamically configured and disconnected**
- **User data up to 76 bytes**
- **Can be used on all S7-300/400 CPUs**
- **Variables can also be read and written in the S7-200 via PROFIBUS-DP (X_GET, X_PUT)**
- **The communication partners can also be found in another S7 Project**

Overview

You can exchange smaller amounts of data between an S7/M7-300/400-CPU and an additional communications capable module with the communications SFCs for non-configured connections.

The communication partners must either be connected on the same MPI subnet or be accessible within the same station via K-Bus or PROFIBUS-DP.

A configured connection is not necessary.

Connection

When a communications SFC is called, a connection is dynamically configured to the addressed communication partner and after completion of the transmission, depending on parameter assignment (Parameter: CONT) is disconnected. For the configured connection, an available connection resource each is required on the communication partners.

If in an SFC call, no available connection resource is available, then a corresponding error number is returned to the user in RET_VAL.

Already existing connections of the communications SFB cannot be used. If the active CPU goes into the Stop state during a data transmission, the existing connections are disconnected.

The communications SFCs may not be deleted in the RUN mode, since otherwise occupied connection resources may possibly not be enabled again. (Program change only in the STOP state).

User Data Size

The amount of transmittable user data is a uniform 76 bytes maximum for all S7/M7/C7-CPUs.

SFC Communication: Block Overview

SFC	NAME	Short Description
SFC 65	X_SEND	Send block for sending data to the X_RCV (Client) block
SFC 66	X_RCV	Receive block for receiving the data of the X_SEND block
SFC 67	X_GET	Read data from the partner PLC
SFC 68	X_PUT	Write data to the partner PLC
SFC 69	X_ABORT	Abort existing connection
SFC 72	I_GET	Read data from the partner CPU
SFC 73	I_PUT	Write data to the partner CPU
SFC 74	I_ABORT	Abort connection to the partner CPU

Overview

With communications SFCs you have an acknowledged data transmission using non-configured S7 connections.

You can address all communication partners on the same MPI subnet with the communications SFCs (X_...); with the SFCs (I_...), all communication partners with an I/O address (e.g. FMs, etc.) within the same station.

Communication using an MPI subnet is then also possible if the communication partner is found in another S7-Project.

The number of successively accessible communications nodes is not limited.

Addressing

In communication (X_...) using an MPI subnet, the addressing of the partner takes place by specifying the MPI address. In communication (I_...) within the same station it takes place by specifying the logical module initial address (I/O address).

If a module has a basis address for inputs (I-address) as well as one for outputs (Q address), then in an SFC call the smaller of the two must be given.

Data Consistency

The size of the maximum data area that can be read (X_PUT, I_PUT) and written (X_GET, I_GET) as a related block by the operating system with S7-300/400-CPUs , is designated as data consistency.

With S7-300/400, the data consistency is :

- S7-300-CPUs: 8 Bytes
- S7-400-CPUs: 32 Bytes

Thus, for example, an array of data type byte, word or double word can be transmitted consistent up to the maximum size.

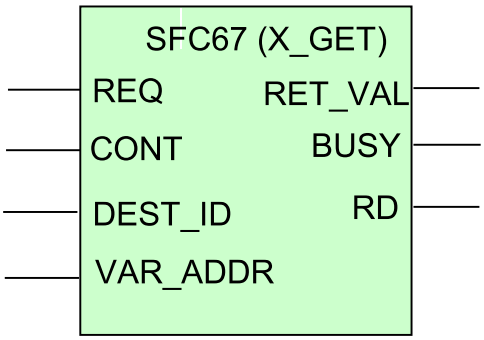
SFC Communication: X_GET (SFC 67) Block

STL Representation

With an example for the parameter assignment

```
CALL SFC 67
  REQ:= I 0.4                    //Trigger
  CONT:= FALSE                   //Disc. connection
  DEST_ID:= W#16#3               //MPI_Address
  VAR_ADDR:= P#M20.0 BYTE 10  //Remote var.
  RET_VAL:= MW100                //Error code
  BUSY:= M 4.1                   //SFC active
  RD:= P#M0.0 BYTE 10            //Local variable
```

LAD/FBD Representation



Description

With SFC 67 (X_GET) , you can read data from a communication partner that is not in the local S7 station. There is no corresponding SFC on the communication partner.

The read job is activated after calling the SFC with REQ=1. Following this, you continue to call the SFC until the data reception is indicated by BUSY=0. RET_VAL then contains the length of the received block of data in bytes.

Make sure that the receive area defined with the RD parameter (on the receiving CPU) is at least as long as the area to be read as defined by the VAR_ADDR parameter (on the communication partner). The data types RD and VAR_ADDR must also match.

Parameters for SFC 67 X_GET

Parameter	Kind	Type	Meaning
REQ	INPUT	BOOL (I,Q,M,D,L, const.)	Activates a transfer for Signal 1
CONT	INPUT	BOOL (I,Q,M,D,L,const.)	CONT=0 disconnect connection CONT=1 connection remains
DEST_ID	INPUT	WORD (I,Q,M,D,L, const.)	MPI address of the partner
VAR_ADDR	INPUT	ANY (I,Q,M,D)	Reference to the area (remote CPU), from where the data is read
RET_VAL	OUTPUT	INT (I,Q,M,D,L)	Return value with the error code
BUSY	OUTPUT	BOOL (I,Q,M,D)	BUSY=1 Send function running BUSY=0 Send function completed
RD	OUTPUT	ANY (I,Q,M,D,L)	Reference to the area (local CPU), in which the read data are written

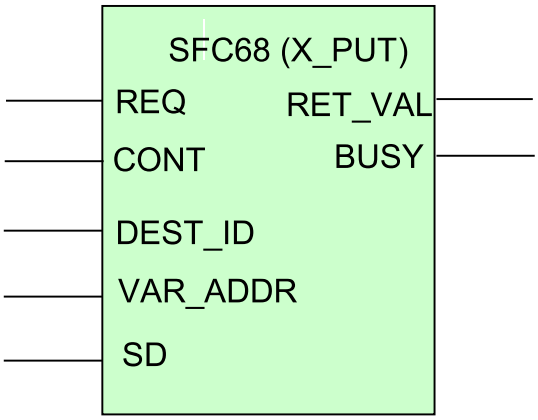
SFC Communication: X_PUT (SFC 68) Block

STL Representation

With an example for the parameter assignment

```
CALL SFC 68
  REQ:= I 0.5                    //Trigger
  CONT:= FALSE                   //Disc. connection
  DEST_ID:= W#16#3               //MPI_Address
  VAR_ADDR:= P#M20.0 BYTE 10  //Remote var.
  SD:= P#M0.0 BYTE 10            //Local variable
  RET_VAL:= MW100                //Error code
  BUSY:= M 4.1                   //SFC active
```

LAD/FBD Representation



Description

With SFC 68 (X_PUT) , you write data to a communication partner that is not in the same local S7 station. There is no corresponding SFC on the communicaiton partner.

The write job is activated after calling the SFC with REQ=1. Following this, you continue to call the SFC until the acknowledgement is received with BUSY=0.

Make sure that the send area defined with the SD parameter (on the sending CPU) is the same length as the receive area defined by the VAR_ADDR parameter (on the communication partner). The data types SD and VAR_ADDR must also match.

Parameters for SFC 68 X_PUT

Parameter	Kind	Type	Meaning
REQ	INPUT	BOOL (I,Q,M,D,L, Const.)	Activates a transfer for Signal 1
CONT	INPUT	BOOL (I,Q,M,D,L,Const.)	CONT=0 disconnect connection CONT=1 connection remains
DEST_ID	INPUT	WORD (I,Q,M,D,L, Const.)	MPI address of the partner
VAR_ADDR	INPUT	ANY (I,Q,M,D)	Reference to the area (remote CPU) in which is written
SD	INPUT	ANY (I,Q,M,D)	Reference to the area (local CPU), that contains the data to be transferred
RET_VAL	OUTPUT	INT (I,Q,M,D,L)	Return value with the error code
BUSY	OUTPUT	BOOL (I,Q,M,D)	BUSY=1 Send function running BUSY=0 Send function completed

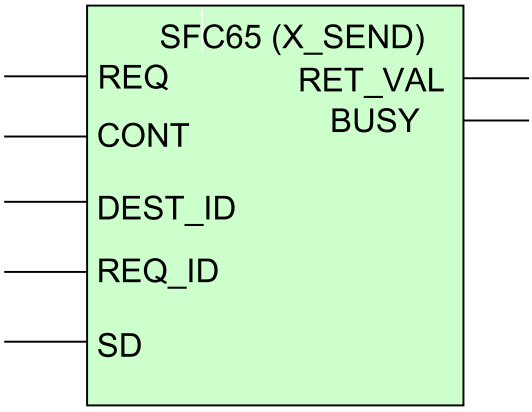
SFC Communication: X_SEND (SFC 65) Block

STL Representation

With an example for the parameter assignment

```
CALL SFC 65
  REQ:= M4.0                 //Trigger
  CONT:= FALSE               //Disc. connection
  DEST_ID:= W#16#4           //MPI_addr.
  REQ_ID:= DW#16#1           //Identifier
  SD:= P#M20.0 BYTE 10       //Variable
  RET_VAL:= MW40             //Error code
  BUSY:= M 4.1               //SFC active
```

LAD/FBD Representation



Description

With SFC 65 (X_SEND) you send data to a communication partner that is not in the same local S7-Station. Data receipt at the communication partner takes place via SFC 66 (X_RCV).

You can identify your sent data with the input parameter REQ_ID. This job identifier is also transmitted. You can evaluate this at the communication partner in order to determine the origin of the data.

The send function takes place after the call of the SFC with REQ=1.

You must make sure that the send area (on the sending CPU) defined via the parameter SD is smaller or the same as the receive area (at the communication partner) defined via the parameter RD.

Parameters for the SFC 65 X_SEND

Parameter	Kind	Type	Meaning
REQ	INPUT	BOOL (I,Q,M,D,L const.)	activates a transfer for Signal 1
CONT	INPUT	WORD (I,Q,M,D,L Const.)	CONT=0 disconnect connection CONT=1 connection remains
DEST_ID	INPUT	WORD (I,Q,M,D,L const.)	MPI address of the partner
REQ_ID	INPUT	DWORD (I,Q,M,D,L, const.)	Request ID to identify data at the partner
SD	INPUT	ANY (I,Q,M,D)	Reference to the send area
RET_VAL	OUTPUT	INT (I,Q,M,D,L)	Return value with the error code
BUSY	OUTPUT	BOOL (I,Q,M,D)	BUSY=1 Send function running BUSY=0 Send function completed

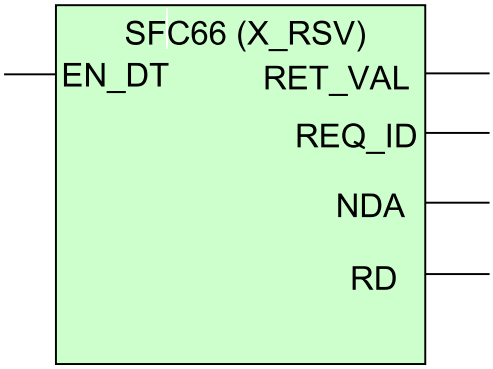
SFC Communication: X_RCV (SFC 66) Block

STL Representation

With an example for the parameter assignment

```
CALL SFC 66
  EN_DT:= TRUE              //Trigger data trans.
  RET_VAL:= MW 50           //Error code
  REQ_ID:= MD52             //Job ID
  NDA:= M40.0               //Data exist
  RD:= P#M20.0 BYTE 10      //Variable
```

LAD/FBD Representation



Description

With SFC 66 (X_RCV) you receive data that one or several communication partner(s) sent with SFC 65 (X_SEND). This (these) communication partner(s) is outside of the actual S7-Station.

With SFC 66 (X_RCV) you can:

- determine whether at this moment sent data are available. These were, if necessary, placed in an internal queue by the operating system.
- copy the oldest data block, that is available in the queue, in a receiving area specified by you.

Selection takes place via the input parameter EN_DT (enable data transfer).

Parameters for the X_RCV SFC 66

Parameter	Kind	Type	Meaning
EN_DT	INPUT	BOOL (I,Q,M,D,L, constant)	EN_DT=0 check whether the data block is present EN_DT=1 copy data block to memory
RET_VAL	OUTPUT	INT (I,Q,M,D,L)	Return value with the error code
REQ_ID	OUTPUT	DWORD (I,Q,M,D,L,)	Request identifier for the X_SEND SFC 66, whose data are present in the first position in the queue
NDA	OUTPUT	BOOL (I,Q,M,D,L)	NDA=0 no data block present NDA=1 at least 1 data block present (for EN_DT=1) or data block was copied to memory (EN_DT=1)
RD	OUTPUT	ANY (I,Q,M,D)	Reference to the receive area

SFB Communication: Overview

- **Data exchange using MPI, K-Bus, Profibus or Industrial Ethernet**
- **Configuring the connections via connection table**
- **The connections are configured during complete restart and exist permanently (even in STOP mode)**
- **User data size up to 64 KBytes**
- **Communications services also for controlling (Stop, Start) the partner**
- **SFBs exist for all S7-400-CPUs**
- **Data can also be read and written by an S7-300 (GET/PUT)**
- **Different tasks can be handled via one connection**

SIMATIC S7

Date: 30.06.2011
File: PRO2_09E.15

SITRAIN Training for Automation and Drives

Overview

The SFB blocks exist on all S7-400-CPUs and are used to exchange data with S7/M7-300/400-CPUs. Data up to 64 Kbytes can be transferred with these blocks by means of various subnets (MPI, PROFIBUS, Industrial Ethernet).

Connections

With communications SFBs you have a protected data transmission using configured S7 connections. Configuration of these connections is made with the help of the "NetPro" (network configuring) tool, that is transparently linked in the SIMATIC Manager.

The configured connections are established during a COMPLETE RESTART of the stations and remain established permanently, even when the station goes into the STOP mode. During a restart, the connections are not establishhed again.

Communication is only possible between stations of an S7-Project. The communication partners must be connected on a common MPI-, PROFIBUS- or Industrial Ethernet subnet.

SFBs

The interfaces for S7 communication to the user program in the SIMATIC S7 form special S7 blocks of the SFB type. The SFBs are oriented to the ISO/IEC 61131-5 standard and offer the user a uniform interface.

Connections must be configured for communication. The associated connection number (Identification Number) references the node assignment and the transmission medium. These Identification Numbers are handed over as block parameter "ID" during an SFB call.

User Data

The user data size depends on the block used and on the communications partner:

- PUT/GET 160 bytes to the S7-300 and 400 bytes to the S7-400/M7
- USEND/UREC up to 440 bytes
- BSEND/BRCV up to 64KBytes

SFB Communication: Block Overview

SFB/SFC	NAME	Comm. Type	Short Description
SFB 8	USEND	two-sided	Send block for sending data to the URCV (Client) block
SFB 9	URCV	two-sided	Receive block for receiving data of the USEND block
SFB 12	BSEND	two-sided	Send block for sending large data blocks to the BRCV block (up to 64 KByte)
SFB 13	BRCV	two-sided	Receive block for receiving large data blocks (up to 64 Kbyte)
SFB 14	GET	one-sided	Read data from partner PLC
SFB 15	PUT	one-sided	Write data to partner PLC
SFB 16	PRINT	one-sided	Send data to remote printer
SFB 19	START	one-sided	Perform complete restart at the partner
SFB 20	STOP	one-sided	Put partner in the Stop state
SFB 21	RESUME	one-sided	Perform restart at the partner
SFB 22	STATUS	one-sided	Status scan of the partner (RUN, STOP, start-up, hold)
SFB 23	USTATUS	one-sided	Receive the partner‘s status messages
SFC 62	CONTROL	---	Scan the internal status of an S7 connection + SFB

SFBs: S7- 400

The SFBs for S7 communication are integrated as system function blocks (SFBs) in the S7-400 CPU’s operating system.

For integration into the user program, the user can locate the block headers in the *Standard Library* in the S7 program folder *System Function Blocks*.

SFBs: S7 - 300

The S7-300 contains no SFBs for expanded communication. However, the S7-300 CPUs’ operating system supports the server functionality of the one-sided S7 communications services. Thus, for example, data from a CPU 3xx can be read or written by a CPU 4xx with the help of the GET and PUT blocks.

Function Classes

The blocks can be subdivided into a total of 4 function classes:

- send and receive functions
- control functions
- monitor functions
- connection status

SFBs for Data Exchange

The SFBs for data exchange are used for data communication between two communications capable partners (S7/M7-CPUs, M7-FMs):

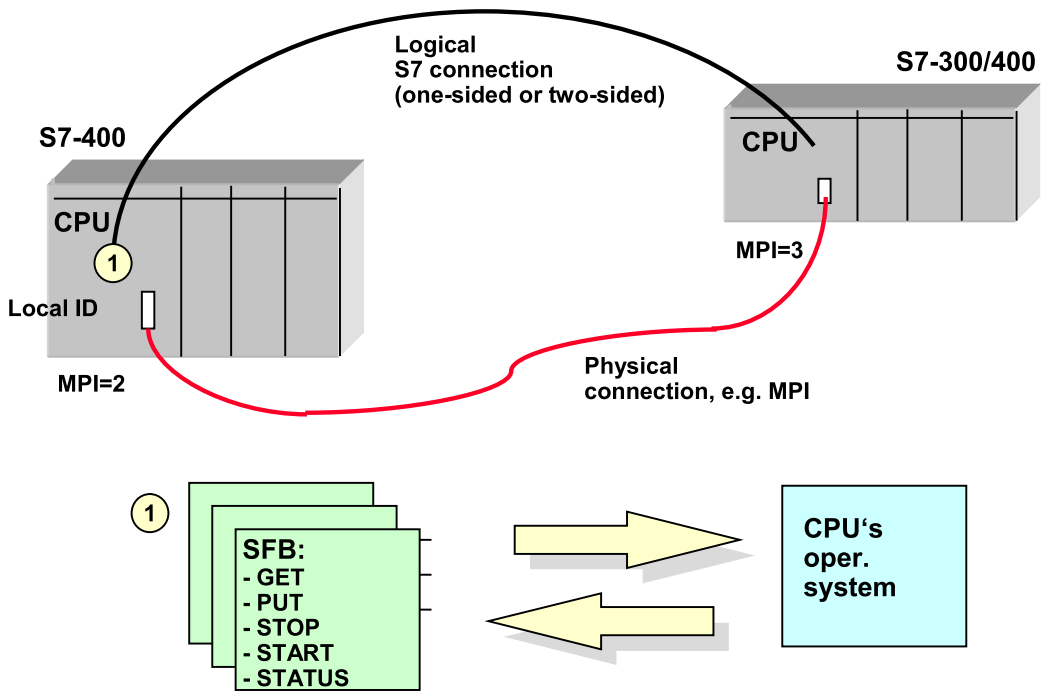
- GET, PUT (one-sided read and write variables)
- USEND/URCV (two-sided, uncoordinated send/receive)
- BSEND/BRCV (two-sided, blocked send/receive)

SFBs for Program Management

The SFBs for program management are used to control and evaluate the operating states of the partner device or the connections.

- START/STOP/RESUME (control functions)
- STATUS/USTATUS (monitor functions)
- CONTROL (connection status)

One-sided Communication Services using S7 Connections



SIMATIC S7

Date: 30.06.2011
File: PRO2_09E.17

SITRAIN Training for Automation and Drives

Overview

So that the SFBs in the respective communication partners can communicate with each other, S7 connections must first of all be configured.

S7 connections can be configured for MPI, Industrial Ethernet and PROFIBUS networks.

One-sided S7 Connections

From an S7-400 to an S7-300, one-sided S7 connections are automatically setup by the configuration tool. For one-sided connections, a local connection ID for identification of the connection, that is, the communication partner and the transmission medium, is assigned only on the S7-400 side (Client side).

No connection ID is assigned on the S7-300 side since the SFBs for addressing the communication connection are not located in the S7-300 CPU‘s operating system.

Only one-sided communication services can be called via one-sided connections. A corresponding SFB call is only necessary on the Client side (S7-400) for one-sided communication services. On the other communications partner (Server), the service is completely handled by the operating system. Programming work by the user is not necessary on the Server side.

One-sided S7 connections are always configured by the Client during start-up.

"One-sided" SFBs

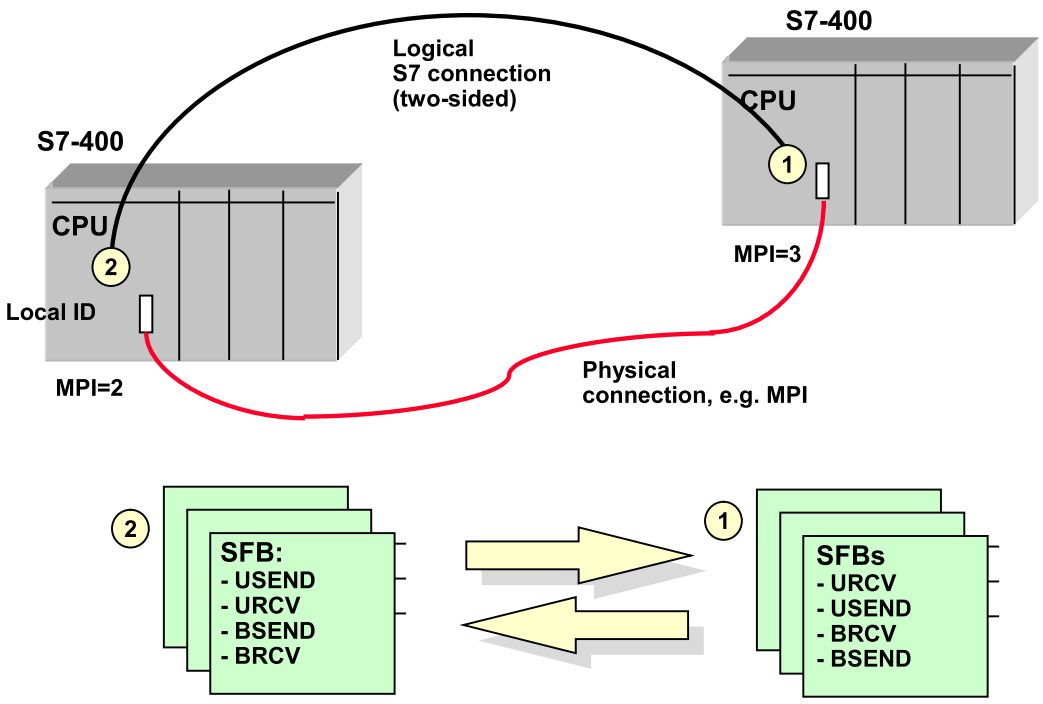
SFBs that are considered as one-sided communication services are:

- GET, PUT
- STOP, START, RESUME
- STATUS, USTATUS

With one-sided communication services, the user program on the Server side is not informed when new data have been transmitted.

.

Two-sided Communication Services using S7 Connections



SIMATIC S7

Date: 30.06.2011
File: PRO2_09E.18

SITRAIN Training for Automation and Drives

Two-sided S7 Connections

Two-sided S7 connections are automatically set up in the configuration of S7 connections between two S7-400 CPUs. A connection ID is assigned on each side of a two-sided connection. Both sides can then reference the connection using this connection ID.

Thus, each of the two partners can appear as the Initiator (Client) of a communication service.

One-sided (PUT, GET, etc.) as well as two-sided communication services can be completed using two-sided connections.

With two-sided S7 connections you can decide in the configuration which node initiates the connection configuration.

"Two-sided" SFBs

The blocks

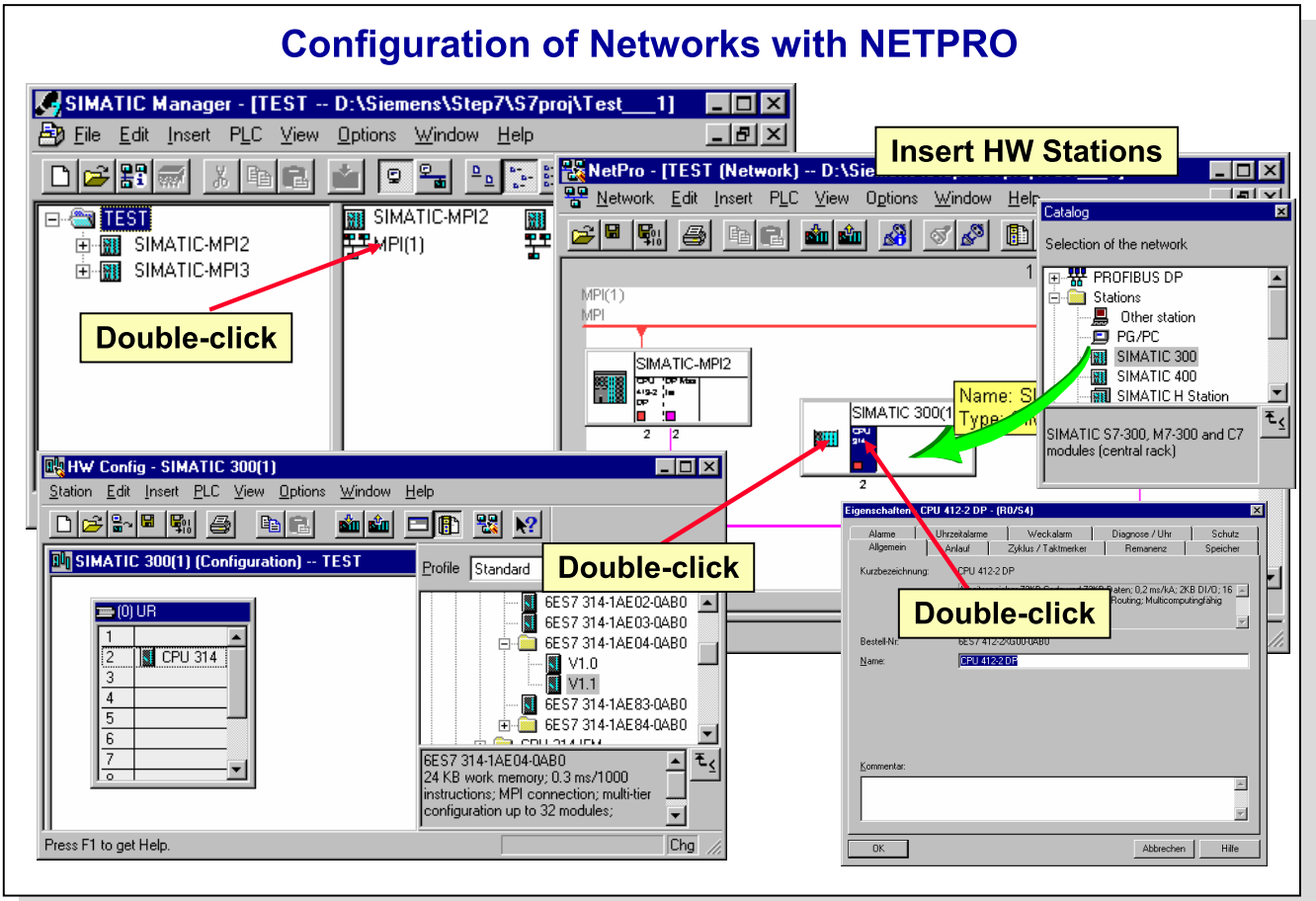
- BSEND=Sender (Client) ==> BRCV Receiver (Server)
- USEND=Sender (Client) ==> URCV Receiver (Server)

are considered as two-sided SFBs.

These blocks must always be installed as block pairs. Two-sided communications functions are always then configured when a data transfer is used for specific further processing of data.

On the one hand, the receiver (Server) can determine by the block call URCV or BRCV, when it is ready to receive new data from the Sender for further processing. On the other hand, the receiver can, by scanning the SFB-Parameters #NDR (New Data Received), be informed if new data were received.

Configuration of Networks with NETPRO



Introduction

A graphical configuration of networks (MPI, Profibus or Industrial Ethernet) can be carried out with the help of the tool "NETPRO". The advantage lies in the clarity, documentation and the easy call of the participating tools such as Hardware Configuration.

Call

The tool is called with a double-click on the network symbol, for example, MPI in the SIMATIC Manager. After the network configuration is opened, a window for the graphic view of the network appears. When NETPRO is called, the following is displayed:

- all subnets that were created in the project so far
- all stations that were configured in the project so far

Insert Objects

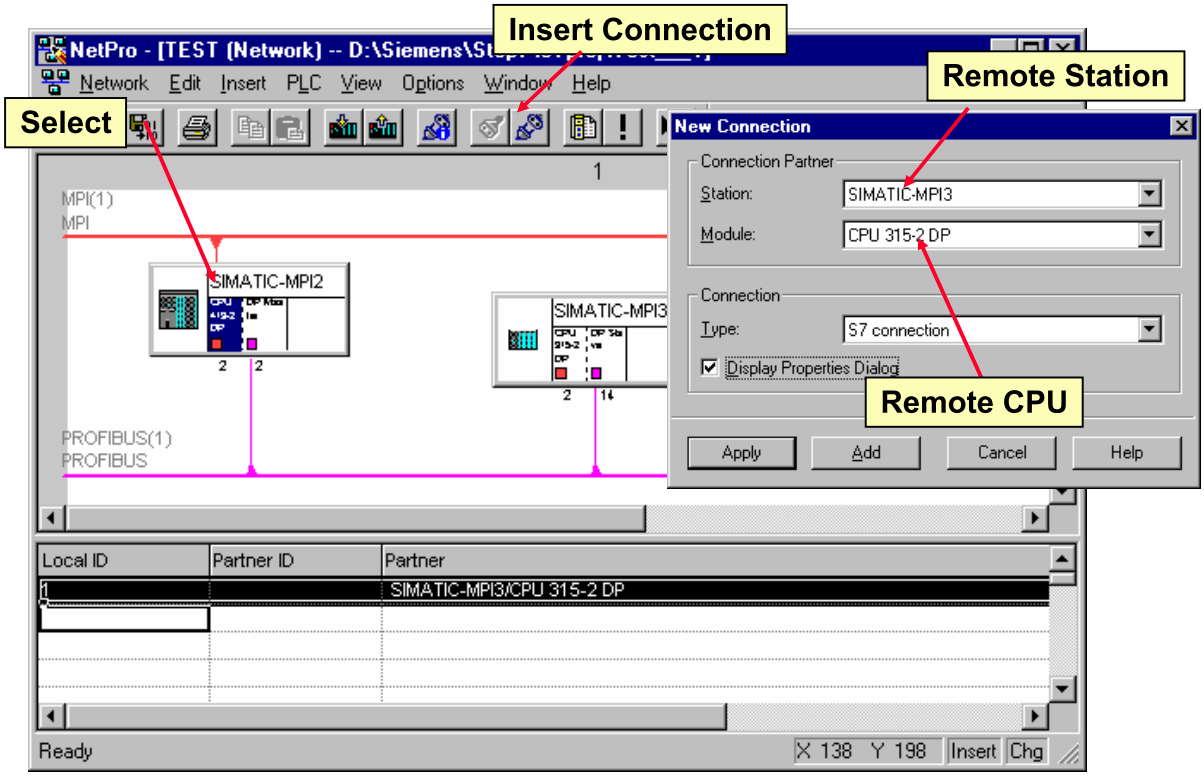
In NETPRO, you can insert all network objects, such as subnets or stations, from a catalog using drag and drop.

Configuring Hardware

After you have inserted the stations, you get to the "Configure Hardware" tool by double-clicking on the "Hardware symbol" of a station. Here, you can insert the modules into the stations and assign parameters to them.

For the interfaces of CPUs, you can, among other things, also define the MPI/PROFIBUS addresses and the connection to the subnet.

Configuration of S7 Connections



Overview

The establishment of the required communication connections is a prerequisite for program controlled data exchange using SFBs. All connections that go out from a module are displayed in the connection table belonging to the module.

Generating Connections

Connections to the distant partner can only then be set up when the local and the distant station are connected to the same subnet.

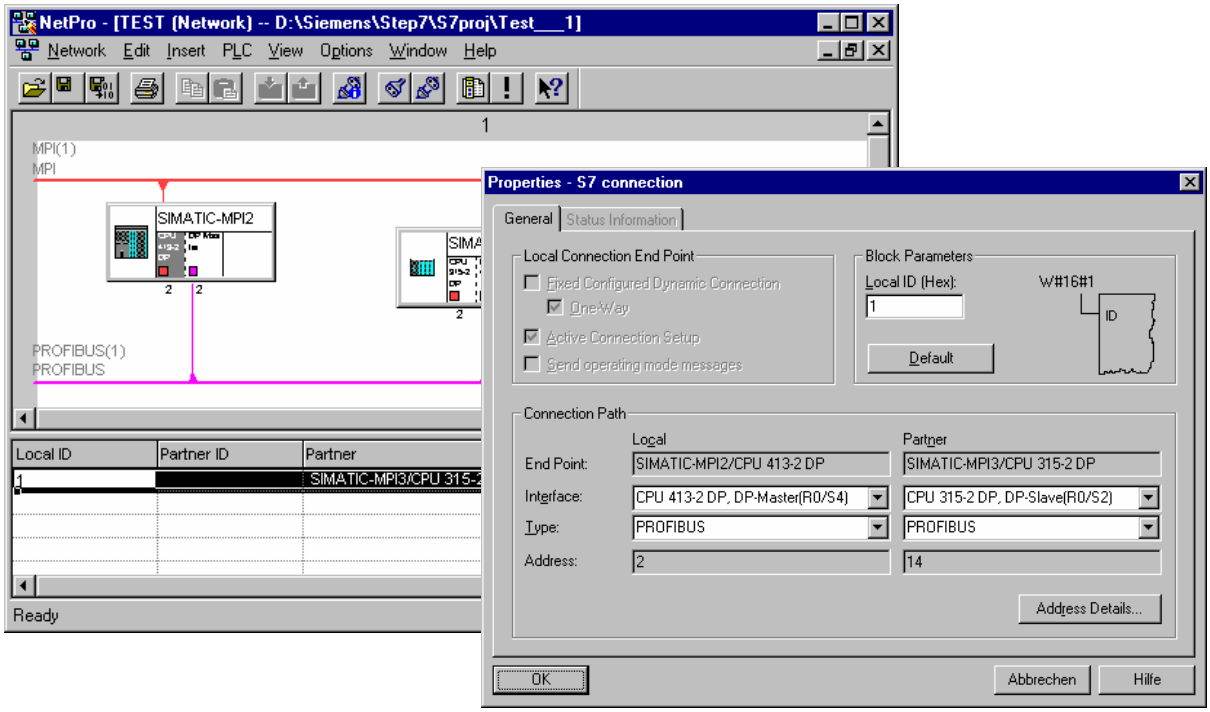
In order to insert a new connection, proceed as follows:

1. In the fields "Station" and "Module" select the programmable module, from which you want to open up a connection (local station).
2. Double-click in an empty line in the connection table or select the menu option *Insert -> Connection...*. The dialog "New Connection" is opened.
3. In the fields "Station" and "Module" select the programmable module, to which the connection is to lead (connection partner or also called "Remote Station").
4. In the field “Type” select the connection type: *S7 Connection*.
5. Activate the check box "Show Properties Dialog Box", if you want to look at or change the connection properties after "OK" or "Add".
6. Confirm your entries by clicking the "OK" button.

Result:

STEP 7 enters the connection in the connection table of the local station and issues the Local ID and, if necessary, the Partner ID for this connection. You require these IDs for programming the communication SFBs (value for the block parameter “ID”).

Establishing Connection Properties



Overview

Besides the establishment of connection partner and connection type, you can, depending on the connection type establish additional properties.

Establishing Object Properties

In order to establish special object properties of a communication connection, proceed as follows:

1. Mark the connection for which you want to establish object properties.
2. Select the menu option *Edit -> Object Properties*. The "Object Properties" dialog is opened.

In this dialog you can establish the following properties.

Active Connection Set Up

In two-sided connections, you can select which of the two nodes is to trigger the connection configuration during the complete restart.

Send Operating Mode Messages

When activated, the local node sends its operating mode messages (STOP,START, HOLD,.....) to the partner or to SFB 23: USTATUS in the partner CPU.

Local ID

The local ID of the connection is displayed here. You can change the local ID. This would make sense, if you have already programmed communication SFBs and you also want to use the ID programmed in the call for the identification of the connection.

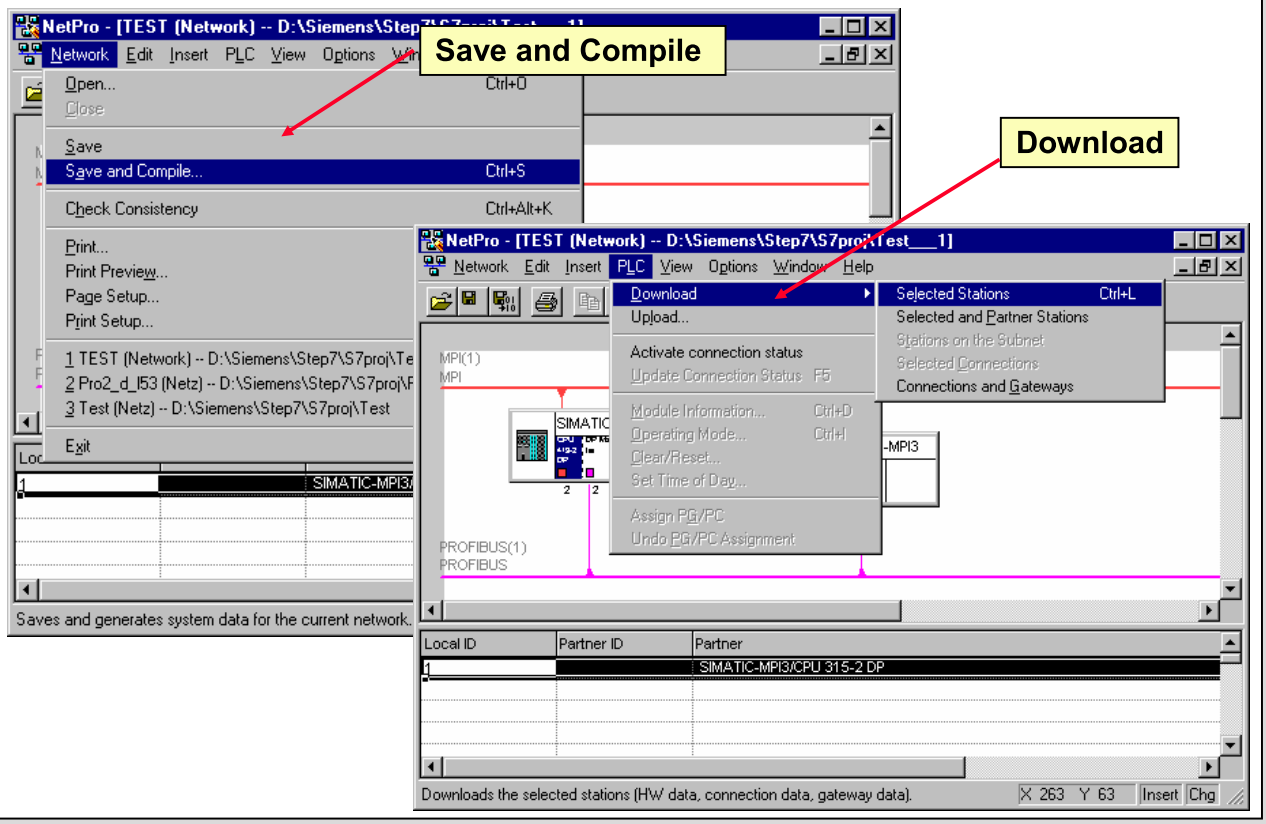
You enter the new local ID as a hexadecimal number. It must be in the value range 1 to FFF for an S7 connection and cannot already have been assigned.

Network Connections

These fields display over which path the data exchange runs

If several communications paths (subnets) exist between the two nodes, a choice can be made over which communications path the data exchange is to be completed.

Compiling and Downloading the Configuration Data



Compiling and Saving

Before you can download the connection data into the individual stations (Download in PLC), the connection table must be saved in NETPRO and compiled into connection data. This takes place with the help of the menu option *File -> Save and Compile*.

In the dialog field that pops up, you can select between two alternatives:

Compile and Check All : Saves all connections and checks all connections for consistency within a project. All connections are compiled and stored in the system data. In case inconsistency arises, a dialog field appears in which the errors are displayed.

Select "Compile and Check All ", if you have made changes in the network configuration (e.g. changed node addresses, deleted node or subnet). It is possible that connections no longer exist and only “Compile and Check All” gives this information.

Compile Changes Only: Saves all connections within the project and compiles those connections that were changed since the last execution of “Save and Compile”.

When you end the connection configuration, a question appears on the screen, asking if the changed data is to be saved or not. After acknowledging the question with "Yes", the changed connection data is saved and compiled into system data.

Downloading the Configuration Data

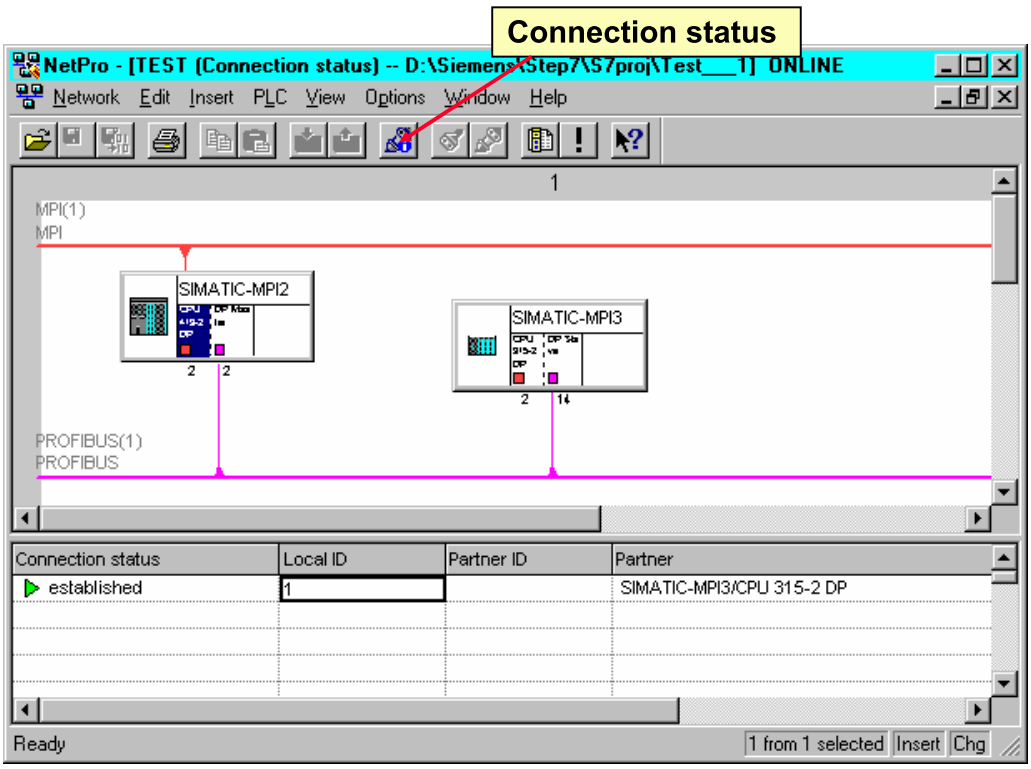
From the saving of the connection table, connection data result that must be downloaded into the participating modules. Downloading the connection table into the module is possible via the MPI, PROFIBUS or Industrial Ethernet interface of the module.

There are five ways to download the data into the PLCs:

- Download -> Selected Stations (+ Partner Stations)
- Download -> Stations on the Subnet
- Download -> Selected Connections (Connections and Gateways)

(for further information: see On-line Help)

Testing the Connection Status



"Connection Status" Column

The "Connection status" column of the connection table is only overlaid when the function "Connection status" was activated.
To activate this function, use the menu options PLC -> Activate connection status.

You can call a dialog box with detailed information on the connection status of every connection status (“being established", for example) using the menu options Edit -> Object properties.

Note

The "Connection status" column has a yellow background when the connection is only available online. That is, offline is not available in the project. The cause for this can be that this connection was loaded into the module without first saving it in the project.

SFB Communication: GET (SFB 14) Block

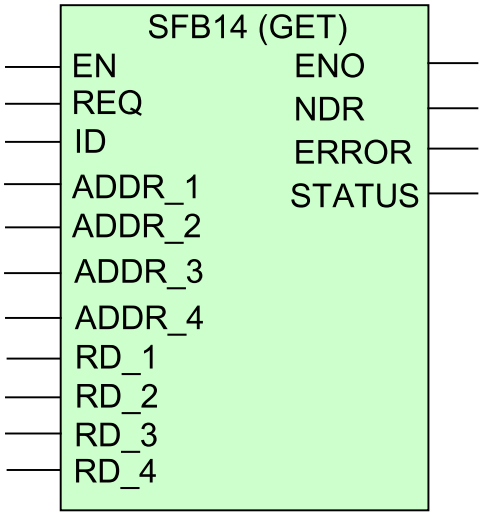
STL Representation

With an example for the parameter assignment

```
CALL GET, I_GET
  REQ:=I 0.2                  //Start
  ID:=W#16#1                  //Connection No.
  NDR:=#NDR_FLAG              //Receive new data
  ERROR:= #ERROR_F            //End with error
  STATUS:= #STATUS_W          //Additional info
  ADDR_1:=P#I 0.0 BYTE 1      //1. remote var.
  ADDR_2:=P#I 4.0 WORD 1      //2. remote var.
  ADDR_3:=                    //3. remote var.
  ADDR_4:=                    //4. remote var.
  RD_1:=P#Q 0.0 BYTE 1        //1. local var.
  RD_2:=P#Q 4.0 WORD 1        //2. local var.
  RD_3:=                      //3. local var.
  RD_4:=                      //4. local var.
```

LAD Representation

DB14 (Instance DB)



Overview

With SFB14 (GET) you can read data from a remote CPU.

With a rising edge at control input REQ, a read task is sent to the partner CPU. The remote partner returns the data.

If no errors occurred, the received data are copied to the configured receive areas (RD_i) in a renewed SFB call. The completion of the job is indicated by a 1 at the status parameter NDR.

Parameter	Kind	Type	Meaning
REQ	INPUT	BOOL (I,Q,M,D,L constant)	Activates a transfer with a positive edge.
ID	INPUT	WORD (I,Q,M,D,L constant)	Refer to the connection table for the connection number.
ADDR_1 ... ADDR_4	IN_OUT	ANY (I,Q,M,D)	Pointer to the areas in the partner CPU to be read.
RD_1 ... RD_4	IN_OUT	ANY (I,Q,M,D)	Pointer to the areas in your own CPU in which the read values are to be stored. (data area of the partner CPU **ADDR_1** ==> **RD_1**-data area of your own CPU
NDR	OUTPUT	BOOL (I,Q,M,D,L)	Positive edge (pulse) signalizes the user program there are new receive data available. “Data transferred from the partner CPU without errors.”
ERROR	OUTPUT	BOOL (I,Q,M,D,L)	Positive edge signalizes errors (pulse).
STATUS	OUTPUT	WORD (I,Q,M,D,L)	Contains a detailed error statement or warning (decimal).

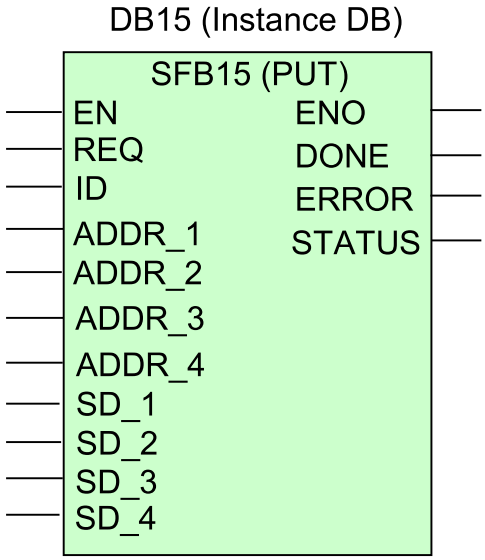
SFB Communication: PUT (SFB 15) Block

STL Representation

With an example for the parameter assignment

```
CALL PUT, I_PUT(Instance DB)
  REQ:=I 0.3                    //Start
  ID:=W#16#1                    //Connection No.
  DONE:= #DONE_F                //Successful end
  ERROR:= #ERROR_F              //End with error
  STATUS:= #STATUS_W            //Error info
  ADDR_1:=P#Q 12.0 WORD 1       //1. remote var.
  ADDR_2:=                      //2. remote var.
  ADDR_3:=                      //3. remote var.
  ADDR_4:=                      //4. remote var.
  SD_1:=P#I 2.0 WORD 1          //1. local var.
  SD_2:=                        //2. local var.
  SD_3:=                        //3. local var.
  SD_4:=                        //4. local var.
```

LAD Representation



Overview

With SFB15 (PUT), you can write data to a remote CPU.

With a rising edge at the control input REQ, the pointers to the areas to be written (ADDR_i) and the data (SD_i) are sent to the partner CPU. The remote partner saves the required data under the addresses supplied with the data and returns an execution acknowledgment.

Parameter	Kind	Type	Meaning
REQ	INPUT	BOOL (I,Q,M,D,L constant)	Activates a transfer with a rising edge
ID	INPUT	WORD (I,Q,M,D,L constant)	Refer to the connection table for the connection number.
ADDR_1 ... ADDR_4	IN_OUT	ANY (I,Q,M,D)	Pointer to the data areas in the partner CPU in which the data from the send CPU is to be written.
SD_1 ... SD_4	IN_OUT	ANY (I,Q,M,D)	Pointer to the data areas in your own CPU to be sent to the partner CPU. (data area of your own CPU- **SD_1** ==> **ADDR_1 data area of the partner** CPU)
DONE	OUTPUT	BOOL (I,Q,M,D,L)	Positive edge (pulse) signalizes the user program: transfer concluded without errors.
ERROR	OUTPUT	BOOL (I,Q,M,D,L)	Positive edge signalizes errors (pulse)
STATUS	OUTPUT	WORD (I,Q,M,D,L)	Contains a detailed error statement or warning (decimal).

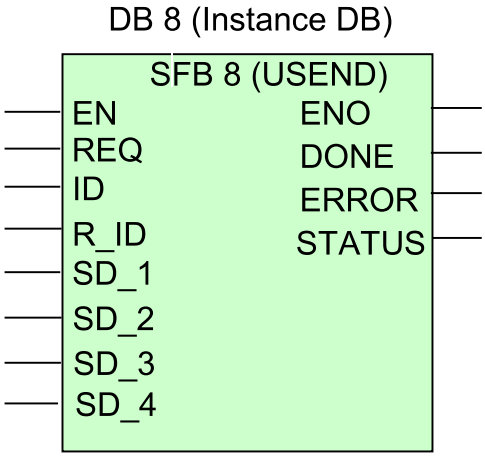
SFB Communication: USEND (SFB 8) Block

STL Representation

with an example for the parameter assignment

```
CALL USEND, I_USEND(Instance DB)
  REQ:= I 0.4                        //Start
  ID:=W#16#3                         //Connection No.
  R_ID:=DW#16#B1                     //Block pair
  DONE:= #DONE_F                     //Successful end
  ERROR:= #ERROR_F                   //End with error
  STATUS:= #STATUS_W                 //Error info
  SD_1 :=P#DB3.DBX0.0 BYTE 100     //1. local var.
  SD_2 :=P#DB3.DBX100.0 BYTE 100 //2. local var.
  SD_3 :=P#DB3.DBX200.0 BYTE 100 //3. local var
  SD_4 :=P#DB3.DBX300.0 BYTE 154 //4. local var.
```

LAD Representation



Overview

SFB8 (USEND) sends data to a remote partner SFB of the type "URCV" (the parameter R_ID must be identical for both SFBs). The data is sent following a rising edge at control input REQ. The function is executed without coordination with the partner SFB.

The data to be sent is referenced by the parameters SD_1, to SD_4 but not all four send parameters need to be used.

Parameter	Kind	Type	Meaning
REQ	INPUT	BOOL (I,Q,M,D,L constant)	Activates a transfer with a rising edge
ID	INPUT	WORD (I,Q,M,D,L constant)	Connection number for the S7 single-system connection (see connection table)
R_ID	INPUT	WORD (I,Q,M,D,L constant)	The parameter must be identical for both CFBs (USEND and URCV). Assignment of the block pairs
DONE	OUTPUT	BOOL (I,Q,M,D,L)	Positive edge (pulse) signalizes the user program: transfer concluded without errors.
ERROR	OUTPUT	BOOL (I,Q,M,D,L)	Positive edge signalizes errors (pulse).
STATUS	OUTPUT	BOOL (I,Q,M,D,L)	Status display if ERROR = 1
SD_1 ... SD_4	IN_OUT	ANY (I,Q,M,D)	Pointer to those data areas in your own CPU to be sent to the partner CPU. (Data area of your own CPU- **SD_1** ==> RD_1 data area of the partner CPU must be in agreement in the number, length, and data type.)

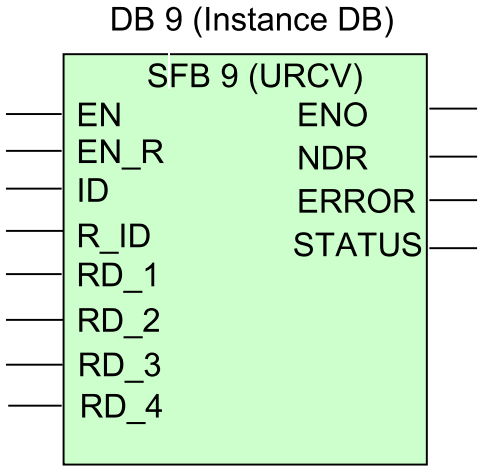
SFB Communication: URCV (SFB 9) Block

STL Representation

with an example for the parameter assignment

```
CALL URCV, I_URCV              //With Instance DB)
  EN_R:= I 0.5                 //Start
  ID:= W#16#3                  //S7 Connection
  R_ID:= DW#16#B1              //Block pair
  NDR:= #NDR_F                 //Receive new data
  ERROR:= #ERROR_F             //End with error
  STATUS:= #STATUS_W           //Error info
  RD_1:=P#DB3.DBX0.0 BYTE 100     //1. var.
  RD_2:=P#DB3.DBX100.0 BYTE 100 //2. var.
  RD_3:=P#DB3.DBX200.0 BYTE 100 //3. var.
  RD_4:=P#DB3.DBX300.0 BYTE 154 //4. var.
```

LAD Representation



Overview

SFB9 (URCV) receives data asynchronously from a remote partner SFB of the type "USEND". (The parameter R_ID must be identical in both SFBs.) If the value 1 is applied to the control input EN_R when the block is called, the received data are copied to the configured receive areas. These data areas are referenced by the parameters RD_1 to RD_4.

When the block is first called, the "receive mail box" is created. With all further calls, the data to be received must fit into this receive mail box.

Parameter	Kind	Type	Meaning
EN_R	INPUT	BOOL (I,Q,M,D,L constant)	For RLO = 1 the received data are copied to the configured data areas.
ID	INPUT	WORD (I,Q,M,D,L constant)	Connection number for the S7 single-system connection (see connection table)
R_ID	INPUT	DWORD (I,Q,M,D,L constant)	The parameter must be identical for both CFBs (USEND and URCV). Assignment of the block pairs
NDR	OUTPUT	BOOL (I,Q,M,D,L)	Positive edge (pulse) signalizes the user program: new data transferred.
ERROR	OUTPUT	BOOL (I,Q,M,D,L)	Positive edge = error (pulse)
STATUS	OUTPUT	BOOL (I,Q,M,D,L)	Status display if ERROR = 1
RD_1 ... RD_4	IN_OUT	ANY (I,Q,M,D)	Pointer to the data areas in the CPU where the received data are to be stored. (SD_i and RD_i must be in agreement as regards number, length, and data type.)

SFB Communication: BSEND (SFB 12) Block

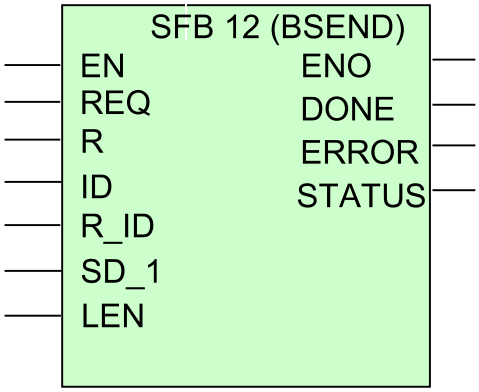
STL Representation

With an example for the parameter assignment

```
CALL BSEND, I_BSEND          //With Instance DB
REQ:= I 0.4                  //Start
R:= I 0.5                    //Reset BSEND
ID:=W#16#3                   //S7 Connection
R_ID:=DW#16#B2               //Block pair
DONE:= #DONE_F               //Successful end
ERROR:= #ERROR_F             //End with error
STATUS:= #STATUS_W           //Additional info
SD_1:=P#DB1.DBX0.0 BYTE 40000 //Send data
LEN:= #DB_LEN                //Length of data
```

LAD Representation

DB 12 (Instance DB)



Overview

SFB12 (BSEND) sends data to a remote partner SFB of the type "BRCV". (The parameter R_ID must be identical in the corresponding SFBs.). With this data transfer up to 64 KByte data can be transferred (applies to all CPUs).

The send job is activated after calling the block and when there is a rising edge at the control input REQ. The transmission of the data from the user memory is asynchronous to the processing of the user program.

Parameter	Kind	Type	Meaning
REQ	INPUT	BOOL (I,Q,M,D,L constant)	Activates a transfer with a positive edge
R	INPUT	BOOL (I,Q,M,D,L constant)	Activates resetting BSEND to the initial state with a positive edge
ID	INPUT	WORD (I,Q,M,D,L constant)	Connection number for the S7 single-system connection (see connection table)
R_ID	INPUT	DWORD (I,Q,M,D,L)	The parameter must be identical for both CFBs (BSEND and BRCV). Assignment of the block pair
SD_1	IN_OUT	ANY (I,Q,M,D,L)	Data to be sent, the length in the any pointer is not evaluated
LEN	IN_OUT	WORD (I,Q,M,D,L)	Length of the data block to be transferred
DONE	OUTPUT	BOOL (I,Q,M,D,L)	Signalizes the error-free termination of the BSEND request (pulse) with a rising edge
ERROR	OUTPUT	BOOL (I,Q,M,D,L)	Positive edge signalizes an error (pulse)
STATUS	OUTPUT	WORD (I,Q,M,D,L)	Contains a detailed error statement or warning

SFB Communication: BRCV (SFB 13) Block

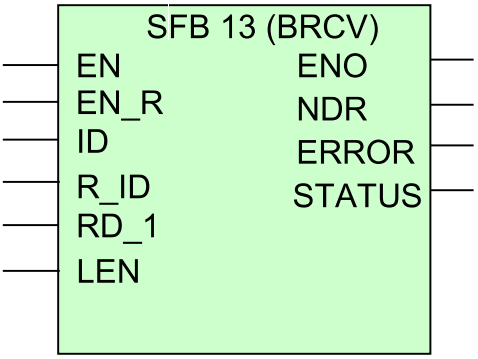
STL Representation

With an example for the parameter assignment

```
CALL BRCV, I_BRCV          // With Instance DB
  EN_R:= I 0.4             //Start
  ID:=W#16#3               //S7 Connection
  R_ID:=DW#16#B2           //Block pair
  NDR:= #NDR_F             //Receive new data
  ERROR:= #ERROR_F         //End with error
  STATUS:= #STATUS_W       //Additional info
  RD_1:=P#DB2.DBX0.0 BYTE 40000 //Rec. mail box
  LEN:= #DB_LEN            //Rec. mail box length
```

LAD Representation

DB 13 (Instance DB)



Overview

SFB13 (BRCV) receives data from a remote partner SFB of the type "BSEND". (The parameter R_ID must be identical in both SFBs.) After it has been called and the value 1 is applied at the control input EN_R, the block is ready to receive data. The start address of the receive area is specified by RD_1.

Following each received data segment, an acknowledgment is sent to the partner SFB and the LEN parameter is updated. If the block is called during asynchronous reception of data, this leads to a warning being output in the STATUS status parameter; if the call is made when the value 0 is applied to control input EN_R, reception is terminated and the SFB returns to its initial state. Error free reception of all the data segments is indicated by the status parameter NDR having the value 1.

Parameter	Kind	Type	Significance
EN_R	INPUT	BOOL (I,Q,M,D,Lconst.)	RLO = 1 SFB is ready to receive RLO = 0 procedure is cancelled
ID	INPUT	WORD (I,Q,M,D,Lconst.)	Connection number of the S7 Single System Connection (see connection table)
R_ID	INPUT	DWORD (I,Q,M,D,L const.)	The parameters must be identical for both CFBs (BSEND and BRCV) . Assignment of the block pairs
RD_1	IN_OUT	ANY	Pointing to the receive mailbox. The length specification specifies the maximum length of the block to be received. (for 2048 words, this corresponds to the joker length for S5.)
LEN	IN_OUT	WORD	Length of the data received until now in bytes
NDR	OUTPUT	BOOL	A positive edge signals the user program : new receive data accepted
ERROR	OUTPUT	BOOL	A positive edge signalizes errors (pulse)
STATUS	OUTPUT	WORD	Contains a detailed error statement or warning

SFB Communication: STOP (SFB20) Block

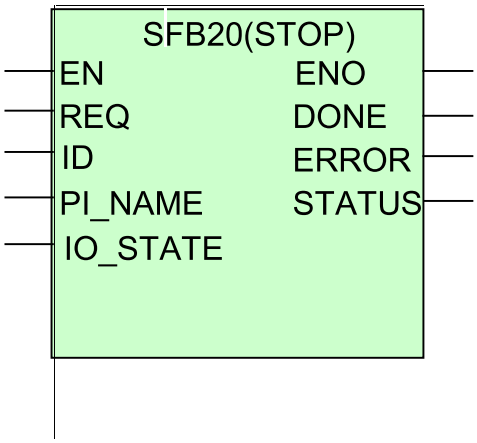
STL Representation

With an example for the parameter assignment

```
CALL "STOP","I_STOP"          //Instance DB
  REQ:= I 0.0                 //Start edge
  ID:= W#16#1                 //Connection No.
  PI_NAME:= P#M100.0 Byte 9   //See footnote
  IO_STATE:=                  //Not used
  DONE:= #DONE_F_20           //Successful end
  ERROR:= #ERROR_F_20         //End with error
  STATUS:= #STATUS_W_20       //Error info
```

LAD/FBD Representation

DB20 (Instance DB)



* Detail of the position in the memory for the beginning of: 'P_PROGRAM"

Overview

If there is a rising edge at control input REQ, SFB20 (STOP) activates a change to the STOP state on the remote device addressed by the ID. The mode change is possible when the device is in the RUN, HOLD or startup modes.

Successful execution of the job is indicated with 1 at the status parameter DONE. If any errors occur they are indicated in the status parameters ERROR and STATUS.

The renewed mode change can only be started again in the same remote device when the previous SFB20 call has been completed.

Parameter	Kind	Type	Meaning
REQ	INPUT	BOOL	With a positive edge, activates a STOP in the device addressed by ID
ID	INPUT	WORD (I,Q;M,D,L, constant)	Refer to the connection table for the connection number.
PI_NAME	IN_OUT	ANY	Pointer to the memory area in which the name of the program to be started (ASCII code) is located. The name must be P_PROGRAM for S7.
IO_STATE	IN_OUT	BYTE	Execution argument (not relevant)
DONE	OUTPUT	BOOL	Positive edge = function executed
ERROR	OUTPUT	BOOL	Positive edge = error
STATUS	OUTPUT	WORD	Contains a detailed error statement or warning (decimal)

SFB Communication: START (SFB19) Block

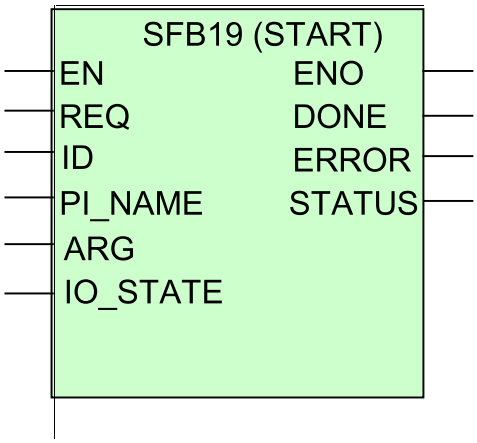
STL Representation

With an example for the parameter assignment

```
CALL "START","I_START"          //With Instance DB
  REQ:= I 0.1                   //Start edge
  ID:= W#16#1                   //Connection No.
  PI_NAME:= P#M100.0 Byte 9     //See footnote
  ARG:=                         //Not used
  IO_STATE:=                    //Not used
  DONE:= #DONE_F_20             //Successful end
  ERROR:= #ERROR_F_20           //End with error
  STATUS:= #STATUS_W_20         //Error info
```

LAD/FBD Representation

DB19 (Instance DB)



* Detail of the position in the memory for the beginning of: 'P_PROGRAM"

Overview

If there is a rising edge at control input REQ, SFB19 (START) activates a complete restart on the remote device addressed by the ID. The following conditions must be met if the remote device is a CPU:

- The CPU must be in the STOP state.
- The keyswitch of the CPU must be set to "RUN" or "RUN-P".

Once the complete restart is completed, the CPU changes to the RUN mode and sends a positive execution acknowledgment. When the positive acknowledgment is evaluated, the status parameter #DONE is set to 1. If any errors occur, they are indicated by the status parameters #ERROR and #STATUS.

Parameter	Kind	Type	Meaning
REQ	INPUT	BOOL	Activates a complete restart in the device the ID addressed with a positive edge.
ID	INPUT	WORD (I,Q;M,D,L, constant)	Refer to the connection table for the connection number.
PI_NAME	IN_OUT	ANY	Pointer to the memory area in which the name of the program to be started (ASCII code) is present. The name P_PROGRAM must be present for S7.
ARG	IN_OUT	ANY	Execution argument (not relevant)
IO_STATE	IN_OUT	ANY	Execution argument (not relevant)
DONE	OUTPUT	BOOL	Positive edge = function executed
ERROR	OUTPUT	BOOL	Positive edge = error
STATUS	OUTPUT	WORD	Contains a detailed error statement or warning (decimal)

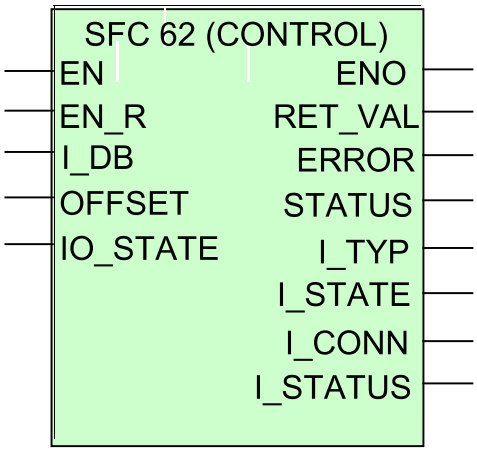
SFB Communication: CONTROL (SFC 62) Block

STL Representation

With an example for the parameter assignment

```
CALL  "CONTROL"
  EN_R:= I 0. 2          //Start
  I_DB:= W#16#F          //Instance DB NO
  OFFSET:= W#16#0        //For multi-instances
  RET_VAL:= MW4          //Error info
  ERROR:= Q 0.4          //End with error
  STATUS:= MW 4          //Status info
  I_TYP:= MB 52          //SFB TYPE
  I_STATE:= MB 53        //SFB State
  I_CONN:= M 54.0        //Connection state
  I_STATUS:= MW102       //Status of SFB
```

LAD Representation



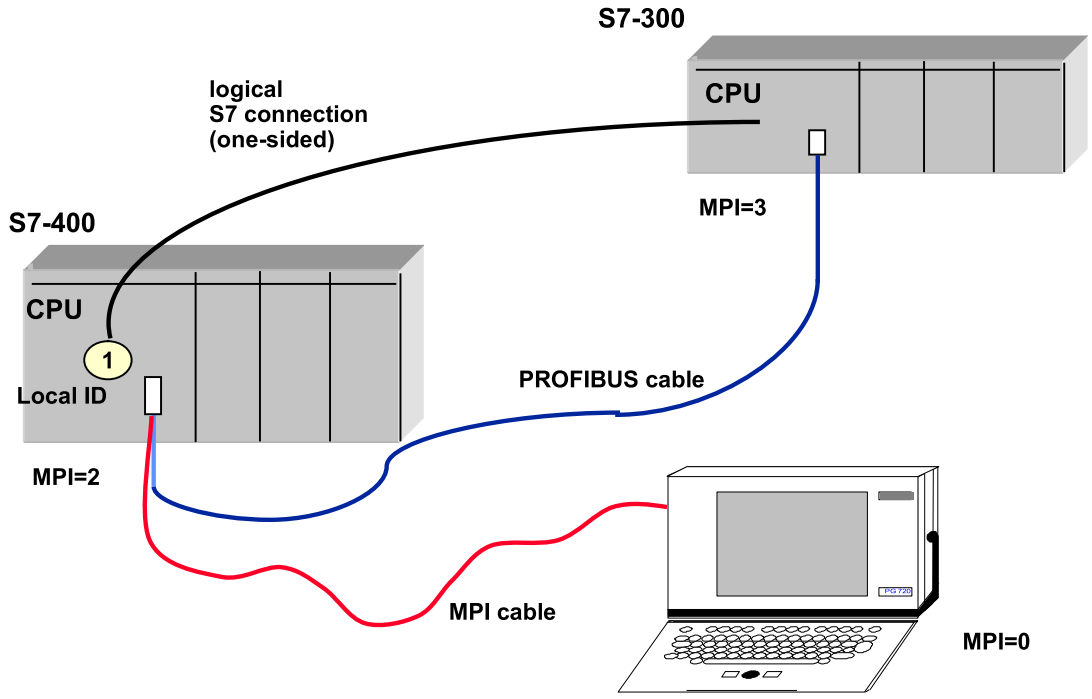
Overview

With SFC62 "CONTROL", you can query the status of a connection belonging to a local communication SFB instance.

After calling the system function with the value 1 at control input EN_R, the current status of the connection belonging to the communication SFB instance selected with I_DB is queried.

Parameter	Kind	Type	Meaning
EN_R	INPUT	BOOL	Control parameter for enabling the function
I_DB	INPUT	BLOCK_DB (I,Q;M,D,L, constant)	Instance DB number
OFFSET	INPUT	WORD (I,Q;M,D,L, constant)	Offset for multi-instances, 1st byte number of the instance DB (if no multi-instance = 0)
RET_VAL	OUTPUT	INT (I,Q;M,D,L)	8000H error for the SFC62
ERROR	OUTPUT	BOOL (I,Q;M,D,L)	RLO = 1 error during execution of SF C62
STATUS	OUTPUT	WORD (I,Q;M,D,L)	Error display for SFC 62
I_TYP	OUTPUT	BYTE (I,Q;M,D,L)	CFB type identifier
I_STATE	OUTPUT	BYTE (I,Q;M,D,L)	Identifier of the current status graph of the CFB
I_CONN	OUTPUT	BOOL (I,Q;M,D,L)	Status of the relevant connection 0 = connection cancelled 1 = connection present
I_STATUS	OUTPUT	WORD (I,Q;M,D,L)	Error or SFBs' STATUS

Exercise 10.1: Configuring an S7 Connection



SIMATIC S7

Date: 30.06.2011
File: PRO2_09E.33

SITRAIN Training for Automation and Drives

Task

Network the two S7-400 and S7-300 stations and configure an S7 connection.

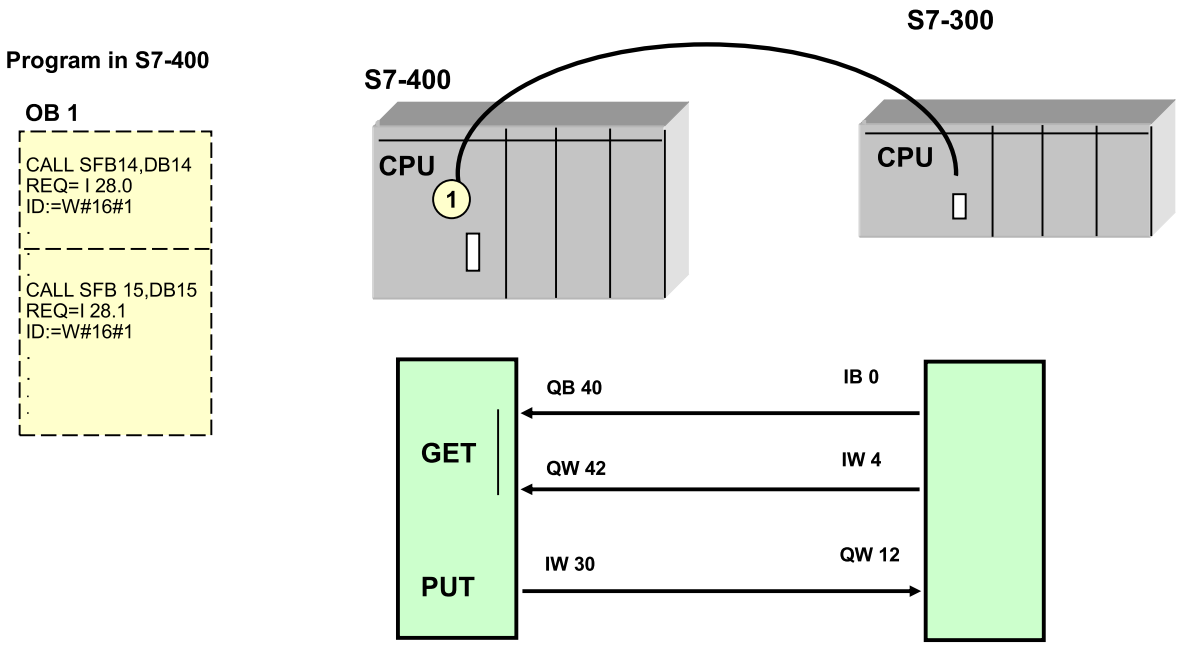
What to Do

1. Create a new project "SFB-Comm".
2. Generate two HW stations for S7-400 and S7-300 in your project.
3. In *HW Config* configure different MPI addresses for the two CPUs and "network" the two CPUs with the mutual "MPI Network" object in your project.
3. Then download the configuration data to the individual CPUs using the *HW Config* tool.
4. Network the two stations via the MPI cable and check the result using the PG function: "Accessible Nodes".
5. Configure an S7 Connection between the two CPUs and download the compiled connection table to the S7-400-CPU.
6. Using the menu options PLC -> Module Information, check whether a connection was actually reserved in the S7-400-CPU (Tab: *Communication -> Reserved Connections*)
7. Carry out a complete restart of the S7-400.
8. Test the connection status with the help of the menu option *PLC -> Activate connection status*.

Note

The S7-300 does not have any configuration data and online data that would give information about reserved and actually used connections.

Exercise 10.2: Communication with the SFBs GET/PUT



SIMATIC S7

Date: 30.06.2011
File: PRO2_09E.34

SITRAIN Training for Automation and Drives

Task

For the S7-400 create an OB1 with the following functionality:

- Via the input 28.0, IB0 and IW 4 of the S7-300 can be read and transferred into QB40 or QW42 of the local S7-400.
- Via the input 28.1, the IW30 of the S7-400 can be written to QW12 in the S7-300.

What to Do

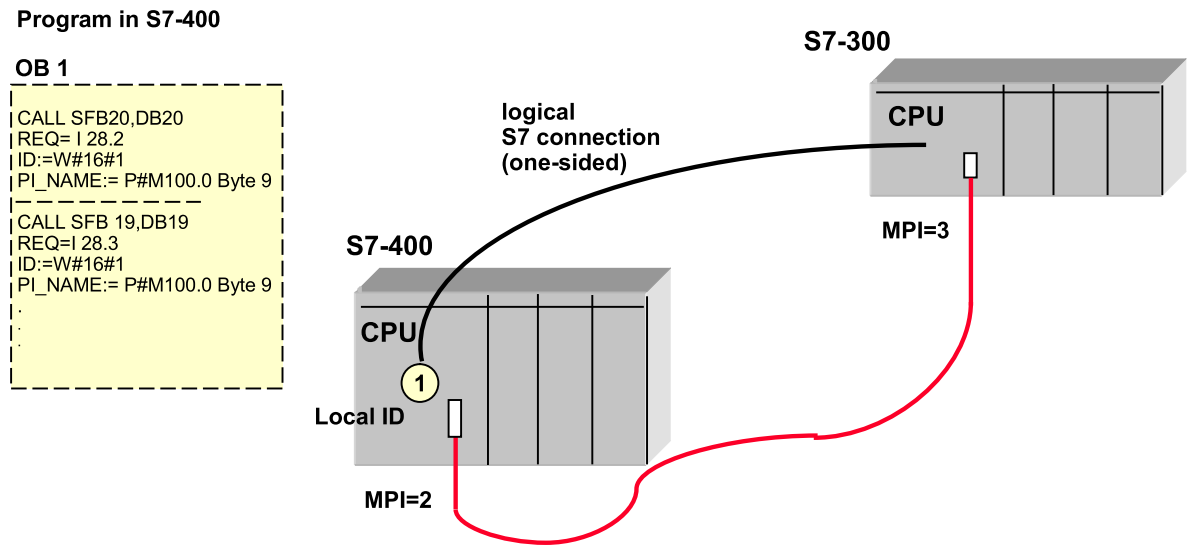
1. Generate an S7 Program folder with the name: SFB_GET_PUT.
2. Edit the OB1. Generate a network “SFB_GET”, in which you call the SFB “GET” (Trigger I 28.0).

   In the “GET” call read out the contents of IB0 from the S7-300 and output the value to the QB40 of the S7-400.

   Also read the contents of IW4 and output this to the QW42 of the S7-400.
3. Generate a network “SFB_PUT” and call the SFB “PUT” (Trigger I 28.1).

   In the “PUT” call transfer the IW30 of the S7-400 to the QW12 of theS7-300.
4. Transfer the output parameter STATUS (pulse) of the SFBs to the digital display (QW38) of the S7-400.
5. Download the OB1 to the S7-400 CPU and test your program.

Exercise 10.3: Communication with the SFBs START/STOP



SIMATIC S7

Date: 30.06.2011
File: PRO2_09E.35

SITRAIN Training for Automation and Drives

Task

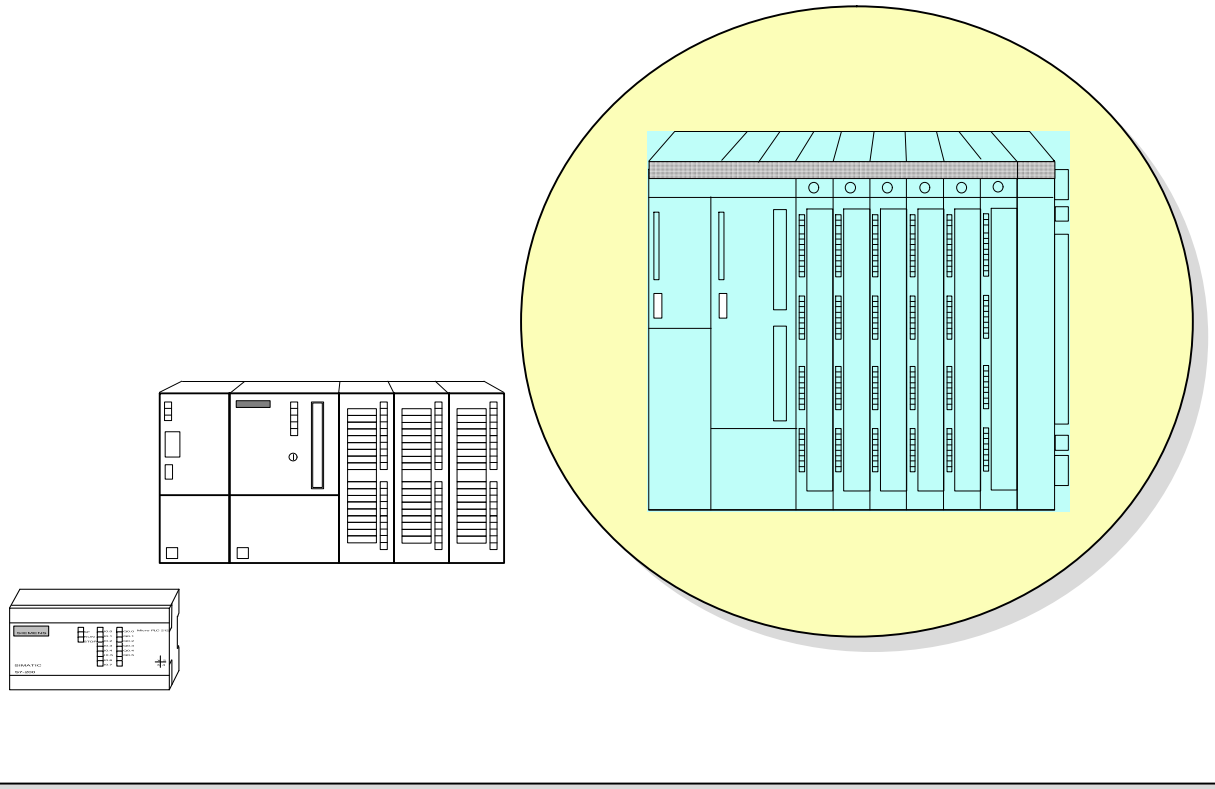
For the S7-400, create an OB1 with the following functionality:

- The target partner (S7-300) can be "stopped" via the input 28.2
- The target partner can be "started" via the input 28.3.

What to Do

1. Generate an S7 Program folder with the name: SFB_START_STOP”
2. Edit the OB1. Generate a network "P_PROGRAM" , in which you store the characters "P_PROGRAM" from MB100 to MB108.
3. Generate a network “SFB_STOP”, in which you call the SFB “STOP” (Trigger 28.2).
4. Generate a network “SFB_START”, in which you call the SFB “START” (Trigger 28.3).
5. In their own networks, transfer the output parameter STATUS (pulse) of the SFBs to the digital display (QW38) of S7-400
6. Download the OB1 to the S7-400-CPU and test your program.

SIMATIC S7-400



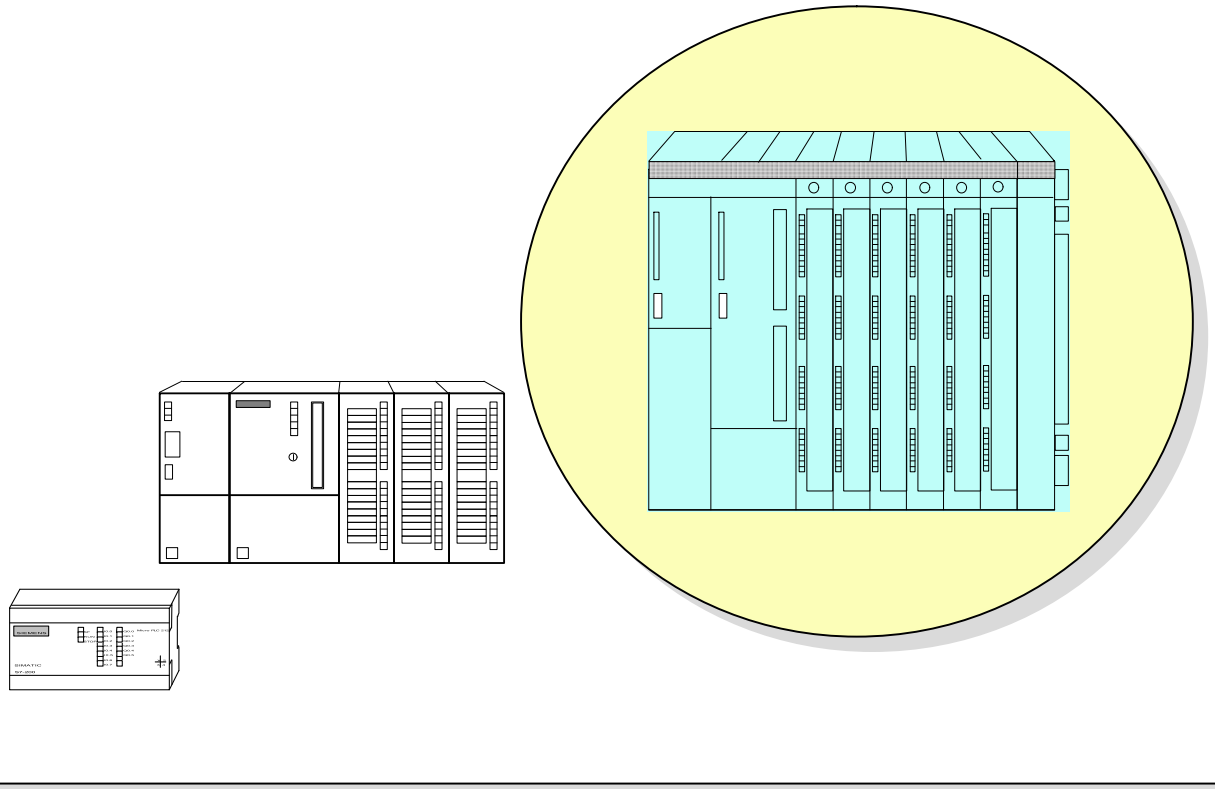
SIMATIC S7

Date: 30.06.2011
File: PRO2_10E.1

SITRAIN Training for Automation and Drives

Contents **Page**

SIMATIC S7-400



SIMATIC S7

Date: 30.06.2011
File: PRO2_10E.2

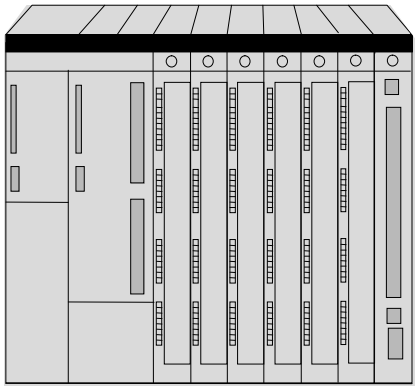
SITRAIN Training for Automation and Drives

Contents **Page**

SIMATIC S7-400: Overview

System expandability

- **high packing density**
- **graduated CPU performance**
- **multicomputing**
- **21 expansion racks can be connected**
- **wide range of modules (SM, FM, CP)**
- **flexible networking facilities**



Performance

- **high processing power (up to 80 nsec per binaray instruction)**
- **up to 16 Mbytes of user memory**
- **powerful communication facilities**

Versatile

- **through special functions**
- **special migration facilities for S5**

SIMATIC S7-400

The SIMATIC S7-400 is the PLC for the mid to upper performance range. The modular and fanless design, the high expandability and ruggedness, the extensive communications facilities and the high performance make it the solution for demanding projects.

Expandability

The special points of the S7-400 are:

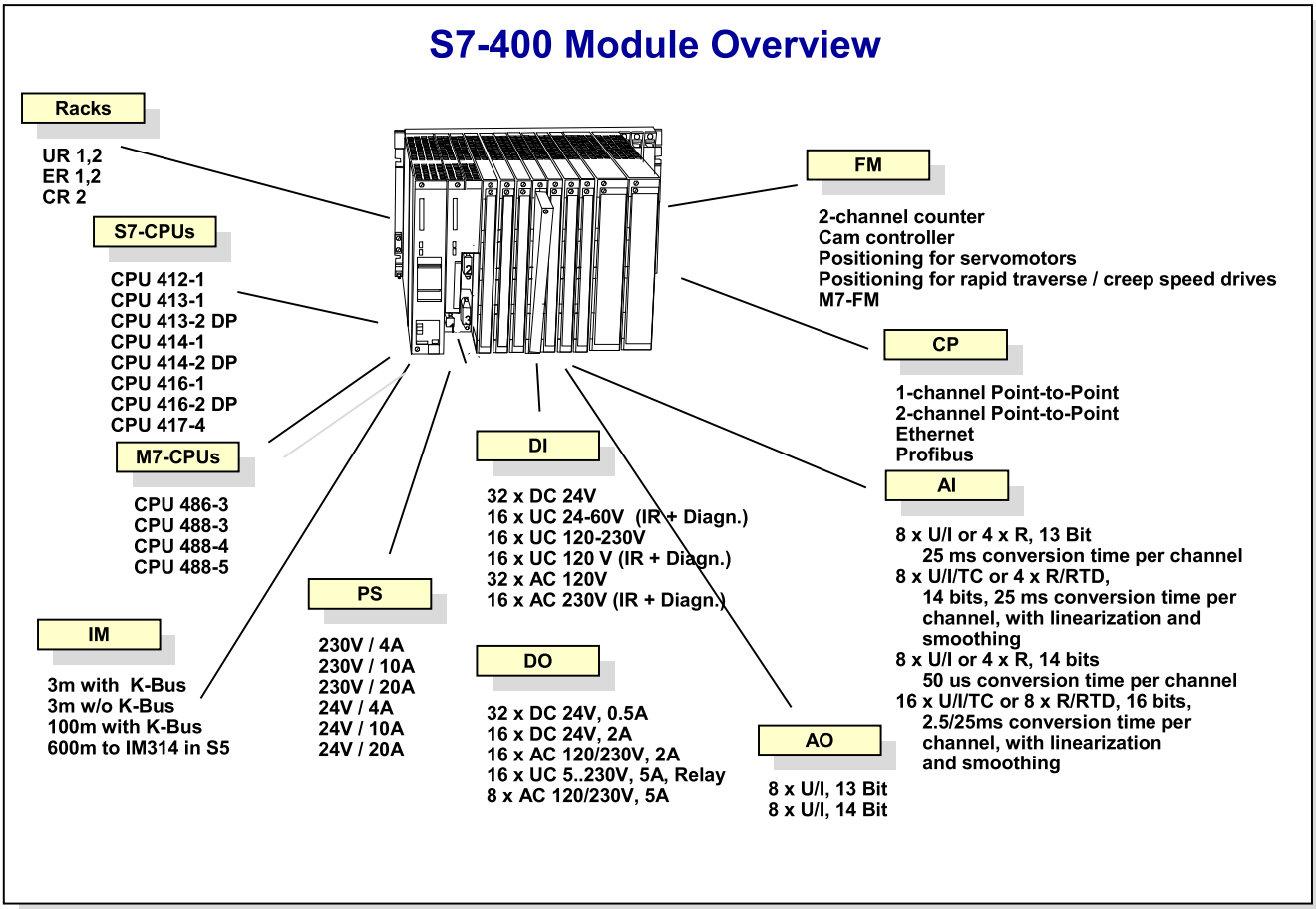
- Simple module mounting in swing-out technique. All modules are of fanless operation and offer a high packing density. Compared to the S5, the mounting space is reduced by 54%, the space per I/O connection is reduced by 45%.
- S7-400 offers a scalable performance through a graduated spectrum of available CPU modules, as well as through symmetrical and asymmetrical multicomputing capability.
- A wide variety of modules, that is, for every application a suitable CPU, signal modules, function modules and communications modules.
- Open-ended with up to 21 expansion racks and additional distributed units via PROFIBUS-DP
- Through the networking facilities using MPI, PROFIBUS and Industrial Ethernet, the S7-400 is also suited for process control tasks.

Performance

- A high processing speed with as little as 80 ns per instruction and a user memory of up to 16 Mbyte make the realization of extensive automation tasks possible.
- The high performance on the communications bus (10.5 Mbaud) guarantees fast communication with high data throughput.

Versatile

- Versatile through special functions, such as: restart, removing and inserting modules in RUN, etc.
- In addition, S7-400 offers special migration facilities from S5 to S7, such as:
  - using S5 IPs or WFs in S7 central racks
  - connecting S5 expansion units to an S7 central rack



SIMATIC S7

Date: 30.06.2011
File: PRO2_10E.4

SITRAIN Training for Automation and Drives

Racks

The following racks are available for the S7-400.

- UR1/UR2 is designed as a universal rack and can be used as a central rack or as an expansion rack. It features 18/9 single-width slots with both P and K bus.
- ER1/ER2 are expansion racks without a K bus.
- CR2 is is a segmented central rack for symmetrical multicomputing.

S7 CPUs:

The S7-400 CPUs are upward compatible for all STEP 7 user programs. They are available in two versions: single-width and double-width with integrated DP Master interface.

A maximum of 64 DP slave stations can be addressed via the integrated DP interface. The maximum baud rate is 12 Mbaud.

FMs

The FMs for positioning, closed-loop control and counting replace the S5 IP spectrum. In addition, an M7 FM can be inserted as a freely C-programmable function module for process control.

IMs

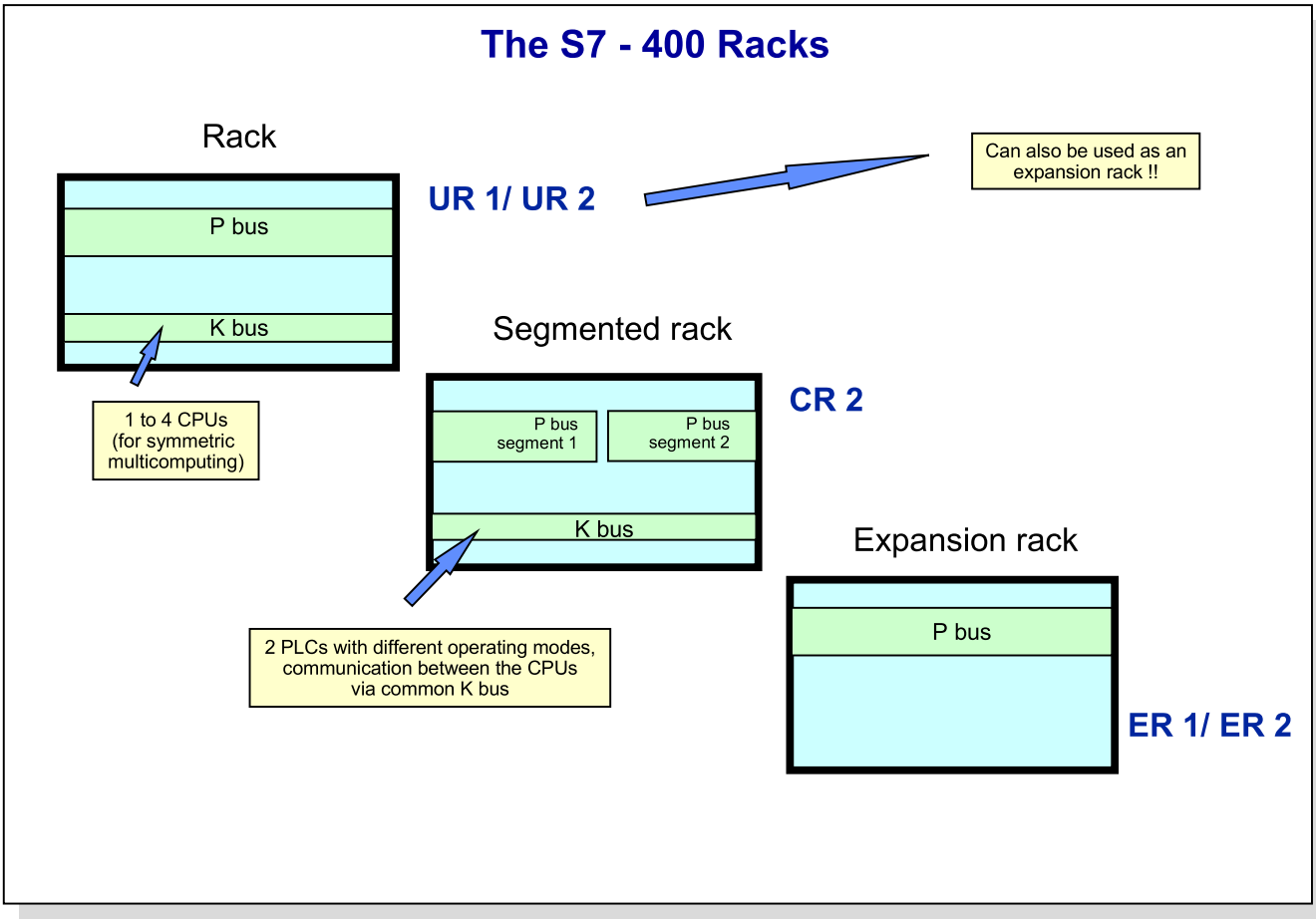
SIMATIC S7 and SIMATIC S5 expansion racks can be connected to an S7-400 central rack via interface modules.

CPs

CP modules make the connection of a CPU to the following networks possible:

- Industrial Ethernet (CP 443-1)
- PROFIBUS (CP 443-5)
- Point-to-Point Network (CP441-1 and CP441-2)

Moreover, every CPU features an MPI interface for connection to an MPI network. A maximum of 32 stations can be connected to an MPI network.



UR 1 / UR 2

UR1/UR2 can be used both as central and as expansion racks. They have a parallel I/O bus (P bus) for the high-speed exchange of I/O signals (1.5 microsec./ byte) and time critical access to the signal module process data.

In addition, UR1 (18 slots) / UR2 (9 slots) have a serial, powerful communication bus (K bus) for high-speed data exchange (10.5 Mbaud) between K bus stations (S7/M7 CPUs, FMs, CPs,).

By separating the P BUS and K BUS, each task is assigned its own bus system. Control and communication have their own separate data highways, thus providing smooth and conflict-free control and communication operations.

CR2

The segmented rack CR 2 features an I/O bus divided into two segments with 10 and 8 slots. One CPU can be used in each segment. Each CPU is master in its respective P bus segment and can only access its own SMs.

Operating mode transitions are not synchronized. That is, the CPUs can be in different operating modes. Both CPUs can communicate via the continuous K bus.

ER 1 / ER 2

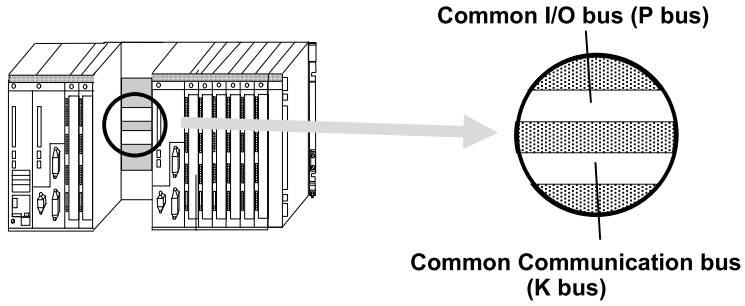
ER1 (18 slots)/ER2 (9 slots) have no K bus, no interrupt lines, no 24 V power supply for the modules and no battery supply.

No Slot Rules

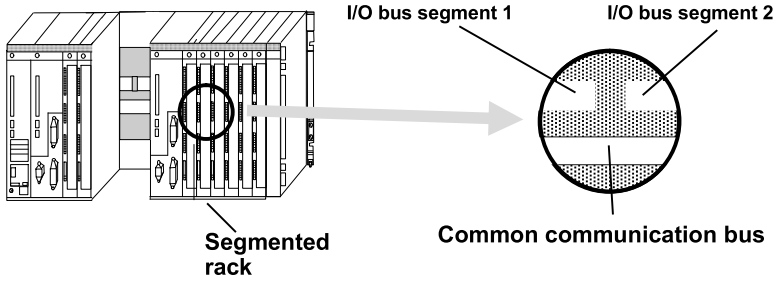
Exception: PS on the far left and Receive IM in the ER on the far right.

Symmetrical and Asymmetrical Multicomputing

Symmetrical Multicomputing



Asymmetrical Multicomputing



SIMATIC S7

Date: 30.06.2011
File: PRO2_10E.6

SITRAIN Training for Automation and Drives

Multicomputing

Multicomputing makes a PLC system scalable. That is, it enables the system performance and system resources (memory, bit memories, timers, counters, etc.) to be increased. That way, a complex task can be divided amongst several CPUs, for example.

Symmetrical

In symmetrical multicomputing, all CPUs (max. 4 CPUs) share a common P bus and K bus. In particular, only one common I/O address space exists in which the addresses of all signal modules are mapped.

Every module inserted must however be assigned to a CPU during configuration. The CPU then takes over the “Master Function” for this module, such as:

- receipt of module interrupts
- module parameter assignment
- access of the modules via L PBxx, T Wxx, etc.

The operating mode transitions are synchronized. That is, all CPUs have the same operating mode. As seen from the outside, the station appears to be one large PLC.

Asymmetrical

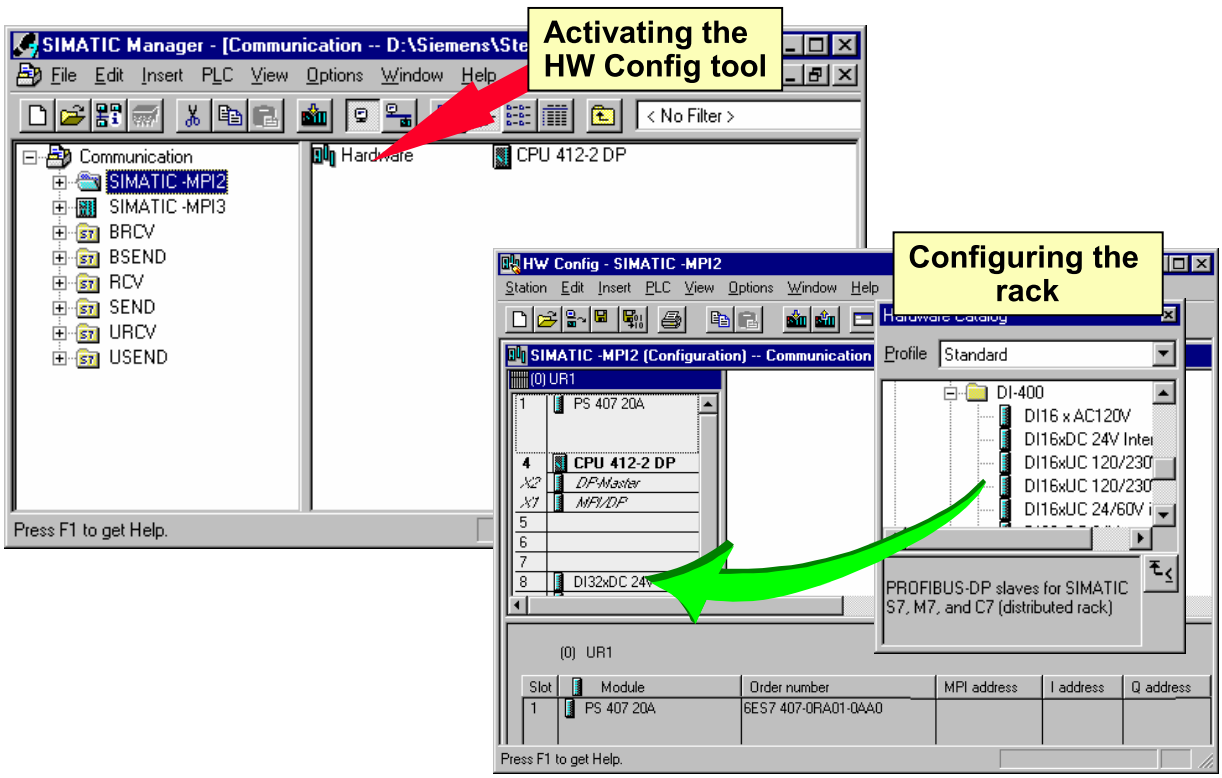
Asymmetrical multicomputing is achieved with the help of the CR2. The segmented rack contains two separate P bus segments.

One CPU is installed per I/O bus segment. The I/O modules are locally assigned to this CPU. The CPUs work independently of one another without synchronization of operating mode transitions. Every CPU has its own I/O address space.

The common communication bus makes communication between the two units possible without the use of additional hardware. As seen from the outside, this corresponds to two individual controllers that communicate via the K bus. Further benefits are:

- space saving in the control cabinet
- cost saving, since only one rack and one power supply are required.

Centralized Configuration



Configuring a Centralized Station

For a centralized configuration, arrange the modules next to the CPU(s) on a central rack and continue on with further expansion racks.

Creating the Configuration

To open the configuration of a station proceed as follows:

1. First of all, select per mouse click the desired hardware station.
2. Select the menu option *Edit -> Object*, open or double-click the *Hardware* symbol in the right hand window. The station window of the selected station is opened.
3. A click on the Catalog symbol displays the HW Catalog with the current components. From this catalog copy per "drag and drop" the rack and the modules into the station window or into the configuration table of the respective rack.

Catalog View

A click on the "+" sign opens the associated substructure or by clicking the "-" sign it is closed. After selecting a module, the most important technical data of the module appears in the status bar of the catalog window.

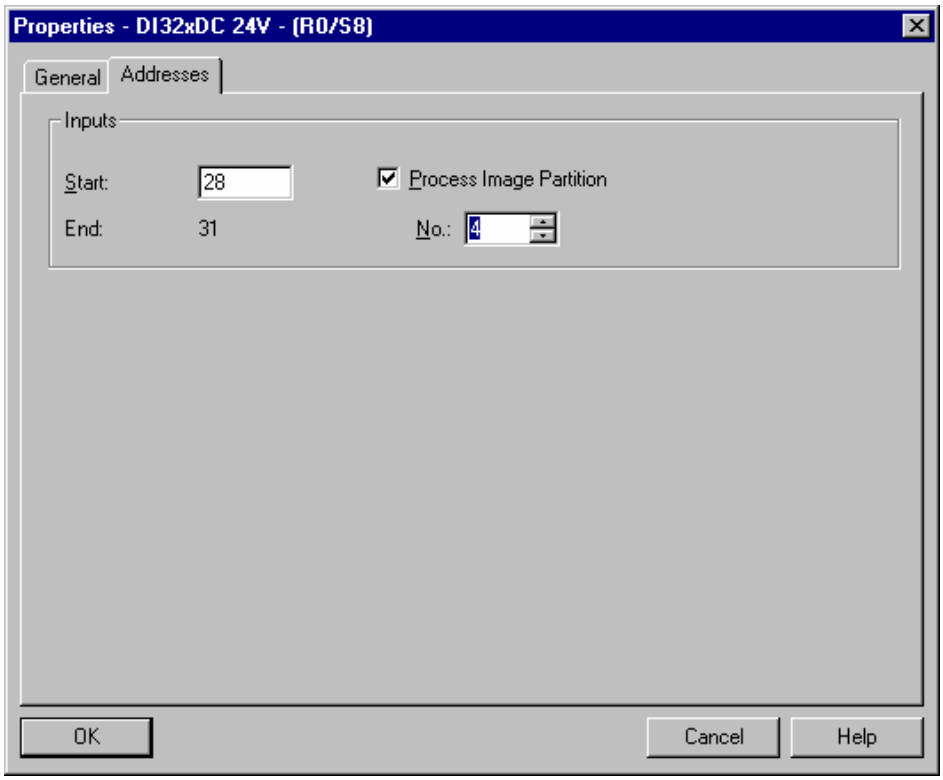
Selecting the Rack

When you have opened the station window and the hardware catalog, you can continue as follows

1. Depending on the station type, select the entry S7-400.
2. First of all, open the subentry RACK and drag the racks into the station window on the left. An empty table is displayed for every rack.

   Racks are represented by configuration tables in STEP 7. These configuration tables have a number of entry lines equal to the number of modules that can be installed in the rack.
3. Then, using "drag and drop" copy the desired modules to the left into the empty slots in the table.

Module Parameters: Logical Addresses, Part Process Images



General

In the S7 -400 system there are default addresses for the I/O modules. These defaults are active until parameter data is downloaded into the CPU.

Default Addresses

The module default addresses depend on:

- The number of the rack. The number is set on the Receive IM with a switch (1..21), the central rack always has the No. 0
- The module's slot in the rack. The default address of a module is calculated from these two values as follows:

Digital start address = [(rack number) x 18 + slot number -1] x 4

Analog start address= [(rack number) x 18 + slot number -1] x 64 + 512

Process Image Partition

Besides the complete process image, you can assign parameters for up to 8 process image partitions. The process image partitions can be updated in the user program using system functions (SFC 26/27).

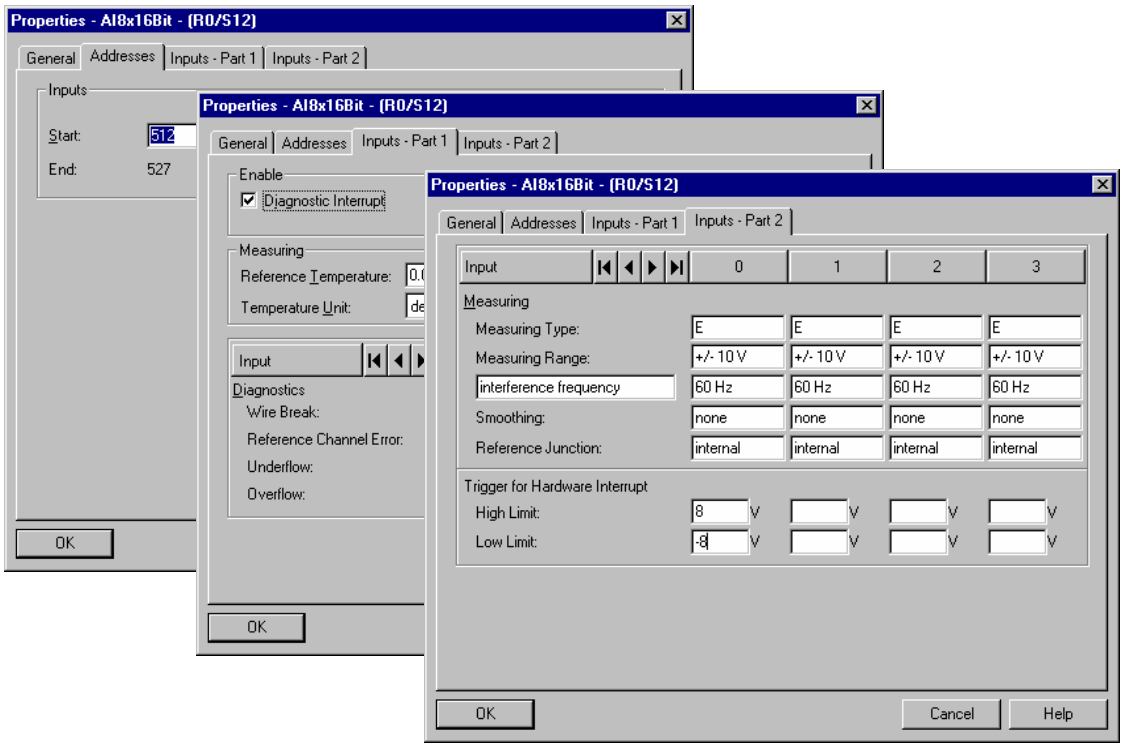
That way, the user can use the process image concept that was originally only designed for the OB1 priority class for other priority classes as well, for example OB35 for control algorithms.

The input as well as the output modules which deal with the actual or setpoint values of the current control algorithm are each assigned to a part process-image input or output table.

Within OB35 you can then proceed with the following sequence:

1. Reading in the current actual values by means of SFC26 into the associated part process-image input table.
2. Calling the control algorithm. The control algorithm writes its new setpoint values into the corresponding part process-image output table.
3. Output of the part process-image output table by means of SFC27 to the I/O.

Module Parameter Assignment: Analog Module



Date: 30.06.2011
File: PRO2_10E.9

Module Parameter Assignment

All parameter-assignable modules, for example analog modules, can be assigned parameters with the HW Config tool.

For analog modules, there are usually several tab pages for parameter assignment. Thus, the following parameters can be set, for example on the individual tab pages of an S7-400 analog module.

Addresses

- Start address of module
- Number of process image partition
- Hardware interrupt OB

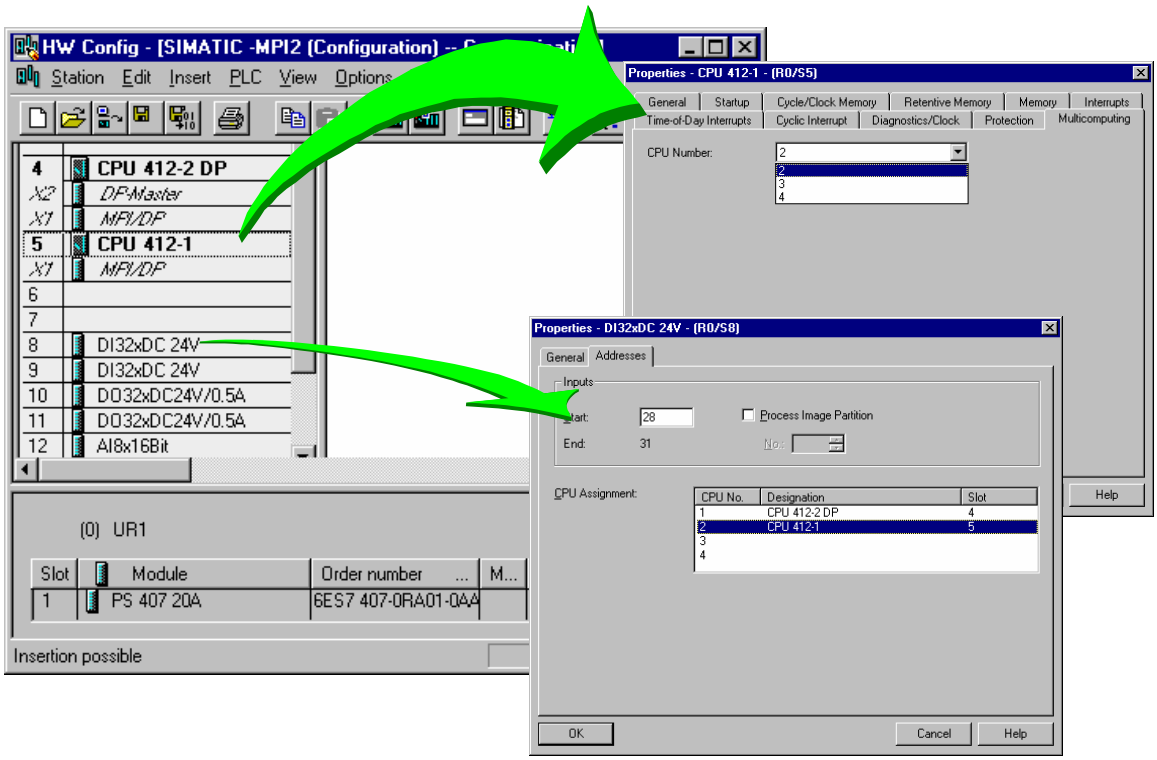
Inputs - Part 1

- Enable for hardware interrupt and diagnostic interrupt
- Enable for channel-specific diagnostic monitoring, like
  - Wire Break check
  - Reference Channel Error
  - Underflow
  - Overflow
  - Short circuit to ground

Inputs - Part 2

- Measuring Type
- Measuring Range
- Interference frequency suppression
- Smoothing
- Cycle end interrupt enable
- Upper and lower limit value for hardware interrupt

Configuring Multicomputing



Overview

Multicomputing is the synchronous operation of several CPUs (2 to 4) in the S7-400 central racks.

The CPUs start up together if they have the same startup mode (complete restart or restart) and they also go into the STOP mode together.

Setting Up Multicomputing

Multicomputing emerges implicitly through the insertion of several multicomputing-capable CPUs in a suitable rack. Whether a CPU is multicomputing-capable can be determined in the infotext in the "Hardware Catalog".

A common address area is divided between the CPUs participating in multicomputing. That is, the address area of a module is always exclusively associated with one CPU.

Procedure

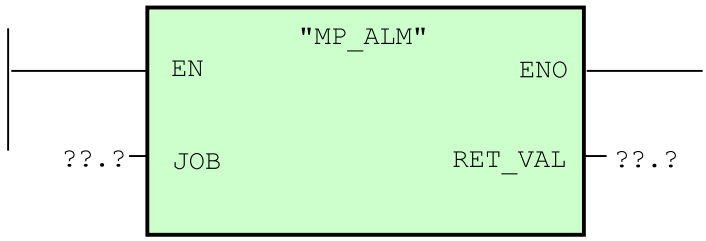
Multicomputing is configured as follows:

1. Position all the CPUs required for multicomputing.
2. Double-click the CPUs and set the CPU number on the “Multicomputing" tab page.
3. In order to assign a module to a particular CPU, proceed as follows:
   - Arrange the modules in the rack.
   - Double-click the modules and select the "Addresses" tab.
   - In the “CPU No.” field select the number of the desired CPU. With interrupt-triggering modules, the CPU assignment is displayed as the target CPU on the "Inputs" or "Outputs" page.

Via the command *View -> Filter -> CPUx Modules,* you can make the modules that are assigned to a particular CPU stand out optically in the table.

The station configuration can only be downloaded into all CPUs. Downloading to one CPU only is not possible. That way, inconsistent configurations are avoided.

SFC 35 for Synchronization in Multicomputing Mode



Parameter	Declaration	Data type	Memory area	Description
JOB	INPUT	BYTE	I, Q, M, D, L, Const.	Job identifier (possible values: 1 to 15)
SFB 9	URCV	Two-sided	Two-sided	Two-sided

Description

The call of SFC 35 ”MP_ALM” triggers the multicomputing interrupt in the Multicomputing mode. This leads to the synchronized start of OB60 on all associated CPUs.

In single processor operation and operation in segmented racks, OB60 is only started on the CPU on which SFC 35 is called.

With the input parameter JOB, you can identify the cause for the multicomputing interrupt. This job identifier is transmitted to all CPUs and can be evaluated via the start information in OB 60.

You can call SFC 35 ”MP_ALM” from any location in your program. Since the call only makes sense in the RUN mode, the multicomputing interrupt is suppressed when called in the STARTUP mode. This is communicated to you via the return value.

Error Code

If an error occurs while the function is being processed, the return value contains an error code:

W#16#0000: No error.

W#16#8090: Input parameter JOB contains an illegal value.

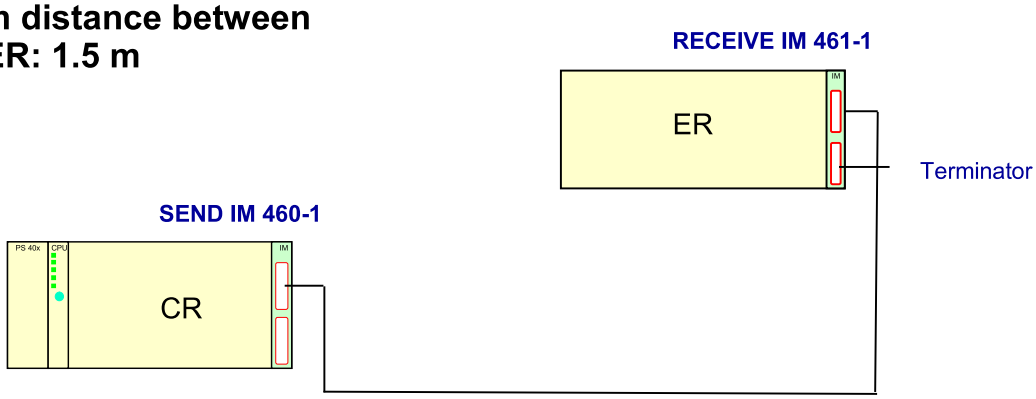
W#16#80A0: Processing of the previous multicomputing interrupt with OB 60 has not yet been completed on the local CPU or on another CPU.

W#16#80A1: Incorrect operating mode (STARTUP instead of RUN).

Centralized Expansion 1

Characteristics

- **P bus and power supply connected through, but not the K bus**
- **1 ER per chain**
- **Maximum distance between CR and ER: 1.5 m**



Centralized Config. 1 The interface modules IM 460-1/IM 461-1 are inserted for the centralized connection of expansion racks to an S7 central rack.

Centralized expansion type 1 has the following characteristics:

- A maximum of 2 expansion racks can be connected (1 per interface).
- The maximum distance between central and expansion rack is 1.5 m.
- A maximum of 2 Send IM 460-1 can be inserted per central rack.
- The Send interface module IM 460-1 connects only the P bus (not the K bus) through to the expansion rack. Moreover, the modules in the expansion rack are supplied with a 5V voltage (max. 5 A per slot) via the connecting cable.

  For this reason, no extra power supply module can be inserted in the expansion rack.
- An unoccupied interface in the Send IM 460-1 does not have to be terminated; an unoccupied interface in the Receive IM 461-1 must be terminated with a terminator.
- A coding switch with which the number of the expansion rack must be selected is located on the Receive IM 461-1.

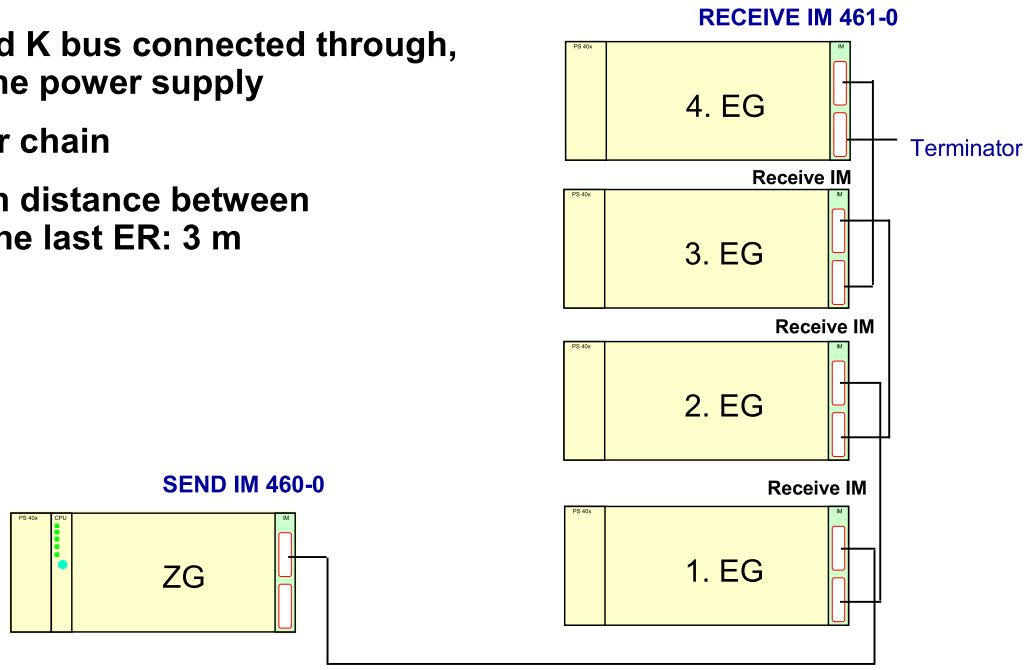
Note A maximum of 21 expansion racks can be connected to one central rack.

The Receive IM must always be inserted in the furthest right slot in the expansion rack.

Centralized Expansion 2

Characteristics

- **P bus and K bus connected through, but not the power supply**
- **4 ERs per chain**
- **Maximum distance between CR and the last ER: 3 m**



SIMATIC S7

Date: 30.06.2011
File: PRO2_10E.13

SITRAIN Training for Automation and Drives

Centralized Config. 2 The interface modules IM 460-0/IM 461-0 are inserted for the centralized connection of expansion racks to an S7 central rack. Centralized expansion type 2 has the following characteristics:

- A maximum of 8 expansion racks can be connected (4 per interface).
- The maximum distance between the central rack and the furthest expansion rack is 3 m.
- A maximum of 6 Send IM 460-0 can be inserted per central rack.
- The Send interface module IM 460-0 connects the P bus and the K bus through to the expansion rack. The modules in the expansion rack are not supplied with voltage via the connecting cable.

  For this reason, each expansion rack must have its own power supply module inserted.
- An unoccupied interface in the Send IM 460-0 does not have to be terminated; an unoccupied interface in the Receive IM 461-0 must be terminated with a terminator.
- A coding switch with which the number of the expansion rack must be selected is located on the Receive IM 461-0.

Note

A maximum of 21 expansion racks can be connected to one central rack.

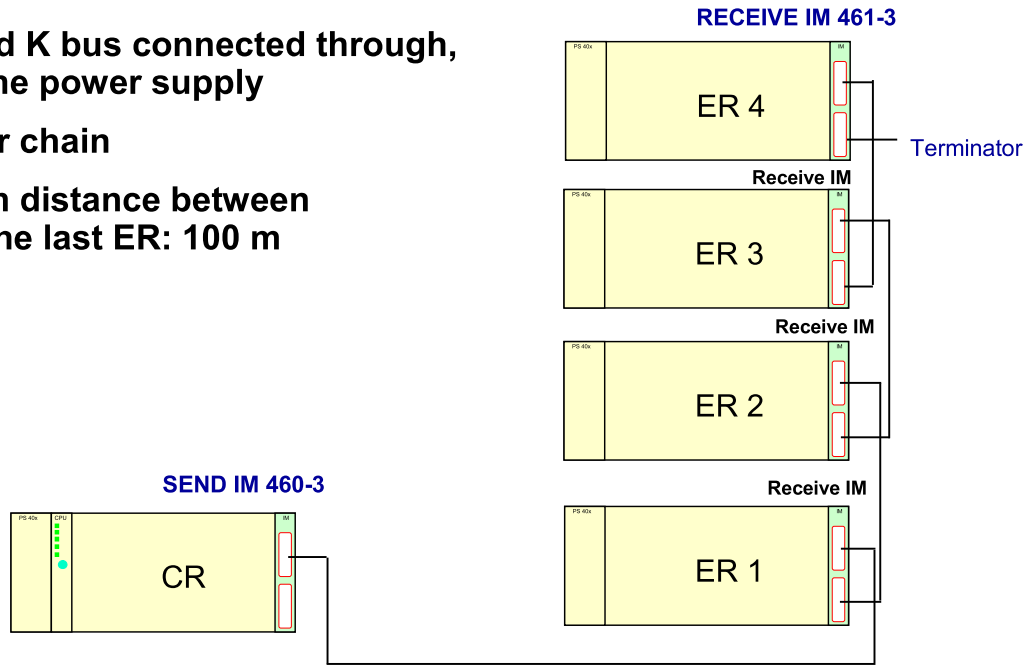
The K bus is only connected through to the first 6 expansion racks. That means that intelligent modules, such as FMs and CPs, can therefore only be operated in the first 6 ERs.

The Receive IM must always be inserted in the furthest right slot in the expansion rack.

Distributed Expansion

Characteristics

- **P bus and K bus connected through, but not the power supply**
- **4 ERs per chain**
- **Maximum distance between CR and the last ER: 100 m**



SIMATIC S7

Date: 30.06.2011
File: PRO2_10E.14

SITRAIN Training for Automation and Drives

Distributed Config.

The interface modules IM 460-3/IM 461-3 are inserted for the distributed connection of expansion racks to an S7 central rack. Distributed expansion has the following characteristics:

- A maximum of 8 expansion racks can be connected (4 per interface).
- The maximum distance between the central rack and the furthest expansion rack is 100 m.
- A maximum of 6 Send IM 460-3 can be inserted per central rack.
- The Send interface module IM 460-3 connects the P bus and the K bus through to the expansion rack. The modules in the expansion rack are not supplied with voltage via the connecting cable.

  For this reason, each expansion rack must have its own power supply module inserted.
- An unoccupied interface in the Send IM 460-3 does not have to be terminated; an unoccupied interface in the Receive IM 461-3 must be terminated with a terminator.
- A coding switch with which the number of the expansion rack must be selected is located on the Receive IM 461-3.

Note

A maximum of 21 expansion racks can be connected to one central rack.

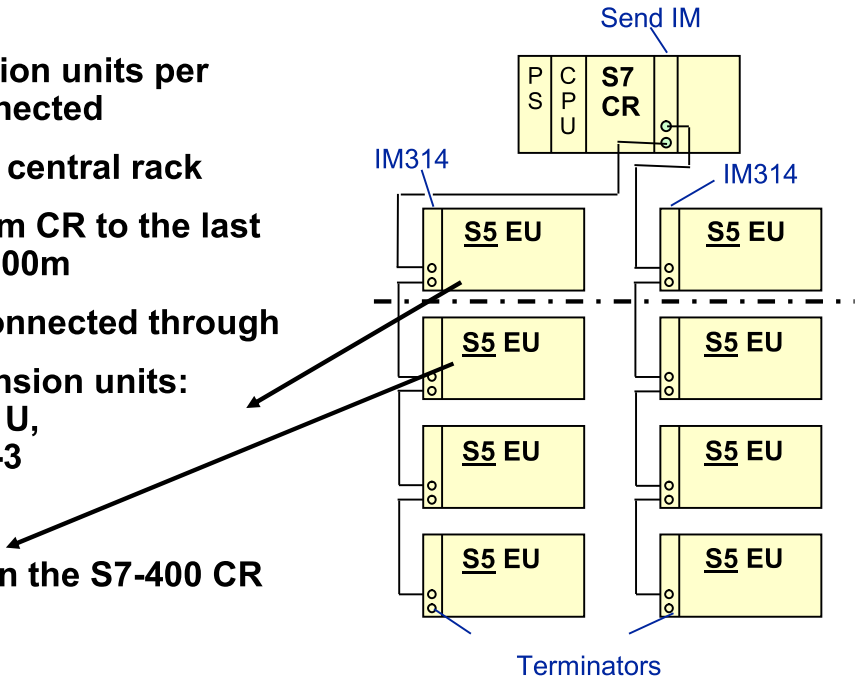
The K bus is only connected through to the first 6 expansion racks. That means that intelligent modules, such as FMs and CPs, can therefore only be operated in the first 6 ERs.

The Receive IM must always be inserted in the furthest right slot in the expansion rack.

Distributed Connection between S7 and S5

Characteristics

- **max. 4 S5 expansion units per chain can be connected**
- **max. 4 send IM in central rack**
- **max. distance from CR to the last EU in the chain: 600m**
- **parallel S5 bus connected through**
- **possible S5 expansion units: EU 183 U, EU 185 U, ER 701-2, ER 701-3**
- **other S5 EUs**
- **max. 32 S5 EUs on the S7-400 CR**



SIMATIC S7

Date: 30.06.2011
File: PRO2_10E.15

SITRAIN Training for Automation and Drives

Distributed Config. with S5 EUs

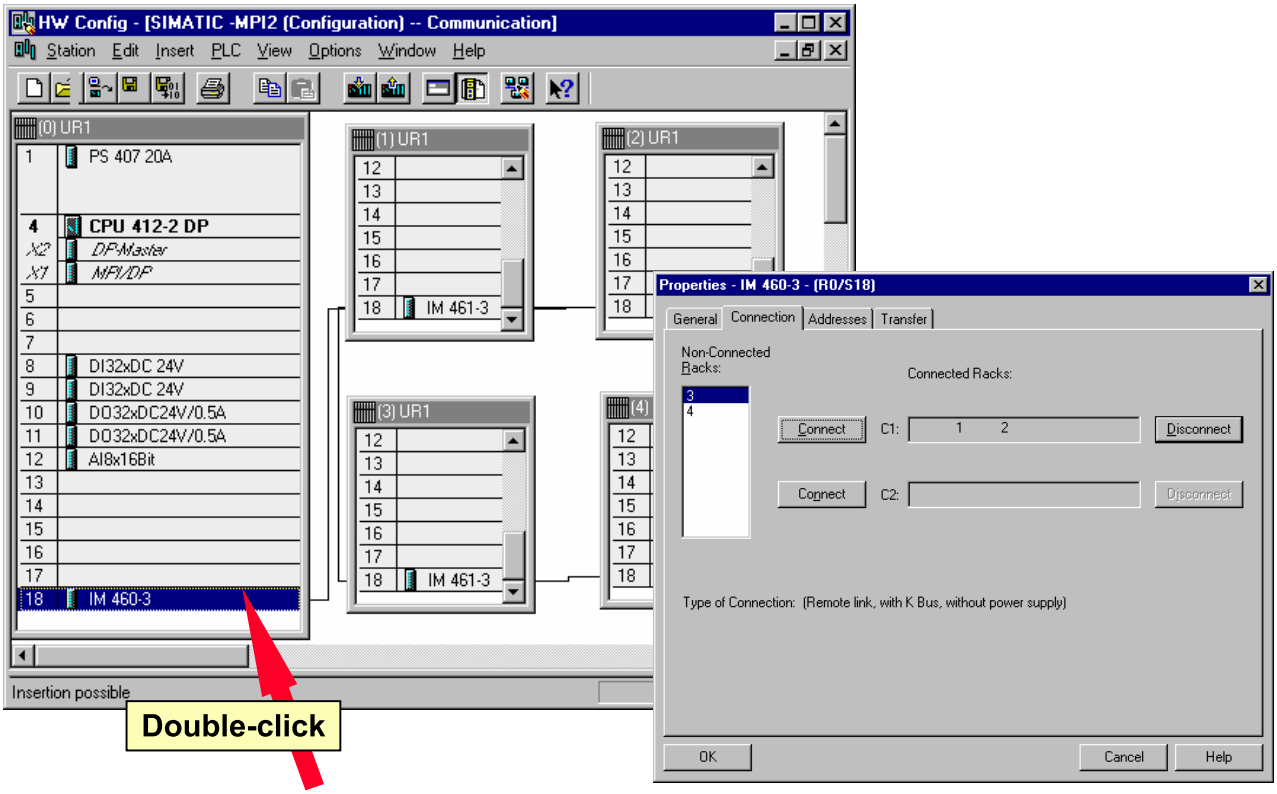
The IM463-2 interface module enables S5 expansion units to be connected to an S7-400 central rack.

The following rules apply to connections with S5 expansion units:

- Max. 4 IM463-2 interface modules in the central rack
- Max. 4 expansion units per chain
- Max. distance between the CR and the last EU 600 m
- Older S5 systems can be connected to an S7-400 central rack if the first S5 EU is an EU-183U/EU-185U for the S5-135U/-155U or an ER-701-2/ ER-701-3 in the case of an S5-115U.

The remaining EUs can be expanded in accordance with the S5 rules.

Expanding a Centralized Configuration



Expanding the Configuration

If you want to “connect” further racks to a central rack and “equip” them, then proceed as follows:

1. From the hardware catalog select the desired (expansion) rack.
2. Drag the racks one after the other into the station window using Drag&Drop.
3. Insert the desired modules into the racks.

   Important: The Receive interface modules must be inserted in all expansion racks before a connection to the Send interface module in the central rack is possible.
4. Only for S7-400: In order to establish a connection between the Send IM and the Receive IMs of the expansion racks, proceed as follows:
   - Double-click on the Send IM
   - Select the "Connection" tab. All racks that are not connected are displayed in a list.
   - Select each rack in turn and using the command button “Connect” connect them to the desired Send IM interface (C1 and C2).

   Subsequently, connection lines between the racks are displayed in the station window.

CR2 Peculiarities

With the CR2 central rack, you must first make the connection between the empty (except for the Receive IM) racks to the Send IMs of the respective CPU, before you can insert modules into an expansion rack.

CPU Modules – Graduated Performance Spectrum

Characteristics

- **Graduated performance range**
- **LEDs for displaying status and errors**
- **Key-operated switch for selecting the operating modes**
- **Memory Card slot**
- **MPI interface (and PROFIBUS-DP)**
- **External battery backup supply**
- **Integrated test and diagnostic functions**

CPU 412-1

CPU 412-2
CPU 414-2
CPU 416-2

CPU 414-3
CPU 416-3

CPU 417-4

SIMATIC S7

Date: 30.06.2011
File: PRO2_10E.17

SITRAIN Training for Automation and Drives

Overview

SIMATIC S7-400 has 7 different CPUs and an additional CPU for S7-400H with graduated performance spectrums for the varying applications.

Area of Use

The following CPUs are available for the SIMATIC S7-400:

- CPU 412-1 CPU 412-2, for smaller systems in the mid performance range
- CPU 414-2 and CPU 414-3, for mid-sized systems with additional requirements in program scope, processing speed and communication
- CPU 416-2 and CPU 416-3, for systems with the greater requirements of the upper performance range
- CPU 417-4 DP for systems with the greatest requirements of the upper performance range
- CPU 417H for the SIMATIC S7-400H

As well, the SIMATIC M7 CPUs can be used for S7-400.

Design

All central processing units have a plastic casing, in which the operating and indicating elements are integrated. The indicating and operating elements are identical in all CPUs.

On the front you will find:

- LEDs; for displaying status and errors
- Key-operated switch; for selecting the operating mode
- Memory Card slot
- MPI interface for communication
- Integrated PROFIBUS-DP interface (CPU 4xx-2,3)
- Battery backup receptacle; for external battery backup supply

The CPUs – Performance Made to Measure

	CPU 412-1 (CPU 412-2)	CPU 414-2 (CPU 414-3)	CPU 416-2 (CPU 416-3)	CPU 417-4
Instructions	16/(24) K	42/(128) K	266/533 K	666 to 2666 K
Program memory	48/(72) KByte	128/(384) KB	0.8/1.6 MB	2 to 8 MB
Data memory	48/(72) KByte	128/(384) KB	0.8/1.6 MB	2 to 8 MB
DI / DO	32(32) / 32(32) K	64(64)/64(64) K	128(128)/128(128)K	128/128 K
AI / AO	2(2) / 2(2) K	4(4)/4(4)	8(8)/8(8) K	8/8 K
Execution time/ 1 K Binary instruc.	0.2 ms	0.1 ms	0.08 ms	0.1 ms
Bit memories	4 KByte	8 KByte	16 KByte	16 K
Counters	256	256	512	512
Timers	256	256	512	512
FB/FC/DB	256/256/511	1024/1024/1023	2048/2048/4095	6144/6144/8192
Communications interfaces	MPI or DP Master, 32 DP Slaves (PROFIBUS DP Master, 125 DP Slaves)	MPI or DP Master, 32 DP Slaves PROFIBUS DP Master, 125 DP Slaves (IF 964-DP Master)	MPI or DP Master, 32 DP Slaves PROFIBUS DP Master, 125 DP Slaves (IF 964-DP Master)	MPI or DP Master, 32 DP Slaves PROFIBUS DP Master, 125 DP Slaves 2 x IF 964-DP Master

Functions

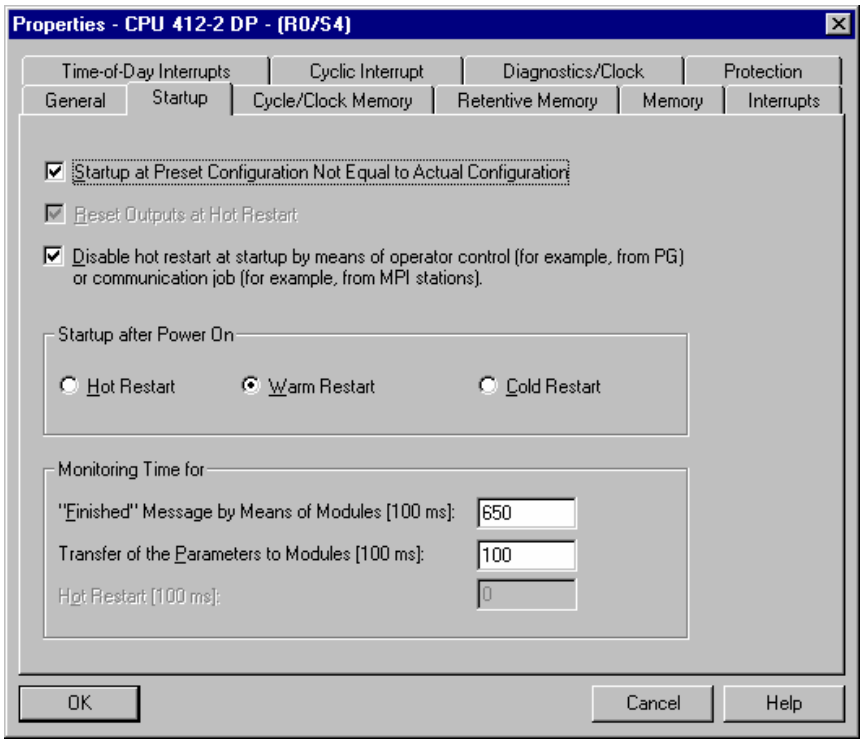
All CPUs have:

- a powerful processor; the CPUs reach execution times down to 0.08 µs per binary instruction (CPU 416)
- fast work memory; for storing those program sections of the user program that are relevant to execution
- MPI; with MPI you can design a simple network with up to 32 nodes and a data transmission speed of up to 12 Mbits/second.

  The CPUs can set up up to 32 connections to nodes of the communications bus (K-Bus) and MPI
- block protection; in addition to the key-operated switch, a password concept protects the user program from unauthorized access
- a diagnostic buffer; the last 120 error and interrupt events are stored in a FIFO (ring buffer) for diagnostic purposes (expandable)
- integrated HMI services; the user only has to specify the source and target of the data in the HMI devices; these are automatically transported cyclically by the system
- a real-time clock; CPU diagnostic messages are given a date and time
- a Memory Card; a Memory Card (RAM or FEPROM) is always necessary for operating the S7-400. It is used as a load memory for the user programs and parameter-assignment data of the S7-400 PLC system.
- PROFIBUS DP interface (except CPU 412-1); The PROFIBUS DP Master interface enables you to have a distributed automation design.

The CPU 4xx-3 have an additional slot, the CPU 417-4 has two. With each slot a further PROFIBUS DP Master chain can be connected using an IF 964 Module.

CPU Parameters: Startup Characteristics



Setpoint/Actual Configuration

By deactivating this field you can, in S7, make it so that the CPU goes into Stop after startup if the actual configuration does not equal the setpoint configuration (according to the configuration).

Hardware Test

By activating this function, the internal RAM on the CPU is tested during startup. The length of the test depends on the size of the memory.

Delete PIQ..

In an S7-400 restart, the process image is deleted by default after the remaining scan cycle is processed. If this characteristic is not desired, you can deselect it.

Disable Restart

Restriction to a Complete Restart for manual startup of the S7-400.

POWER ON

With S7-400 you have the choice between:

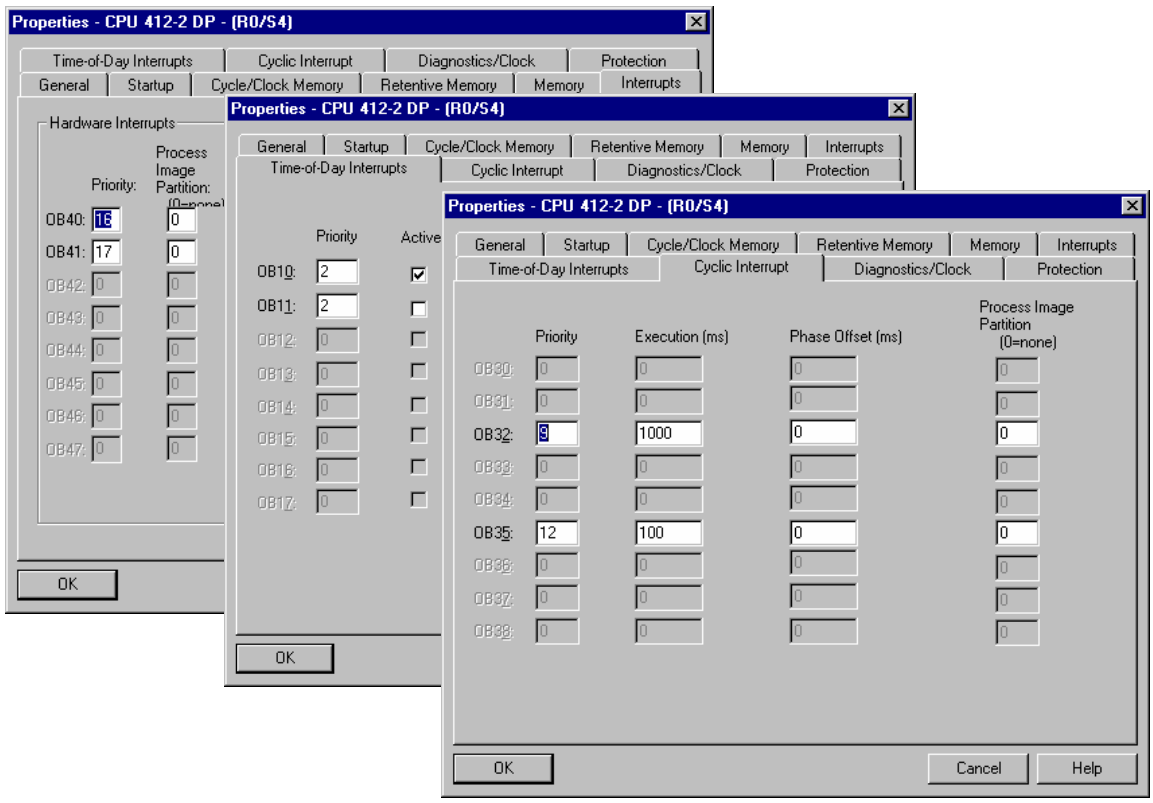
- Complete Restart (deletion of the non-retentive areas and program processing begins with the first instruction in block OB1).
- Restart (all memory areas remain intact and the program continues from the point of interruption).
- Cold restart (deletion of the retentive and non-retentive areas and program processing begins with the first instruction in block OB1).

Monitoring Times

The following times can be specified:

- maximum waiting time until all modules have reported to the CPU.
- maximum time until a module must have acknowledged a parameter transfer.
- for S7-400, the maximum time following a power failure, after which a restart can still be performed.

CPU Parameters: Interrupts



Priorities

On the S7-400 you can change the priorities of the interrupt handling blocks and thus establish the order in which they are processed when several interrupt events occur simultaneously. The priorities are from 1 to 24 and the interrupt with the highest priority is processed first.

Hardware Interrupts

In this parameter block you set the priorities of the hardware interrupt organization blocks. You can set the priorities 0 and from 2 to 24 (0 = deselect).

Time-of-Day Interrupt

You can use these interrupts to set the starting time for executing a program once only or for an activity to be repeated regularly from this time onwards (every minute, hourly, daily, weekly, monthly, yearly).

Cyclic Interrupt

The cyclic interrupt can be used to execute a program part at fixed intervals. You can select intervals from 1 to 60000 ms. This enables you to implement closed-loop control tasks that have to be processed at fixed intervals (sampling time), for example.

On the S7-400 there are eight different cyclic interrupts with different intervals. To prevent all the cyclic interrupts from having to be processed at the same time, you can set the "Phase Offset" so that calling of the cyclic interrupts is staggered.

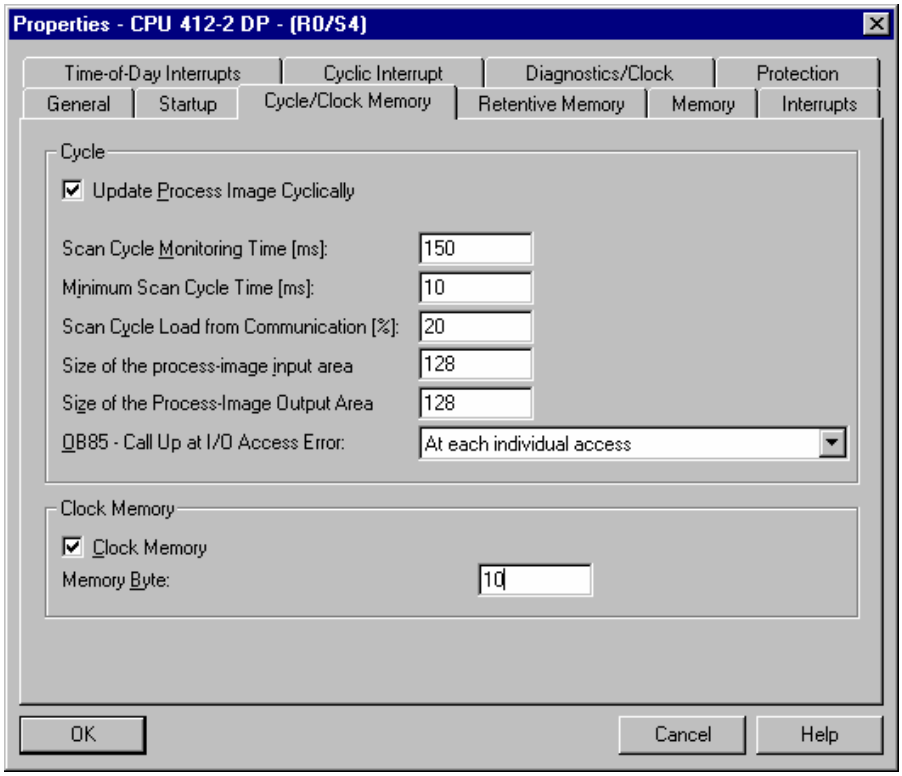
Time-Delay Interrupts

A time-delay interrupt is a one-off call to an organization block which is activated with a time delay after a process signal is received, for example.

The time-delay interrupts are handled in the user program with the aid of SFCs 32 to 34.

- SFC32 "SRT_DINT" = Start time-delay interrupt.
- SFC33 "CAN_DINT" = Cancel time-delay interrupt
- SFC34 "QRY_DINT" = Query status of time-delay interrupt

CPU Parameters: Cycle/Clock Memory



Update Process Image Cyclically

Activate the check box when you want the (OB 1) process image to be updated cyclically. The scan time increases because of the updating.

In the S7-300, the process image is always updated cyclically.

Scan Cycle Monitoring Time

Enter the scan cycle monitoring time in ms. When the scan time exceeds the scan cycle monitoring time, the CPU goes into the STOP mode.

Minimum Scan Cycle Time

With the minimum scan cycle time, you determine in the S7-400 in which interval the CPU program is called.

If the scan time is less than the specified minimum scan cycle time, the CPU waits until the minimum scan cycle time is reached. In this additional program execution time, the CPU processes the OB 90 (background OB) if it is loaded. The minimum scan cycle time is not available in the S7-300.

Scan Cycle Load from Communication

With the parameter "Scan Cycle Load from Communication", you can limit to a point that portion of the communication processor (data transmission to another CPU, loading of blocks with the PG, for example) that always prolongs the scan time.

The CPU‘s operating system continuously provides the communication with the projected percentage of the entire CPU processing performance (time-slice technique).

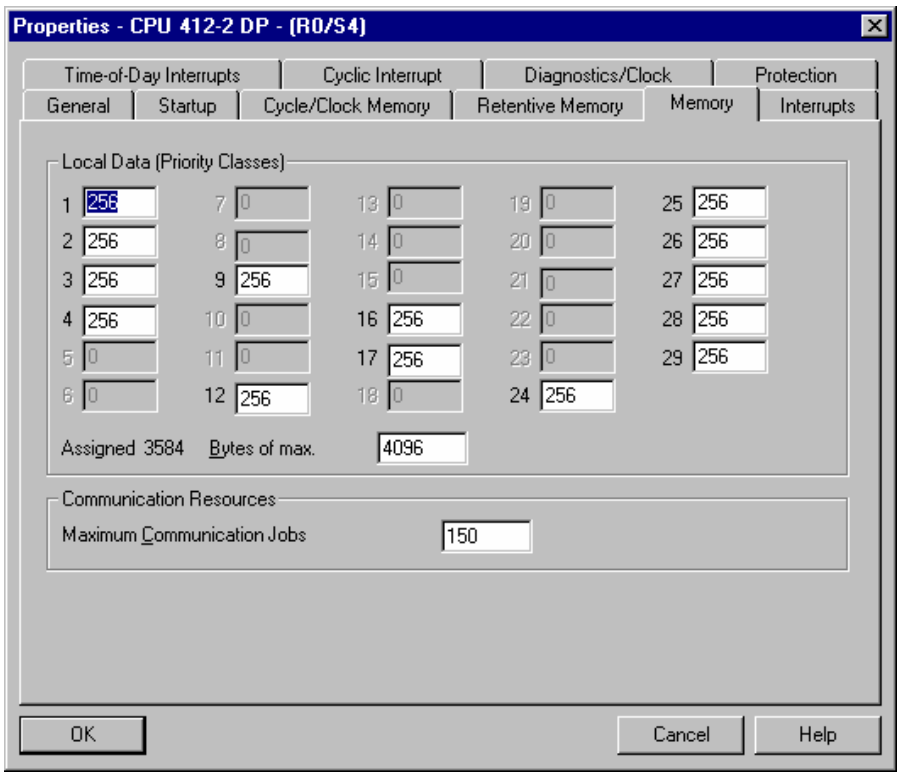
OB85 - Call Up

If you want to activate the OB 85 - Call Up at I/O Access Error, select the option "Only for incoming and outgoing errors“. That way, the CPU‘s scan time is not increased because of the repeated call of OB 85. This could be the case with the default option "At each individual access" (S7-400).

Clock Memory

Clock memories are bit memories that periodically change their binary value (pulse duration modulation ratio: 1:1).

CPU Parameters: Memory



Variable Memory Division

The new S7-400 CPUs allow you to have a variable division of the available work memory. You change the size of the work memory for logic or data blocks by changing the following parameters:

- size of the process image (byte-by-byte; in the "Cycle/Clock Memory" tab)
- Communication Resources (S7-400 only; in the "Memory" tab)
- size of the diagnostic buffer (in the "Diagnostics/Clock" tab)
- number of local data for all priority classes ("Memory" tab)

Parameters

So that you do not exceed the CPU‘s available work memory size, you have to take into consideration the following memory sizes during parameter assignment:

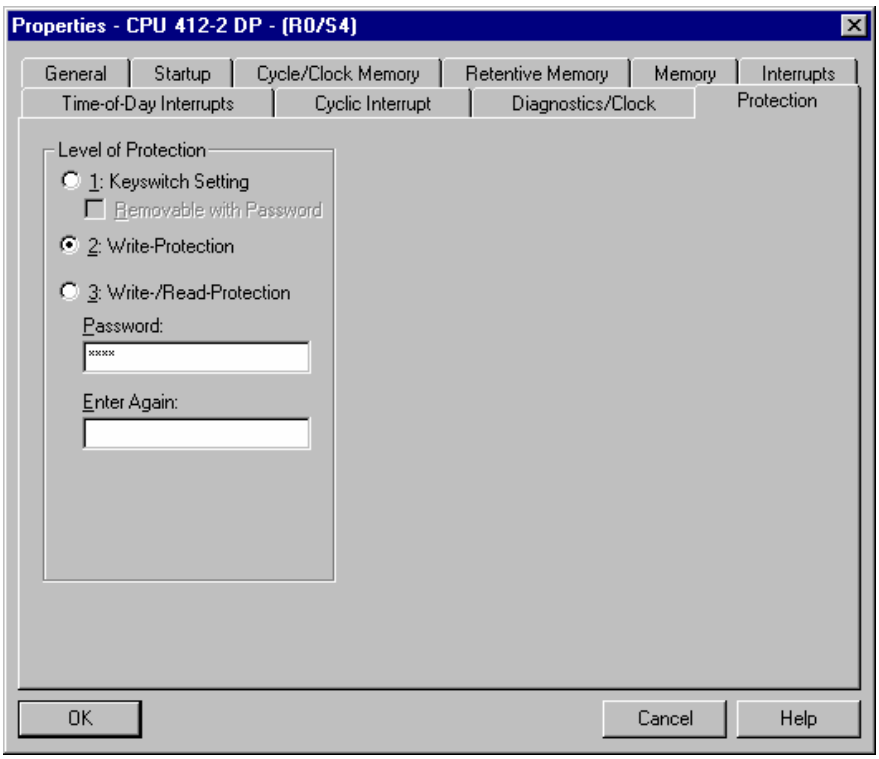
Parameter	Required Memory	Logic/Data Memory
Process Image size	12 bytes per byte	Logic memory
Communication jobs	72 bytes per job	Logic memory
Diagnostic buffer size	20 bytes per entry	Logic memory
Local data	1 byte per byte local data	Data memory

Note

When the division of the work memory is changed using parameter-assignment, the work memory is then reorganized when the system data is loaded into the CPU.

The result is that the data blocks that were generated by SFC are deleted and the other data blocks are preset with initial values from the load memory.

CPU Parameters: Protection Concept



Function

In this dialog you can select one of three protection levels in order to protect the CPU from unauthorized access.

Preset Characteristics

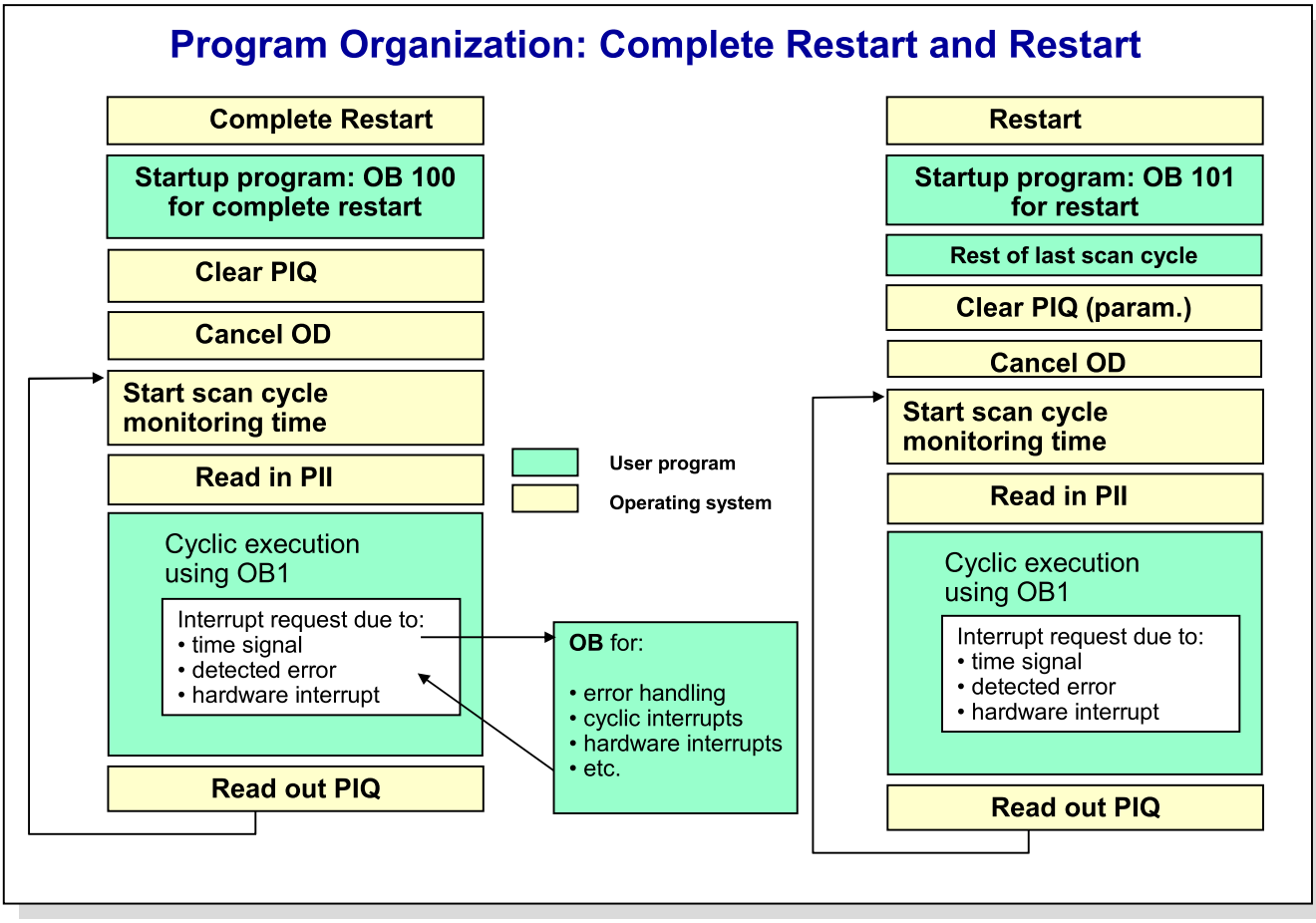
Protection level 1 (no password parameterized): The position of the CPU's key switch (mode selector) determines the protection:

- key switch in position RUN-P or STOP: no restrictions
- key switch in position RUN: read-only access possible!

Parameterized Protection Level

If you have parameterized a protection level with password:

- read and write access is possible for "Password Owners", independent of the position of the key switch and independent of the parameterized protection level.
- the following restrictions are applicable for "Password Non-owners":
  - Protection level 1: corresponds to the preset characteristics
  - Protection level 2: read-only access possible, regardless of the key switch position
  - Protection level 3: neither reading nor writing access possible, regardless of the key switch position



Complete Restart

On complete restart, the non-retentive memory areas of the bit memories, counters and timers are reset after OB100 has been processed. OB1 always starts with the 1st statement.

Cold Restart

The actions taken during a cold restart are similar to those for a complete restart, with the following differences:

- OB102 is called instead of OB100
- The data blocks generated by SFCs at runtime are deleted, the other data blocks are initialized with the value from the load memory.
- The process image and all the timers, counters and bit memories are reset, whether they have been parameterized as retentive or not.

Restart

On restart, after the restart OB101 has been executed, execution of OB1 continues at the point where it was interrupted. That is, the remainder of the cycle is completed with the retentive bit memories, timers and counters.

OD

Output disable, bus cable in S7 (corresponds to the command output inhibit in S5)

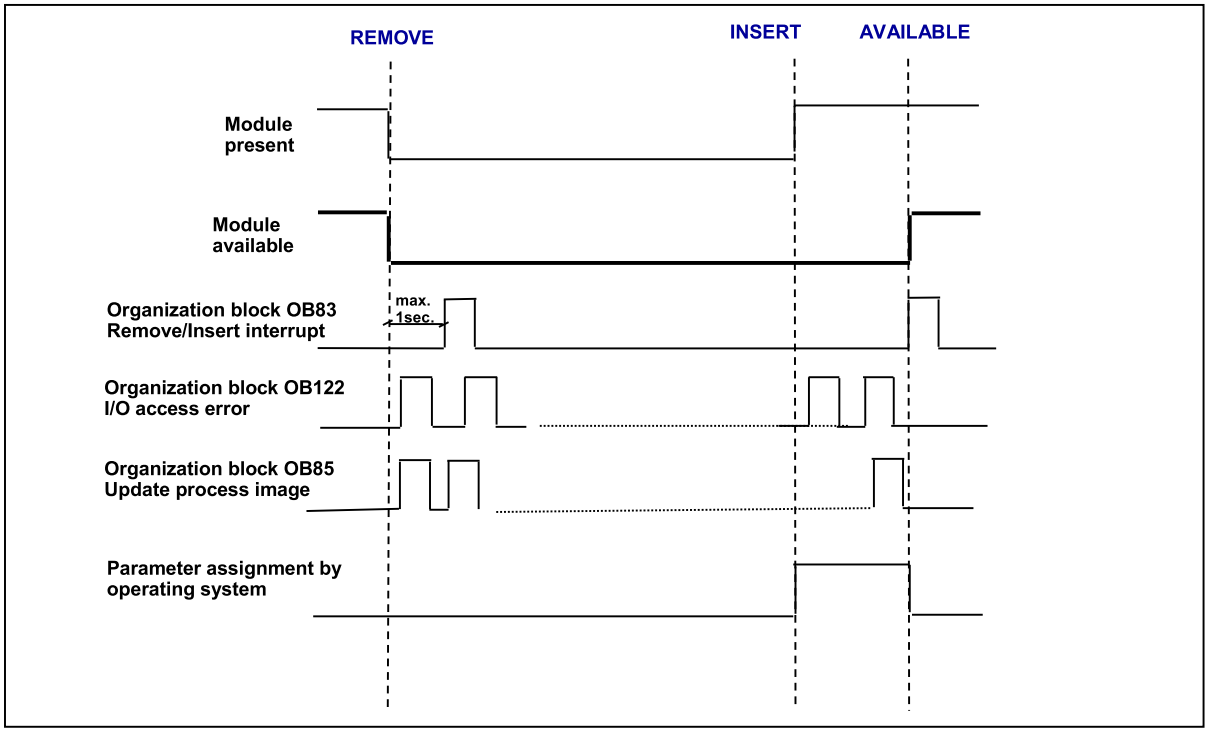
Actions

The operating system performs the following actions on startup:

- Clear stacks (C/CR)
- Reset non-retentive bit memories, timers and counters (CR)
- Reset all bit memories, timers and counters (C)
- Clear process-image output table PIQ (C/CR), if parameterized (R)
- Reset output memory area (C, CR), if parameterized (R)
- Clear interrupts (C/CR/R) via OD
- Update system status list (C/CR/R)
- Transfer configuration to modules (C/CR/R)

(CR= Complete Restart, C=Cold Restart, R= Restart).

The Insert/Remove-Module Interrupt on the S7 - 400



Remove and Insert Interrupt OB83

In the S7-400, you can remove and insert modules while powered up in RUN or in STOP mode. The exceptions are CPUs, PSs, S5 modules in adapter modules and IMs.

After removing a module in the RUN mode, the CPU's operating system can call one of the following organization blocks - depending on the situation:

- OB85-Process image update
- OB122-I/O access error
- OB83- Remove&Insert event.

The user must take into consideration that OB83 is only called after approximately 1sec., while the other OBs, as a rule, become active much sooner.

After inserting the module, it is checked by the CPU and - if no type error exists - is assigned parameters. After an orderly parameter assignment, the module is available for use.

If an error is recognized in the parameter assignment, the OB82 diagnostic interrupt is automatically started.

Start Information in OB83

The following information exists in the OB83's local data:

- Remove/Insert of a module
- Logical address of the module
- Actual type of the module

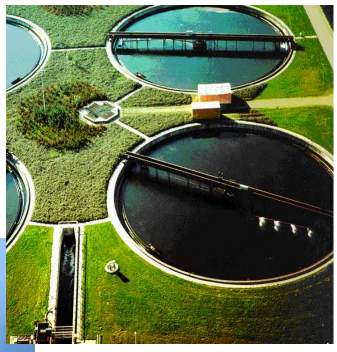
Replacement Value

You can use the system function SFC 44 (RPL_VAL) to substitute a replacement value for the missing process signals from an input module in the error OB.

SIMATIC S7-400H – The Fault-tolerant PLC

Motivation for Installing H-Systems

- **Avoid high restart costs after a controller breakdown**
- **Avoid expensive downtimes (in the production of expensive materials, for example)**
- **Make operation without supervision or maintenance personnel possible**



Areas of Use

- **Power Generation, Power Distribution**
- **Environmental Technologies**
- **Chemical, Petro-chemical, Pharmaceutical**
- **Mining, Transport**
- **Pulp and Paper, Printing**
- **Airport Automation**

SIMATIC S7

Date: 30.06.2011
File: PRO2_10E.26

SITRAIN Training for Automation and Drives

Applications

In many areas of automation, ever greater requirements are being made on the availability and thus the fault tolerance of automation systems. These are areas in which a system downtime would cause great expense. Only redundant systems can meet the requirements of availability here.

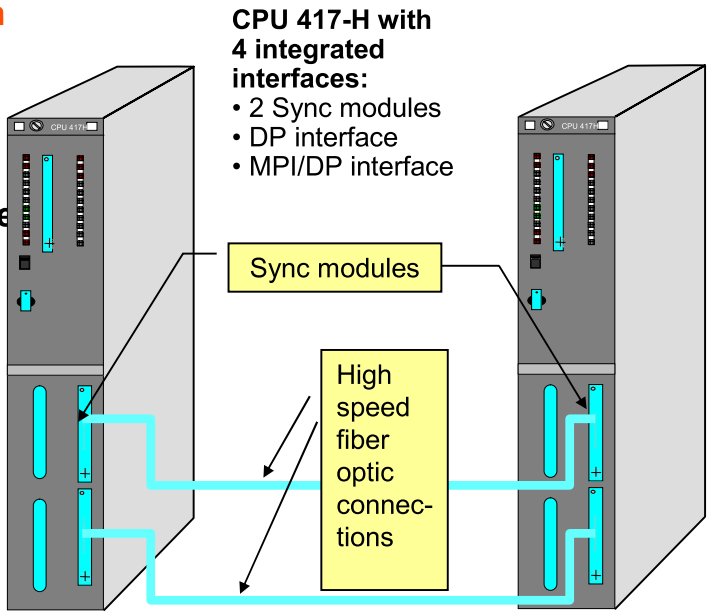
The fault-tolerant SIMATIC S7-400H fulfills these requirements. It even continues to work when parts of the controller have failed because of one or more errors. SIMATIC S7-400H is particularly well suited for the following areas of application:

- processes with high restart cost after a controller breakdown (generally in the process industry)
- processes with expensive downtimes
- processes for processing expensive materials (in the pharmaceutical industry, for example)
- unsupervised applications
- applications with reduced maintenance personnel

SIMATIC S7-400H – Redundancy Connection of the CPU

Features of redundancy functions

- **Automatic event synchronization with guaranteed smooth Master Slave changeover (Siemens patent)**
- **Fault tolerant communication as operating system performance**
- **Online functions:**
  - **Module exchange while in RUN**
  - **Firmware update while in RUN**



Redundancy Principle

The S7-400H works according to the principle of active redundancy in "Hot Stand-By" mode (automatic changeover when there is a failure). With this principle, both sub-units are active in a 'healthy' operation. When there is a failure, the intact device takes over control of the process.

In order to guarantee that this take over is smooth, a fast and dependable data exchange through the central controller connection is necessary.

The controllers automatically receive:

- the same user program
- the same data blocks
- the same process image contents
- the same internal data such as timers, counters, bit memories, ...

That way, both controllers are always up-to-date and can carry on the control alone anytime there is a failure.

Synchronization

A synchronizaton of the sub-units is required for a non-interacting changeover. The S7-400H works with "event-driven synchronization". A synchronization always takes place when events could produce a different internal state of the sub-units, such as:

- direct access to the I/O
- interrupts and alarms
- updating the user times
- modification of data through communication functions

Synchronization takes place automatically through the operating system and does not need to be taken into consideration during programming.

Fault-tolerant Communication

Communication can be carried on when there is a failure with the fault-tolerant communication using up to 4 redundant connections. The required changeover is invisible to the user when it takes place. The redundancy function is established exclusively during parameter assignment.

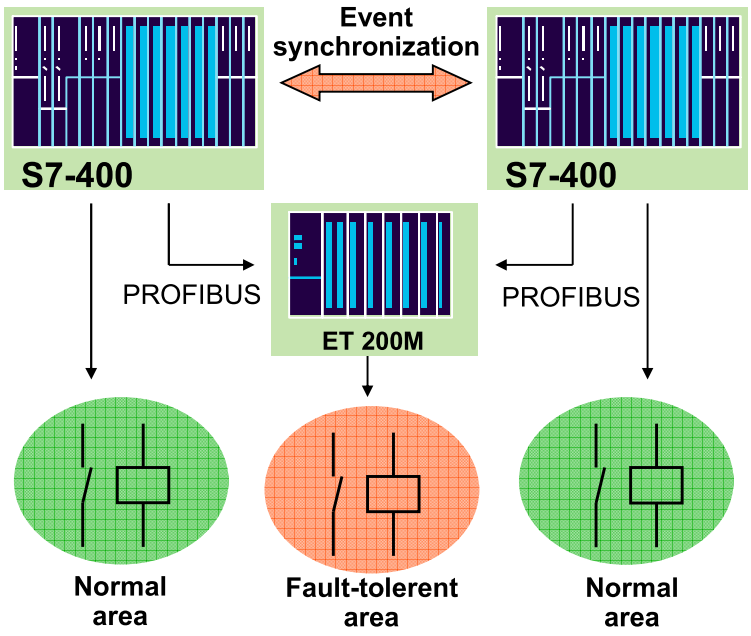
SIMATIC S7-400H – Topology for Fault-Tolerance

Design

- **2 separated central controllers UR1/UR2 or 2 areas in one (UR2-H)**
- **1 CPU 417-4H for each central controller**
- **2 Sync modules for each central controller (connection using fiber optic)**
- **Peripherals**

Peripherals

- **normal availability (single-sided)**
- **fault-tolerant (switched)**



Design

The SIMATIC S7-400H consists of the following components:

- Either 2 separate central controllers UR1/UR2 or 2 areas in one divided central controller (UR2-H)
- 2 Sync modules for each central controller for connecting both devices using fiber optics
- 1 CPU 417-4H for each central controller
- S7-400 I/O modules in the central controllers
- Expansion devices UR1/UR2/ER1/ER2 and/or distributed peripheral devices
  ET 200M with I/O modules

I/O Connection

I/Os can be configured as normal availability and/or switched (increased availability).

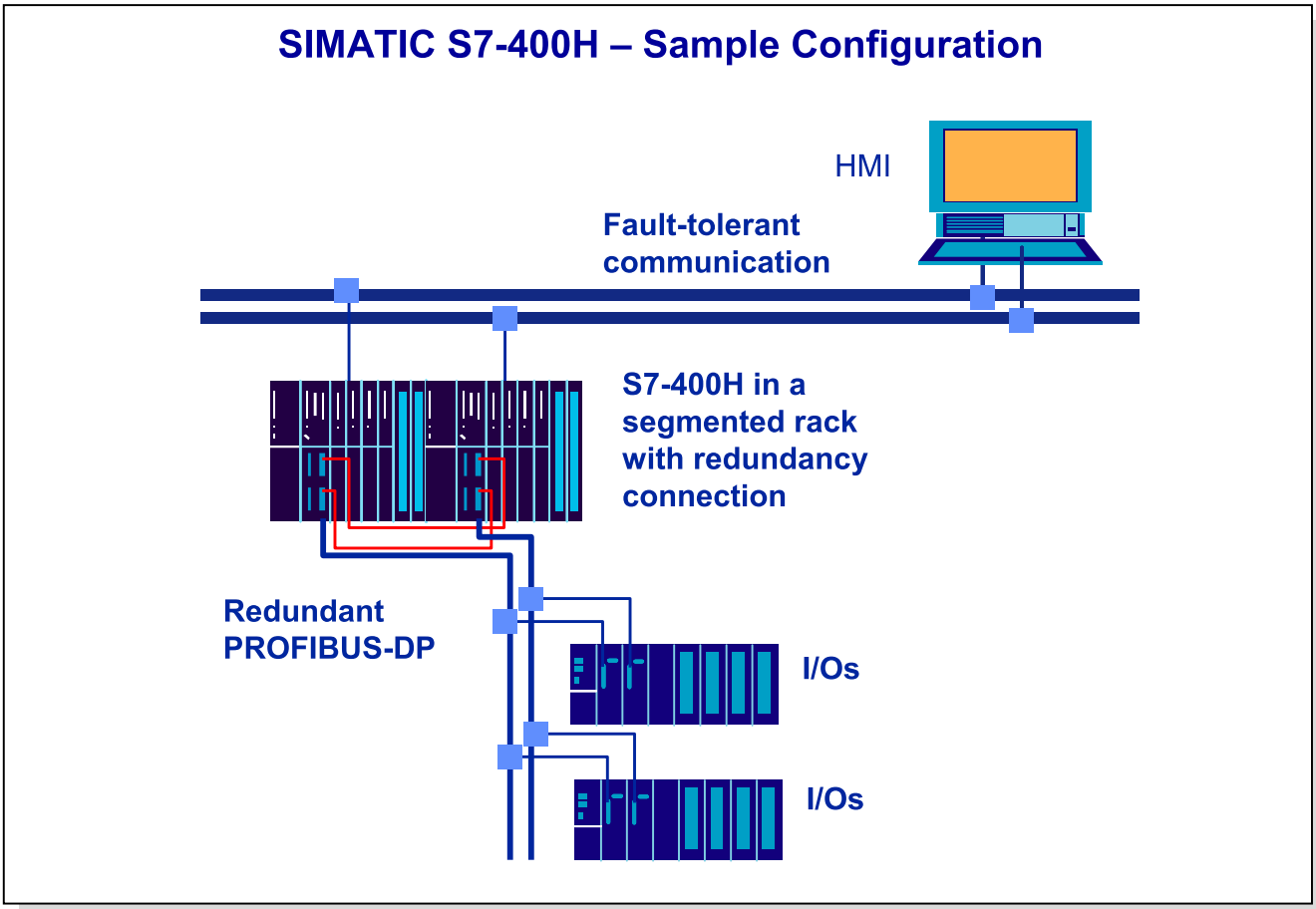
Normal Availability I/Os

With normal availability I/Os (single-sided design), I/O modules have a one channel design and are only addressed by one of the two central controllers. I/O modules that have a single-sided operation can either be slotted in a central controller and/or in expansion devices/distributed peripheral devices.

Single-sided read-in information is available to both central controllers, as long as the controller that addresses the peripherals functions correctly.

When there is a failure, the I/O modules of the central controller that is affected are out of service.

The single-sided design is used for those sections of a system that do not require an increased availability.



Fault-tolerant I/O

With a switched design, I/O modules have a one channel design. They are however addressed by both central controllers using a redundant PROFIBUS-DP. I/O modules that have a switched operation can only be slotted in an ET 200M distributed peripheral device.
A connection to the central controllers is made using PROFIBUS-DP. The switched ET 200M are connected to both sub-units.

Redundancy of the I/O Peripherals

I/O modules can be operated as redundant in two ways:

- by using two identical modules in two switched ET 200M stations
- by using two identical modules that are each assigned to the two sub.units

Technically, the program is executed at the user level.

Redundancy of the FM and CP

Function modules (FM) and communications modules (CP) can be installed as redundant in two different configurations:

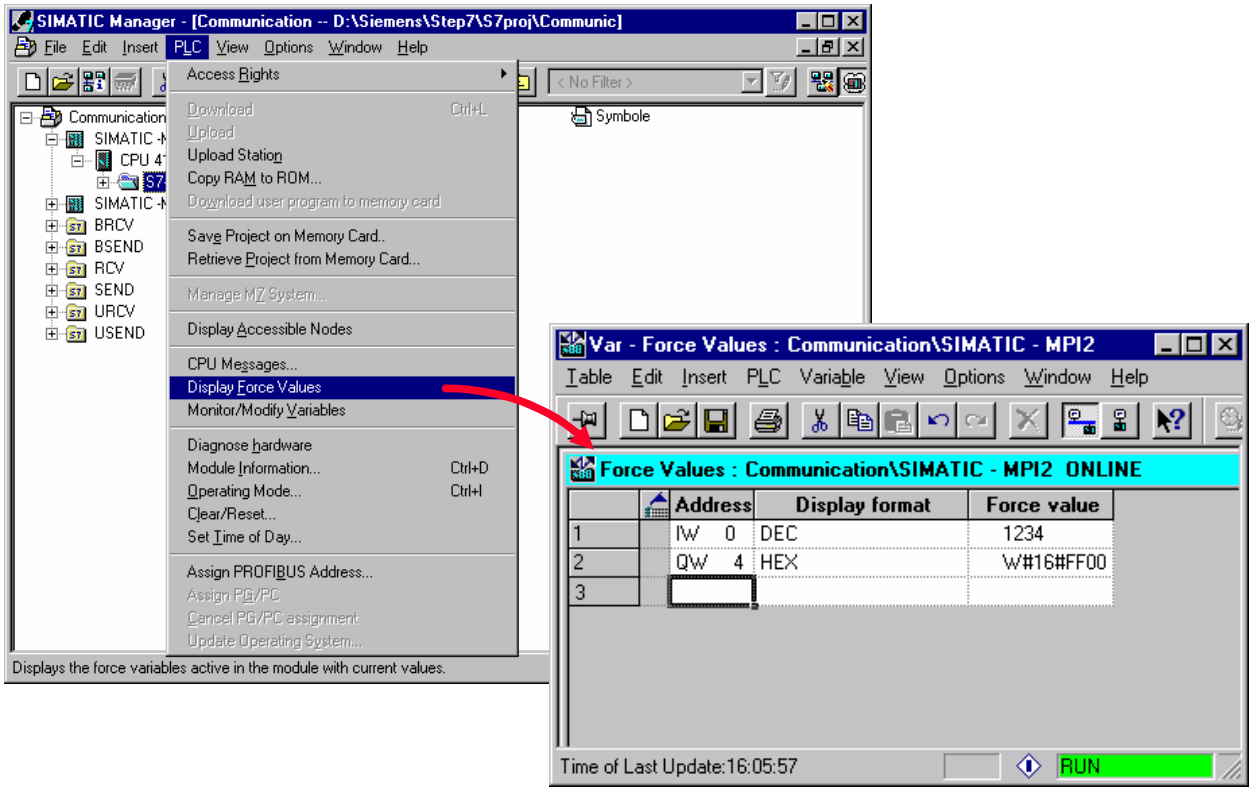
- switched redundant configuration; the FM/CP can be double slotted in a separated or in a switched ET 200M design.
- two channel redundant design; FM/CP can each be slotted in both sub-units or in expansion devices connected to it.

The redundancy of the modules is thereby achieved in different manners:

- programming by the user; the redundancy function can generally be programmed by the user for the FM and the CP.
  The active module is thereby specified and a possible fault recognized, in order to cause a changeover.

  The required program corresponds to that of a simple CPU with redundant FM/CP.
- direct support by the operating system; the operating system directly supports the redundancy for the SIMATIC NET CPs (CP 443-1, CP 443-1TCP, CP 443-5 Basic and CP 443-5 Ext.).

Forcing on the S7-400



Forcing

With the function *Forcing* you can, in S7-400, establish predefined values for user program variables.

Notes for Forcing

- Before you start the "Force" function, you should make sure that no one else is performing this function simultaneously on the same CPU.
- A force job can only be cancelled or ended with the menu option *Variable -> Stop Forcing*.
- "Forcing" cannot be undone with the menu option *Edit -> Undo* .
- The closing of the Force Values window or the ending of the "Monitor/Modify Variables" application does not cancel the force task.

Addresses

The following addresses can be forced in the S7-400:

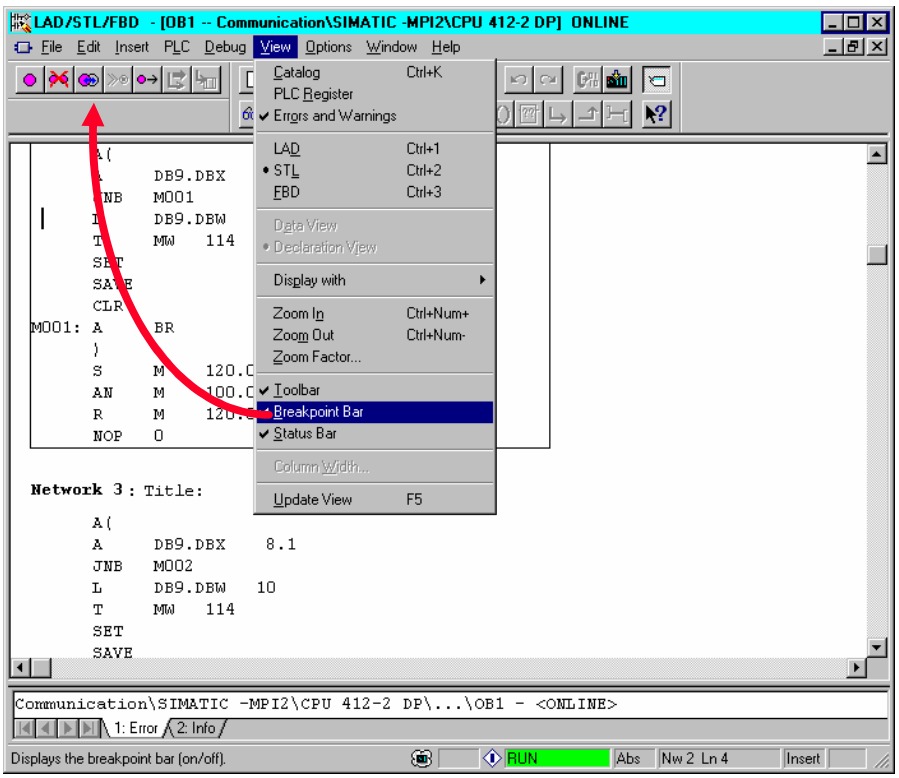
- Peripheral inputs and outputs
- Process images of inputs and outputs
- Bit memories

Selecting the "Force" Function

1. Within the SIMATIC Manager select the CPU to be forced and than use the menu option *PLC -> Display Force Values* to open the Force Values window in which the current status of the selected CPU is displayed.

   If no force job is currently active, the window is empty. If a force job is active already, the variables are displayed in bold face with the corresponding force values.
2. In the "Address" column, enter the variables you want to force. In the "Force Value" column, enter the values which you want to assign to the variables.
3. Start forcing with the menu option *Variable -> Force*.
4. You can terminate the force job with the menu option *Variable -> Stop Forcing*.

Activating the Breakpoint Bar



Breakpoints

With the help of this test function, you can test a program you have created in the STL programming language in single-step mode and thus follow the sequence of the executed instructions as well as the associated register contents.

Several breakpoints can be set in a block. The number of possible breakpoints depends on the CPU used.

Activating

So that the breakpoint function can be activated, the following conditions must be fulfilled:

- the block must be opened "on-line"
- Test operation (*Debug -> Operation -> Test Operation)* must be selected
- in the Editor, STL (*View -> STL*) must be explicitly selected

Breakpoint Functions

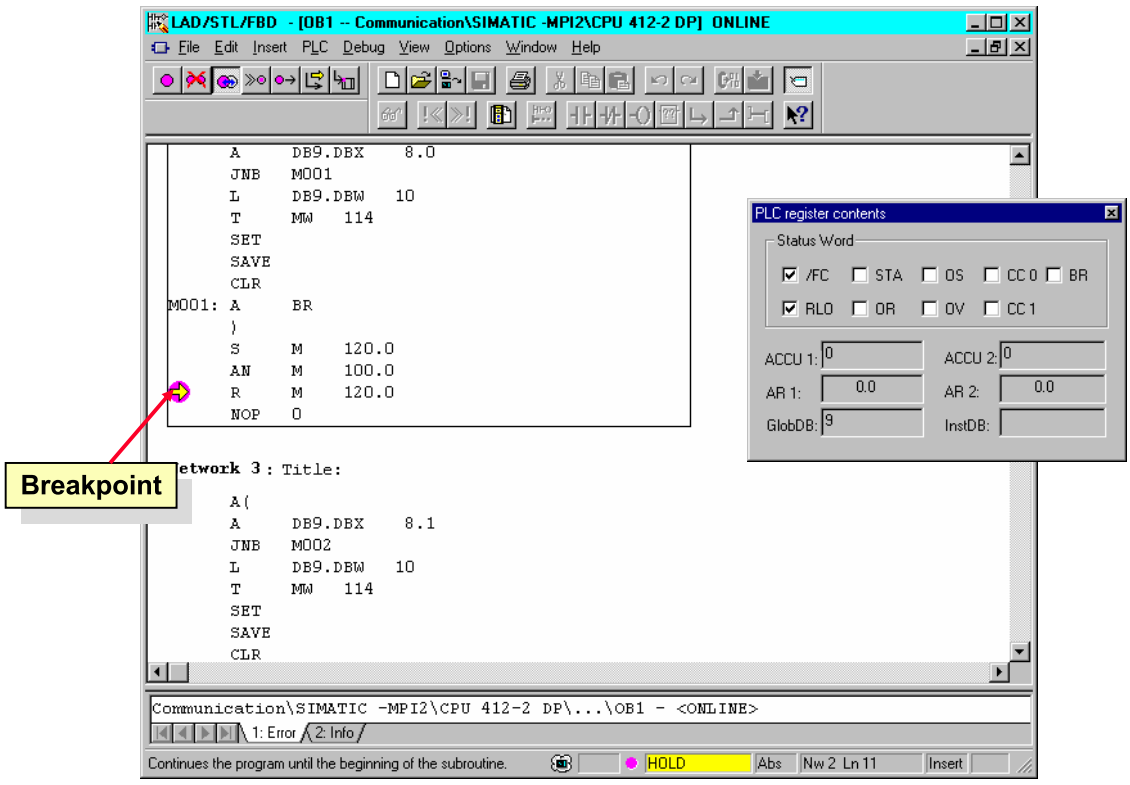
You can choose the breakpoint functions in the Program Editor by selecting the menu option "Test" or through the Breakpoint Bar.

Breakpoint Bar

You can activate the breakpoint bar by selecting the menu option *View -> Breakpoint Bar* in the Program Editor.

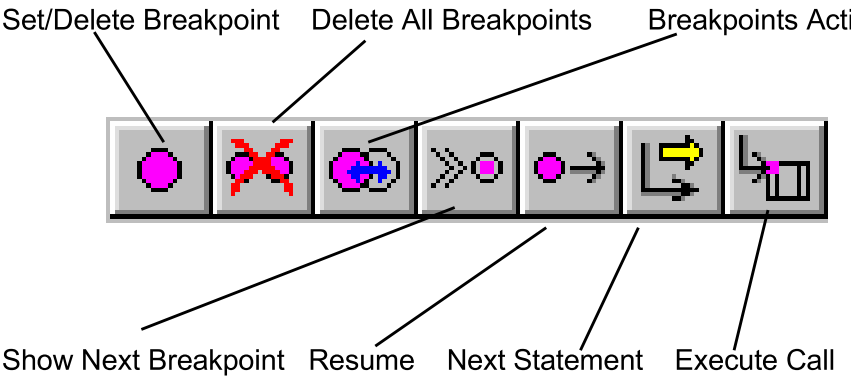
- The menu options *Execute Next Statement* or *Execute Call* require a free breakpoint for the internal implementation.
- If the program execution comes up against a breakpoint, the CPU switches from RUN to HOLD. In this mode, the STOP LED lights up and at the same time the RUN LED flashes.

Program Execution using Breakpoints



Breakpoint Bar

The breakpoint bar has the following buttons for "Testing in Single-Step Mode".



Set/Delete Breakpoint

With "Set/Delete Breakpoint", you determine where the program execution is to be halted. The breakpoint‘s statement is not executed.

Delete All Breakpoints

All breakpoints are deleted.

Breakpoints Active

With "Breakpoints Active", you activate all breakpoints; not only those already set but also those still to be set.

Show Next Breakpoint

With "Show Next Breakpoint", the Editor jumps to the next selected breakpoint, without executing the program.

Resume

With "Resume", the program runs until the next active breakpoint.

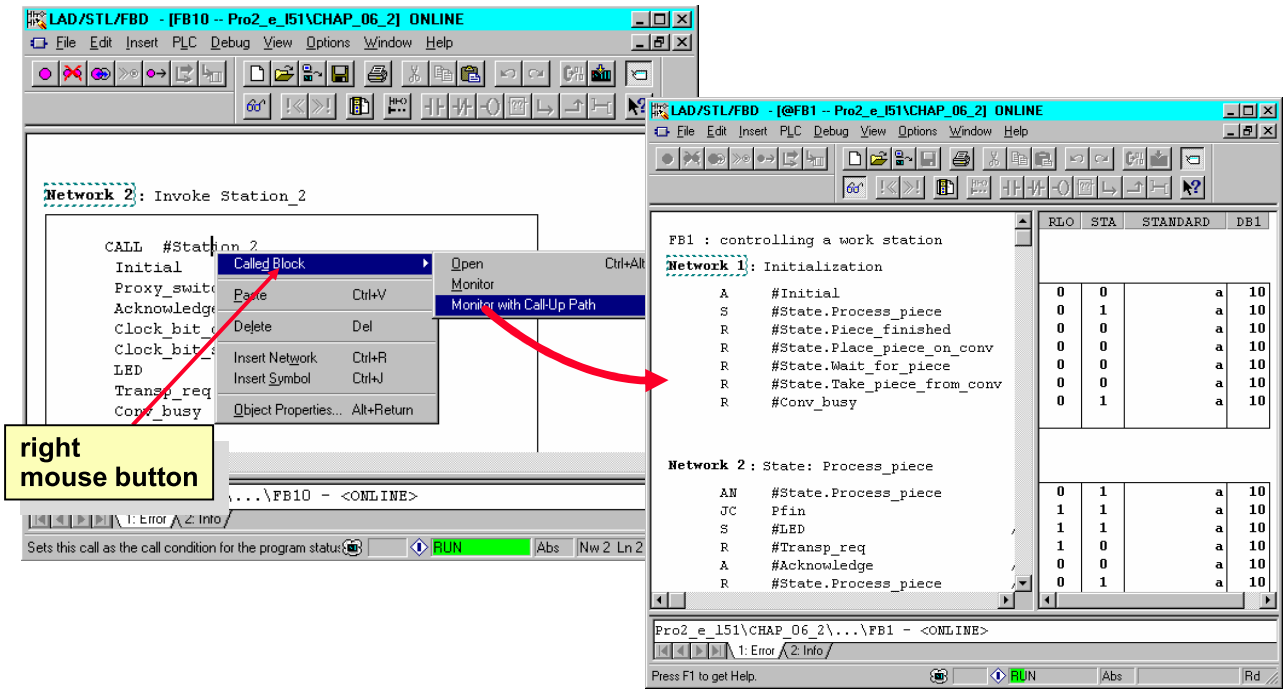
Next Statement

With "Next Statement", you execute the program in single step. If you reach a block call, you jump to the first statement after the block call with "Next Statement".

Execute Call

Here, when you reach a block call you branch into the block with "Execute Call". At the end of the block you jump back to the next statement after the block call.

Monitoring a Block with Call-up Path



Area of Use

In the above example, three instances of FB1 (Station_1, Station_2 and Station_3) are called within the FB10. Neither a separate call-up path nor open data blocks (global DB and instance DB) can be specified as trigger conditions for these three calls.

A targeted monitoring of a single call (instance) is only possible in this case using the function "Monitor with Call-Up Path".

Selection

For a targeted monitoring of an FC/FB call, proceed as follows:

1. Open the calling block online (FB 10 in the example).
2. With the right mouse button, click on the call of the block to be monitored (Call box in LAD/FBD or Call line in STL)
3. In the pop-up menus that appear, select the options:
   Called Block > Monitor with Call-Up Path

The block to be monitored is then opened online (FB1 in the example) and the test function Monitor Block is activated.

Note

The function described above is unrestricted in S7-400. For S7-300, only as of the October 2000 version.

CP 440 for Point-to-Point Connections

Area of application:

- **For high performance transmission of short messages**

Properties:

- **RS 422/RS 485 (X.27)**
- **maximum 115.2 kbits/s**
- **up to 32 connections**

Protocols:

- **ASCII,**
- **3964 (R)**

CP 440

Date: 30.06.2011
File: PRO2_10E.34

Description

Point-to-point connections is a powerful and economical alternative to a bus system. Simple devices, such as barcode readers, scales, etc., as well as other automation systems can be connected.

The CP 440 communication processor is used for high performance transmission of short messages using RS 422/RS 485 (X27).

Protocols

Various implemented standard protocols enable data exchange with the different partners:

- ASCII;
  for connection to third-party systems with simple transmission protocols, such as, protocols with start and end-of-text character or protocols with block check character.
  The handshake signals of the interface can be queried and controlled through the user program.
- 3964 (R);
  for connection to Siemens devices or third-party components using the standardized and open Siemens protocol 3964(R). A 3964(R) driver with standard values as well as a parameter-assignable 3964(R) driver are implemented.

CP 441 for Point-to-Point Connections

Interfaces:

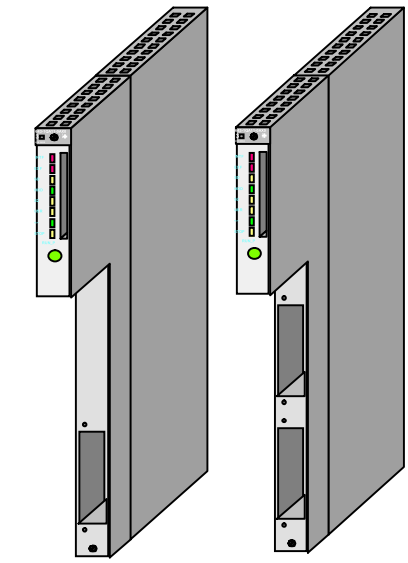
- **CP 441-1: 1 plug-in interface module**
- **CP 441-2: 2 plug-in interface module**

Interface modules:

- **20mA (TTY):**
  max. 19.2 kbits/s
- **RS 232C (V.24):**
  max. 38.4 kbits/s (CP 441-1)
  max. 115.2 kbits/s (CP 441-2)
- **RS 422/485 (X.27):**
  max. 38.4 kbits/s (CP 441-1)
  max. 115.2 kbits/s (CP 441-2)

Protocols:

- **integrated standard protocols, such as:**
  ASCII, 3964 (R), printer
- **loadable Non-Siemens protocols (special drivers) - for CP 441-2**



SIMATIC S7

Date: 30.06.2011
File: PRO2_10E.35

SITRAIN Training for Automation and Drives

Description

Point-to-point connection is a cost-effective and high-performance alternative to a bus system. It is possible to connect simple devices such as bar-code readers and scales, as well as PLCs.

CP441-1 Protocols

With one plug-in interface

- 3964 (R) parameter assignable
- ASCII protocol parameter assignable
- Printer

CP441-2 Protocols

With two plug-in interfaces

- 3964 (R) parameter assignable
- RK 512
- ASCII protocols parameter assignable
- Printer
- Loadable external protocols such as:
  - Modbus master / slave (Modicon)
  - Allen Bradley (DF1 protocol)

Interface Modules

- TTY, 9-pin sockets, active/passive interface module
- V.24 (RS232 C), 9-pin plug connector
- RS 422/485, 15-pin socket

CP 443-5: Connection to PROFIBUS

Protocols:

- **S7 functions**
- **PG/OP communication including Routing**
- **SEND/RCV**
- **time synchronization**
- **FMS (CP 443-5 Basic only)**
- **DP Master (CP 443-5 Extended only)**

Baud rate:

- **9.6 Kbps to 12 Mbps**

Connection:

- **electr. cable: using 9-pin sub-D socket**
- **FO cable: using bus terminal**

CP 443-5 Basic
CP 443-5 Extended

Description

The CP 443-5 communication processor permits connection of the S7-400 to the PROFIBUS network.

Protocols

The following protocols are available:

S7 Functions:

For communications based on layer 7 of the ISO/OSI reference model between SIMATIC S7/M7/C7 and PCs.

The S7 protocol provides an SFB interface for the S7-CPUs for communication within the SIMATIC S7 family. Additionally, functions for programming, testing, object administration and diagnostics are provided.

SEND/RCV:

For communication based on layer 2 (FDL layer) between SIMATIC S7, SIMATIC S5, PC/PGs and non-SIEMENS devices. The SEND/RCV interface provides simple, reliable communication of unstructured data blocks.

The SEND/RCV interface is implemented with

- Handling blocks in SIMATIC S5
- Function calls in SIMATIC S7
- C function calls on PGs/PCs

PROFIBUS FMS (Fieldbus Message Specification)

For open communication between SIMATIC S5, S7, PCs/PGs, field devices and non-SIEMENS products. (EN 50170, Vol. 2, PROFIBUS).

The FMS protocol permits object-oriented communication on layer 7 of the ISO/OSI reference model. The typical data packet size is 240 bytes.

PROFIBUS DP (Distributed Peripheral)

For open communication between SIMATIC S5, S7, PCs/PGs or non-SIEMENS systems and field devices. (EN 50170, Vol. 2, PROFIBUS).

The DP protocol is designed for fast exchange of small amounts of data between PLCs or PCs and field devices with response times < 10 ms.

CP 443-1: Connection to Industrial Ethernet

Protocols:

- **S7 functions**
- **PG/OP communication including Routing**
- **SEND/RCV**
- **time synchronization**
- **H communication:**
  For redundant S7 communication in SIMATIC H-Systems

Interfaces:

- **Connection to Industrial Ethernet (10/100 Mbit/s) using 15pole Sub-D socket (automatic changeover between AUI and ITP)**
- **Connection to 10BaseT, 100BaseTX using RJ45**

CP 443-1 (ISO and TCP/IP)

Description

The CP 443-1 is the communication processor of the SIMATIC S7-400 for Industrial Ethernet. With its own processor, it relieves the CPU of communications tasks and enables further connections. S7-400 communication facilities using CP 443-1 with:

- programming devices, computers, HMI devices
- other SIMATIC S7 systems
- SIMATIC S5 automation devices

Functions

The CP 443-1 controls data communication itself using Industrial Ethernet. The module has its own processor. The levels 1 to 4 correspond to international standards. The multiple protocol operation of the transport protocols ISO and TCP/IP is possible.

The CP 443-1 works in multiple protocol operation for the following communication services:

- PG/OP communication: for remote programming of all S7 stations connected to the network. With the help of S7-Routing, it is possible to have cross-network PG/OP communication.
- S7 communication for connection to S7-300, S7-400 and PCs.
- H-communication for the redundant S7 communication in SIMATIC H-Systems.
- time synchronization
- S5-compatible communication based on level 4 represents a simple and optimized interface for data communication.

  With this interface:

  - ISO Transport
  - TCP Transport with RFC 1006 (CP 1430 TCP, for example) or without RFC 1006
  - UDP

  can be used as the transmission protocol of the CP 443-1.

CP 443-1 IT: Connection to the Internet

Same format and functionality as CP 443-1:

- **S7 functions**
- **PG/OP communication including Routing**
- **Send/Receive using RFC 1006 and UDP**

Additional Internet functionality:

- **CP 443-1 IT is a WWW server:**
  - **HTML pages and applets on board for S7 functions.**
  - **WWW server is used for operator control / monitoring of small controllers**
  - **no costs at the client end**
  - **platform-independent**
  - **familiar Internet operating philosophy**
- **CP 443-1 is an e-mail client:**
  - **e-mail can be used for the easy alerting to faults**
  - **mobile phone, pager, PC, Fax etc. are accessible**

CP 443-1 IT

SIMATIC S7

Date: 30.06.2011
File: PRO2_10E.38

SITRAIN Training for Automation and Drives

Description

The CP 443-1 IT communications processors allow you to connect S7-400 to the Internet.

Protocols

The following protocols are available for the CP 443-1 IT:

S7 Functions

For communication based on layer 7 of the ISO/OSI reference model between SIMATIC S7/M7/C7 and PCs.

SEND/RCV

For communication based on layer 4 (TCP Transport Stack) between SIMATIC S7, SIMATIC S5 and PC/PGs.

Communication via Internet

As well as industrial communication facilities, this Internet CP provides access to the Internet. This means:

- Users can log into the system from anywhere using a password.
- You can read the process and operating data of the plant with any Internet browser (Internet Explorer, Netscape, etc.). (Integral HTML pages for system information).

  Operator intervention is also possible, allowing servicing to be performed worldwide.

The Internet CP also enables you to send all important production or plant status information anywhere in the world, for example:

- by e-mail via the Internet
- to mobile phones or fax machines
- to external PCs
- to pagers or palmtops with Internet access.

CP 444-1: Use of MMS Services according to MAP 3.0

Functionality:

- **Implemented MMS services:**
  - **Environment Management (Initiate, Conclude and Abort)**
  - **VMD Support Services (Unsolicited Status, Status, GetNameList, Identify and GET CapabilityList)**
  - **Variable Access Services (Read, Write, Information Report and GetVariableAcessAttributes)**

Connection:

- **connection to Industrial Ethernet according to the Ethernet Standard IEEE 802.3**
- **15-pole Sub-D socket with sliding lock for the bus connection to Industrial Ethernet**
- **automatic changeover between AUI and Twisted Pair interface**

CP 444

SIMATIC S7

Date: 30.06.2011
File: PRO2_10E.39

SITRAIN Training for Automation and Drives

Description

The CP 444 communication processor make the connection of the SIMATIC S7-400 to Industrial Ethernet possible. The MMS services (Manufacturing Message Specification) according to the communications standard MAP 3.0 are used for it.

Design

The CP 444 communication processor has all the benefits of the SIMATIC S7-400 design:

- compact design; on the front, the robust plastic casing contains: 15-pole Sub-D socket with sliding lock for the bus connection to Industrial Ethernet (automatic changeover between AUI and Twisted Pair interface)
- easy mounting; the CP 444 is mounted on the rack of the S7-400 and is connected with the other S7-400 modules through the backplane bus.

  The CP 444 has a fanless operation. A backup battery is not required.

Note

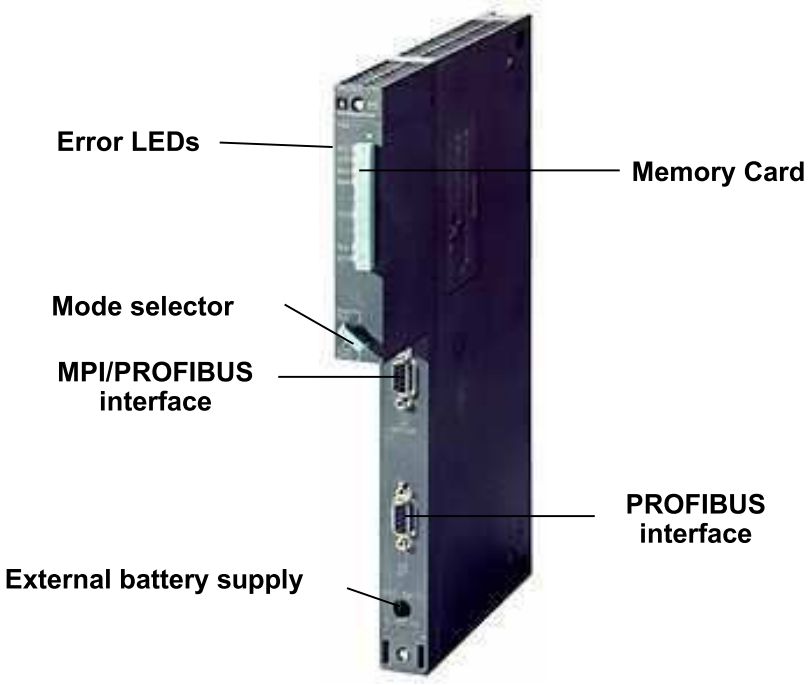
The CP 444 can only be connected to self-supporting end devices such as the SSV104 using AUI.

Function

- Connection to Industrial Ethernet according to the Ethernet Standard IEEE 802.3
- MMS services: VMD (device monitoring) and variable services (language-neutral data transfer)
  - Environment Management (Initiate, Conclude and Abort)
  - VMD Support Services (Unsolicited Status, Status, GetNameList, Identify and GETCapabilityList)
  - Variable Access Services (Read, Write, Information Report and GetVariableAcessAttributes)

The MMS services VMD Support Services and Variable Access Services are called because of an event using function blocks from the user program of the interested CPU.

Distributed I/O and Parameter Assignment

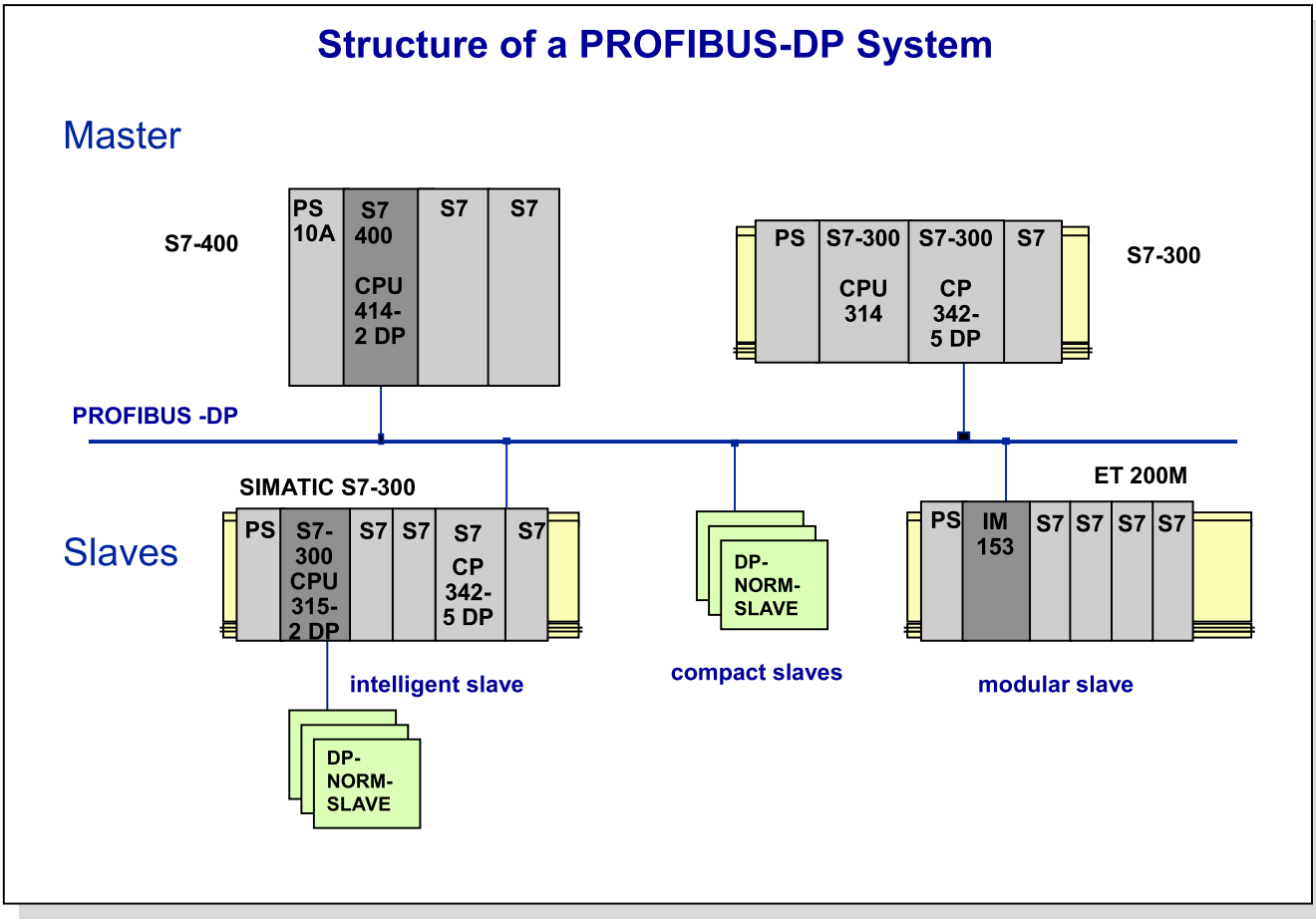


SIMATIC S7

Date: 6/30/2011
File: PRO2_11E.1

SITRAIN Training for Automation and Drives

Contents **Page**



Overview

The equipment installed in the field for automating technical processes, such as sensors, actuators, transducers and drives, is making increasing use of fieldbus systems for exchanging information with the higher-level control units.

PROFIBUS is an established fieldbus system which can be used by all automation equipment, such as PLCs, PCs, human-machine interfaces and actuators and sensors, for exchanging data.

PROFIBUS-DP

PROFIBUS-DP is a protocol optimized for speed, which was specially designed for communication between PLCs (DP masters) and distributed I/Os (DP slaves).

PROFIBUS-DP is a low-cost, flexible substitute for transmission of signals on cumbersome parallel 24V and 20mA lines.

PROFIBUS-DP is based on DIN 19245 Part 1 and the user-specific extensions in DIN 19245 Part 3. In the course of the European fieldbus standardization process, PROFIBUS-DP was integrated into the European fieldbus standard EN 50170.

Masters

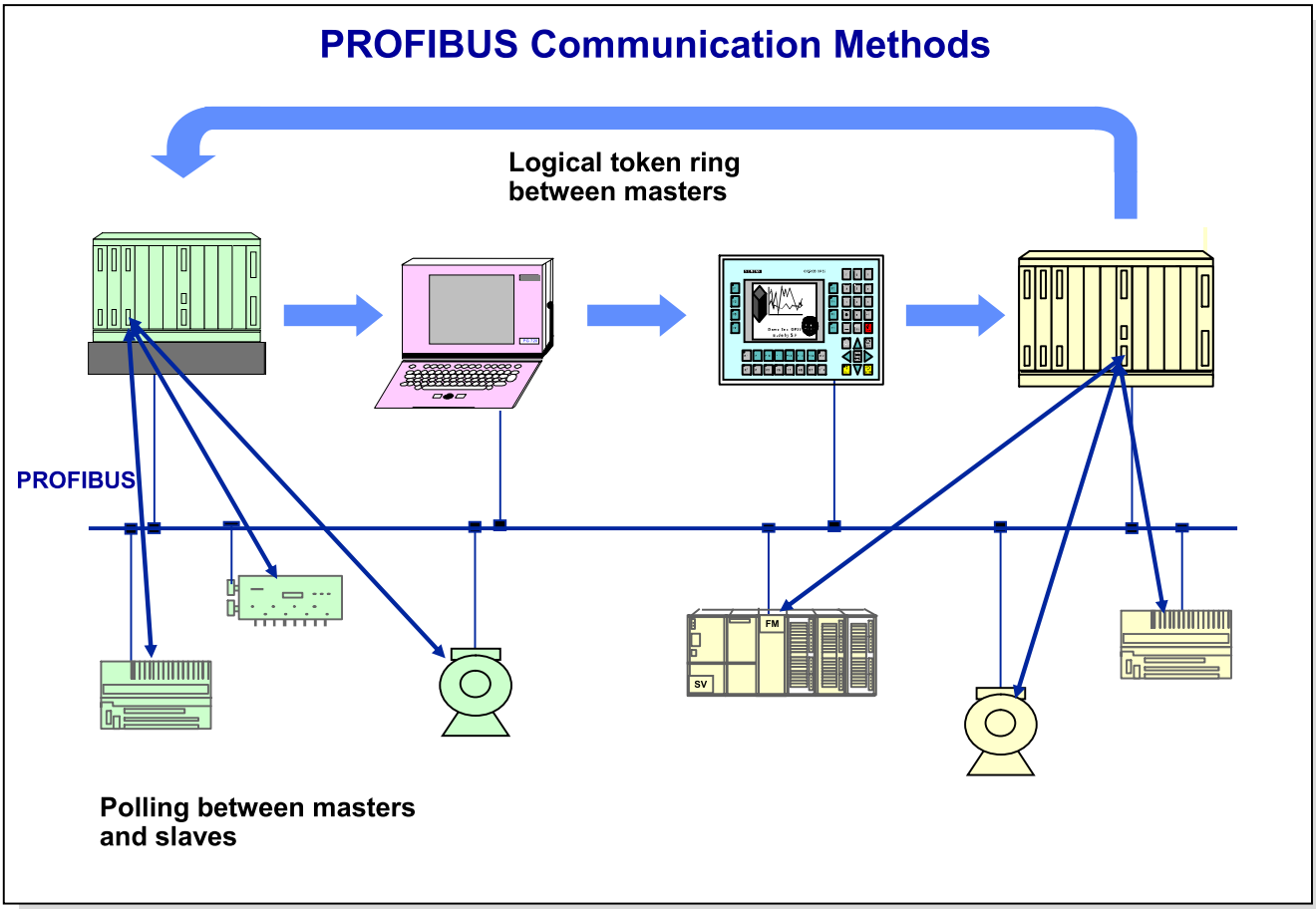
PROFIBUS makes a distinction between masters and slaves.

The PROFIBUS masters are in charge of data traffic on the bus. A master can send messages without being requested to do so, provided it is in possession of the token that entitles it to access the bus.

Masters are also referred to in the PROFIBUS protocol as active nodes.

Slaves

PROFIBUS slaves are simple I/O devices, such as actuators, sensors, transducers, etc. They do not receive the token, that is, they can only acknowledge the receipt of messages or send messages (data) to a master on request. Slaves are also referred to as passive nodes.



Bus Access Control

The bus access control method determines when a node can send data. It is essential that only one node has the right to send data at any one time.

The PROFIBUS protocol caters for two basic requirements placed on a bus access control method:

- For communication between complex stations of equal status (masters), it must be ensured that each of these stations has sufficient opportunity of dealing with its communication tasks at defined intervals.
- For communication between a complex master and the simple I/Os associated with it (slaves), cyclic, real time-oriented data exchange must be implemented with as little overhead as possible.

The PROFIBUS bus access control method therefore employs token passing for communication between the complex masters and the master-slave principle for communication between the masters and the simple I/O devices (slaves).

Token Passing Method

The token passing method ensures that the right to access the bus (token) is allocated within a precisely defined time.

The token, a special message frame that passes the right to send from one master to the next, must be given once to each master in turn within a maximum token circulation time.

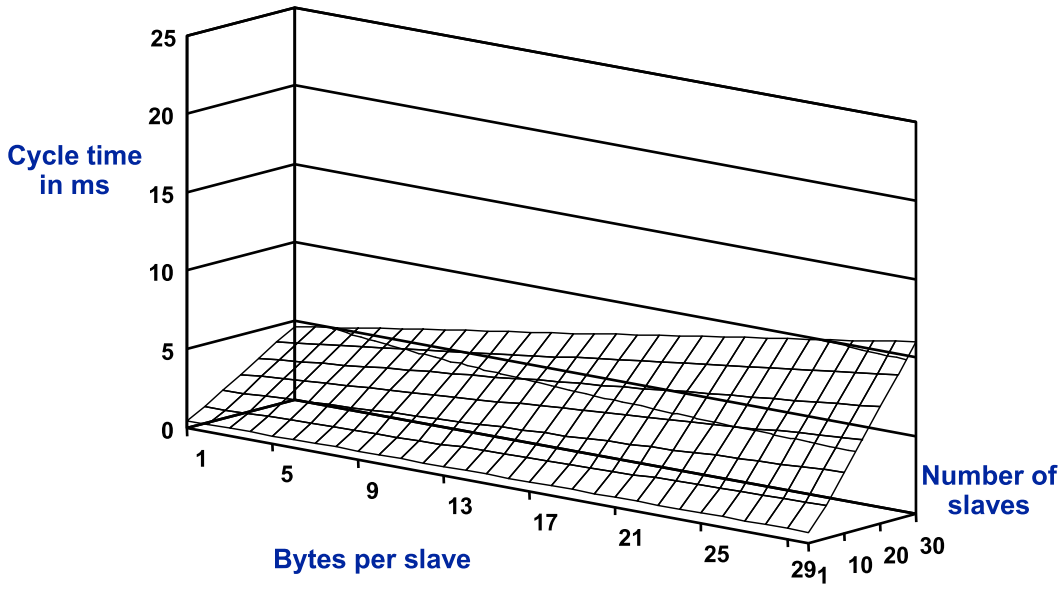
Master-Slave Principle

The master-slave principle enables the master (active node) that is currently in possession of the token to address the slaves assigned to it (passive nodes).

The master can send messages (user data) to the slaves or fetch messages (user data) from the slaves.

Bus Cycle Time of a PROFIBUS-DP Mono-Master System

Bus run times at 1.5 MBaud



SchnittStellenCenter Fürth

Date: 6/30/2011
File: PRO2_11E.4

PROFIBUS-DP

The PROFIBUS-DP protocol is designed for fast exchange of data at the sensor/actuator level. At this level central control units, such as PLCs, communicate with distributed input and output devices via a high-speed serial connection. Data exchange with these distributed devices is mainly cyclic.

The central controller (master) reads the input data from the slaves and writes the output information to the slaves. The bus cycle time must be shorter than the PLC scan time.

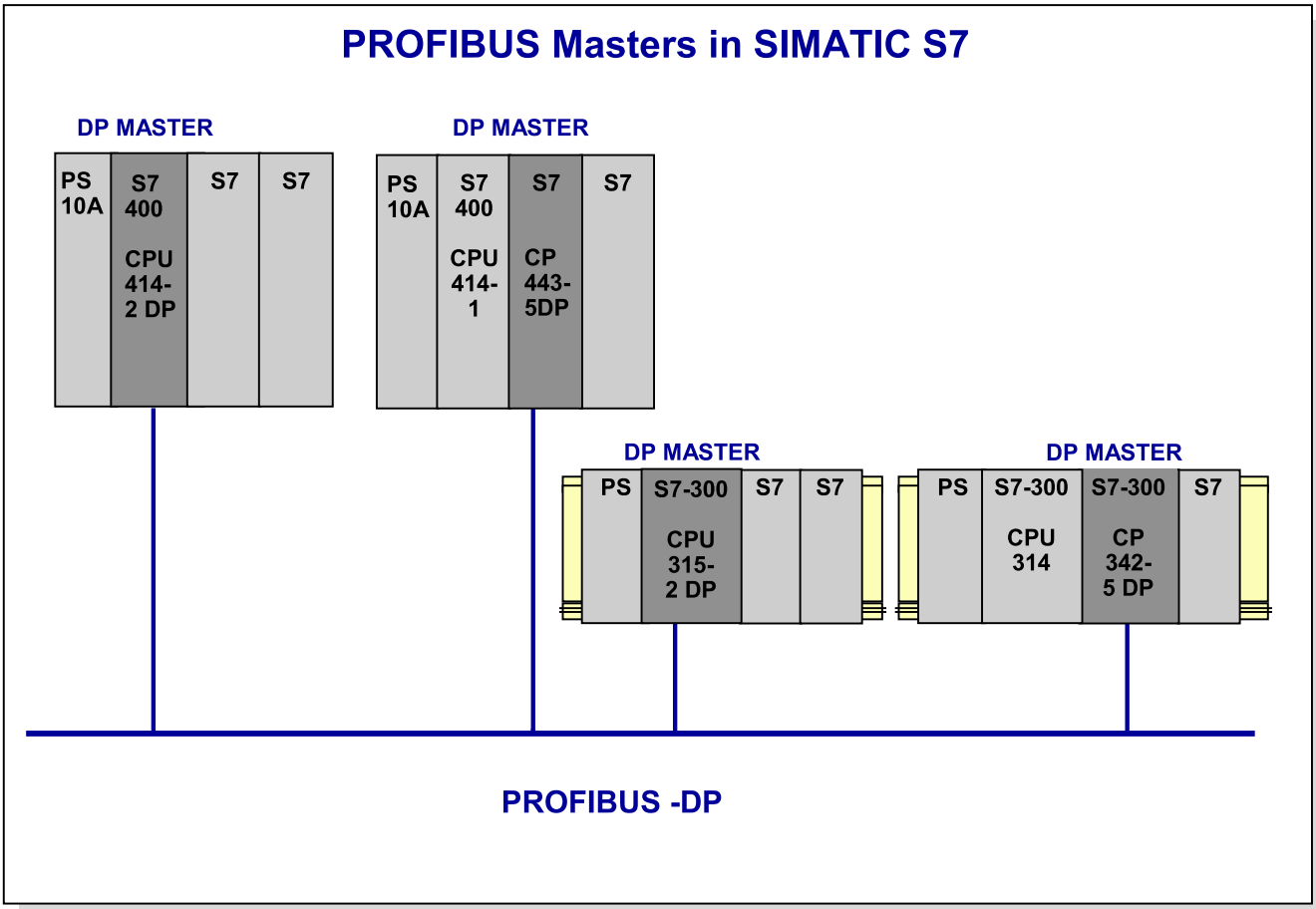
Note

The PROFIBUS-DP protocol cannot be used for exchanging information between masters.

Speed

For the transmission of 512 bits of input data and 512 bits of output data divided between 32 nodes, PROFIBUS-DP takes approx. 6 ms at a transmission speed of 1.5 Mbit/s and less than 2 ms at 12 Mbit/s.

The far superior speed of this protocol over that of the PROFIBUS-FMS protocol is mainly due to the fact that the input and output data are transferred in one message cycle using the layer 2 SRD service (Send and Receive Data).



SIMATIC S7

Date: 6/30/2011
File: PRO2_11E.5

SITRAIN Training for Automation and Drives

Overview

SIMATIC S7/M7 continues the trend towards distributed automation by integrating distributed I/O into the automation system. The innovative automation technology of SIMATIC S7/M7 forms an ideal partnership with the internationally established PROFIBUS-DP/PA fieldbus and the distributed I/O stations.

PROFIBUS Master

The S7-300 and S7-400 PLCs can be connected to the PROFIBUS as masters either via CPUs with integral PROFIBUS-DP interface or via communications processors (CPs).

The CPUs with integral PROFIBUS-DP interface allow you to configure distributed automation systems with transmission speeds of up to 12 Mbaud.

Integration

The total system integration of the PLC and distributed I/Os has the following advantages for the user:

- Uniform configuring: You configure both central and distributed I/Os with STEP 7. This means a uniform configuring tool for the user, regardless of the type of automation solution
- Centralized and distributed programming: You program the PLC with STEP 7
  irrespective of the type of configuration. This means that you can write programs without regard for the final hardware configuration.
- Full system performance, whether in a centralized or distributed configuration: SIMATIC S7/M7 offers powerful system support. This includes
  software-parameter assignable I/Os, a wide range of diagnostics facilities and easy-to-connect function modules.
- Programming, testing and startup via PROFIBUS-DP: Distributed automation
  structures call for distributed startup facilities. With STEP 7 you can program,
  test and start up the central PLC from the field just as easily as you can via the central programming port on the CPU.

Available DP Slaves


ET 200M


ET 200U

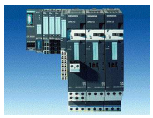
Modular slaves consisting of an interface module and modules from the S7-300 spectrum (ET 200M) or S5 spectrum (ET 200U).


ET 200B


ET 200L

Small, compact I/O stations (degree of protection IP 20) with integrated input and output channels


ET 200X


ET 200S

Interface module plus input/output modules, load feeders, etc.
Degree of protection: ET 200X: IP 65/67, ET 200S: IP 20


CPU 215


CPU 315-2 DP

Intelligent DP slaves from the S7-200 and S7-300 spectrum for data preprocessing


CPU 316-2 DP


CPU 318-2 DP


CP 342-5

SIMATIC S7

Date: 6/30/2011
File: PRO2_11E.6

SITRAIN Training for Automation and Drives

Modular Slaves ET 200M

The ET200M consists of an IM153-1 interface module that is connected to an S7/M7-PROFIBUS master. All S7-300 modules addressed via the P bus can be inserted in the ET 200M.

Maximum address space per ET 200M: 128/128 bytes each for inputs/outputs with a maximum of 12 Mbaud

Compact Slaves ET 200L and ET 200B

The ET 200L and ET 200B both consist of a terminal block and an electronics block. There are electronics blocks with digital and analog channels. The ET 200L is used where few inputs and outputs are required with baud rates up to 1.5 Mbaud.

The ET 200B is used where there is a limited mounting depth. The maximum baud rate is 12Mbaud.

Compact Slaves ET 200C

The compact ET 200C in the high degree of protection IP66/IP77 is of rugged design and meant for rough industrial operation. (Can also be used in the open). With a maximum baud rate of up to 12Mbaud for digital inputs/outputs and up to 1.5 Mbaud for analog inputs/outputs.

Modular Slaves ET 200X

The ET 200X is a compact I/O station with the high degree of protection IP 65/IP 67 and consists of a basic module and expansion modules (input/output modules, AS-interface master, load feeder modules, pneumatic modules, SITOP power supply, for example).

Modular Slaves ET 200S

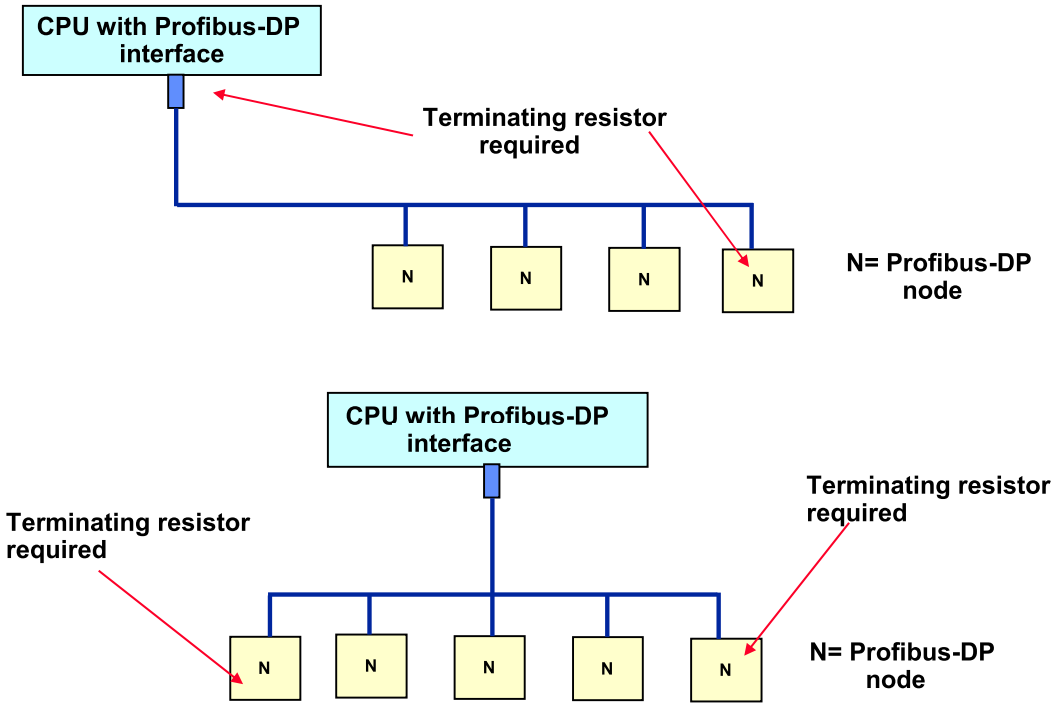
The ET 200S is a distributed I/O station with degree of protection IP 20. Its highly modular design enables it to be adapted quickly and perfectly to any application.

The ET 200S consists of a PROFIBUS-DP interface module, digital and analog electronics modules, technological function modules (counting, position decoding, for example) and load feeders.

Intelligent Slaves

For ex., CPU 315-2, CP 342-5 or S5-95-PROFIBUS with slave functionality

PROFIBUS - DP Terminating Resistor



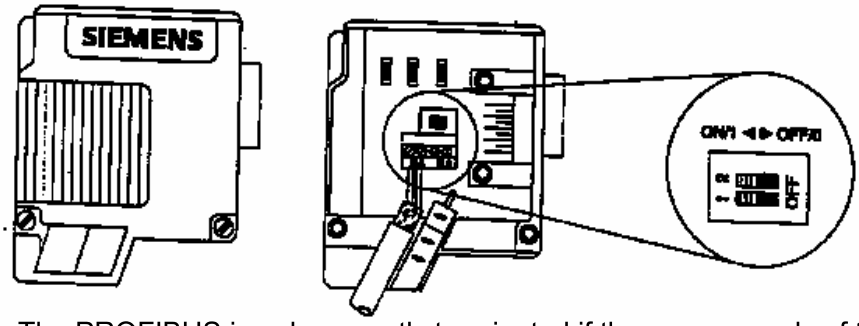
SIMATIC S7

Date: 6/30/2011
File: PRO2_11E.7

SITRAIN Training for Automation and Drives

Setting the Terminating Resistor

Switch on the terminating resistor on the first and last nodes of a segment. To do this, you open the cover of the bus connector and set the switch to the ON position (see diagram).



The PROFIBUS is only correctly terminated if the power supply of the node in which the terminating resistor is inserted is actually switched on. If this is not always the case, the PROFIBUS can also be terminated with an active RS485 terminating resistor (6ES7972-0DA00-0AA0). The terminating resistor then receives a permanent power supply separate from that of the other I/O components or supplied to it before the I/Os.

Termination of the bus system enables the nodes (ET 200L, for example) to be connected and disconnected at will, without causing malfunctions

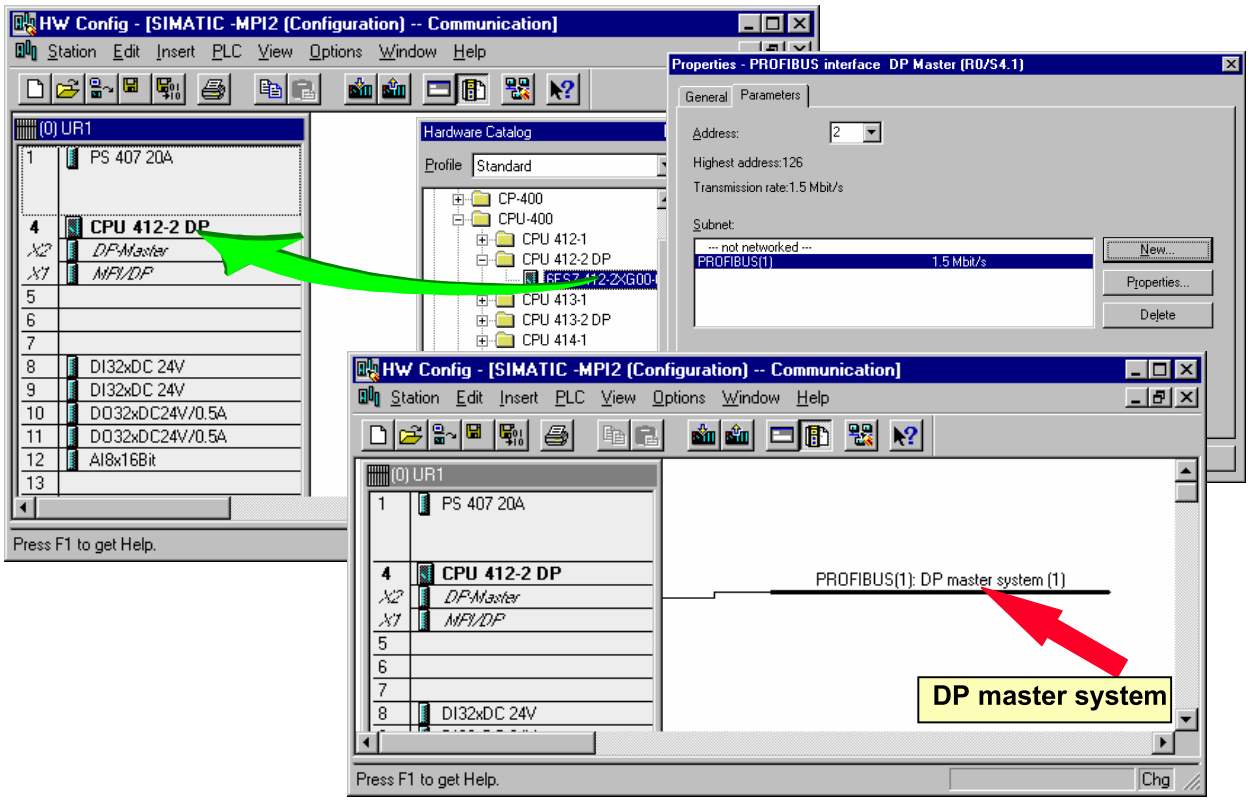
Cable Lengths

The maximum Profibus segment lengths depend on the baud rate:

Baud rate	Segment length
9.6 to 187.5 Kbaud	10000 m
500 Kbaud	400 m
1.5 Mbaud	200 m
3 to 12 Mbaud	100 m

The maximum segment length for MPI is 50 m. Up to 9 repeaters can be connected in a row.

Configuring a DP Master System



Date: 6/30/2011
File: PRO2_11E.8

Distributed I/O

All master systems consisting of DP master and DP slaves that are connected using a bus cable and that communicate via the PROFIBUS-DP protocol are designated as distributed I/Os.

DP Master

You can install the following as DP master:

- S7-CPU with integrated DP master interface (e.g. CPU 414-2, etc.)
- interface submodule, that is assigned to an M7-CPU/M7-FM
- CP in connection with a CPU (e.g. CP 443-5, etc.)

Setting Up DP Master

In order to set up a master system, proceed as follows:

1. Select a DP master from the "Hardware Catalog" window.
2. Using Drag&Drop, drag the module into an allowed line of the rack.

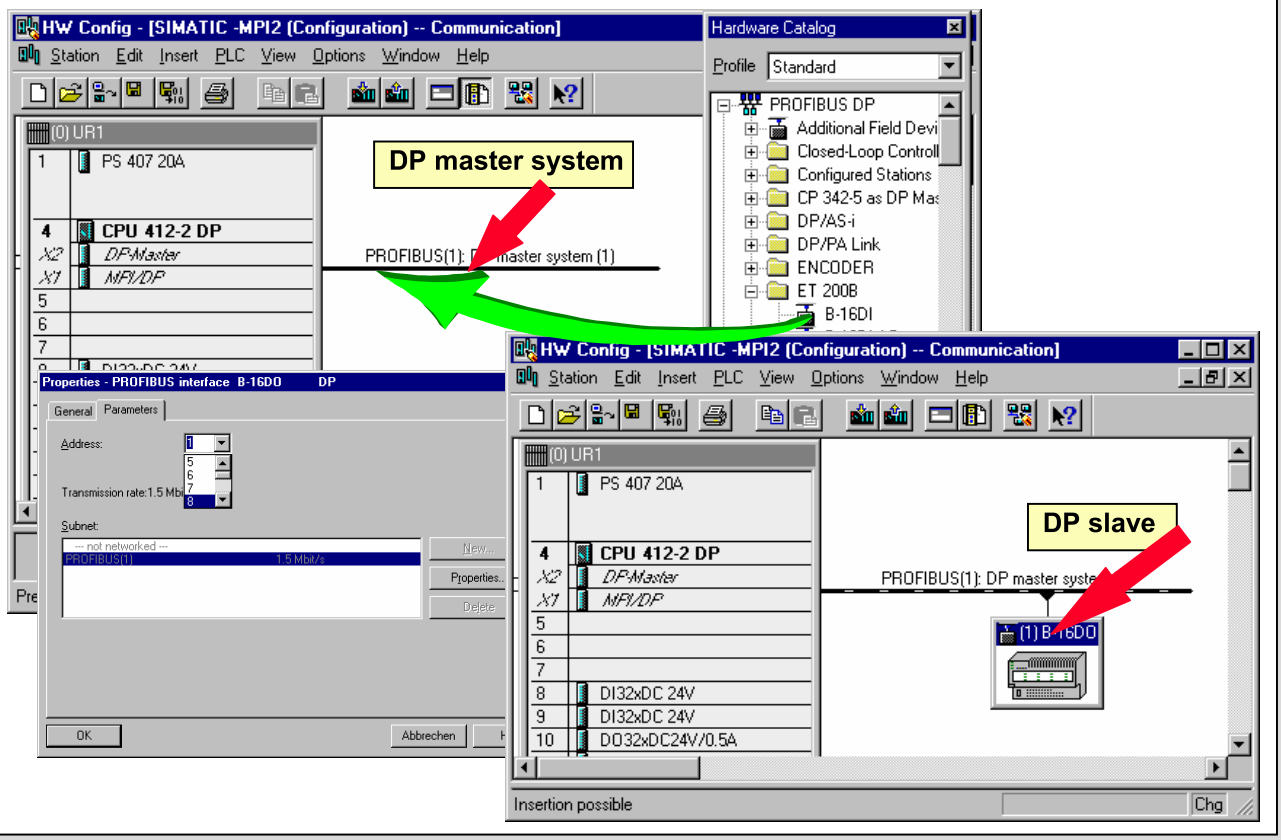
   The dialog box "Properties - PROFIBUS Nodes" is opened. In this dialog you can define the following properties:

   - set up a new PROFIBUS subnet or select an existing one.
   - set the properties of the PROFIBUS subnet (baud rate, etc.).
   - define the PROFIBUS address of the DP master.
3. Acknowledge the settings with "O.K.". The following symbol appears: for the DP master system. This symbol is used as a "hanger" for the DP slaves.

Note

You can have mono-master operation as well as multi-master operation on one PROFIBUS-DP subnet. In mono-master operation, only one DP master is operated on one PROFIBUS subnet. In multi-master operation, several DP masters with their respective master system are operated on one PROFIBUS subnet.

Configuring Compact and Modular DP Slaves



SIMATIC S7

Date: 6/30/2011
File: PRO2_11E.9

SITRAIN Training for Automation and Drives

DP Slaves

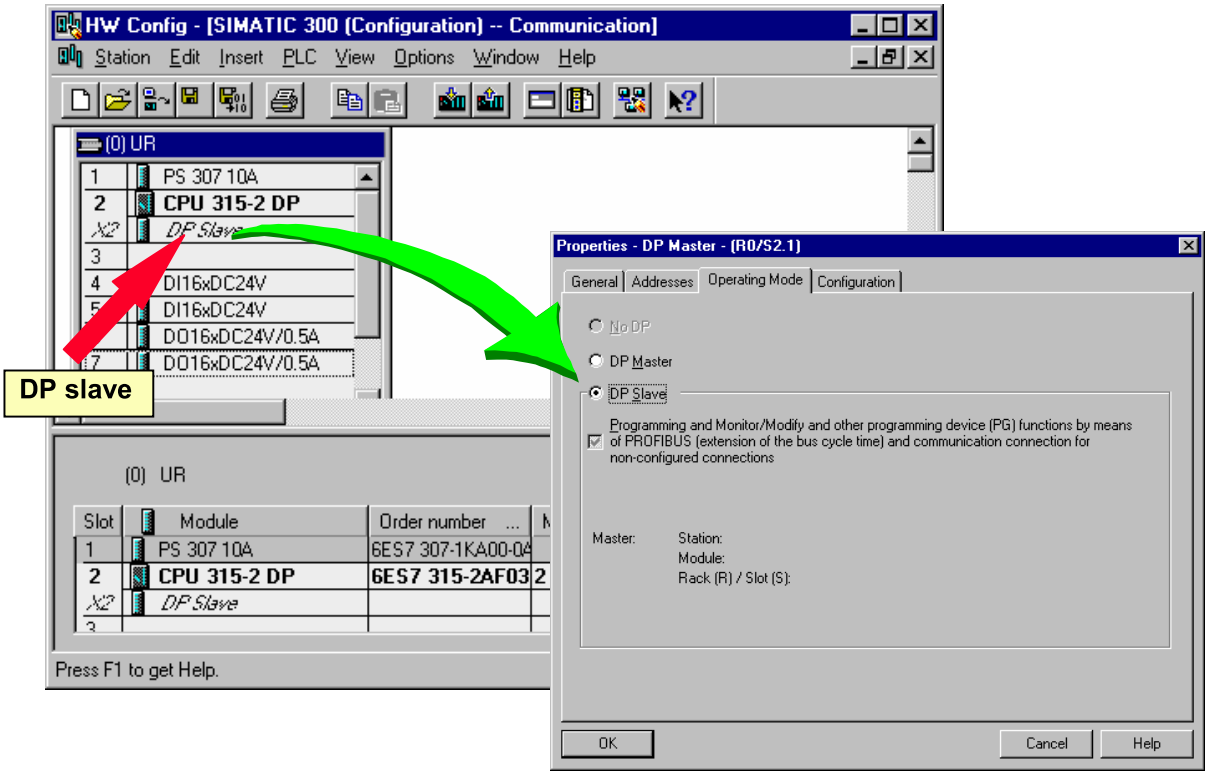
- Modules with integrated digital/analog inputs and outputs (compact DP slaves, ET200B for example).
- Interface modules with assigned S5 or S7 modules (modular DP slaves, ET200M for example).
- S7-200/300 stations with modules that support the "Intelligent Slave" function (CPU 215-DP, CPU 315-2 for example).

Selecting DP Slaves

In order to configure a DP slave, proceed as follows:

1. From the "Hardware Catalog", select the desired compact DP slave (ET200B for example) or the interface module (IM153 for ET200M for example) for a modular slave
2. Drag the "slave" onto the symbol for the master system
   The dialog box "Properties - PROFIBUS Nodes" is opened. Here you can set:
   - properties of the PROFIBUS subnet (baud rate, etc.).
   - the PROFIBUS address of the DP slave.
3. Acknowledge the settings with "O.K.". The Editor attaches an icon to the symbol.
4. For a modular DP slave, you now insert the desired modules from the "Hardware Catalog" into the configuration table.
   The addressing and parameter assignment of the modules then takes place in the same way as in the central configuration.

Configuring Intelligent DP Slaves (CPU 315-2, for example)



Intelligent Slaves

The chief characteristic of an intelligent DP slave is that its input/output information is not made directly available to the DP master. Rather, a preprocessing or an after processing by the user program takes place first.

With an intelligent DP slave, the DP master does not access the inputs/outputs of the intelligent DP slave, but rather the address area of the “preprocessing” CPU. The user program of the preprocessing CPU must take care of the data exchange between the address area and the input/output area.

Note

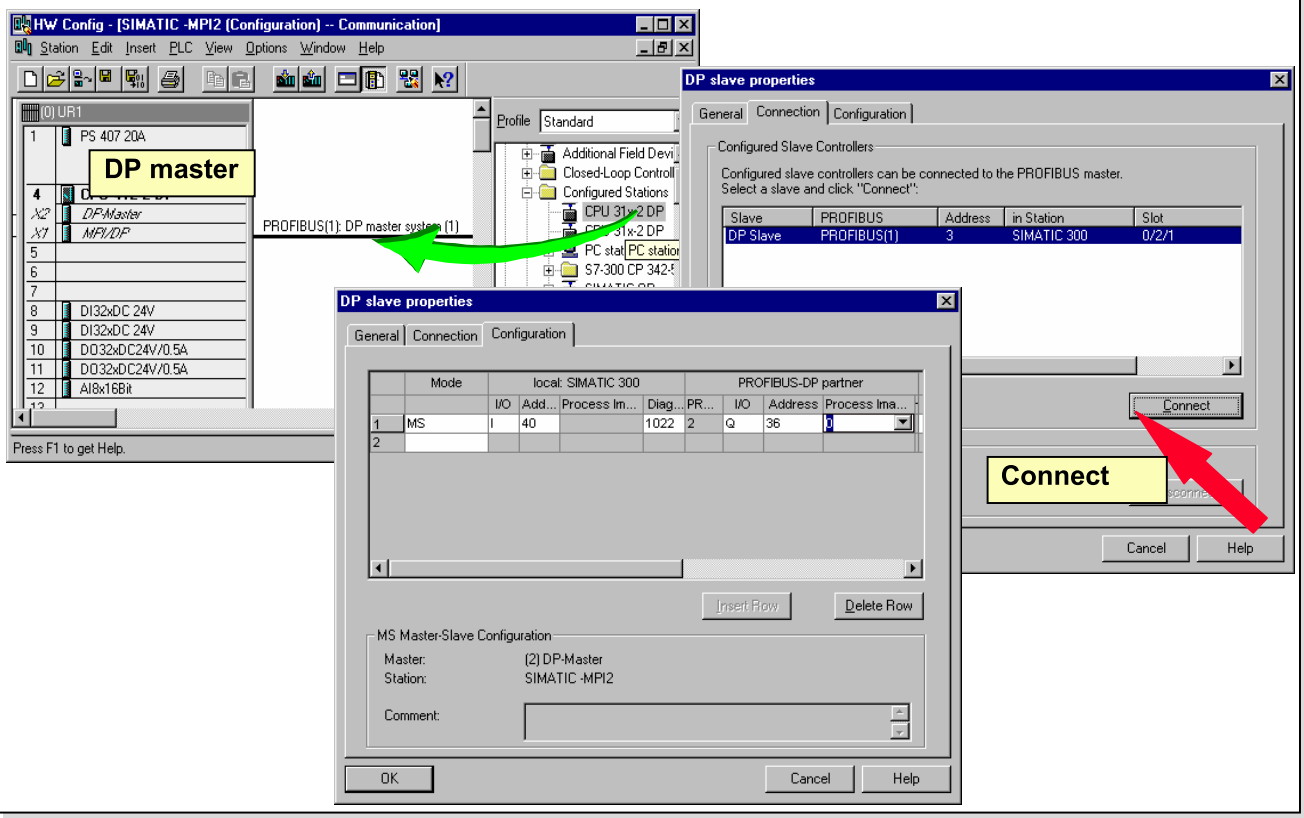
An intelligent DP slave (CPU 315-2 DP, for example) cannot simultaneously be configured as a DP master and as a DP slave. A CPU 315-2 DP configured as a DP slave cannot simultaneously be a DP master for other DP slave stations.

Configuring DP Slaves

In order to configure a CPU 315-2 DP as an intelligent DP slave, proceed as follows:

1. Insert an S7-300 Station into your project.
2. Open the HW Config Editor by selecting the station and then double-clicking on the symbol "Hardware".
3. Insert a CPU 315-2 DP from the Hardware Catalog into the configuration table.
4. Double-click on the line X2 in the configuration table. The dialog "Properties - DP Master" is opened.
5. Activate "DP Slave" in the "Operating Mode" tab.
6. Then select the rest of the DP parameters such as DP Address and PROFIBUS network assignment in the tab: "General -> Properties".
7. Confirm all the settings with "O.K." and save the hardware configuration.

Inserting Intelligent DP Slaves in a Master System



Inserting an Intelligent DP Slave

In order to insert a CPU 315-2 DP as an intelligent DP slave into a master system, proceed as follows:

1. In your project, create a DP master-capable station (S7-400, for example) with a DP master system.
2. Using drag & drop, drag the CPU 31x-2 DP onto the master system from the "Hardware Catalog" (folder: PROFIBUS -> Configured Stations).

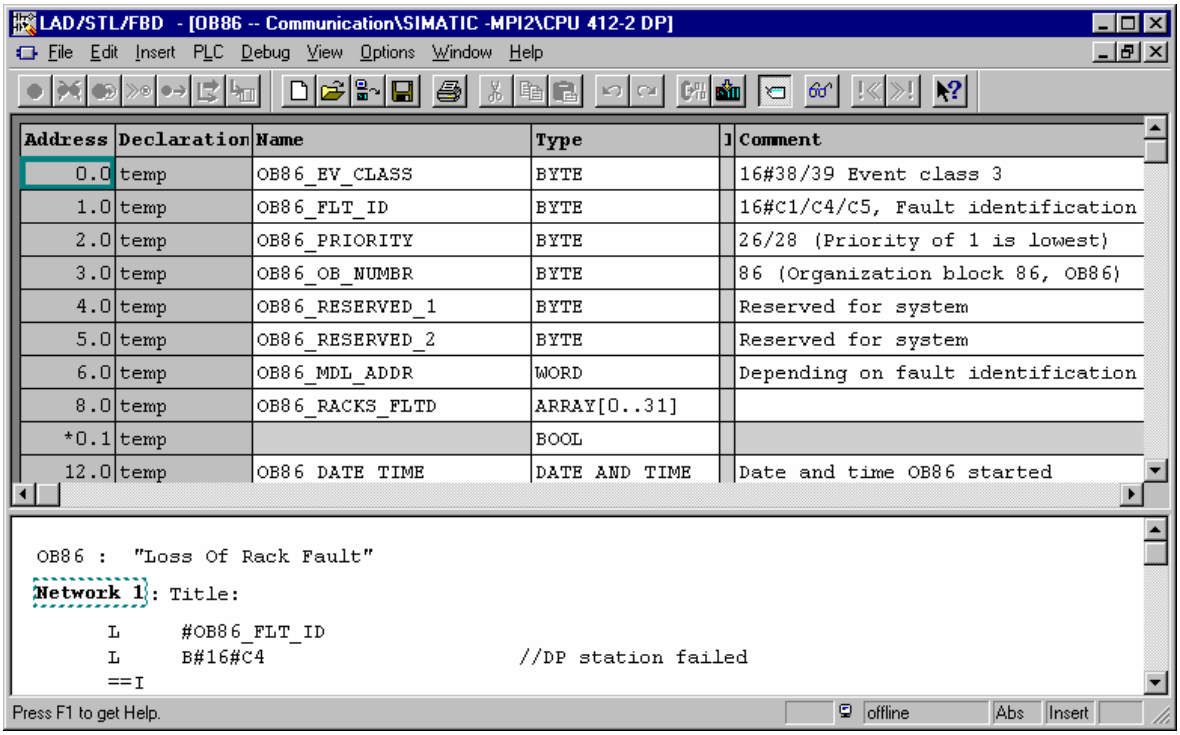
   The dialog "DP slave properties" is opened.
3. In the tab "Connection", select the desired DP slave station from the list of Configured Slave Controllers and click on the button "Connect".
4. In the tab "Configuration", configure the input/output ranges that are to be exchanged between the DP master and the DP slave.

   The input ranges of the DP master are the output ranges of the DP slave and vice versa.
5. Confirm the settings with "OK". The data preprocessing CPU 315-2 DP is now assigned to the DP master system of the CPU 413-2 DP as an intelligent DP slave.

Note

For the correct power-up of the DP master and the intelligent DP slave system, the associated error OBs (OP 85, OB 86, etc.) must be downloaded into the respective CPUs.

Error/Fault Analysis in OB 86 when Slave Failure Occurs



Station Failure

The CPU operating system (CPU 315-2DP or S7-400) activates OB86, if the failure of a rack, a subnet or a distributed I/O station is detected, both in a coming or going event.

If you have not programmed OB86 and such an error occurs, the CPU goes into the STOP mode.

Variables in OB86

- OB86_FLT_ID: B#16#C4 //DP-Station failed
- OB86_FLT_ID: B#16#C5 //DP-Station faulted
- OB86_MDL_ADDR: Logical base address of the DP master (Diagnostics address)
- OB86_ RACKS_FLTD: ==> Change data type to DWORD

  Contents:

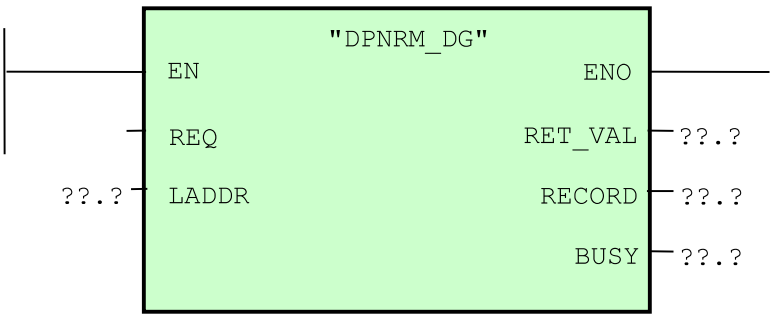
  Bit 0 to 7: Number of the DP station (PROFIBUS address)

  Bit 8 to15: DP subnet ID

  Bit 16 to 30: Logical base address of the DP-Slave (Diagnostics address)

  Bit 31: I/O identifier

Slave Diagnosis with SFC 13 (DPNRM_DG)



Parameter	Declaration	Data type	Memory area	Description
REQ	INPUT	BOOL	I, Q, M, D, L, Const.	REQ = 1: Request to read
LADDR	INPUT	WORD	I, Q, M, D, L, Const.	Configured diagnostics address of the DP slave
RET_VAL	OUTPUT	INT	I, Q, M, D, L	If an error occurs during processing of the function, the return value contains an error code. If no error occurred, RET_VAL contains the length of the data actually transmitted.
RECORD	OUTPUT	ANY	I, Q, M, D, L	Destination area for the diagnostic data read. Only data type BYTE is allowed. The minimum length of the data record to be read and of the destination area is 6.
BUSY	OUTPUT	BOOL	I, Q, M, D, L	BUSY = 1: Reading is not yet completed.

Slave Diagnosis

With SFC 13 "DPNRM_DG" you read the diagnostic data of a DP slave in the form stipulated in EN 50 170.

If no error occurs during transmission, the data read is entered into the destination area specified by RECORD (OUT 2).

You start the reading function by assigning 1 to the input parameter REQ (IN0) when you call SFC 13.

Structure of the Slave Diagnosis

The basic structure of the slave diagnostic data is shown in the following table. For further information please see the manuals for the DP slaves (for example, error numbers in the NCM-S7 manual).

Basic structure of the slave diagnosis

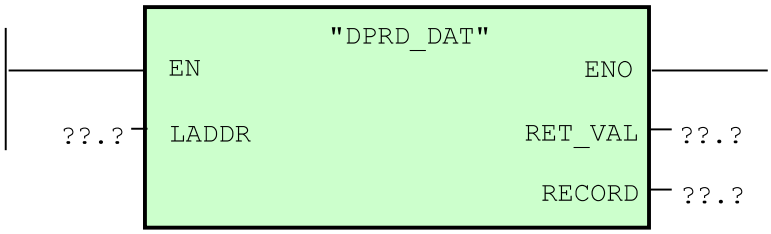
Byte	Meaning
0	Station status 1
1	Station status 2
2	Station status 3
3	Master Station number
5	Manufacturer identification (low byte)
6...	Further slave-specific diagnosis

Note

In the case of normslaves for which the number of standard diagnosis data is larger than 240 bytes and not more than 244 bytes, the first 240 bytes are transferred to the destination area and the corresponding overflow bit is set in the data.

Reading Consistent Data from DP Normslaves with SFC 14

You need the SFC 14 "DPRD_DAT" to be able to read out more than four consecutive bytes of data (consistent data).



Parameter	Declaration	Data type	Memory area	Description
LADDR	INPUT	WORD	I, Q, M, D, L, Const.	Configured starting address in the input area of the module from which data is to be read.
RET_VAL	OUTPUT	INT	I, Q, M, D, L	If an error occurs during processing of the function, The return value contains an error code.
RECORD	OUTPUT	ANY	I, Q, M, D, L	Destination area for the user data read. It must be exactly the same length as the area you have configured for the selected module with STEP7. Only data type BYTE is allowed.

Function

With SFC 14 "DPRD_DAT" you read consistent data from a DP normslave.

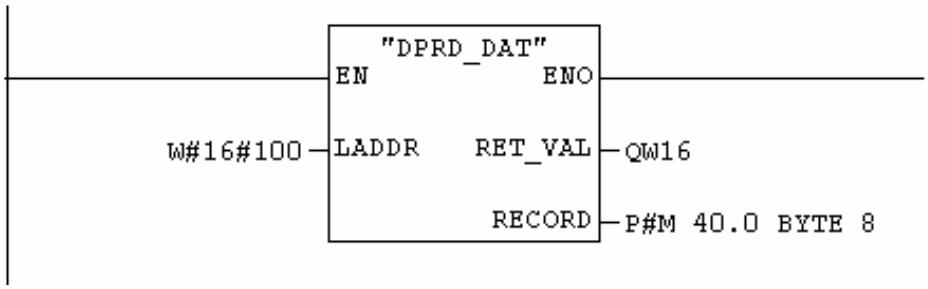
The length of the data must amount to three or more than four bytes. The maximum length depends on the CPU. You will find this in the technical specifications for your CPU. If no error occurs during transmission, the data read is entered into the destination area specified by RECORD.

The destination area must be the same length as the area you have configured for the selected module with STEP 7.

If the standard DP slave is of modular design or has several DP identifiers, you can only access the data of one module/DP identifier at a time at the configured starting address with one SFC 14 call.

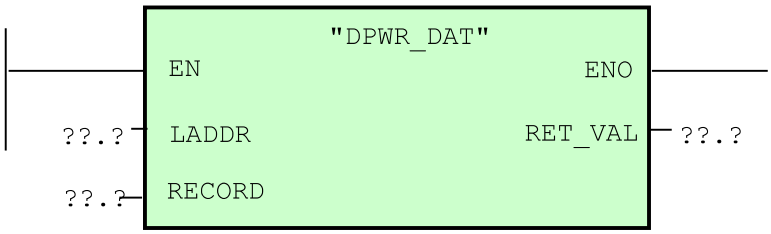
Example

```
Network 1: Read 4 analog values from starting address 256
```



Writing Consistent Data to DP Normslaves with SFC 15

You need the SFC 15 "DPWR_DAT", to be able to write more than four consecutive bytes of data (consistent data).



Parameter	Declaration	Data type	Memory area	Description
LADDR	INPUT	WORD	I, Q, M, D, L, Const.	Configured starting address in the output area of the module to which the data is to be written.
RECORD	INPUT	ANY	I, Q, M, D, L	Source area for the user data to be written. It must be exactly the same length as the area you have configured for the selected module with STEP7. Only data type BYTE is allowed.
RET_VAL	OUTPUT	INT	I, Q, M, D, L	If an error occurs during processing of the function, The return value contains an error code.

Function

With SFC 15 "DPWR_DAT" you write the data in RECORD consistently to the DP normslave addressed.

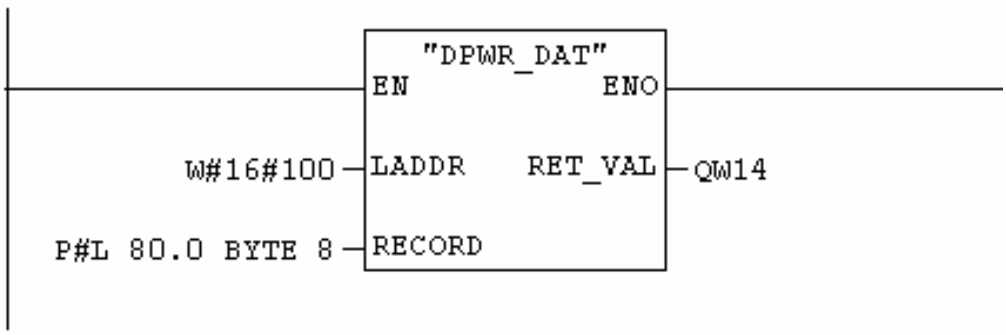
The length of the data must amount to three or more than four bytes. The maximum length depends on the CPU. You will find this in the technical specifications for your CPU. The data is transmitted synchronously, i.e. writing is completed when execution of SFC is finished.

The source area must be the same length as the area you have configured for the selected module with STEP 7.

If the standard DP slave is of modular design, you can only access one module of the DP slave.

Example

```
Network 2: Output analog values to address 256
```



Synchronizing DP Slaves with SFC 11 (DPSYC_FR)

Parameter	Declaration	Data type	Memory area	Description
REQ	INPUT	BOOL	I, Q, M, D, L, Const.	Level-triggered control parameter REQ=1: Trigger for SYNC/FREEZE job
LADDR	INPUT	WORD	I, Q, M, D, L, Const.	Logical address of the DP master
GROUP	INPUT	BYTE	I, Q, M, D, L, Const.	Group selection, Bit 0 = 1: Group 1 selected Bit 1 = 1: Group 2 selected ... Bit 7 = 1: Group 8 selected You can select several groups for one job.
MODE	INPUT	BYTE	I, Q, M, D, L, Const.	Job identifier (coded according to EN 50 170 V 3) Bit 0, 1, 6, 7:Reserved (value 0) Bit 2 = 1: UNFREEZE is executed Bit 3 = 1: FREEZE is executed Bit 4 = 1: UNSYNC is executed Bit 5 = 1: SYNC is executed
RET_VAL	OUTPUT	INT	I, Q, M, D, L	Error code. You must evaluate RET_VAL after every execution of the block.
BUSY	OUTPUT	BOOL	I, Q, M, D, L	BUSY=1: The job is not yet completed.

Description

With SFC 11 ”DPSYC_FR”, you can synchronize one or more groups of DP slaves. Do do this, you send one of the following control commands or a combination of these to the affected groups:

- SYNC (simultaneous output and freezing of output states of the DP slaves)
- UNSYNC (cancels the control command SYNC)
- FREEZE (freezing of input states of the DP slaves and reading in the frozen inputs)
- UNFREEZE (cancels the control command FREEZE)

Prerequisites

Before you send off the above mentioned control commands, you must have divided the DP slaves into SYNC or FREEZE groups with STEP 7.

What is the effect of SYNC?

With the control command SYNC, the DP slaves of the named groups are switched into the Sync mode. That is, the DP master transmits the current output data and causes the affected DP slaves to freeze the outputs. When they receive further output messages, the DP slaves merely save the output data in an internal buffer; the state of the outputs remains unchanged for the time being.

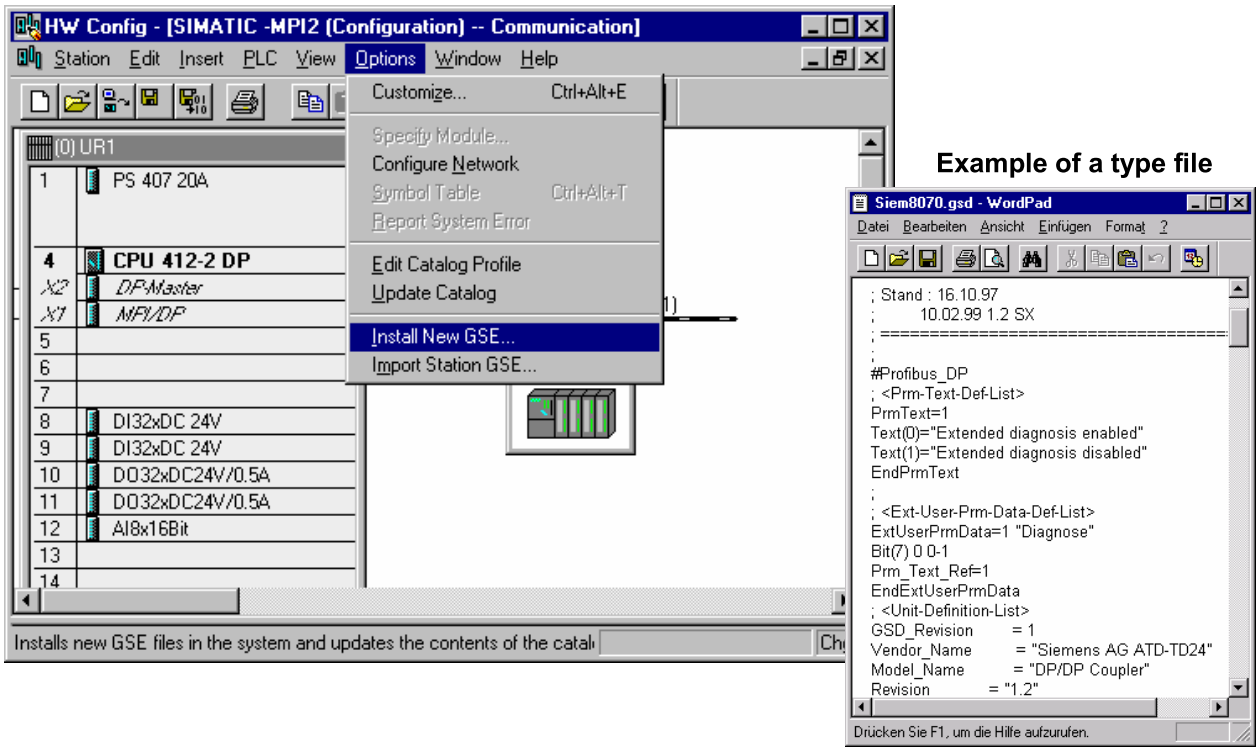
After every SYNC command, the DP slaves of the selected groups place their buffer’s output data simultaneously on the peripheral outputs to the process (simultaneous output of control signals).

What is the effect of FREEZE?

With the control command FREEZE, the affected DP slaves are switched into the Freeze mode. Every FREEZE command from the DP master causes the affected DP slaves to save the current state of the inputs simultaneously. After that, the DP master transmits the saved data into the input area of the CPU.

The inputs or outputs are only cyclically updated again when the control command UNSYNC or UNFREEZE is sent.

Later Installation of PROFIBUS-DP Slaves



Type Files

STEP 7 needs a device database file (GSD) or type file for every DP slave so that you can select the slave from the Hardware Catalog in the HW Configuration tool.

A GSD file contains all the properties of a DP slave in accordance with the PROFIBUS standard. Type files are in accordance with the SIEMENS specifications.

There is a type file for every DP slave type of the SIEMENS AG.
A GSD or type file is supplied with DP slaves from other manufacturers.

Integrating DP Slaves

You can integrate a new DP slave in the Hardware Catalog as follows:

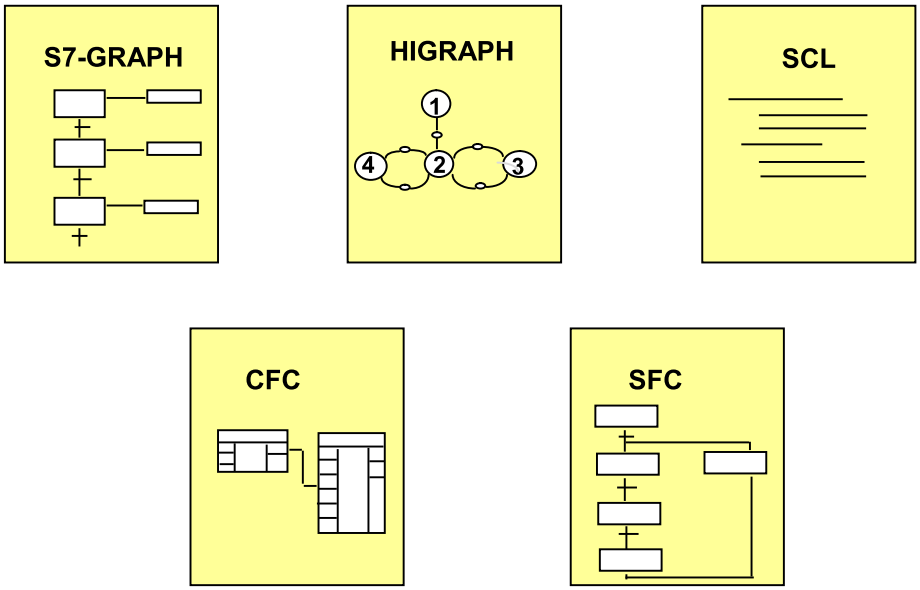
1. Select the menu options *Options -> Install new GSD*.
2. In the dialog box which then appears open the drive/directory containing the relevant GSD file.

   The slave is entered in the "Hardware Catalog" window (catalog profile "Standard" only!) under "PROFIBUS-Additional Field Devices" and is available for configuring.

When DP slaves are installed or imported in this way, the existing GSD files and symbols are not deleted completely, but are saved in the backup directory \\Step7\S7data\Gsd\Bkp[No.].

No. is a serial number which is issued automatically by STEP 7

Engineering Tools for S7/M7



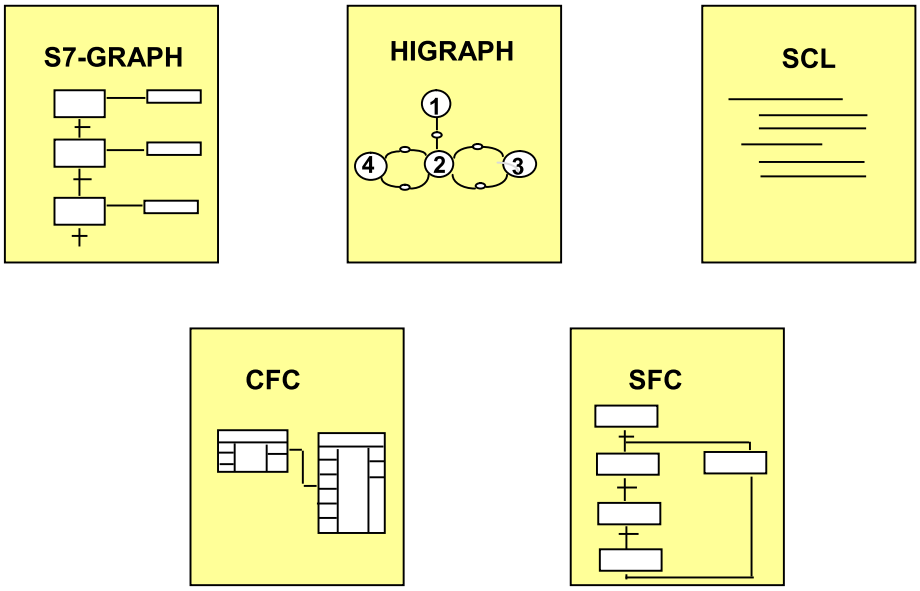
SIMATIC S7

Date: 30.06.2011
File: PRO2_12E.1

SITRAIN Training for Automation and Drives

Contents **Page**

Engineering Tools for S7/M7



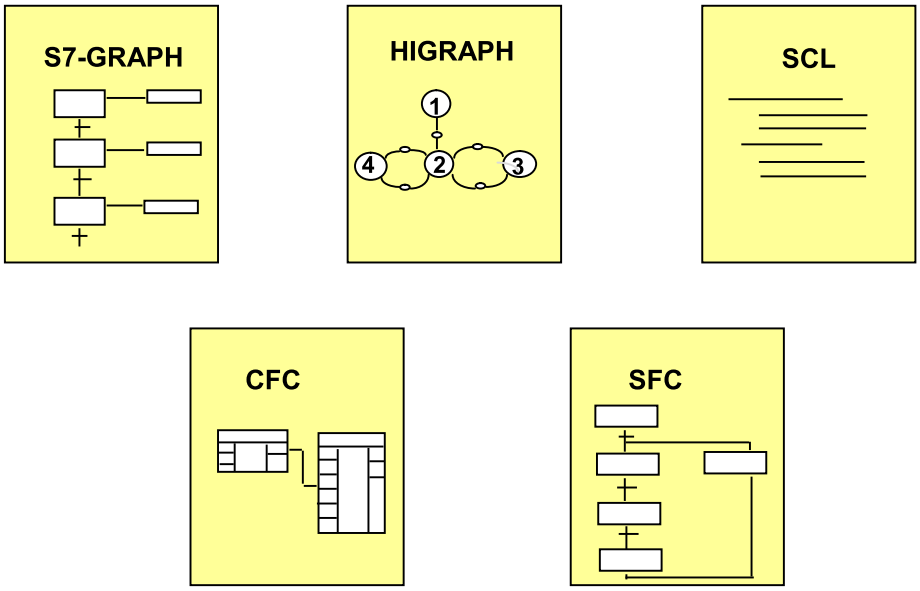
SIMATIC S7

Date: 30.06.2011
File: PRO2_12E.2

SITRAIN Training for Automation and Drives

Contents **Page**

Engineering Tools for S7/M7



SIMATIC S7

Date: 30.06.2011
File: PRO2_12E.3

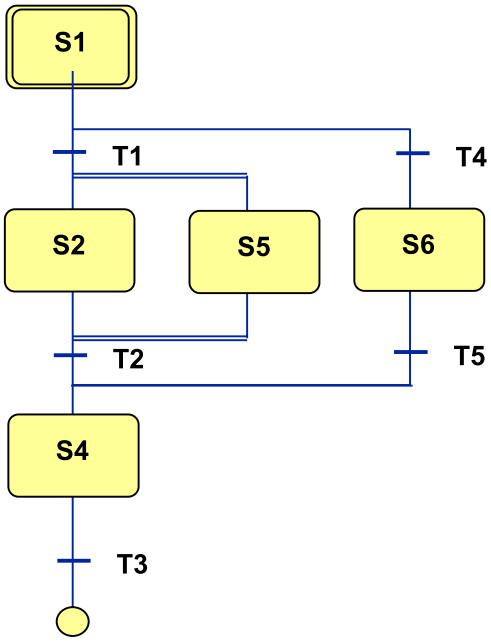
SITRAIN Training for Automation and Drives

Contents **Page**

The S7- GRAPH Software Package

Tool for Programming Sequencers

- **Compatible to IEC 61131-3**
- **Designed for the requirements of the manufacturing industry**
- **Graphical breakdown of the process into steps and transitions**
- **Steps contain actions**
- **Transitions check the step-enabling conditions**



Automating the following phases

- **Planning, configuring**
- **Programming**
- **Debugging**
- **Startup**
- **Maintenance, diagnostics**

SIMATIC S7

Date: 30.06.2011
File: PRO2_12E.4

SITRAIN Training for Automation and Drives

S7-GRAPH

With the S7-GRAPH programming language, you can clearly and quickly configure and program sequential sequences that you wish to control with an S7 PLC system.

The process is split into single steps with their own function scope. The sequence is represented graphically and can be documented with pictures and text.

The actions to be performed are defined in the single steps, the conditions for moving on to the next step are defined by transitions. The definitions of these, as well as interlocks and supervisions are written in a subset of the STEP 7 programming language LAD (Ladder Diagram).

S7-GRAPH for S7-300/400 is compatible with the sequential function chart language defined in the IEC 61131-3 standard.

Functionality

The following functions are offered:

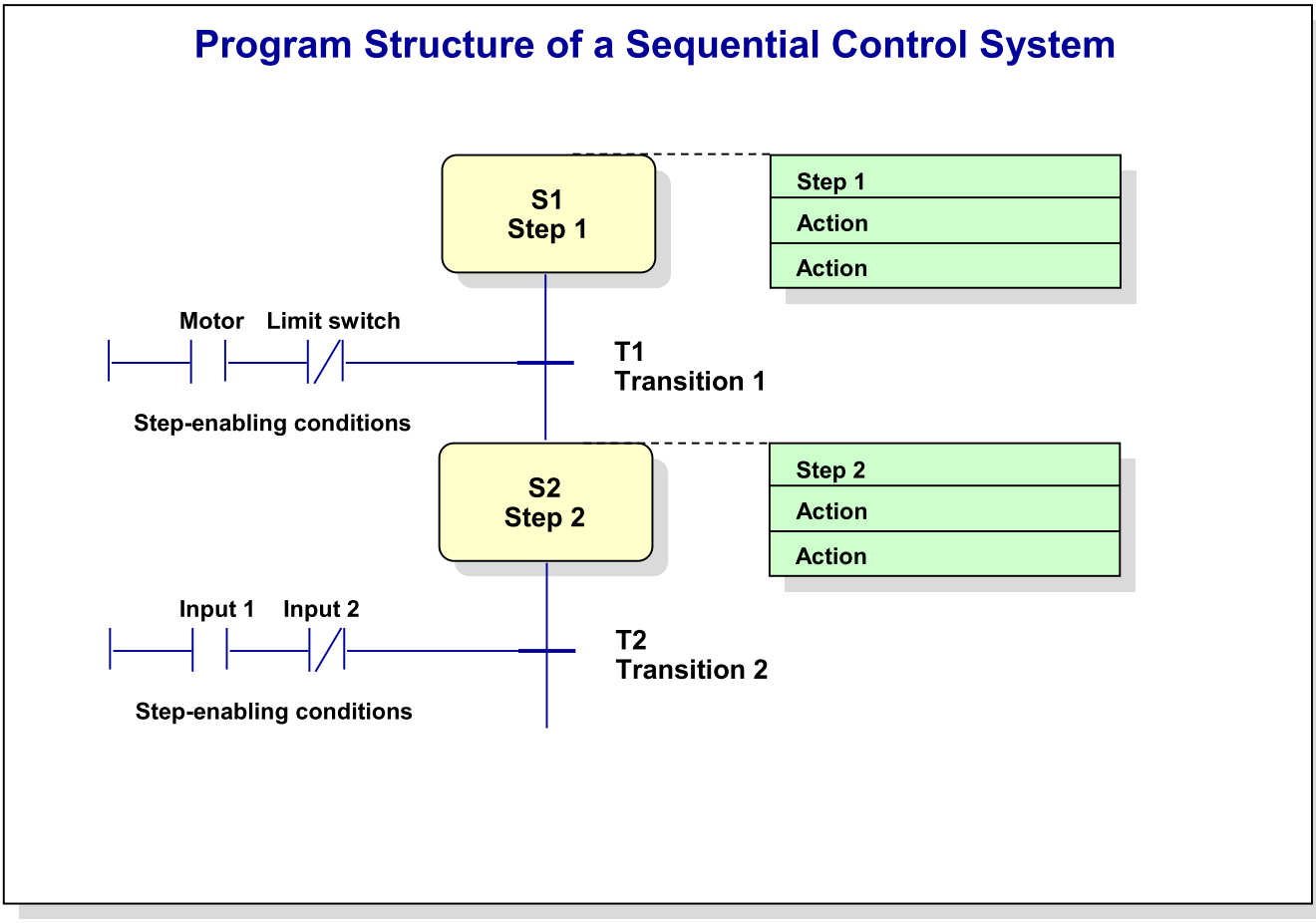
- Several sequencers (max. 8) in the same S7-GRAPH function block
- Free number assignment (1 to 999) of the steps (max. 250 per sequencer complex) and transitions (max. 250)
- Parallel branches and alternative branches (each a max. of 250)
- Jumps (also to other sequencers)
- Starting/stopping of sequencers as well as activating/holding of steps.

Test Functions

- Display of active steps or faulted steps
- Monitor Status and Modify Variable
- Switching between operating modes: manual, automatic and jogging mode

User Interface

- Overview, Single Page and Single Step representation
- Graphical distinction between interlock conditions (interlock, max. 32 conditions), actions (max. 100 per step) and supervision conditions (supervision, max. 32 conditions).



Overview

A sequential control system is a control system that executes step by step. It switches from one step to the next when the step-enabling condition is fulfilled. A characteristic of a sequential control system is therefore the subdivision of the control task into

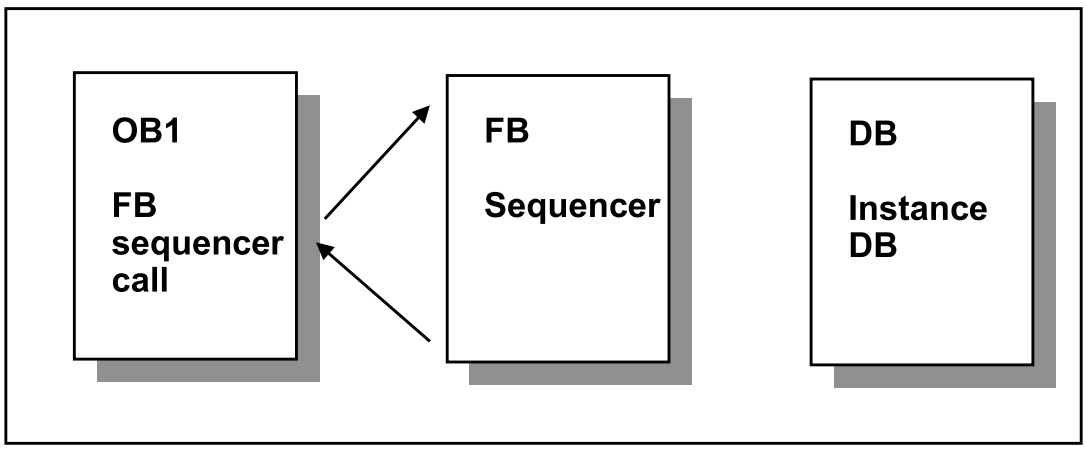
- Steps and
- Transitions (step-enabling conditions)

Sequencer

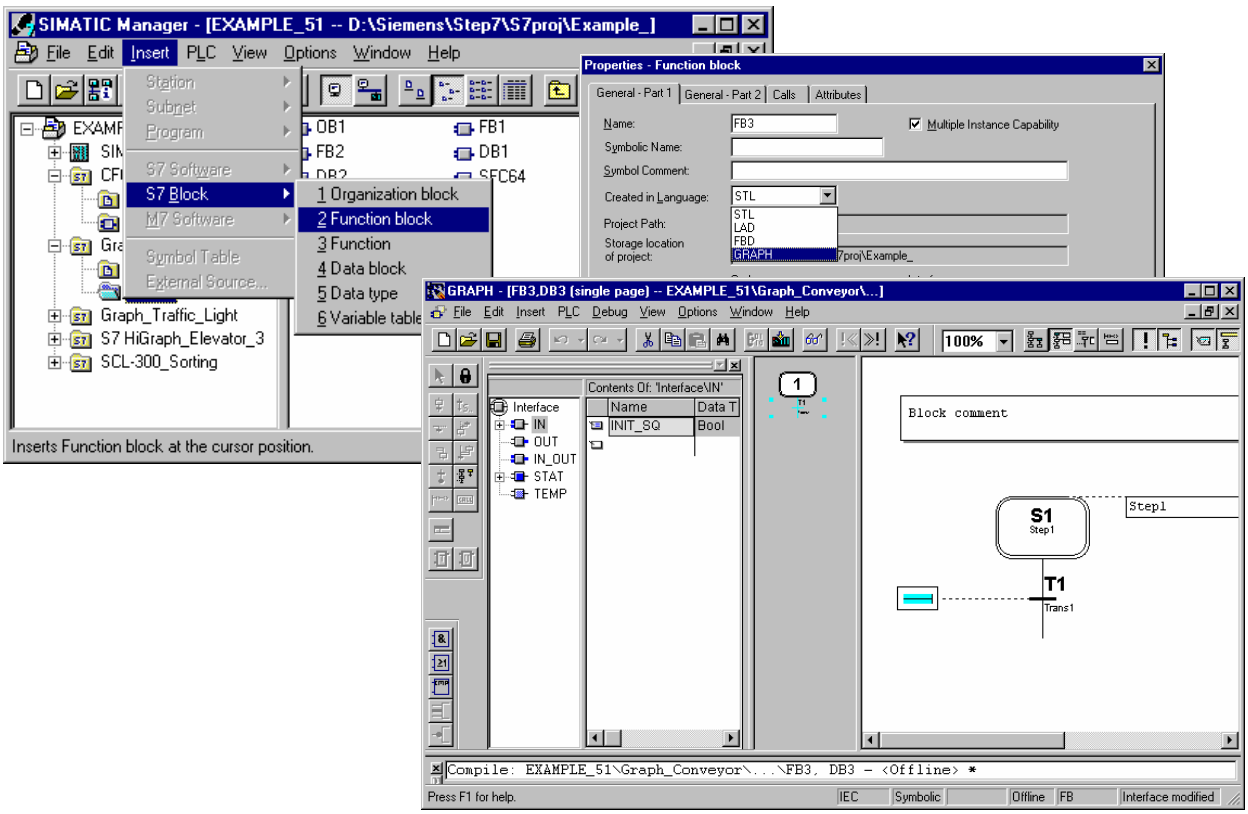
Steps and transitions form a sequencer. The sequencer is stored in an FB. An instance DB, that contains the sequencer data, is assigned to this FB.

At least three blocks are necessary for an executable program:

- the FB, that contains the sequencer(s)
- an instance DB with the sequencer data
- an organization block, function block or function containing the FB call. The parameters and the number of the instance DB are passed in the FB call.



Creating a Sequencer FB



Date: 30.06.2011
File: PRO2_12E.6

Overview

You can create S7-GRAPH FBs either with the SIMATIC Manager or with the S7-GRAPH Editor. In both cases, you must first create a project and a corresponding user program.

Creating an FB

To set up an S7-GRAPH FB with the SIMATIC Manager, proceed as follows:

1. Open the user program in which the FB is to be inserted.
2. Select the menu option *Insert -> S7 Block -> Function Block*
3. In the "Properties" dialog box enter the FB's number (e.g. FB1) and enter GRAPH as the programming language. Confirm with "OK"
4. Double-click on the inserted block. The S7-GRAPH Editor is called and the block is opened.

S7-GRAPH Editor

In entering and editing a sequencer, you are supported by the S7-GRAPH Editor with context-sensitive functions. The toolbars contain icons which give you quick access to frequently used menu options.

The following toolbars can be selected via the menu option *View -> Toolbar*:

Standard: contains functions for file handling (Open, Save, etc.) and for editing (Copy, Insert, etc.).

View: contains functions for changing the view

Sequencer: contains functions that insert sequencer elements in the program.

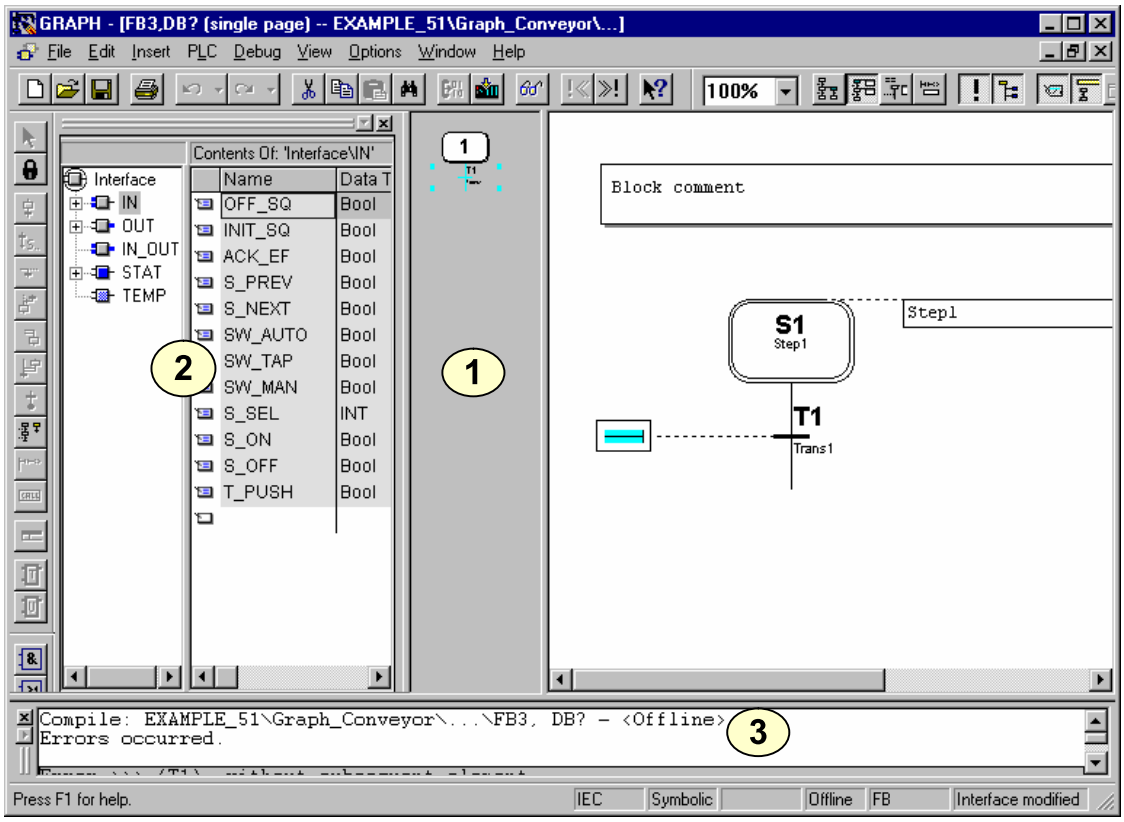
LAD/FBD :contains functions that insert LAD/FBD elements in the program.

Note

You can position the individual toolbars anywhere within the S7-GRAPH Editor.

For this, click on the grey area of the toolbar and drag the bar to the desired position with the mouse.

The User Interface of S7- GRAPH



Date: 30.06.2011
File: PRO2_12E.7

User Interface

The S7-GRAPH user interface is a combination of different project windows which can be overlaid or masked according to need. In order to optimally use the available screen size, you can enlarge, reduce or shift the project windows with the mouse.

- Editing window (1): In this window you can process the sequences of an S7 GRAPH FB.

  Within the main window of S7-GRAPH, you can open up to 10 windows simultaneously. As well, a further overview window can be overlaid using the menu options: *Window -> Split*.

- Variable declaration window (2): It is displayed to the left of the work area and is used to declare variables. The declaration sections are displayed in the left-hand pane of the variable declaration window. In the right-hand pane you can see the detailed information for the section selected in the left-hand pane.

  You can change existing parameter sets in the variable declaration window. System parameters can be deleted but cannot be processed.
  As well, you can declare your own parameters.

- Output window (3). Errors and warnings are displayed in a further window. The window is automatically opened after every compilation run (save). It can also be overlaid with the menu options *View -> Errors and Warnings*.

Declaration of Variables

In the variable declaration, you specify the local variables and parameters of the Graph FB. The variable declaration consists of the following sections:

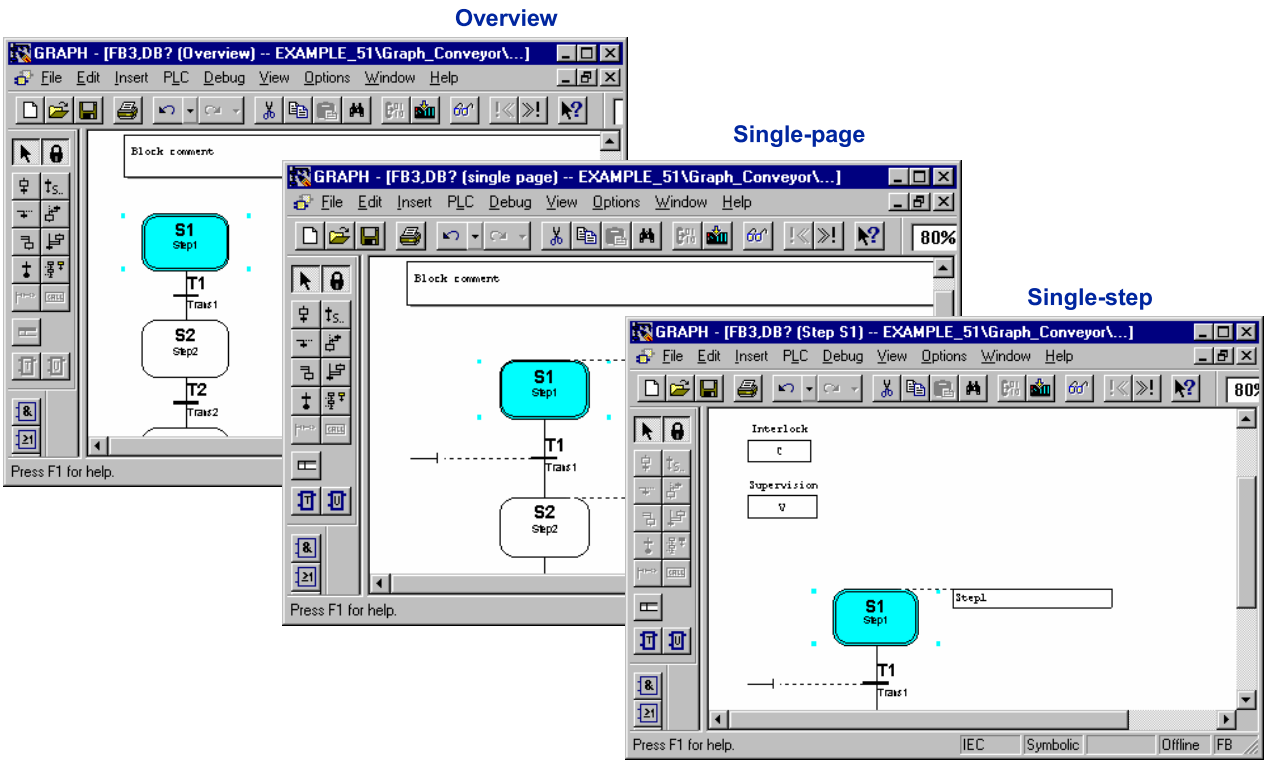
IN: contains the input parameters as well as the predefined parameters for controlling the sequence (*Options -> Block Settings -> Compile/Save*)

OUT: contains its own as well as the predefined output parameters

IN_OUT: contains the in/out parameters

STAT: contains its own as well as the predefined static variables

Sequencer Views



Date: 30.06.2011
File: PRO2_12E.8

Overview

The S7-GRAPH Editor allows you to edit a sequencer with all its elements, such as steps, transitions, branches, jumps, end-of-sequencer and comments, in different views.

You can select the view by choosing the following menu options:

View ->Overview/Single Page/Single Step/Permanent Instructions

or from the toolbar.

Overview

This view gives you an overall impression of the sequential control system as a whole. This is especially suitable for configuring the sequential structure.

Several sequencers (max. 8) in one FB appear side by side. The overview contains the block comments and the names and numbers of the steps and transitions.

Single Page

Single page view shows actions, the numbers and names of the steps, the step-enabling conditions for the transitions, the numbers and names of the transitions and the block comments.

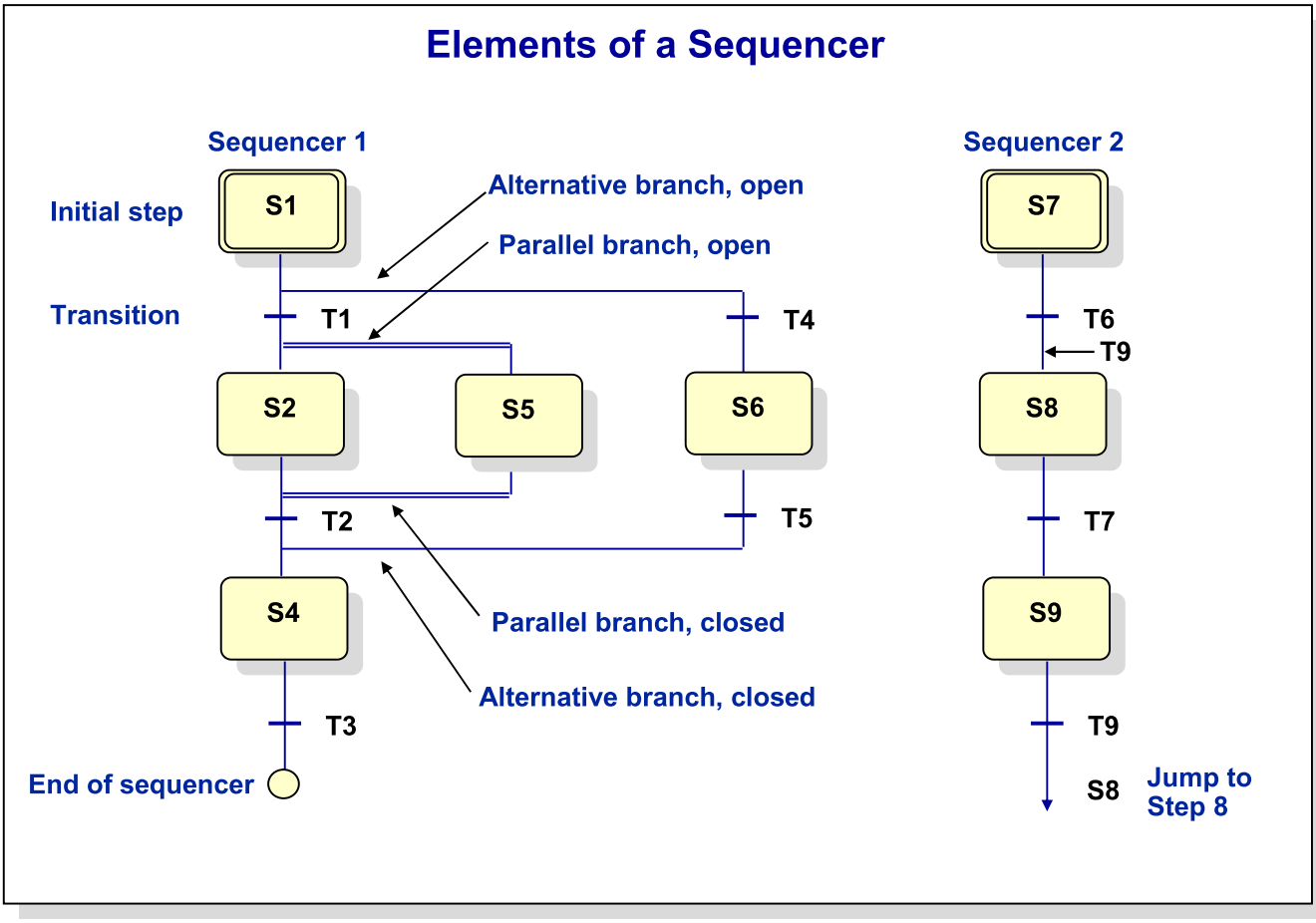
Several sequencers in one FB appear one below the other. This view is suitable both for configuring and for programming a sequential control system.

Single Step

This view shows one step with its transitions and all their contents. This view is used for programming not only actions and step-enabling conditions but also supervisions and interlocks. You can also edit step comments in this view.

Permanent Instructions

You use this window for programming "permanent instructions". Permanent instructions are conditions and/or block calls which come before or after the sequencer. They are executed once in every cycle regardless of the status of the sequencer.



Overview

When writing a sequencer, you are supported by functions for graphical programming. You can generate a sequencer structure simply, without requiring extensive programming experience, by arranging the S7-GRAPH structure elements to produce a graphical representation of the sequencer.

Structure of a Sequencer

A sequencer consists of a series of steps and transitions. The sequencer can be linear or branched.

- Within the steps, you formulate the instructions for the system.
- The transitions contain the conditions for switching from one step to the next.

The following icons represent the elements a sequencer can consist of. You can select these icons from the toolbar

Step/Transition Pair	Alternative branch, closed
Jump	Parallel branch, open
Alternative branch, open	Parallel branch, closed
End of sequencer	Insert new sequencer

There are also icons for inserting permanent instructions and for changing the insertion mode.

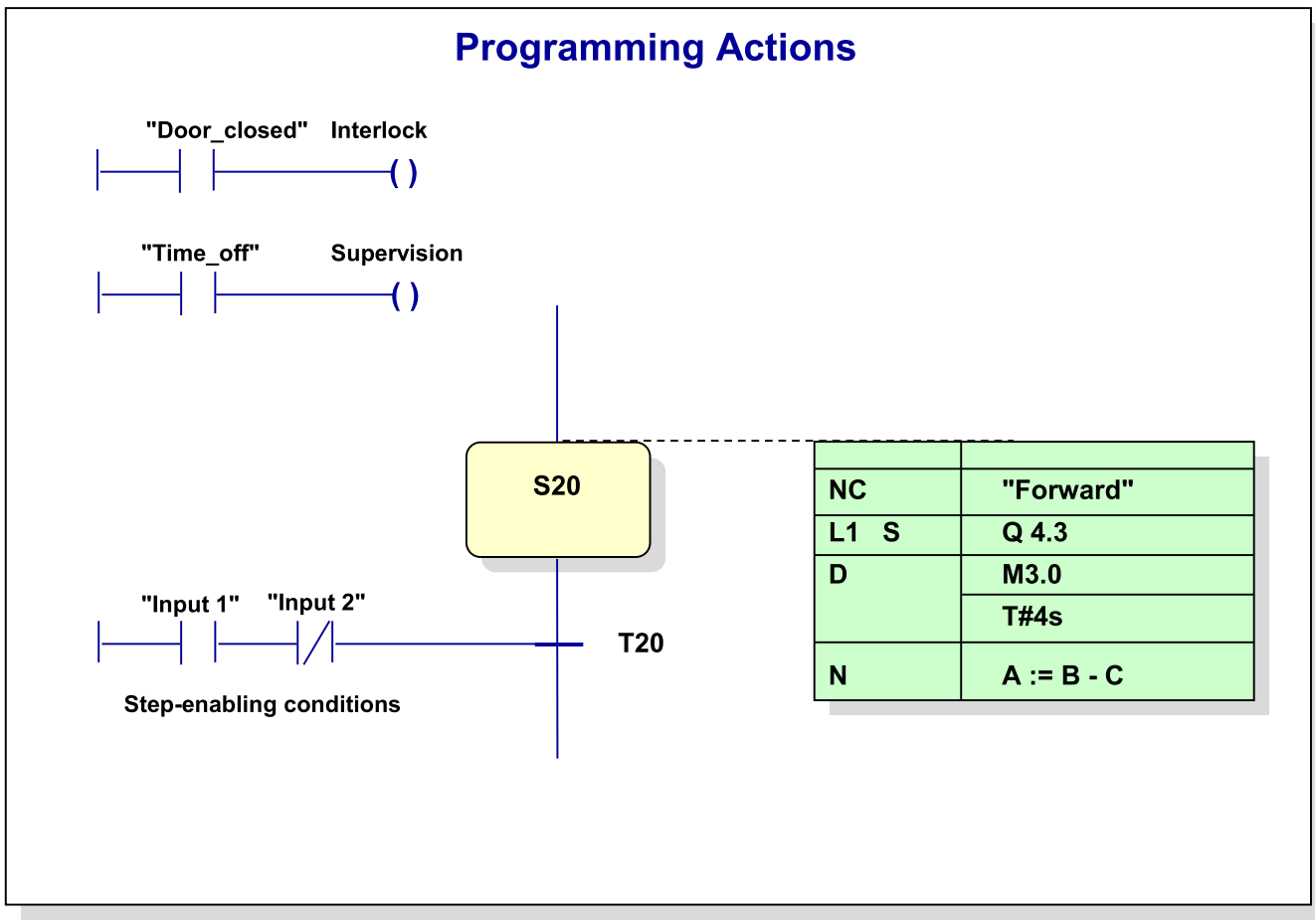
Editing

There are two editing modes for inserting elements·

Insert -> Direct: In this mode you select the position in the sequencer where you want to insert an element and then you choose the element which you want to insert.

·

Insert -> Drag&Drop: In this mode you choose the element which you want to insert and then you select the position with the mouse pointer where the element is to be inserted.



Overview

The active parts of a step are the actions that are executed by the controller in the active step. In the steps, instructions are programmed which set outputs or activate or deactivate steps in the sequencer.

An action consists of instruction and address and can be attached to a condition and/or combined with an event. The actions are divided into:

- standard actions with and without interlock (C)
- event-driven actions with and without interlock or for activating and deactivating steps
- counters, timers and arithmetic in actions

Programming Actions

You program actions in single step or single page view. To do so, proceed as follows:

1. Select the table to the right of the step and press the Tab key.
2. Now program the actions by entering the instructions which are to be executed in the table.
3. Press the Tab key once again to program a further action. The table is expanded by one line.

If you want to attach an instruction to a condition (all instructions that contain the letter C), then you must program the condition in single step view as an interlock.

Interlock Condition (C)

The interlock logic is used for conditional enabling of specific actions in a step. If the condition is fulfilled, then all actions conditioned with "C" are executed.
Advancing to the next step takes place independent of the interlock condition.

Standard Actions in a Step

Action block with simple instructions

Action_block _1	
N	M1.1
S	M1.2
R	M1.3
D	M1.4
	T#1H2M3S
L	M1.5
	T#4MS
CALL	FC1

- **N = Assign Non-stored**
- **S = Set (Stored)**
- **D = Time Delayed, Assign non-stored delayed by the time T**
- **L = Time Limited, Assign non-stored for a limited time**
- **CALL = Block call**

Overview These actions are all defined in the IEC 61131-3 standard under 2.6.4.

D, L, N The associated actions are reset as soon as the step is completed.

D, L The times can be specified as a constant or as a variable.

S, R The corresponding actions remain active even after execution of the step.

CALL Block call FBi.DBi, FCi, SFBi.DBi, SFCi: calls the specified block. After the processing of the block, the GRAPH program is further executed.

Note The addresses in the action block can be either symbolic or absolute.

Times The times must be entered in the IEC time format. This is as follows:

T#mDnHoMpSqMS

mD: m Days
nH: n Hours
oM: o Minutes
pS: p Seconds
qMS q Milliseconds

The maximum time is limited to approximately 24 D20H.

Actions Dependent on an Interlock

Action block with conditional instructions

Action_block _2	
NC	M1.1
SC	M1.2
RC	M1.3
DC	M1.4
	T#1H2M3S
LC	M1.5
	T#4MS
CALLC	FB5.DB3

Conditions

- **An action identified with a "C" (Condition) is only executed if the interlock condition for the step is true ("C" = 1).**
- **An interlock error exists if the condition is zero. The action subject to the condition C is then not executed. The step is marked and the error message "Error" is issued.**

Special Case An action in a step which has the qualifier "C" but for which no interlock is programmed is executed unconditionally. The qualifier "C" is ignored.

Examples:

Instruction	Address	Explanation
NC	Q1.0	As long as the step is active and the condition is fulfilled, the signal at Q1.0 = 1, otherwise 0.
SC	Q1.0	As long as the step is active and the condition is fulfilled, Q1.0 is = 1, and remains set to 1.
RC	Q1.0	As long as the step is active and the condition is fulfilled, Q1.0 is = 0 and remains set to 0.
DC	Q1.0 T#<const>	After the end of the time specified in <const> and as long as the step is active and the condition is fulfilled the signal at Q1.0 is = 1. If the step is not active the signal is = 0.
LC	Q1.0 T#<const>	As long as the step is active and the condition is fulfilled, the signal at Q1.0 is = 1 for the specified time <const>. If the step is not active, the signal is = 0.
CALLC	FB5.DB3	Calls the specified block, when the condition is fulfilled. After the processing of the block, the GRAPH program is further executed.

Actions Triggered by an Event

Action block with event-driven instructions

Action_block _3	
A1 N	M1.1
L1 N	M1.2
L0 N	M1.3
S1 N	M1.4
S0 N	M2.4
V1 N	M2.5
V0 N	M2.6

The action is executed once in the cycle in which the event occurs

Events

- **A1 = Acknowledge**
- **L1 = Interlock error coming**
- **L0 = Interlock error going**
- **S1 = Step activated**
- **S0 = Step deactivated**
- **V1 = Supervision error coming**
- **V0 = Supervision error going**

Overview

An event can be detected and linked with an action. That way it is possible not only to monitor individual steps but also to monitor and influence the entire sequential control system.

Activating and Deactivating Steps

Other steps of a sequencer can be activated or deactivated with the instructions ON and OFF.

Event	Instruction	Address	Explanation
S1 S0 V1 V0	ON OFF	Si i=Step number	Dependent on the event activate or deactivate step
L1 L0 A1	OFF	S_ALL	Dependent on the event activate or deactivate all steps; except for the step that contains the action

Timers and Counters in Actions

Action block with timers and counters

Action_block _4	
S1 CU	C3
A1 CSC	C2
	C#123
S1 TL	T5
	S5T#2M3S
L1 TDC	T6
	S5T#4MS

Always linked to an event.

Causes the one-time execution of the action in the scan cycle in which the event occurred

Can also be combined with the interlock condition C

Counters

- **CS = Set count**
- **CU = Count up**
- **CD = Count down**
- **CR = Reset count**

Timers

- **TL = Extended pulse**
- **TD = ON Delay**
- **TR = Reset time**

Counters

All counters in actions always depend upon an event. That is, the event determines the point in time when the operation is activated. As an option, counters can be attached to an interlock (C).

The count operations that are linked to the interlock condition are only then executed when the interlock conditions are fulfilled in addition to the event (supplement letter C). Count operations without interlock are executed when the event occurs.

The counters in actions behave like the S5-compatible counters of other S7 programming languages: there is no overflow and no underflow and when the count is 0, the counter bit is also 0, otherwise it is 1.

Counter Start Value

All actions that contain the operation CS require the specification of a counter start value. The counter start value is programmed with the following syntax:

- <Counterstartvalue> = IWy, QWy, MWy, LWy, DBWy, DIWy; variables of the WORD data type; C#0 to C#999; Y = 0 to 65534

Timers

Just as with counters, all timers in actions always depend on an event. That is the event determines the point in time when the operation is activated. As an option, timers can be linked to an interlock (C). The time operations that are linked to an interlock are only then executed when the conditions are fulfilled in addition to the event.

Duration

All actions that contain one of the operations TL or TD, require the specification of a duration. The duration is programmed with the following syntax:

- <Duration> = IWy, QWy, MWy, LWy, DBWy, DIWy; variables of the S5TIME, WORD types; S5T#time_constant, Y = 0 to 65534

Arithmetic Operations in Actions

Action block with instructions

Action_block _5	
N	MW0 := IW40
S1 NC	A_1 := B_2
N	C_1 := BCD_TO_NUM(IW4)
A1 N	E_5 := EXP(MD22)
S1 N	X_Diff := X_new - X_old
S1 N	Path:= V_act * Delta_T

Always requires the action identifier N

Can also be linked to an event or/and combined with interlock condition C

Assignments

- **Direct assignment, such as: A := B**
- **Assignment with built-in functions**
  - **conversion functions, such as: C := ROUND(D)**
  - **arithmetic functions, such as: E := SQRT(D)**
  - **other functions, such as: F := RLDA(G)**
- **Assignment with operator, such as: A := B + C**

Overview

Actions can be linked with simple arithmetic terms. An action that contains an arithmetic term, always requires the operation N.

As well, the action can be made dependent on an event and/or linked with an interlock (supplement letter C).

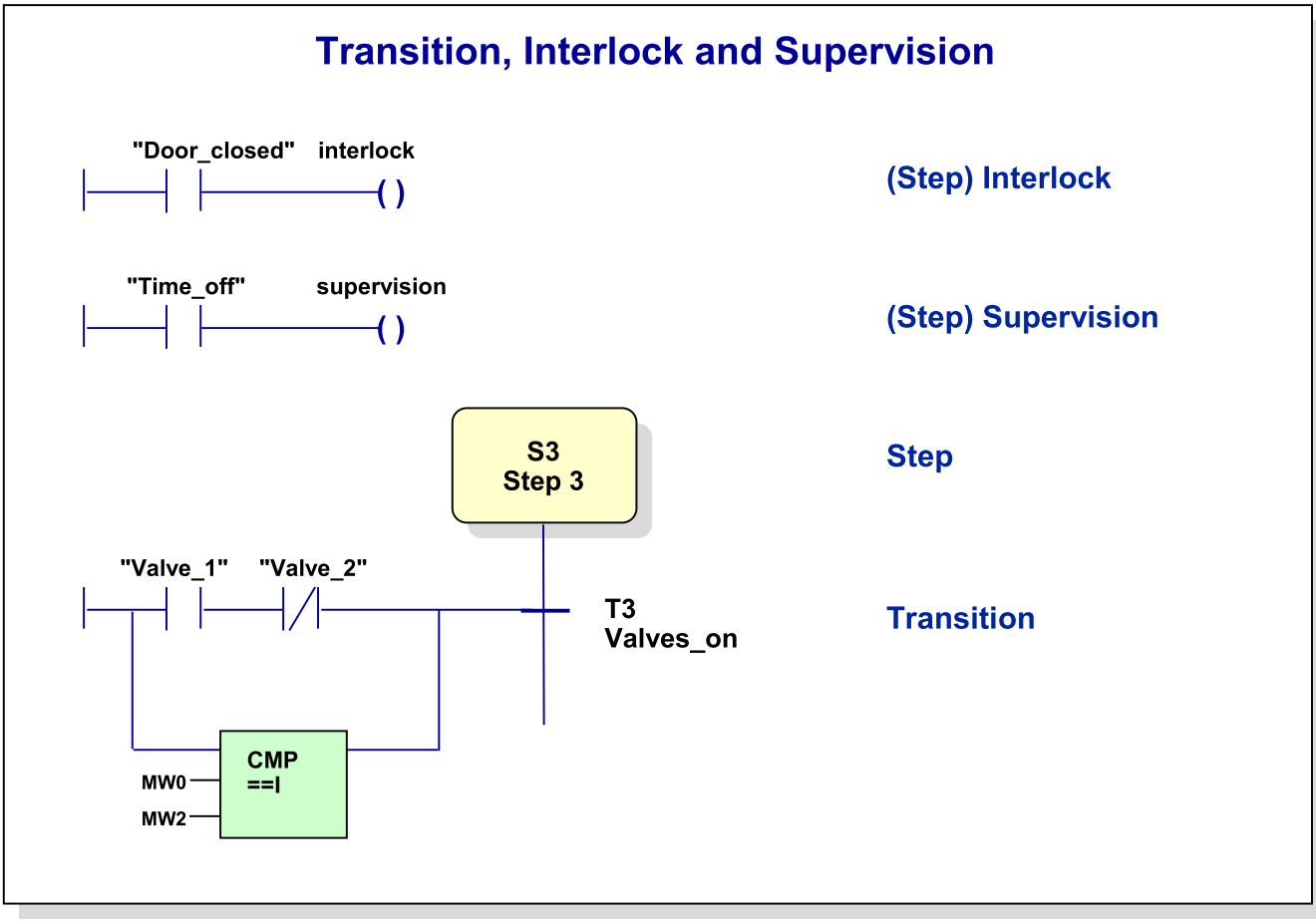
Assignments

The assignments with arithmetic operations can be divided into the following three groups:

- Direct assignments: they are entered according to the syntax A := B . The following data types can be used in a direct assignment:
  - 8 bit: BYTE, CHAR
  - 16 bit: WORD, INT, DATE, S5TIME
  - 32 bit: DWORD, DINT, REAL, TIME, TIME_OF_DAY
- Assignments with built-in functions: they are entered according to the syntax A:=func(B) . The built-in functions can be divided into three groups, in turn:
  - conversion functions, such as: INT_TO_DINT, DINT_TO_REAL, ROUND, BCD_TO_NUM, ...
  - complex arithmetic functions, such as: SIN, COS, EXP, LN, ASIN, ...
  - other functions, such as: NOT, SWAP, RLDA and RRDA
- Assignments with operator: they are entered according to the syntax A:=B<operator>C . The operators are basic arithmetic forms +, -, *, / and bit-by-bit links, such as: AND, OR, XOR, SHL, ...

Note

For overloaded functions and operators, the data type of the assigned operand A always determines the data type of the term.



Overview

Conditions are binary states of processes that can be linked to one another as LAD or FBD elements (NO contact, NC contact, And box, Or box, compare) in ladder logic or in function block diagram according to boolean logic.

The result of a logic operation (RLO) can affect individual actions of a step, the transition to the next step or the entire sequencer. You can program logic operations in LAD or FBD. You can switch the programming languages with *View -> LAD/FBD*.

Conditions

Transition: In a transition, step-enabling conditions are programmed that control the transition from one step to the next. A transition is displayed and programmed in the single step or single page view. The transition switches to the next step when the result of the logic operation is 1.

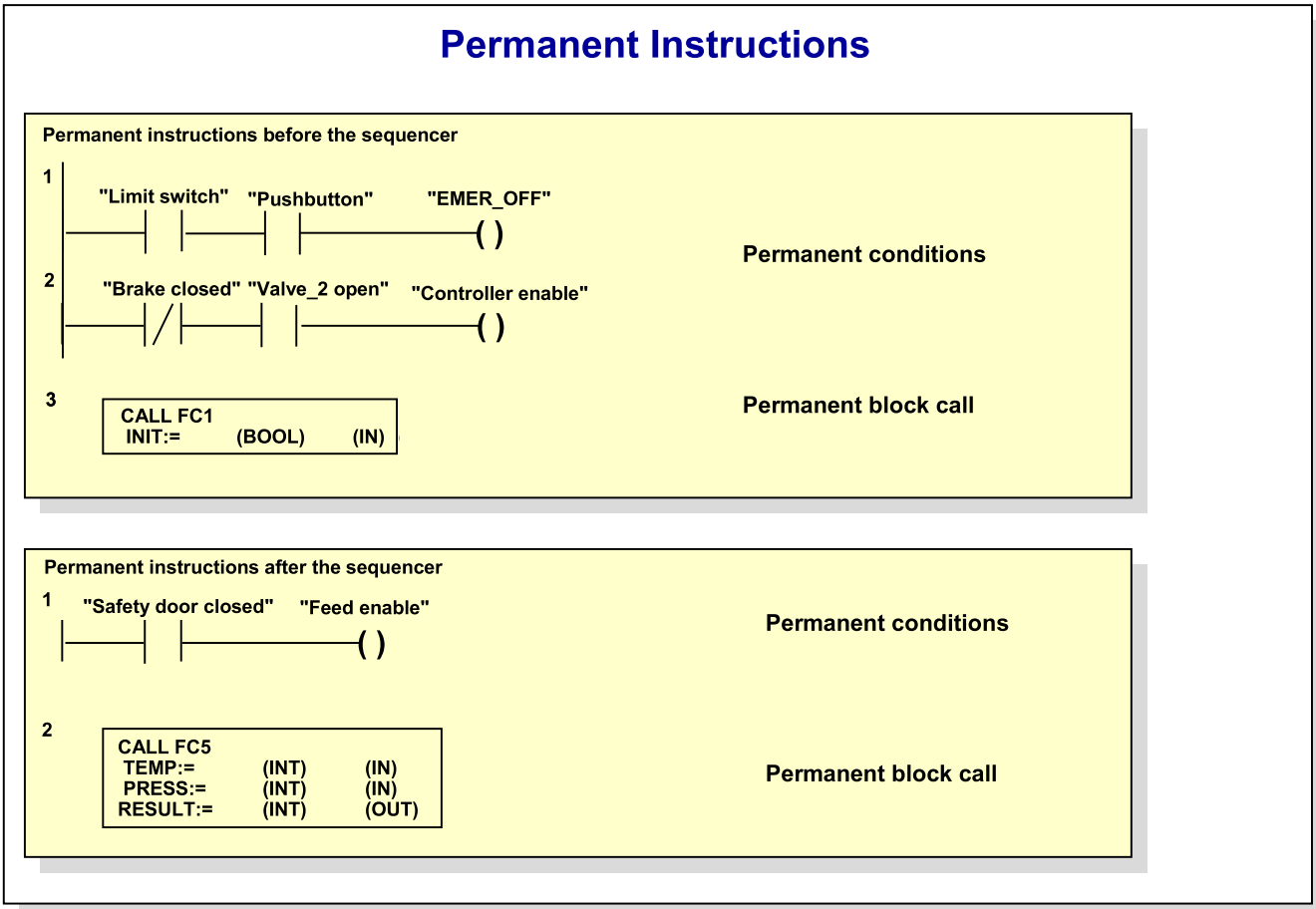
The transition does not switch to the next step in the sequencer when the result of logic operation is 0. The active step stays active.

(Step) Interlock: The interlock defines conditions that affect the execution of individual actions. If the logic operation of the conditions is fulfilled, then the actions with C are executed. If the logic operation is not fulfilled, if there is a malfunction and the actions with C are not executed, an interlock error is indicated (Event L1).

(Step) Supervision: The supervision defines conditions that affect the enabling of the sequencer to the next step.

If the logic operation of the conditions is not fulfilled, there is no malfunction. The sequencer switches to the next step, when the following transition is fulfilled. If the logic operation of the conditions is fulfilled, there is a malfunction. (Event V1). The sequencer does not switch to the next step - regardless of whether the next transition is fulfilled or not.

Interlock and supervision conditions can only be programmed in the single page view.



Overview

Permanent instructions are conditions or block calls located before or after the sequencer and executed once per cycle.

You can use the permanent instructions, for example, to:

- Program only once conditions that need to be fulfilled at several points in the sequencer.
- Call blocks containing STL, LAD, FBD or SCL instructions from S7-GRAPH.

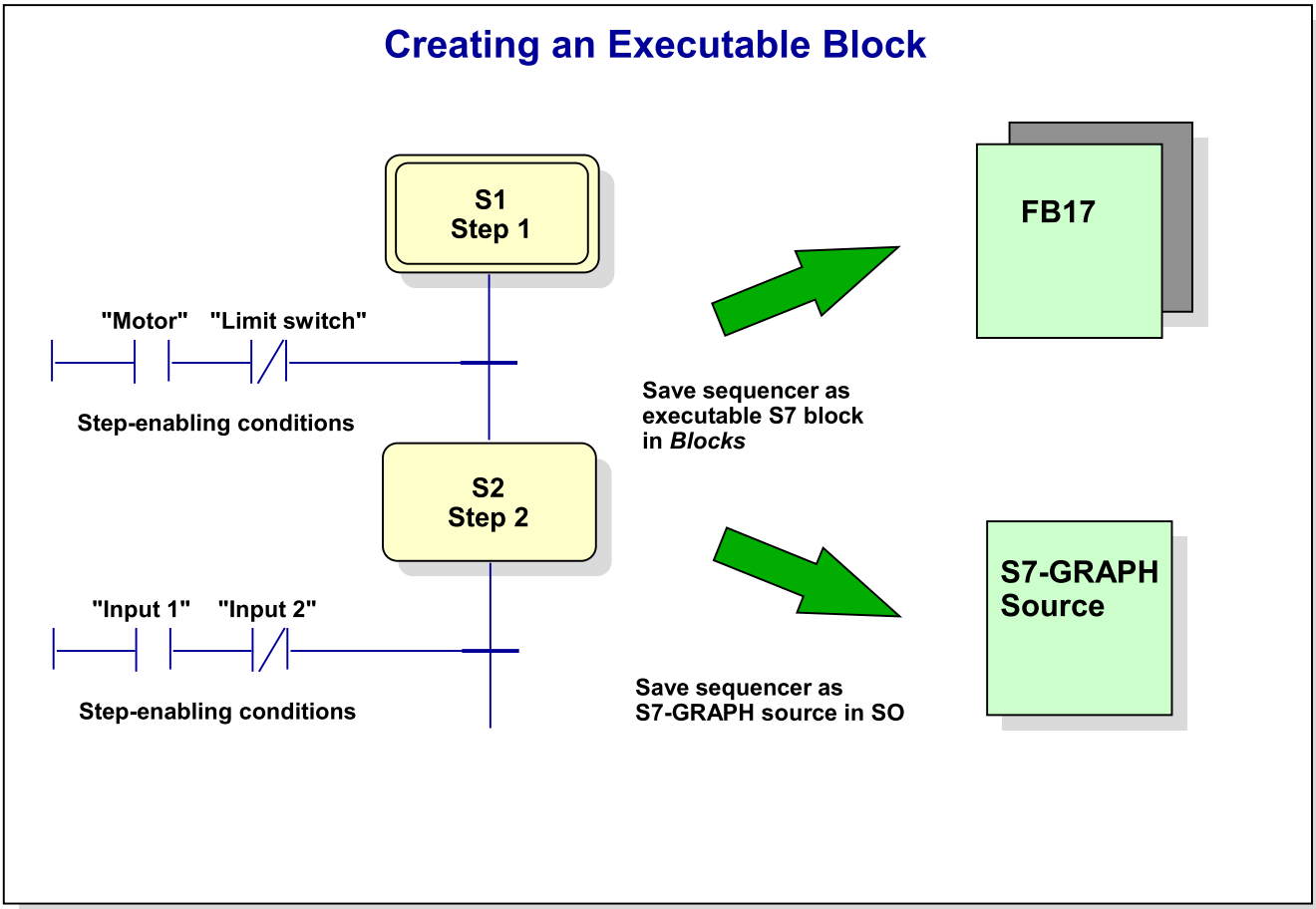
You can have as many permanent instructions in a GRAPH FB as you like.

Permanent Instructions

You program permanent conditions or block calls in the *Permanent Instructions* view. Select the Drag&Drop insertion mode with the menu options *Insert -> Drag&Drop* and proceed as follows:

1. Select the menu options *View -> Permanent Instructions*, if the necessary window is not yet displayed.
2. Choose the menu options *Insert -> Permanent Instruction- > Condition/Call* or click with the mouse pointer on the correponding icon in the toolbar.

   The mouse pointer changes its shape to represent an LAD network or a CALL instruction.
3. Place the mouse pointer in the space provided.
4. Choose the menu option or click the icon in the toolbar again.
5. Insert the LAD elements you want to use to program the condition or enter the name of the block you want to call after the CALL instruction and press the Enter key.
6. In the case of a block call, assign actual parameters to the formal parameters displayed.



Overview

When saving an S7-GRAPH function block, a compilation function is triggered. That is, a function block that can be downloaded to the PLC is created from the sequencer. Only error-free user programs can be saved. If the program cannot be compiled because of errors then it can also not be saved.

If you have to interrupt your work even though it still contains several configuration errors, you can, at any time, save the current state in an S7-GRAPH source by using the menu command *File -> Generate Source*.

Options

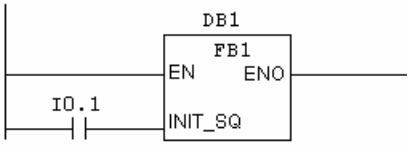
You can set the following options for compiling by choosing the menu items *Options -> Customize* and then the "Compile" tab:

- Which parameters are to be available when you call S7-GRAPH FBs (minimum, standard or maximum parameter set)
- How steps and transitions are to be stored in the instance DB (as structure arrays or as individual structures) and whether this interface description is also to be downloaded to the PLC or not.
- Whether the generated FB is to be executable on its own or whether a standard FC (FC 70 or FC 71) containing the major part of the management code is to be used. If you choose the standard FC option, you can store more steps in the FB than if it is to be executable on its own
- Whether condition analysis data is to be written in the instance DB, whether "Skip mode" is activated and whether supervision errors that occur during operation have to be acknowledged.
- Whether program and process are to be synchronized and whether all interlock conditions are always to be processed in manual operation. When displaying the status, a missing interlock and the step which is faulted as a consequence are then displayed.
- Whether warnings are to be displayed in the message window during compiling.
- Whether interlock and supervision errors are to be handled by SFC 52 (diagnostic buffer entry) or using SFC 17 and 18 (send to HMI devices).

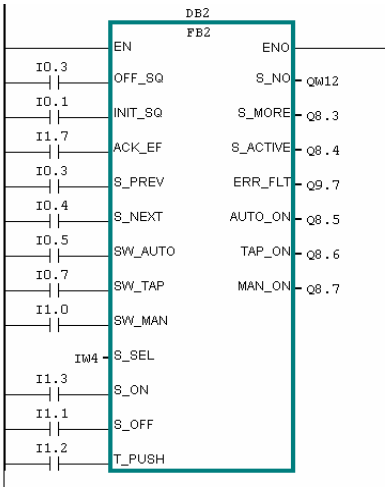
Integrating an FB Call in OB1

Parameter sets in block creation or in the call

- **Minimum call parameters**
  - **1 input parameter for controlling the sequencer**
- **Standard call parameters**
  - **12 input parameters for controlling the sequencer**
  - **7 output parameters for displaying operating states**
- **Maximum parameter set**
  - **17 input parameters for controlling the sequencer**
  - **12 output parameters for display**



Minimum parameter set



Standard parameter set

Sequencer FB By selecting the menu options *Options -> Cutomize* and the "Compile" tab page you can also choose whether the FB is to be assigned the minimum, standard or maximum set of parameters when it is created or called.

Minimum Parameters The FB with the minimum parameter set has only one input parameter, INIT_SQ, and activates its sequencers as soon as it has been processed in OB1. This means that the sequencers are executed immediately in Automatic mode.

You use the minimum parameter set when you are running the sequencer in Automatic mode only and when you do not need any additional modifying and monitoring functions.

The initial step(s) is/are activated by a rising edge at the parameter INIT_SQ.

Standard Parameters The operating mode must also be selected when calling this FB. You always use the standard parameter set when the sequencer is to run in different operating modes and when you require feedback information about the process and facilities for acknowledging messages.

The sequencer FB always remains in the operating mode last activated. The previous operating mode can only be deselected by selecting a different one. Parameters that are not required remain unassigned.

Maximum Parameters You always use the maximum parameter set when you need more operator intervention and monitoring facilities for service and startup than those provided by the standard parameter set.

Activating the Debugging Functions

Procedure

- **Download sequencer FB and instance DB**
  - **Use the menu options *PLC -> Download* to download the sequencer FB and the instance DB to the PLC**
- **Select instance DB**
  - **Select the instance DB you want to use for the test by choosing the menu options *Debug -> Test Environment***
- **Start the "Monitor" function**
  - **Select the desired section of the sequencer control system. The status information will be given for the part currently visible in the open window.**
  - **Activate the menu option *Debug -> Monitor* (checkmark)**
- **Exit the "Monitor" function**
  - **Deactivate the menu option *Debug -> Monitor***

Debug Functions

You can execute the S7-GRAPH sequencer in test mode with the Debug functions. The status of the sequencer elements and the signal state of the addresses are displayed on the screen.

Downloading to PLC

In order to download the S7-GRAPH FB with the associated instance DB into the PLC, proceed as follows:

1. With the FB opened, select the options *PLC -> Download*.
2. In the “Download” dialog box select the instance DB which you wish to download into the CPU together with the opened FB,.
3. If necessary, confirm the query as to whether you want to overwrite blocks of the same name that are already in the CPU.

Note

If possible, download the blocks in STOP mode, because error conditions can occur if you overwrite an old program in RUN mode.

Activate Monitor Mode

Select the test mode using the menu options *Debug -> Monitor* or through the appropriate icon in the toolbar. This function is performed simultaneously for all opened windows of the same sequencer FB.

Sequencer monitoring (status) must always be interrupted first before you can make changes in the sequencer or in the logic operations. Only then are actions in the sequencer possible.

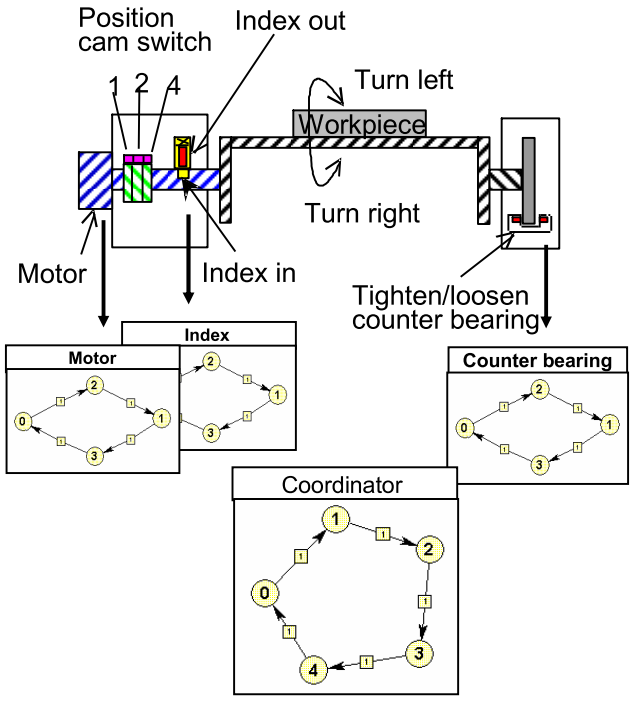
The S7- HiGraph Software Package

Tool for programming using state graphs:

- **Machine is divided up into function units**
- **State graphs are created for each function unit**
- **States contain actions**
- **State graphs communicate by means of messages**

You can optimize the following phases in an automation project with S7-HiGraph:

- **Planning, configuring**
- **Programming and debugging**
- **Startup**
- **Maintenance, diagnostics**
- **Supports reusability**



Motivation for State Graphs

The construction of machinery and plants is an extensive field in which the control of asynchronous, mechanical movements and responses over time presents the main problem.

The language most commonly used in this field at present is STL and sometimes LAD, FBD and sequencer languages (GRAPH5, S7-GRAPH, etc.). However, these languages are not particularly suitable for mechanical engineers, for example for formulating the application of the mechanical construction.

The result is that each group, for example the mechanical engineers or the automation engineers, uses its own methods (languages), which makes it difficult for them to exchange information.

The aim of a working group (approx. 1980) with the task of investigating CASE tools for PLCs was to specify a tool that could be used in all phases of a project right through from the definition of the problem to programming and even servicing. This tool was to be suitable for use in all areas and allow an increasingly detailed approach to the automation problem. It was also a requirement that the documentation and programs should be re-usable for other projects of similar type.

The result was the state graph method described here, which is marketed by SIEMENS under the product name S7-HiGraph. S7-HiGraph can be used on the PLCs of the S7-300 series (CPU 314 and higher) and S7-400.

Advantages

The "object-oriented" approach of S7-HiGraph is ideally suited for:

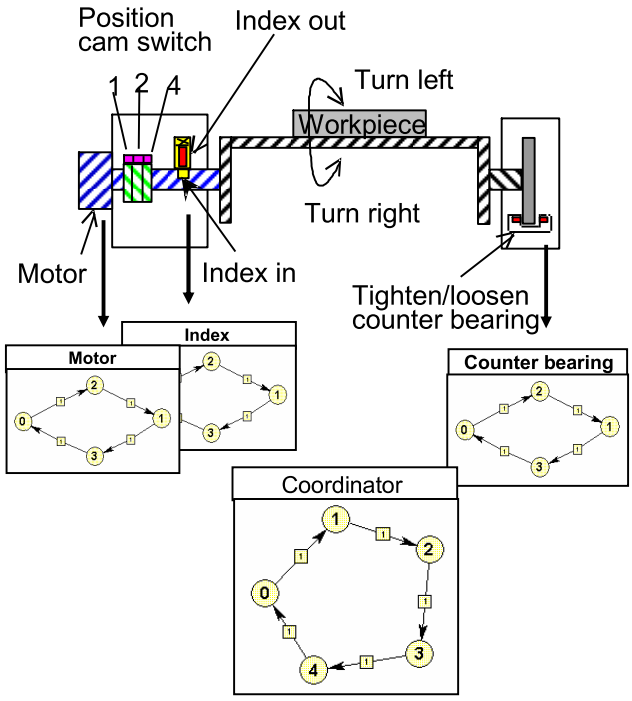
- Machine and plant engineers (mechanical design)
- Automation specialists (electrical engineers) as a common means of description
- Commissioning and maintenance engineers

The state graph method enables the entire process of constructing a machine or plant to be optimized by reducing the development and turnaround times and the commissioning time.

Principle of the State Graph Method

Example: Rotary table for milling machine

- **Function units (FU)**
  - **Motor**
  - **Index**
  - **Counter bearing**
- **State graphs**
  - **One graph for each FU**
  - **Additionally one coordinating graph**



Date: 30.06.2011
File: PRO2_12E.22

Overview

The state graph method orients itself towards the "objects" in the real world when used for drafting out and programming automation projects. The machine or plant to be automated is seen as a combination of separate elements, or function units.

Function Units

The function unit (FU) is the smallest mechanical unit of a machine or plant. An FU is normally composed of basic mechanical and electrical elements.

During programming, each function unit is assigned a state graph in which the functional, that is the mechanical and electrical, properties of the FU are represented.

For the state graph method the object to be automated is broken down into separate function units.

State Graph

The state graph describes the dynamic behavior of a function unit. It describes the states a function unit can assume and the transitions between them.

State graphs are program sections that can be used again and again. State graphs created for a particular function unit can be used in other places in a program where a similar functionality is required.

Graph Group and Instances

The entire functionality of a machine or plant can be described by a combination of parallel state graphs.

All the state graphs drawn up within an S7 program are stored centrally in the "Sources" folder. From there you can insert and call them as often as you like in one or more graph groups.

A state graph call in a graph group is referred to as an instance.

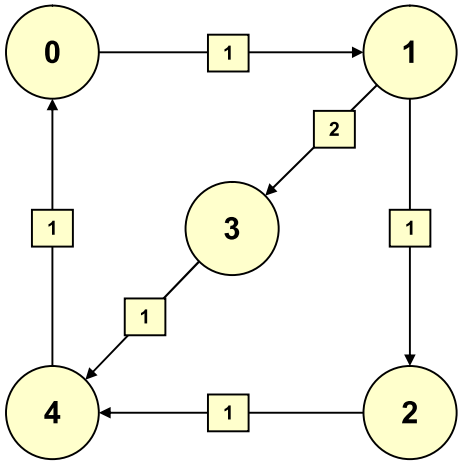
Elements of a State Graph

States 0,1, ...

- **Represented as circles**
- **Static states**
- **Dynamic states**
- **Always one active state**
- **Actions are assigned to the states**

Transitions

- **Represented as arrows**
- **Transition conditions and actions are assigned to the transitions**



SIMATIC S7

Date: 30.06.2011
File: PRO2_12E.23

SITRAIN Training for Automation and Drives

States

A state graph is a continuous, directed graph, that represents all states of a function unit as circles and all transitions as arrows.

The states of a function unit can be of the static type (Door_is_open, Motor_off, etc.) or the dynamic type, that is, movement states (Door_opens, Motor_turns_left, etc).

At any given time, the subsystem which is described by a state graph, or graph, is in exactly one state.

Actions

Actions can be assigned to states in state graphs. These actions can be subdivided into:

- Actions that are executed once when entering a new state.
- Actions that are executed cyclically, as long as the function unit is in the respective state.
- Actions that are executed once when leaving a state.

The actions are formulated in a language similar to STL.

Transitions

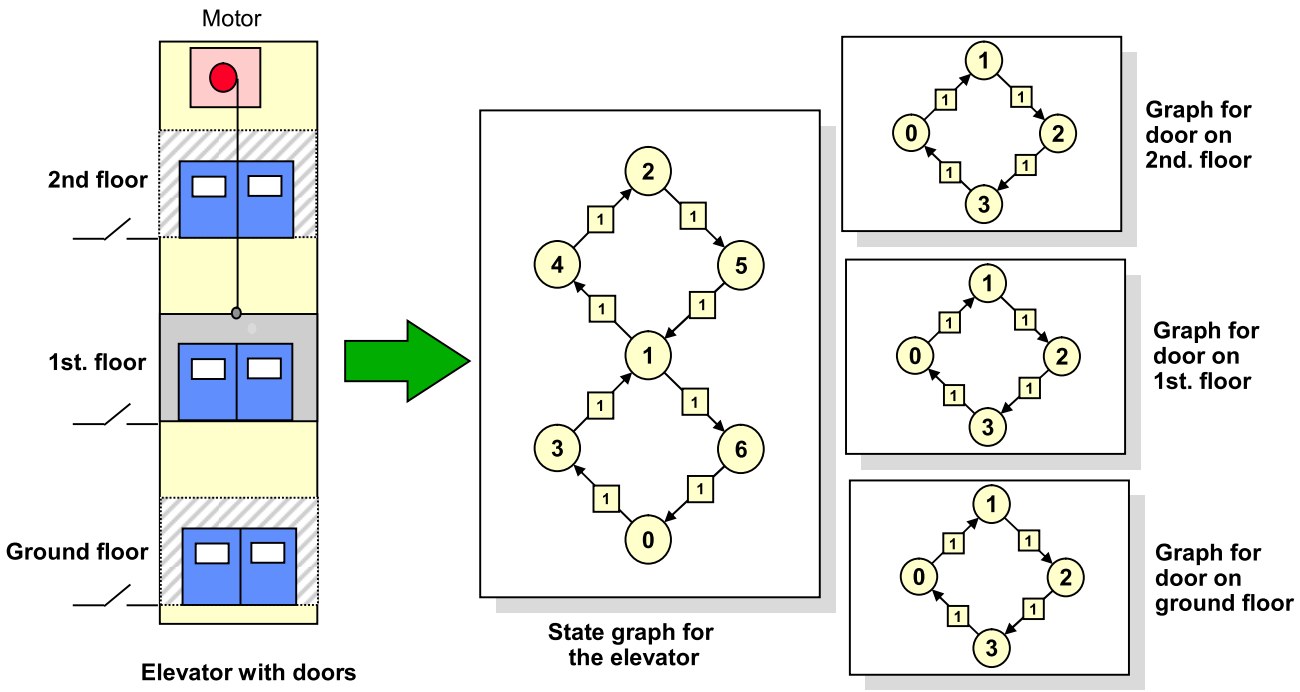
The transitions identify the state changes of a function unit. The transitions between the states are dependent on conditions that are true or not true at any given time.

The subsystem described by a state graph changes its state when the condition of a transition that leads out of a state is fulfilled. A maximum of one transition occurs per cycle for each state graph.

The transition conditions are also formulated in a language similar to STL..

For each transition one action can be formulated which is to be performed when the transition takes place.

Example: State Graphs for an Elevator Controller



Overview

The above example demonstrates the state graph method by means of an elevator in a three story building.

Breaking down into FUs

The object to be automated (elevator with doors) can be broken down into the following mechanical function units with respective state graphs:

- Elevator cage including control
- A door on each floor

State Graph for Elevator Cage

Within a state graph, or graph, the circles represent the possible states of the function unit and the arrows represent the state transitions.

For the function unit "elevator cage", the states 0, 1 and 2 represent the stopping positions of the elevator cage on the respective floors.

The states 3, 4 and 5 and 6, 7 and 8 represent the dynamic up and down movement of the elevator cage between the floors.

State Graphs for Door

In the function unit "door" the states 0 and 1 represent the static states "Door is closed" and "Door is open", the states 2 and 3 represent "Door is opening" and "Door is closing".

Messages

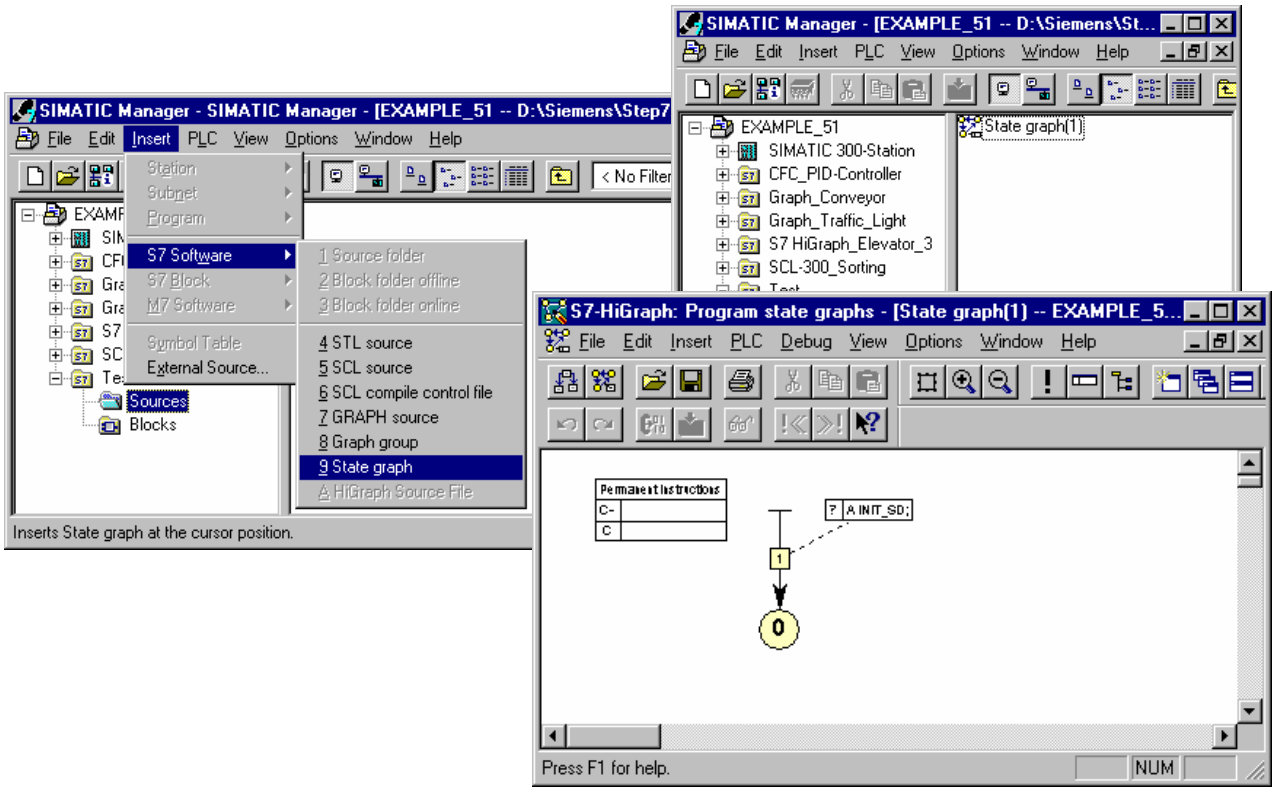
The coordination of the state graph for the elevator cage with the individual state graphs of the doors can take place without an additional coordination diagram.

On reaching the "desired floor" the state graph “Elevator Cage” sends the message "Open_Door" to the associated state graph "Door".

This state graph receives the message and "opens" the door, that is, on receipt of the message the corresponding transition is executed. When the door is closed again, the state graph “Door” sends the message "Door_closed" to the state graph "Elevator Cage".

The elevator cage can then move to the next desired floor.

Creating a State Graph



Date: 30.06.2011
File: PRO2_12E.25

Overview

In order to edit state graphs, S7-HiGraph requires an existing project. You may have to create this first with the SIMATIC Manager before you invoke the HiGraph Editor.

Inserting a State Graph

To insert a state graph with the SIMATIC Manager, proceed as follows:

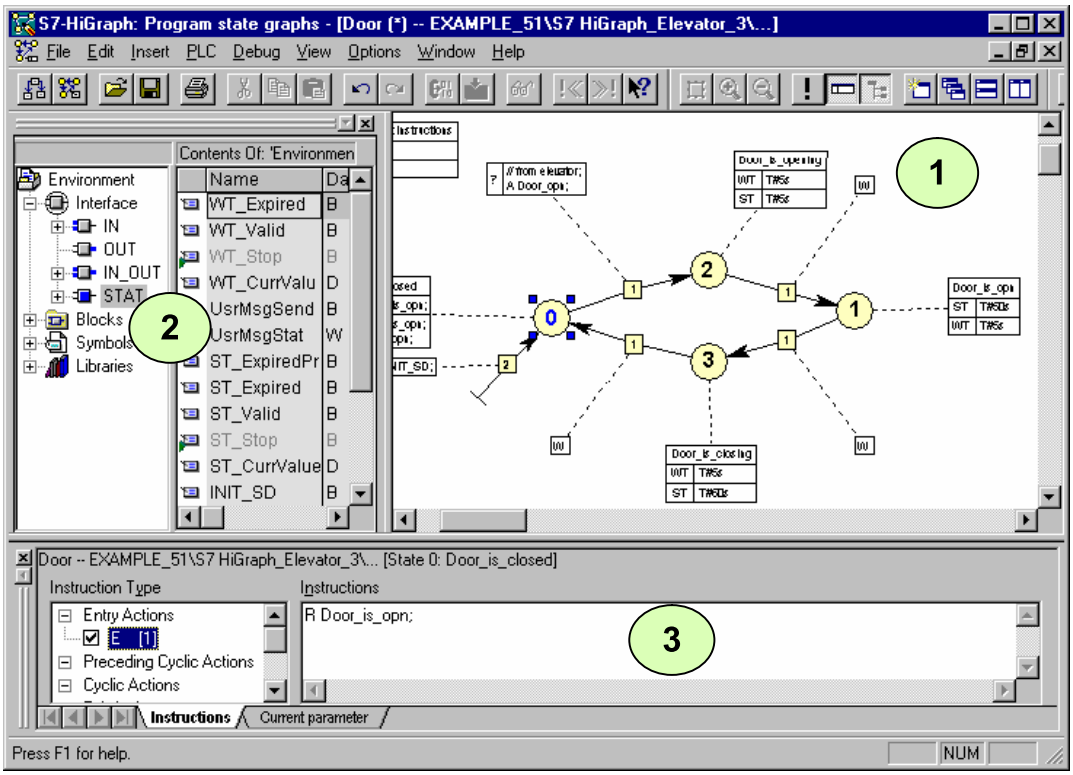
1. Open the Sources folder.
2. Select the menu options *Insert -> S7 Software -> State Graph.*

   A state graph is created under a default name in the *Sources* folder. Before you start editing, you should change the name of the state diagram (to Elevator, for example).
3. Double-click the state graph. The HiGraph Editor is started and the state diagram is opened.

Initial State

The state graph you have opened already contains a state with the number 0 and an ANY transition. The state with the number 0 is the initial state of the state graph. This is automatically assumed when the power is switched on.

The User Interface of HiGraph



User Interface

The HiGraph user interface consists of various windows, which you can display or not, as required. To make optimum use of the screen space available, you can change the size of the windows and move them around.

In addition to the editing windows (1) in in which you edit the state graphs and graph groups, you can also use the following windows:

- The variable declaration window (2) is used for declaring the variables of the state graph. You display this window by selecting the menu options *View -> Variables*.
- You use the instruction input window (3) to program the contents of states, transitions and permanent instructions. You display this window by selecting the menu options *View -> Instructions/Actual parameter*.
- There is another window in which errors and warnings arising during compilation of a graph group are displayed. This window is automatically opened after each compilation. You can also display it by selecting the menu options *View -> Messages*.
- The input window for actual parameters is only available if a graph group is opened. You use this window for assigning the actual parameters of instances.

Declaration of Variables

In the variable declaration you declare the local variables and parameters of the state graph. You also declare the variables to be used for exchanging messages.

The variable declaration consists of the following sections:

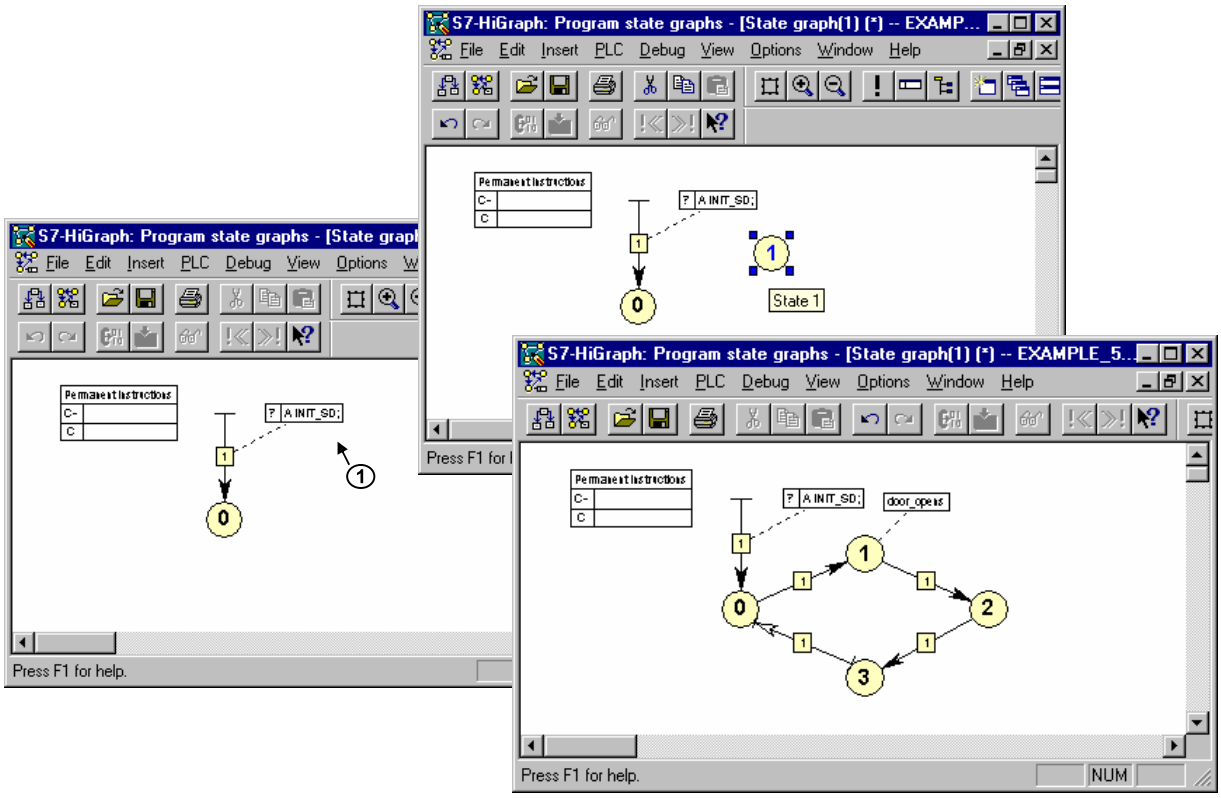
IN: contains the input parameters of the state graph and the pre-defined variables "AutomaticMode", "ManualMode" and "UsrMsgQuit".

OUT: contains the output parameters of the state graph

IN_OUT: contains the input/output parameters of the state graph and the variables used for exchanging messages.

STAT: contains static variables and variables pre-defined by HiGraph

Inserting States and Transitions



Inserting States

To insert new states into a state graph, proceed as follows:

1. With an editing window selected, choose the menu options *Insert -> State*. The cursor assumes the shape of a circle with the next free state number.
2. Position the cursor to the desired position and click with the left mouse button.

   At the selected position, a circle is automatically inserted for a state with a default number 0, 1, 2, etc. Repeat this procedure until you have inserted all the necessary states for the description of the state graph.
3. Exit the input mode by clicking the empty space in the editing window with the left mouse button.

Each state has a number which is unique within the state graph. To make the diagram clearer, you can give each state a name by selecting the menu options *Edit -> Object Properties*.

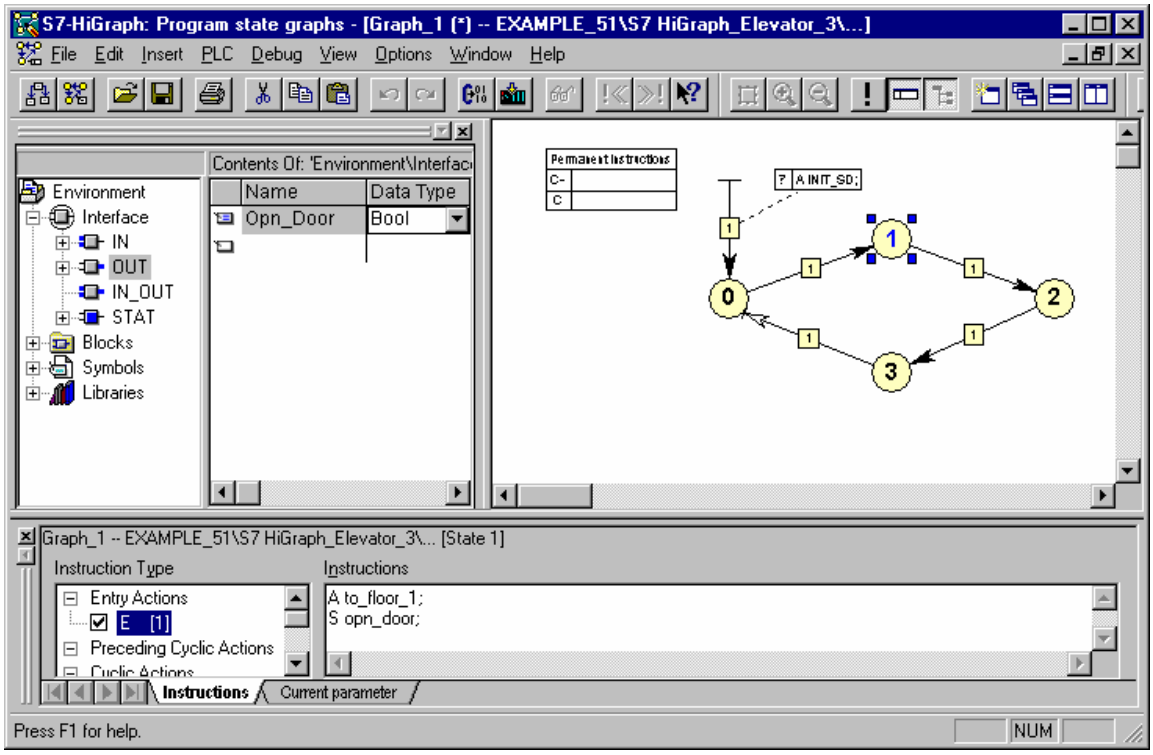
Inserting Transitions

To insert transitions between individual states, proceed as follows:

1. Select the menu item *Insert -> Transition*. The cursor assumes the shape of the transition symbol.
2. Then click with the left mouse button on the initial state and, keeping the left mouse button pressed, drag the arrow to the target state.
   A transition is inserted between initial and target state.
3. Exit the input mode by clicking the left mouse button.

As with the states, you can give each transition a name by selecting the menu options *Edit -> Object Properties*.

Programming Actions



Action Types

You can program actions for each state in the instruction window. The actions are subdivided into the following types:

Entry actions (E): Actions which are performed only once on entering the state.

Preceding cyclic actions (C-): Actions which are executed whilst in the state before the transition conditions are checked.

Cyclic actions (C): Actions which are executed whilst in the state after the transition conditions have been checked.

Exit actions (X): Actions which are only executed once on leaving the state.

To enter instructions, proceed as follows:

1. Select the type of action in the left-hand pane of the instruction window.
2. Using the right mouse button open the pop up menu and select ‘Insert‘. A new entry action, for example, E[1] is generated.
3. Mark this entry and enter the instructions in the right-hand pane of the instruction window.

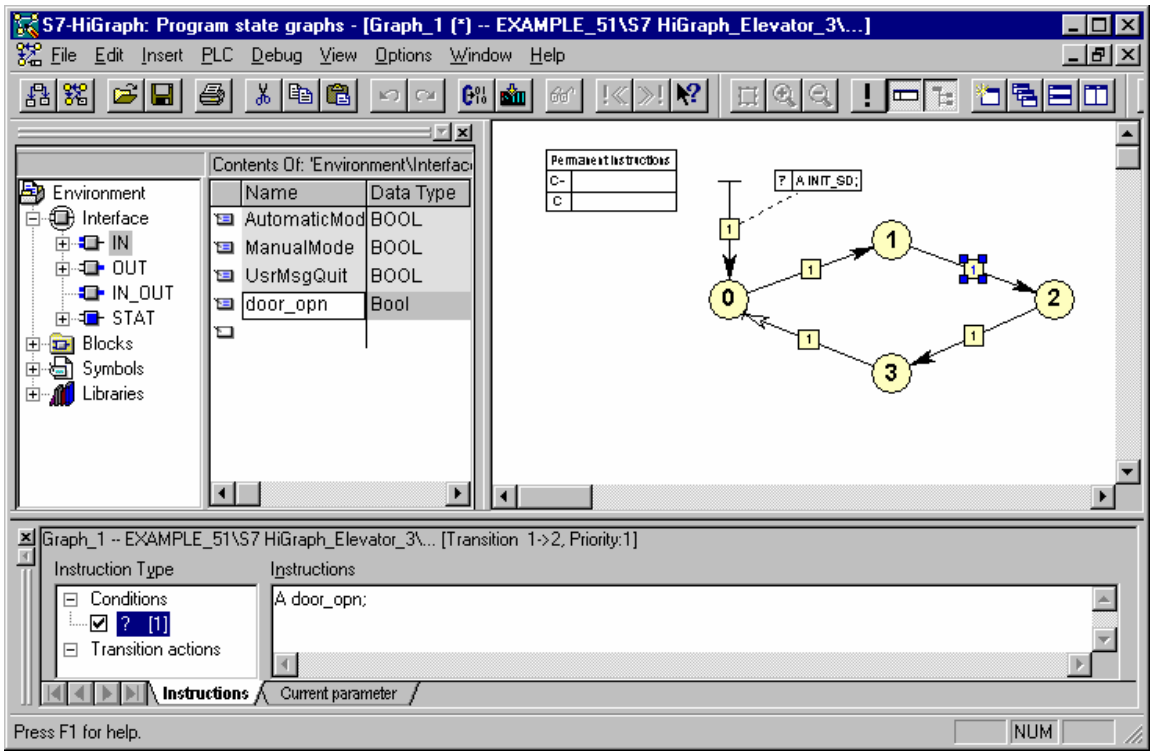
Wait Time

You can specify whether the PLC is to remain in a state for a minimum length of time before checking for step-enabling transition conditions. You must also assign the attribute "Waiting" to the transitions that are to observe the wait time.

Supervision Time

The supervision time is used for monitoring the time spent in a certain state. If the state is not exited within the specified time, the predefined variable "ST_Expired“ is set and an error message is entered in the diagnostic buffer.

Programming Transitions

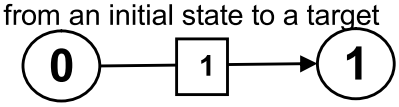


Transitions

A transition contains the conditions for switching from one state to another. Several outgoing transitions can be assigned to one state. HiGraph operates with the following transitions:

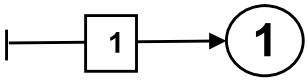
Normal transition: A normal transition leads from an initial state to a target state.

It is represented by the following symbol:



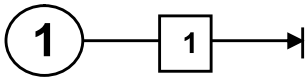
Any transition: An Any transition leads from all states to a target state. It has higher priority than all other transitions.

It is represented by the following symbol:



Return transition: A Return transition leads back from the current state to the state which was previously active.

It is represented by the following symbol:



Priority

Transitions leading out of the same state can be arranged in the right order by assigning them different priorities. If the transition conditions are fulfilled at the same time, the transition with the highest priority is activated.

The highest priority possible in S7-HiGraph has the numerical value 1.

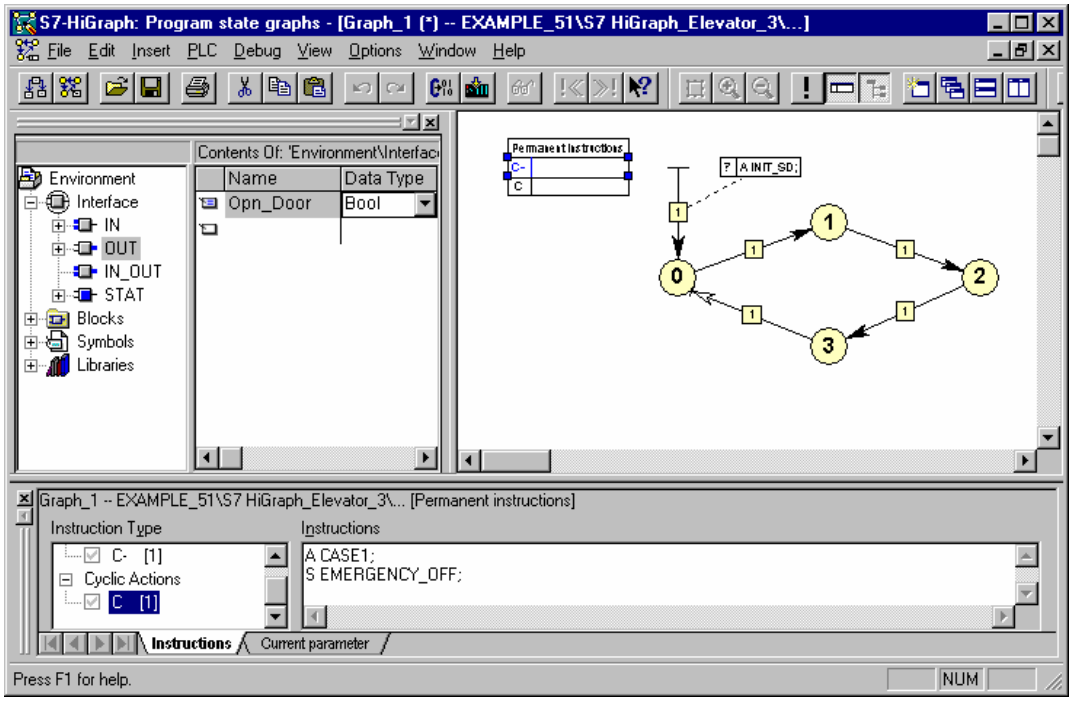
Instructions

You can program conditions and actions for a transition in the instruction window:

Conditions (?): These instructions describe the conditions that must be fulfilled before a change of state can take place.

Actions (!): These instructions are executed once when the transition is activated.

Programming Permanent Instructions



Permanent Instructions

Permanent instructions are executed once in every cycle regardless of the current state. For example, you can program the following activities centrally in permanent instructions:

- Calculation of process variables that are checked in several places.
- Detection and processing of events which require a response not dependent on the current state (example: monitoring of a safety screen).

The following types of permanent instruction are available:

Preceding cyclic actions (C-): These are always performed before the actual state graph is executed.

Subsequent cyclic actions (C): These are always performed after the actual state graph has been executed.

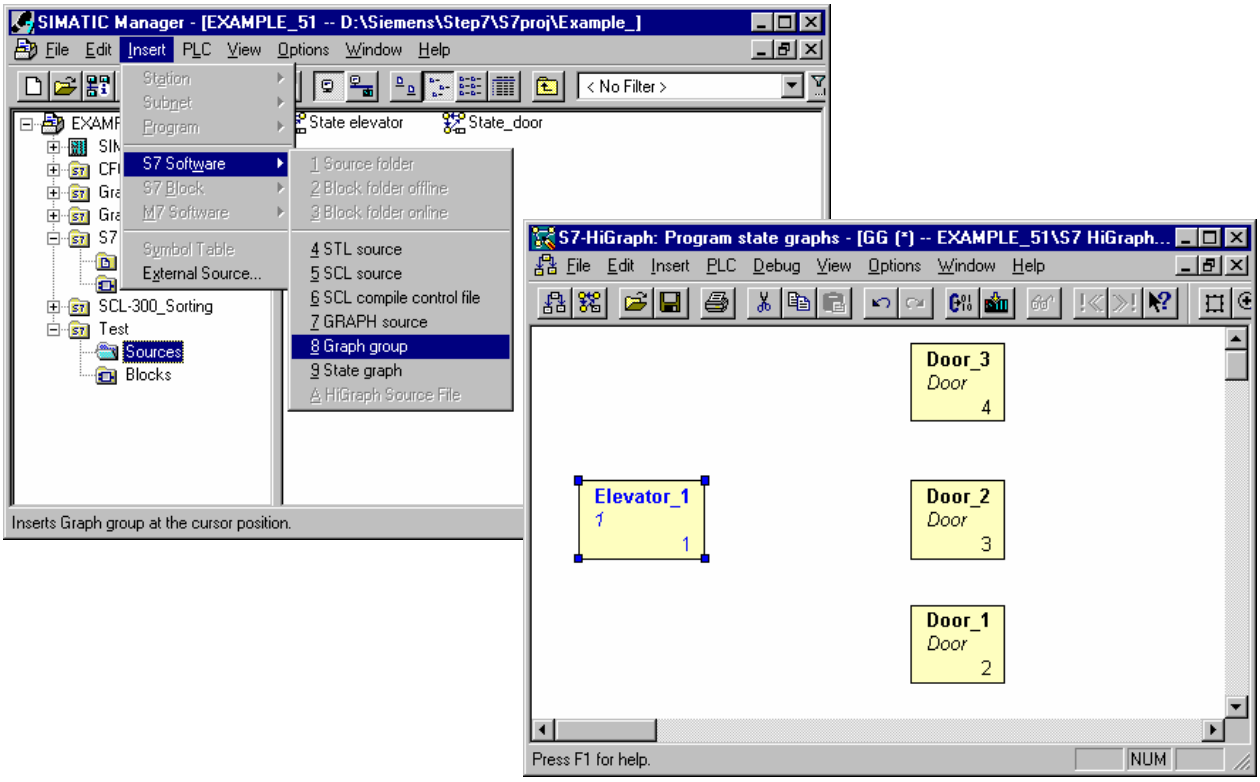
Programming

You program permanent instructions as follows:

1. Double-click the instruction table with the title ”Permanent Instructions”. The instruction input window is opened.
2. Select a type of instruction in the left-hand pane of the window and enter the instruction in STL in the right-hand pane.

When you have entered the instructions, they are displayed in the form of a table in the editing window for the state graph.

Programming Graph Groups



Date: 30.06.2011
File: PRO2_12E.31

Overview

State graphs represent individual function units of a machine. In order to describe a whole machine or plant, the state graphs for the separate function units must be collected together in a graph group.

Creating a Graph Group

To create a graph group with the SIMATIC Manager, proceed as follows:

1. Open the Sources folder in which you want to insert the graph group.
2. Select the menu options *Insert -> S7 Software -> Graph Group.*

   A graph group is created under a default name in the *Sources* folder. You should change the name of the graph group before you start editing it (to Elevator, for example).
3. Double-click the graph group. The HiGraph Editor is activated and the graph group is opened.

Instance Formation of a State Graph

The insertion (calling) of a state graph in a graph group is known as instantiation (instance formation). When you have inserted the state graph into a graph group, you must assign actual parameters to the formal parameters declared in it.

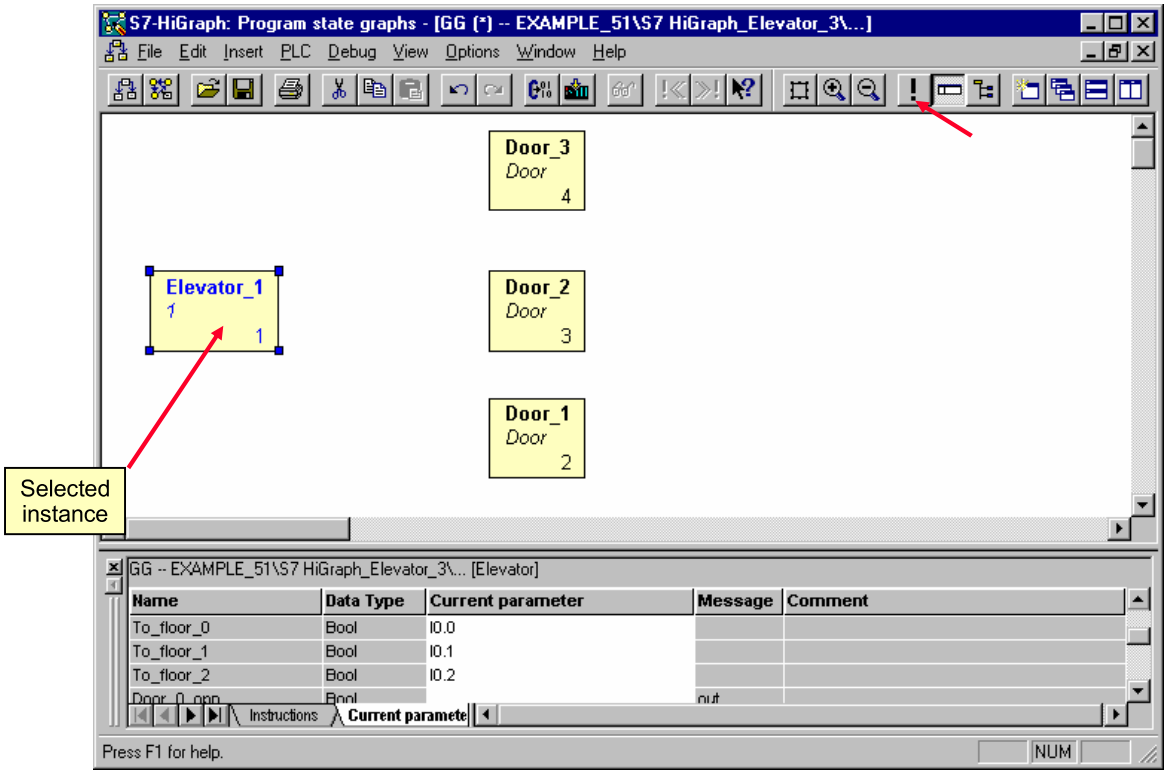
It is the assignment of these actual parameters that enables an instance of the state graph to control a real function unit (motor, elevator door, valve, etc.). If several identical function units (motors of the same type, elevator doors, etc.) need to be controlled, you can do this using several instances of the same state graph.

Inserting State Graphs

To insert instances of state graphs into a graph group, select the menu options *Insert -> Instance*.

You can then give the inserted instance a name by selecting the menu options *Edit -> Object Properties*.

Assigning Actual Parameters



Date: 30.06.2011
File: PRO2_12E.32

Overview

To enable you to use state grpahs again and again by means of instance formation, all the signals used in the state graph must be declared as formal parameters in the declaration window.

The formal parameters save spaces for the actual parameters and are assigned the "actual signals" when an instance is created.

Actual Parameters

When you have inserted the instances of a state graph into a graph group, you assign actual parameters to the formal parameters as follows:

1. First select the required instance of the state graph in the graph group window.
2. Select the menu options *View -> Instructions/Actual parameter* to open the actual parameters window. The names and data types of all the formal parameters in the state graph appear in this window.
3. Assign addresses to the formal parameters in the "Actual Value" column.

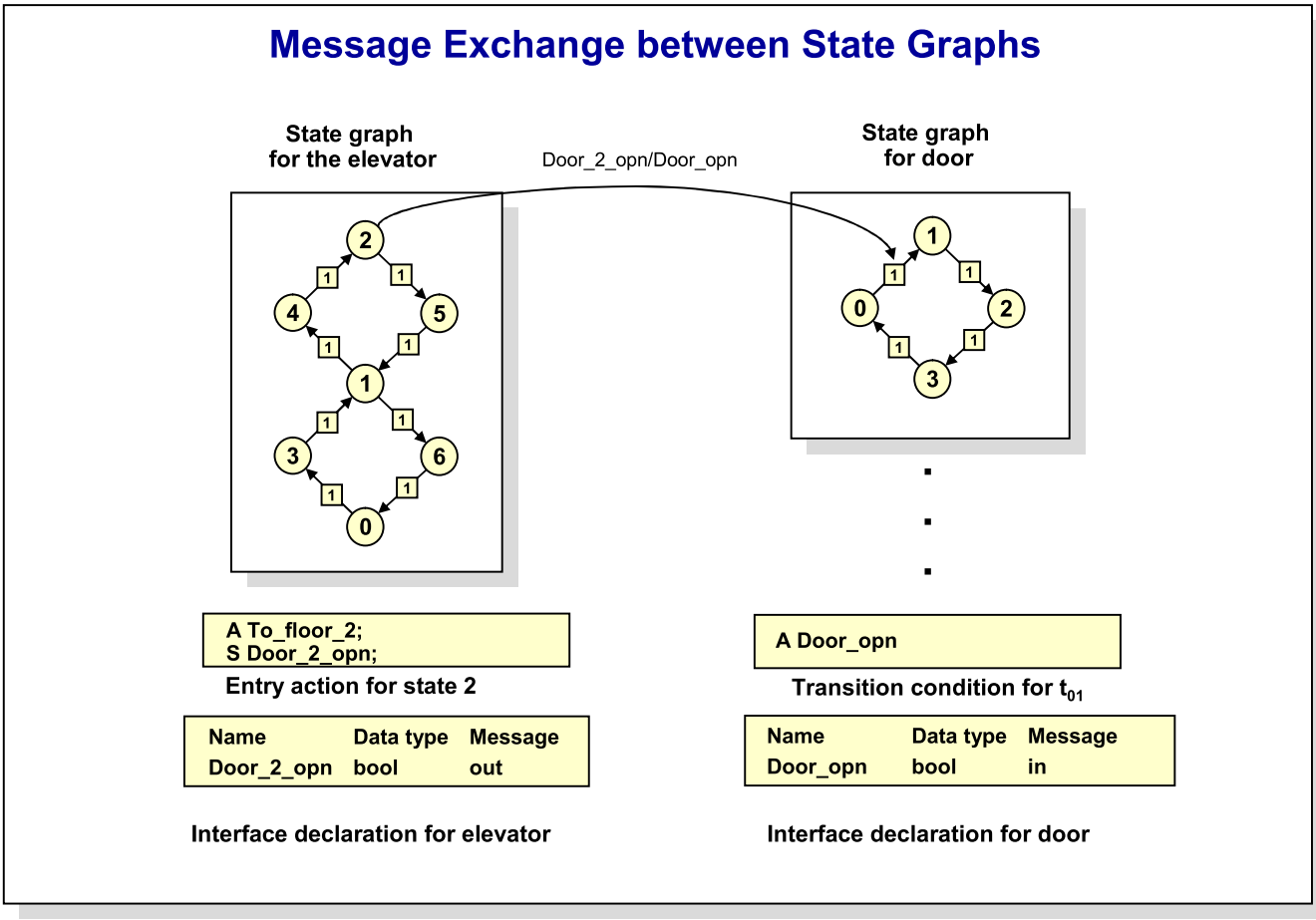
   As actual parameters for a HiGraph program you can use the addresses of I/O signals, bit memories, counters, timers, as well as data and logic blocks.

   In your program you can use either absolute addresses (I 1.1, M 2.0, FB 21, for example) or symbolic names (”Motor_ON” for example).

   You use the symbol table of your S7 program to assign symbolic names to the absolute addresses. You can toggle between absolute and symbolic addressing by selecting the menu options *View -> Symbolic addresses/Absolute addresses*.

Messages?

You will find out how to assign messages to "actual parameters" on the following pages.



Overview

State graphs can influence each other by exchanging messages. Messages are normally sent by a source graph and then evaluated by the destination graph.

Messages

A message is a binary variable, that can be sent by a graph within its action part and that can be received by the receiving graph within its action or transition parts.

A message is always sent to a specific state graph (internal messages) or to an address (external message).

Internal Messages

Internal messages are used for synchronization within the same graph group. They are mapped by the system in the bits of the associated DB.

External Messages

External messages are used for synchronization of graphs in different graph groups (FCs). A global bit variable is declared when assigning the actual parameter.

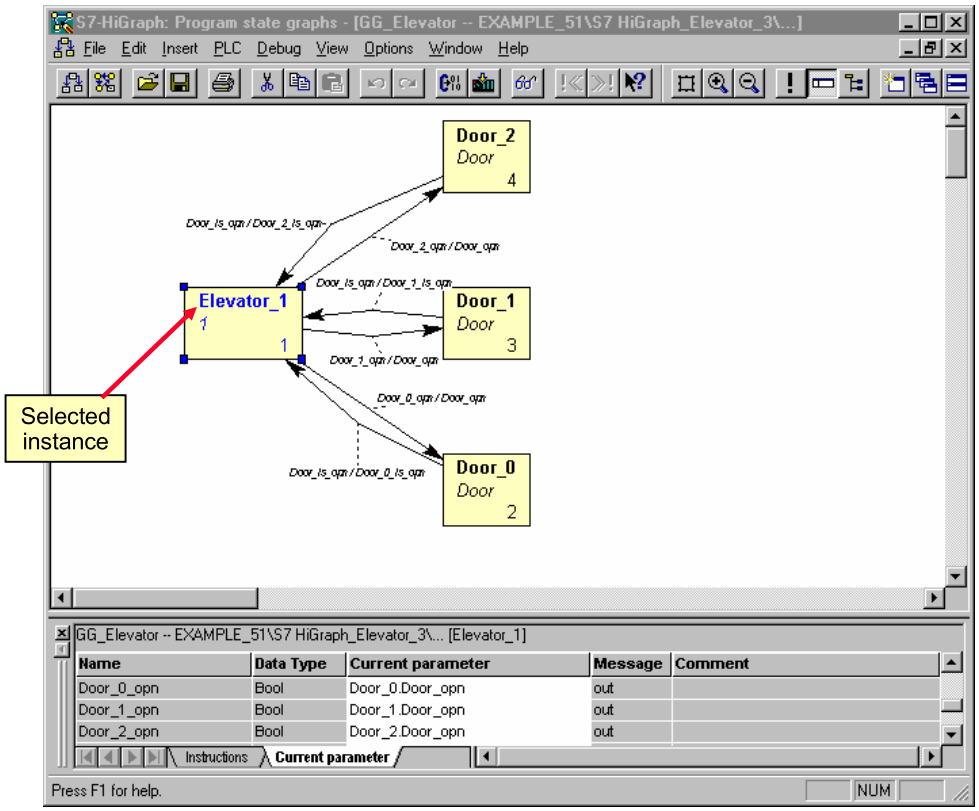
Declaration of Messages

You first declare messages in the IN_OUT section of the declaration window of a state graph. In addition to the name and the data type (always BOOL) of the message you must also specify the message type, that is input or output message.

~~Name~~	~~Data Type~~	~~Message~~	~~Comment~~
Door_0_opn	bool	out;	//Output message
Door_closed	bool	in	//Input message

You assign the actual recipient of the message (internal message) or the bit variable with which the message is connected (external message) in the actual parameter window of the relevant graph group.

Assigning Actual Values for Messages



Date: 30.06.2011
File: PRO2_12E.34

Overview

When you have inserted the instances of a sending and receiving state graph in a graph group, you must tell the sending instance which instance is to receive the message (internal message) or which bit variable the message is connected to (external message).

Assignment for Internal Messages

To assign a recipient to an internal message, proceed as follows:

1. Select the instance of the sending state graph in the graph group window.
2. In the actual parameter input window, select the message to be sent and enter the receiving instance in the "Actual Value" column.

   The full name of the receiving instance consists of the name of the instance that is to receive the message and - separated by a period - the name of the message (type: in), declared as an input message in the receiving graph.

Name	Data Type	Actual Value	Message
Door_0_opn	BOOL	Door_0.Door_opn	out
Door_1_opn	BOOL	Door_1.Door_opn	out

For internal input messages you do not need to assign an actual value in the actual parameter window of the receiving instance.

Assignment for External Messages

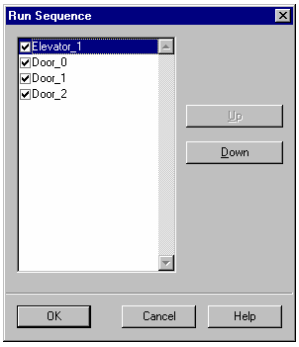
To link an external message to a bit variable proceed as follows:

1. Select the instance of the sending state graph.
2. In the actual parameter input window, select the message to be sent and enter a global bit address in the "Actual Value" column.
3. Now select the instance of the receiving state graph and assign the same global bit address to the relevant input message.

Saving and Compiling

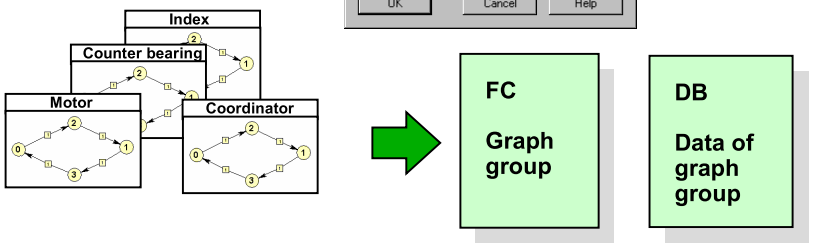
Establish execution sequence

- **Menu:**
  Edit -> Execute Order



Compile

- **Menu:**
  File -> Compile



Integrate in OB1

- **Assign parameter INIT_SD**

```
OB1 : Title:
Network 1: Title:
      CALL  "Elevator"
       INIT_SD:=I0.7
```


Saving

When you save HiGraph objects, they are stored in the "Sources" folder of the S7 program in their current state. The syntax is not checked. This means that it is possible to save and later use objects containing errors.

You save HiGraph objects by selecting the menu options *File -> Save.* Please note that any changes you make to a state graph affect all the instances of it that have already been inserted in graph groups.

Code Execution Sequence

The state graph system is executed cyclically. You can establish the order in which the individual instances within a graph group are to be executed by selecting the menu options *Edit -> Execute Order*.

Compiling

In HiGraph you only compile graph groups; you cannot compile state graphs individually. When compiling, HiGraph checks the syntax of the program, generates a function (FC) and a data block (DB) and stores them in the "Blocks" folder of the relevant S7 program.

Any syntax errors detected during compiling are reported in the message window. In this case, no blocks are generated.

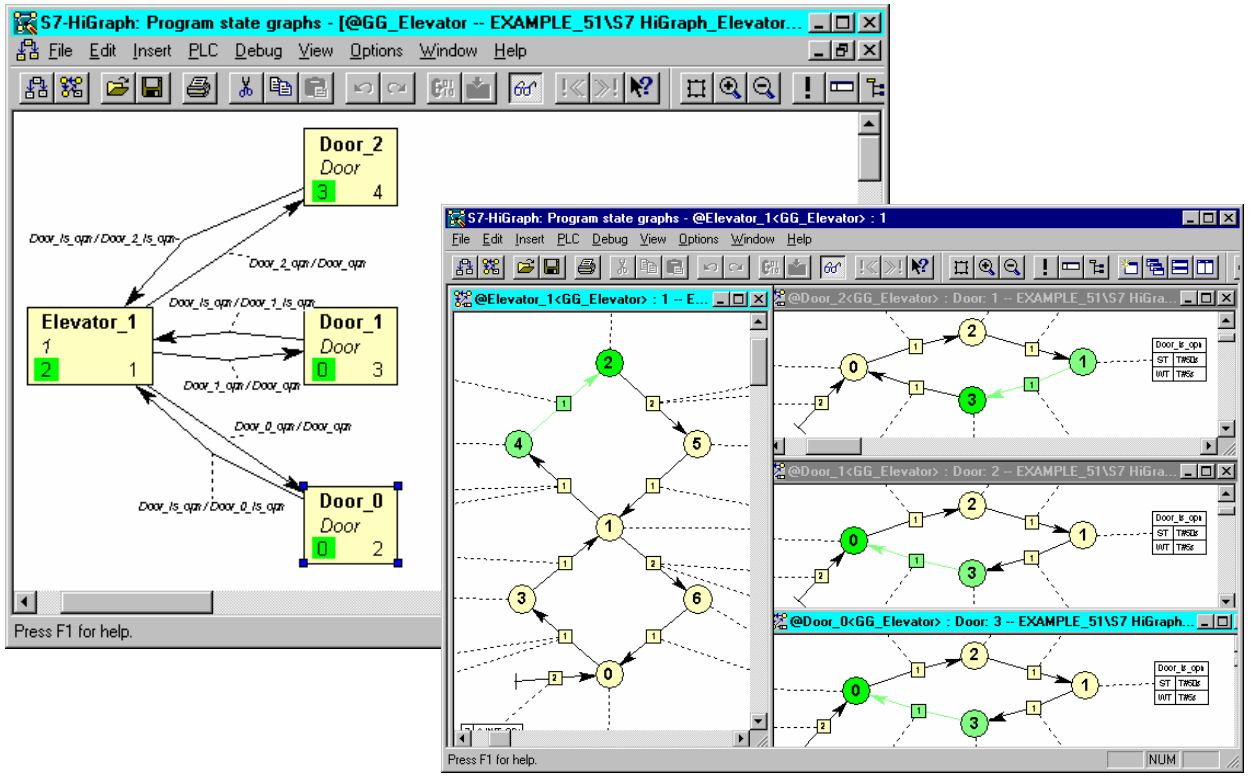
To compile a graph group, you take the following steps:

1. First select the menu options *Options > Settings for graph groups/state graphs* and enter the names for the FC and DB in the "Compile" tab as well as the setting "No diagnosis" in the "Diagnose" tab.
2. In the graph group window select the menu options *File -> Compile*.
3. Watch for any error messages in the message window. To jump to the position of the error, simply double-click the error message.
4. Compile the graph group again.

Calling the FC

To enable the HiGraph program to run in the CPU, the FC must be called in a block that is executed cyclically (OB1, for example) and the initialization parameter INIT_SD must be assigned.

Debug Functions in S7-HiGraph



Date: 30.06.2011
File: PRO2_12E.36

Overview

The monitoring functions enable you to monitor and check a program when it is executing in the CPU.

HiGraph provides the following debugging and monitoring functions:

- Monitoring the program status
- Monitoring and modifying variables (as in STL/LAD/FBD)
- Evaluation of reference data (as in STL/LAD/FBD)

Program Status

You can use the program status monitoring function to check the execution of all the instances in a graph group. The execution of the individual states and transitions is highlighted in color, and information about the instruction tables is also displayed.

The following program status monitoring facilities are available in the various HiGraph windows:

- Graph group window: Here you can see the status of all the instances in the graph group. The current state is shown in each instance.
- State graph window: Here you can obtain detailed status information for a selected instance (current state, transition, etc.).

Procedure

To start monitoring the program status, proceed as follows:

1. With the graph group open, select the menu options *Debug ->Monitor*. The status overview for the graph group is displayed.
2. Select one or more instances and select the menu options *Edit -> Open Object* or double-click the instance. Each instance you have selected is opened on-line and detailed status information is displayed.
4. To monitor more instances, change to the status overview and click the instance you want.
5. To stop monitoring the program status, deactivate the menu option *Debug -> Monitor.*

Programming in the S7- SCL High-Level Language

High-level language for writing PLC programs

- **Compatible to IEC 61131-3 Text (ST=Structered Text))**
- **Certified to PLCopen Base Level**
- **Contains all typical elements of a high-level language, such as operators, expressions, control statements**
- **PLC functions are integrated, such as access to I/O, timers, counters...)**

Advantages:

- **Well structured, easy-to-read program**
- **For complex algorithms, larger amounts of data**

```
FUNCTION_BLOCK Integrator
VAR_INPUT
   Init      : BOOL;    // Reset output value
   x         : REAL;    // Input value
   Ta        : TIME;    // Sampling time in ms
   Ti        : TIME;    // Integration time in ms
   ulim      : REAL;    // Output value upper limit
   llim      : REAL;    // Output value lower limit
END_VAR

VAR_OUTPUT
   y   : REAL:= 0.0;    // Initialize output value with 0
END_VAR

BEGIN
   IF TIME_TO_DINT(Ti) = 0 THEN      // Division by ?
      OK := FALSE;
      y  := 0.0;
      RETURN;
   END_IF;
   IF Init THEN
      y:= 0.0;
   ELSE
      y := y+TIME_TO_DINT(Ta)*x/TIME_TO_DINT(Ti);
      IF y > ulim THEN y := ulim; END_IF;
      IF y < llim THEN y := llim; END_IF;
   END_IF;
END_FUNCTION_BLOCK
```


S7-SCL

SCL (Structured Control Language) is a PASCAL-similar high-level textual language. It simplifies the programming of mathematical algorithms and complex data processing tasks for PLCs.

SCL therefore also enables S7 PLCs to be used for more complex tasks such as closed-loop control or statistical evaluation.

The SCL program is created and stored in an SCL source (source file). Executable blocks are then generated during compilation.

SCL is compatible with the ST (Structured Text) language defined in IEC 61131-3 and has PLCOpen certification (Base Level)

Functionality

SCL offers the functional scope of a high-level language such as:

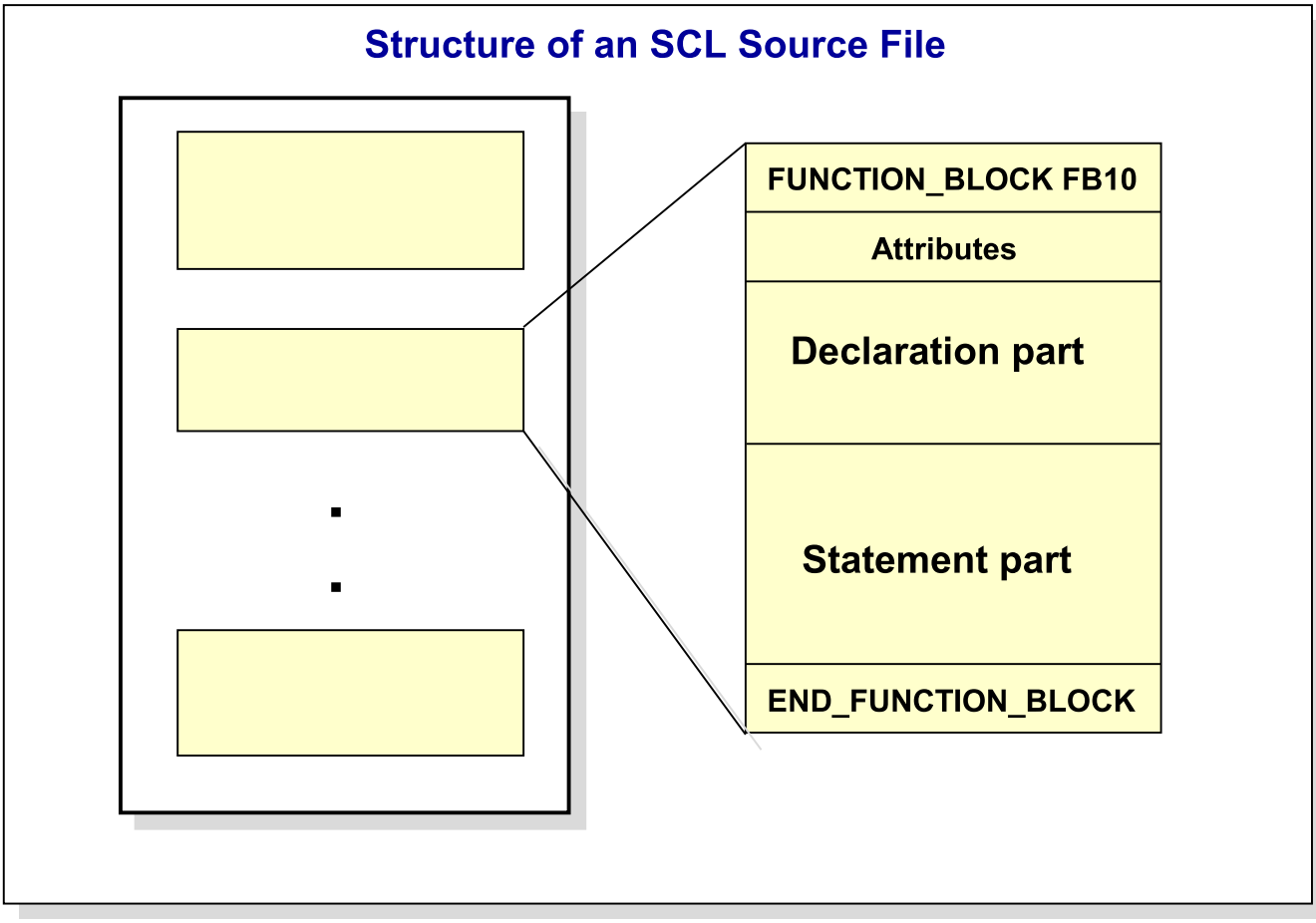
- Loops
- Alternatives
- Branch distributors, etc.

combined with PLC-specific functions such as:

- Bit accesses to the I/O, bit memories, timers, counters etc.
- Access to the symbol table
- STEP 7 block access

Advantages of SCL

- Simple to learn programming language, especially for beginners
- Simple to read (understandable) programs are created.
- Simpler programming of complex algorithms and processing of complex data structures
- Integrated debugger for symbolic debugging of the source code (single-step, breakpoints, etc.).
- System integration in S7 languages, such as STL, LAD and FBD.
- Relatively easy for PLC technicians to understand through similarity with S7 languages.



SIMATIC S7

Date: 30.06.2011
File: PRO2_12E.38

SITRAIN Training for Automation and Drives

Structure of an SCL Source File

An SCL source file can include as many blocks as you like (OBs, FBs, FCs, DBs and UDTs).

Structure of a Block

Within the SCL source file, every individual block, dependent on the block type, is framed by a standard identifier for the beginning and for the end of the block. The body of the block itself consists of the declaration part and the statement part.

As an option, an attribute part can be inserted between the identifier for the beginning of the block and the declaration part.

Attributes

Attributes identify block properties, that can also be displayed within the SIMATIC Manager after compilation via the command *Edit -> Object Properties*.

Declaration Part

The local variables, block parameters, constants and jump labels are declared in the declaration part of a block.

Statement Part

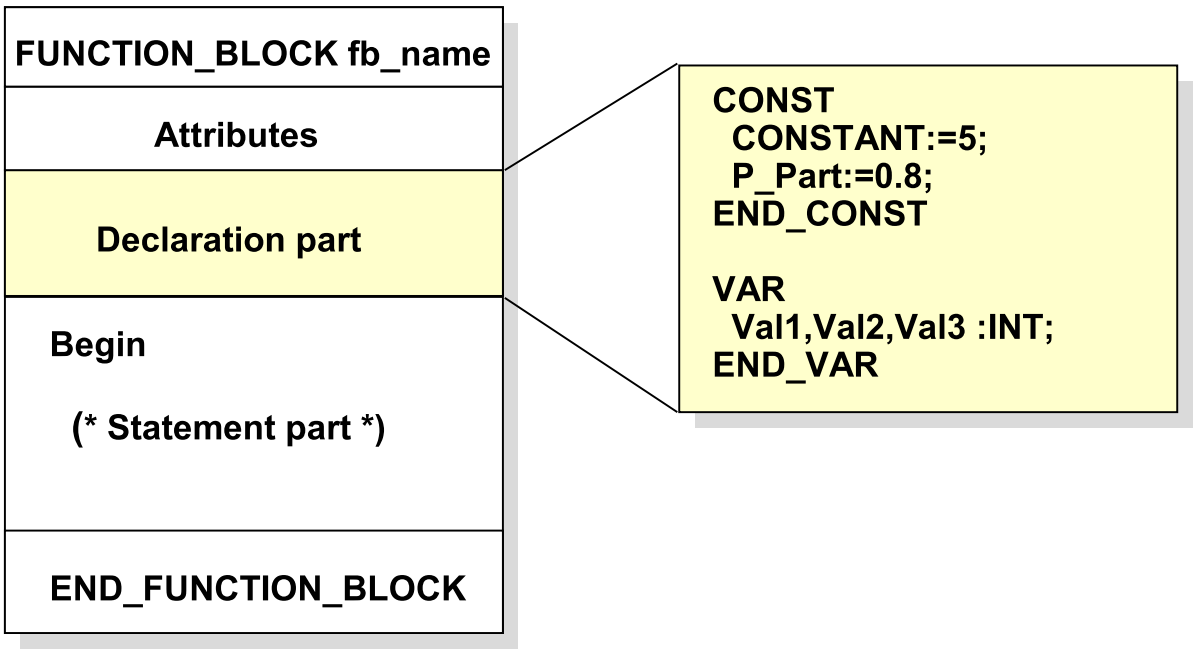
The statement part contains the individual statements to be executed.

Block Sequence

So that your SCL source file can be compiled, you must pay attention to the following in regard to the sequence of the blocks:

Called blocks must always be located before the calling blocks.

The Declaration Part of a Block



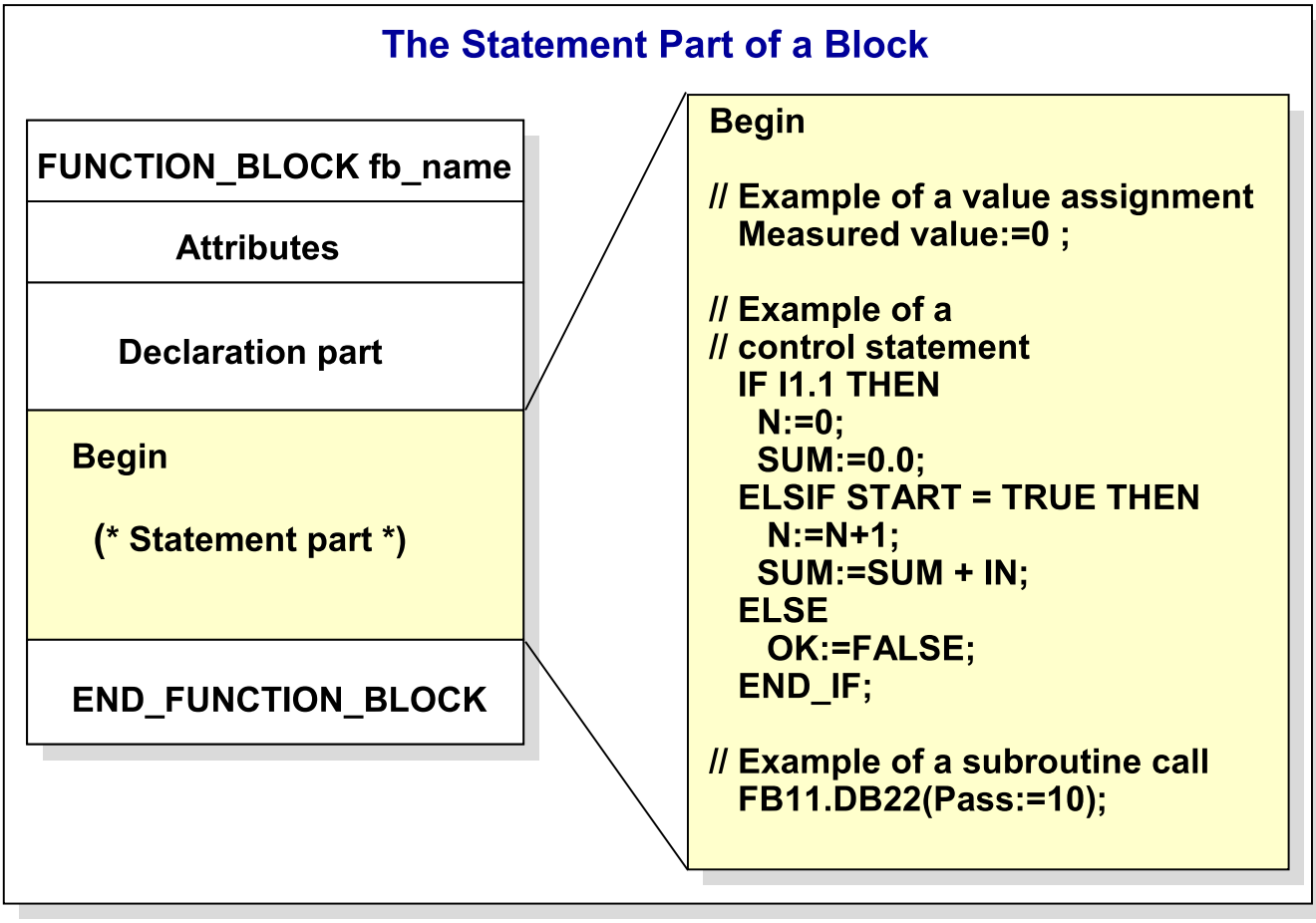
Structure

The declaration part is used to define the local and global variables, block parameters, constants and jump labels. It is divided into individual declaration blocks, that are in each case identified by their own keyword pair.

The blocks can be inserted into an SCL source file via *Insert -> Block Template -> Constant, Parameter*.

Blocks

Data	Syntax	FB	FC	OB	DB	UDT
Constants	CONST *Declaration list* END_CONST	X	X	X		
Jump Labels	LABEL *Declaration list* END_LABEL	X	X	X		
Temporary Variables	VAR_TEMP *Declaration list* END_VAR	X	X	X		
Static Variables	VAR (STRUCT) *Declaration list* END_VAR	X			(X)	(X)
Input Parameter	VAR_INPUT *Declaration list* END_VAR	X	X			
Output Parameter	VAR_OUTPUT *Declaration list* END_VAR	X	X			
In/Out Parameter	VAR_IN_OUT *Declaration list* END_VAR	X	X			



Statement Part

The statement part contains instructions that are executed after the call of a logic block (OB, FB, FC). These statements are used to process data and addresses or - in the case of data blocks - to preset individual values within the DB.

Subdivision

The individual statements can essentially be divided into three groups:

- Value assignments: they are used to assign an expression or a value to a variable.
- Control statements: they are used to branch within a program or to repeat groups of instructions.
- Subroutine call: they are used to call functions and function blocks.

Note

You must pay attention to the following points when programming statements:

- The statement part begins with the keyword BEGIN and ends with the keyword for block end (e.g. END_FUNCTION).
- Every statement must be closed with a semicolon.
- All identifiers (names) used in the statement part must be declared.

Templates

Templates for control structures can be inserted in an SCL source file via *Insert -> Control Structure ->IF, CASE, FOR, WHILE, REPEAT*.

Expressions, Operators and Operands in S7-SCL

Expressions

- **Mathematical expressions** **((3+CONST_INT) * (VAR_INT ** 37) / 3.14)**
- **Comparison expressions** **A >=9**
- **Logical expressions** **(n >5) AND (n < 20)**

Operators

- **Assignment operator** **:=**
- **Mathematical operators** ***, /, MOD, DIV, +, -, ****
- **Comparison operators** **<, >, <=, >=, = <>**
- **Logical operators** **NOT, AND or &, XOR, OR**

OPERANDS

- **Constants** **30. 0, FACTOR, 'SIEMENS'**
- **Extended variables** **Status, IB5, DB10.DW5, Motor.Current, FC12(A:=On)**
- **Expression in (...)** **((3+CONST_INT) * (VAR_INT ** 37))**

SIMATIC S7

Date: 30.06.2011
File: PRO2_12E.41

SITRAIN Training for Automation and Drives

Expressions

Expressions consist of operands, operators and round brackets. Within an expression the operators (e.g. +, -, *, /, etc.), that is, the active components of an expression, are linked with the passive elements, such as constants, variables and function values, in order to form a new value.

An expression therefore stands for the value it represents.

SCL permits the formation of standard expressions, that is, mathematical, logical and comparative expressions. Variables from data blocks, arrays, structures and CPU memory areas (inputs, outputs, etc.) can be enlisted for the formation.

Operators and Operands

Expressions consist of operators and operands. Most SCL operators link two operands (e.g. A +B) and are therefore termed *binary* operators. The others work with only one operand and are thus called *unary* operators.

The result of an expression can

- be assigned to a variable (e.g. A := B + C;)
- be used as a condition for a control statement (e.g. IF A< B DO ...)
- be used as an actual parameter for the call of a function or a function block (e.g. FB20 (Input := A + B))

Statements in S7-SCL

Value assignments

- **Example:** A := B + C;

Control statements

- **IF statement** IF I 1.1 THEN ... ELSIF ... ELSE ... END_IF
- **CASE statement** CASE SELECTOR OF 1: ...; 2: ... ELSE: ... END_CASE
- **FOR statement** FOR INDEX := 1 TO 49 BY 2 DO ... END_FOR
- **WHILE statement** WHILE INDEX <= 50 DO ... END_WHILE
- **REPEAT statement** REPEAT ... UNTIL INDEX:= 51 ... END_REPEAT
- **CONTINUE statement** WHILE BOOL_1 DO ... CONTINUE ... END_WHILE
- **EXIT statement** WHILE BOOL_1 DO ... EXIT ... END_WHILE
- **GOTO statement** IF INDEX <23 THEN GOTO MARK; ...
- **RETURN statement** IF ENABLED THEN RETURN; ...

Function block and function calls

- **FB or SFB call** FB11.DB20(IN:=VAL1, BY:=VAL2);
- **FC or SFC call** RETURN := FC32(IN:=VAL1,OUT:=VAL2);

SIMATIC S7

Date: 30.06.2011
File: PRO2_12E.42
SITRAIN Training for Automation and Drives

Statements

In the declaration part of a block, the actions are described that are to be executed with the variables introduced in the declaration part as well as with global data. Normally, the statements are executed in the sequence in which they are listed in the program text.

Value Assignments

These are used for assigning new values to variables. The variable's old value is then lost.

Control Statements

These are used for changing the sequence in which the statements are normally processed.

A choice from the various alternatives in program execution can be made with conditional statements (IF and CASE statements).

Loop statements (FOR, WHILE and REPEAT statement) are used to repeatedly execute statements.

Jump statements (CONTINUE, EXIT and GOTO statements) permit the sequence of processing to be interrupted and to jump to a resumption point.

FB and FC Calls

According to the principle of structured programming, other functions (FC and SFC) and function blocks can also be called from an SCL block. Blocks that can be called are:

- additional functions and function blocks, that were generated in SCL or in another STEP 7 language (STL, LAD, etc.).
- standard functions and standard function blocks supplied with SCL.
- system functions (SFC) and system function blocks (SFB) that are available in your CPU's operating system.

Value Assignments in S7-SCL

Local Variables

- **Elementary data type** — `COUNTER := (5 + RUNVAR) * 2;`
- **Structures**
  - **Complete structure** — `STRUCT_1 := STRUCT_2;`
  - **Components** — `STRUCT_1.COMP3 := STRUCT_2.COMP1;`
- **Array**
  - **Complete array** — `ARRAY_1 := ARRAY_2;`
  - **Components** — `ARRAY_1[I] := ARRAY_2 [J];`

Global Variables

- **CPU memory areas**
  - **Absolute access** — `VALUE := IW10;`
  - **Symbolic** — `VALUE := INPUT;` `// "INPUT" in symbol table`
  - **Indexed** — `VALUE := IW[INDEX];`
- **Data blocks**
  - **Absolute access** — `VALUE := DB11.DW5;`
  - **Symbolic** — `VALUE := MOTOR.CURRENT;` `// MOTOR and CURRENT must`
  - **Indexed** — `VALUE := MOTOR.DW[Index];` `// be in the symbol table`
  - **Via input parameters** — `VALUE := I_PAR.DW[Index];` `// I_PAR is decl. as VAR_IN`

Date: 30.06.2011
File: PRO2_12E.43

Principle

Value assignments replace a variable's present value with a new value, which is specified in an expression. This expression can also contain identifiers of functions (FC), which are thereby called and return corresponding values.

The value of an expression assigned to a variable must be compatible with the variable's type.

Value Assignments with Complex Variables

A complex variable represents either the complete type (the complete structure,
the complete array, the string) or a component of the complex variable. There are thus two possibilities for assignment to a complex variable. You can

- Assign the contents of another complete complex variable (structure, array, or string) to each complex variable (structure, array, string).

  Please note that, for example, a complete structure can only be assigned to another structure if the structure components coincide in their data type as well as in their name.

  A complete array can only be assigned to another array if the component's data types as well as the array limits coincide exactly.
- Assign a type-compatible variable, a type-compatible expression or another component to each component of a complex variable.

The IF Statement in S7-SCL

Syntax

```
IF        <expression> THEN <statements>
[ELSIF    <expression> THEN <statements>]      //optional
.
.
[ELSE   <statements>]                          //optional
END_IF
```

Example

```
IF INPUT_OK THEN
   N := 0;
   SUM := 0.0;
   OK := FALSE;                                // Set OK flag to FALSE
ELSIF START_OK THEN
   N := N + 1;
   SUM := SUM + IN;
ELSE
   OK := FALSE;
END_IF;
```


Principle

The problem often occurs in programs, that various statements are to be executed dependent on specific conditions. It is possible, with program branches, to branch the program flow into alternative statement sequences.

The IF statement is a conditional statement. It offers one or more options and selects one of its statements for execution (or none, if applicable).

Execution

The IF statement is processed according to the following rules:

1. If the value of the first expression is TRUE, then the statement part after THEN is executed, otherwise, the expressions in the ELSIF branches are evaluated.
2. If no Boolean expression is TRUE in the ELSIF branches, the statement sequence for ELSE is executed (or no statement sequence, if the ELSE branch does not exist).

Any number of ELSIF statements may exist. You must note that the ELSIF branches and/or the ELSE branch may be missing. These cases are handled as if these branches existed with empty instructions.

Note

The use of one or more ELSIF branches as opposed to a sequence of IF statements offers the advantage that the logical expressions that follow a valid expression are no longer evaluated. The run-time of a program can thus be shortened.

The WHILE Statement in S7-SCL

Syntax

```
WHILE    <expression> DO <statements>
END_WHILE
```

Example

```
FUNCTION_BLOCK SEARCH          // SEARCH is declared in the symbol table
VAR
  INDEX              : INT;
  KEYWORD            : ARRAY[1..50] OF STRING;
END_VAR

BEGIN
  INDEX := 1;
  WHILE INDEX <= 50 AND KEYWORD[INDEX] <> 'KEY'
    DO
    INDEX := INDEX + 2;
  END_WHILE;
END_FUNCTION_BLOCK
```


Principle

The WHILE statement permits the repeated execution of a sequence of statements on the basis of an execution condition. The execution condition is formed according to the rules of a logical expression.

The statement part that follows DO is repeated as long as the execution condition has the value TRUE.

Execution

The WHILE statement is processed according to the following rules:

1. The execution condition is evaluated before every execution of the statement part.
2. If the value TRUE occurs, then the statement part is executed.
3. If the value FALSE occurs, the execution of the WHILE statement is concluded. This can already be the case with the first evaluation.

Calling Function Blocks

Call as global instance

- **Absolute call**

```
FB10.DB20(X1 := 5, X2 := IW12, ...);      (*  Call FB10 with instance data
                                              block DB20 *)
```

- **Symbolic call**

```
DRIVE.ON(X1 =5, X2 := IW12,...);          (*  DRIVE and ON are declared in
                                              the symbol table *)
```

Call as local instance

- **Call using identifier**

```
VAR
   MOTOR              : FB10;
END_VAR

BEGIN
 . . .
MOTOR(X1 := 5, X2 := IW12,...);           (*  Call as local instance is
                                              possible within other
                                              function blocks   *)
```

SIMATIC S7

Date: 30.06.2011
File: PRO2_12E.46

SITRAIN Training for Automation and Drives

FB Calls

The multiple instance concept offered by STEP 7 is also available in SCL. You can therefore call FBs referencing an associated instance data block as well as call FBs using the multiple instance model.
The call of an FB as a multiple instance differs from the call with separate DB in the storage of the data.

In the case of a multiple instance, the necessary data area is not in a separate instance DB, but is embedded in the instance data area of the calling FB.

Call with Instance DB

The call is made in a statement specifying:

- the name of the function block or system function block
- the instance data block (DB identifier)
- as well as the parameter assignment (FB parameters)

You can use either absolute or symbolic addresses in the call with a separate instance DB. You can call FBs with a separate instance DB in all logic blocks (OB, FB, FC).

Call as Multiple Instance

The call is made in a call statement specifying:

- the local name of the instance (identifier)
- as well as the parameter assignment (FB parameters).

The call of an FB as a local instance is only possible in FBs. The multiple instance must also be declared as a variable of data type FBx in the declaration part (VAR ...END_VAR) of the calling FB.

The "OK" Flag for Error Evaluation

Global bit for error detection
(copied to the BR bit at the end of the block)

Example:

```
                          // Set OK variable to TRUE to enable
                          // a check to be made to see whether
                          // the following actions are performed
                          // correctly

OK := TRUE;
SUM := SUM + IN;
IF OK THEN                // Addition was performed correctly
...
ELSE                      // Overflow in addition
...
END_IF;
```


Description

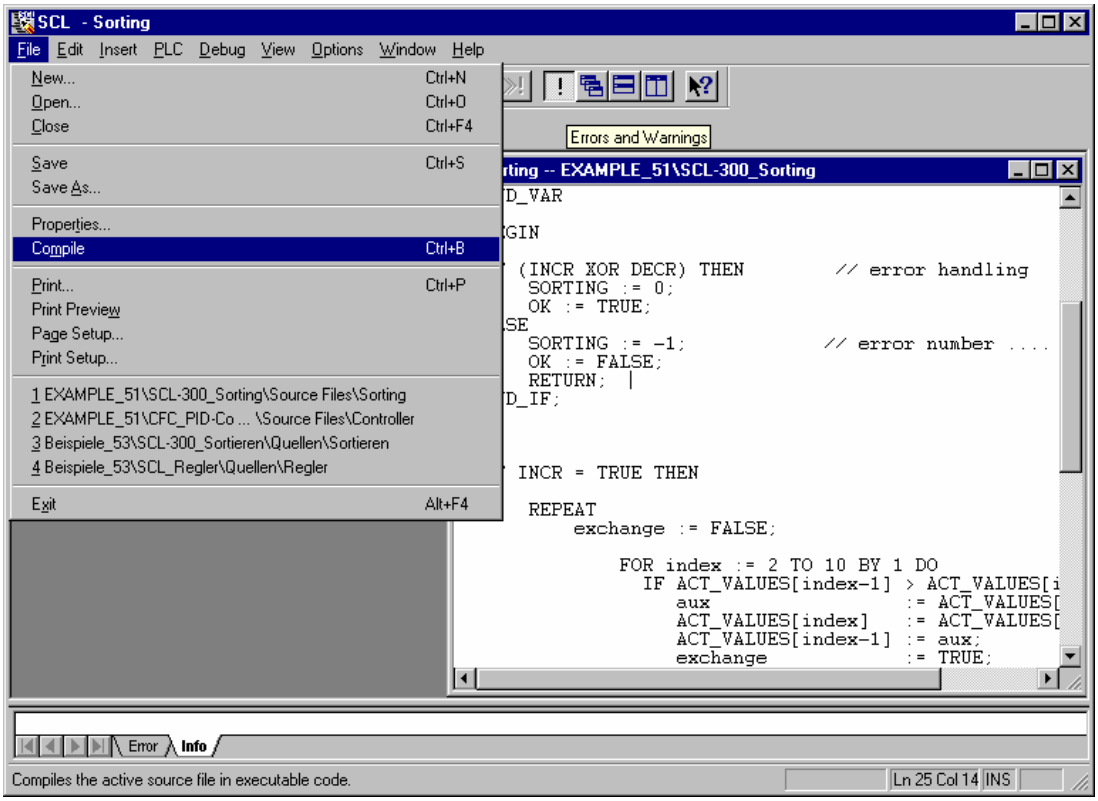
SCL provides a global variable of the BOOL type, known as the OK Flag, for error identification within logic blocks. This flag is used to identify correct or faulty executions of statements and to react accordingly.

The Way it Works

If an error occurs during the execution of a statement (e.g. overflow), the OK flag is set to FALSE by the system. Upon exiting a block, the OK flag is then copied into the BR bit of the CPU status word and can then be evaluated with the OK flag of the calling block.

The OK flag is set to the value TRUE at the beginning of a program execution. It can be checked anywhere in the block or be set to TRUE or FALSE.

Compiling an SCL Source File



Compiling

Before you can run or test your program, you must compile it. You activate the compiling function via the *Compile* option in the *File* Menu or via the icon in the toolbar.

The compiler has the following properties:

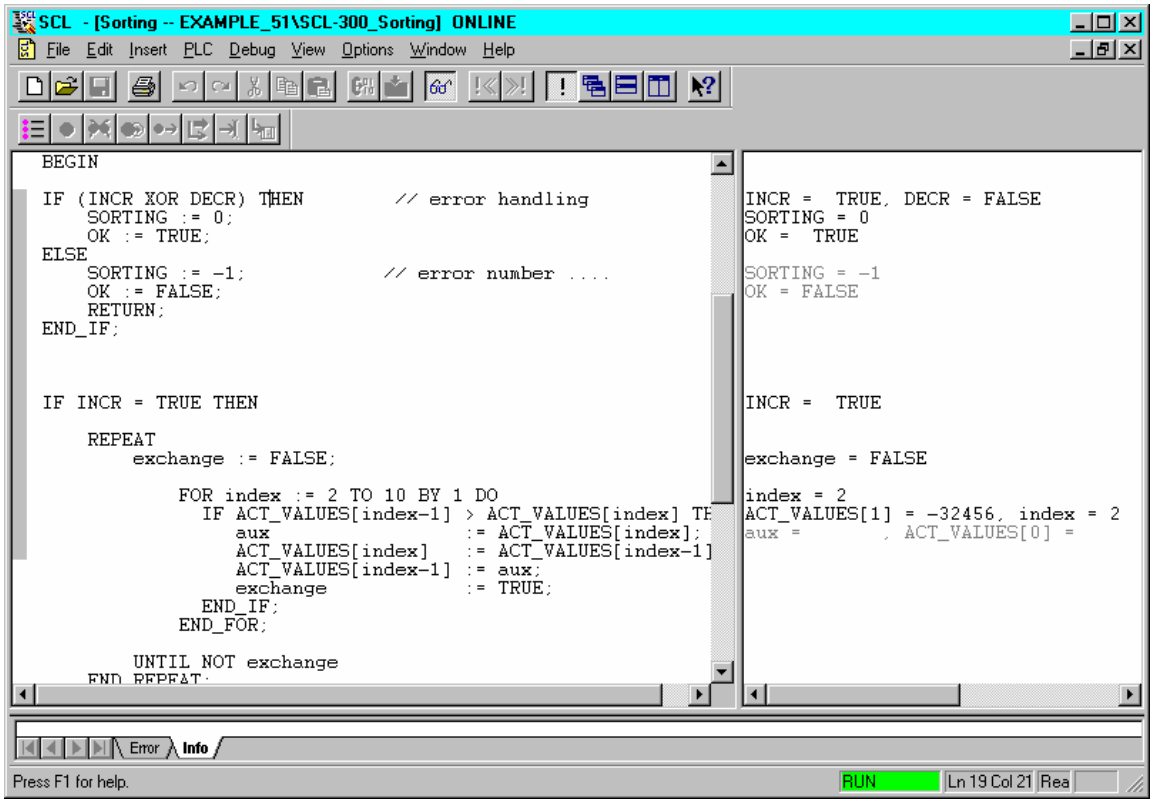
- The compiler operates in batch mode. That is, it processes an SCL source as a unit. The compilation of individual blocks in an SCL source file is not possible.
- The compiler checks the syntax of an SCL source file and subsequently displays all errors that occurred during compilation.
- The compiler creates STL statements or test information, if the SCL source is error-free and the corresponding options are set. You must select the Create Debug Info option for every program that you subsequently want to test in a high-level language.
- The compiler generates for every function block call an associated instance data block.

Settings

You adapt the compiling function via the entry *Options -> Customize -> Compiler*:

- Create Object Code: You use this option to specify whether or not you want to generate executable code. If you activate the Compile function without selecting this option, only a syntax check is performed.
- Optimize Object Code: Create shorter code
- Monitor Array Limits: Array indexes are checked for allowed range at runtime. If an array index is outside the permissible range, the OK flag is set to FALSE.
- Create Debug Info: Generate debug information for high-level language debugger
- Set OK Flag: This option enables the OK flag to be checked in an SCL source file.

Continuous Monitoring



Selection

The Debugger function *Monitor Continuously* can be selected as follows:

1. Make sure that the program has been compiled with the options "Create Object Code" and "Debug Information" activated.
2. Select the window containing the source of the program to be tested.
3. Position the cursor in the line of the source text containing the first statement of the section you want to test.
4. Select the menu options *Debug -> Monitor Continuously.*

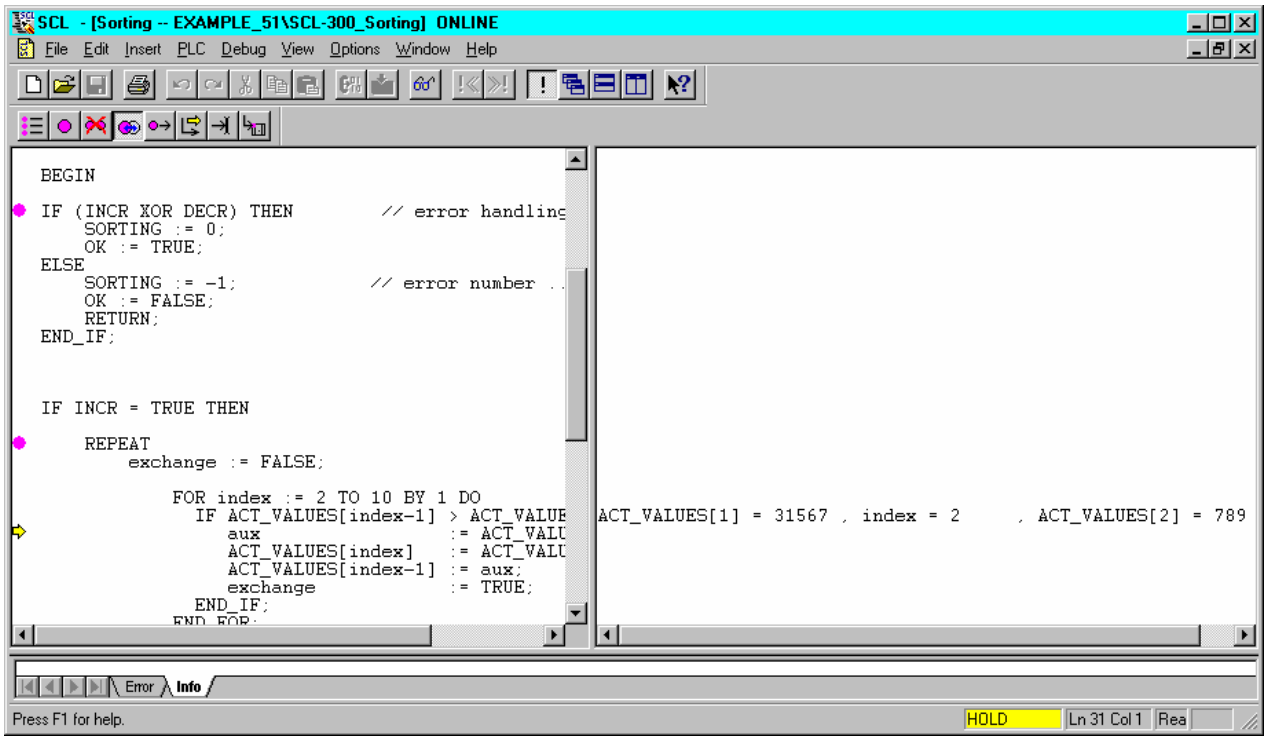
   The largest area that can be monitored is determined and indicated by a grey bar at the left-hand edge of the window. The names and current values of the variables in the area being monitored are displayed in the right-hand section of the window.
5. Select the menu options *View -> Symbolic Representation* to activate or deactivate the display of symbolic names in the program.
6. Select the menu options *Debug -> Monitor Continuously* if you want to interrupt monitoring.
7. Select the menu options *Debug -> Finish Testing* to stop monitoring.

Debug Mode

You can change the monitoring area with the entry *Debug -> Test Environment*:

- Process: In this test environment, the SCL debugger reduces the maximum monitoring area so that the cycle time is not or only negligibly prolonged.
- Laboratory: In the Laboratory test environment, the monitoring area is only restricted by the capacity of the CPU. The maximum monitoring area is larger than in the Process test environment.

Setting and Editing Breakpoints



Overview

When you select the debugging mode "Breakpoints Active", you can monitor your program step by step. You can execute the program one statement at a time and observe the changes in the contents of the variables processed.

After setting breakpoints, you can execute the program up to a breakpoint and monitor it step by step from there on.

Setting Breakpoints

You set breakpoints as follows:

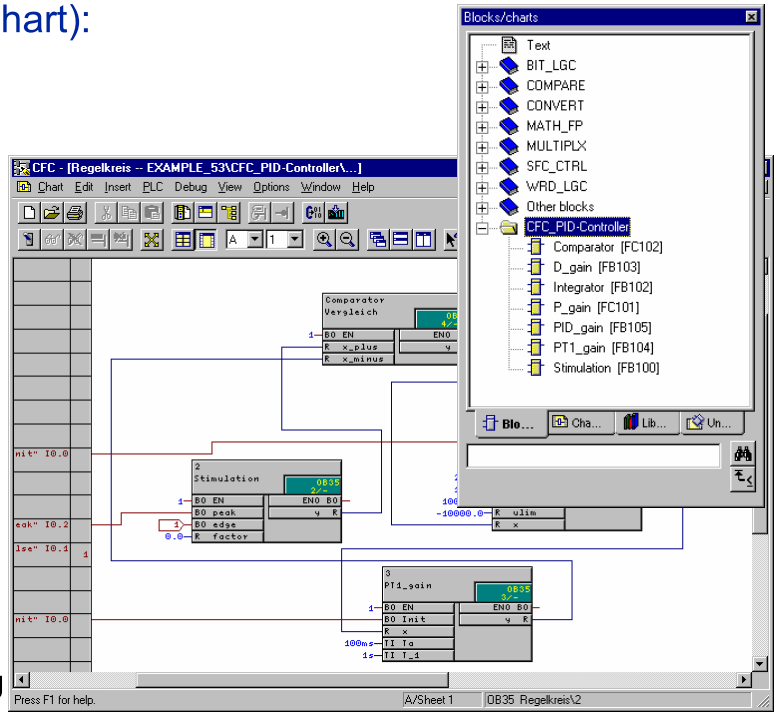
1. Open the SCL source of the program you want to test.
2. Define the breakpoints by positioning the cursor to the required place and selecting the menu options *Debug -> Set Breakpoint*. The breakpoints appear as red circles at the edge of the window.
3. Select the menu options *Debug -> Breakpoints Active*. The window is vertically divided into two halves. When the next breakpoint is reached, the CPU goes into HALT mode, the red breakpoint is marked with a yellow arrow.
4. To continue:
   - Select the menu options *Debug -> Execute Next Statement*. When the CPU has processed the next statement, it goes into HALT mode again.
   - Select the menu options *Debug -> Continue*. When the CPU reaches the next breakpoint, it goes into HALT mode again.
   - Select the menu options *Debug -> Execute To Cursor*. When the CPU reaches the selected point in the program, it goes into HALT mode again.
   - Select the menu options *Debug -> Execute Call* if the current line contains a block call. If the block that is called is an SCL generated block, then the menu option Execute Call causes the block to be opened and the program to halt at the first statement of the block.

     If the block that is called is not an SCL generated block, the call is jumped over and the following program line is selected.

CFC for SIMATIC S7 and SIMATIC M7

CFC (Continuous Function Chart): Graphical tool for writing PLC programs

- **You position the blocks on a drawing sheet and interconnect them**
- **Interconnections are possible:**
  - **between I/O fields**
  - **to blocks in other charts**
- **Sources and destinations are managed in the margin bar**



Advantages

- **Programming for process engineers**
- **Speeds up writing, debugging and startup**

Date: 30.06.2011
File: PRO2_12E.51

Overview

The CFC (Continuous Function Chart) engineering tool enables you to create automation applications for SIMATIC S7 or SIMATIC M7 by drawing a process flow chart - similar to a function diagram for programming a PLC.

In this graphical programming method, you position the blocks in an area like a sheet of drawing paper and interconnect them graphically. With CFC you can quickly and easily convert technological requirements into executable automation programs.

Scope of Supply

The scope of supply of CFC includes:

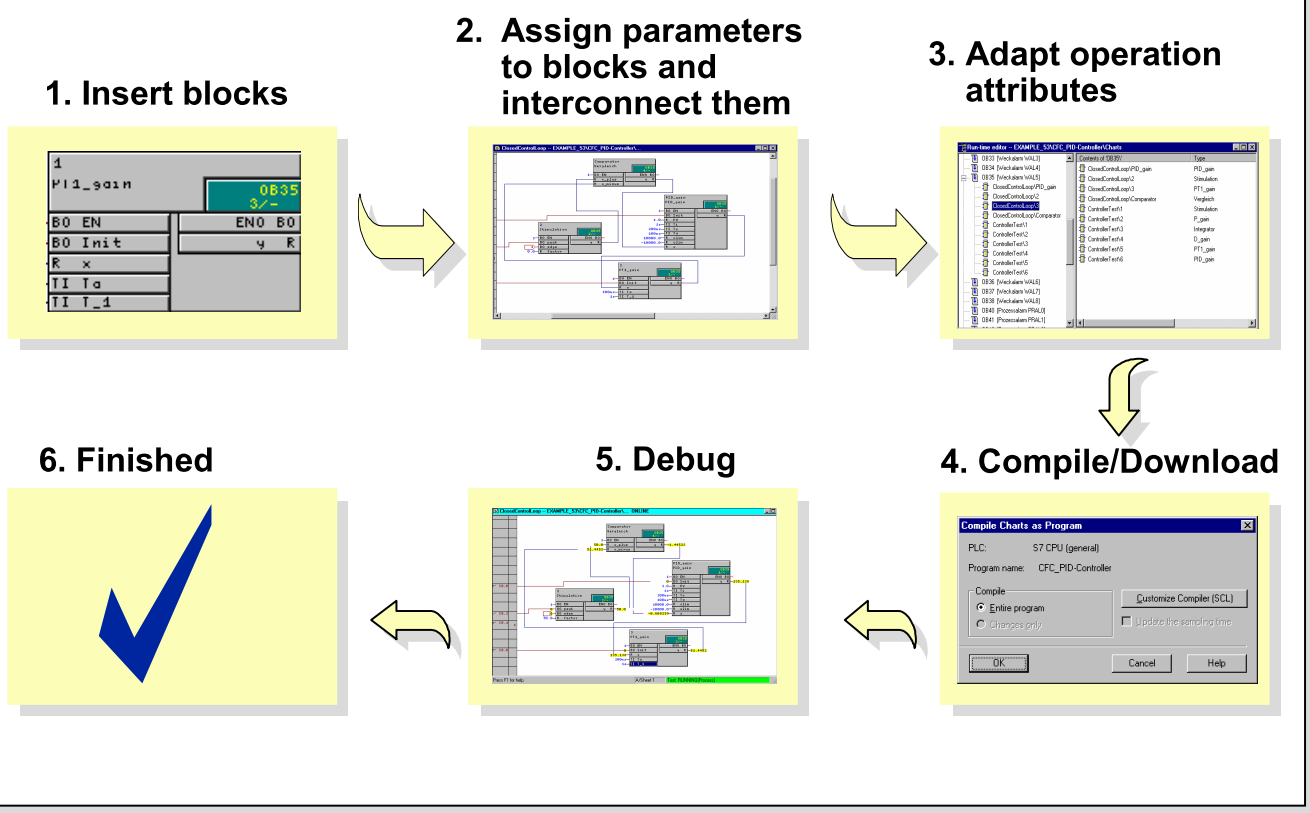
- CFC Editor
- Code generator
- Debugger
- Standard block libraries

Customer Benefits

The CFC product is smoothly integrated into the STEP 7 architecture as an optional package with a unified look&feel, and with universal data management. This makes CFC easy to use and easy to learn, and provides consistent data.

- CFC can be used for simple tasks as well as for very complex task definitions.
- Simple interconnection technology makes communication between blocks user-friendly to configure.
- Manual handling and management of machine resources is no longer necessary.
- User-friendly testing and debugging are supported.

Configuring CFC Applications instead of Programming



SIMATIC S7

Date: 30.06.2011
File: PRO2_12E.52

SITRAIN Training for Automation and Drives

What is CFC ?

CFC is a graphic Editor based on the STEP 7 software package. It is used to create a complete software structure for a CPU from prefabricated blocks.

How It Works

You work with graphic means in the CFC Editor. The blocks are selected from the available block libraries, placed in a chart and interconnected with one another by mouse click. You don‘t have to worry about details such as algorithms. Instead you can concentrate on the technological aspects of the configuration.

The runtime properties are predefined. They can, however, be adapted for every block. A considerable labor-saving feature is that individual blocks or entire groups of blocks can be copied or shifted across the charts. The connection of the blocks amongst themselves remains intact as fas as possible.

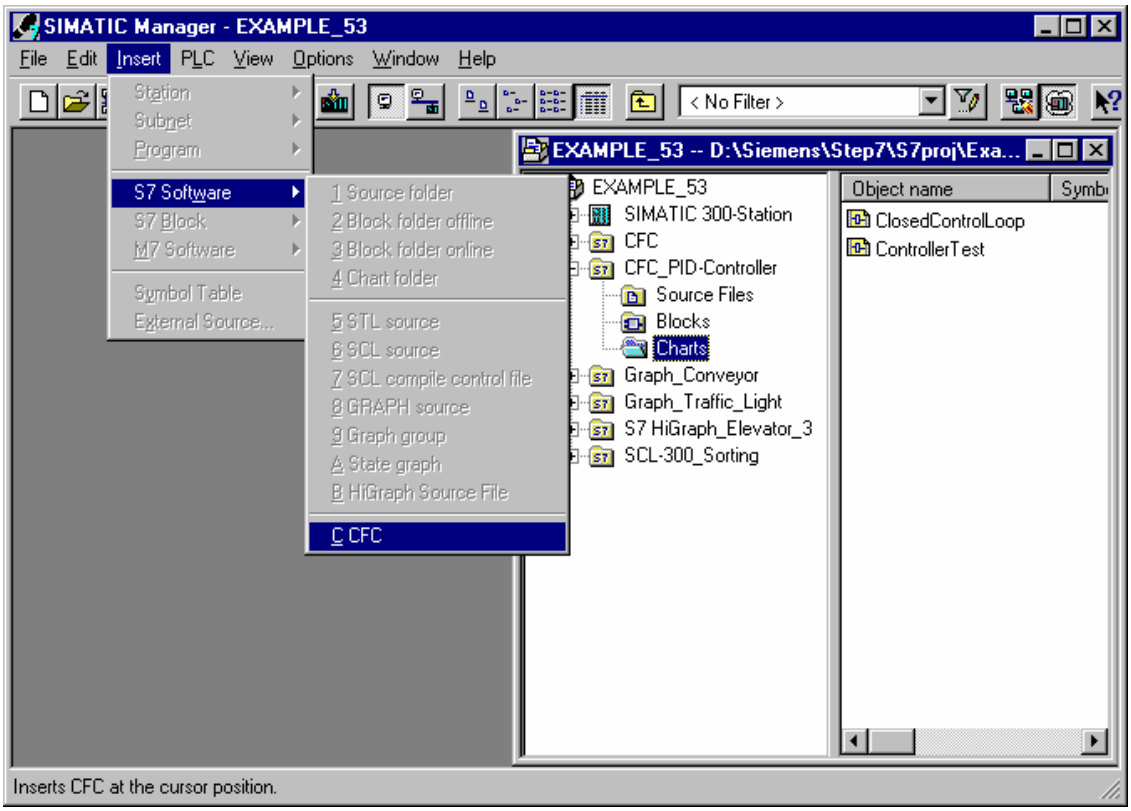
When all blocks have been connected and all functions have been created, an executable machine code is generated per mouse click. This machine code is loaded into the CPU and is tested online with the CFC.

PLCs

CFC can be used for configuring various PLCs, SIMATIC S7, SIMATIC M7, SIMADYN D among others.

For the user, the way CFC works is identical for the most part. Since there are differences, however, it must be clearly noted here that the executions in the following chapters refer to the use of CFC in SIMATIC S7.

Charts in the STEP7 Project



Requirements

To generate CFC charts, you require an S7 Program folder. Normally, S7 Program folders are assigned to a specific CPU. You can, however, also create a program directly underneath a project without assigning it to a hardware.

Charts Folder

When you generate an S7-Program folder, a charts folder is not automatically created. It must be explicitly inserted. When you have selected the S7 Program, there are several possibilities to make the insertion:

- shortcut menu: *Insert New Object -> Chart Folder*
- menu bar: *Insert -> S7 Software -> Chart folder*

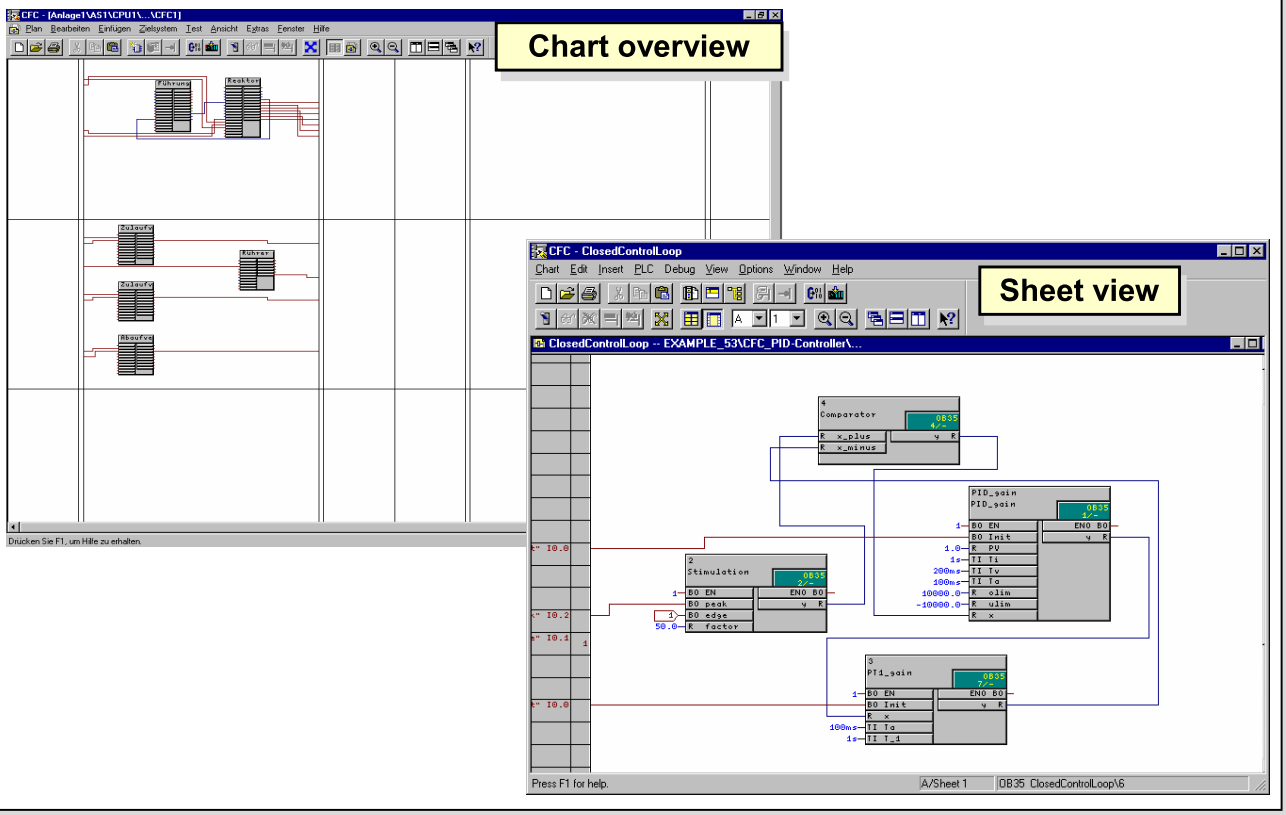
Generating a Chart

After the chart folder is available, you can create CFC charts in it. To do so, you select the chart folder and then choose:

- shortcut menu: *Insert New Object -> CFC*, or
- menu bar: *Insert -> S7 Software -> CFC*

The chart names have to be unique throughout the CPU. The SIMATIC Manager monitors this. Whenever names are identical, a sequential number is added on in order to reestablish a uniqueness.

The CFC Editor



Date: 30.06.2011
File: PRO2_12E.54

Editor

The Editor is a Windows 9x/NT application. Just like other Windows applications, it has elements such as menu, toolbar, statusbar and dialogs. The CFC Editor is started from the SIMATIC Manager as usual with a double-click on the symbol of the desired chart.

Chart Overview

After starting, the chart is displayed in the overview format, that is, the complete chart. This format is used for copying and shifting blocks as well as inserting large blocks.

Sheet View

Since many details cannot be shown in the overview, you switch to the sheet view for editing the blocks. This switch can be carried out in a number of ways:

- double-click in the required sheet. The position of the cursor determines the centering of the sheet view.
- selecting the sheet using *Edit -> Go To -> Sheet* (CTRL + E)
- selecting the sheet using the  button in the toolbar.

By double-clicking in the sheet, the Editor returns to the overview format.

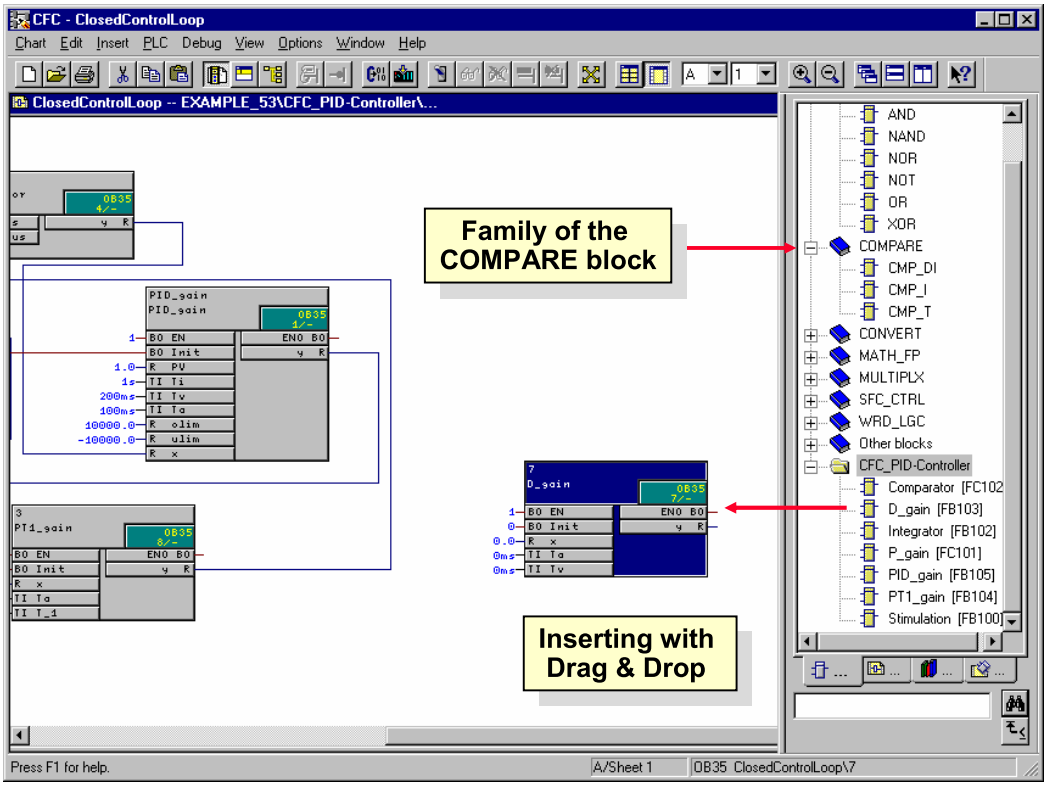
Sheets and Margins

Every chart consists of 6 sheets that are arranged in 2 columns of 3 sheets each. In turn, a sheet consists of a central workspace on which the blocks can be placed.
To the left and right of the workspace are the margins. The margins are needed when connections lead-in to or lead-out of the blocks in the current sheet.

Attention

All changes made in the CFC Editor are immediately saved and cannot therefore be reversed. When you work with the CFC Editor, it is recommended that you copy the program completely into a backup project. That way, you can access old versions at any time.

The Block Concept - Inserting Blocks



Date: 30.06.2011
File: PRO2_12E.55

Basic Principle

The basic principle of the block concept is:

"Don't reinvent the wheel"

The user software can be programmed, or, structured when using prefabricated software. Individual blocks can be assembled into a complete structure that fulfills a defined automation function.

The blocks represent the basis for configuring with CFC. When you use prefabricated blocks, it enables you to completely concentrate on the fundamental automation task, since no programming details have to be taken into consideration. The finished partial solutions merely have to be adapted to the given requirements.

The benefit of the concept lies in the high degree of re-usability. Blocks can be re-used in the same chart or in other charts.

Libraries

Libraries are groups of prefabricated blocks that are organized and summarized according to their functionality. An elementary library is provided along with the CFC. Other libraries with function blocks can be added as required.

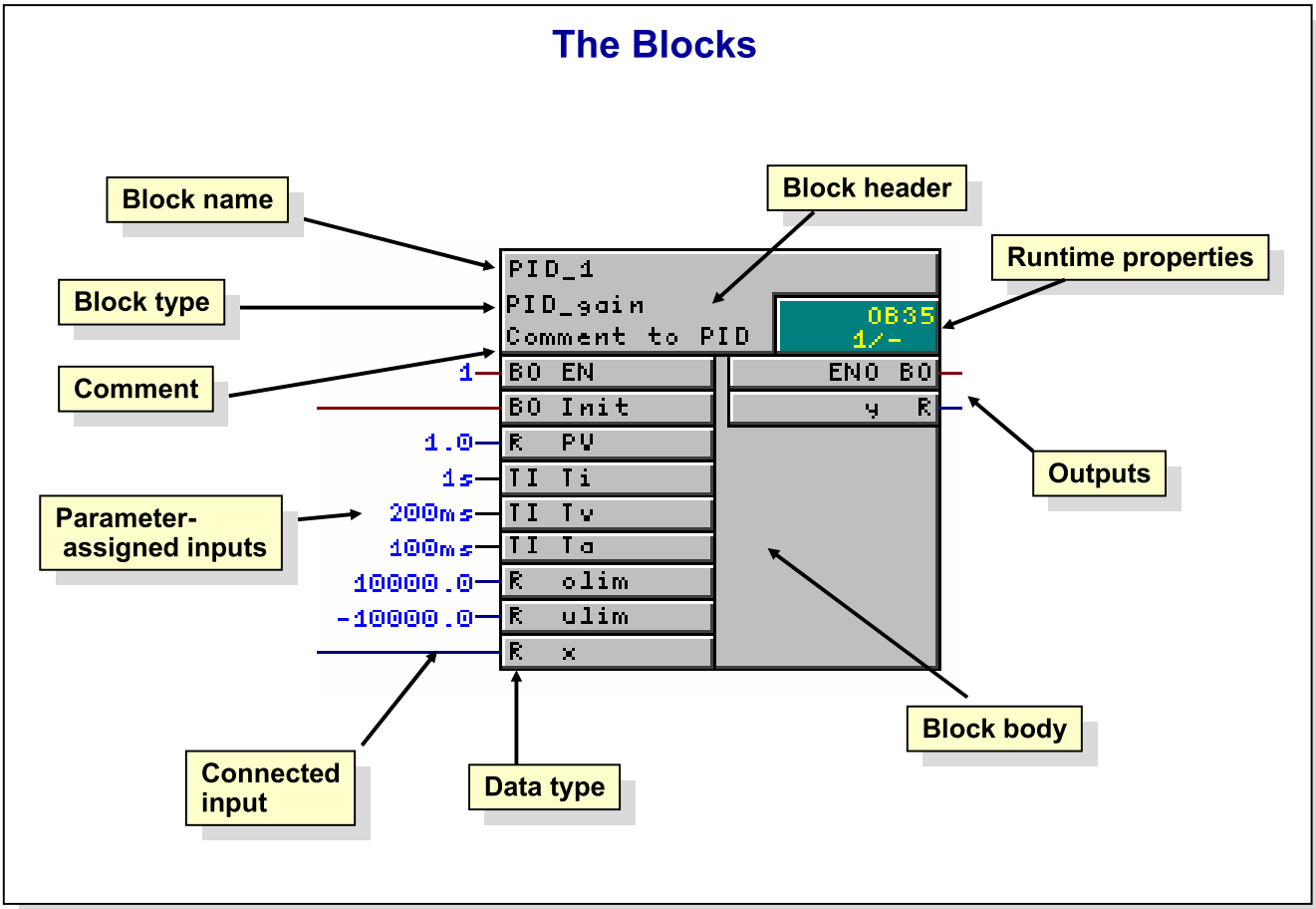
As well, you can create your own blocks with S7 Program Editors (S7-SCL for example) and gather them in their own library.

Inserting

To insert a block means to select a block type in the browser and to place it in the chart. The insert is handled easily using Drag & Drop. The block receives a chart-wide unique name in the form of a number during the insert.

Instances

The inserted block is an instance of the block type. That is, it has its own local data. You can generate as many instances as you like from every block type.



Block Types

A block type represents a template for any number of instances and describes how these instances are structured internally. All instances of a block type follow the same basic definition with regards to their behavior and their data structure.

Block Instances

A block instance is a concrete object which is generated according to its type description. The type describes the behavior and the data structure of an instance. The current state of an instance depends on the instructions actually executed and the contents of the information that is stored. Each instance has a unique identifier that guarantees that the instance is distinct.
In CFC, the identifier of a block instance consists of the CPU-wide unique chart name, the separator '.' and the chart-wide unique block name, Conveyormotor, for example (maximum 24 characters).

Block Header

The block header mainly shows the block's properties. This includes:

Block name: This is the name of the block instance. It is given during creation, can however be changed later on. The CFC makes sure that the names are unique throughout the chart.

Block type: This is the type from which the block instance is generated.

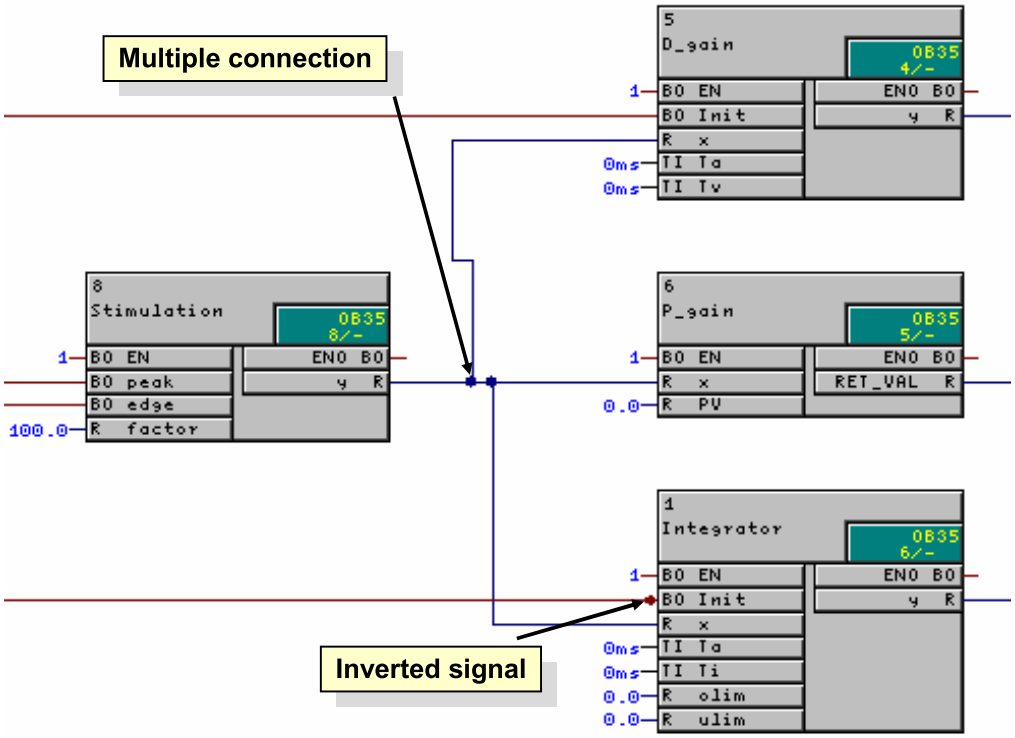
Comment: The comment is used to describe the block in more detail and can be freely defined by the user.

Runtime properties: The runtime properties indicate how the block is inserted in the runtime sequence. This includes, for example, the OB in which the block is called or the call position within the OB.

Block Body

The block body contains the input and output parameters of the block. Every parameter is represented with its data type and its parameter name. As well, for every parameter, the default value is displayed - in so far as it exists.

Interconnecting Inputs / Outputs



Interconnecting

An interconnection is the connecting of an output of a block to one or more inputs of another or the same block. Beyond that, there are interconnections to run-time groups or global addresses.

In these interconnections, the CFC pays particular attention that the data types between inputs and outputs are the same.

There are several ways to make an interconnection:

- single/multiple: clicking the output and clicking the input.
  This way an output can also be connected more than once.
- chain connection: select output. Then with the SHIFT key held down, click on one or more inputs.
- Drag & Drop: click on the output with the left mouse button and holding it down, let it go over the input.

Inverting

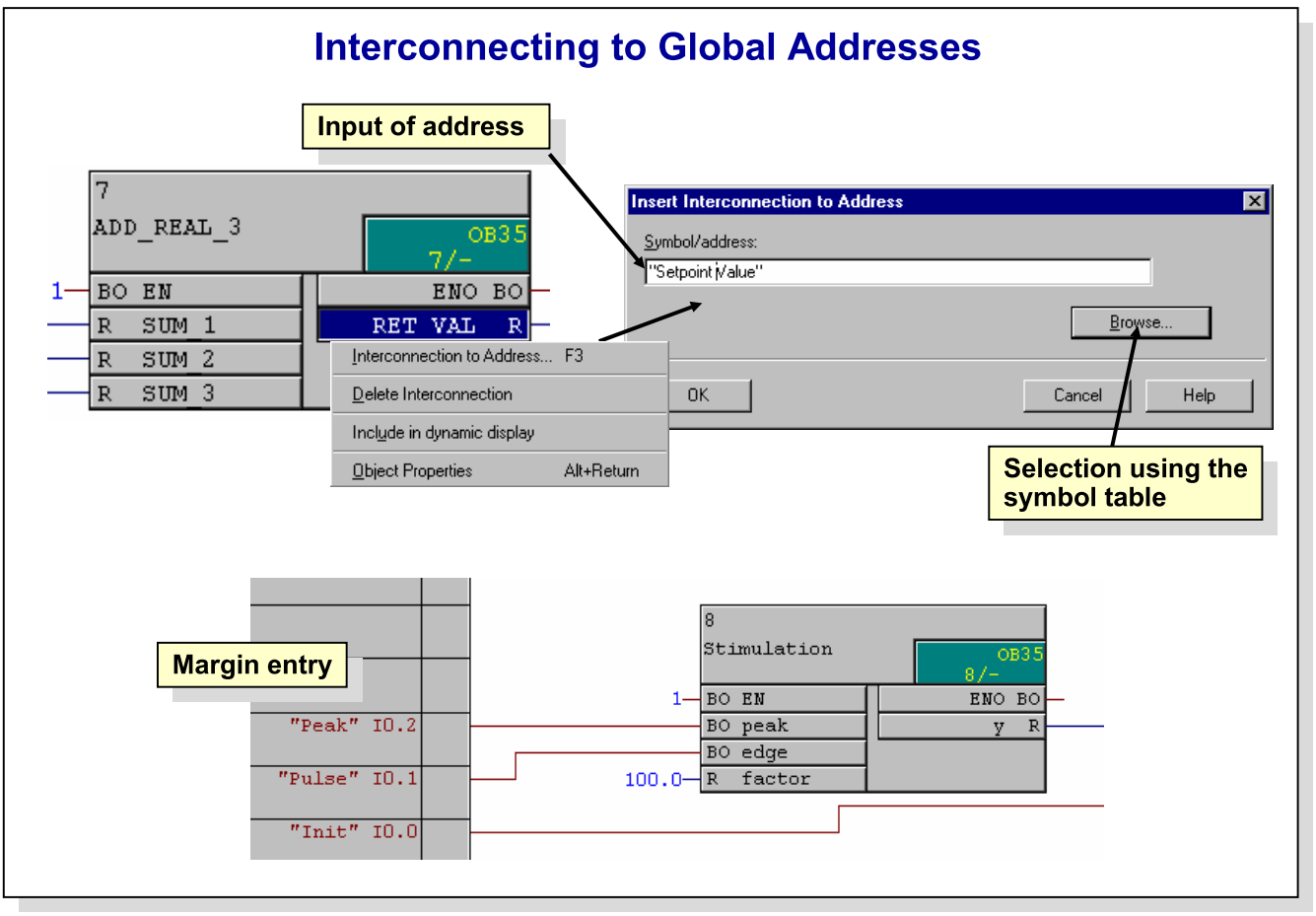
Connected binary inputs can be inverted. This is done using the shortcut menu: Invert.

Autorouting

Connection lines and margins are automatically generated. When a block is shifted, copied or deleted, the autorouter makes sure that the connections are automatically adapted.

Note

When connections are made beyond several sheets, it is advisable to open a second window of the same chart.



Global Addresses

Global addresses are connection partners that lie outside of the CFC charts, for example, global data blocks, I/O signals, bit memories, timers or counters. These addresses can be specified symbolically or absolute.
For symbolic addressing, the absolute address must be defined in the symbol table of the S7 program.

Margin Entries

The interconnection is generated using the shortcut menu of an input / output. The connection line leads to a margin entry in which the source or the target of the connection is described.

A margin entry is always generated when a connection to a partner exists that is not located in the current sheet.

An entry consists of two fields with a height of two lines each. The contents of the entries depends on the type of connection:

- connection to global addresses

  1st. line: symbol name or absolute address
  2nd. line: absolute address and symbol comment

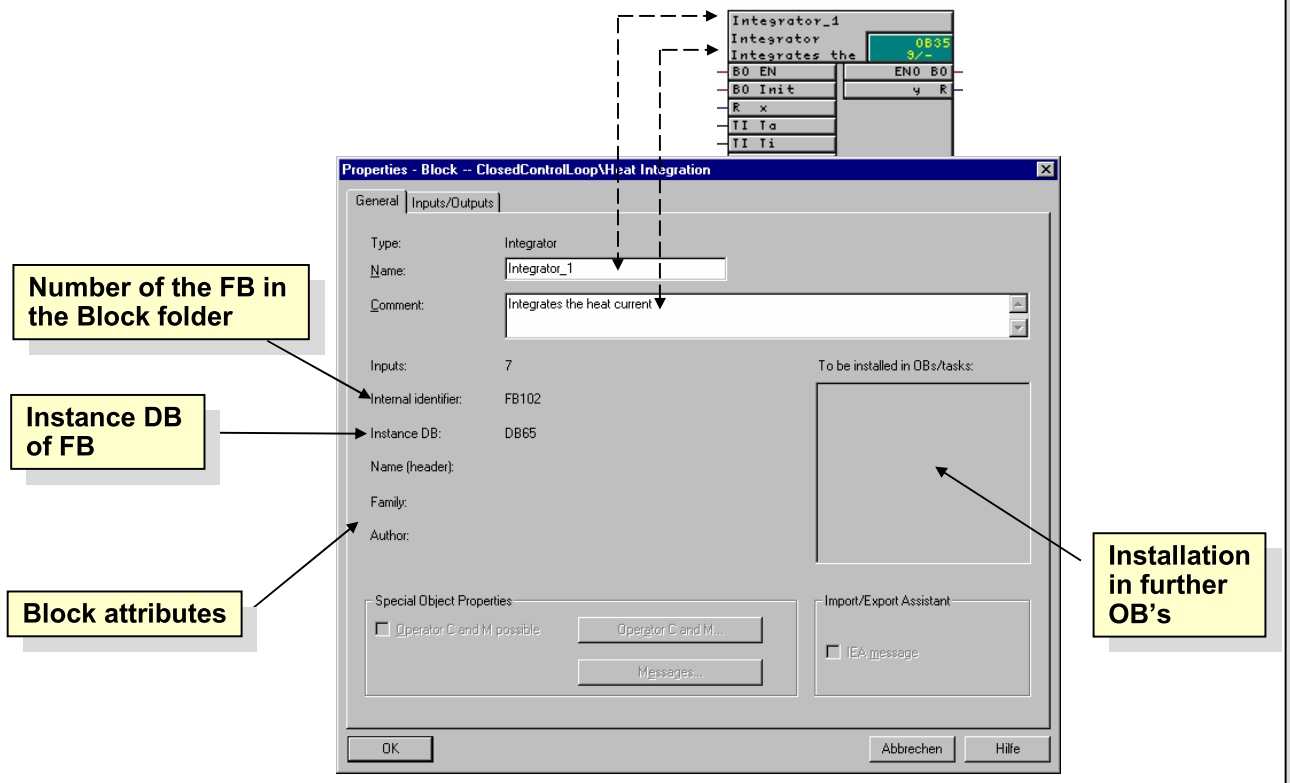
- connection between blocks:

  1st. line: chart name\block name
  2nd. line: parameter name and parameter comment

The display of the margin entries for global addresses depends on which setting was selected under the menu option

View -> Customize -> Addresses.

The Block Properties



Block Properties

After the blocks have been inserted into the chart, the properties can be set for each instance separately. The properties can refer to a block as a whole as well as to the individual inputs and outputs.

The associated dialog box is called by double-clicking with the left mouse button on the block header. This dialog has the three tabs

- General
- Inputs/Outputs
- Runtime Properties

General

Here properties that affect the block as a whole are set.

Name

The name of the block selected by the user must be unique throughout the chart. This is checked by the CFC. The name is displayed in the block header and can be a maximum of 16 characters long.

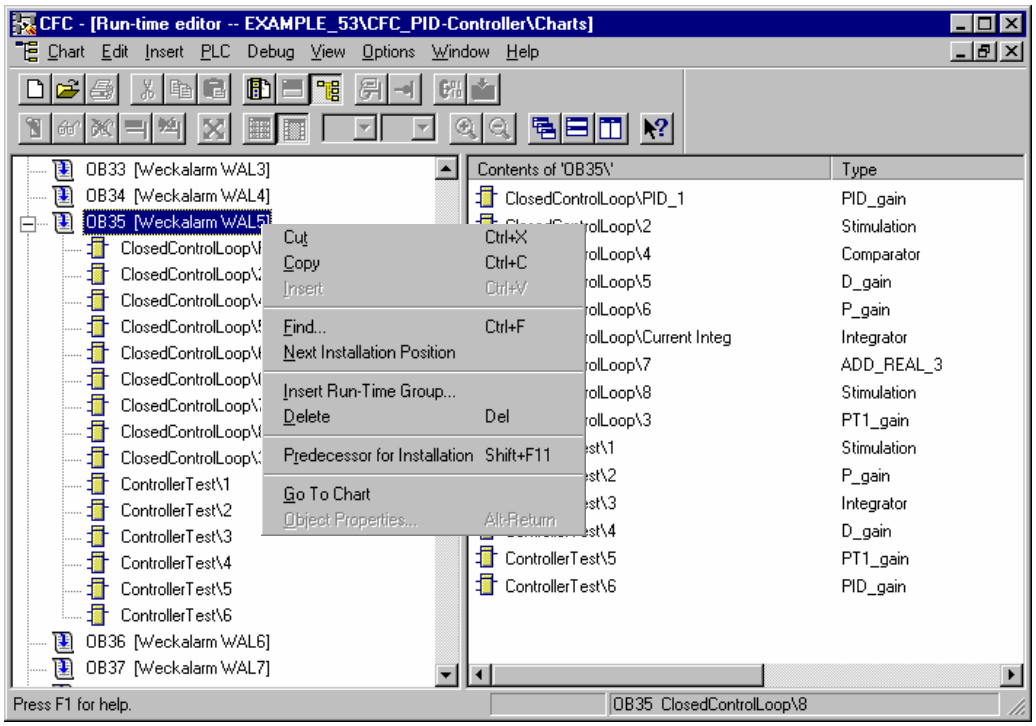
Comment

The comment is used to further describe the block. A maximum of 14 characters is displayed in the block header.

Installing in OB

Here the OBs are displayed in which the block is installed. This installation takes place automatically and is established using the block attribute “S7_tasklist” .

The Runtime Properties



Runtime Properties The runtime properties define in which OBs and in which sequence the individual blocks are called. A block can be installed in several OBs but only once per OB.

Runtime Sequence This is the sequence in which the blocks are called within an OB.

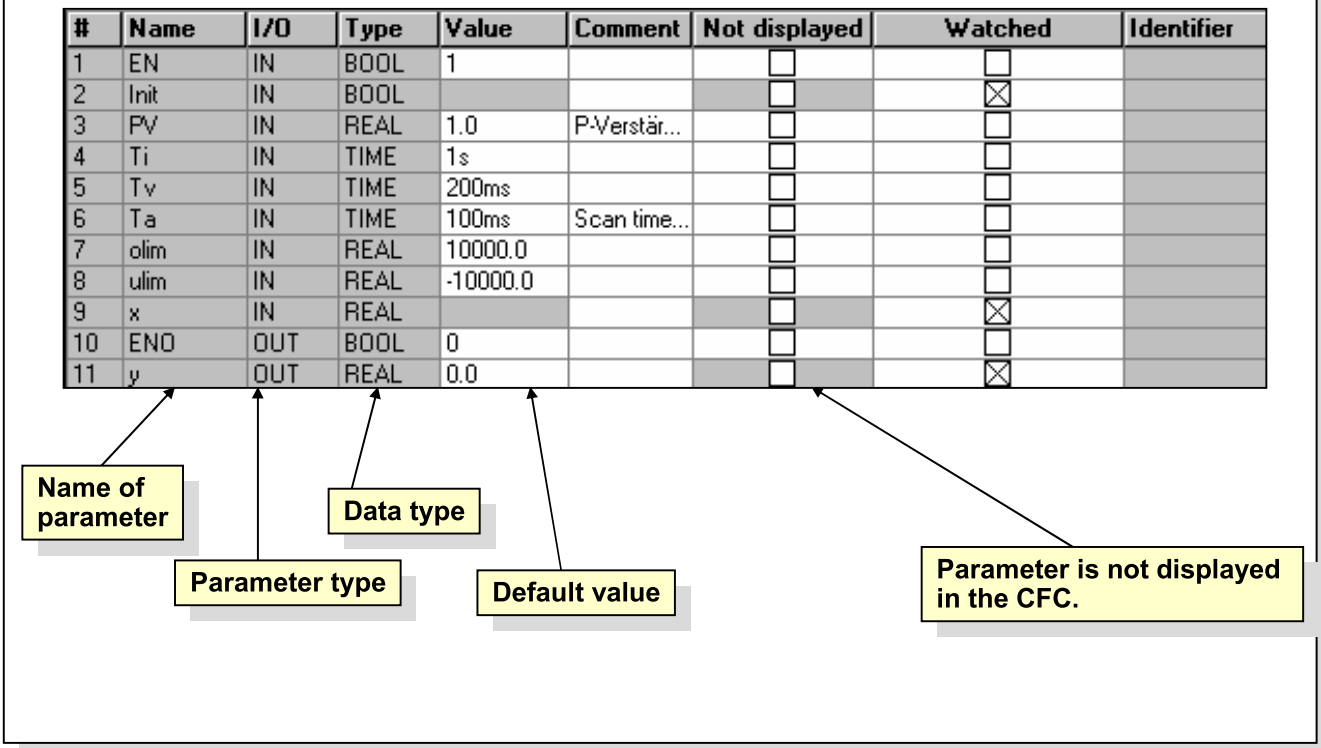
Restructuring When a block is inserted in a chart, the block is automatically inserted after the block that is indicated as “Predecessor for Installation”. This position can be changed later on. This restructuring takes place in two steps.

Removal For this, the block must first be selected and removed. Since every block has be be installed at least once, this removing is rejected if this is the only installed position.

Installing To install, the block or OB is selected in the list according to how the installation is to take place . A click on “Insert” inserts the block. This is only possible, however, if the block hasn‘t already been installed in the OB, since the multiple installation is not possible in an OB.

Note In this dialog, you can only change the runtime properties of the associated block.

The Block Inputs/Outputs



Inputs/Outputs Tab — All inputs/outputs (I/Os) of a block are displayed in this tab. The desired changes can be made in the fields that are not in grey.

Value — The default value of the parameter must correspond to that of the data type.

Comment — The comment can be a maximum of 80 characters. This comment can be displayed in the Block Inputs/Outputs instead of the type and name. This switch is made in the option

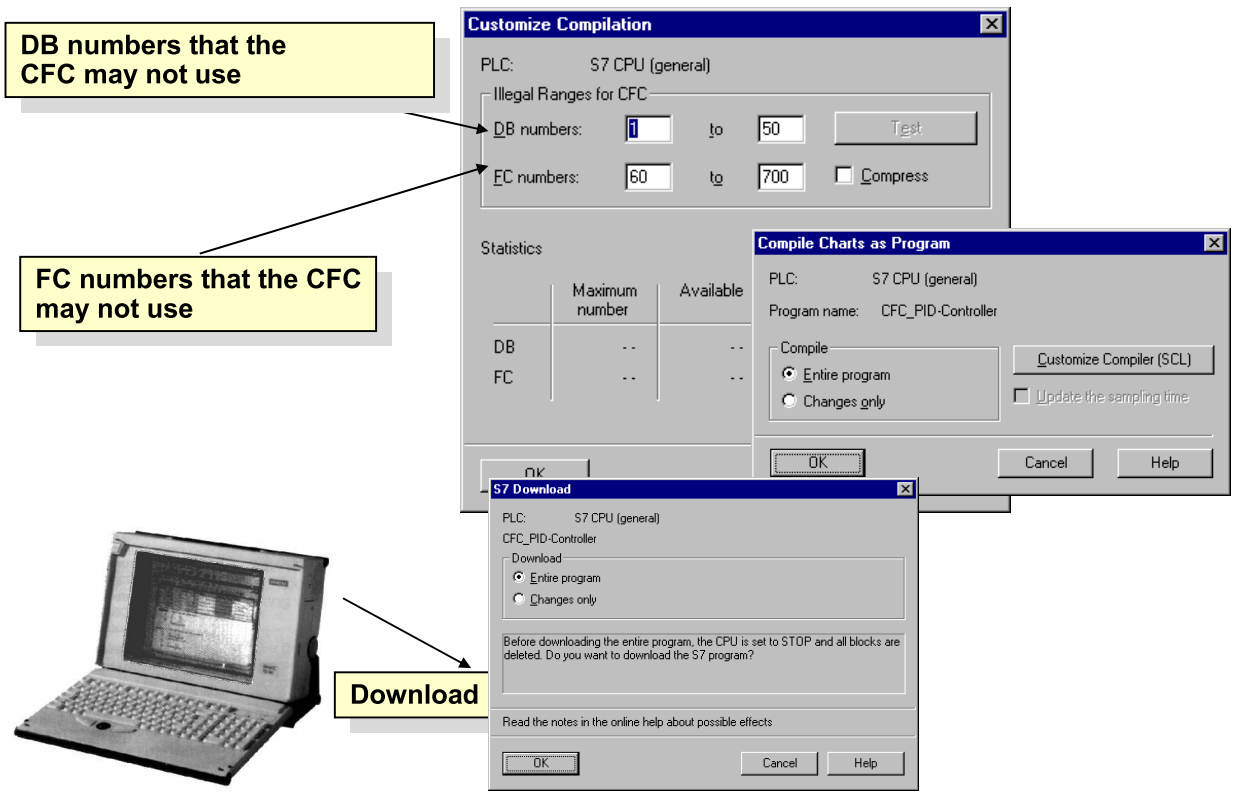
View -> Customize -> Inputs/Outputs.

Hide — By selecting this, the not interconnected I/O is made invisible for the representation in the chart.

For Test — Here, the I/O is logged on for monitoring in test mode.

Note — The I/Os can also be edited separately. You get to the associated dialog by double-clicking on the I/O.

Compiling and Downloading the Program



Disabling Numbers

During compilation, the charts of the current CPU are converted into machine code. Depending on the PLC, a different compiler is used. During the compilation of the charts for a SIMATIC S7 PLC, the CFC generates an instance DB for every function block and several functions (FC) for the calls from the OB.

The numbers are thereby given in ascending order. If an automation task has been implemented not only with charts, but also with other blocks (STL, SCL etc.), their DB/ FC numbers may not collide with the numbers of the CFC.

These numbers have to be disabled for the CFC for that reason. This setting is made using the dialog

Options -> Customize Compilation.

Downloading

The prerequisite for downloading is that the user program to be downloaded has been compiled without errors. A check is made as to whether the program was modified before downloading. If this is the case, a message appears. You can however download.

The program is downloaded using the menu options:

PLC -> Dow

or using the  icon in the toolbar.

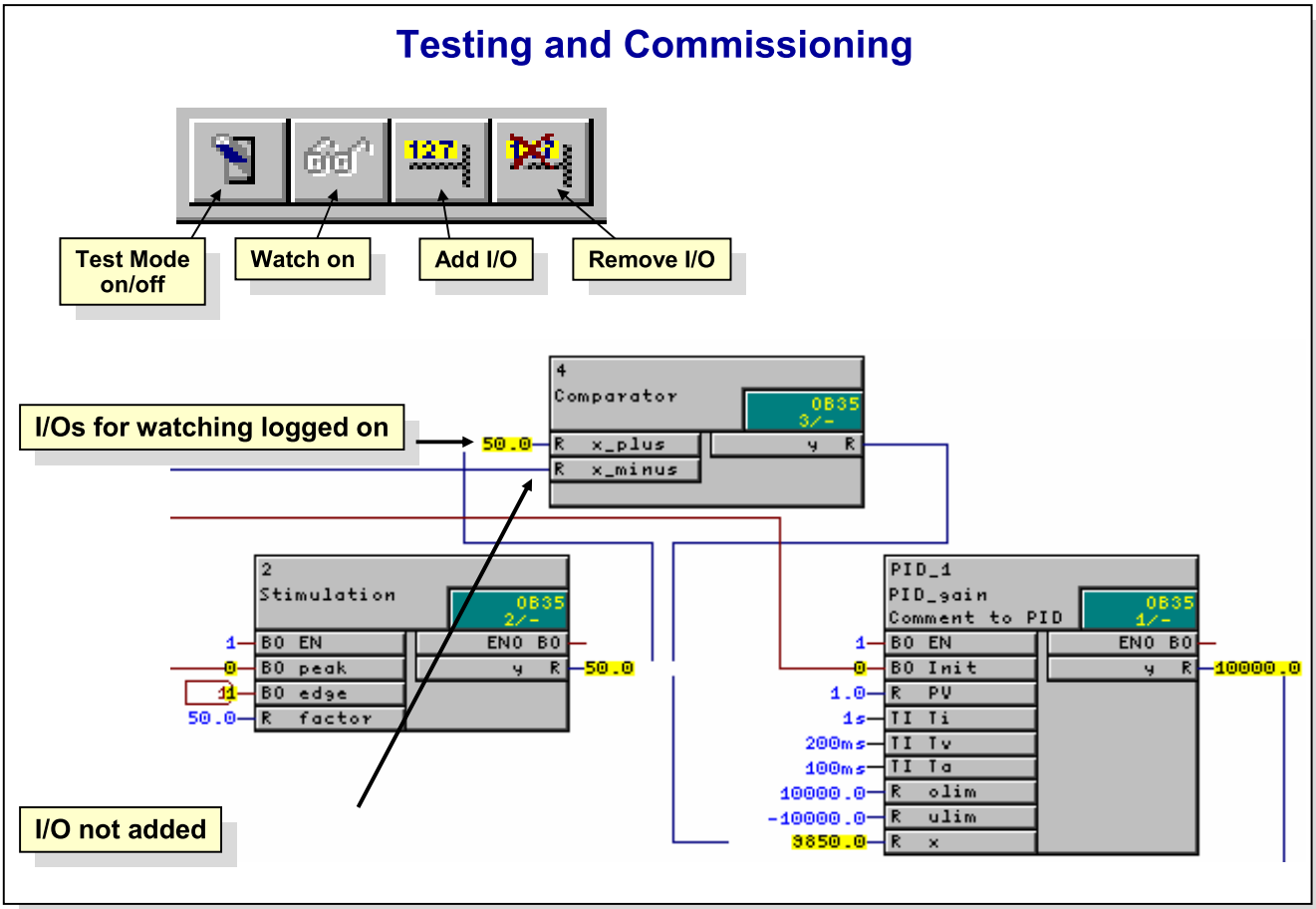
Download…..

Entire Program

This means that the entire contents of the block folder is downloaded into the CPU. All OBs, FBs, FCs and DBs are to be deleted from the CPU before the entire program is downloaded.

Changes Only

In Changes Only, the time stamps are checked and only that which was changed is downloaded. This check makes sure that blocks that are not required are removed from the CPU.



Testing

After the program has been downloaded, it has to be tested. Test functions were integrated in the CFC Editior to support commissioning. These test functions enable you to monitor and assign parameters to block I/Os online in the CPU.

Note

In addition to the testing possibilities of CFC, the SIMATIC Manager has additional functions such as "Monitor/Modify Variable". These functions can be found in the menu option "PLC".

Switching

Switching into the Test mode and back into the Editing mode takes place using the menu options

Test -> Test mode

or the Switch - icon in the toolbar.

Data Consistency

During testing, there must be a unique assignment of data offline to the physical data online. For that reason, it is not possible to make modifications to the chart, such as interconnection or insertion of blocks, while in test mode.

Watch

When you switch on the test mode, the "Watch" function is also automatically activated.

Thus, the values of the block I/Os are dynamically displayed that are logged on for monitoring. When the test mode is started for the first time, it is those I/O that were given the attribute "For Test" I/O properties.

During the test, additional I/O can also be logged on or logged off for monitoring.

Configuring Sequential Control Systems with S7- SFC

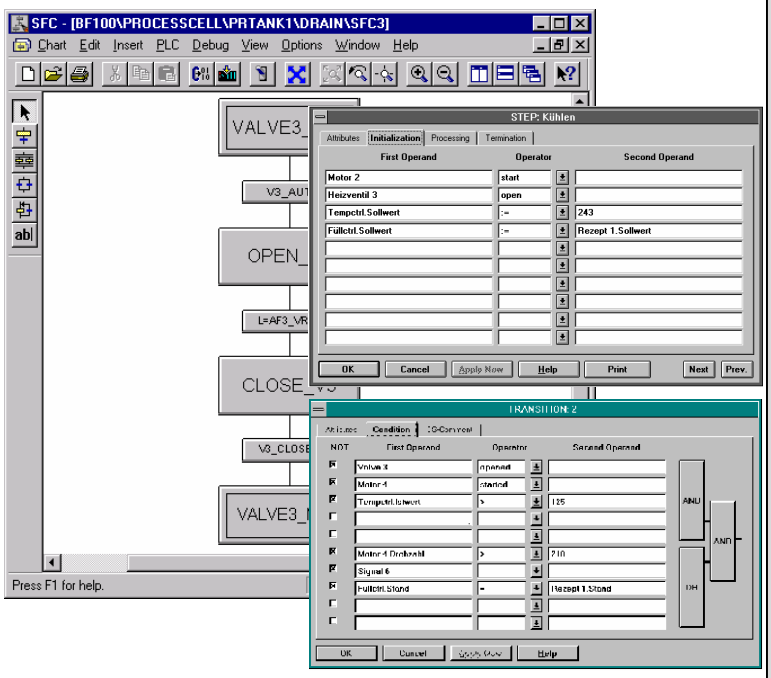
S7-SFC: Tool for programming sequencers

- **Designed for the requirements of process automation**
- **Compatible with IEC 61131-3**
- **Steps assign values to the blocks in the CFC**
- **Transitions check the step-enabling conditions**
- **Syntax checked during creation**

Direct link to CFC

- **Transfer of values by "Drag&Drop"**
- **Cross reference selections**

Visualization within WinCC



Date: 30.06.2011
File: PRO2_12E.64

SFC (Sequential Function Chart)

SFC is a sequential control system that always operates step by step. It was specially designed for the requirements of process automation (process engineering, process control, etc.).

The typical fields of application for sequential control systems of this type are plants with discontinuous operation. Sequential control systems can, however, also be used in plants which operate continuously. For example, for startup or shutdown, working point changes as well as state changes due to disturbances etc.

With SFC, product manufacturing specifications can be written as event-driven processes, for example.

How It Works

In the SFC-Editor you generate the flow chart by graphic means. The structure elements of the chart are positioned according to fixed rules. You do not have to worry about details such as algorithms or the allocation of machine resources, but instead can concentrate on the technological aspects of the configuration.

After generating the chart topology, you switch to the detailed view (zoom) and there assign parameters to the individual elements. That is, you configure the actions (steps) and conditions (transitions).

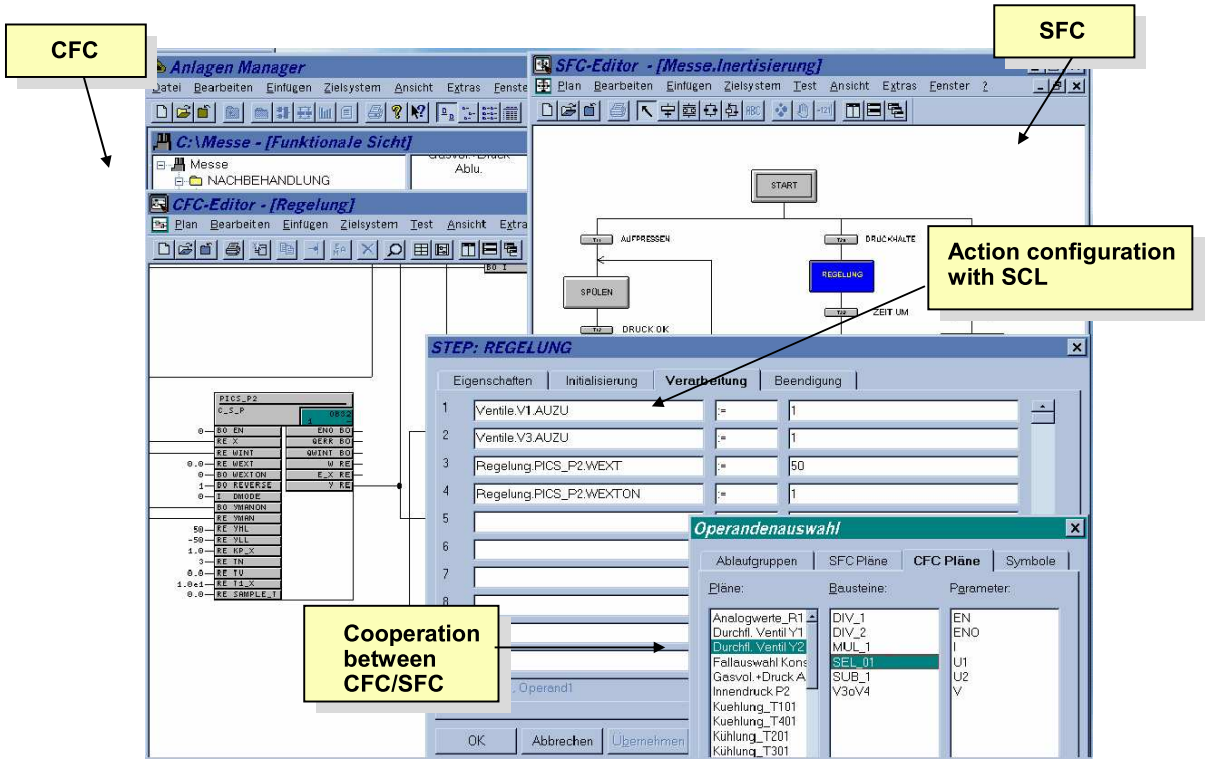
You normally program actions by selectively editing basic automation functions created with CFC using mode and status changes.

After configuration, generate the executable machine code with SFC, download it into the PLC and test it with the SFC debugging functions.

Volume of Project Data

- Sequencers per chart 1
- Steps per chart 2 ... 255
- Transitions per chart 1 ... 255
- Instructions per step <= 50
- Conditions per transition <= 10

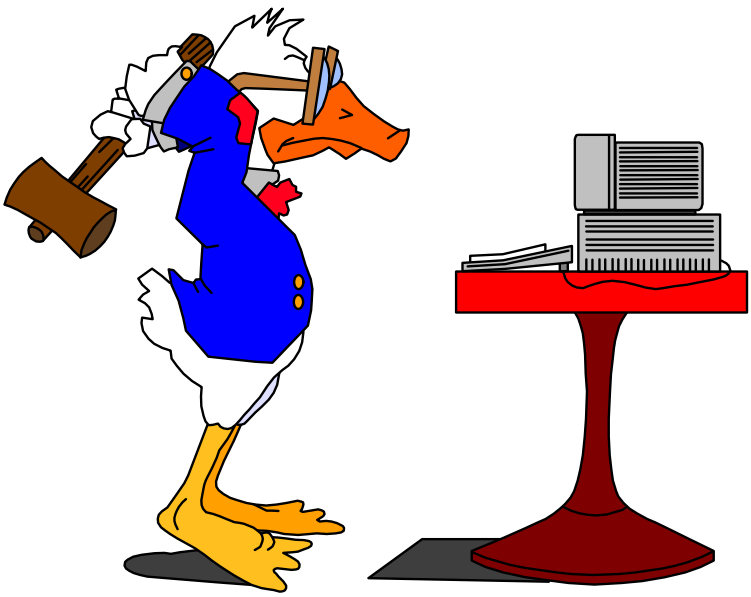
Cooperation betweem CFC/SFC and SCL



Cooperation

- Block programming in the SCL high-level language
- Common data management and code generation with CFC
- Direct cross access from SFC to CFC block instances
- Integration in the STEP 7 SIMATIC Manager

Solutions to the Exercises



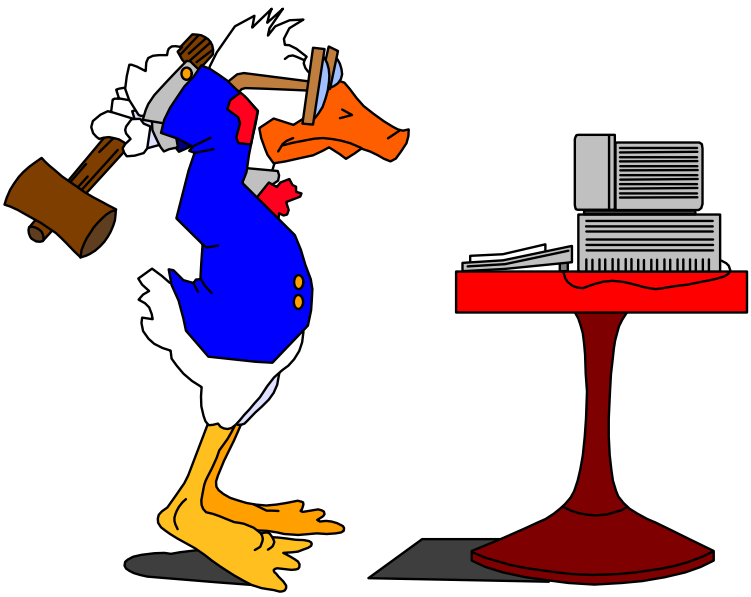
SIMATIC S7

Date: 30.06.2011
File: PRO2_13E.1

SITRAIN Training for Automation and Drives

Contents **Page**

Solutions to the Exercises



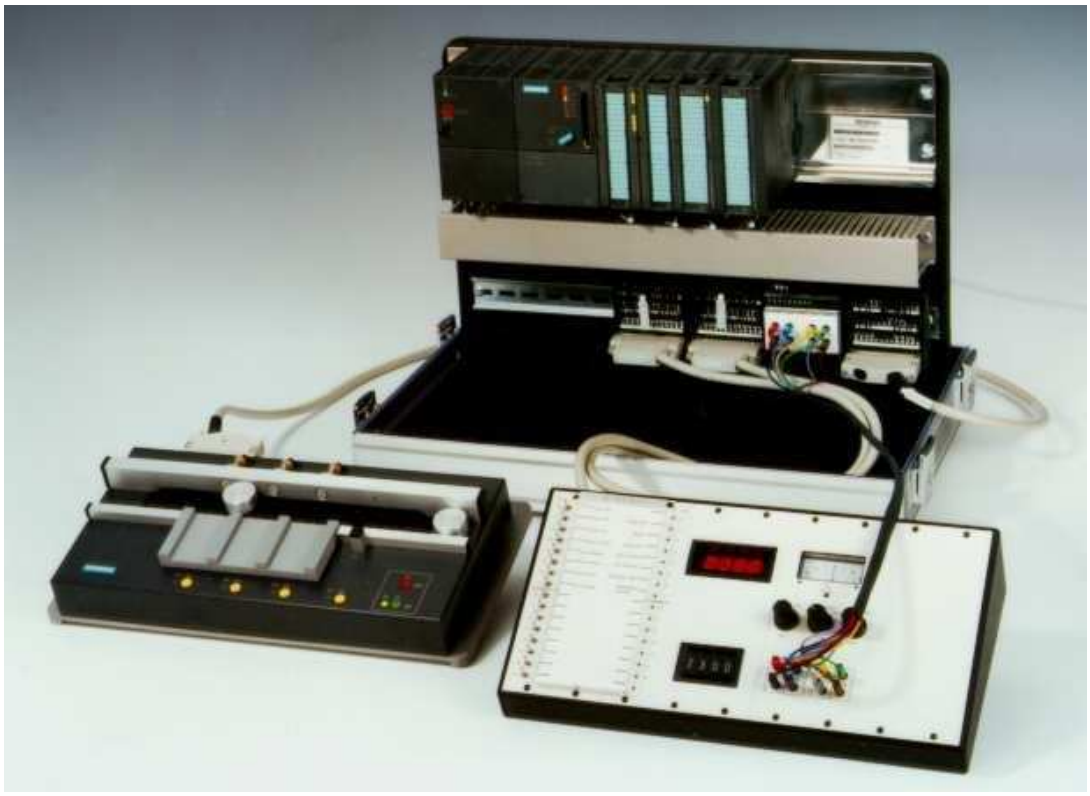
SIMATIC S7

Date: 30.06.2011
File: PRO2_13E.2

SITRAIN Training for Automation and Drives

Contents **Page**

Setup of a Training Area with S7-300



Date: 30.06.2011
File: PRO2_13E.3

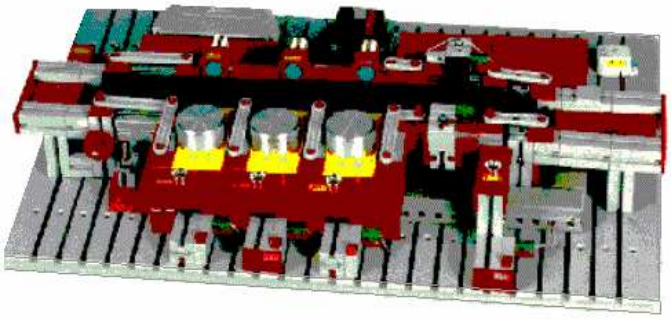
Contents of the Training Kit

The training kit consists of the following components:

- An S7-300 programmable logic controller with CPU 314 or CPU 315-D
- Digital input and output modules, analog modules
- Simulator with digital and analog sections
- Conveyor model

Note

It is quite possible that your training area is not equipped with the conveyor model shown in the slide above, but rather with the conveyor model pictured below.

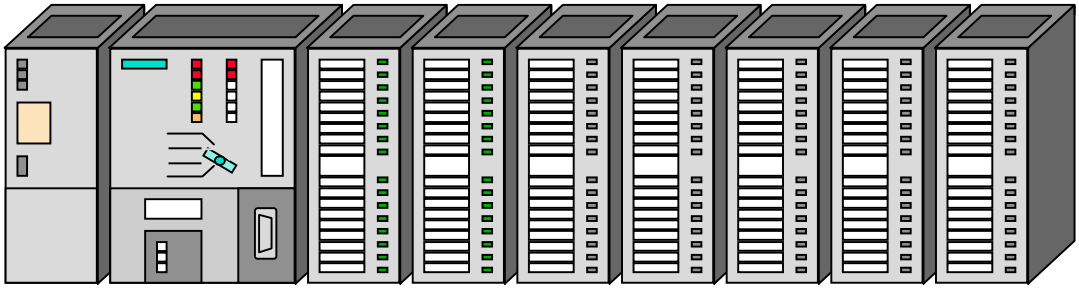


Note

The training kit is also available with an S7-400 central rack

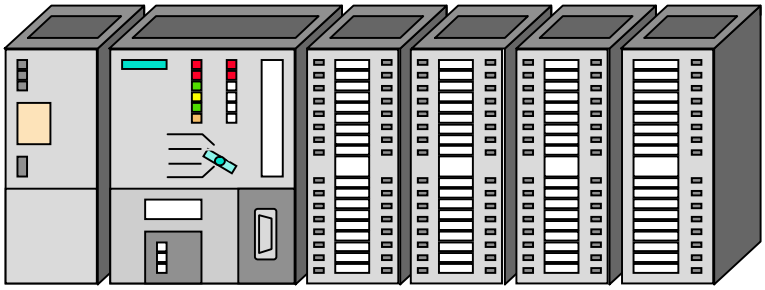
Configuration of the S7-300 Training Unit

Version A
(16 channel
I/O modules)



Module	-->	PS	CPU	DI 16	DI 16	DO 16	DO 16	DI 16	DO 16	AI/AO4
Slot No.	-->	1	2	4	5	6	7	8	9	10
I/O Address	-->			0	4	8	12	16	20	352

Version B
(32 channel
I/O modules)



Module	-->	PS	CPU	DI 32	DO 32	DI8/DO8	AI 2
Slot No.	-->	1	2	4	5	6	7
I/O Address	-->			0	4	8	304

SIMATIC S7

Date: 30.06.2011
File: PRO2_13E.4

SITRAIN Training for Automation and Drives

Configuration of Version A

The programmable controller is configured with the following modules:

Slot 1:	Power Supply 24V/5A	
Slot 2:	CPU 314	
Slot 4:	Digital input 16x24V	Simulator switch
Slot 5:	Digital input 16x24V	Pushwheel buttons
Slot 6:	Digital output 16x24V 0.5A	Output LEDs of the simulator
Slot 7:	Digital output 16x24V 0.5A	Digital display
Slot 8:	Digital input 16x24V	Inputs from the conveyor model
Slot 9:	Digital output 16x24V 0.5A	Outputs to the conveyor model
Slot 10:	Analog module 4 AI/4 AO	Adjustable at the simulator

Configuration of Version B

The programmable controller is configured with the following modules:

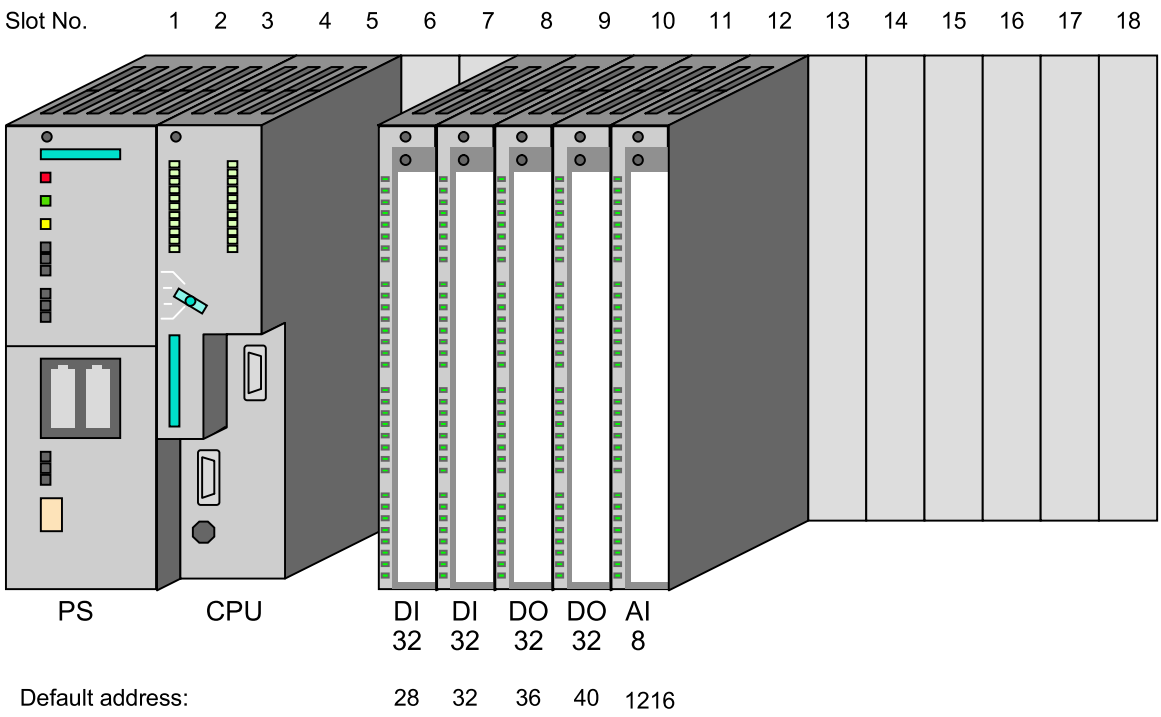
Slot 1:	Power Supply 24V/5A	
Slot 2:	CPU 314	
Slot 4:	Digital input 32x24V	Simulator and pushwheel button switch
Slot 5:	Digital output 32x24V/0.5A	Output LEDs of the simulator and digital display
Slot 6:	Digital module 8X24V/ 8x24V 0.5A	Conveyor model
Slot 7:	Analog input	Adjustable from the simulator

Addresses

Fixed slot addressing is used for the S7-300 (CPU 312-314). The module addresses are shown in the slide.

The starting addresses of the modules can be set by parameter assignment on the CPU 315-2DP and for S7-400.

Configuration of the S7-400 Training Unit



SIMATIC S7

Date: 30.06.2011
File: PRO2_13E.5

SITRAIN Training for Automation and Drives

Configuration

The controller is configured with the following modules:

Slot 1, 2 and 3:	Power Supply 220V/20A	
Slot 4:	CPU 412, 414 or CPU 416	
Slot 5, 6 and 7:	vacant (only used with M7)	
Slot 8: simulator	Digital input 32x24V	Inputs from the
Slot 9: conveyor	Digital input 32x24V	Inputs from the model
Slot 10:	Digital output 32x24V 0.5A	Outputs to the simulator
Slot 11:	Digital output 16x24V 0.5A	Outputs to the conveyor model
Slot 12:	Analog input module 8AI can be addressed from the simulator	

Addresses

For the S7-400, the above basic addresses are only valid when you have default addressing, that is, when no parameter assignment data (SDBs) were loaded into the CPU for the modules.

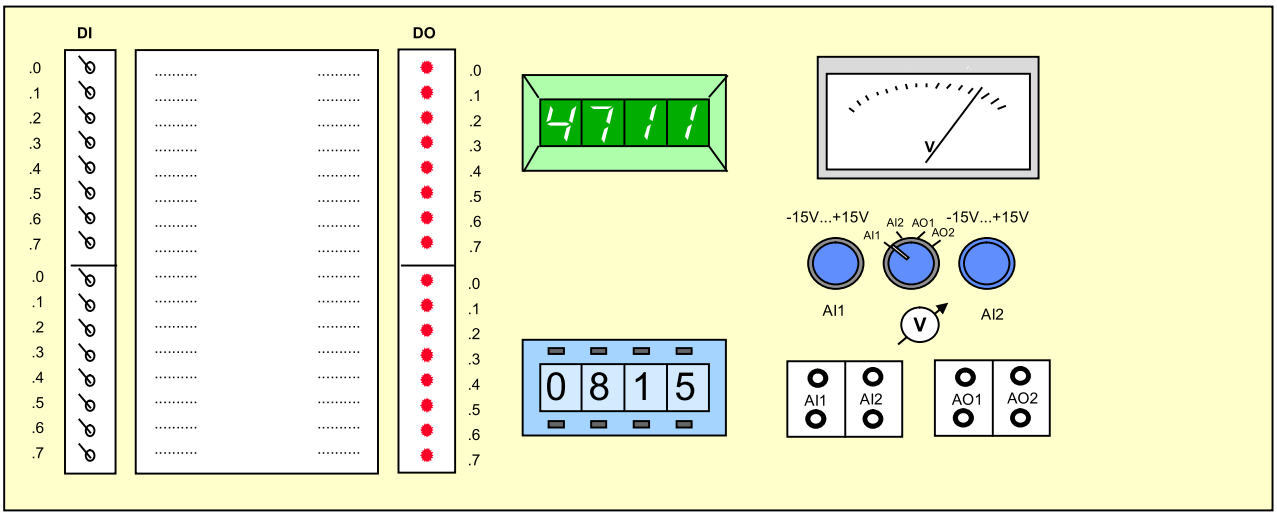
The default basic addresses of the modules in the central rack of the S7-400 are calculated according to the following formula:

Digital modules: (Slot number - 1) x 4

Analog modules: (Slot number - 1) x 64 + 512

For the CPUs of the S7-400 system, you can assign any parameters to the initial addresses of the modules.

The Simulator



SIMATIC S7

Date: 30.06.2011
File: PRO2_13E.6

SITRAIN Training for Automation and Drives

Design

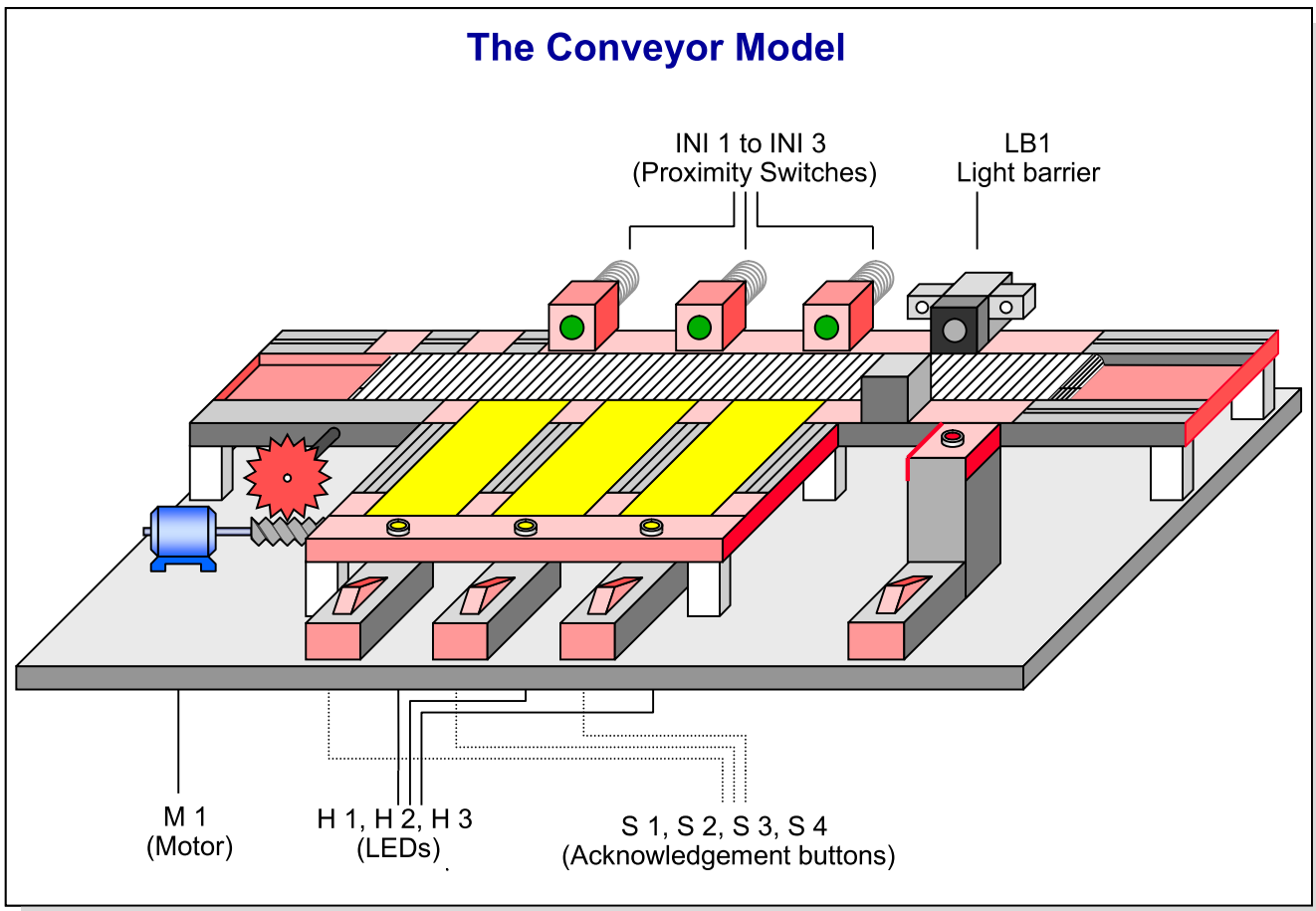
The Simulator is connected to the S7-300 or S7-400 training unit by two cables. It is in three sections:

- Binary section with 16 switches/momentary-contact switches and 16 LEDs
- Digital section with a 4 position pushwheel button and a digital display. These operate with BCD values,
- Analog section with a voltmeter for displaying the values at analog channels 0 and 1 or the analog outputs 0 and 1. You use the selector switch to choose the voltage value you want to monitor. There are two separate potentiometers for setting the values for the analog inputs.

Addressing

You use the following addresses to address the inputs and outputs of the S7-300/S7-400 in your user program:

Sensor / Actuator	S7-300 A	S7-300 B	S7-400
Switch / M.C. Sw.	IW 0	IW 0	IW 28
LEDs	QW 8	QW 4	QW 36
Pushwheel buttons	IW 4	IW 2	IW 30
Digital display	QW 12	QW 6	QW 38
Analog channels	PIW 352/354	PIW 304/306	PIW 1216/1218



SIMATIC S7

Date: 30.06.2011
File: PRO2_13E.7

SITRAIN Training for Automation and Drives

Design

You can see the design of the conveyor model with its existing sensors and actuators in the slide above.

The addresses, for S7-400 refer to the addressing of the S7-400 model with default addresses. You can make changes to these settings when you change the addresses of the input or output modules in HW Config.

When you make changes to the module addresses you have to make sure that the input signals from the conveyor model are input to the second input module (slot 9) and the output signals from the conveyor model are output from the second output module (slot 11).

The signals of the simulator are connected to the first input module and the first output module.

Addresses

S7-300 A	S7-300 B	S7-400	Sensor/ Actuator	Symbol
I 16.0	I 8.0	I 32.0	Light barrier LB 1	LB1
I 16.1	I 8.1	I 32.1	Ackn. switch, Bay 1	S1
I 16.2	I 8.2	I 32.2	Ackn. switch, Bay 2	S2
I 16.3	I 8.3	I 32.3	Ackn. switch, Bay 3	S3
I 16.4	I 8.4	I 32.4	Ackn. switch, Final Assy	S4
I 16.5	I 8.5	I 32.5	Proximity switch 1	INI1
I 16.6	I 8.6	I 32.6	Proximity switch 2	INI2
I 16.7	I 8.7	I 32.7	Proximity switch 3	INI3
Q 20.1	Q 8.1	Q 40.1	LED at Bay 1	H1
Q 20.2	Q 8.2	Q 40.2	LED at Bay 2	H2
Q 20.3	Q 8.3	Q 40.3	LED at Bay 3	H3
Q 20.4	Q 8.4	Q 40.4	LED at Final Assembly	H4
Q 20.5	Q 8.5 K1_CONVR	Q 40.5	Conveyor oper. to right	
Q 20.6	Q 8.6 K2_CONVL	Q 40.6	Conveyor oper. to left	
Q 20.7	Q 8.7	Q 40.7	Horn	HORN1

Solution to Exercise 1.1: Jump After a Subtraction

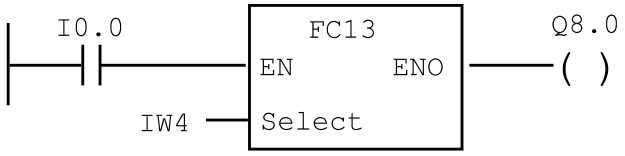
```
                FUNCTION FC 11 : VOID
                TITLE =Exercise 1.1 : Jump  After a Subtraction
                //Version for 16Bit SM
                AUTHOR : PT41
                FAMILY : A4_0
                NAME : ST7PRO2
                VERSION : 0.0

                BEGIN
                NETWORK
                TITLE =

                L     IW    4;              // Thumbwheel switch
                BTD ;                       // Convert format from BCD to DINT
                L     IW    0;              // Input word 0
                 BTD;
                 -D;
                JM    NEG;                  // Jump, if result negative
                L     IW    4;
                JU    END;
NEG:            L     0;
END:            T     QW  12;             // Digit display
                END_FUNCTION
```

Solution to Exercise 1.2: Jump After a Multiplication

```
        FUNCTION FC 12 : VOID
        TITLE =Exercise 1.2 : Jump After a Multiplication
        //Version for 16Bit SM
        AUTHOR : PT41
        FAMILY : A4_0
        NAME : ST7PRO2
        VERSION : 0.0

        BEGIN
        NETWORK
        TITLE =

        L     IW    4;              // Thumbwheel switch
        BTD;                        // Conversion from BCD to DINT
        L     IW    0;              // Toggle switch on the Simulator
        BTD;
        *I;
        JO    OVL;                  // Jump if overflow
        DTB;                        // Conversion from DINT to BCD
        JU    END;
OVL:    L     0;
END:    T     QW  12;               // Digital display
        END_FUNCTION
```

Solution to Exercise 1.3: Programming a Jump Distributor

OB1




```
FUNCTION FC 13: VOID
//Version for 16Bit SM
// Programming a Jump Distributor
VAR_INPUT
Select: INT;
END_VAR
BEGIN
SET;
SAVE;
L     #Select;
AW              W#16#FF00;   // check if selection >255 or
JN              Err;         // Jump if > 255
L               #Select;     // Load value again
JL   GT5;                    // Jump target if ACCU1-L-L >5
JU   Err;                    // if selection = 0 (not possible)
JU   Dr_1;                   // Belt to the right (selection=1)
JU   Dr_2;                   // Belt to the left (selection=2)
JU   Dr_3;                   // Stop belt (selection=3)
JU   Ho_1;                   // Horn on
JU   Ho_2;                   // Horn off
GT5:  JU   Err;
Dr_1: S    Q     20.5;       // Belt to the right
      R    Q     20.6;
      JU   End;
Dr_2: S    Q     20.6;       // Belt to the left
      R    Q     20.5;
      JU   End;
Dr_3: R    Q     20.5;       // Stop belt
      R    Q     20.6;
      JU   End;
Ho_1: S    Q     20.7;       // Horn on
      JU   End;
Ho_2: R    Q     20.7;       // Horn off
      JU   End;
Err:  R          Q     20.5; // Stop belt
      R          Q     20.6;
      R          Q     20.7; // Horn off
      CLR;                   // Reset ENO
      SAVE;
End:  BE;
      END_FUNCTION
```

Solution to Exercise 2.1: Calculation of Exponents

```
FUNCTION FC 21 : VOID
TITLE =Exercise 2.1: Calculation of Exponents
//Version for 16Bit SM
AUTHOR : PT41
FAMILY : A4_0
NAME : ST7PRO2
VERSION : 0.0

BEGIN
NETWORK
TITLE =
L     IB    5;          // Load right-hand byte of the pushweel switch
BTI;                    // BCD to INT  -> input value
 PUSH;                  // Copy ACCU1 into ACCU2
*D;                     // Form square of inp. val. in ACCU1
PUSH;                   // Copy square of inp. val. from ACCU1 to ACCU2
PUSH;                   // Necessary for S7-400: square -> ACCU3
*D;                     // Form inp. val. power of 4 in ACCU1
*D;                     // Form inp. val. power of 6 in ACCU1
 DTB;                   // Convert to BCD
T     QW  12;           // Transfer LOW Word to the digit display
END_FUNCTION
```

Solution to Exercise 2.2: Data Exchange in ACCU1

```
FUNCTION FC 22 : VOID
TITLE =Exercise 2.2: Data Exchange in ACCU1
//Version for 16Bit SM
AUTHOR : PT41
FAMILY : A4_0
NAME : ST7PRO2
VERSION : 0.0

BEGIN
NETWORK
TITLE =

L     IW    4;              // Load BCD number
CAW;                        // Swap the two bytes  in AKKU1-L
T     QW  12;               // Display  result
END_FUNCTION
```

Solution to Exercise 2.3 : Forming Ones Complements

```
FUNCTION FC 23 : VOID
TITLE = 2.3: Forming Complements
//Version for 16Bit SM
AUTHOR : PT41
FAMILY : A4_0
NAME : ST7PRO2
VERSION : 0.0

BEGIN
NETWORK
TITLE =Ones Complement in STL

L     IW    0;              // Load input word from the toggle switches
INVI;                       // Form ones complement
T     QW  8;                // Transfer result to simulators LEDs
END_FUNCTION
```

Solution to Exercise 3.1: Calculating the Distance

```
FUNCTION FC 31 : REAL
TITLE =Exercise 3.1: Calculating the Distance
AUTHOR : PT41
FAMILY : A4_0
NAME : ST7PRO2
VERSION : 0.0

VAR_INPUT
  X1:  REAL;
  Y1:  REAL;
  X2:  REAL;
  Y2:  REAL ;
END_VAR
VAR_TEMP
  XSquare : REAL;
END_VAR
BEGIN
  NETWORK
  TITLE =
     L     #X1;                  // Load X-Coordinate of P1
     L     #X2;                  // Load X-Coordinate of P2
     -R;                         // Calculate (X1-X2)
     SQR;                        // Form square of (X1-X2)
     T     #XSquare;             // store result in TEMP Var.
     L     #Y1;                  // Load Y-Coordinate of P1
     L     #Y2;                  // Load Y-Coordinate of P2
     -R;                         // Calculate (Y1-Y2)
     SQR;                        // Form square of (Y1-Y2)
     L     #XSquare;             // Retrieve square from (X1-X2)
     +R;                         // Form Sum
     SQRT;                       // Compute Square Root
     T     #RET_VAL;             // Transfer to RET_VAL
END_FUNCTION
```

Solution to Exercise 4.1: Loop Programming with Indirect Addressing (part 1)

```
FUNCTION FC 41 : VOID
TITLE =Exercise 4.1:Loop Programming with Memory Indirect Addressing

VAR_INPUT
  DB_Num : WORD ;
END_VAR
VAR_TEMP
  L_Counter : INT ;
  Ini_Value : REAL ;
  I_DB_Num : WORD;
  Par_Pointer : DWORD ; END_VAR
BEGIN
NETWORK
TITLE =Open the DB

      L     #DB_Num;                // load DB-number
      T     #I_DB_Num;              // and transfer to temp. Variable;
      OPN   DB [#I_DB_Num];         // open DB


NETWORK
TITLE =LOOP

      L     P#0.0;                  // Load address of the 1st Component of Tank
      T     #Par_Pointer;           // and transfer to #T_Pointer
      L     1.0;                    // Load constant 1.0 and
      T     #Ini_Value;             // transfer to #Ini_Value
      L     100;                    // Initialize loop counter with 100
BEGN: T     #L_Counter;             // and transfer to #L_Counter
      L     #Ini_Value;
      T     DBD [#Par_Pointer];     // Transfer #Ini_Value Meas_Value[i]
      L     1.0;                    // Incrementing ACCU1 (#Ini_Value)
      +R    ;                       // by 1.0
      T     #Ini_Value;             // and transfer to #Ini_Value
      L     #Par_Pointer;           // Load #Par_Pointer into ACCU1
      L     P#4.0;                  // Incrementing the byte address
      +D    ;                       // of #Par_Pointer by 4 units
      T     #Par_Pointer;           // and transfer result to #Par_Pointer
      L     #L_Counter;             // Load loop counter,
      LOOP  BEGN;                   // Decrement loop counter and if necessary jump

END_FUNCTION
```

Solution to Exercise 4.1: Loop Programming with indirect Addressing (part 2)

```
ORGANIZATION_BLOCK OB 1
TITLE =
VERSION : 0.1
VAR_TEMP
  OB1_EV_CLASS : BYTE ;       //Bits 0-3 = 1 (Coming event), Bits 4-7 = 1 (Event class 1)
  OB1_SCAN_1 : BYTE ;         //1 (Cold restart scan 1 of OB 1), 3 (Scan 2-n of OB 1)
  OB1_PRIORITY : BYTE ;       //1 (Priority of 1 is lowest)
  OB1_OB_NUMBR : BYTE ;       //1 (Organization block 1, OB1)
  OB1_RESERVED_1 : BYTE ;  //Reserved for system
  OB1_RESERVED_2 : BYTE ;  //Reserved for system
  OB1_PREV_CYCLE : INT ;      //Cycle time of previous OB1 scan (milliseconds)
  OB1_MIN_CYCLE : INT ;       //Minimum cycle time of OB1 (milliseconds)
  OB1_MAX_CYCLE : INT ;       //Maximum cycle time of OB1 (milliseconds)
  OB1_DATE_TIME : DATE_AND_TIME ;  //Date and time OB1 started
END_VAR
BEGIN
NETWORK
TITLE =
     CALL FC    41 (
          DB_Num := W#16#29);
NOP    0;

END_ORGANIZATION_BLOCK
```

Solution to Exercise 4.2: Loop Programming with Register Indirect Addressing

```
FUNCTION FC 42: VOID
TITLE =Exercise 4.2: Loop Programming with Register Indirect Addressing
// Version for S7-300 and S7-400

VAR_INPUT
  DB_Num : WORD ;
END_VAR
VAR_TEMP
  I_DB_Num : WORD ;
END_VAR
BEGIN
NETWORK
TITLE =Open the DB

     L     #DB_Num;              // load DB number
     T     #I_DB_Num;            // and transfer to temp. Variable;
     OPN   DB [#I_DB_Num];       // open DB


NETWORK
TITLE =LOOP

     LAR1  P#DBX0.0;             // Load address of the 1st Component of Tank
     L     L#1;                  // 1 into ACCU1(Ini_Value.)
     L     100;                  // 100 into AKKU1 (L_Counter); 1 into ACCU2 (Ini_Value)
BEGN: TAK   ;                    // L_Counter in ACCU2, Ini_Value in ACCU1
     T     D [AR1,P#0.0];        // Transfer Ini_Value to Tank[i]
     +     L#1;                  // Incrementing Ini_Value
     +AR1  P#4.0;                // Incrementing AR1 by 2 units
     TAK   ;                     // L_Counter in ACCU1, Ini_Value in ACCU2
     LOOP  BEGN;                 // Decrementing and jumping

END_FUNCTION
```

Solution to Exercise 4.3: Function for Calculating Sum and Mean Value

```
FUNCTION FC 43 : VOID
TITLE = Exercise 4.3: Calculating Sum and Mean Value
VERSION : 0.0
VAR_INPUT
  Measured_values : ANY ;
END_VAR
VAR_OUTPUT
  Sum : REAL ;
  Mean_value : REAL ;
END_VAR
VAR_TEMP
  Num_Elements : WORD ;
  L_Counter : WORD ;
  DB_No : WORD ;
END_VAR
BEGIN
NETWORK
TITLE =
      L     P##Measured_values;   // Load address of "ANY"-Pointer
      LAR1  ;                     // Transfer address into AR1
      L     B [AR1,P#1.0];        // Load identifier of data type
      L     8;                    // Load identifier of REAL (16#08)
      ==I   ;
      JC    REAL;                 // Jump if data type is equal to REAL
      NOP   0;                    // Instructions for data type unequal to REAL
      CLR   ;                     // RLO=0
      SAVE  ;                     // BR=0
      L     L#-1;                 // Load invalid REAL number
      T     #Sum;
      T     #Mean_value;
      BEU   ;
REAL: NOP   0;                    // Instructions for data type: REAL
      L     W [AR1,P#2.0];        // Load number of array elements
      T     #Num_Elements;        // Store number of elements
      L     W [AR1,P#4.0];        // Load DB number or 0
      T     #DB_No;               // If: DB_No=0, Then: OP DB[DB_No]=NOP
      OPN   DB [#DB_No];          // Runtime Error!!, if DB does not exist
      L     D [AR1,P#6.0];        // Load Pointer to actual operand
      LAR1  ;                     // into AR1, Error !! if area identifier is equal to "DI"
      L     0.000000e+000;        // 0 into ACCU1 (Sum =0.0)
      L     #Num_Elements;        // Num_Elements into ACCU1; Sum=0 into ACCU2
BEGN: T     #L_Counter;           // Set L_Counter
      TAK   ;                     // Sum in ACCU1
      L     D [AR1,P#0.0];        // Array element in ACCU1, Sum in ACCU2
      +R    ;                     // Sum in ACCU1
      +AR1  P#4.0;                // Incrementing AR1 by 4 units
      L     #L_Counter;           // L_Counter in ACCU1, Sum in ACCU2
      LOOP  BEGN;                 // Decrementing and jumping
      TAK   ;                     // Sum in ACCU1
      T     #Sum;                 // Sum to #Sum
      L     #Num_Elements;        // Sum into ACCU2, number of elements in ACCU1
      DTR   ;                     // Convert unsigned Integer (16 bit) to REAL
      /R    ;                     // Mean value in ACCU1
      T     #Mean_value;          // Transfer mean value to #Mean_value
      SET   ;                     // Set BR bit
      SAVE  ;
END_FUNCTION
```

Solution to Exercise 5.2: Accessing Complex Data Types

```
FUNCTION FC 52 : VOID
TITLE =Monitoring Motors
//Version for S7-300 and S7-400
VERSION : 0.1

VAR_INPUT
  Motor : "Motor";
END_VAR
VAR_OUTPUT
  Motor_OK : BOOL ;
  SetActDiff : DINT ;
  SetActDiffDisp : DWORD ;
END_VAR
VAR_TEMP
  SetActDifference : REAL ;
END_VAR
BEGIN
NETWORK
TITLE =

//Computing the percent deviation
      SET   ;                       //enforce first check, set RLO to"1"
      SAVE  ;                       //Set BR bit to "1"
      L     #Motor.SetSpeed;        //Set speed in ACCU1
      PUSH  ;                       //only for S7-400, set speed in ACCU2
      PUSH  ;                       //Set speed in ACCU3
      L     #Motor.ActualSpeed;     //Set speed in ACCU2, actual speed in ACCU1
      -R    ;                       //Difference in ACCU1, Set speed in ACCU2
      T     #SetActDifference;      //Store difference in temp. variable
      TAK   ;                       //Difference in ACCU2, set speed in ACCU1
      /R    ;                       //Actual percent deviation in ACCU1
      ABS   ;                       //absolute percent deviation in ACCU1
      L     #Motor.SetActDiffMax;   //load max. percent deviation in ACCU1
      <=R   ;                       //Actual deviation equal or smaller setpoint deviation?
      AN    #Motor.Disturbance;     //and no disturbance
      =     #Motor_OK;              //than Motor is OK
NETWORK
TITLE =Displaying the difference between et speed and actual speed

      L     #SetActDifference;      //load intermediate SetActDifference
      RND   ;                       //convert this to DINT
      PUSH  ;                       //Store SetActDifference in ACCU2
      DTB   ;                       //DINT number in ACCU2, BCD number in ACCU1
      JO   ERR;                     //jump if conversion error
      T     #SetActDiffDisp;        //transfer valid BCD number to digital display
      TAK   ;
      T     #SetActDiff;            //transfer valid DINT number to #SetActDiff
      BEU   ;                       //If no error terminate
ERR: CLR   ;
      SAVE  ;                       //clear BR bit

END_FUNCTION
```

Solution to Exercise 5.3: Read System Clock with SFC 1 (READ_CLK)

```
FUNCTION FC 53 : VOID
TITLE = Exercise 5.3: Read System Clock
//Version for 16Bit-SM
AUTHOR : PT41
FAMILY : A2_0
NAME : ST7PRO2
VERSION : 0.0


VAR_TEMP
  Date_Time : DATE_AND_TIME ;          //Current Time and Date
  RET_VAL_SFC1 : INT ;                 //Return value of SFC 1
END_VAR
BEGIN
NETWORK
TITLE =Call SFC 1 (READ_CLK)

     CALL SFC1 (
        RET_VAL          := #RET_VAL_SFC1,
        CDT              := #Date_Time);

     NOP   0;
NETWORK
TITLE = Display hours and minutes

     LAR1  P##Date_Time;             // Get address of #Date_Time
     L     LB [AR1, P#3.0];          // Read hours
     T     QB       12;              // and transfer to dig. display
     L     LB [AR1, P#4.0];          // read minutes
     T     QB     13;                // and transfer to display

END_FUNCTION
```

Solution to Exercise 6.1a: Bottling plant - Operating Mode

```
FUNCTION_BLOCK "Mode_Selection"          //FB15
TITLE =Mode_Selection
VERSION : 0.1

VAR_INPUT
  Start : BOOL ;
  Stop : BOOL ;
  Auto_Man : BOOL ;
  OM_activate : BOOL ;
END_VAR
VAR_OUTPUT
  Plant_on : BOOL ;
  OM_Man : BOOL ;
  OM_Auto : BOOL ;
END_VAR

BEGIN
NETWORK
TITLE =Plant on/off
      A     #Start;                 // is plant switched on,
      S     #Plant_on;              // set output plant_on;
      AN    #Stop;                  // is plant switched off,
      R     #Plant_on;              // reset output plant_on;
      A     #Plant_on;              //
      =     #Plant_on;              //

NETWORK
TITLE =OM: Manual
      A     #Plant_on;              // is plant switched on,
      AN    #Auto_Man;              // is manual mode preselected,
      A     #OM_activate;           // is input enter_mode active,
      S     #OM_Man;                // than set output manual_mode;
      A(    ;
      ON    #Plant_on;              // is plant switched off,
      O     ;                       // or
      A     #Auto_Man;              // is automatic mode preselected
      A     #OM_activate;           // and enter_mode active,
      )     ;
      R     #OM_Man;                // reset output manual_mode;
      A     #OM_Man;                //
      =     #OM_Man;                //

NETWORK
TITLE =OM: Automatic
      A     #Plant_on;              // is plant switched on,
      A     #Auto_Man;              // is automatic mode preselected,
      A     #OM_activate;           // is input enter_mode active,
      S     #OM_Auto;               // than set output automatic_mode;
      A(    ;
      ON    #Plant_on;              // is plant switched off,
      O     ;                       // or
      AN    #Auto_Man;              // is manual mode preselected
      A     #OM_activate;           // and enter_mode is active,
      )     ;
      R     #OM_Auto;               // reset output automatic_mode;
      A     #OM_Auto;               //
      =     #OM_Auto;               //

END_FUNCTION_BLOCK
```

Solution to Exercise 6.1b: Bottling plant - Conveyor Control (part 1)

```
FUNCTION_BLOCK "Conveyor_Control"   // FB16
TITLE =
VERSION : 0.1

VAR_INPUT
  OM_Man : BOOL ;
  OM_Auto : BOOL ;
  Jog_for : BOOL ;
  Jog_back : BOOL ;
  Sensor_fill : BOOL ;
  Sensor_full : BOOL ;
END_VAR
VAR_OUTPUT
  Conv_for : BOOL ;
  Conv_back : BOOL ;
  Filling_active : BOOL ;
  Full_bottles : WORD ;
END_VAR
VAR
  Filling_time : "TP";              // SFB3 from Standard library -> System Function Blocks
  Bottle_counter : "CTU";           // SFB0 from Standard library -> System Function Blocks
END_VAR
VAR_TEMP
  bottles : INT ;
END_VAR

BEGIN
NETWORK
TITLE =Branch between Manual and Automatic Mode
      SET   ;                       // enforce first check,
      SAVE  ;                       // and set BR Bit to "1";
      A     #OM_Man;                // if manual_mode is active,
      JC    Man;                    // jump to manual mode;
      A     #OM_Auto;               // if automatic_mode is active,
      JC    Auto;                   // jump to automatic mode;
      R     #Conv_for;              // if no OM is active,
      R     #Conv_back;             // reset belt drive,
      R     #Filling_active;        // reset filling_active
      CALL #Bottle_counter (
                    R  := TRUE);    // reset counter

      L     0;                      // reset display of full_bottles
      T     #Full_bottles;
      BEU   ;

NETWORK
TITLE =OM_Man
//Controlling the conveyor belt via JOG keys
Man: A     #Jog_for;
      AN    #Jog_back;
      =     #Conv_for;
      A     #Jog_back;
      AN    #Jog_for;
      =     #Conv_back;
      BEU   ;


// (Continued on the next page)
```

Solution to Exercise 6.1b: Bottling plant - Conveyor Control (part 2)

```
NETWORK
TITLE =OM_Auto
//Start Filling_time
Auto: A     #Sensor_fill;
      =     L      2.0;
      BLD   103;
      CALL #Filling_time (
           IN                  := L      2.0,
           PT                  := T#3S,
           Q                   := #Filling_active);

      NOP   0;

NETWORK
TITLE =OM_Auto
//Counting full bottles
      A     #Sensor_full;
      =     L      2.0;
      BLD   103;
      CALL #Bottle_counter (
           CU                   := L      2.0,
           R                   := FALSE,
           CV                   := #bottles);
      NOP   0;

NETWORK
TITLE =OM_Auto
//Converting #bottles to BCD
//
      L     #bottles;
      ITB   ;
      T     #Full_bottles;
      NOP   0;

NETWORK
TITLE =OM_Auto
//Conveyor_forward is on, as long as filling ist not in progress
      AN    #Filling_active;
      =     #Conv_for;
END_FUNCTION_BLOCK
```

Solution to Exercise 6.1b: Bottling plant - Conveyor Control (part 3)

```
ORGANIZATION_BLOCK "Cycle"
TITLE =
VERSION : 0.1

VAR_TEMP
  OB1_EV_CLASS : BYTE ;       //Bits 0-3 = 1 (Coming event), Bits 4-7 = 1 (Event class 1)
  OB1_SCAN_1 : BYTE ;         //1 (Cold restart scan 1 of OB 1), 3 (Scan 2-n of OB 1)
  OB1_PRIORITY : BYTE ;       //1 (Priority of 1 is lowest)
  OB1_OB_NUMBR : BYTE ;       //1 (Organization block 1, OB1)
  OB1_RESERVED_1 : BYTE ;     //Reserved for system
  OB1_RESERVED_2 : BYTE ;     //Reserved for system
  OB1_PREV_CYCLE : INT ;      //Cycle time of previous OB1 scan (milliseconds)
  OB1_MIN_CYCLE : INT ;       //Minimum cycle time of OB1 (milliseconds)
  OB1_MAX_CYCLE : INT ;       //Maximum cycle time of OB1 (milliseconds)
  OB1_DATE_TIME : DATE_AND_TIME ;  //Date and time OB1 started
  Full_bottles : INT ;
END_VAR

BEGIN
NETWORK
TITLE =Operating mode
CALL "Mode_selection" , "Mode_Selection_DB" (
          Start                 := "Start",
          Stop                  := "Stop",
          Auto_Man              := "Man/Auto",
          OM_activate           := "Enter_Mode",
          Plant_on              := "Plant_on",
          OM_Man                := "Manual_Mode",
          OM_Auto               := "Automatic_Mode");

NETWORK
TITLE =Controlling the belt
CALL "Conveyor_Control" , "Conveyor_Control_DB" (
          OM_Man                := "Manual_Mode",
          OM_Auto               := "Automatic_Mode",
          Jog_for               := "Jog_forward“,
          Jog_back              := "Jog_backward",
          Sensor_fill           := "Filling_Position",
          Sensor_full           := "Counting_Bottles",
          Conv_for              := "Conveyor_forward",
          Conv_back             := "Conveyor_backward",
          Filling_active        := "Filling_in_progress",
          Full_bottles          := "Display");
      NOP   0;

END_ORGANIZATION_BLOCK
```

Solution to Exercise 6.2a: FB1 for the Work Station (part 1)

```
FUNCTION_BLOCK "Station"
TITLE =controlling a work station
AUTHOR : PT41
FAMILY : A2_0
NAME : ST7PRO2
VERSION : 0.0

VAR_INPUT
  Initial : BOOL ;
  Proxy_switch : BOOL ;
  Acknowledge : BOOL ;
  Clock_bit_q : BOOL ;
  Clock_bit_s : BOOL ;
END_VAR
VAR_OUTPUT
  LED : BOOL ;
  Transp_req : BOOL ;
END_VAR
VAR_IN_OUT
  Conv_busy : BOOL ;
END_VAR
VAR
  State : STRUCT
   Process_piece : BOOL ;
   Piece_finished : BOOL ;
   Place_piece_on_conv : BOOL ;
   Wait_for_piece : BOOL ;
   Take_piece_from_conv : BOOL ;
  END_STRUCT ;
FL_initial : BOOL ;
END_VAR
BEGIN
NETWORK
TITLE =Initialization
//By means of input "Initial" the basic state #Process_piece is set
      A     #Initial;
      FP    #FL_initial ;
      S     #State.Process_piece;
      R     #State.Piece_finished;
      R     #State.Place_piece_on_conv;
      R     #State.Wait_for_piece;
      R     #State.Take_piece_from_conv;
      R     #Conv_busy;

NETWORK
TITLE =State: Process_piece
//In this state the workpiece is processed. Processing is terminated
// when the operator acknowledges the termination of the workpiece
//by means of the button "S1"
      AN    #State.Process_piece;
      JC    Pfin;
      S     #LED;               //LED is on permanently ;
      R     #Transp_req;
      A     #Acknowledge;          //when the operator acknowledges,
      R     #State.Process_piece;  //a change of state is performed;
      R     #LED;
      S     #State.Piece_finished;

// (Continued on the next page)
```

Solution to Exercise 6.2a: FB1 for the Work Station (part 2)

```
NETWORK
TITLE =State: Piece_finished
//In the State #Piece_finished the operator waits for the permission
//to place the workpiece on the conveyor. The signal #Conv_busy indicates,
//whether the conveyor is busy or not. When the Conveyor is free, a state change
//to the state Place_piece_on_conv is performed.
Pfin: AN    #State.Piece_finished;
      JC    PpCo;
      A     #Clock_bit_s;           //slow flashing;
      =     #LED;
      AN    #Conv_busy;             //when the conveyor is free,
      S     #Conv_busy;             //it is marked busy
      R     #LED;                   //an a state change is performed;
      R     #State.Piece_finished;
      S     #State.Place_piece_on_conv;

NETWORK
TITLE =State: Place_piece_on_conv
PpCo: AN    #State.Place_piece_on_conv;
      JC    Wait;
      A     #Clock_bit_q;           //quick flashing;
      =     #LED;
      A     #Proxy_switch;          //When the piece is placed on the conveyor,
      S     #Transp_req;            //the transport is started,
      R     #LED;                   //and the LED is cleared;
      A     #Transp_req;            //When the belt is moving,
      AN    #Proxy_switch;          //and the workpiece has left the proxy switch,
      R     #State.Place_piece_on_conv; // a state change is performed;
      S     #State.Wait_for_piece;

NETWORK
TITLE =State: Wait_for_piece
//Waiting for an new raw piece. The arrival of a new piece is indicated by
//the proxy switch of the conveyor
Wait: AN    #State.Wait_for_piece;
      JC    TpCo;
      R     #LED;                   //the LED is switched off;
      A     #Proxy_switch;          //a new raw piece arrives,
      R     #Transp_req;            //the conveyor is stopped,
      R     #State.Wait_for_piece;  //and a state change is performed;
      S     #State.Take_piece_from_conv;

NETWORK
TITLE =State: Take_piece_from_conv
//In this State the new raw piece is taken from the conveyor to
//the working place
TpCo: AN    #State.Take_piece_from_conv;
      JC    END;
      A     #Clock_bit_q;           //the LED flashes quickly
      =     #LED;                   //
      AN    #Proxy_switch;          //when the raw piece is taken from the belt,
      R     #Conv_busy;             //the conveyor is set free,
      R     #LED;                   //the LED is switched off
      R     #State.Take_piece_from_conv; //and a state change is performed;
      S     #State.Process_piece;
END: BEU   ;

END_FUNCTION_BLOCK
```

Solution to Exercise 6.2a: FB2 for the Transport (part 1)

```
FUNCTION_BLOCK "Transport"
TITLE =Controlling the conveyor belt
VERSION : 0.1
VAR_INPUT
  Initial : BOOL ;
  L_Barrier : BOOL ;
  Acknowledge : BOOL ;
  Transp_req : BOOL ;
  Clock_Bit : BOOL ;
END_VAR
VAR_OUTPUT
  LED : BOOL ;
  Conv_right : BOOL ;
  Conv_left : BOOL ;
END_VAR
VAR
  State : STRUCT
   Waiting : BOOL ;
   Conv_right : BOOL ;
   Assembly : BOOL ;
   Conv_left : BOOL ;
  END_STRUCT ;
FL_initial : BOOL ;
END_VAR
BEGIN
NETWORK
TITLE =Initialization
      A     #Initial;
      FP    #FL_initial ;
      S     #State.Waiting;
      R     #State.Conv_right;
      R     #State.Assembly;
      R     #State.Conv_left;
NETWORK
TITLE =State: Waiting
//The belt waits in this state for a finished part.
      AN    #State.Waiting;
      JC    RECH;
      R     #Conv_right;
      R     #Conv_left;
      R     #LED;
      A     #Transp_req;
      R     #State.Waiting;
      S     #State.Conv_right;
NETWORK
TITLE =State: Conv_right
//This state describes the finished part's transport in the direct final assembly
RECH: AN    #State.Conv_right;
      JC    ENDM;
      S     #Conv_right;
      A     #Clock_Bit;
      =     #LED;
      AN    #L_Barrier;
      R     #Conv_right;
      R     #State.Conv_right;
      S     #State.Assembly;
      AN    #L_Barrier;
      =     #L_Barrier;
// (Continued on the next page)
```

Solution to Exercise 6.2a: FB2 for Transport (part 2)

```
NETWORK
TITLE =State: Assembly
//In this state, the finished part is removed and a new raw piece is laid
//on the belt. After that, the raw piece's transport in the direction of the empty
//processing station is started with S4.
//
ENDM: AN    #State.Assembly;
      JC    LINK;
      S     #LED;
      A     #Acknowledge;
      R     #LED;
      R     #State.Assembly;
      S     #State.Conv_left;

NETWORK
TITLE =State: Conv_left
//In the state, the raw piece's transport to the station takes place, that delivered
//the finished piece.

LINK: AN    #State.Conv_left;
      JC    ENDE;
      S     #Conv_left;
      A     #Clock_Bit;
      =     #LED;
      AN    #Transp_req;
      R     #Conv_left;
      R     #State.Conv_left;
      S     #State.Waiting;
ENDE: BEU   ;

END_FUNCTION_BLOCK
```

Solution to Exercise 6.2a: OB1

```
ORGANIZATION_BLOCK "Cycle"
TITLE = "Main Program Sweep (Cycle)"
VERSION : 0.1

VAR_TEMP
  OB1_EV_CLASS : BYTE ;          //Bits 0-3 = 1 (Coming event), Bits 4-7 = 1 (Event class 1)
  OB1_SCAN_1 : BYTE ;            //1 (Cold restart scan 1 of OB 1), 3 (Scan 2-n of OB 1)
  OB1_PRIORITY : BYTE ;          //1 (Priority of 1 is lowest)
  OB1_OB_NUMBR : BYTE ;          //1 (Organization block 1, OB1)
  OB1_RESERVED_1 : BYTE ;        //Reserved for system
  OB1_RESERVED_2 : BYTE ;        //Reserved for system
  OB1_PREV_CYCLE : INT ;         //Cycle time of previous OB1 scan (milliseconds)
  OB1_MIN_CYCLE : INT ;          //Minimum cycle time of OB1 (milliseconds)
  OB1_MAX_CYCLE : INT ;          //Maximum cycle time of OB1 (milliseconds)
  OB1_DATE_TIME : DATE_AND_TIME ;  //Date and time OB1 started
END_VAR

BEGIN
NETWORK
TITLE =Invoke the station control block

     CALL "Station" , "Station_DB" (
          Initial                  := "INITIALIZATION",
          Proxy_switch             := "INI1",
          Acknowledge              := "S1",
          Clock_bit_q              := "CLOCK_BIT_FAST",
          Clock_bit_s              := "CLOCK_BIT_SLOW",
          LED                      := "H1",
          Transp_req               := "Transport_DB".Transp_req);

NETWORK
TITLE =Invoke the transport control block

     CALL "Transport" , "Transport_DB" (
          Initial                  := "INITIALIZATION",
          L_Barrier                := "LB1",
          Acknowledge              := "S4",
          Clock_bit                := "CLOCK_BIT_FAST",
          LED                      := "H4",
          Conv_right               := "K1_CONVR",
          Conv_left                := "K2_CONVL");

END_ORGANIZATION_BLOCK
```

Solution to Exercise 6.2b: Extension to 3 Stations (FB10, part 1)

```
FUNCTION_BLOCK "ASSEMBLY_LINE"
TITLE =
VERSION : 0.1
VAR
  Station_1 : "STATION";
  Station_2 : "STATION";
  Station_3 : "STATION";
  Transport : "TRANSPORT";
  Conv_busy : BOOL ;
END_VAR
VAR_TEMP
  trans_1 : BOOL ;
  trans_2 : BOOL ;
  trans_3 : BOOL ;
  trans : BOOL ;
END_VAR

BEGIN
NETWORK
TITLE =Invoke Station_1

CALL #Station_1 (
        Initial          := "INITIALIZATION",
        Proxy_switch     := "INI1",
        Acknowledge      := "S1",
        Clock_bit_q      := "CLOCK_BIT_FAST",
        Clock_bit_s      := "CLOCK_BIT_SLOW",
        LED              := "H1",
        Transp_req       := #trans_1,
        Conv_busy        := #Conv_busy);

NETWORK
TITLE =Invoke Station_2

CALL #Station_2 (
        Initial          := "INITIALIZATION",
        Proxy_switch     := "INI2",
        Acknowledge      := "S2",
        Clock_bit_q      := "CLOCK_BIT_FAST",
        Clock_bit_s      := "CLOCK_BIT_SLOW",
        LED              := "H2",
        Transp_req       := #trans_2,
        Conv_busy        := #Conv_busy);

NETWORK
TITLE =Invoke Station_3

    CALL #Station_3 (
        Initial          := "INITIALIZATION",
        Proxy_switch     := "INI3",
        Acknowledge      := "S3",
        Clock_bit_q      := "CLOCK_BIT_FAST",
        Clock_bit_s      := "CLOCK_BIT_SLOW",
        LED              := "H3",
        Transp_req       := #trans_3,
        Conv_busy        := #Conv_busy);

// (Continued on the next page)
```

Solution to Exercise 6.2b: Extension to 3 Stations (FB10, part 2)

```
 NETWORK
TITLE =Linking the outputs to inputs

      O     #trans_1;
      O     #trans_2;
      O     #trans_3;
      =     #trans;

NETWORK
TITLE =Invoke Transport

CALL #Transport (
          Initial                 := "INITIALIZATION",
          L_Barrier               := "LB1",
          Acknowledge             := "S4",
          Transp_req              := #trans,
          Clock_Bit               := "CLOCK_BIT_FAST",
          LED                     := "H4",
          Conv_right              := "K1_CONVR",
          Conv_left               := "K2_CONVL");

END_FUNCTION_BLOCK


ORGANIZATION_BLOCK "CYCLE"
TITLE =
VERSION : 0.1

VAR_TEMP
  OB1_EV_CLASS : BYTE ;       //Bits 0-3 = 1 (Coming event), Bits 4-7 = 1 (Event class 1)
  OB1_SCAN_1 : BYTE ;         //1 (Cold restart scan 1 of OB 1), 3 (Scan 2-n of OB 1)
  OB1_PRIORITY : BYTE ;       //1 (Priority of 1 is lowest)
  OB1_OB_NUMBR : BYTE ;       //1 (Organization block 1, OB1)
  OB1_RESERVED_1 : BYTE ;     //Reserved for system
  OB1_RESERVED_2 : BYTE ;     //Reserved for system
  OB1_PREV_CYCLE : INT ;      //Cycle time of previous OB1 scan (milliseconds)
  OB1_MIN_CYCLE : INT ;       //Minimum cycle time of OB1 (milliseconds)
  OB1_MAX_CYCLE : INT ;       //Maximum cycle time of OB1 (milliseconds)
  OB1_DATE_TIME : DATE_AND_TIME ;  //Date and time OB1 started
END_VAR

BEGIN
NETWORK
TITLE =Invoke Assembly Line

      CALL "ASSEMBLY_LINE" , "ASSEMBLY_LINE_DB" ;
      NOP   0;

END_ORGANIZATION_BLOCK
```

Solution to Exercise 7.2: Testing the Data Block

```
FUNCTION FC 72 : INT
TITLE =Exercise 7.2: Testing Data Blocks
VERSION : 0.1

VAR_INPUT
  DB_NUM : WORD ;
END_VAR
VAR_TEMP
  I_DB_Length : WORD ;
  I_RET_VAL : INT ;
  I_Write_Protect : BOOL ;
END_VAR
BEGIN
NETWORK
TITLE =Testing DB
//only for S7-400

      CALL SFC 24 (
           DB_NUMBER      := #DB_NUM,
           RET_VAL        := #I_RET_VAL,
           DB_LENGTH      := #I_DB_Length,
           WRITE_PROT     := #I_Write_Protect);
      L     #I_RET_VAL;
      L     W#16#0;
      ==I   ;
      JC    DBOK;                    // DB available in Working Memory
      TAK   ;
      L     W#16#80A1;
      ==I   ;
      JC    NODB;                    // DB not available on CPU
      TAK   ;
      L     W#16#80B1;
      ==I   ;
      JC    NODB;                    // DB available in Working Memory
      TAK   ;
      L     W#16#80B2;
      ==I   ;
      JC    DBLM;                    // DB only in Load Memory
NODB: L     -1;
      T     #RET_VAL;                // DB not available on CPU
      BEU   ;
DBLM: L     1;
      T     #RET_VAL;                // DB only in Load Memory
      BEU   ;
DBOK: L     0;
      T     #RET_VAL;                // DB available in working Memory

END_FUNCTION
```

Solution to Exercise 7.3: Creating a DB

```
ORGANIZATION_BLOCK OB 100
TITLE =Exercise 7.3: Generating a DB
//Version for S7-400
VERSION : 0.1


VAR_TEMP
  OB100_EV_CLASS : BYTE ;        //16#13, Event class 1, Entering event state, Event
                                 // logged in diagnostic buffer
  OB100_STRTUP : BYTE ;          //16#81/82/83/84 Method of startup
  OB100_PRIORITY : BYTE ;        //27 (Priority of 1 is lowest)
  OB100_OB_NUMBR : BYTE ;        //100 (Organization block 100, OB100)
  OB100_RESERVED_1 : BYTE ;      //Reserved for system
  OB100_RESERVED_2 : BYTE ;      //Reserved for system
  OB100_STOP : WORD ;            //Event that caused CPU to stop (16#4xxx)
  OB100_STRT_INFO : DWORD ;      //Information on how system started
  OB100_DATE_TIME : DATE_AND_TIME ;   //Date and time OB100 started
END_VAR

BEGIN
NETWORK
TITLE =Creating DB10

     CALL SFC 22(
          LOW_LIMIT       := W#16#A,       // identical with dec. 10 (DB10)
          UP_LIMIT        := W#16#A,       //   "
          COUNT           := W#16#28,      // identical with dec. 40 ( 40Bytes)
          RET_VAL         := MW     0,
          DB_NUMBER       := QW    38);

END_ORGANIZATION_BLOCK
```

Solution to Exercise 7.4: Copying a DB from Load into Working Memory

```
ORGANIZATION_BLOCK OB 1
TITLE =Exercise: Copying a DB from Load into Working Memory
//Version für S7-400
VERSION : 2.10

VAR_TEMP
  OB1_EV_CLASS : BYTE ;       //Bits 0-3 = 1 (Coming event), Bits 4-7 = 1 (Event class 1)
  OB1_SCAN_1 : BYTE ;         //1 (Cold restart scan 1 of OB 1), 3 (Scan 2-n of OB 1)
  OB1_PRIORITY : BYTE ;       //1 (Priority of 1 is lowest)
  OB1_OB_NUMBR : BYTE ;       //1 (Organization block 1, OB1)
  OB1_RESERVED_1 : BYTE ;     //Reserved for system
  OB1_RESERVED_2 : BYTE ;     //Reserved for system
  OB1_PREV_CYCLE : INT ;      //Cycle time of previous OB1 scan (milliseconds)
  OB1_MIN_CYCLE : INT ;       //Minimum cycle time of OB1 (milliseconds)
  OB1_MAX_CYCLE : INT ;       //Maximum cycle time of OB1 (milliseconds)
  OB1_DATE_TIME : DATE_AND_TIME ;  //Date and time OB1 started
END_VAR
BEGIN
NETWORK
TITLE =


     A    I     28.0;
     FP   M     0.0;
     JNB  _001;
     CALL SFC   20 (
          SRCBLK          := P#DB20.DBX 0.0 BYTE 40,
          RET_VAL         := QW    38,
          DSTBLK          := P#DB10.DBX 0.0 BYTE 40);
_001: NOP   0;
END_ORGANIZATION_BLOCK
```

Solution to Exercise 7.5: Initialize DB

```
FUNCTION FC 75 : BOOL
TITLE =Exercise 7.5: Initializing DB (only S7-400)
VERSION : 0.1

VAR_INPUT
  DB_NUM : WORD ;
  INI : BYTE ;
END_VAR
VAR_TEMP
  I_RET_VAL : INT ;
  I_DB_Length : WORD ;
  I_WRITE_PROT : BOOL ;
  I_ANY : ANY ;
  DB_No : WORD ;
  I_INI : BYTE ;
  I_RET_VAL1 : INT ;
END_VAR
BEGIN
NETWORK
TITLE =
//Check if DB is in work memory
     CALL "TEST_DB" (
          DB_NUMBER             := #DB_NUM,
          RET_VAL               := #I_RET_VAL,
          DB_LENGTH             := #I_DB_Length,
          WRITE_PROT            := #I_WRITE_PROT);

     L     #I_RET_VAL;
     L     W#16#0;
     ==I   ;                          // DB in work memory
     AN    #I_WRITE_PROT;
     JC   OK;
     CLR   ;                          // Initialization not possible
     =     #RET_VAL;                  // Return FALSE
     BEU   ;
OK:  LAR1  P##I_ANY;                  // Assign temp. ANY variable
     L     B#16#10;                   // Identifier for ANY
     T     LB [AR1,P#0.0];            // to Byte-Offset 0
     L     B#16#2;                    // Identifier for data type BYTE
     T     LB [AR1,P#1.0];            // to Byte-Offset 1
     L     #I_DB_Length;              // load DB length in bytes
     T     LW [AR1,P#2.0];            // to Byte-Offset 2
     L     #DB_NUM;                   // load DB number
     T     LW [AR1,P#4.0];            // to Byte-Offset 4
     L     P#DBX 0.0;                 // load pointer to DBX0.0
     T     LD [AR1,P#6.0];            // to Byte-Offset 6
     L     #INI;                      // Initialization byte
     T     #I_INI;                    // in temp. variable

     CALL SFC 21 (
          BVAL          := #I_INI,    // only possible with temp. Var.
          RET_VAL       := #I_RET_VAL,
          BLK           := #I_ANY);
     SET   ;
     =     #RET_VAL;
     BE    ;

END_FUNCTION
```

Solution to Exercise 7.7: Counter Block with "Contact Debouncing" Function

```
FUNCTION_BLOCK FB 71
TITLE =Exercise 7.7:
//Version for S7-300 16 bit modules
VERSION : 0.1

VAR_INPUT
  CU : BOOL ;
  R : BOOL ;
  PV : INT ;
  PT : TIME ;
END_VAR
VAR_OUTPUT
  Q : BOOL ;
  CV : INT ;
END_VAR
VAR
  Pulse_Counter : "CTU";
  Pulse_Time : "TON";
END_VAR
VAR_TEMP
  Edge_memory : BOOL ;
END_VAR

BEGIN
NETWORK
TITLE =

CALL #Pulse_Time (
        IN              := #CU,
        PT              := #PT,
        Q               := #Edge_memory);


CALL #Pulse_Counter (
        CU              := #Edge_memory,
        R               := #R,
        PV              := #PV,
        Q               := #Q,
        CV              := #CV);

END_FUNCTION_BLOCK
```

Solution to Exercise 8.1: Error Handling in FC43 (part 1)

```
FUNCTION FC 81 : INT
TITLE =Exercise 8.1: Calculation of sum, mean value with error handling
// Solution for S7-300/400
VERSION : 0.0
VAR_INPUT
  Measured_values : ANY ;
END_VAR
VAR_OUTPUT
  Sum : REAL ;
  Mean_value : REAL ;
END_VAR
VAR_TEMP
  Num_Elements : WORD ;
  L_Counter : WORD;
  DB_No : WORD ;
  Sum_1 : REAL ;
  sfc_ret_val : INT ;
  sfc_prgflt : DWORD ;
  sfc_accflt : DWORD ;
  I_BR : BOOL ;              //user BR bit
END_VAR
BEGIN
NETWORK
TITLE =
      L     P##Measured_values;   // Load area pointer on "ANY" pointer
      LAR1  ;                     // Area pointer in AR1
      L     B [AR1,P#1.0];        // Read identifier for data type
      L     8;                    // Load identifier REAL (16#08)
      ==I   ;
      L     -1;                   // Identifier for data type not equal to REAL
      JCN  ERRO;                  // Jump if data type not equal to REAL

// The following events are masked:
// Faulty number of a Gobal DB
// Faulty number of an Instance DB,
// Area error in reading,
// Area length error in reading
      CALL SFC 36 (
           PRGFLT_SET_MASK  := DW#16#40C0014,
           ACCFLT_SET_MASK  := DW#16#0,
           RET_VAL          := #sfc_ret_val,
           PRGFLT_MASKED    := #sfc_prgflt,
           ACCFLT_MASKED    := #sfc_accflt);

      L     W [AR1,P#2.0];        // Load number of array elements
      T     #Num_Elements;        // Initialize loop counter
      L     W [AR1,P#4.0];        // Load DB number or 0
      T     #DB_No;               // If: DB_No=,0 then: OPN DB[DB_No]=NOP
      OPN   DB [#DB_No];          // Run-time error is now masked
      L     D [AR1,P#6.0];        // Load area pointer to actual address
      LAR1  ;                     // in AR1, error!! if area identifier "DI"
      L     0.000000e+000;        // 0 to Accu1 (Sum =0.0)
      L     #Num_Elements;        // Counter to ACCU1; Sum=0 to ACCU2
BEGN: T     #L_Counter;           // Set L_Counter
      TAK   ;                     // Sum in ACCU1
      L     D [AR1,P#0.0];        // Array element in ACCU1, Sum in ACCU2
      +R    ;                     // Sum in ACCU1
      +AR1  P#4.0;                // Incrementing AR1 by 4 units
// (Continued on the next page)
```

Solution to Exercise 8.1: Error Handling in FC43 (part 2)

```
      L     #L_Counter;             // L_Counter in ACCU1, Sum in ACCU2
      LOOP  BEGN;                   // Decrementing and jumping
      TAK   ;                       // Sum in ACCU1
      T     #Sum_1;                 // Sum to #Sum_1
// Error evaluation
      CALL SFC 38 (
           PRGFLT_QUERY       := DW#16#40C0014,
           ACCFLT_QUERY       := DW#16#0,
           RET_VAL            := #sfc_ret_val,
           PRGFLT_CLR         := #sfc_prgflt,
           ACCFLT_CLR         := #sfc_accflt);

      L     #sfc_prgflt;            // Check for faulty DB
      L     DW#16#40C0000;
      AD    ;                       // Bitwise "Roundup"
      L     -2;                     // Error code for DB does not exist
      JN    ERRO;                   // Jump if error
      L     #sfc_prgflt;            // Check for area or area length error
      L     DW#16#14;
      AD    ;
      L     -4;                     // Identifier for area or area length error
      JN    ERRO;                   // Jump if error
//
// no error occurred, proceed with "normal" processing
      L     #Sum_1;
      T     #Sum;                   // assign out parameter #Sum
      L     #Num_Elements;          // Sum in ACCU2, Number in ACCU1
      DTR   ;                       // unsigned integer (16-Bit) to REAL
      /R    ;                       // Mean value in ACCU1
      T     #Mean_value;            // Mean value to #Mean_value
      SET   ;                       // Set BR-Bit to 1
      =     #I_BR  ;
       L    0 ;                     // Identifier all O.K.
       T    RET_VAL;
      JU    DMSK;                   // Jump to demasking synchronous error
//
//error evaluation
//
ERRO: CLR   ;                       // Instructions in case of error RLO=0
      =     #I_BR  ;                // BR =0
      T     #RET_VAL;               // Transfer error code to RET_VAL
      L     L#-1;                   // Load invalid Real number
      T     #Sum;
      T     #Mean_value;
DMSK: NOP   0;                      // Demasking the synchronous faults
      CALL SFC 37 (
           PRGFLT_RESET_MASK     := DW#16#40C0014,
           ACCFLT_RESET_MASK     := DW#16#0,
           RET_VAL               := #sfc_ret_val,
           PRGFLT_MASKED         := #sfc_prgflt,
           ACCFLT_MASKED         := #sfc_accflt);
      CLR  ;                        // enforce first check, RLO = 0
      A     #I_BR  ;                // copy user BR Bit
      SAVE  ;                       // into system BR Bit
     BEU  ;
END_FUNCTION
```

Solution to Exercise 9.2: Communication with the SFBs PUT/GET (part 1)

```
// Download the compiled blocks into S7-400

DATA_BLOCK DB 14
VERSION : 0.1

"GET"
BEGIN
END_DATA_BLOCK

DATA_BLOCK DB 15
VERSION : 0.1

"PUT"
BEGIN
END_DATA_BLOCK

ORGANIZATION_BLOCK OB 1
TITLE = S7400 writes into S7-300 and reads from S7-300
AUTHOR : PT41
FAMILY : A2_0
NAME : ST7PRO2
VERSION : 0.0

VAR_TEMP
  OB1_EV_CLASS : BYTE ;       //Bits 0-3 = 1 (Coming event), Bits 4-7 = 1 (Event class 1)
  OB1_SCAN_1 : BYTE ;         //1 (Cold restart scan 1 of OB 1), 3 (Scan 2-n of OB 1)
  OB1_PRIORITY : BYTE ;       //1 (Priority of 1 is lowest)
  OB1_OB_NUMBR : BYTE ;       //1 (Organization block 1, OB1)
  OB1_RESERVED_1 : BYTE ;     //Reserved for system
  OB1_RESERVED_2 : BYTE ;     //Reserved for system
  OB1_PREV_CYCLE : INT ;      //Cycle time of previous OB1 scan (milliseconds)
  OB1_MIN_CYCLE : INT ;       //Minimum cycle time of OB1 (milliseconds)
  OB1_MAX_CYCLE : INT ;       //Maximum cycle time of OB1 (milliseconds)
  OB1_DATE_TIME : DATE_AND_TIME ;  //Date and time OB1 started
  NDR_FLAG_14 : BOOL ;
  ERROR_FLAG_14 : BOOL ;
  DONE_FLAG_15 : BOOL ;
  ERROR_FLAG_15 : BOOL ;
  STATUS_WORD_14 : WORD ;
  STATUS_WORD_15 : WORD ;
END_VAR

BEGIN
NETWORK
TITLE ="SFB_GET"
     CALL SFB   14 , DB    14 (
          REQ                 := I     28.0,
          ID                  := W#16#1,
          NDR                 := #NDR_FLAG_14,
          ERROR               := #ERROR_FLAG_14,
          STATUS              := #STATUS_WORD_14,
          ADDR_1              := P#I 0.0 BYTE 1,
          ADDR_2              := P#I 4.0 WORD 1,
          RD_1                := P#Q 40.0 BYTE 1,
          RD_2                := P#Q 42.0 WORD 1);
```

// (Continued on next page)

Solution to Exercise 9.2: Communication with the SFBs PUT/GET (part 2)

```
NETWORK
TITLE ="SFB_PUT"
     CALL SFB   15 , DB    15 (
          REQ               := I      28.1,
          ID                := W#16#1,
          DONE              := #DONE_FLAG_15,
          ERROR             := #ERROR_FLAG_15,
          STATUS            := #STATUS_WORD_15,
          ADDR_1            := P#Q 12.0 WORD 1,
          SD_1              := P#I 30.0 WORD 1);

NETWORK
TITLE =STATUS_WORD to QW38
     A(    ;
     O     #NDR_FLAG_14;
     O     #ERROR_FLAG_14;
     )     ;
     JCN _002;
     L     #STATUS_WORD_14;
     T     QW     38;
_002: NOP   0;

NETWORK
TITLE =
     A(    ;
     O     #DONE_FLAG_15;
     O     #ERROR_FLAG_15;
     )     ;
     JCN _001;
     L     #STATUS_WORD_15;
     T     QW     38;
_001: NOP   0;

NETWORK
TITLE =
// Otherwise FFFF into QW38
     A     I      28.0;
     BEC   ;
     A     I      28.1;
     BEC   ;
     L     W#16#FFFF;
     T     QW     38;
END_ORGANIZATION_BLOCK
```

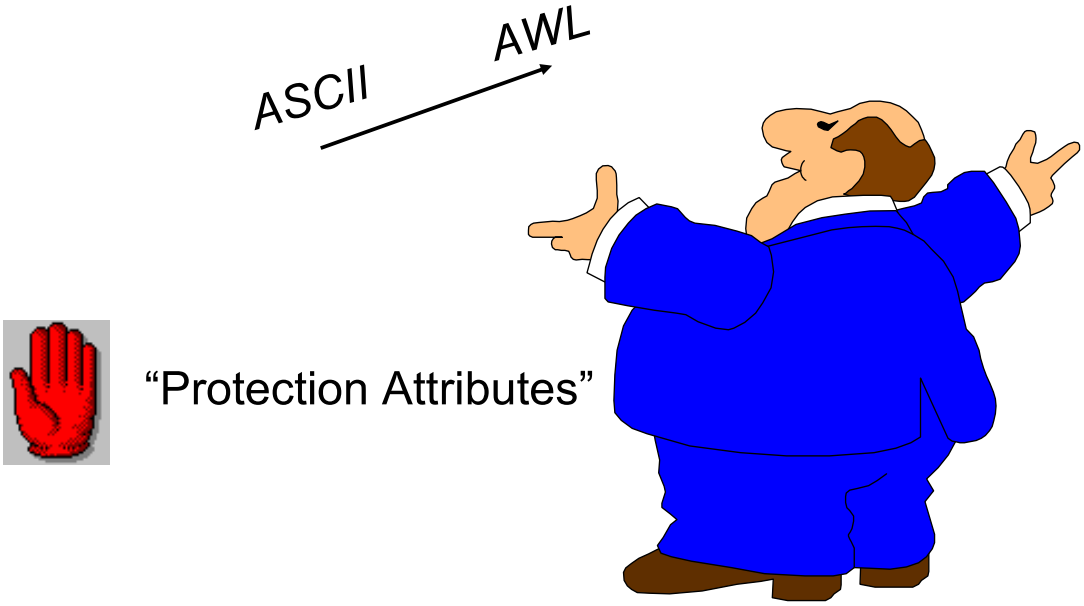
Solution to Exercise 9.3: Communication with the SFBs START/STOP (part 1)

```
// Download the compiled blocks into S7-400

DATA_BLOCK DB 19
VERSION : 0.1
"START"
BEGIN
END_DATA_BLOCK

DATA_BLOCK DB 20
VERSION : 0.1
"STOP"
BEGIN
END_DATA_BLOCK

ORGANIZATION_BLOCK OB 1
TITLE =
VERSION : 0.1
VAR_TEMP
  OB1_EV_CLASS : BYTE ;       //Bits 0-3 = 1 (Coming event), Bits 4-7 = 1 (Event class 1)
  OB1_SCAN_1 : BYTE ;         //1 (Cold restart scan 1 of OB 1), 3 (Scan 2-n of OB 1)
  OB1_PRIORITY : BYTE ;       //1 (Priority of 1 is lowest)
  OB1_OB_NUMBR : BYTE ;       //1 (Organization block 1, OB1)
  OB1_RESERVED_1 : BYTE ;     //Reserved for system
  OB1_RESERVED_2 : BYTE ;     //Reserved for system
  OB1_PREV_CYCLE : INT ;      //Cycle time of previous OB1 scan (milliseconds)
  OB1_MIN_CYCLE : INT ;       //Minimum cycle time of OB1 (milliseconds)
  OB1_MAX_CYCLE : INT ;       //Maximum cycle time of OB1 (milliseconds)
  OB1_DATE_TIME : DATE_AND_TIME ;  //Date and time OB1 started

  DONE_FLAG_20 : BOOL ;
  ERROR_FLAG_20 : BOOL ;
  DONE_FLAG_19 : BOOL ;
  ERROR_FLAG_19 : BOOL ;
  STATUS_WORD_20 : WORD ;
  STATUS_WORD_19 : WORD ;
END_VAR

BEGIN
NETWORK
TITLE =
//Enter STRING "P_PROGRAM" in PI_NAME
     L    'P_PR';
     T    MD   100;
     L    'OGRA';
     T    MD   104;
     L    'M';
     T    MB   108;

NETWORK
TITLE ="SFB_STOP"
     CALL SFB   20 , DB    20 (
          REQ                    := I     28.0,
          DONE                   := #DONE_FLAG_20,
          ERROR                  := #ERROR_FLAG_20,
          STATUS                 := #STATUS_WORD_20);

// (Continued on next page)
```

Solution to Exercise 9.3: Communication with the SFBs START/STOP (part 2)

```
NETWORK
TITLE ="SFB_START"
CALL SFB   19 , DB    19 (
        REQ                 := I     28.1,
        DONE                := #DONE_FLAG_19,
        ERROR               := #ERROR_FLAG_19,
        STATUS              := #STATUS_WORD_19);

NETWORK
TITLE = STATUS_WORD to QW38
     A(    ;
     O     #DONE_FLAG_19;
     O     #ERROR_FLAG_19;
     )     ;
     JCN _001;
     L     #STATUS_WORD_19;
     T     QW     38;
_001: NOP   0;

NETWORK
TITLE = STATUS_WORD to QW38
     A(    ;
     O     #DONE_FLAG_20;
     O     #ERROR_FLAG_20;
     )     ;
     JCN _002;
     L     #STATUS_WORD_20;
     T     QW     38;
_002: NOP   0;

NETWORK
TITLE =

     A     I     28.2;          // Otherwise FFFF into QW38
     BEC   ;
     A     I     28.3;
     BEC   ;
     L     W#16#FFFF;
     T     QW     38;

END_ORGANIZATION_BLOCK
```

Appendix 1: Program Generation with the Text Editor



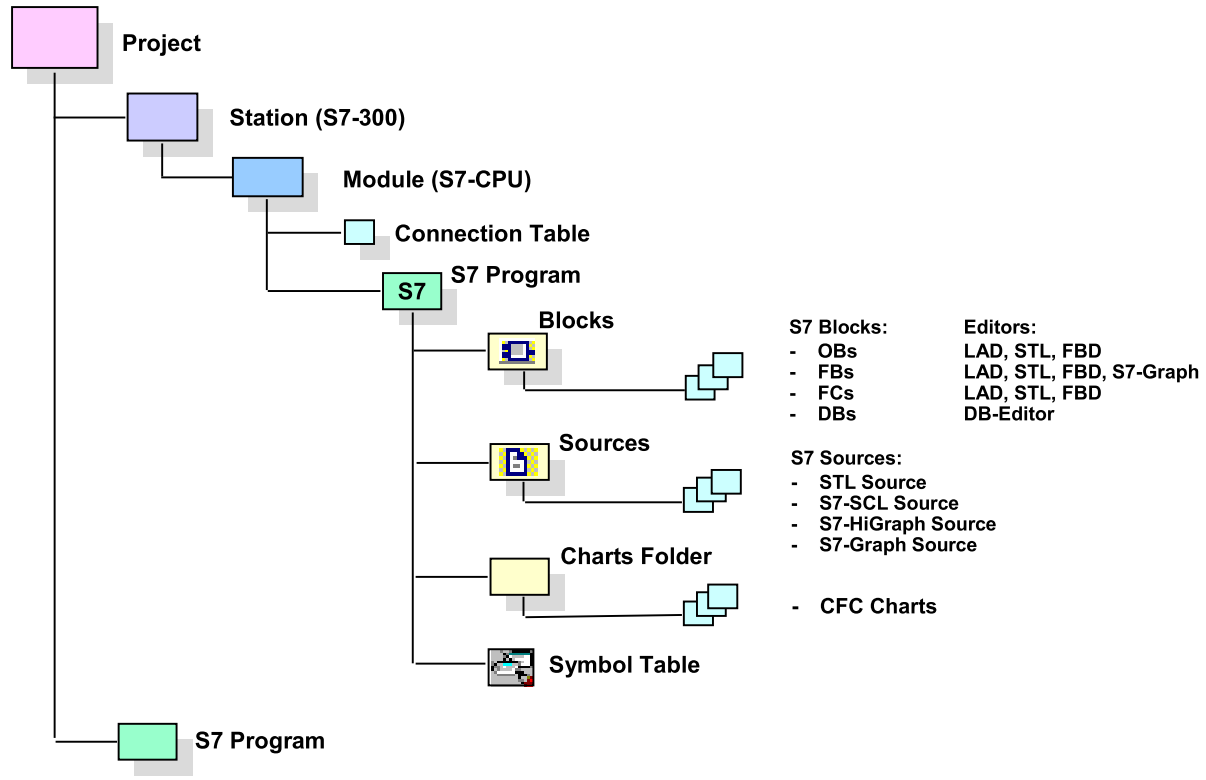
SIMATIC S7

Date: 30.06.2011
File: PRO2_14E.1

SITRAIN Training for Automation and Drives

Contents **Page**

S7 Programs in the Project Structure



SIMATIC S7

Date: 30.06.2011
File: PRO2_14E.2

SITRAIN Training for Automation and Drives

Overview

In order to be able to create a new S7 Program, a project must first of all be generated in the SIMATIC Manager. Subsequently, there are two possibilities for setting up an S7 program folder:

- Module independent: In this case you must insert the program folder for S7-Programs directly underneath the project root. The programs created there can later be assigned to a programmable module.
- Module dependent: In this case the project must contain at least one SIMATIC 300/400 Station with a programmable module (CPU). An S7 program folder is then automatically inserted underneath the programmable module.

If you want to use global symbols in your user program, you should make the corresponding assignment of identifiers and absolute addresses in the symbol table beforehand.

Blocks, Sources and Charts

You can store the S7 Program as user program (block), source or chart. Sources and charts are only used, however, as a basis for block generation in S7 programming. Only blocks can be downloaded to an S7-CPU.

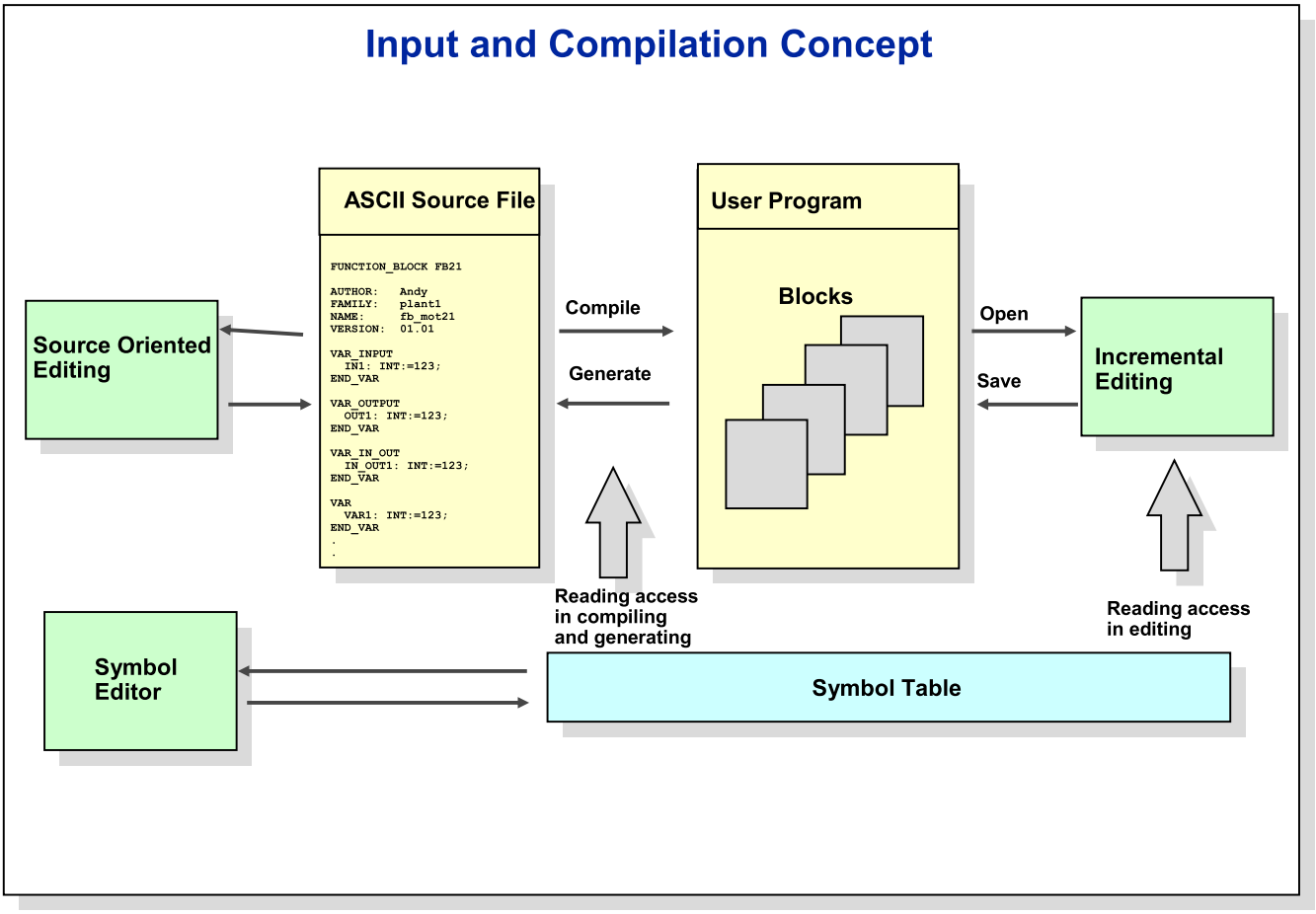
Whether you generate a block, a source or a chart depends on the selected programming language or on the language editor

User Program

Only the blocks of the user program can be downloaded to an S7-CPU. Depending on scope, this includes organization blocks (OBs), functions (FCs), function blocks (FBs) and data blocks (DBs).

The user-defined data types (UDTs) that are created merely simplify the programming. They cannot, however, be downloaded to an S7-CPU.

The same is valid for the variable tables (VATs) in which addresses for the function *Monitor/Modify Variables* are saved.



Input Possibilities

Depending on the programming language that you have chosen for program generation, you can enter your program incrementally and/or source-oriented.

- Incremental Input (STL, LAD, FBD, S7-Graph, S7-HiGraph, CFC)

  Every line or every element is immediately examined for syntax errors after input. Possible existing input errors are indicated (marked in red) and must be corrected before saving.

  Syntax correct inputs are automatically compiled and displayed in black. In incremental input, the symbols used must already be defined in the symbol table, otherwise the input is marked in red and a corresponding error message is displayed in the status bar.

- Source-oriented Input (STL, S7-SCL)

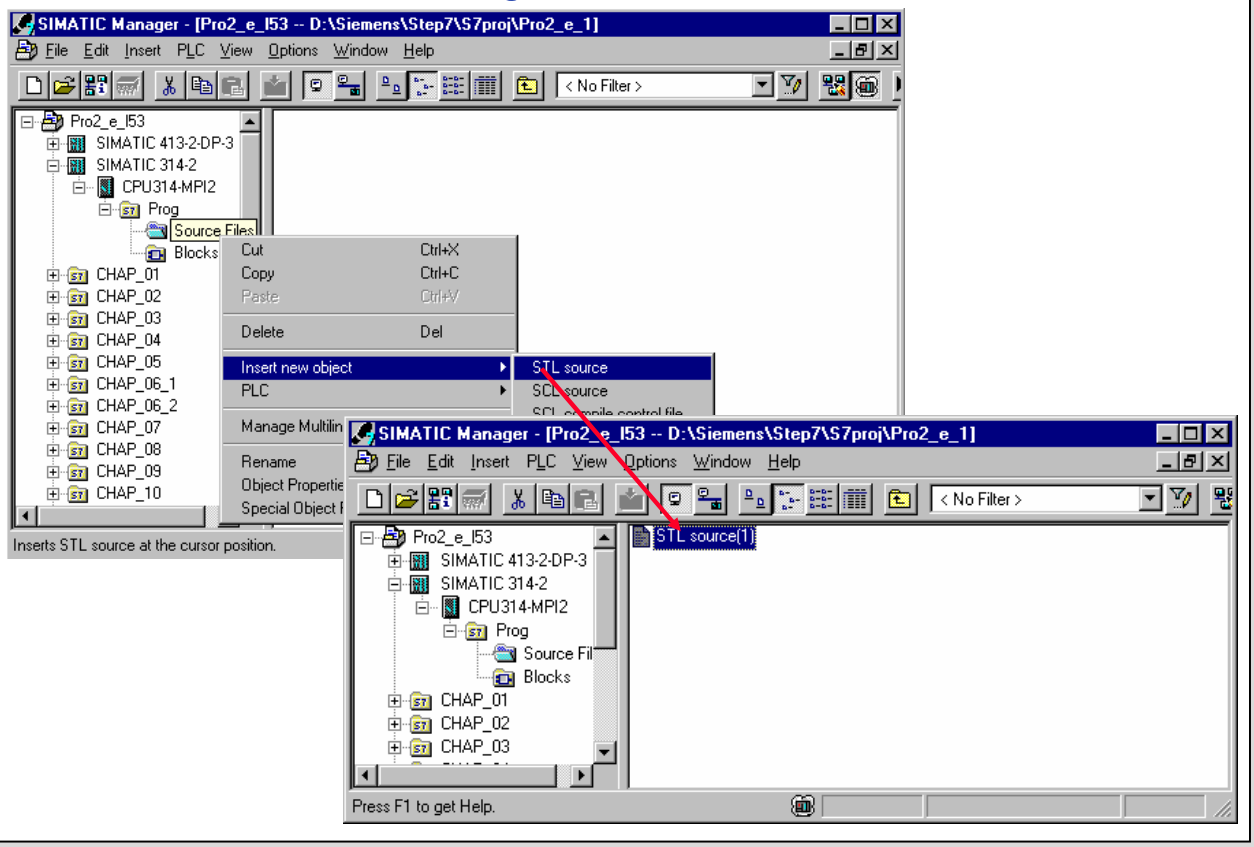
  In the source-oriented input, the program or a block is edited in a text file and the text file is then compiled, whereby errors are first indicated in the compilation by the associated Compiler.

  In the source-oriented input, the symbols must only be defined in the symbol table at the time of compilation. Source files have the advantage that they can be exported - then processed with the tools of choice - and can then be re-imported.

Advantages of the Source-oriented Input

- Several blocks can be stored in a source file (the blocks must, however, be stored is such a way that the called blocks are always located before the calling blocks).
- A source file can be saved with syntax errors.
- You can generate your source file with other editors, import them in the SIMATIC Manager and then compile them in blocks.
- A block protection can only be entered in ASCII mode.
- Changes (addition of block parameters, for example) in nested block calls can be better handled with the ASCII Editor (with Find and Replace, for example), than with the Incremental Editor.

Starting the Text Editor



Starting from the SIMATIC Manager

You start the Text Editor from the SIMATIC Manager. Prerequisite is that you have set up a project with an S7-Program there. You can generate the program dependent or independent of the hardware.

With the Text Editor you process source files exclusively. You subsequently compile these sources into blocks which are stored in the folder Blocks.

Creating a Source

When you want to generate a new source file for the first time, you must first of all create an empty file in the SIMATIC Manager through which you open the Text Editor. When you have opened the Editor you can create further source files there.

- In the SIMATIC Manager select the Source Files folder and insert a file with the menu option *Insert New Object -> STL source*. The new source file appears in the right side of the project window with a preset name.
- In the Text Editor itself, you can simply create a new file using the menu option *File ->New*. In the follow-up dialog you enter the name of the new source file.

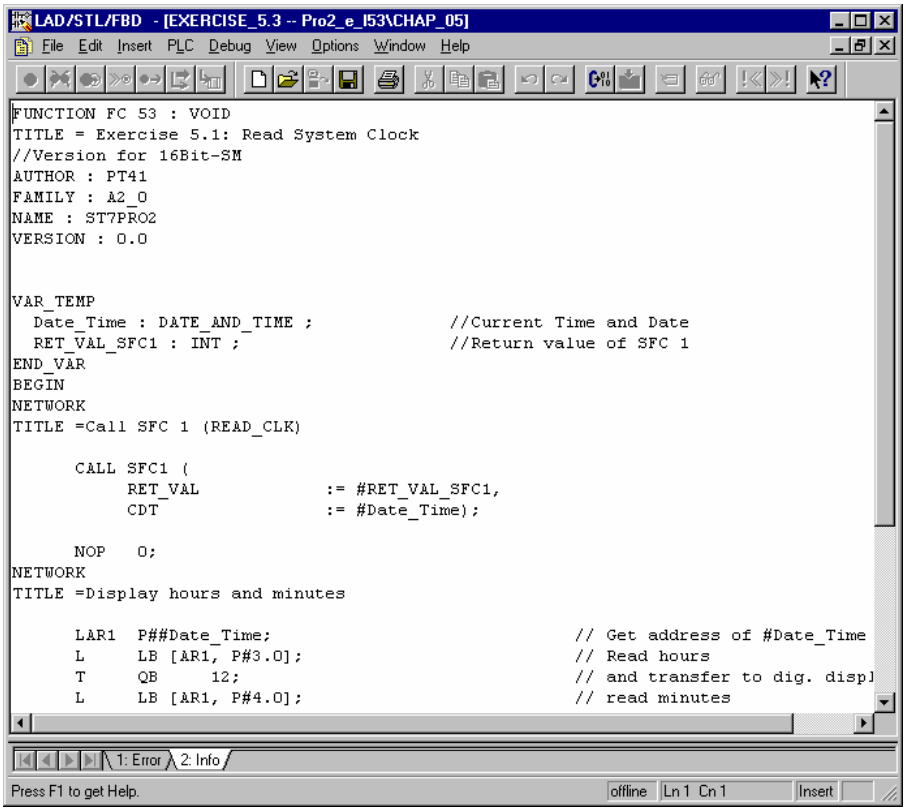
Opening a Source File

You open a source file in the SIMATIC Manager through a double click on its symbol. Alternatively, you can arrive at this using the menu option *Edit -> Open Object* or the corresponding icon in the toolbar.

Generating a Source File

It is also possible to compile already existing blocks back into a source file, in order to further process these. In the Text Editor select the menu option *File -> Generate Source*. In the follow-up dialog you can then select all blocks from which you want to generate a source file.

Program Generation with the Text Editor



Text Editor

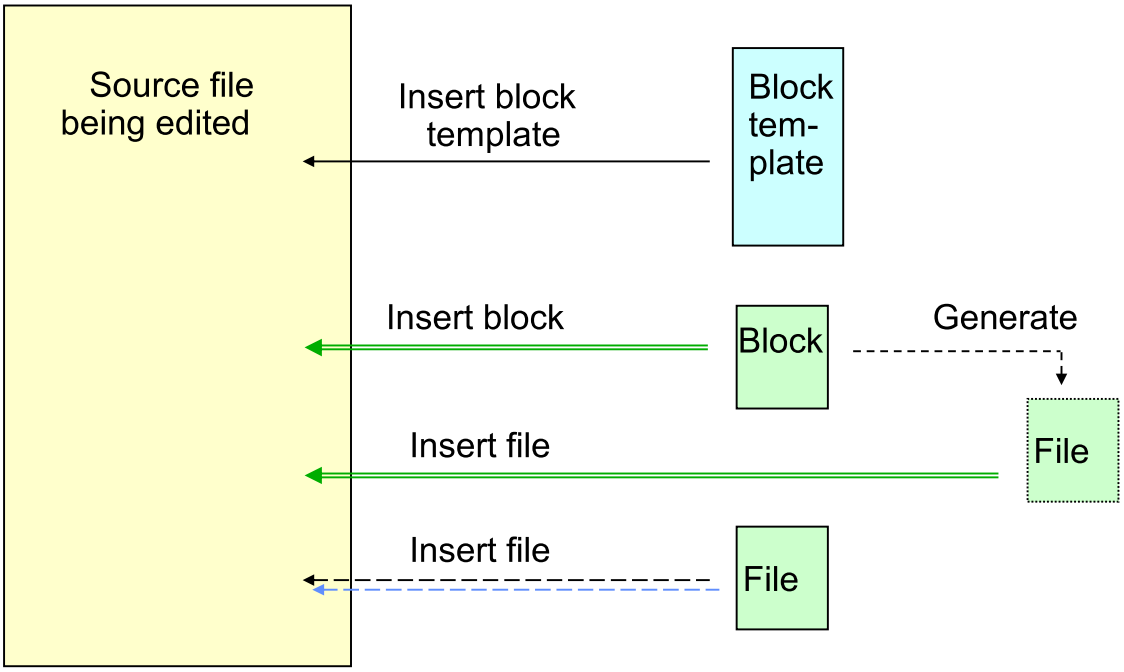
Instead of programming in STL, you can generate your program with the integrated Text Editor and thereby create a source file. You enter your blocks one after the other (possibly several blocks in one source file). A syntax check does not take place.

Settings

Before you begin to program in the Text Editor, you should become familiar with the customization possibilities, in order to be able to work comfortably and according to your personal preferences.

You open a Register Dialog using the menu option *Options Settings*. In the ”Editor” tab you can set the default for the font (type and size) for the source file. The color in which instruction lines are marked is changed in the ”LAD/FBD” tab.

Inserting Block Templates, Blocks and Source Files



SIMATIC S7

Date: 30.06.2011
File: PRO2_14E.6

SITRAIN Training for Automation and Drives

Inserting Block Templates

Block templates for OBs, FBs, FCs, DBs, Instance DBs, DBs from UDTs and UDTs are integrated in the Editor for simpler programming. A block template contains all required keywords in the necessary sequence. You simply delete templates for optional declarations which you do not wish to make. Block templates facilitate the input and adherence to the syntax and structure at the same time.

In order to insert a block template in your source file, select the menu option *Insert -> Block Template -> OB/FB/FC/DB/IDB/ DB from UDT/UDT*.

Inserting Blocks

You can insert the corresponding source code of blocks that have already been generated into your source file. To do so, select the menu option *Insert -> Object -> Block*. In the follow-up dialog select the blocks whose code you wish to insert as text.
A source file is generated from the selected blocks. Their contents is inserted behind the cursor position in the source file just being edited.

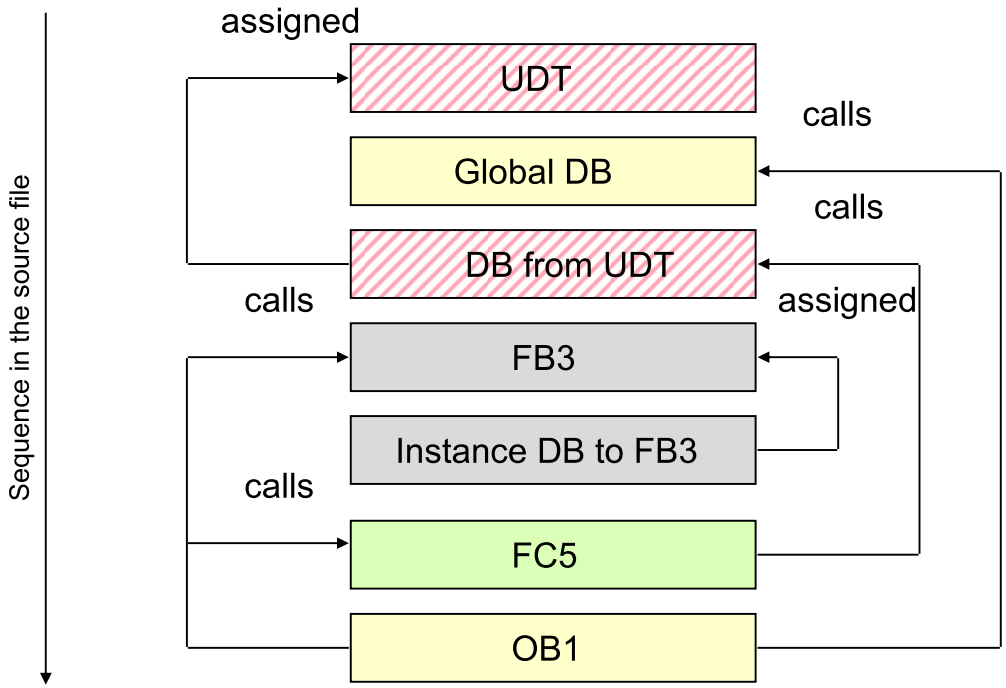
Inserting Source Files

You can insert the contents of any other source files in your source file. To do so, select the menu option *Insert -> Object -> File* and in the follow-up dialog select the file to be inserted.
In this way, the contents of any text file can be inserted in your source file.

Note

Any text contents can also be inserted into your source file using the Windows clipboard.

General Input Rules and Structure



SIMATIC S7

Date: 30.06.2011
File: PRO2_14E.7

SITRAIN Training for Automation and Drives

Input Rules

The following general rules are valid for the generation of user programs as source file:

- The syntax of the STL instructions is the same as in the incremental STL Editor. An exception exists with block calls and the declaration of Arrays and Structures.
- In general, the Text Editor does not take into account upper case or lower case. The exception is jump labels.
- Identify the end of every STL instruction and every variable declaration with a semicolon (;). You can enter more than one instruction per line.
- Begin every comment with two diagonal strokes (//) and end every comment input with the RETURN key.

Block Sequence

With regard to the sequence of the blocks, you must pay attention to the following in the generation of the source file:
Called blocks are located before the calling blocks. This means:

- OB1, which is used most often and which calls other blocks, comes last. Blocks, which on the other hand are called by blocks that are called from OB1, must come before these etc.
- User defined data types (UDTs) come before the blocks in which they are used.
- Data blocks with assigned user defined data types (UDT) come after the UDT.
- Global data blocks come before all blocks from which they are called.
- Data blocks with assigned function blocks (instance DB) come after the function block.

Syntax for Logic Blocks

Configuration	Keyword with Example
Block start with block specification (absolute or symbolic)	ORGANIZATION_BLOCK *OB1* FUNCTION_BLOCK *FB1* FUNCTION FC 1 : int
Block title (optional)	TITLE = *Block title*
Block comment (optional)	*// Block comment*

System attributes for blocks (optional)	{Attr1 := 'block_val1'; // Block attribute1
	Attr2 := 'block_val2'; // Block attribute2
	Attr3 := 'block_val3' // Block attribute3}

Block properties (optional)	KNOW_HOW_PROTECT AUTHOR: *PT41* FAMILY: *Motors* NAME: *Motorone* VERSION: *0815*
Variable declaration part (Declaration type depending on block type Termination of every declaration type with	VAR_IN VAR_OUT VAR_IN_OUT VAR VAR_TEMP .. END_VAR
Instruction part consisting of Networks with Network title Network comment	BEGIN NETWORK TITLE=*first network* //
Block end	END_ORGANIZATION_BLOCK END_FUNCTION_BLOCK END_FUNCTION

Rules

With the input of a logic block, you must pay attention to the following rules:

- In the block start, there is a space between the keyword for the block type and the block specification. In specifying the symbolic block name, you can identify it in quotation marks, so as to guarantee the differentiation between names of local variables and names of the symbol table.
- With functions (FCs), the function type is also given. This can be of the elementary or complex data type and determines the data type of the return value (#RET_VAL). If no value is to be returned, VOID is to be entered.
- The specification of a network number is not allowed.

Block Calls with "CALL"

The syntax for the call of FBs and FCs with the CALL instruction deviates slightly from that in the incremental STL Editor. In a source file you enter the parameters in brackets. The individual parameters are then separated from one another by commas.

Example: CALL FC1 (param1 := I 0.0, param2 := I 0.1);

Comments in the Instruction Part

In order to guarantee a 1:1 representation of the comments in the later editing in the incremental Editor, you must pay attention to the following:

- Block call: In source files, you should retain the sequence of the formal parameters as they are in the block's variable declaration when you assign actual parameters to the formal parameters. Although the sequence of the parameters is of choice, comments to the parameters can however be switched during compilation of the source into blocks.
- With instructions for the access to data blocks that directly follow the "OPN" instruction, it is possible that a loss of instruction comments can occur during the compilation into blocks. In order to avoid this, program in compact form (e.g. L DB5.DBW20; //Comm.) or write in an "NOP" instruction (e.g. OPN DB5; //Comment1 NOP 0; L DBW20; //Comment2).

Syntax for Data Blocks

Configuration	Keyword with Example
Block start with block specification (absolute or symbolic)	DATA_BLOCK *DB 26*
Block title (optional)	TITLE = *Block title*
Block comment (optional)	*// Block comment*

System attributes for blocks (optional)	{Attr1 := 'block_val1'; // Block attribute1
	Attr2 := 'block_val2'; // Block attribute2}

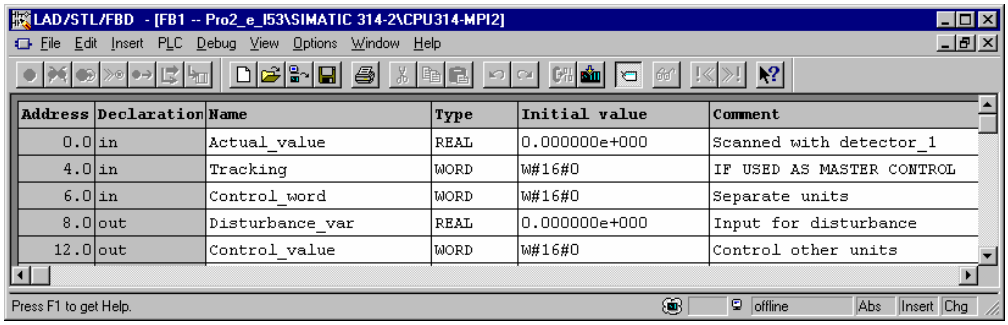
Block properties (optional)	KNOW_HOW_PROTECT AUTHOR: *Müller* FAMILY: *Motors* NAME: *Motorone* VERSION: *0815*
Declaration part - depending on the DB	
Global data block: Variable declaration (optional with initial values)	STRUCT .. .. END_STRUCT
DB from UDT: UDT specification (absolute or symbolic)	UDT *16*
Instance DB: FB specification (absolute or symbolic)	FB *20*
Assignment part with current values	BEGIN ..
Block end	END_DATA_BLOCK

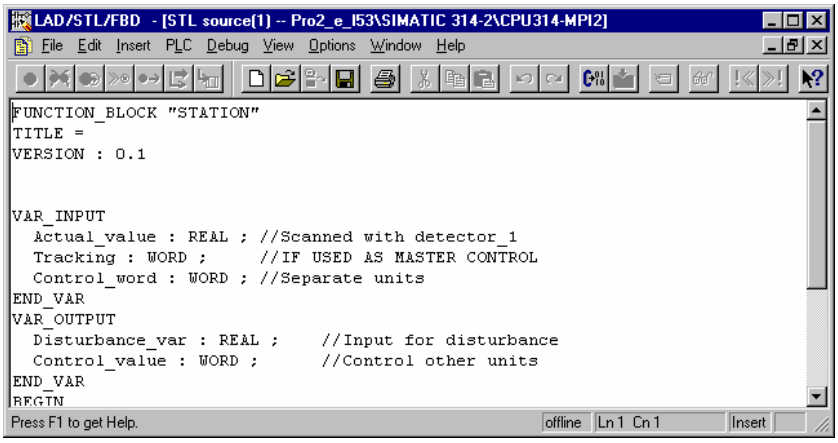
Rules

In inputting data blocks you must pay attention to the following rules:

- You cannot generate a DB 0.
- You can optionally specify current values for all or some variables. For variables, which you do not assign current values, the initial value is assigned - if available - , otherwise the default value for the data type is assigned.
- Instruction comments in the assignment section for current values (between BEGIN and END_DATA_BLOCK) are not displayed in the incremental Editor after compilation into blocks. Therefore only write comments for data blocks in the declaration part.

Rules for Variable Declarations





Variable Types

With logic blocks, the declaration type of the variable is identified by a keyword, that is found in its own line. Depending on the block type, only particular declaration types are allowed.

Declaration Type	Keyword	OB	FB	FC
Input parameter	VAR_INPUT	-	yes	yes
Output parameter	VAR_OUTPUT	-	yes	yes
In/out parameter	VAR_IN_OUT	-	yes	yes
Static variable	VAR	-	yes	-
Temporary variable	VAR_TEMP	yes	yes	yes
Each termination with	END_VAR			

Input Rules

In inputting the variable declaration, you must pay attention to the following:

- The variables must be declared in the sequence of the declaration types. All variables of one type are thereby together.
- The keywords are each found in their own line or are separated by a blank.
- The variable name is found at the beginning of the line and must begin with a letter. It cannot correspond to any of the reserved keywords.
- Optional system attributes can be assigned for the individual parameters after the variable name. The system attributes are enclosed in { } brackets.

  Example: Var_1 { ident1 := 'string1' ; ident2 := 'string2' } : INT;

  Var_2 { message :='TRUE' ; OPERABLE := 'TRUE'} : INT;
- The data type is given, separated by a colon, after the name or after the system attribute. Elementary, complex and user-defined data types are allowed.
- Every variable declaration is ended by a semicolon.
- Comments are separated from the declaration part by two diagonal strokes.

Allocation of Block Attributes

Attribute	Logic Blocks (OB, FB, FC)	Data Blocks	UDT
KNOW_HOW_PROTECT	yes	yes	no
AUTHOR	yes	yes	no
FAMILY	yes	yes	no
NAME	yes	yes	no
VERSION	yes	yes	no
UNLINKED	no	yes	no
READ_ONLY	no	yes	no

System Attributes

You can assign system attributes to blocks, for example, for process diagnostics or control system configuration. They control the message configuration and connection configuration, operator interface functions and the control system configuration.

Block Properties

You can specify the block name, family, version and author with the help of keywords. For that the following is valid:

- Block properties are specified before the variable declaration part.
- There is no (!) semicolon at the end of the line.

Block Protection

You can set up a block protection for logic blocks and data blocks by specifying the keyword KNOW_HOW_PROTECT :

- When you look at the compiled block in the incremental STL Editor, the block's instruction part cannot be looked into.
- Only the in, out and in/out parameters are displayed in the block's variable declaration. The internal variables VAR and VAR_TEMP remain hidden.
- The compiled block can be compiled in a source file, but only as a block without instruction part.

The keyword KNOW_HOW_PROTECT must be entered before all other block attributes.

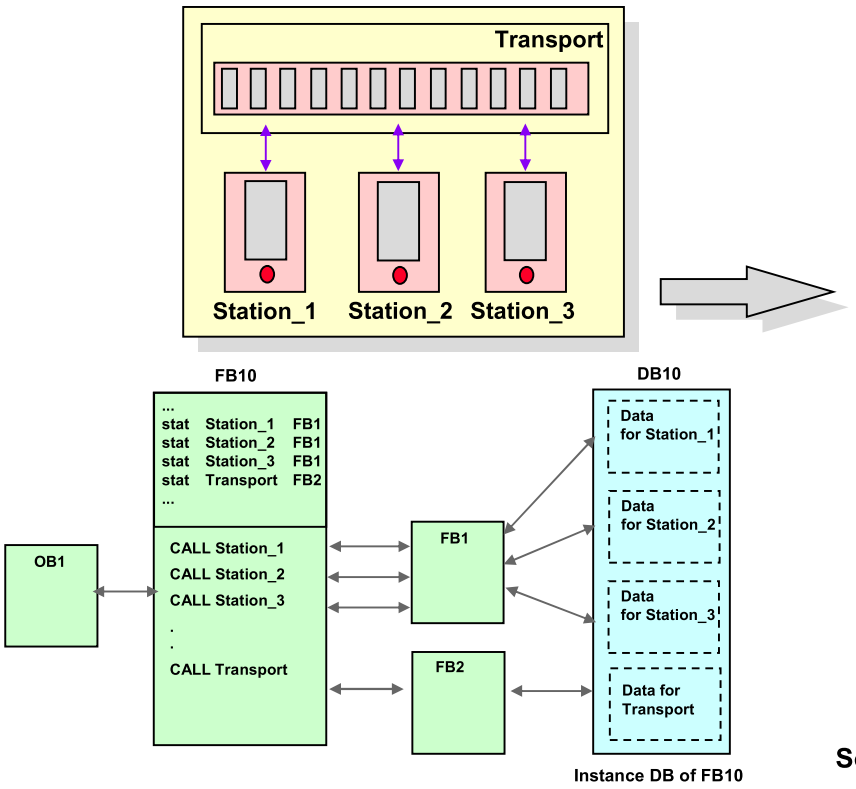
Write Protect READ_ONLY

You can set up a write protect for data blocks in source files so that the data values stored therein cannot be overwritten during program execution. To do so, enter the keyword READ_ONLY . This must be found in its own line directly before the declarations.

UNLINKED

The attribute UNLINKED can only occur with data blocks. It says that the DB is not to be loaded from the load memory into the CPU's work memory.

Exercise A1.1: Creating a Source File




```
FUNCTION_BLOCK "Station"
TITLE =
VERSION : 0.1

VAR_INPUT
  Initial : BOOL ;
  Proximity_switch : BOOL ;
  Acknowledge : BOOL ;
  Clock_bit : BOOL ;
END_VAR
VAR_OUTPUT
  LED : BOOL ;
  Transp_req : BOOL ;
END_VAR
VAR_IN_OUT
  Conv_busy : BOOL ;
END_VAR
VAR
  State : STRUCT
   Busy : BOOL ;
   Completed : BOOL ;
   Waiting : BOOL ;
  END_STRUCT ;
  Aux_1 : BOOL ;
  Aux_2 : BOOL ;
END_VAR
BEGIN
NETWORK
TITLE =Initialization
//"Busy" is set with the
//initialization
      A     #Initial;
...
```

Source File: "Assembly Line"

SIMATIC S7

Date: 30.06.2011
File: PRO2_14E.12

SITRAIN Training for Automation and Drives

Overview

First of all, a source file is to be created from the final program of Exercise 6.2b "Assembly line". Subsequently, additional count functionality is to be introduced into the block for the parts transport.

Goal of the Exercise

In the *Sources* folder of the PRO2-Project (program folder "Assembly line"), generate a source file that includes the entire user program of Exercise 6.2b and that can be compiled without error messages.

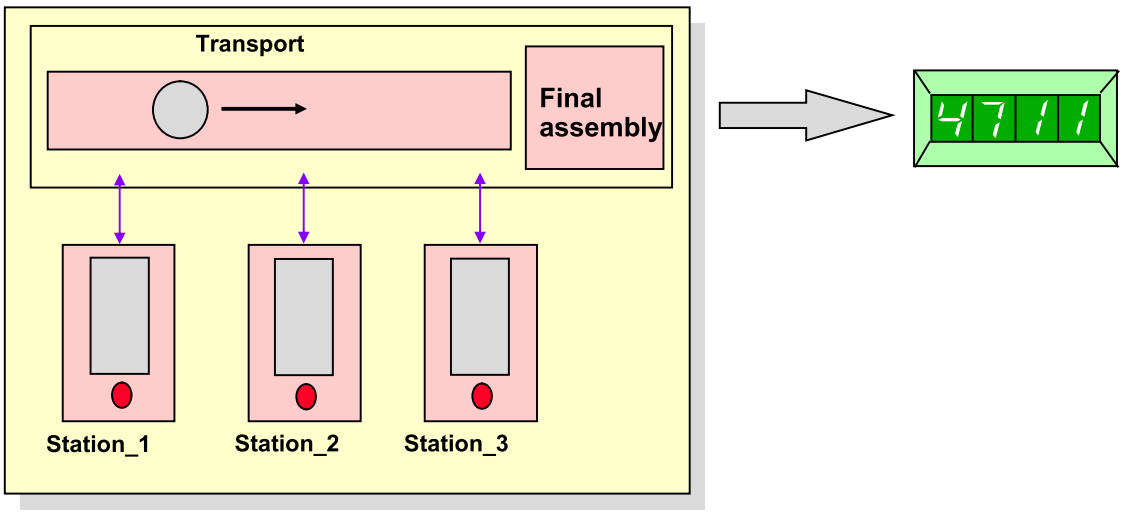
Procedure

1. First of all, in the *Blocks* folder of the program folder "Assembly line", open a block of your choice with the help of the STL/LAD/FBD Editor.
2. Then select the menu option *File -> Generate Source*. The dialog "New" for inputting the desired name for the source file pops up.
3. Enter the name of the source file (assembly line, for example) and confirm the dialog with "OK". The follow-up dialog "Select STEP7 Blocks" pops up.
4. Select the desired block and acknowledge with "OK".

   Note: You can select the check box "Program structure (XREF) sorted". That way, the blocks are automatically arranged in the correct sequence in the source file.

   Generation of the source file is started.
5. With the Text Editor, open the created source file.
6. With the help of the menu option *File -> Check Consistency,* test if the source file can be compiled error-free.

Exercise A1.2: Counting the Finished Parts



Goal of the Exercise

In the function block "Transport", integrate a counter that counts the completed parts coming to the final assembly. The counter's properties should include the following functionality:

- The counter is to be implemented with the help of the IEC 61131-3 conforming up-counter (SFB 0 "CTU").
- With every falling edge of the light barrier, the counter increments, in the state #Transport_right, its count.
- The counter is reset with the input signal #Initial.
- The current count is passed to the calling block via an additional output parameter #Count_Value (data type: INT).
- The count value is displayed on the digital display of the simulator.
- Perform all program steps exclusively in the source file.
- Insert a block protection in all FBs and DBs using the keyword KNOW_HOW_PROTECT .

What to Do

1. Copy the SFB 0 from the Standard Library into your block folder.
2. Open the source file *Assembly line*.
3. In the FB2 "Transport", declare the static variable #Counter of the data type SFB 0 as well as the output parameter #Count_Value of the INT data type.
4. Insert the necessary instructions for the count function in the FB2 "Transport".
5. In FB10 insert the instructions for displaying the count value (BCD coded) on the digital display.
6. Compile the modified source file and download the new blocks to the CPU. Test the program.
7. Insert a block protection in all participating FBs and DBs.

Solution to Exercise A1.2: Counting the Completed Parts (FB1, part 1)

```
FUNCTION_BLOCK "Station"
TITLE =controlling a work station
AUTHOR : PT41
FAMILY : A2_0
NAME : ST7PRO2
VERSION : 0.0
VAR_INPUT
  Initial : BOOL ;
  Proxy_switch : BOOL ;
  Acknowledge : BOOL ;
  Clock_bit_q : BOOL ;
  Clock_bit_s : BOOL ;
END_VAR
VAR_OUTPUT
  LED : BOOL ;
  Transp_req : BOOL ;
END_VAR
VAR_IN_OUT
  Conv_busy : BOOL ;
END_VAR
VAR
  State : STRUCT
   Process_piece : BOOL ;
   Piece_finished : BOOL ;
   Place_piece_on_conv : BOOL ;
   Wait_for_piece : BOOL ;
   Take_piece_from_conv : BOOL ;
  END_STRUCT ;
FL_initial: BOOL;
END_VAR
BEGIN
NETWORK
TITLE =Initialization
//By means of input "Initial" the basic state #Process_piece is set
      A     #Initial;
      FP    #FL_initial;
      S     #State.Process_piece;
      R     #State.Piece_finished;
      R     #State.Place_piece_on_conv;
      R     #State.Wait_for_piece;
      R     #State.Take_piece_from_conv;
      R     #Conv_busy;
NETWORK
TITLE =State: Process_piece
//In this state the workpiece is processed. Processing is terminated
// when the operator acknowledges the termination of the workpiece
//by means of the button "S1"
      AN    #State.Process_piece;
      JC    Pfin;
      S     #LED;                    //LED is on permanently ;
      R     #Transp_req;
      A     #Acknowledge;            //when the operator acknowledges,
      R     #State.Process_piece;    //a change of state is performed;
      R     #LED;
      S     #State.Piece_finished;

// (Continued on the next page)
```

Solution to Exercise A1.2: Counting the Completed Parts (FB1, part 2)

```
NETWORK
TITLE =State: Piece_finished
//In the State #Piece_finished the operator waits for the permission
//to place the workpiece on the conveyor. The signal #Conv_busy indicates,
//whether the conveyor is busy or not. When the Conveyor is free, a state change
//to the state Place_piece_on_conv is performed.
Pfin: AN    #State.Piece_finished;
      JC    PpCo;
      A     #Clock_bit_s;           //slow flashing;
      =     #LED;
      AN    #Conv_busy;             //when the conveyor is free,
      S     #Conv_busy;             //it is marked busy
      R     #LED;                   //an a state change is performed;
      R     #State.Piece_finished;
      S     #State.Place_piece_on_conv;

NETWORK
TITLE =State: Place_piece_on_conv
PpCo: AN    #State.Place_piece_on_conv;
      JC    Wait;
      A     #Clock_bit_q;           //quick flashing;
      =     #LED;
      A     #Proxy_switch;          //When the piece is placed on the conveyor,
      S     #Transp_req;            //the transport is started,
      R     #LED;                   //and the LED is cleared;
      A     #Transp_req;            //When the belt is moving,
      AN    #Proxy_switch;          //an the workpiece has left the proxy switch,
      R     #State.Place_piece_on_conv; // a state change is performed;
      S     #State.Wait_for_piece;

NETWORK
TITLE =State: Wait_for_piece
//Waiting for an new raw piece. The arrival of a new piece is indicated by
//the proxy switch of the conveyor
Wait: AN    #State.Wait_for_piece;
      JC    TpCo;
      R     #LED;                   //the LED is switched off;
      A     #Proxy_switch;          //a new raw piece arrives,
      R     #Transp_req;            //the conveyor is stopped,
      R     #State.Wait_for_piece;  //and a state change is performed;
      S     #State.Take_piece_from_conv;

NETWORK
TITLE =State: Take_piece_from_conv
//In this State the new raw piece is taken from the conveyor to
//the working place
TpCo: AN    #State.Take_piece_from_conv;
      JC    END;
      A     #Clock_bit_q;           //the LED flashes quickly
      =     #LED;                   //
      AN    #Proxy_switch;          //when the raw piece is taken from the belt,
      R     #Conv_busy;             //the conveyor is set free,
      R     #LED;                   //the LED is switched off
      R     #State.Take_piece_from_conv; //and a state change is performed;
      S     #State.Process_piece;
END: BEU   ;

END_FUNCTION_BLOCK
```

Solution to Exercise A1.2: Counting the Completed Parts (FB2, part 3)

```
FUNCTION_BLOCK "Transport"
TITLE =Controlling the conveyor belt
VERSION : 0.1

VAR_INPUT
  Initial : BOOL ;
  L_Barrier : BOOL ;
  Acknowledge : BOOL ;
  Transp_req : BOOL ;
  Clock_Bit : BOOL ;
END_VAR
VAR_OUTPUT
  LED : BOOL ;
  Conv_right : BOOL ;
  Conv_left : BOOL ;
  Count_Value :INT ;
END_VAR
VAR
  State : STRUCT
   Waiting : BOOL ;
   Conv_right : BOOL ;
   Assembly : BOOL ;
   Conv_left : BOOL ;
  END_STRUCT ;
  FL_initial: BOOL;
  Count: "CTU";              // SFB 0 must be included
END_VAR
BEGIN
NETWORK
TITLE =Initialization

      A     #Initial;
      FP    #FL_initial;
      S     #State.Waiting;
      R     #State.Conv_right;
      R     #State.Assembly;
      R     #State.Conv_left;
      CALL  #Count (R:= #Initial,
                    CV := #Count_Value);
NETWORK
TITLE =State: "Waiting"
//The conveyor waits for a completed part in this state.
      AN    #State.Waiting;
      JC    RIGH;
      R     #Conv_right;
      R     #Conv_left;
      R     #LED;
      A     #Transp_req;
      R     #State.Waiting;
      S     #State.Conv_right;

// (Continued on the next page)
```

Solution to Exercise A1.2: Counting the Completed Parts (FB2, part 4)

```
NETWORK
TITLE =State: Conv_right
//This state describes the finished part's transport in the direct
//final assembly
RIGH: AN    #State.Conv_right;
      JC    FINM;
      S     #Conv_right;
      A     #Clock_Bit;
      =     #LED;
      AN    #L_Barrier;
      R     #Conv_right;
      R     #State.Conv_right;
      S     #State.Assembly;
      AN    #L_Barrier;
      =     #L_Barrier;
      CALL #Count    (CU          := #L_Barrier,
                      CV          := #Count_Value);

NETWORK
TITLE =State: Assembly
//In this state, the finished part is removed and a new raw piece is laid
//on the belt. After that, the raw piece's transport in the direction of the empty
//processing station is started with S4.
//
FINM: AN    #State.Assembly;
      JC    LEFT;
      S     #LED;
      A     #Acknowledge;
      R     #LED;
      R     #State.Assembly;
      S     #State.Conv_left;

NETWORK
TITLE =State: Conv_left
//In the state, the raw piece's transport to the station takes place, that delivered
//the finished piece.
LEFT: AN    #State.Conv_left;
      JC    ENDE;
      S     #Conv_left;
      A     #Clock_Bit;
      =     #LED;
      AN    #Transp_req;
      R     #Conv_left;
      R     #State.Conv_left;
      S     #State.Waiting;
ENDE: BEU   ;

END_FUNCTION_BLOCK
```

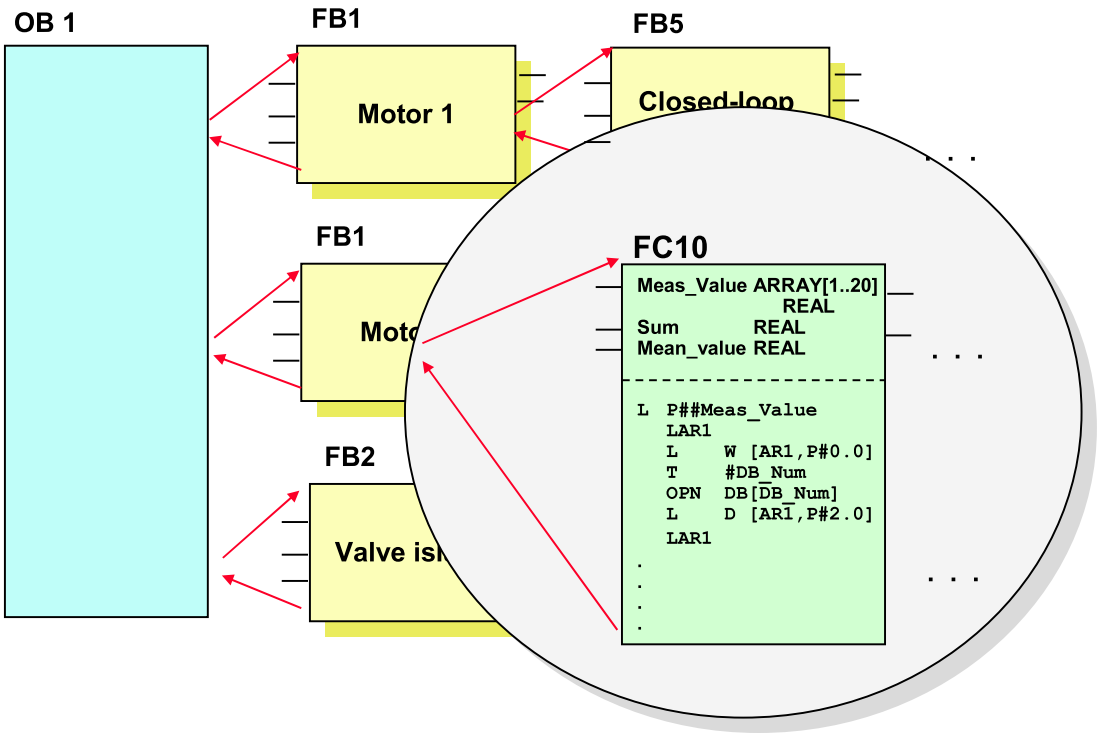
Solution to Exercise A1.2: Counting the Completed Parts (FB10, part 5)

```
FUNCTION_BLOCK FB 10
TITLE =
VERSION : 0.1

VAR
  Station_1 : "Station";
  Station_2 : "Station";
  Station_3 : "Station";
  Transport : "Transport";
  Conv_busy : BOOL ;
END_VAR
VAR_TEMP
  Trans_1 : BOOL ;
  Trans_2 : BOOL ;
  Trans_3 : BOOL ;
  Trans : BOOL ;
END_VAR
BEGIN
NETWORK
TITLE =Station_1

     CALL #Station_1 (
          Initial              := "INITIALIZATION",
          Proxy_Switch         := "INI1",
          Acknowledge          := "S1",
          Clock_Bit_q          := "CLOCK_BIT_FAST",
          Clock_Bit_s          := "CLOCK_BIT_SLOW",
          LED                  := "H1",
          Transp_req           := #Trans_1,
          Conv_busy            := #Conv_busy);

NETWORK
TITLE =Station_2

     CALL #Station_2 (
          Initial              :="INITIALIZATION",
          Proxy_Switch         := "INI2",
          Acknowledge          := "S2",
          Clock_Bit_q          := "CLOCK_BIT_FAST",
          Clock_Bit_s          := "CLOCK_BIT_SLOW",
          LED                  := "H2",
          Transp_req           := #Trans_2,
          Conv_busy            := #Conv_busy);

NETWORK
TITLE =Station_3

     CALL #Station_3 (
          Initial              := "INITIALIZATION",
          Proxy_Switch         := "INI3",
          Acknowledge          := "S3",
          Clock_Bit_q          := "CLOCK_BIT_FAST",
          Clock_Bit_s          := "CLOCK_BIT_SLOW",
          LED                  := "H3",
          Transp_req           := #Trans_3,
          Conv_busy            := #Conv_busy);
// (Continued on next page)
```

Solution to Exercise A1.2: Counting the Completed Parts (FB10, part 6)

```
NETWORK
TITLE =Logic: Transp_req
//Forming the logic for #Transp_req
     O     #Trans_1;
     O     #Trans_2;
     O     #Trans_3;
     =     #Trans;

NETWORK
TITLE =Transport

     CALL #Transport (
          Initial                  := "INITIALIZATION",
          L_Barrier                := "LB1",
          Acknowledge              := "S4",
          Transp_req               := #Trans,
          Clock_Bit                := "CLOCK_BIT_FAST",
          LED                      :=  "H4",
          Conv_right               := "K1_CONVR",
          Conv_left                := "K2_CONVL");

      L          #Transport.Count_Value ;
      ITD ;                                  // Expand to DINT
      DTB ;                                  // Convert to BCD
      T          QW12 ;

END_FUNCTION_BLOCK
DATA_BLOCK "ASSEMBLY_LINE_DB"
VERSION : 0.1
"ASSEMBLY_LINE"
BEGIN
END_DATA_BLOCK

ORGANIZATION_BLOCK OB 1
TITLE =
VERSION : 0.1

VAR_TEMP
  OB1_EV_CLASS : BYTE ;       //Bits 0-3 = 1 (Coming event), Bits 4-7 = 1 (Event class 1)
  OB1_SCAN_1 : BYTE ;         //1 (Cold restart scan 1 of OB 1), 3 (Scan 2-n of OB 1)
  OB1_PRIORITY : BYTE ;       //1 (Priority of 1 is lowest)
  OB1_OB_NUMBR : BYTE ;       //1 (Organization block 1, OB1)
  OB1_RESERVED_1 : BYTE ;     //Reserved for system
  OB1_RESERVED_2 : BYTE ;     //Reserved for system
  OB1_PREV_CYCLE : INT ;      //Cycle time of previous OB1 scan (milliseconds)
  OB1_MIN_CYCLE : INT ;       //Minimum cycle time of OB1 (milliseconds)
  OB1_MAX_CYCLE : INT ;       //Maximum cycle time of OB1 (milliseconds)
  OB1_DATE_TIME : DATE_AND_TIME ;  //Date and time OB1 started
END_VAR

BEGIN
NETWORK
TITLE =

     CALL FB    10 , DB    10 ;

END_ORGANIZATION_BLOCK
```

Appendix 2: Indirect Access to FC and FB Parameters



SIMATIC S7

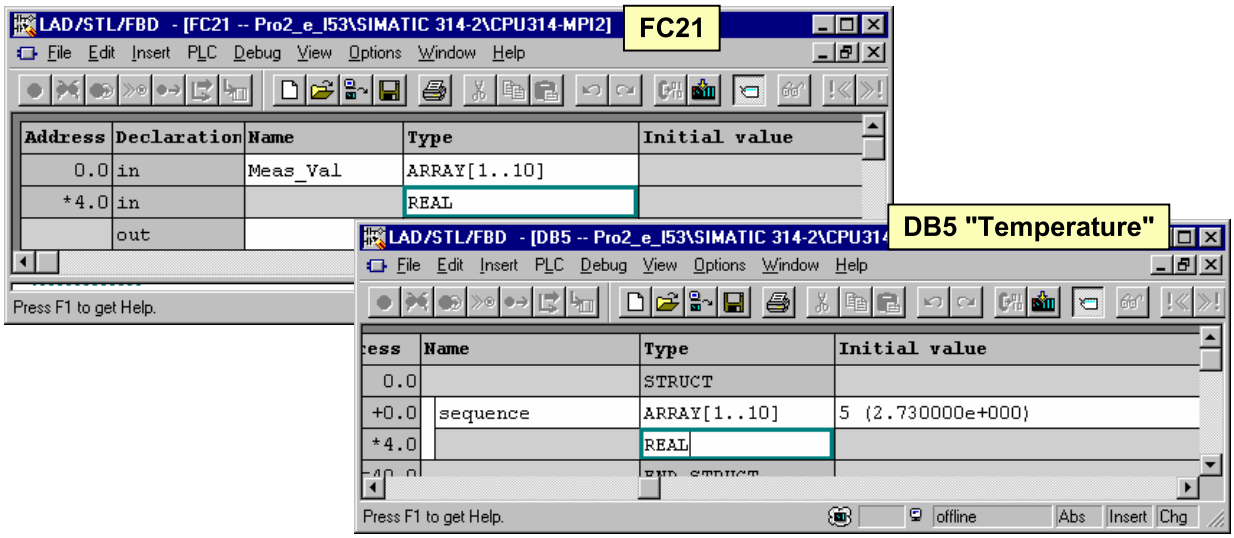
Date: 30.06.2011
File: PRO2_15E.1

SITRAIN Training for Automation and Drives

Contents

Function Call with Complex Data Types

Example: Passing an ARRAY to a Function



Assignment of the parameters is only possible symbolically

```
Network 1: Meas_Val is declared as an array in FC21

      CALL   FC     21
       Meas_Val:="Temperature".sequence
```


Date: 30.06.2011
File: PRO2_15E.2

Overview

Parameters of the complex data type (ARRAY and STRUCT) offer a clear and efficient way for transferring larger amounts of related data between the calling and the called block and thus accommodating the concept of "Structured Programming".

An array or a structure can be passed to a called function as a complete variable.

Assigning the Parameters

For the pass, a parameter of the same data type as the actual parameter to be passed must be declared in the called function. Such a parameter (data type: ARRAY, STRUCT, DATE_AND_TIME and STRING) can only be assigned symbolically.

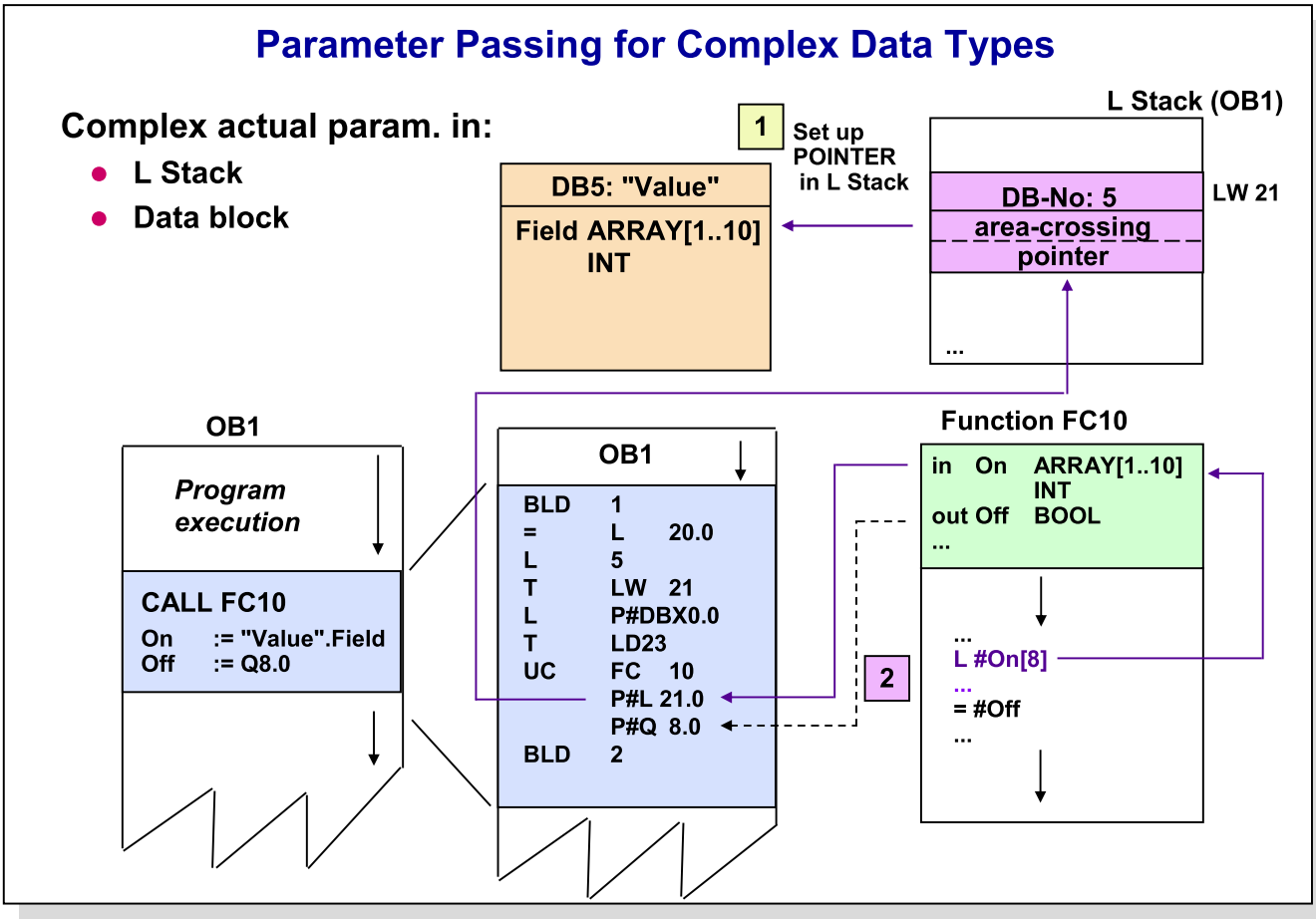
Since variables of the complex data type can only be set up in data blocks or in local data stacks, the actual parameter must either be located in a data block (global or instance DB) or in the local data stack of the calling block.

After the STL/LAD/FBD Editor has checked the compatibility of the data types of actual parameter and block parameter in the parameter passing to an FC, only a POINTER with the DB number and area-crossing pointer to the actual parameter is passed to the called FC.

This POINTER is set up in the L Stack of the calling block (V area) via the CALL macro. This POINTER is then of great importance for the programmer in particular, when the passed parameter has to be accessed indirectly (see appendix).

Notes

- The number of occupied local data can be determined by selecting the menu options *View -> Block Properties*.
- Components of ARRAYs or STRUCTs can also be passed to a block parameter if the block parameter and the ARRAY or STRUCT components are of the same data type.



Parameter Passing

With complex data types (DT, STRING, ARRAY, STRUCT and UDT) the actual parameters lie either in a data block or in the L-Stack of the calling block (V-area).

Since a 32-bit area-crossing pointer cannot reach an actual parameter in a DB, the STL/LAD/FBD Editor stores a 48-bit "POINTER", that points to the actual parameter, in the L-Stack of the called block.

During a call, a 32-bit area-crossing pointer is passed to this "POINTER". Within the FC a parameter access of the actual parameters then takes place by means of double branching.

The setting up of the "POINTER" in the L-Stack of the calling block takes place before the actual switch to the called FC.

Consequences

Parameters of the complex data type are “more easy going” than elementary parameter types. Input parameters of the complex data type can be written within the called FC without second thoughts.

Just the same, output parameters can be scanned without second thoughts.

Indirect Access to Complex Data Types

Address	Declaration	Name	Type	Start value	Comment
0.0	in	Meas_Val	ARRAY[1..8]		
*4.0			REAL		
32.0	out	Sum	REAL		
36.0	out	Mean_Val	REAL		
	in_out				
0.0	temp	DB_Num	WORD		

```
Network 1: Determining the DB-No. and the start address

      L     P## Meas_Val            // Load address of POINTER into ACCU1
      LAR1                          // and from there load into AR1;
      L     W [AR1,P#0.0]           // Determine DB number
      T     #DB_Num                 // and load into temp. variable;
      OPN   DB[DB_Num]              // Open DB
      L     D [AR1,P#2.0]           // Determine area pointer
      LAR1                          // and load into AR1;

Network 2: Calculation of the sum

      L     0.000000e+000           // 0 to ACCU1 (sum =0.0)
      L     8                       // Counter to ACCU1; Sum=0 to ACCU2
BEGN: TAK                           // Sum to ACCU1, Counter to ACCU2
      ENT                           // Counter to ACCU3
      L     D[AR1,P#0.0]            // Field components into ACCU1
      +R                            // Sum into ACCU1, counter to ACCU2
      +AR1  P#4.0;                  // Increment AR1 by 4 bytes
      TAK                           // Loop counter into ACCU1, sum into ACCU2
      LOOP  BEGN;                   // Decrement loop counter and jump if necessary
      T     #Sum                    // Transfer sum to #Sum
```


Indirect access

When passing complex data types such as ARRAYs and STRUCTs, the full power can only be exploited if the parameter passing is combined with the indirect access within the called block.

An indirect access to the passed actual parameters of the complex data type is made in two steps:

1. First, an area-crossing pointer to the POINTER that has been passed in the local data stack of the caller is determined by means of the operation:
   L P##Meas_Val.
2. For the actual access to the actual parameters, it is then necessary to evaluate the information in the POINTER, which references the actual current operands, such as:

```
L  P##Meas_Val     // returns area-crossing pointer to POINTER
LAR1               // loads area-crossing pointer into address register
L  W[AR1,P#0.0]    // returns DB number of the actual parameter, if it is
                   // stored in a DB, otherwise 0
L  D[AR1,P#2.0]    // returns area-crossing pointers to the actual
parameter
```

The result is then calculated in the usual way.

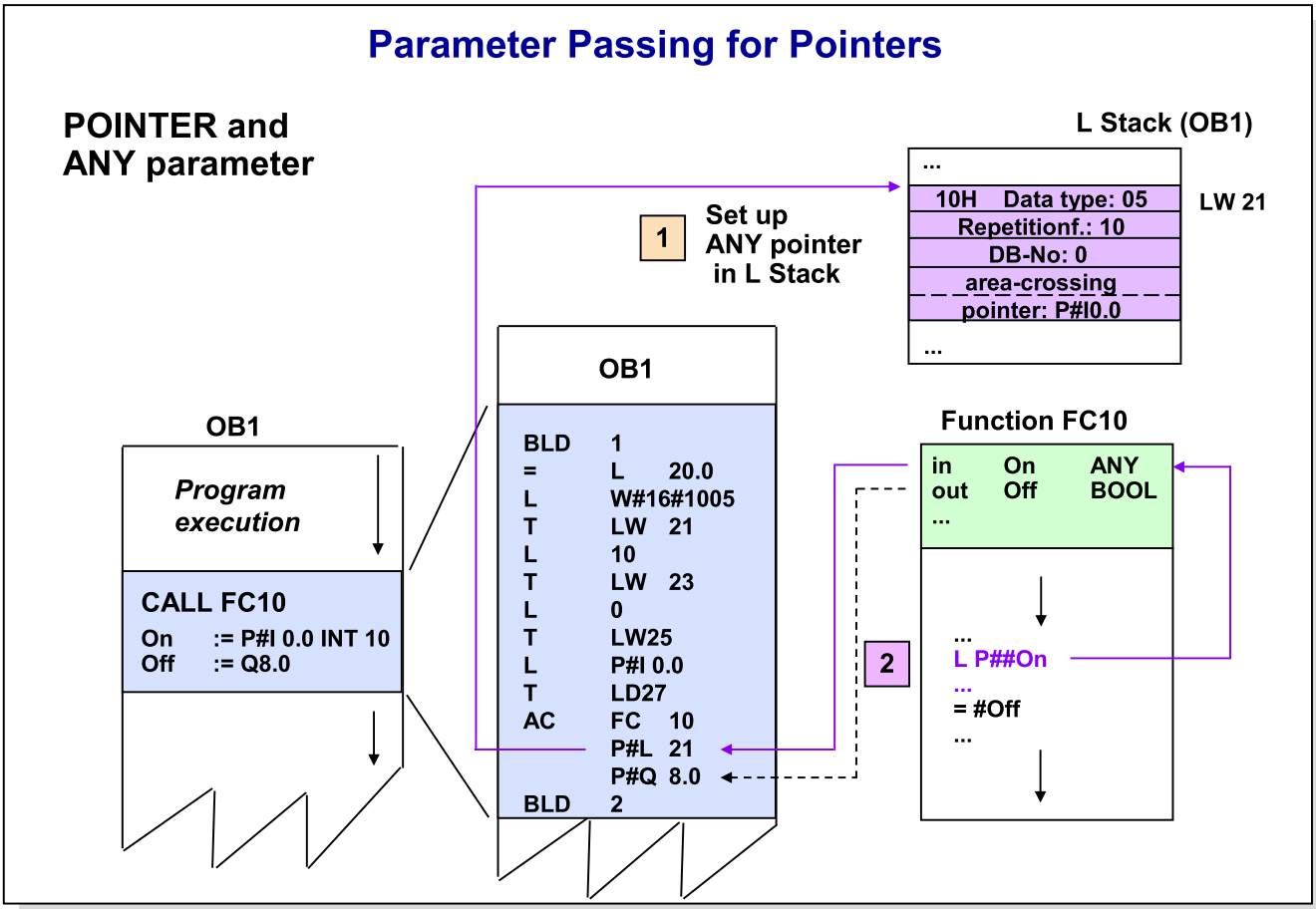
Note

To obtain access in the above example, the programmer must set the contents of the DB register and of the AR1 register in such a way that the first component of the transferred field is addressed.

The statement

```
L  Meas_Val[0]
```

also ensures that the required data block is opened by means of the DB register and the AR1 register is set to the start of the transferred ARRAY.



Parameter Passing

If a parameter of the "POINTER" or "ANY" data type is passed to an FC, then the STL/LAD/FBD Editor sets up the corresponding data structure in the L-Stack of the calling block.

With the FC call, a 32-bit area-crossing pointer that points to this data structure ("POINTER" or "ANY") is then passed to the FC that is called.

Within the called FC it is not possible to access parameters directly because of missing type information that are referenced through this "POINTER" or "ANY" pointer.

The evaluation of the contents of "POINTER"- or- "ANY" must be carried out by the user within the called FC.

The setting up of the "POINTER" or "ANY" structure on the L-Stack of the calling block takes place before the actual switch into the called FC.

Exception

An exception of the above rule is the STL/LAD/FBD Editor, when at a block parameter of the "ANY" data type an additional actual parameter of the "ANY " data type is set up that is saved in the L-Stack of the calling block.

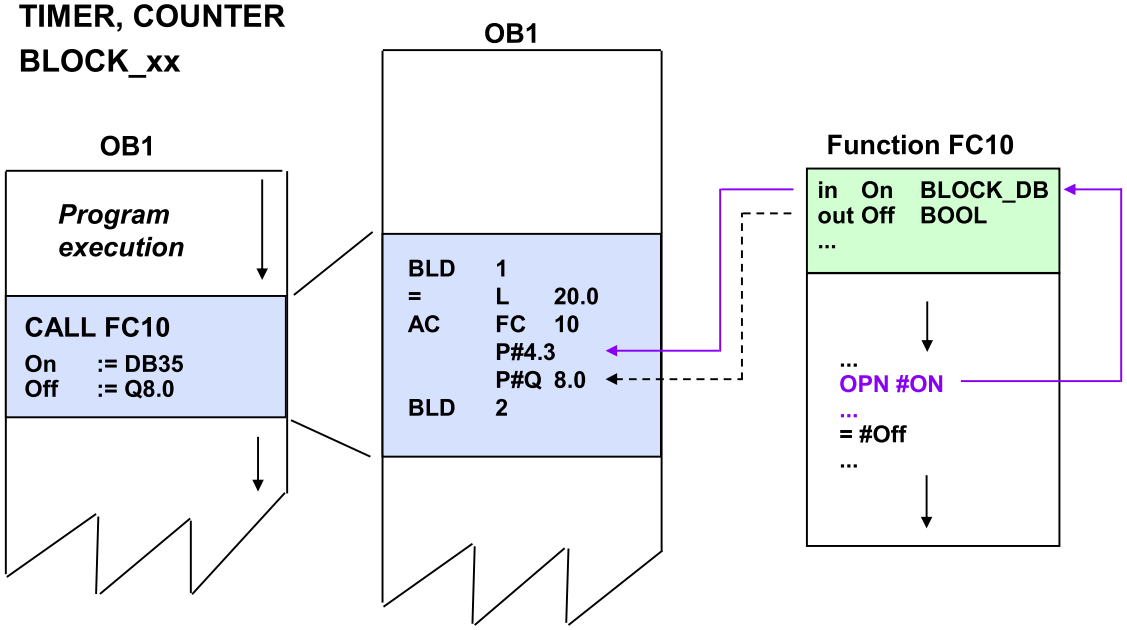
In this case the STL/LAD/FBD Editor does not set up an additional "ANY" pointer on the L-Stack of the caller, but passes directly at the FC call a 32-bit area-crossing pointer to the already existing "ANY" pointer (in the L-Stack of the caller).

Thus, at runtime, this "ANY" pointer can be manipulated by the calling block and so implement an "indirect" assignment of the FC with actual parameters.

Parameter Passing for Parameter Types

Block parameter:

- TIMER, COUNTER
- BLOCK_xx



SIMATIC S7

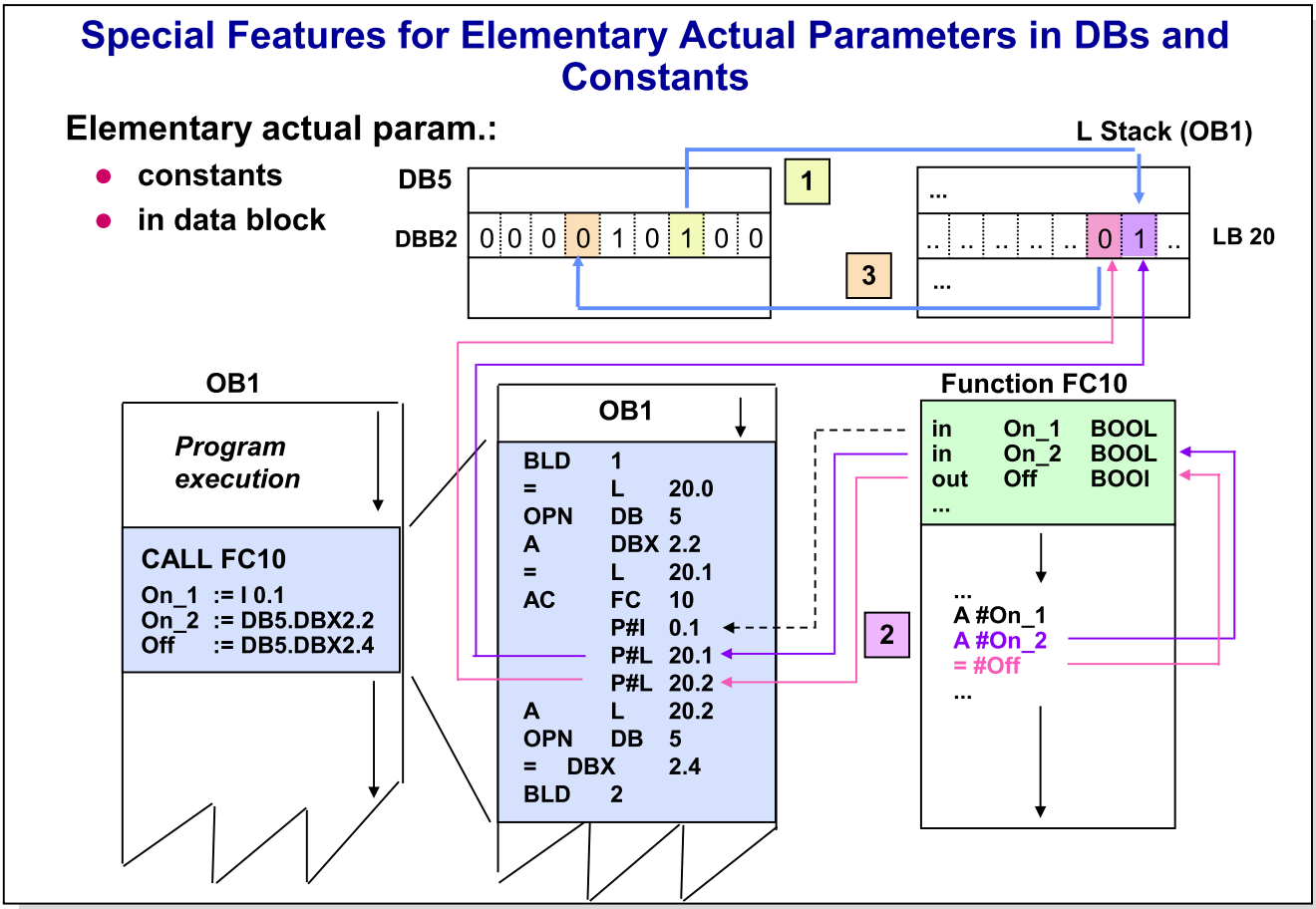
Date: 30.06.2011
File: PRO2_15E.6

SITRAIN Training for Automation and Drives

Parameter Passing

The passing of parameters of the type: TIMER, COUNTER and BLOCK_x is the easiest. In this case, instead of a 32-bit area-crossing pointer, the number of the current TIMER or COUNTER or BLOCK_xs is simply passed to the called FC.

Special Features for Elementary Actual Parameters in DBs and Constants



Parameter Passing

If an input, output, or in/out parameter of an FB is assigned with a constant or with a parameter that is stored in a DB, then the STL/LAD/FBD Editor first of all reserves the necessary memory on the callers L-Stack and then copies - with the input and in/out parameter - the value of the actual parameters on the L-Stack.

For the output parameter, a reservation of memory area on the L-Stack takes place but no initialization.

Only after that, does the actual switch into the called FC takes place, whereby the STL/LAD/FBD Editor passes in each case an area-crossing pointer to the L-Stack memory areas on the called FC.

After a jump back into the calling block, the results - with output and in/out parameter - will be copied back into the actual parameters.

Consequences

This mechanism shows that within a called FC, input parameters can only be scanned and output parameters can only be written to.

If an input parameter is written to, the corresponding value is stored in the L-Stack. However, it is not copied into the actual parameters after the processing of the FC.

In the same way, output parameters can only be written and not read. With the scanning of an output parameter an undefined value is read from the L-Stack because of missing initialization.

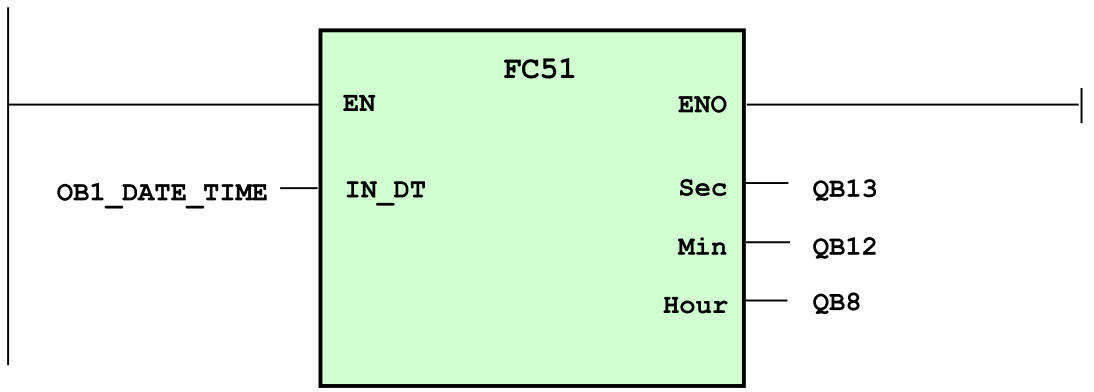
In/out parameters cause the least problems. They are assigned with the values of the actual parameters before the FC call as well as removed after the call.

Notes

Output parameters must be written in the called FC (avoid instructions such as "S" and "R"), otherwise an undefined value from the L-Stack will be copied back to the actual parameter.

If you cannot ensure that output parameters will be written then you have to use in/out parameters instead.

Exercise A2.1: Evaluate Date-and-Time Parameter in an FC



SIMATIC S7

Date: 30.06.2011
File: PRO2_15E.8

SITRAIN Training for Automation and Drives

Overview

The following exercise should show you how you can indirectly access input, output and in/out parameters of the complex data type within a function.

You must use the same technique if you have to indirectly access other complex data types, such as ARRAYs, STRUCTs or STRINGs.

Task

Create an FC51 with the following properties:

- The FC51 has an input parameter #IN_DT of data type: DATE_AND_TIME
- In its 3 output parameters #Sec, #Min and #Hour, the FC51 returns the seconds, minutes and hour components of the DT-parameter passed to it.

Procedure

1. Create an FC51 with the required properties.
2. Call the FC51 in OB1. Supply the block parameter #IN_DT with the variable OB1_DATE_TIME from the start information of OB1.
3. Download the blocks into the CPU and test the program.

FB Call with Complex Data Types

Example: Passing an ARRAY to a Function Block

FB17

FB17 -- test\SIMATIC 412-MPI2\CPU 413-2 DP

Address	Decl.	Name	Type	Initial Value	Comment
0.0	in	Meas_1	ARRAY[1..10]		
*4.0	in		REAL		
40.0	out	Sum_1	REAL	0.000000e+000	
44.0	out	Sum_2	REAL		
48.0	in_out	Meas_2	ARRAY		
*4.0	in_out		REAL		
54.0	stat	DB_Num	INT		
	temp				

DB2 "Temperature"

DB2 -- test\SIMATIC 412-MPI2\CPU 413-

Address	Name	Type	Initial
0.0		STRUCT	
+0.0	Cylinder	ARRAY[1..10]	
*4.0		REAL	
+40.0	Shaft	ARRAY[1..15]	
*4.0		REAL	
=100.0		END_STRUCT	

Assigning complex parameters is only possible symbolically

```
Network 1:

  CALL   FB   17, DB 30
   Meas_1     :="Temperature".Cylinder
   Sum_1      :=MD20
   Sum_2      :=MD30
   Meas_2     :="Temperature".Shaft
```


Complex Data Types

Just as with the functions, addresses of the complex data type (ARRAY, STRUCT, DATE_AND_TIME, and STRING) can be passed completely to a called function block.

For the pass, a parameter of the same data type as the actual parameter to be passed must be declared in the called function block.

The assignment of such a parameter is only possible symbolically.

Input and Output Parameters

For input and output parameters of the complex data type, corresponding areas for the values of the actual parameters are created in the instance DB. When the FB is called, the actual parameters of the input parameter are copied into the instance DB using SFC 20 (BLKMOV) ("passing by value"), before the actual switch to the instruction section of the FB.

In the same manner, the values of the output parameter are copied from the instance DB back into the actual parameter after the FB has been processed.

As a result, a not insignificant amount of copying (processing time) can occur in the assignment of input and output parameters. This amount of copying is bypassed with in/out parameters.

In/out Parameters

No "passing by value" occurs with in/out parameters of the complex data type. Six bytes are merely reserved for every in/out parameter in the instance data area. A POINTER to the actual parameters is entered in these bytes ("passing by reference").

Notes

Input and output parameters of the complex data type can be initialized in the declaration table of an FB, but not in/out parameters.

Input and output parameters of the complex data type do not have to be assigned in an FB call. In/out parameters have to be assigned.

The memory or register-indirect access to input/output parameters or in/out parameters of the complex data type is different to that of elementary parameters.

Indirect Access to Input/Output Parameters

Address	Declaration	Name	Type	Start value	Comment
0.0	in	Meas_1	ARRAY[1..10]		
*4.0			REAL		
40.0	out	Sum_1	REAL	0.000000e+000	
44.0	out	Sum_2	REAL	0.000000e+000	
48.0	in_out	Meas_2	ARRAY[1..15]		
*4.0	in_out		REAL		
54.0	stat	DB_Num	INT	0	

```
Network 1: Determining the start address of Meas_1

      LAR1  P##Meas_1        // Load area-crossing pointer to parameter without
                             // address offset (multi-instances) in AR1
      TAR2                   // Load address offset in ACCU1
      +AR1                   // Add address offset to AR1;
                             // AR1 now points to parameter in the instance-DB
                             // Instance-DB is already opened up

Network 2: Access to Meas_1

      L     0.000000e+000    // 0 to ACCU1 (Summe =0.0)
      L     10               // Counter to ACCU1; Sum=0 to ACCU2
BEGN: TAK                    // Sum in ACCU1, counter in ACCU2
      ENT                    // Counter to ACCU3
      L     D[AR1,P#0.0]     // Field component in ACCU1
      +R                     // Sum in ACCU1, counter to ACCU2
      +AR1  P#4.0;           // Increment AR1 by 4 bytes
      TAK                    // Loop counter in ACCU1, sum in ACCU2
      LOOP  BEGN;            // Decrement loop counter and jump if necessary
      T     #Sum_1           // Transfer sum to #Sum_1
```


Indirect Access

When passing complex data types such as ARRAYs and STRUCTs, the full power can only be exploited if the parameter passing is combined with the indirect access within the called block.

An indirect access to the passed input or output parameters of the complex data type is made in two steps:

1. First, by means of the instruction:

   LAR1 P##Meas_1 // delivers area-crossing pointer to parameter
   // without address offset

   an area-crossing pointer to the parameters in the instance-DB is loaded into AR1.

   The pointer that is determined in this way contains the area identifier DI and the same byte.bit address that is also displayed by the Editor in the parameter declaration in the first column of the declaration table.

   In the case of a multiple instance, this does not correspond to the actual address of the input/output parameter in the instance DB. The address offset from AR2 that identifies the start of the instance data area in the multiple instance case has to be added to the pointer in AR1.

   TAR2 // Load address offset in ACCU1
   +AR1 // Add address offset to AR1;

2. After this, the actual access to the input/output parameter can take place. The instance DB does not have to be opened separately as it has already been opened by the CALL macro in the FB call.

   L D[AR1,P#0.0] // Load 1st component of Meas_1 etc.

Note

Indirect access to input, output and in/out parameters of the elementary data type or to static variables is made in the same way since the values of the addresses are also stored in the instance DB in this case.

Indirect Access to In/Out Parameters

Address	Declaration	Name	Type	Start value	Comment
0.0	in	Meas_1	ARRAY[1..10]		
*4.0			REAL		
40.0	out	Sum_1	REAL	0.000000e+000	
44.0	out	Sum_2	REAL	0.000000e+000	
48.0	in_out	Meas_2	ARRAY[1..15]		
*4.0	in_out		REAL		
54.0	stat	DB_Num	INT	0	

```
Network 3: Determining the start address of Meas_2

      LAR1  P##Messung_2     // Load area-crossing pointer to POINTER without
      TAR2                   // Load address offset in ACCU1, add to AR1;
      +AR1                   // AR1 now points to POINTER in the instance DB
      L      W [AR1,P#0.0]   // Load DB-number from POINTER into ACCU1
      T      #DB_Num         // Transfer DB-number (or 0) into stat. variable
      OPN    DB [#DB_Num]    // Open DB
      L      D [AR1,P#2.0]   // Load area-crossing pointer to parameter
      LAR1                   // Load pointer in AR1, AR1 points to parameter


Network 4: Access to Meas_2

      L     0.000000e+000    // 0 to ACCU1 (Sum =0.0)
      L     15               // Counter to ACCU1; Sum=0 to ACCU2
BEGN: TAK                    // Sum in ACCU1, counter in ACCU2
      ENT                    // Counter to ACCU3
      L     D[AR1,P#0.0]     // Field components in ACCU1
      +R                     // Sum in ACCU1, counter to ACCU2
      ...                    // ...
```


Indirect Access

The indirect access to in/out parameters of the complex data type differs in structure when compared to the access to input and output parameters.

With in/out parameters of the complex data type, it is not the value that is copied, but only a POINTER to the in/out parameter in the instance DB.

The actual access takes place in three steps:

1. First, by means of the instruction:

   LAR1 P## Meas_2 // delivers area-crossing pointer to POINTER

   an area-crossing pointer to the POINTER that is passed is loaded into the AR1 register. As in the previous case, the byte.bit address of the pointer in AR1 does not identify the actual address of the POINTER in the instance DB.

   In the case of a multiple instance, the offset from AR2 must still be added to the pointer in the AR1 Register:

   ```
   TAR2                 // Load address offset in ACCU1, add to AR1;
   +AR1                 // AR1 now points to POINTER in the instance DB;
   ```

2. In the next step, the information in the POINTER is evaluated; if necessary, the DB in which the actual parameter is located is opened and an area-crossing pointer to the actual address is loaded into the AR1 register:

   ```
   L     W [AR1,P#0.0]  // Load DB number from POINTER into ACCU1
   T     #DB_Num        // Transfer DB number (or 0) into variable
   OPN   DB [#DB_Num]   // Open DB
   L     D [AR1,P#2.0]  // Load area-crossing pointer to parameter
   LAR1                 // Load pointer in AR1, AR1 points to parameter
   ```

3. After this, the actual access to the actual parameter can take place:

   ```
   L   D[AR1,P#0.0]     // Load 1st component of Meas_2 etc.
   ```

"Passing On" Parameters

Nesting depth:

- S7-300: max. 8 S7-400: max. 16

Passing on depends on block, data and parameter type:

Call	FC calls FC	FB calls FC	FC calls FB	FB calls FB
Data type	E C P	E C P	E C P	E C P
Input -> Input	x - -	x x -	x - x	x x x
Output -> Output	x - -	x x -	x - -	x x -
In/Out -> Input	x - -	x - -	x - -	x - -
In/Out -> Output	x - -	x - -	x - -	x - -
In/Out -> In/Out	x - -	x - -	x - -	x - -

E: Elementary data type
C: Complex data type
P: Parameter type (Timer, Counter, Block_x)

Overview

The "passing on" of block parameters is a special form of access or assignment of block parameters. "Passing on" means that the formal parameter of the calling block becomes the actual parameter of the called block.

Restrictions with Reference to Parameter Type

As a general rule, the actual parameter must be of the same data type as the formal parameter. Furthermore, input parameters of the calling block can only be set up at an input parameter of the called block and output parameters only at output parameters.

An in/out parameter of the calling block can in principle be set up at input, output, and in/out parameters of the called block.

Restrictions with Reference to Data Type

With reference to the data types, there are restrictions dependent on the different storage of block parameters in FC or FB calls.
Block parameters of the elementary data type can be passed on without restrictions. Complex data types at input and output parameters can only then be passed on if the calling block is an FB.

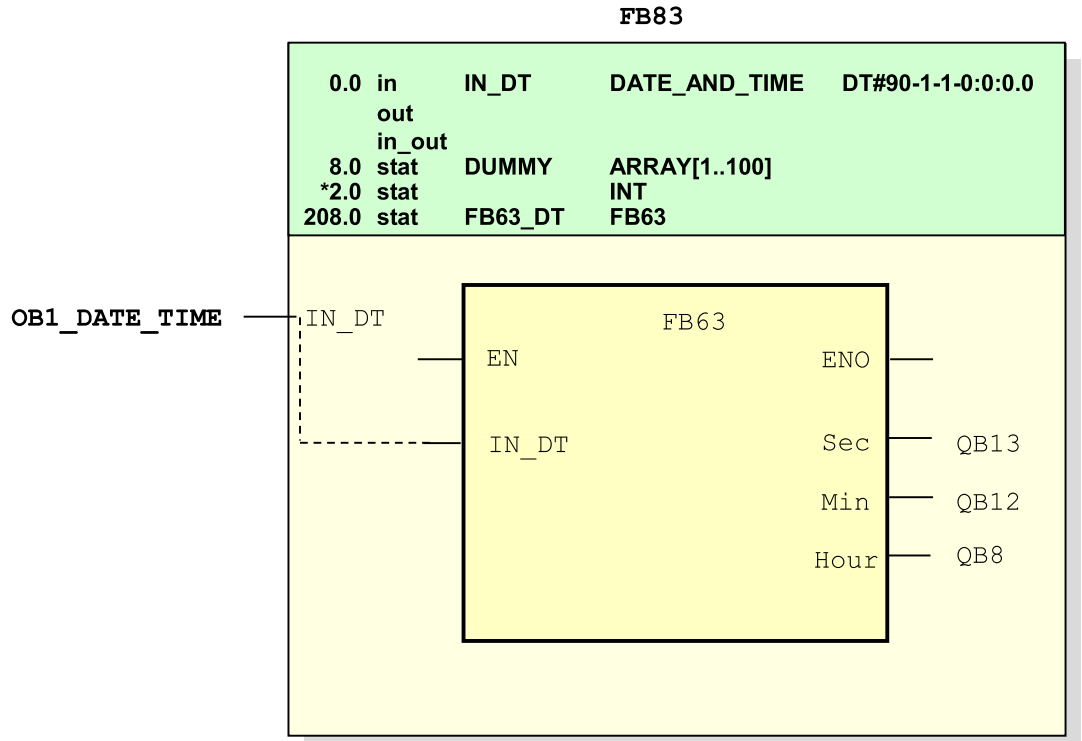
Block parameters with the parameter types: TIMER, COUNTER and BLOCK_x can then only be passed on from an input parameter to an input parameter, if the called block is an FB.

Note

The "passing on" of the parameter types: TIMER, COUNTER and BLOCK_x can be implemented in FCs with the help of indirect addressing. The number of the desired TIMER, COUNTER or BLOCK_x is passed on as a parameter of the WORD data type from the calling block to the called block, for example.

Within the last called block, this parameter can then be transferred into a temporary variable and the respective TIMER, COUNTER or BLOCK_x can then be processed via this variable.

Exercise A2.2: Evaluate Date-and-Time Parameters in an FB



Overview

The following exercise should show you how you can indirectly access an input or output parameter of the complex data type within a multiple instance-capable function block.

You must use the same technique if you have to indirectly access other complex data types, such as ARRAYs, STRUCTs or STRINGs.

Task

Create an FB63 with the following properties:

- The FB63 has an input parameter *#IN_DT* of data type: DATE_AND_TIME
- In its 3 output parameters #Sec, #Min and #Hour, the FB63 returns the seconds, minutes and hour components of the DT-parameter passed to it.

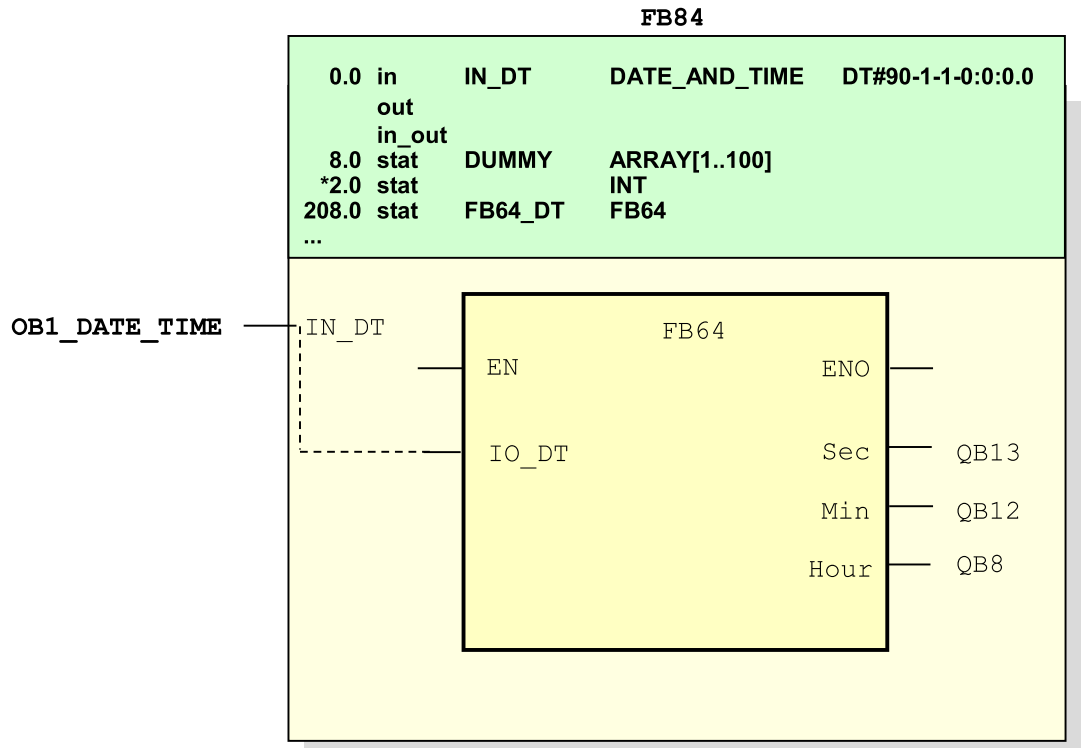
Procedure

1. Create an FB63 with the required properties.
2. To test whether the FB63 generated is also really capable of multiple instances call, an instance of the FB63 in a higher level FB

   Create a higher-level FB83. First, declare an input parameter *#IN_DT* of type DT in the FB83. Then declare a static variable *#DUMMY* of type ARRAY[1..100] OF INT and an instance of the FB63 with the name *#FB63_DT*.
3. Call the instance *#FB63_DT* within the FB83 and supply the input parameter *#IN_DT* of this instance with the input parameter *#IN_DT* of the FB83.

   Supply the output parameter of the instance FB63_DT with the output bytes QB8, QB12 and QB13.
4. Call the FB83 with instance DB83 in OB1. Supply the input parameter *#IN_DT* with the variable OB1_DATE_TIME from the start information of OB1.
5. Download the blocks into the CPU and test your program.

Exercise A2.3: Evaluate In/Out Parameters in an FB



SIMATIC S7

Date: 30.06.2011
File: PRO2_15E.14

SITRAIN Training for Automation and Drives

Overview

The following exercise should show you how you can indirectly access an in/out parameters of the complex data type within a multiple instance-capable function block.

You must use the same technique if you have to indirectly access other complex data types, such as ARRAYs, STRUCTs or STRINGs.

Task definition

Create an FB64 with the following properties:

- The FB64 has an in/out parameter *#IO_DT* of data type: DT
- In its 3 output parameters #Sec, #Min and #Hour, the FB64 returns the seconds, minutes and hour components of the DT-parameter passed to it.

Execution

1. Create an FB64 with the required properties.
2. Create an FB84. First, declare an input parameter *#IN_DT* of type DT in the FB84. Then declare a static variable *#DUMMY* of type ARRAY[1..100] OF INT and an instance of the FB64 with the name *#FB64_DT*
3. Call the instance *#FB64_DT* within the FB84 and supply the in/out parameter *#IO_DT* of this instance with the value of the input parameter *#IN_DT* of the FB84. Note that direct passing of an input parameter to an in/out parameter is not permitted. Which solution is recommended?

   Supply the output parameter of the instance FB64_DT with the output bytes QB8, QB12 and QB13, as in the previous task.
4. Call the FB84 with instance DB84 in OB1. Supply the input parameter *#IN_DT* of the FB84 with the variable OB1_DATE_TIME from the start information of OB1.
5. Download the blocks into the CPU and test the program.

Solution to Exercise A2.1: Access to DT Parameters in an FC

```
FUNCTION FC 51 : VOID
TITLE =
VERSION : 0.1
VAR_INPUT
  IN_DT : DATE_AND_TIME ;
END_VAR
VAR_OUTPUT
  Sec : BYTE ;
  Min : BYTE ;
  Hour : BYTE ;
END_VAR
VAR_TEMP
  DB_Num : WORD ;
END_VAR
BEGIN
NETWORK
TITLE =
// In the case of an input, output, or in/out parameter of the complex data type, an area-
// crossing pointer to a POINTER variable (6 bytes), that is set up in the L-stack of the caller
// by the CALL macro, is transferred to the called function for the parameter passing. The
// content of the POINTER variable points to the actual operands. For indirect access, an
// area-crossing pointer to this POINTER is created first.
// In the next stage, the content of the POINTER variable is read out and access made to
// the actual operands via this information.
//
     L     P##IN_DT;           // Load area-crossing pointer to POINTER in ACCU1
     LAR1  ;                   // Load pointer in AR1, AR1 now points to the POINTER
     L     W [AR1,P#0.0];      // Load DB-number from POINTER in ACCU1
     T     #DB_Num;            // Transfer DB-number (or 0) into temp. variable
     OPN   DB [#DB_Num];       // Open DB
     L     D [AR1,P#2.0];      // Load area-crossing pointer to DT variable from
                               // POINTER
     LAR1  ;                   // Load pointer in AR1, AR1 now points to DT variable
     L     B [AR1,P#3.0];      // Load hours component from DT variable
     T     #Hour;              // Transfer into output parameter #Hour
     L     B [AR1,P#4.0];      // Load minutes component from DT variable
     T     #Min;               // Transfer into output parameter #Min
     L     B [AR1,P#5.0];      // Load seconds component from DT variable
     T     #Sec;               // Transfer into output parameter #Sec
     SET   ;
     SAVE  ;                   // Set BR-bit to 1
END_FUNCTION

ORGANIZATION_BLOCK OB1
TITLE =
VERSION : 0.1
VAR_TEMP
  OB1_TEMP: ARRAY[1..20] OF BYTE;     //Start information of the OB1
BEGIN
NETWORK
TITLE =
     CALL FC    51 (
          IN_DT   := #OB1_DATE_TIME,
          Sec     := QB    13,
          Min     := QB    12,
          Hour    := QB     8);
END_ORGANIZATION_BLOCK
```

Solution to Exercise A2.2: Access to DT Parameters in FBs

```
FUNCTION_BLOCK FB 63
TITLE =
VERSION : 0.1
VAR_INPUT
  IN_DT : DATE_AND_TIME ;
END_VAR
VAR_OUTPUT
  Sec : BYTE ;
  Min : BYTE ;
  Hour : BYTE ;
END_VAR
BEGIN
NETWORK
TITLE =
// In the case of an input, output parameter of the complex data type, the value
// of the complex variable is copied or copied back into the instance DB.
// For indirect access, an area-crossing pointer including the address offset AR2,
// which occurs in the case of a multi-instance, is created first.
//
     LAR1  P##IN_DT;       // Load area-crossing pointer to #IN_DT without the
                           // additional address offset in the AR2
     TAR2  ;               // Transfer the address offset into ACCU1 (address offset in AR2
                           // is made by CALL-macro
     +AR1  ;               // Load content of ACCU1 (address offset AR2) to AR1, AR1now
                           // points to #IN_DT
     L     B [AR1,P#3.0];  // Load hours component from DT variable
     T     #Hour;          // Transfer into output parameter #Hour
     L     B [AR1,P#4.0];  // Load minutes component from DT variable
     T     #Min;           // Transfer into output parameter #Min
     L     B [AR1,P#5.0];  // Load seconds component from DT variable
     T     #Sec;           // Transfer into output parameter #Sec
     SET   ;
     SAVE  ; // Set BR-bit to 1
END_FUNCTION_BLOCK

FUNCTION_BLOCK FB 83
TITLE =
VERSION : 0.1

VAR_INPUT
  IN_DT : DATE_AND_TIME ;
END_VAR
VAR
  DUMMY : ARRAY  [1 .. 100 ] OF INT ;
  FB63_DT : FB 63;
END_VAR
BEGIN
NETWORK
TITLE =

     CALL #FB63_DT (
          IN_DT              := #IN_DT,
          Sec                := QB    13,
          Min                := QB    12,
          Hour               := QB     8);

END_FUNCTION_BLOCK
```

Solution to Exercise A2.3: Access to DT Parameters in FBs (1)

```
FUNCTION_BLOCK FB 64
TITLE =
VERSION : 0.1
VAR_OUTPUT
  Sec : BYTE ;
  Min : BYTE ;
  Hour : BYTE ;
END_VAR
VAR_IN_OUT
  IO_DT : DATE_AND_TIME ;
END_VAR
VAR_TEMP
  DB_Num : WORD ;
END_VAR
BEGIN
NETWORK
TITLE =

// In the case of an in/out parameter of the complex data type, the value of the complex
// variable is not copied into the instance DB, but instead a POINTER to the actual
// operands is stored in the instance DB. For indirect access, an area-crossing pointer
// to this POINTER, including the address offset AR2, which occurs in the case of a
// multi-instance, is created first.
// After this, access is made to the actual operands in the usual way.

     LAR1  P##IO_DT;        // Load area-crossing pointer to the POINTER
                            // without address offset
     TAR2  ;                // Transfer the address offset from AR2 into ACCU1
     +AR1  ;                // Add address offset to AR1, AR1 now points to the
                            // POINTER
     L     W [AR1,P#0.0];   // Load DB-number from POINTER into ACCU1
     T     #DB_Num;         // Transfer DB-number (or 0) in temp. variable
     OPN   DB [#DB_Num];    // Open DB
     L     D [AR1,P#2.0];   // Load area-crossing pointer to the DT variable from POINTER
     LAR1  ;                // Load pointer into AR1, AR1 points to DT variable
     T     #Hour;           // Transfer into output parameter #Hour
     L     B [AR1,P#4.0];   // Load minutes component from DT variable
     T     #Min;            // Transfer into output parameter #Min
     L     B [AR1,P#5.0];   // Load seconds component from DT variable
     T     #Sec;            // Transfer into output parameter #Sec
     SET   ;
     SAVE  ;                // Set BR-bit to 1


END_FUNCTION_BLOCK
```

Solution to Exercise A2.3: Access to DT Parameters in FBs (2)

```
FUNCTION_BLOCK FB 84
TITLE =
VERSION : 0.1
VAR_INPUT
  IN_DT : DATE_AND_TIME ;
END_VAR
VAR
  DUMMY : ARRAY  [1 .. 100 ] OF INT ;
  FB64_DT : FB 64;
END_VAR
VAR_TEMP
  DT_TEMP : DATE_AND_TIME ;
  I_Ret_VAL : INT ;
END_VAR
BEGIN
NETWORK
TITLE =

     CALL SFC 20 (
          SRCBLK       := #IN_DT,
          RET_VAL      := #I_Ret_VAL,
          DSTBLK       := #DT_TEMP);


     CALL #FB64_DT (
          Sec          := AB    13,
          Min          := AB    12,
          Hour         := AB     8,
          IO_DT         := #DT_TEMP);

END_FUNCTION_BLOCK

ORGANIZATION_BLOCK OB1
TITLE =
VERSION : 0.1
VAR_TEMP
  OB1_EV_CLASS : BYTE ;       //Bits 0-3 = 1 (Coming event), Bits 4-7 = 1 (Event class 1)
  OB1_SCAN_1 : BYTE ;         //1 (Cold restart scan 1 of OB 1), 3 (Scan 2-n of OB 1)
  OB1_PRIORITY : BYTE ;       //1 (Priority of 1 is lowest)
  OB1_OB_NUMBR : BYTE ;       //1 (Organization block 1, OB1)
  OB1_RESERVED_1 : BYTE ;     //Reserved for system
  OB1_RESERVED_2 : BYTE ;     //Reserved for system
  OB1_PREV_CYCLE : INT ;      //Cycle time of previous OB1 scan (milliseconds)
  OB1_MIN_CYCLE : INT ;       //Minimum cycle time of OB1 (milliseconds)
  OB1_MAX_CYCLE : INT ;       //Maximum cycle time of OB1 (milliseconds)
  OB1_DATE_TIME : DATE_AND_TIME ;  //Date and time OB1 started
END_VAR
BEGIN
NETWORK
TITLE =

     CALL FB    84 , DB    84 (
          IN_DT    := #OB1_DATE_TIME);

END_ORGANIZATION_BLOCK
```