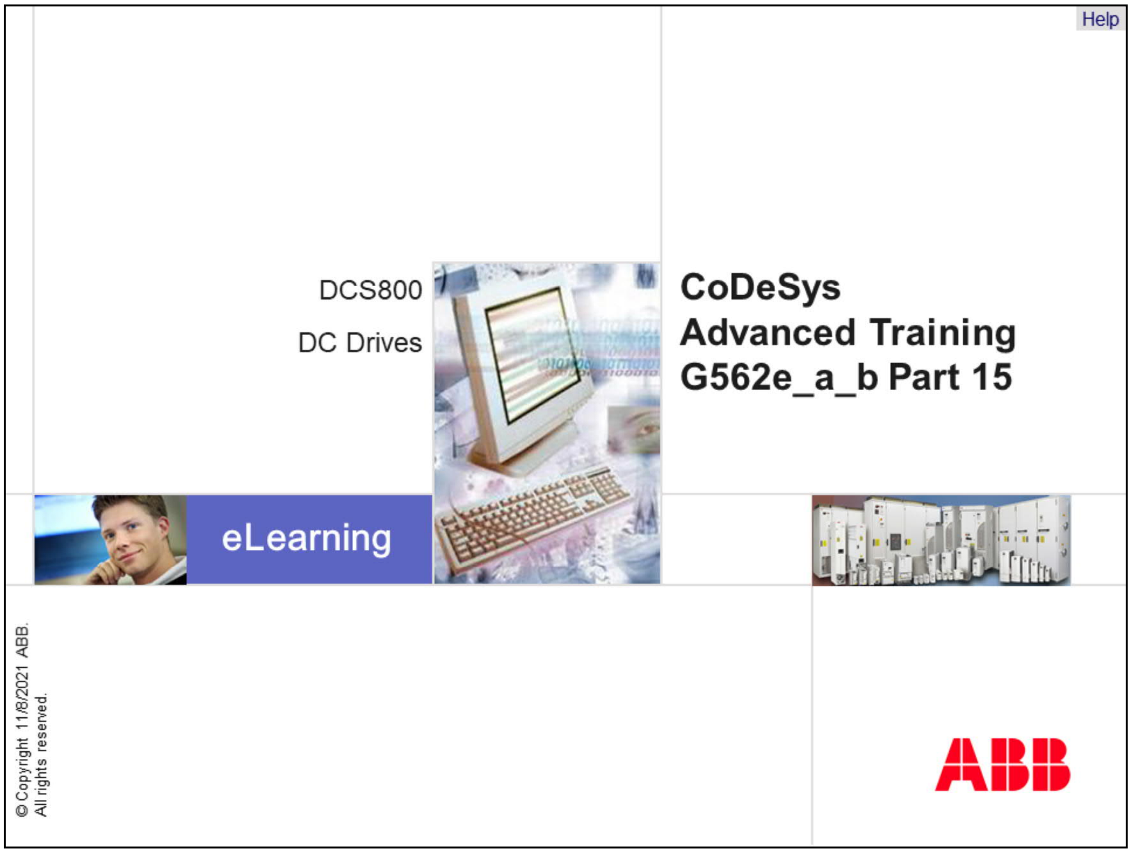


Help

DCS800

DC Drives

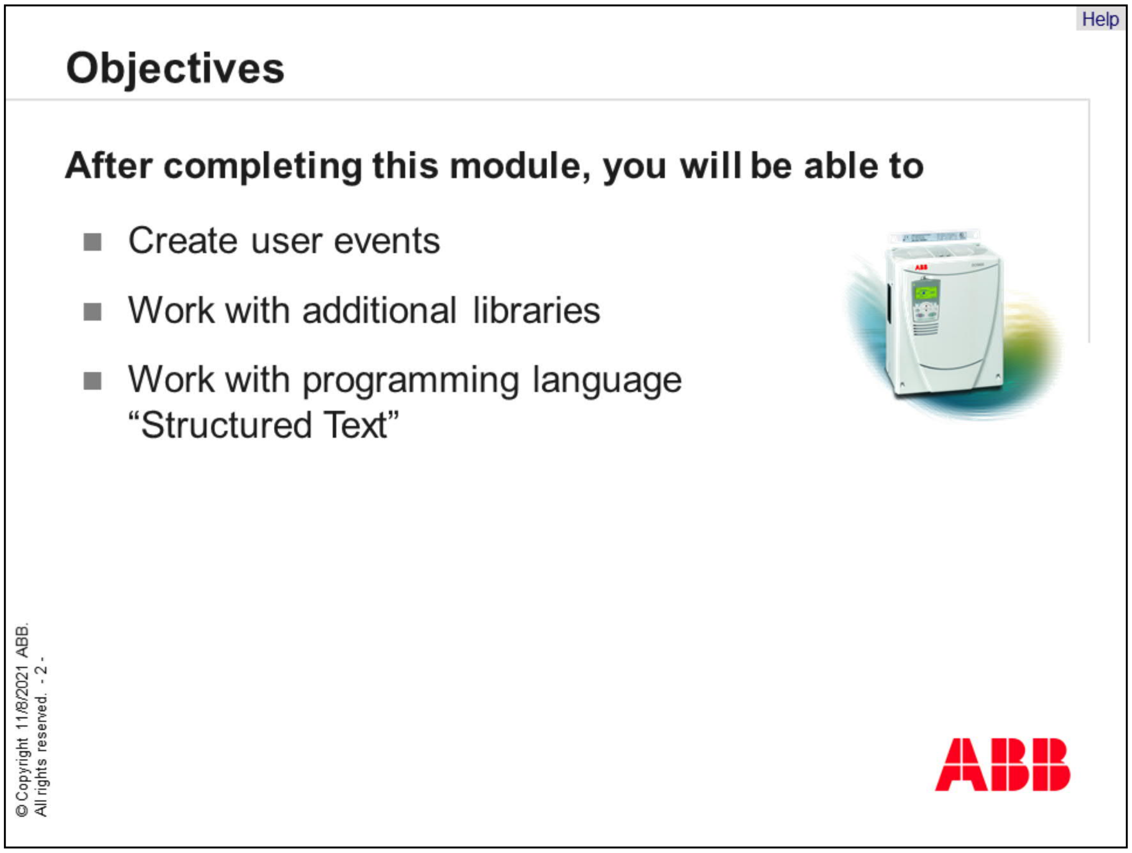
CoDeSys Advanced Training G562e_a_b Part 15

eLearning

ABB

Welcome to the CoDeSys training module for the DCS800, ABB DC drives.

If you need help navigating this module, please click the Help button in the top right-hand corner. To view the presenter notes as text, please click the Notes button in the bottom right corner.



Help

Objectives

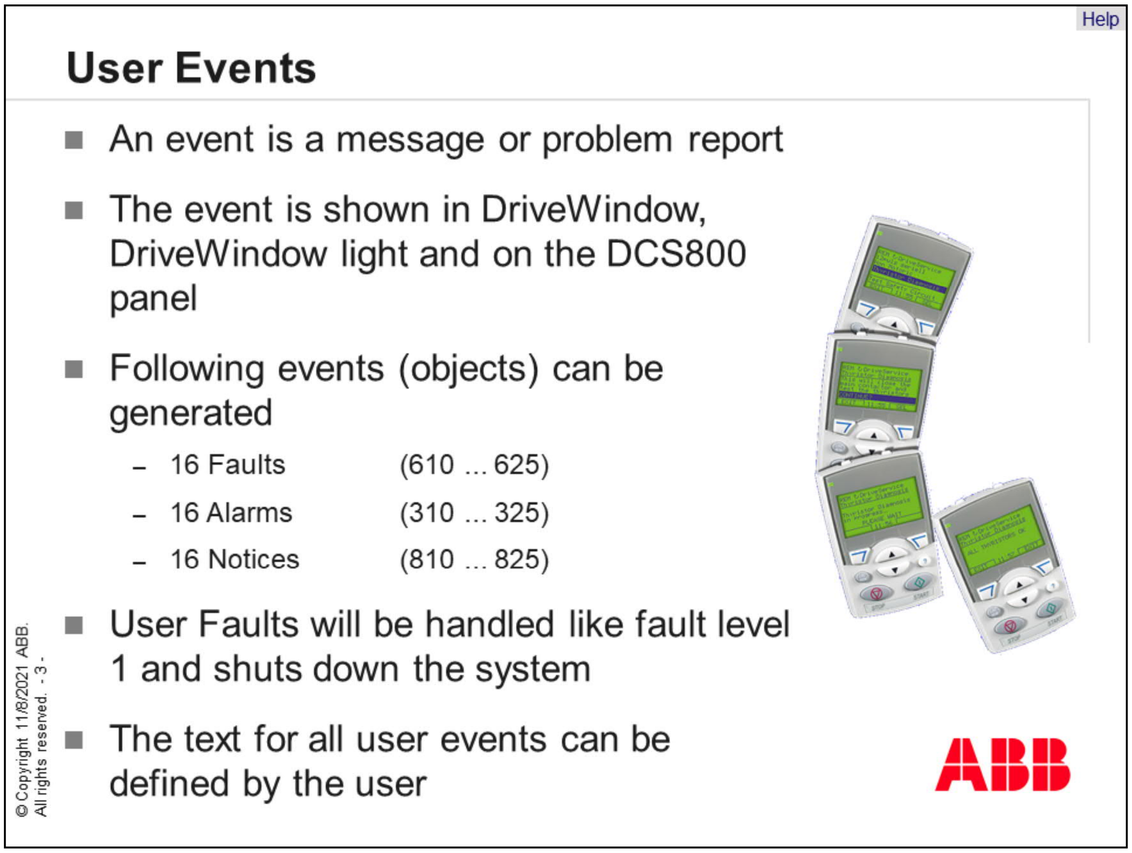
After completing this module, you will be able to

- Create user events
- Work with additional libraries
- Work with programming language “Structured Text”

ABB

After completing this module, you will be able to

- Create user events
- Work with additional libraries
- Work with the programming language "Structured Text"



Help

User Events

- An event is a message or problem report
- The event is shown in DriveWindow, DriveWindow light and on the DCS800 panel
- Following events (objects) can be generated

– 16 Faults	(610 ... 625)
– 16 Alarms	(310 ... 325)
– 16 Notices	(810 ... 825)

- User Faults will be handled like fault level 1 and shuts down the system
- The text for all user events can be defined by the user

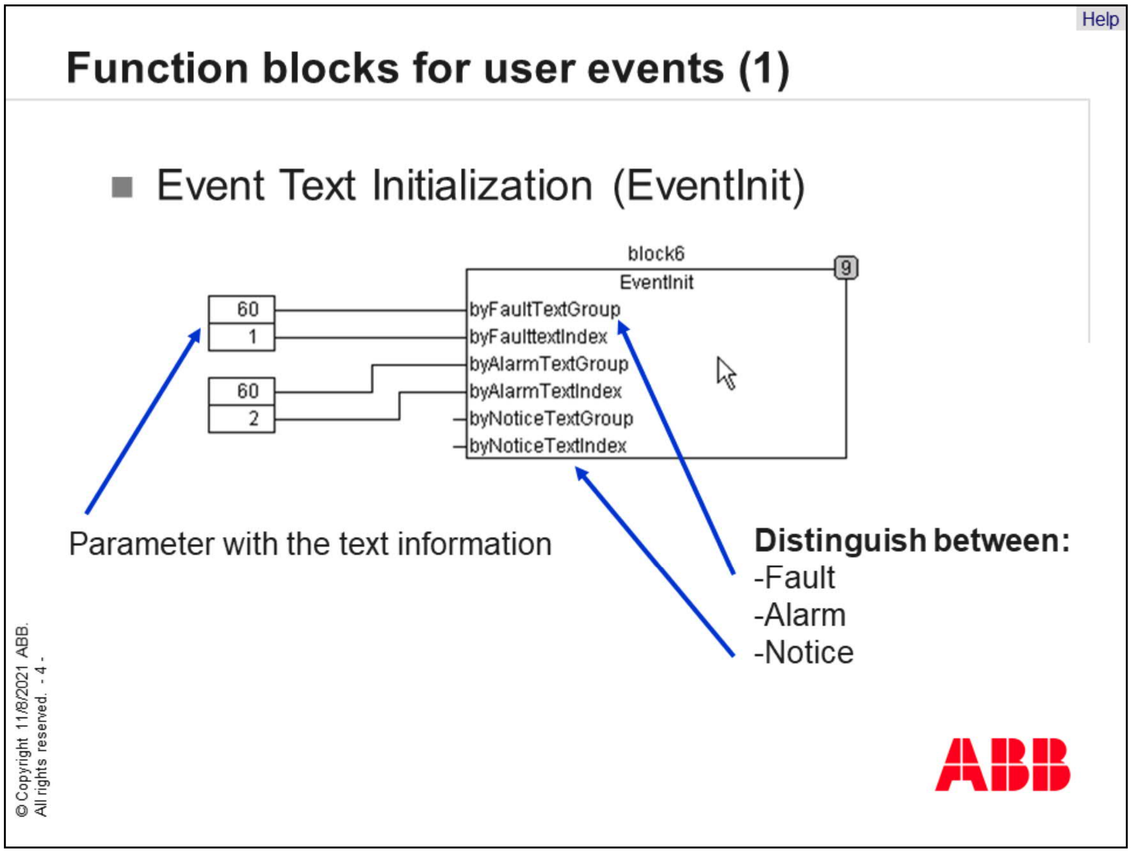
User events can be defined using CoDeSys.

An event is a message or an error report which provides the user with information about the drive status. The event is shown in "DriveWindow", "DriveWindow light" and on the DCS800 panel. Current faults are also saved in the fault logger!

The following events can be generated with CoDeSys: 16 faults, 16 alarms and 16 notices.

These events are reserved for application programming and sorted to fixed event numbers in the drive.

A "User Fault" will be handled like fault level 1 and shuts down the system. The text for all user events can be defined by the user.



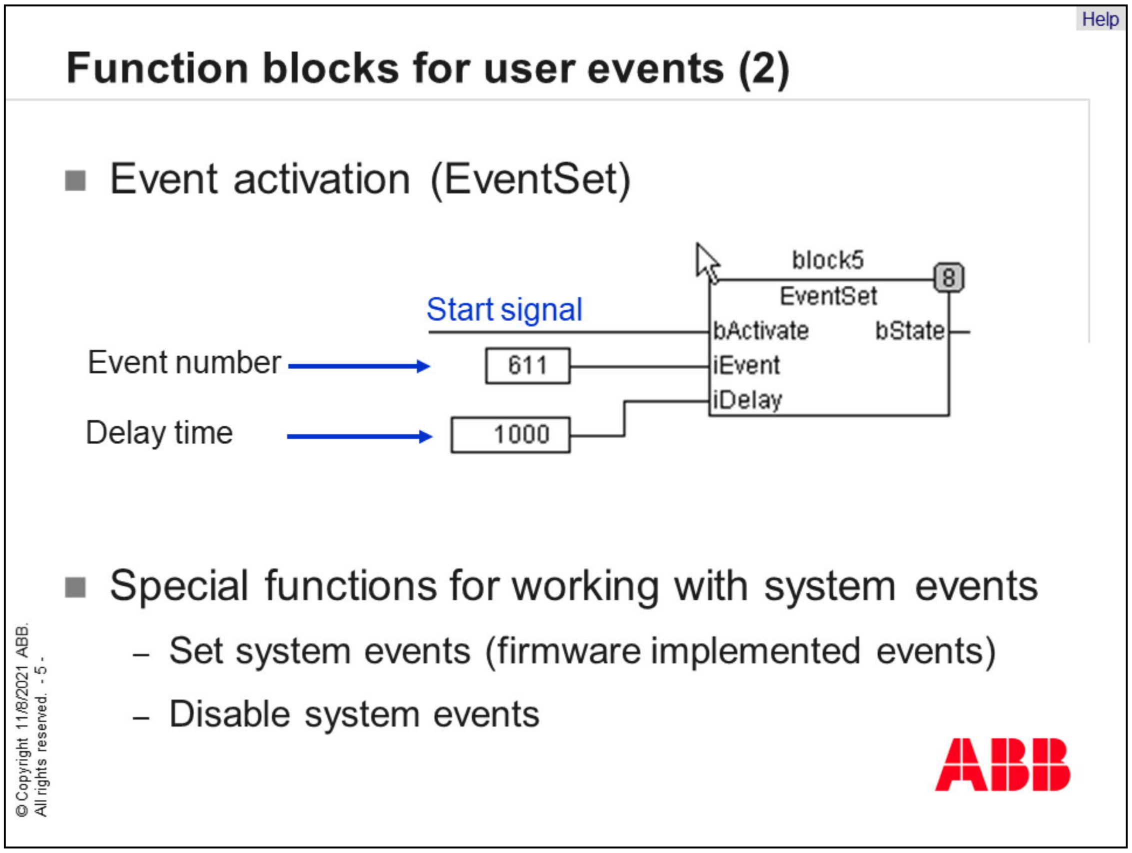
Function blocks for user events (1)

- Event Text Initialization (EventInit)

Function blocks for user events are implemented in the DCS800 Library.

The first function block to implement events is the event text initialization block. It is used to sort the texts for the events, which are saved in a user parameter, to the correct event type.

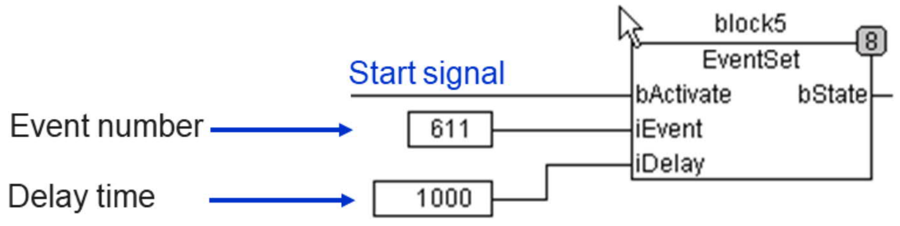
So it is important to distinguish between a fault, an alarm or a notice to avoid mixing up event texts!



Help

Function blocks for user events (2)

- Event activation (EventSet)



- Special functions for working with system events
  - Set system events (firmware implemented events)
  - Disable system events

The next function block for user events is the "EventSet" function block. This function block is used to activate an user event.

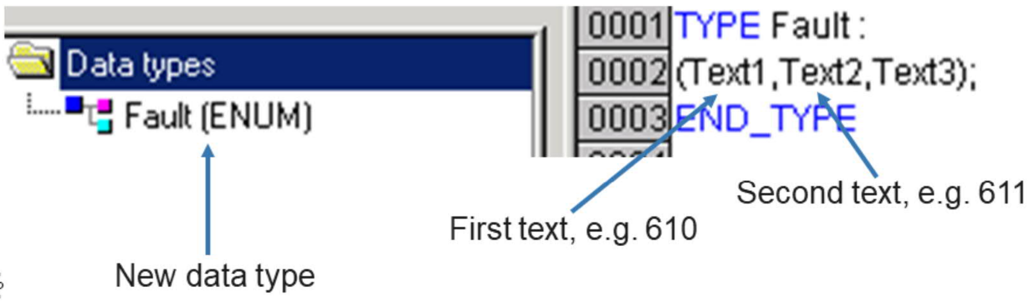
If the start signal reaches a high level the user event will be triggered. The user event with the event number will appear after a delay time.

A separate function block has to be used for each user event!

There is a special function for working with system events. So it is possible to activate or disable system events from drive firmware. Be careful with this functionality!

Event text declaration

- Event texts are declared as **Enums**
  This is a special data type for texts
- <u>Example:</u> New data type “Fault”



```
TYPE Fault :
(Text1,Text2,Text3);
END_TYPE
```

The next step in this procedure is to declare the event texts.

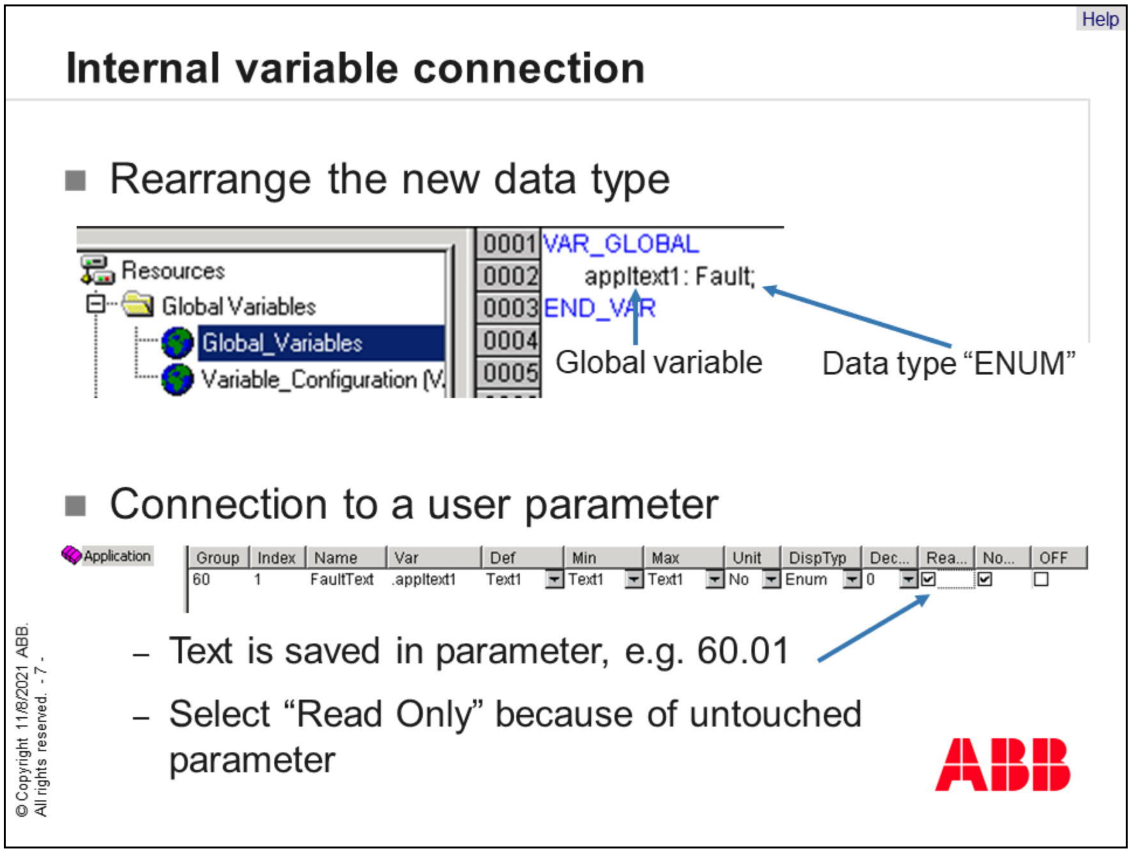
Event texts are declared as data type "enum". This is a special data type for texts.

Let us explain this procedure using a small example: New data type "Fault" has to be defined.

The list with the new data types includes the data type "fault" which is an "enum". Possible texts for this data type have to be defined in brackets. In this example the texts: (Text1, Text2 and Text3) are defined and can be used.

Note: The first text is sorted to the first event of this category and so on. In this example "Text1" is sorted to user event number 610.

The Next step is to connect the new data type "fault" to the parameter manager. This is necessary to save the text in the drive!



Help

Internal variable connection

- Rearrange the new data type

```
VAR_GLOBAL
    appltext1: Fault;
END_VAR
```

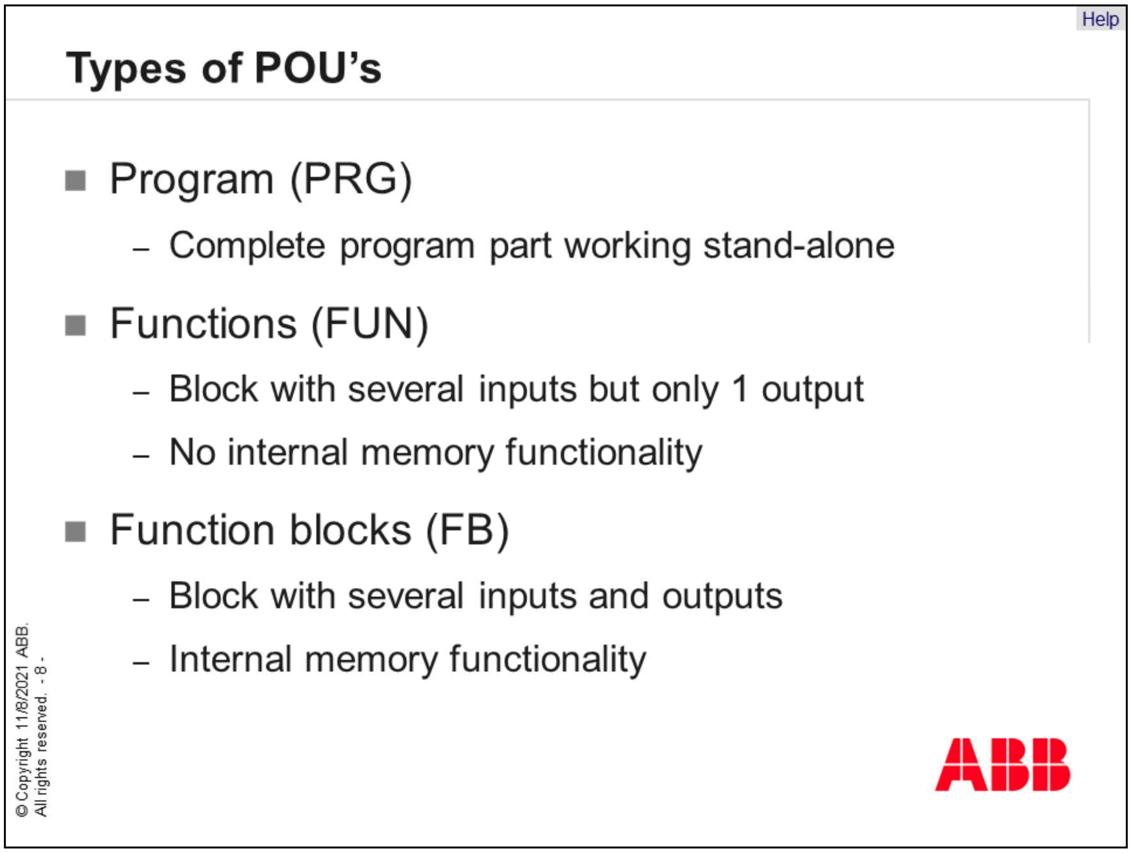
- Connection to a user parameter

Group	Index	Name	Var	Def	Min	Max	Unit	DispTyp	Dec...	Rea...	No...	OFF
60	1	FaultText	.appltext1	Text1	Text1	Text1	No	Enum	0	☑	☑	☐

  - Text is saved in parameter, e.g. 60.01
  - Select “Read Only” because of untouched parameter

The next step is the internal variable connection. In the last folder, the new data type "Fault" was created. Now this data type must be connected as a global variable to rearrange the data.

Then this global parameter has to be connected with a user parameter. Further settings must be defined to protect this parameter with the included text for the events in the parameter manager. If the read only box has a tick in it, the parameter will be a signal and is untouched by the user.

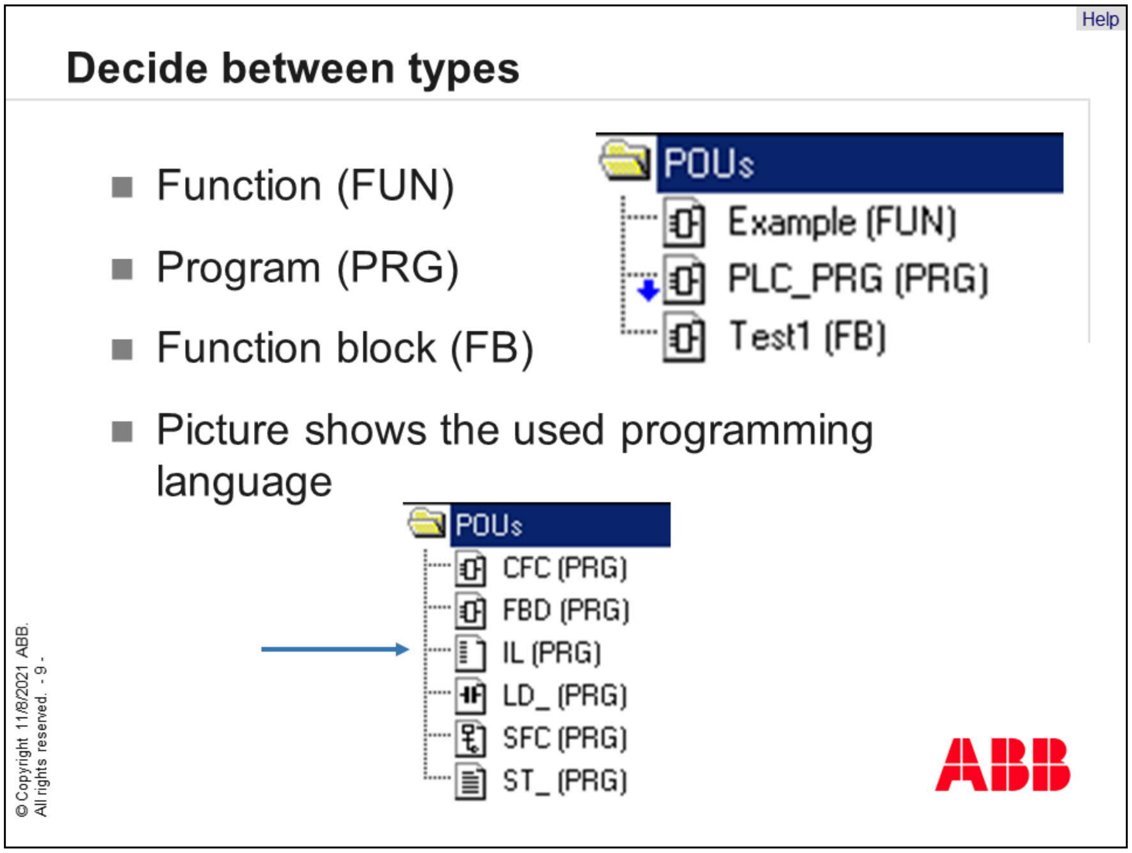

Help

Types of POU's

- Program (PRG)
  - Complete program part working stand-alone
- Functions (FUN)
  - Block with several inputs but only 1 output
  - No internal memory functionality
- Function blocks (FB)
  - Block with several inputs and outputs
  - Internal memory functionality

In CoDeSys there are several types of program organization units, called "POUs".

A program is a complete part which is able to work stand-alone. Functions are blocks with several inputs but only one output. The function blocks have the highest functionality with several inputs and outputs. A memory functionality is available internally.



Help

Decide between types

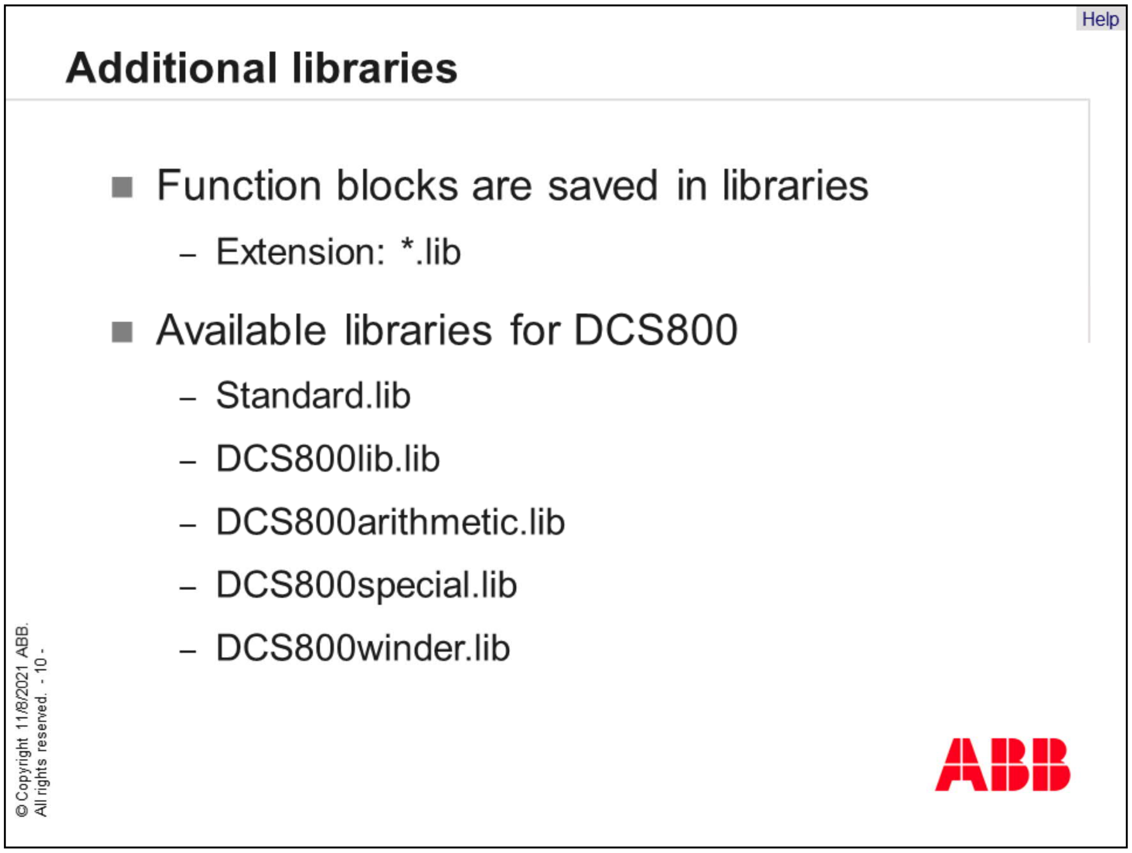
- Function (FUN)
- Program (PRG)
- Function block (FB)
- Picture shows the used programming language

POUs
Example (FUN)
PLC_PRG (PRG)
Test1 (FB)
POUs
CFC (PRG)
FBD (PRG)
IL (PRG)
LD_ (PRG)
SFC (PRG)
ST_ (PRG)

It is essential to decide between the types of POUs. Which type has to be used depends on the application and the required functionality.

After the definition, the selected type will be shown in the organization directory with the correct extension.

Furthermore, a graphic displays the programming language being used.


Help

Additional libraries

- Function blocks are saved in libraries
  - Extension: *.lib
- Available libraries for DCS800
  - Standard.lib
  - DCS800lib.lib
  - DCS800arithmetic.lib
  - DCS800special.lib
  - DCS800winder.lib

Additional libraries are available for the DCS800 application system.

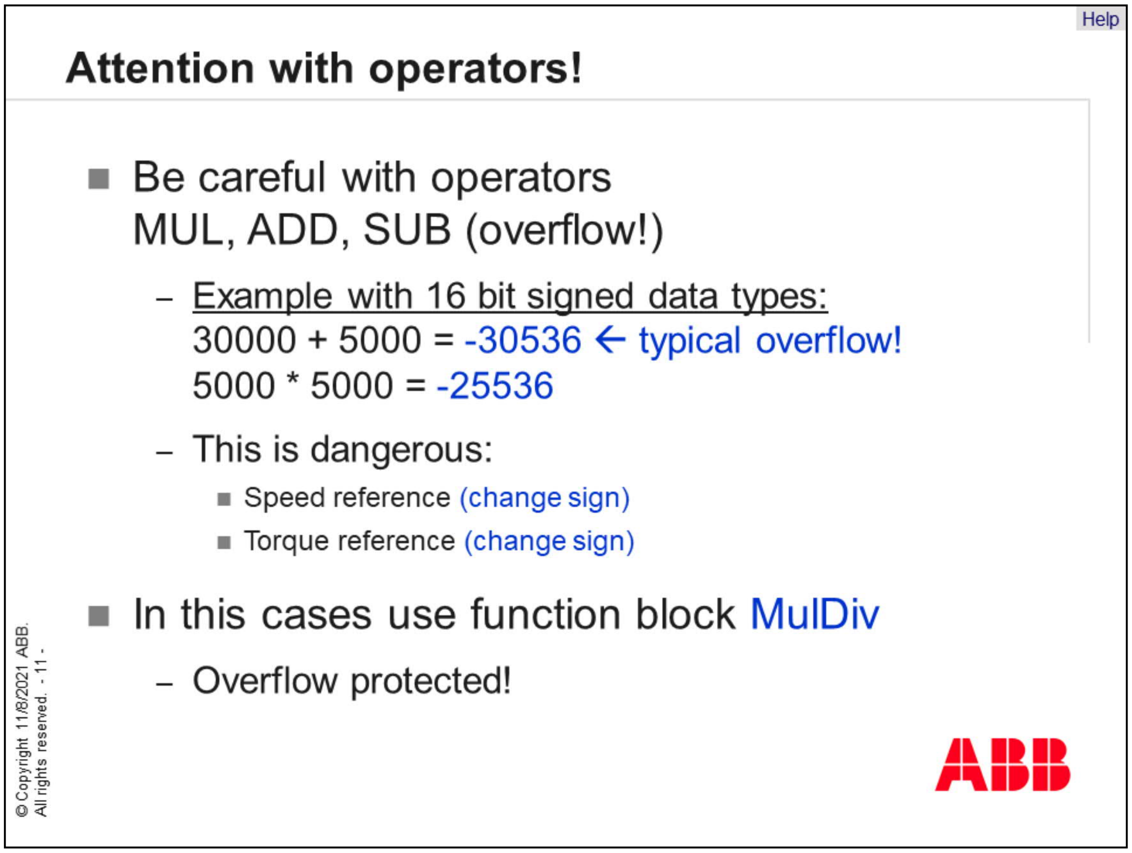
Function blocks are saved in libraries with the extension "LIB". These libraries can be used to create application programs.

The following libraries for the DCS800 are available:

- The Standard Library. This library includes function blocks for timer, counter or bistable functions
- The DCS800 Library. This library includes function blocks to communicate with the drive's firmware
- The DCS800 Arithmetic Library. This library includes function blocks to realize arithmetical functions
- The DCS800 Special Library. This library includes function blocks for bit operations
- The DCS800 Winder Library. This library includes function blocks for winding

It is only possible to compile, if the libraries are included. In many cases it is necessary to have more function blocks available than the interface library contains. Note that these libraries are only compatible with DCS800. The "standard library" is a global library made from the CoDeSys manufacturer and is available in all PLC systems.

The "DCS800 libraries" are developed specially for the DCS800 drive system.



Help

Attention with operators!

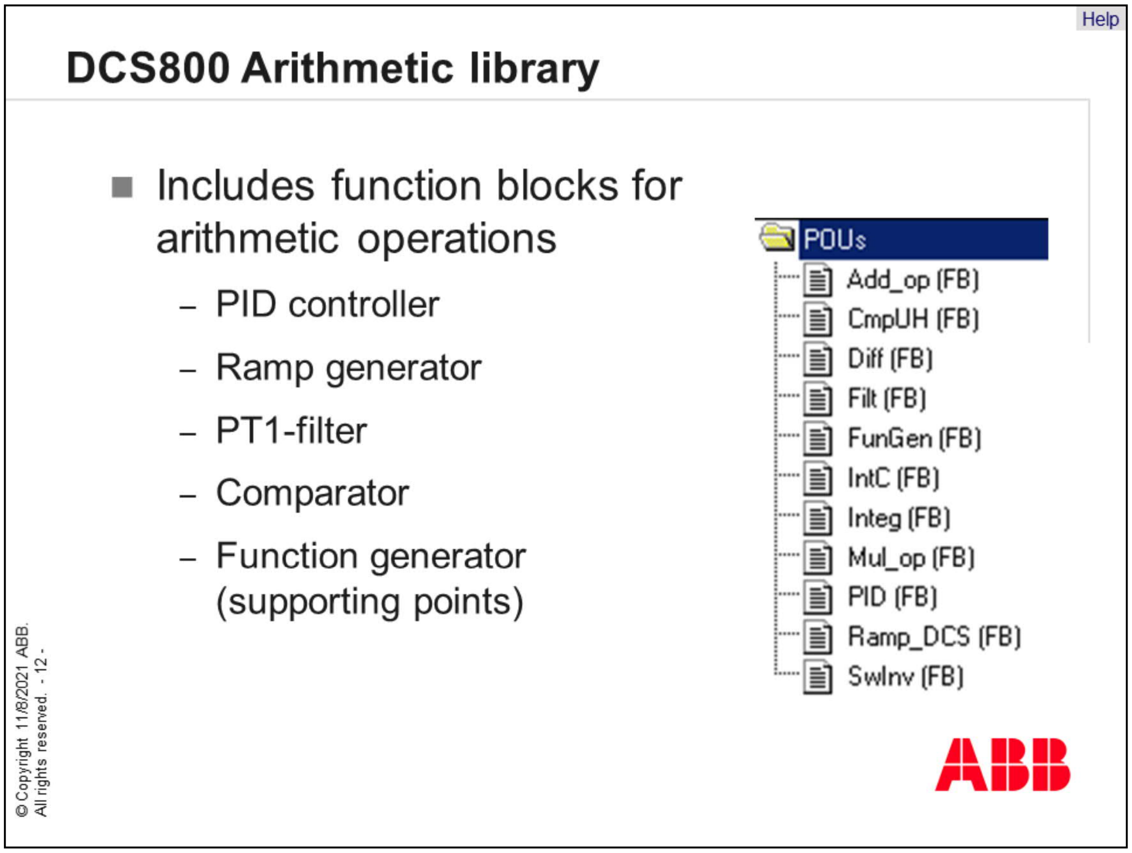
- Be careful with operators MUL, ADD, SUB (overflow!)
  - <u>Example with 16 bit signed data types:</u>
    30000 + 5000 = -30536 ← typical overflow!
    5000 * 5000 = -25536
  - This is dangerous:
    - Speed reference (change sign)
    - Torque reference (change sign)
- In this cases use function block MulDiv
  - Overflow protected!

In some cases, it is dangerous to work with the IEC operators. None of the IEC operators have overflow protection.

Be careful with operators like "MUL, ADD or SUB", if the result is to be used to set references for the drive. The sign of this value will change if an overflow occurs!

The example shows typical overflows with operators. If 16-bit signed data types are used, the result could be negative if there is no overflow protection. This is dangerous if the result should be used to connect a value to a speed or torque reference.

In these cases, use function block "MulDiv" from the "DCS800 Library". Only this function block has overflow protection.



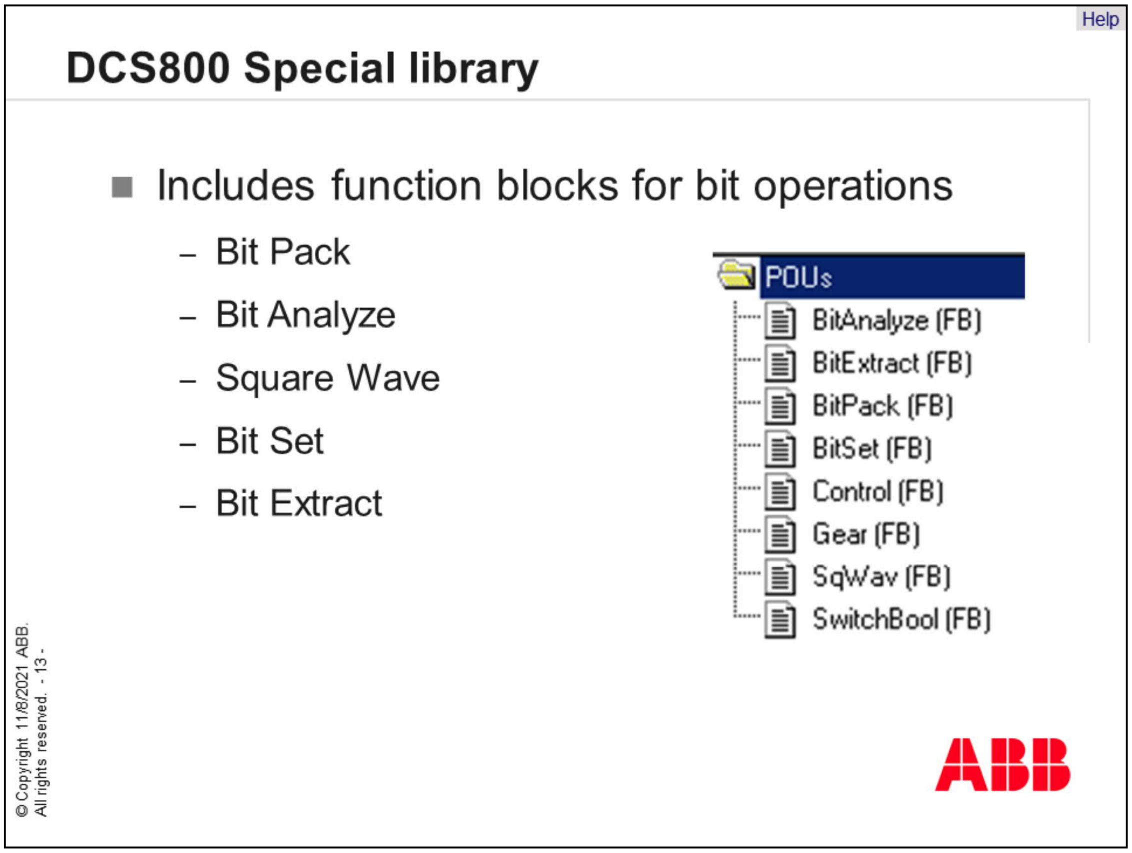
Help

DCS800 Arithmetic library

- Includes function blocks for arithmetic operations
  - PID controller
  - Ramp generator
  - PT1-filter
  - Comparator
  - Function generator (supporting points)

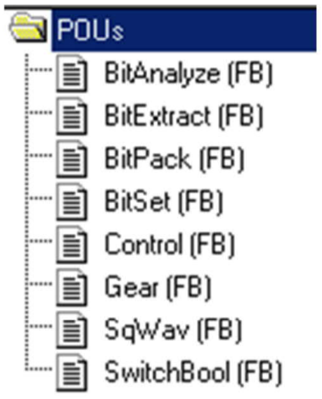
POUs
Add_op (FB)
CmpUH (FB)
Diff (FB)
Filt (FB)
FunGen (FB)
IntC (FB)
Integ (FB)
Mul_op (FB)
PID (FB)
Ramp_DCS (FB)
SwInv (FB)

The "DCS800 Arithmetic library" includes function blocks like “PID controller”, “ramp generator”, “PT1-filter”, “comparator” and “function generator”. This library only works with the DCS800 system! A description of this library can be found in the "DCS800 Application Manual".


Help

DCS800 Special library

- Includes function blocks for bit operations
  - Bit Pack
  - Bit Analyze
  - Square Wave
  - Bit Set
  - Bit Extract

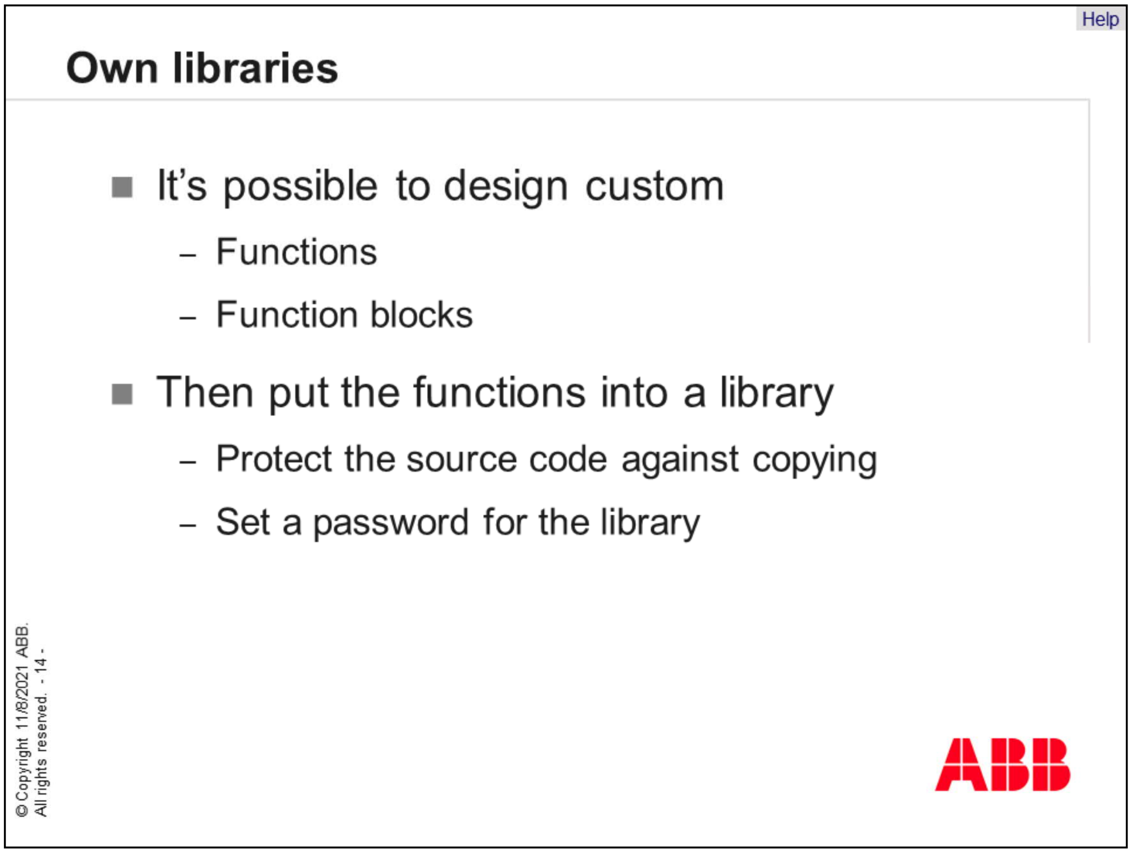


The "DCS800 Special library" includes function blocks for bit operations.

With "Bit Pack" and "Bit Extract" function blocks, "WORD values" can be split into "Boolean" values and vice versa.

Function block "Bit Analyze" and "Bit Set" can be used to analyze one bit in a WORD or to set a bit in a WORD.

This library only works with the DCS800 library. A description of the available function blocks can be found in the "DCS800 Application Manual".



Help

Own libraries

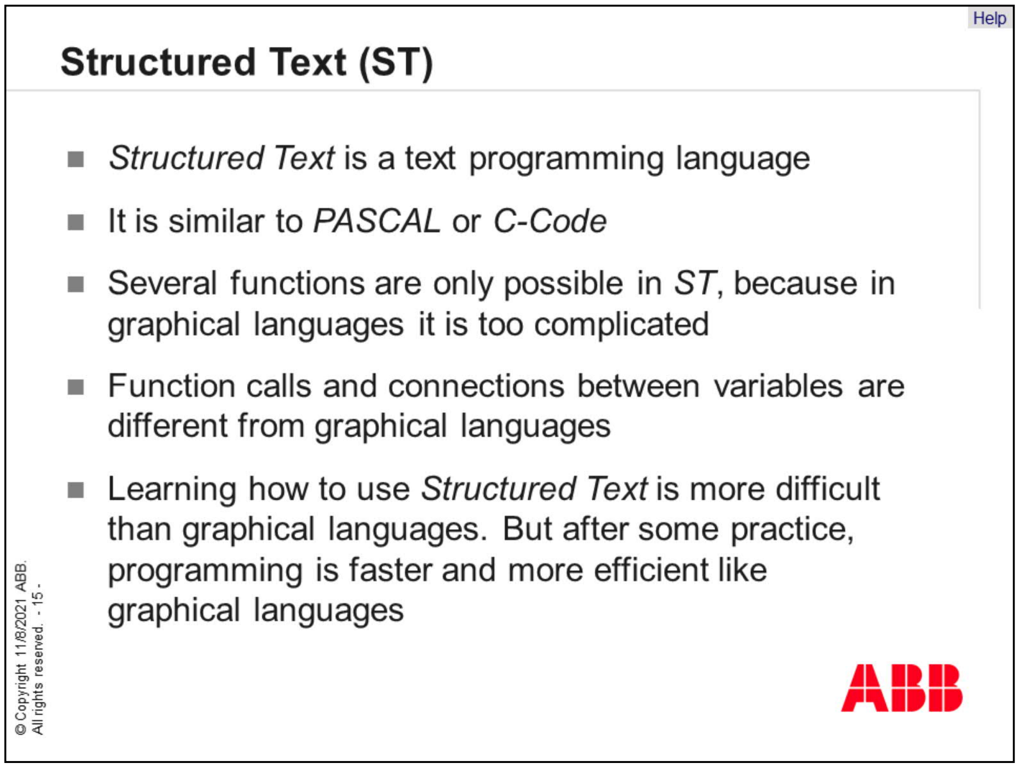
- It’s possible to design custom
  - Functions
  - Function blocks
- Then put the functions into a library
  - Protect the source code against copying
  - Set a password for the library

Custom libraries can be created by using CoDeSys. This allows structured work and provides the possibility to administer libraries.

It is possible to design custom functions and function blocks. All available programming languages can be used.

Then put the functions and function blocks into a library. A library protects the source code against copying, and it is possible to set a password for the library if know how should be protected.

Note: Only functions and function blocks will be added to a library.



Help

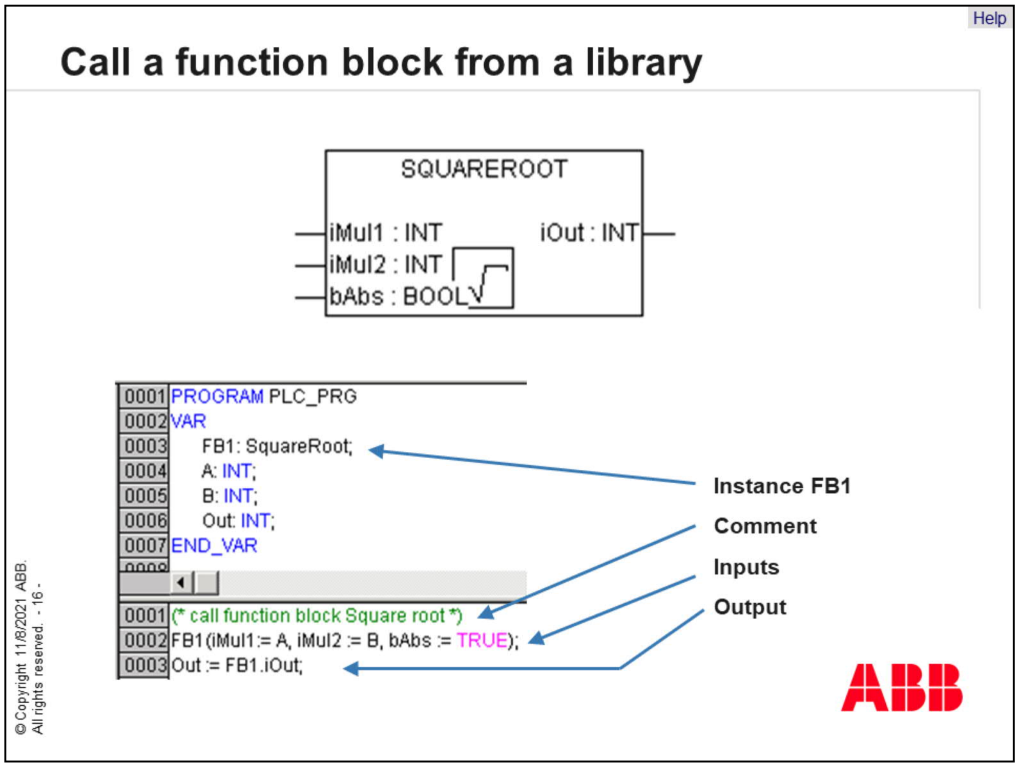
Structured Text (ST)

- *Structured Text* is a text programming language
- It is similar to *PASCAL* or *C-Code*
- Several functions are only possible in *ST*, because in graphical languages it is too complicated
- Function calls and connections between variables are different from graphical languages
- Learning how to use *Structured Text* is more difficult than graphical languages. But after some practice, programming is faster and more efficient like graphical languages

ABB

The next topic is the programming language "Structured Text".

- "Structured Text" is a text-based programming language in CoDeSys.
- It is similar to "PASCAL" or "C-Code".
- Several functions are only possible in "Structured Text" because in graphical languages it is too complex.
- Function calls and connections between variables are different from graphical languages.
- Learning how to use "Structured Text" is more difficult than graphical languages. But after some practice, programming is faster and more efficient like graphical languages.



Call a function block from a library

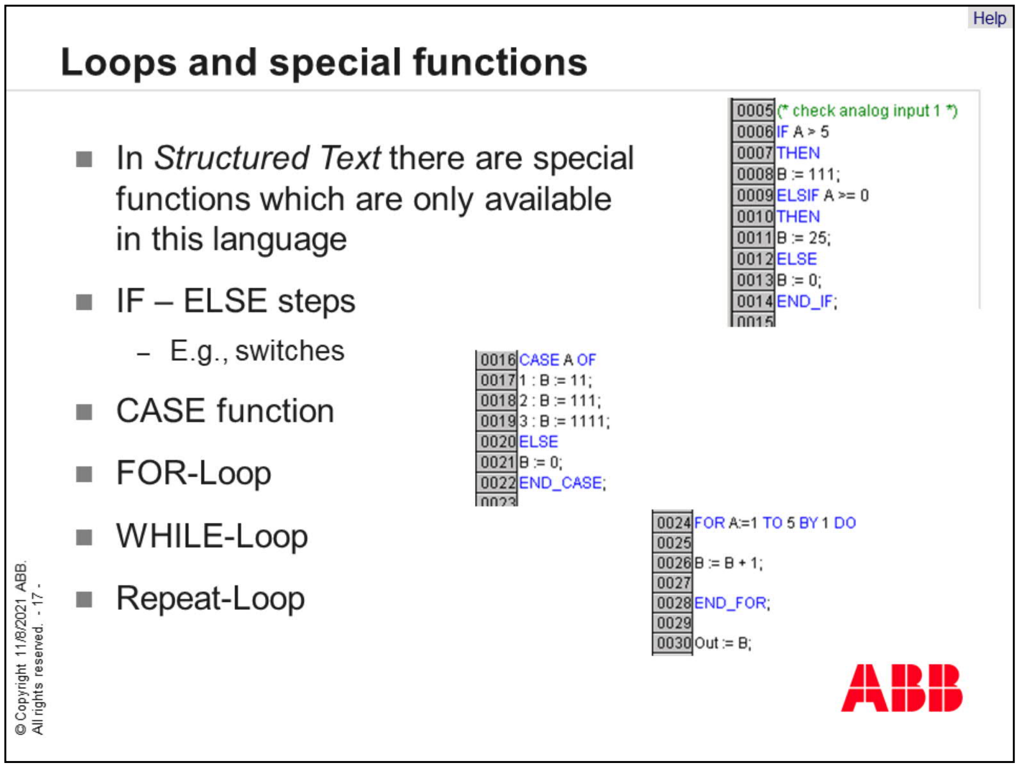
```
PROGRAM PLC_PRG
VAR
    FB1: SquareRoot;
    A: INT;
    B: INT;
    Out: INT;
END_VAR
```

```
(* call function block Square root *)
FB1(iMul1:= A, iMul2 := B, bAbs := TRUE);
Out := FB1.iOut;
```

In graphical languages a call of a function block is easy to do when you put in function blocks. In a text language the function block is not visible at all. You only see a line with variables and allocations. But with "Structured Text" you can do the same things and more as with graphical languages.

To call a function block from the library, type in an instance name, for example "FB1". Then set brackets by pressing F2. Now you can select the function block from the library. What you see inside the brackets are the inputs of the selected function block. Connections between an input and other functions are possible with variables.

Outputs are not inside the brackets because it is easier to use the instance of the function block with the selected output.



Help

Loops and special functions

- In *Structured Text* there are special functions which are only available in this language
- IF – ELSE steps
  - E.g., switches
- CASE function
- FOR-Loop
- WHILE-Loop
- Repeat-Loop

```
(* check analog input 1 *)
IF A > 5
THEN
B := 111;
ELSIF A >= 0
THEN
B := 25;
ELSE
B := 0;
END_IF;
```

```
CASE A OF
1 : B := 11;
2 : B := 111;
3 : B := 1111;
ELSE
B := 0;
END_CASE;
```

```
FOR A:=1 TO 5 BY 1 DO

B := B + 1;

END_FOR;

Out := B;
```

© Copyright 11/8/2021 ABB. All rights reserved. - 17 -

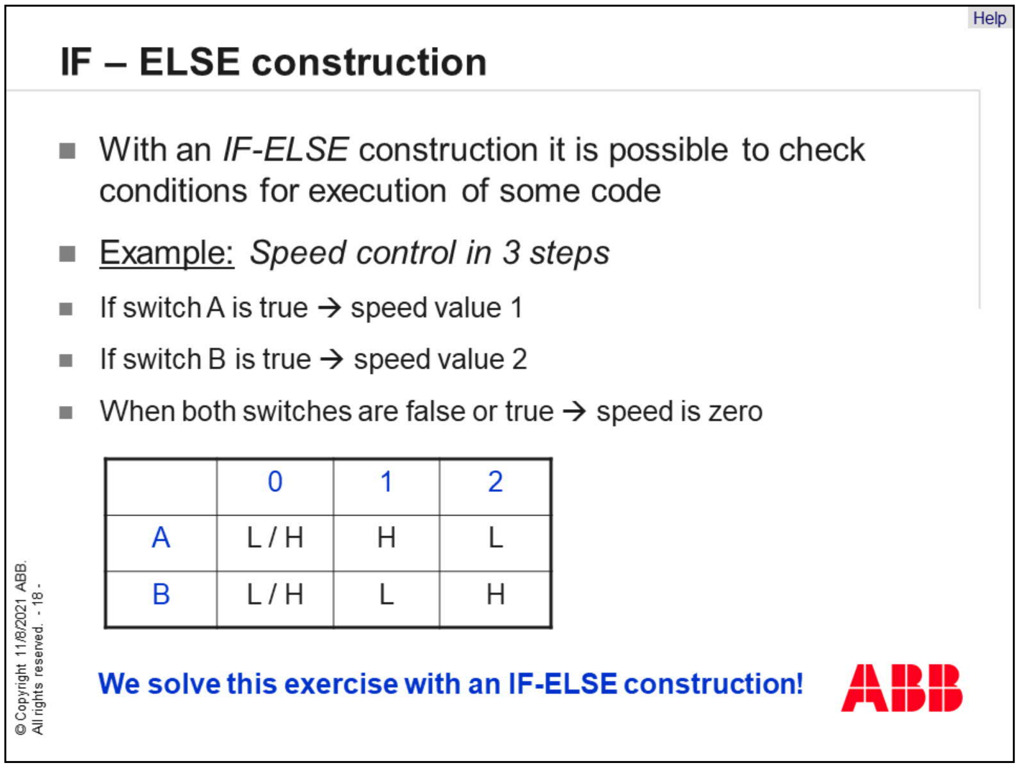
Loops and special functions are discussed in the following chapter.

In "Structured Text" there are special functions which are only available in this programming language.

An "IF-ELSE-configuration" includes functionalities such as a Boolean switch with several conditions.

The "CASE-function" is a selection due to conditions. The integer switch is used to distinguish between the "IF-ELSE" configurations.

The loops in CoDeSys are comparable with loops in high level programming languages. With loops an operation in a line can be executed several times until a condition is fulfilled.


Help

IF – ELSE construction

- With an *IF-ELSE* construction it is possible to check conditions for execution of some code
- Example: *Speed control in 3 steps*
- If switch A is true → speed value 1
- If switch B is true → speed value 2
- When both switches are false or true → speed is zero

	0	1	2
A	L / H	H	L
B	L / H	L	H

We solve this exercise with an IF-ELSE construction!

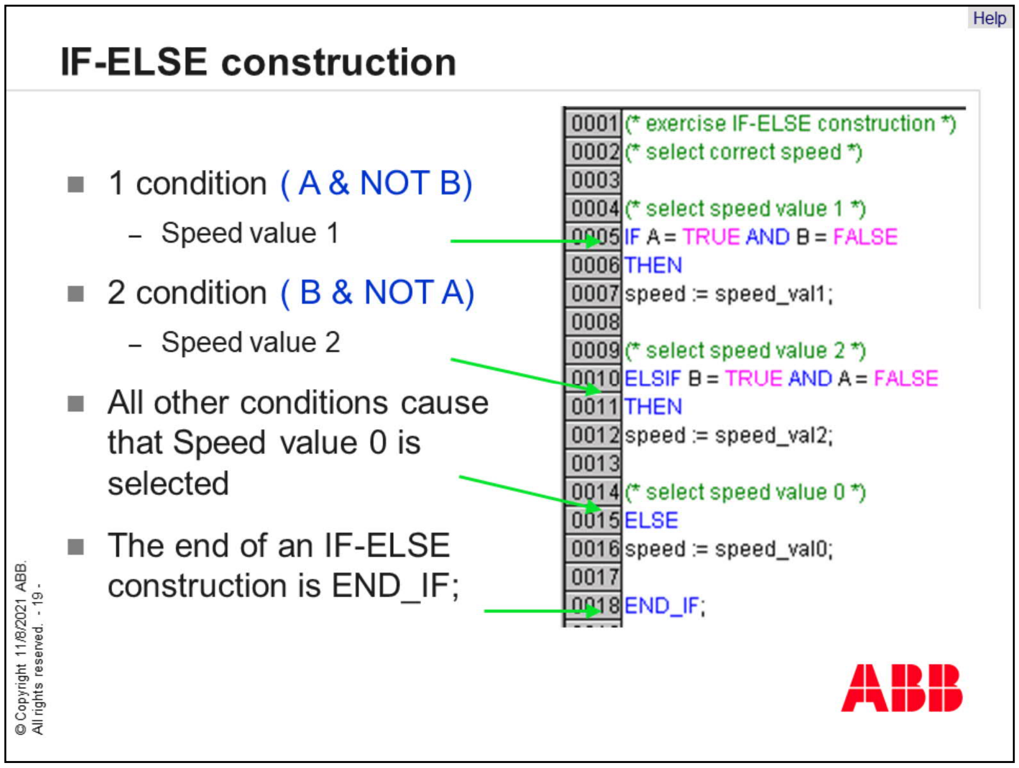
An "IF-ELSE-construction" is used to check conditions for execution of some code. Only if the condition is TRUE the code will be executed. Otherwise, it jumps over to the next condition check.

An example is a speed control in 3 steps using 2 switches.

If switch A is TRUE, the speed value 1 is selected.

If switch B is TRUE, the speed value 2 is selected.

When both switches are FALSE or TRUE, the speed should be zero.



IF-ELSE construction

- 1 condition (A & NOT B)
  - Speed value 1
- 2 condition (B & NOT A)
  - Speed value 2
- All other conditions cause that Speed value 0 is selected
- The end of an IF-ELSE construction is END_IF;

```
(* exercise IF-ELSE construction *)
(* select correct speed *)

(* select speed value 1 *)
IF A = TRUE AND B = FALSE
THEN
speed := speed_val1;

(* select speed value 2 *)
ELSIF B = TRUE AND A = FALSE
THEN
speed := speed_val2;

(* select speed value 0 *)
ELSE
speed := speed_val0;

END_IF;
```

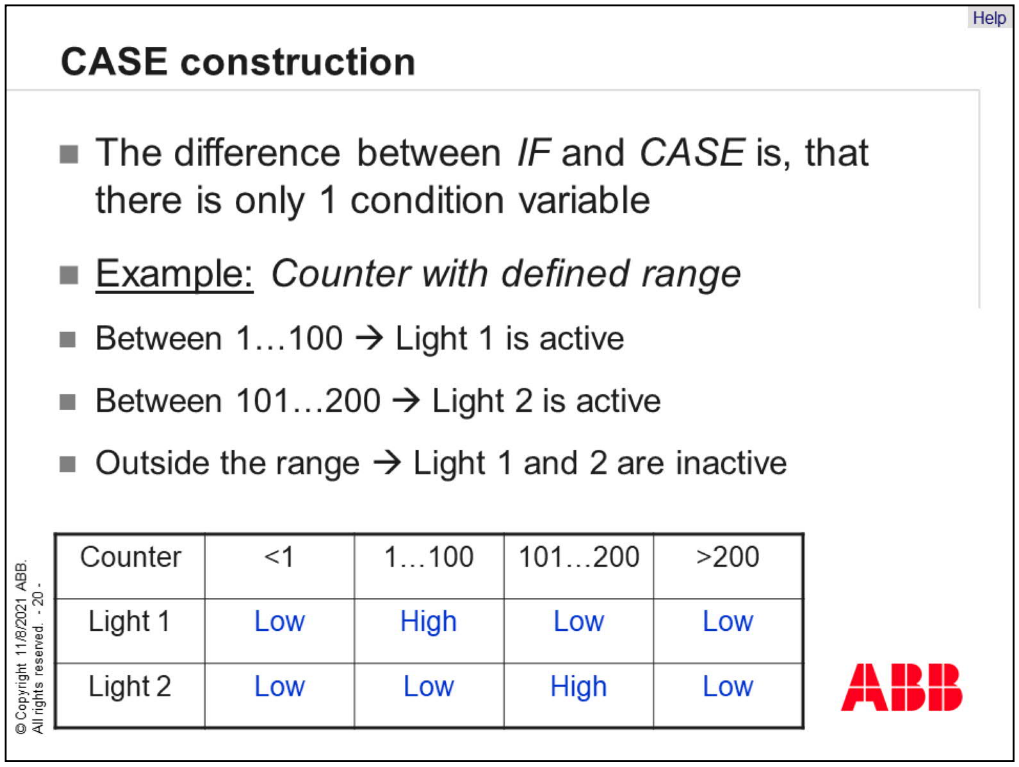
This is an example of an "IF_ELSE_construction".

In the first selection the variables A and B are used. The logical connection is like the following: If variable “A” is “TRUE” and variable “B” is “FALSE” then “speed value 1” is selected.

When this condition is not fulfilled, the program jumps to the next request. If variable “B” is “TRUE” and variable “A” is “FALSE” then speed value 2 is selected.

If none of these conditions are fulfilled, speed value 0 is selected.

The whole construction ends with the command "END_IF".



Help

CASE construction

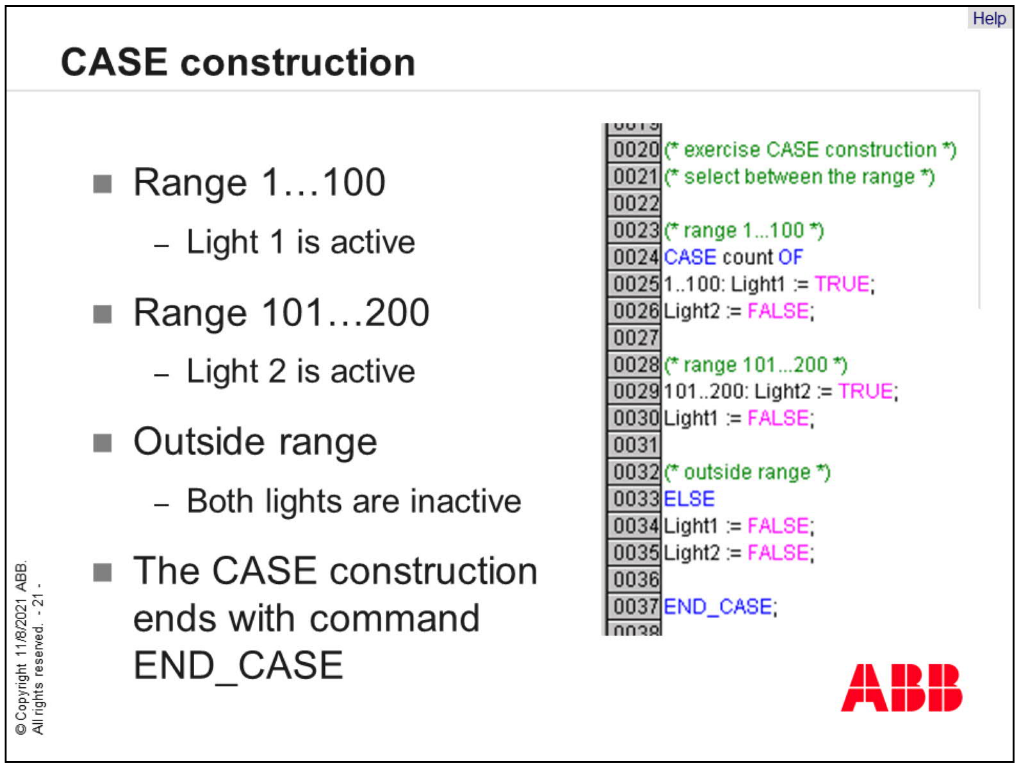
- The difference between *IF* and *CASE* is, that there is only 1 condition variable
- Example: *Counter with defined range*
- Between 1…100 → Light 1 is active
- Between 101…200 → Light 2 is active
- Outside the range → Light 1 and 2 are inactive

Counter	<1	1…100	101…200	>200
Light 1	Low	High	Low	Low
Light 2	Low	Low	High	Low

ABB

Now let's discuss the "CASE-Construction". The difference between "IF" and "CASE" is that there is only 1 condition variable. This variable has more than 2 conditions and the "CASE-Construction" checks the actual state of the variable. Then it decides!

An example is a counter with a defined range. The selection works between the defined ranges. So, the counter goes up and down and the result is shown at light 1 or 2 if the counter is inside the defined range.



Help

CASE construction

- Range 1…100
  - Light 1 is active
- Range 101…200
  - Light 2 is active
- Outside range
  - Both lights are inactive
- The CASE construction ends with command END_CASE

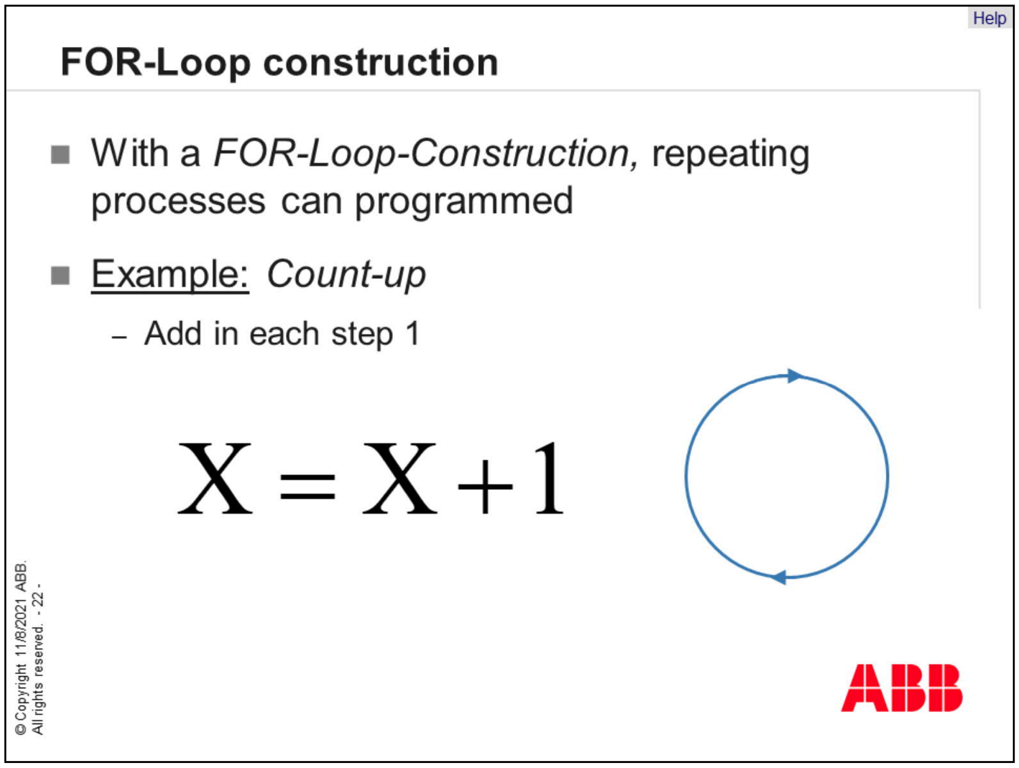
```
(* exercise CASE construction *)
(* select between the range *)

(* range 1...100 *)
CASE count OF
1..100: Light1 := TRUE;
Light2 := FALSE;

(* range 101...200 *)
101..200: Light2 := TRUE;
Light1 := FALSE;

(* outside range *)
ELSE
Light1 := FALSE;
Light2 := FALSE;

END_CASE;
```

The "CASE-Construction" looks like the graphic on the right side. We have one variable, here "count", and several steps with the defined range.

The example logic is like the following:

If the variable "count" is between 1 and 100, light 1 is active. Between 101 and 200, light 2 is active. If the variable "count" is out of range, both lights are inactive.

Each "CASE-Construction" ends with the command "END_CASE".



Help

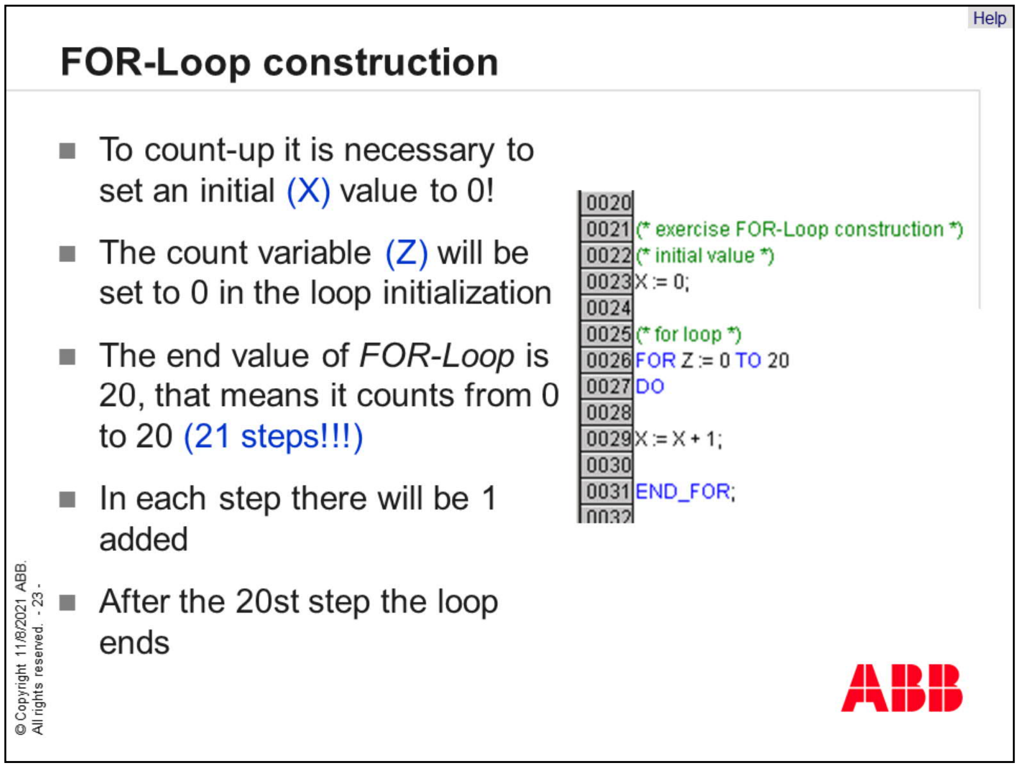
FOR-Loop construction

- With a *FOR-Loop-Construction,* repeating processes can programmed
- Example: *Count-up*
  - Add in each step 1

$$X = X + 1$$

ABB

Another important function is the "FOR-Loop-Construction". It has the same functionality as a counter because the loop will be executed as long as the condition is fulfilled. An example is an upward counter. The user defines the starting point and the end point. Later the loop runs from the starting point to the end and increases the counter value in each cycle.


Help

FOR-Loop construction

- To count-up it is necessary to set an initial (X) value to 0!
- The count variable (Z) will be set to 0 in the loop initialization
- The end value of *FOR-Loop* is 20, that means it counts from 0 to 20 (21 steps!!!)
- In each step there will be 1 added
- After the 20st step the loop ends

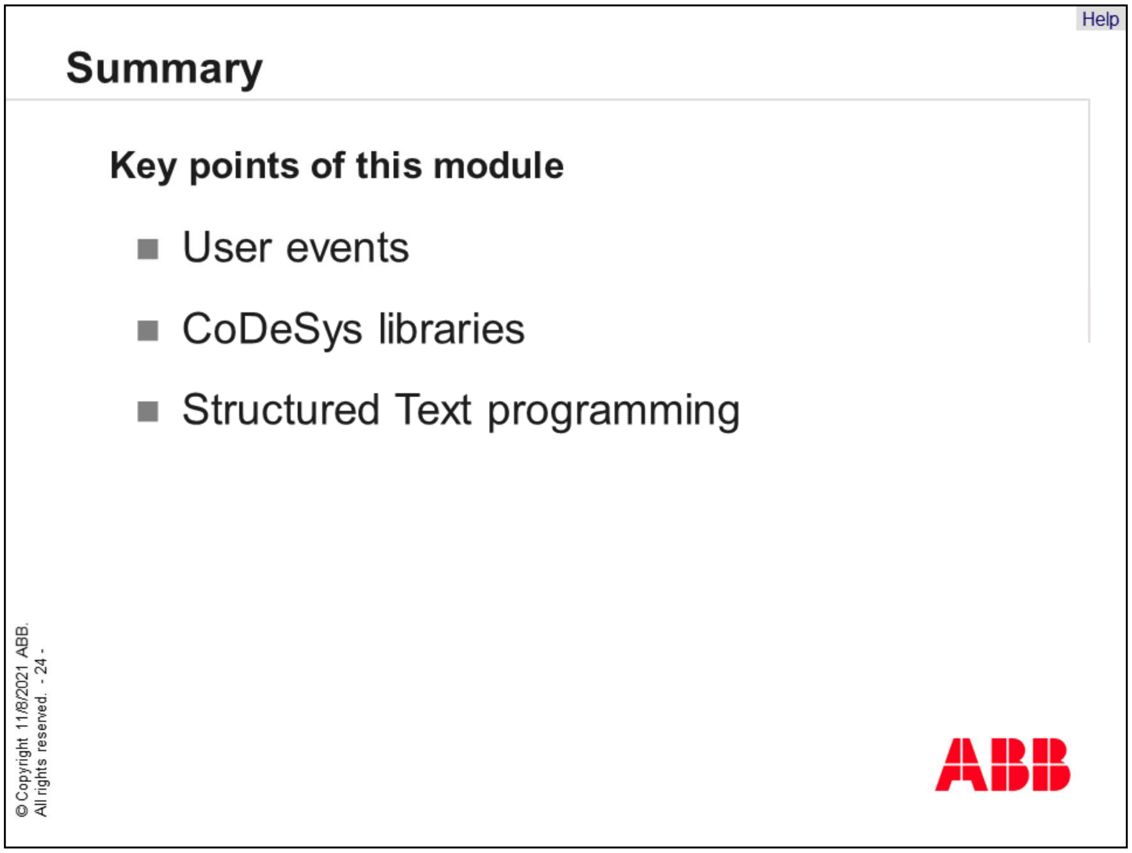
```

(* exercise FOR-Loop construction *)
(* initial value *)
X := 0;

(* for loop *)
FOR Z := 0 TO 20
DO

X := X + 1;

END_FOR;
```

The graphic shows the code of a "FOR-loop construction".

It starts with the initial value which has to be set.

The counter variable will be set to zero when the loop initializes. The next step is to define the end value.

In this example the loop counts-up the variable “X” in each step until “Z” is 20. This adds up to 21 steps because zero is counted first. The loop ends after the 20th step.


Help

Summary

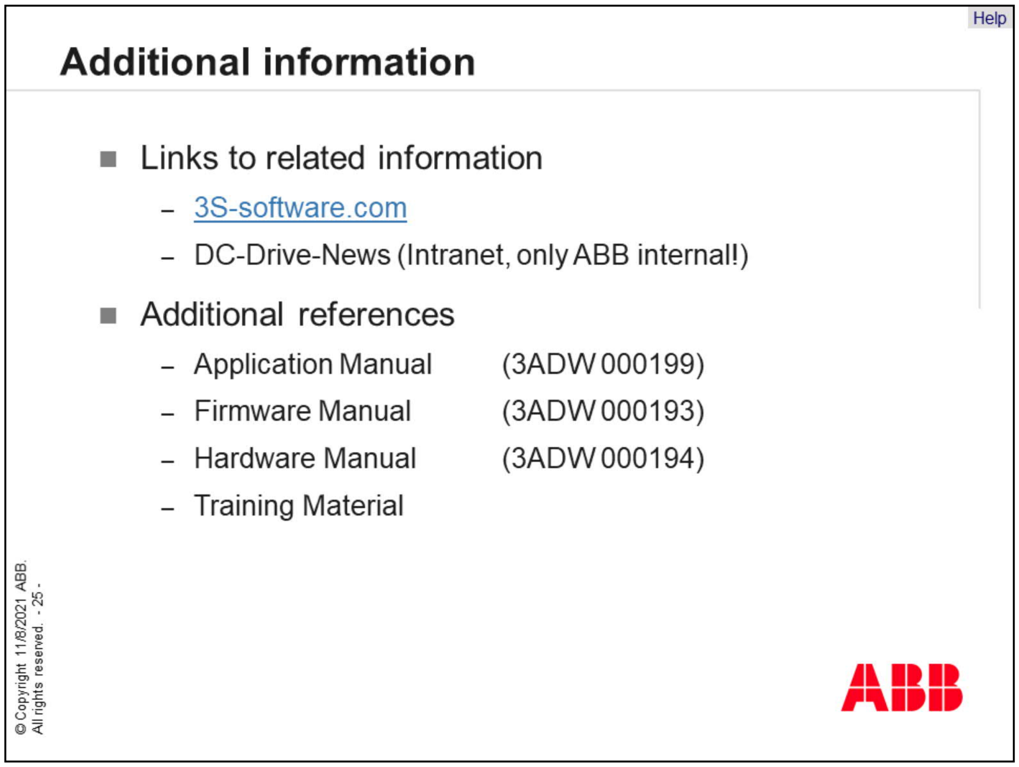
Key points of this module

- User events
- CoDeSys libraries
- Structured Text programming

ABB

The key points of this module are

- User events,
- the CoDeSys libraries and
- Structured Text programming.



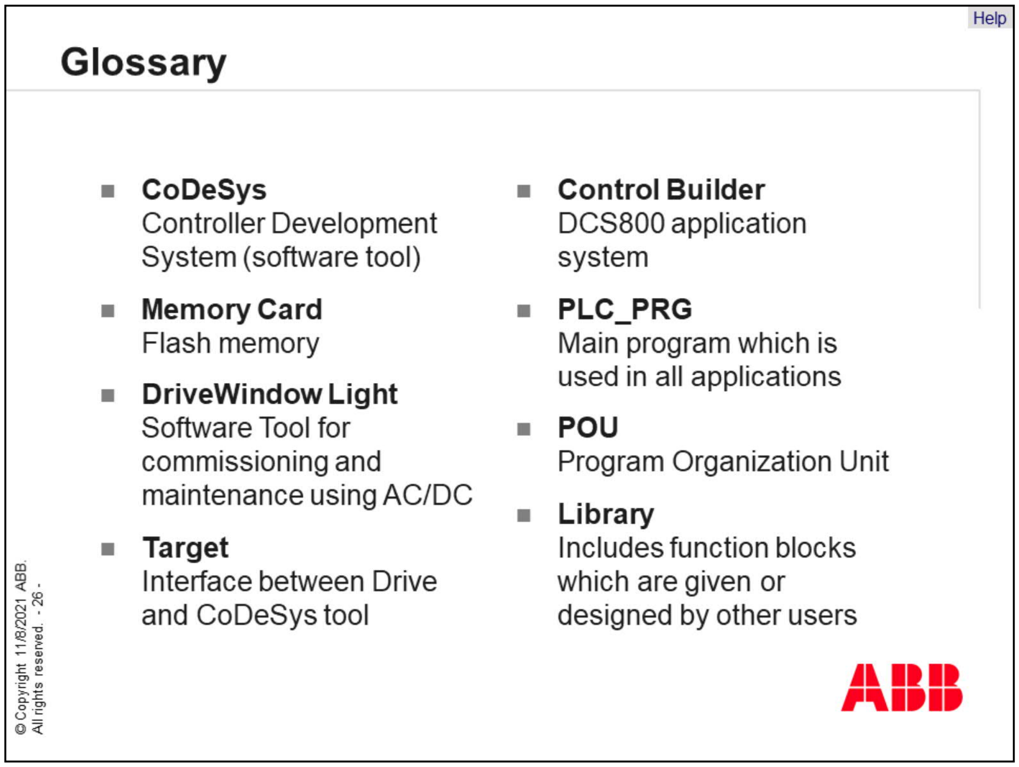
Help

Additional information

- Links to related information
  - 3S-software.com
  - DC-Drive-News (Intranet, only ABB internal!)
- Additional references

– Application Manual	(3ADW 000199)
– Firmware Manual	(3ADW 000193)
– Hardware Manual	(3ADW 000194)
– Training Material	

ABB



Help

Glossary

- **CoDeSys**
  Controller Development System (software tool)
- **Memory Card**
  Flash memory
- **DriveWindow Light**
  Software Tool for commissioning and maintenance using AC/DC
- **Target**
  Interface between Drive and CoDeSys tool
- **Control Builder**
  DCS800 application system
- **PLC_PRG**
  Main program which is used in all applications
- **POU**
  Program Organization Unit
- **Library**
  Includes function blocks which are given or designed by other users

ABB

Power and productivity
for a better world™

Thank you for your attention. You may now go ahead and move on to the next unit.