

# Development Environment

## CP1131

V.2.0

User Handbook

A u t o m a t i o n   S y s t e m

**CAN**trol® //

Copyright © Berghof GmbH 1999

Reproduction and duplication of this document and utilisation and communication of its content is prohibited, unless with our express permission.  
All rights reserved.  
Damages will be payable in case of infringement.

#### Disclaimer

The content of this publication was checked for compliance with the hardware and software described. However, discrepancies may arise, therefore no liability is assumed regarding complete compliance. The information in this document will be checked regularly and all necessary corrections will be included in subsequent editions. Suggestions for improvements are always welcome.

Subject to technical changes.

#### Trademark

**CAN**trol® // is a registered trademark of Berghof Labor- and Automationstechnik GmbH

Ident. No.: 2800620

You can reach us at our headquarters at:

Berghof Labor- und Automationstechnik GmbH  
Automation Technology Business Line  
Harretstr. 1 D-72800 Eningen / Germany  
Telefon: +49 (0) 71 21 / 8 94-0  
Telefax: +49 (0) 71 21 / 89 41 00  
<http://www.cantrol.de>  
e-mail: [info@berghof.com](mailto:info@berghof.com)

The Berghof Labor- und Automationstechnik GmbH is DIN EN ISO 9001 certified

### Update

Version	Date	Subject
2.0	3-2000	First English Version / Translation of german version 2.0

blank page

**Contents**

**A BRIEF OVERVIEW OF CP1131 .....9**

What is CP1131? ..... 9

Overview of the functionality of CP1131 ..... 9

**WHAT’S WHAT IN CP1131 ..... 11**

Components of a project ..... 11

The languages ..... 19

    Instruction List (IL)..... 19

    Structured Text (ST)..... 21

    Sequential Function Chart (SFC)..... 28

    Function Block Diagram (FBD) ..... 32

    Ladder Diagram (LD)..... 32

Debugging, online functionalities ..... 35

The standard ..... 36

**LET’S WRITE A SHORT PROGRAM.....37**

Controlling a set of traffic lights..... 37

Visualising a set of traffic lights..... 49

**A DETAILED LOOK AT COMPONENTS.....53**

The main window ..... 53

Options ..... 56

Managing projects ..... 65

Objects: creating, deleting, etc..... 78

General editing functions ..... 86

General Online Functions..... 92

Arrange windows ..... 99

The Help Tool ..... 100

**THE EDITORS IN CP1131 .....105**

The Declaration Editor ..... 105

The Text Editors ..... 113

    The Instruction List Editor ..... 117

    The Structured Text editor ..... 118

<b>The Graphical Editors</b> .....	<b>119</b>
The Function Block Diagram Editor .....	120
The Ladder Diagram Editor .....	127
The Sequential Function Chart Editor .....	133
<b>THE RESOURCES</b> .....	<b>143</b>
<b>Overview of the resources</b> .....	<b>143</b>
<b>Global variables</b> .....	<b>143</b>
Global variables .....	144
Variable configuration .....	145
Document template .....	146
<b>PLC Configuration</b> .....	<b>147</b>
<b>Task Configuration</b> .....	<b>149</b>
<b>Trace recording</b> .....	<b>153</b>
<b>Watch and Receipt Manager</b> .....	<b>158</b>
<b>LIBRARY MANAGEMENT</b> .....	<b>163</b>
<b>VISUALIZATION</b> .....	<b>165</b>
<b>Creating a Visualization</b> .....	<b>165</b>
<b>Inserting Visualization Elements</b> .....	<b>166</b>
<b>Using Visualization Elements</b> .....	<b>167</b>
<b>Configuring Visualization Elements</b> .....	<b>169</b>
<b>Other Functions for Visualization Elements</b> .....	<b>176</b>
<b>DDE INTERFACE</b> .....	<b>179</b>
<b>APPENDIX A: KEYBOARD OPERATION</b> .....	<b>181</b>
<b>Operation</b> .....	<b>181</b>
<b>Key Combinations</b> .....	<b>181</b>
<b>APPENDIX B: THE DATA TYPES</b> .....	<b>185</b>
<b>Standard Data Types</b> .....	<b>185</b>
<b>Defined Data Types</b> .....	<b>187</b>

**APPENDIX C: THE IEC OPERATORS .....191**

The IEC Operators..... 191

Arithmetic Operators ..... 191

Bitstring Operators ..... 193

Bit Shift Operators ..... 195

Selection Operators ..... 196

Comparison Operators ..... 198

Address Operators..... 201

Call Operator ..... 201

**APPENDIX D: ELEMENTS IN THE STANDARD LIBRARY .....203**

Type Conversion Functions ..... 203

Numeric Functions..... 206

String Functions..... 207

Bistable Function Blocks..... 210

Edge Detection ..... 211

Counters ..... 212

Timer..... 214

**APPENDIX E: OPERANDS IN CP1131 .....216**

Constants ..... 216

Variables..... 218

Addresses..... 219

Functions..... 220

**APPENDIX F: TRANSLATION ERRORS .....221**

blank page

## A Brief Overview of CP1131

### What is CP1131?

**CP1131** provides the SPC programmer with a simple introduction to the powerful IEC language aids. Its use of editors and debugging functions is modelled on the advanced development environments of higher programming languages (such as Visual C++).

## Overview of the Functionality of CP1131

### How is a Project Structured?

A project is stored in a file bearing the name of the project. The first block created in a new project is automatically assigned the name **PLC\_PRG**. This is where execution starts (similar to the main function in a C program), and other blocks (such as programs, function blocks and functions) can be called from here.

If you have defined a task configuration, it is not necessary to create a program with the name PLC\_PRG. Further information on this may be found in the section on task configuration.

**CP1131** differentiates between different types of objects within a project: blocks, data types, display elements (visualization) and resources.

The Object Organizer contains a list of all the objects in your project.

### How do I Create my Project?

You must first configure your controller in order to be able to check that the addresses used in the project are correct.

You can then create the blocks required for your particular task.

Now you can program the blocks you require in the desired languages.

After completion of programming, you can compile the project and correct any errors displayed.

### **How Can I Test my Project?**

When all errors have been corrected, activate simulation, log into the simulated controller and 'load' your project to the controller. **CP1131** is now in online operation.

You can now open the window with your controller configuration and check that your project is running properly. To do this, allocate the inputs manually and monitor whether the outputs are set as required. You can also monitor the value sequence of the local variables in the blocks. You can configure the data records whose values you wish to view in the watch and receipt manager.

### **Debugging using CP1131**

If there is a programming error, you can set breakpoints. When execution stops at a breakpoint, you can look at the values of all project variables at this point. Step-by-step processing (single-step) allows you to check the logical correctness of your program.

### **Other Online Functionalities**

Another debugging function of **CP1131** allows you to set program variables and inputs/outputs to particular values. You can then use the program check to check which program lines have run. Trace recording offers you the option of tracing and displaying the value sequence of variables according to cycle over a longer period.

Once the project has been created and tested, it can be loaded to the hardware and tested here also. The same online functions are available here as in simulation.

### **Other Features of CP1131**

The entire project can be documented or exported to a text file at any time.

### **Summary**

**CP1131** is a complete development tool for programming your controller, allowing you considerable time savings in the creation of your applications.

## What's What in CP1131

To make it easier for you to start using **CP1131**, this chapter contains a list of the most important terms in connection with the product.

---

## Components of a Project

### Project

A project contains all the objects in a control program. A project is stored in a file bearing the name of the project. The following objects are part of the project:

blocks, data types, visualizations, resources and libraries.

### Block

Functions, function blocks and programs are all blocks.

Each block consists of a declaration part and a body. The body is written in one of the IEC programming languages IL, ST, SFC, FBD or LD.

**CP1131** supports all IEC standard blocks. If you want to use these blocks in your project, you must integrate the 'standard.lib' library into your project.

Blocks can call other blocks, but recursions are not permitted.

### Function

A function is a block that produces exactly one data element (which can in turn contain several elements, such as fields or structures) on execution. They may be called in text languages as an operator in expressions.

When declaring a function, it is important to remember that the function must have a type. A colon must be entered after the function name, followed by a type.

The following is an example of a correct function declaration:

```
FUNCTION Fct: INT
```

The function must also be assigned a result. This means that the function name is used in the same way as an output variable.

A function declaration begins with the keyword `FUNCTION` and ends with `END_FUNCTION`.

The following is an example in IL of a function that takes three input variables and returns as a result the product of the first two, divided by the third:

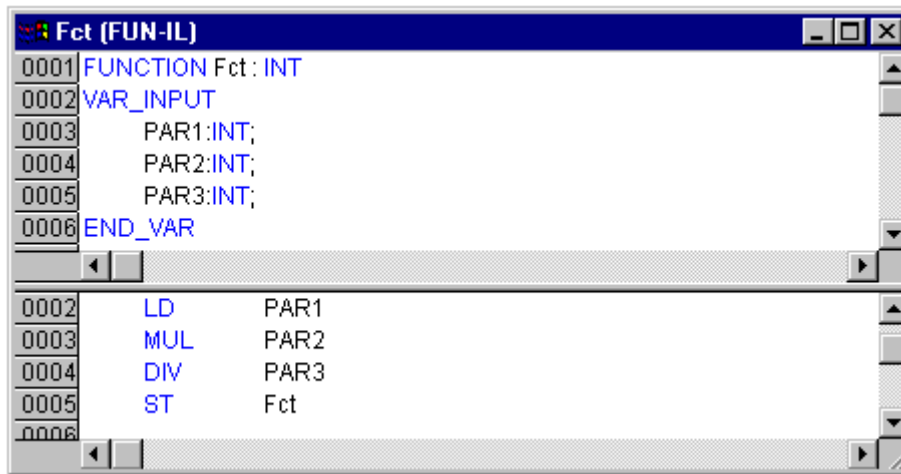


Figure 2\_1: Function

A function can be called in ST as an operand in expressions.

Functions do not have any internal states. This means that function calls using the same arguments (input parameters) always return the same value (output).

Examples of calls for the function described above:

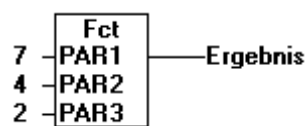
in IL:

```
LD 7
Fct2,4
ST Result
```

in ST:

```
Result := Fct(7, 2, 4);
```

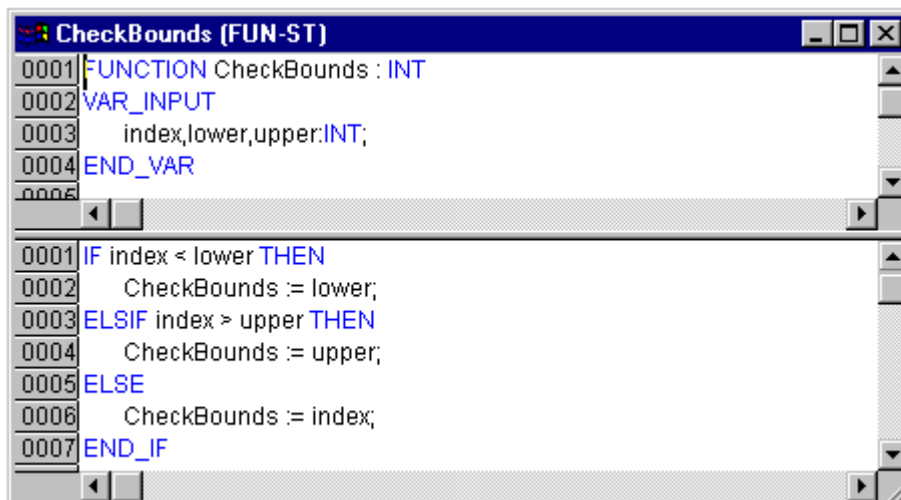
in FBD:



In SFC, a function can only be called within a step or transition.



**Note:** If you define a function with the name **CheckBounds** within your project, you can use it to check range overruns automatically. The name of the function is fixed, and the function must bear this name only. An example of an implementation of this function is outlined below:



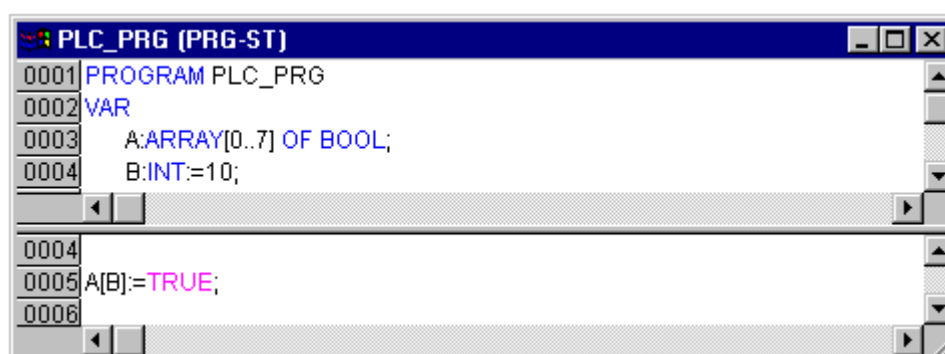
```

0001 FUNCTION CheckBounds : INT
0002 VAR_INPUT
0003     index,lower,upper:INT;
0004 END_VAR
0005
0001 IF index <= lower THEN
0002     CheckBounds := lower;
0003 ELSIF index > upper THEN
0004     CheckBounds := upper;
0005 ELSE
0006     CheckBounds := index;
0007 END_IF

```

Figure 2\_2: Example of an implementation of the CheckBounds function

The following sample program for testing the CheckBounds function intervenes outside the limits of a defined array. The CheckBounds function ensures that the TRUE value is assigned not at position A[10], but at the upper valid limit A[7]. The CheckBounds function therefore allows the correction of accesses outside array limits.



```

0001 PROGRAM PLC_PRG
0002 VAR
0003     A:ARRAY[0..7] OF BOOL;
0004     B:INT:=10;
0005
0004
0005 A[B]:=TRUE;
0006

```

Figure 2\_3: Test program for the CheckBounds function

## **Function Block**

A function block is a block that produces one or more values on execution. Unlike a function, a function block does not produce a return value.

A function block declaration begins with the keyword `FUNCTION_BLOCK` and ends with `END_FUNCTION_BLOCK`.

The following is an example in IL of a function block with two input variables and two output variables. One output is the product of the two inputs, and the other is a comparison of equality:

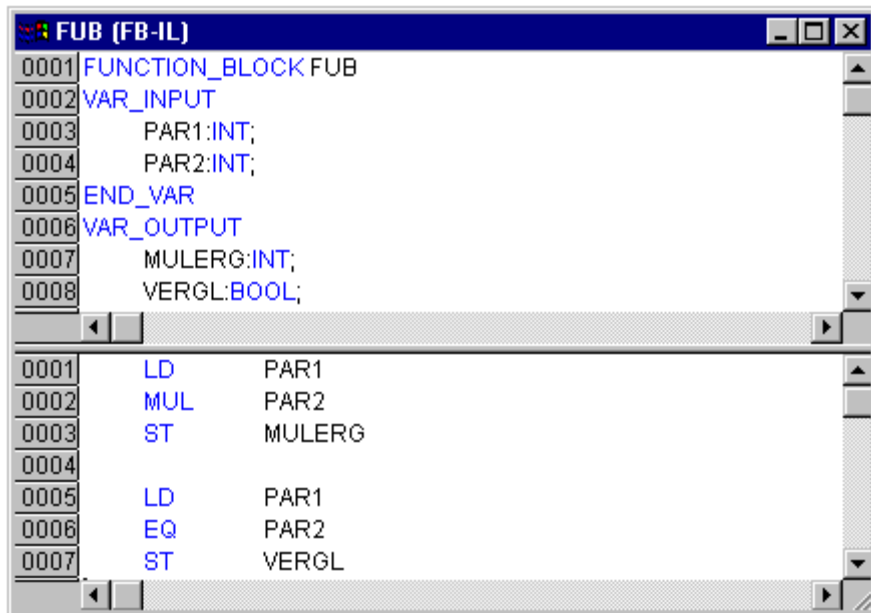


Figure 2\_4: Function block

## **Function Block Instances**

It is possible to create duplicates of a function block, known as *instances* (copies). Each instance has a corresponding identifier (the instance name), and a data structure containing its inputs, outputs and internal variables. Like variables, instances are declared locally or globally, with the name of the function block being specified as the type of an identifier.

Example of an instance of the function block FUB with the name INSTANCE:

```
INSTANCE: FUB;
```

Function blocks are always called using the instances described above.

The input and output parameters can only be accessed from outside an instance of a function block, i.e. the internal variables of the function block remain invisible to the user of the function block.

Example of access to an input variable:

The function block fb has an input variable in1 of the type int.

```
PROGRAM prog
VAR
  inst1:fb;
END_VAR
LD 17
ST  inst1.in1
CAL  inst1
END_PROGRAM
```

The declaration parts of function blocks and programs can contain instance declarations. Instance declarations are not permitted within functions.

Access to the instance of a function block is limited to the block in which it was 'instantiated', unless it has been declared globally.

The instance name of a function block instance can be used as the input of a function or function block.



**Note:** All values are retained from one execution of the function block to the next. This is why calls of a function block with the same arguments do not always return the same output values.

### Calling a Function Block

It is possible to use the variables of the function block by specifying the instance name, followed by a dot and then the variable name.

If you wish to set the input parameters during a call, this can be done in the textual languages IL and ST by assigning the parameter values in brackets after the instance name of the function block (assignment using ":= " as in the initialisation of variables at the declaration point).

The following are examples of calling the function block FUB described above. The multiplication result is stored in the RES variables, and the result of the comparison is stored in QUAD. An instance of FUB with the name INSTANCE is then declared:

The function block is implemented in IL as follows:

```

ILcall (PRG-IL)
0001 PROGRAM ILcall
0002 VAR
0003     QUAD:    BOOL;
0004     INSTANZ: FUB;
0005     ERG:     INT:=0;
0006 END_VAR
0007
0001     CAL     INSTANZ(PAR1:=5,PAR2:=5)
0002
0003     LD     INSTANZ.VERGL
0004     ST     QUAD
0005
0006     LD     INSTANZ.MULERG
0007     ST     ERG
0008
    
```

Figure 2\_5: Calling a function block in IL

In the example below, the call is shown in ST, with the declaration part being the same as in IL:

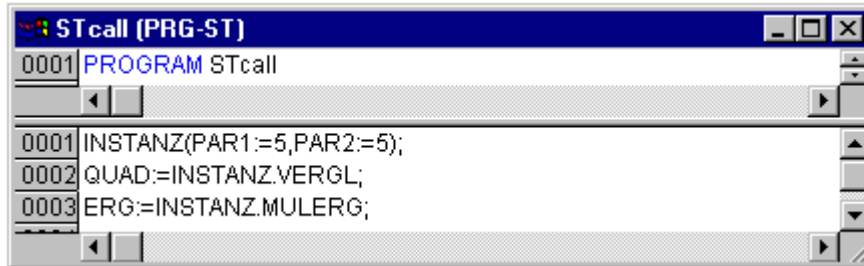


Figure 2\_6: Calling a function block in ST

In FBD it would appear as follows (declaration part the same as in IL):

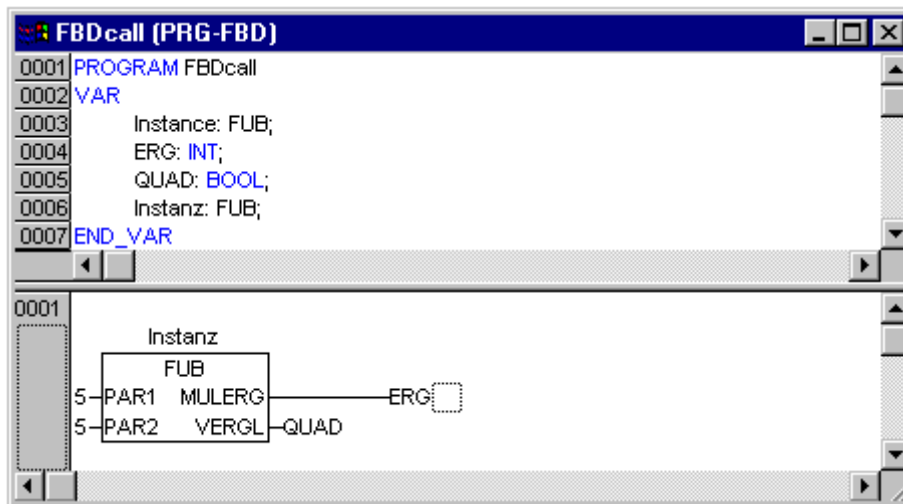


Figure 2\_7: Calling a function block in FBD

In SFC, function blocks can only be called in steps.

## Program

A program is a block that produces one or more values on execution. Programs are identified globally throughout the project. All values are retained from one execution of the program to the next.

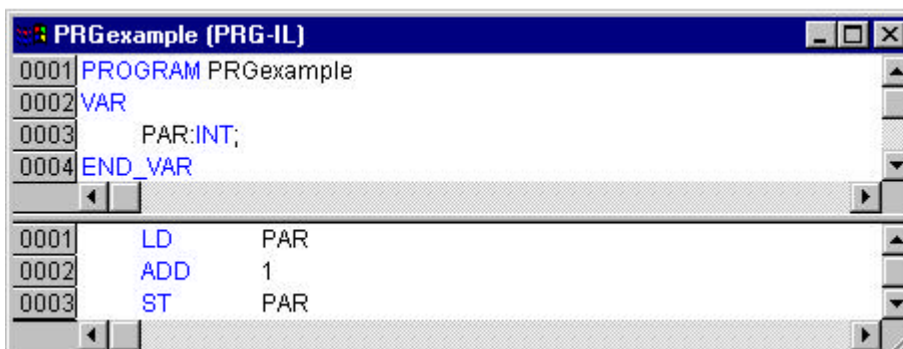


Figure 2\_8: Example of a program

Programs can be called from programs and function blocks. A program call is not permitted within a function. There are no instances of programs.

If a program is called from a block and the values of the program are changed in the process, these changes are retained the next time the program is called, even if the program is called from a different block.

This is different to calling a function block. Only the values in the relevant instance of a function block are changed. These changes are therefore only relevant if the same instance is called. A program declaration begins with the keyword PROGRAM and ends with END\_PROGRAM.

Examples of calls of the program described above:

In IL:

```
CAL PRGexample
LD PRGexample.PAR
ST RES
```

In ST:

```
PRGexample;
Result := PRGexample.PAR;
```

In FBD:



Example of a possible call sequence of PLC\_PRG:

```
LD 0
ST PRGexample.PAR (*PAR is preset with 0*)
CAL SLcall (*RES in SLcall results in 1*)
CAL STcall (*RES in STcall results in 2*)
CAL FBDcall (*RES in FBDcall results in 3*)
```

If the variable PAR of the program PRGexample is first initialised with 0 from a main program and programs are then called in sequence using the above program calls, the result Res in the programs will have the values 1, 2 and 3. If the sequence of calls is mixed up, the values of the respective result parameters will change accordingly.

## PLC PRG

PLC\_PRG is a special pre-defined block. Every project must contain this special program. This block is always called once in every control cycle.

When 'Project' 'Add object' is executed for the first time after creation of a new project, the presetting in the block dialog box is a block with the name PLC\_PRG, of the type Program. You must not change this presetting. If tasks have been defined, the project must not contain any PLC\_PRG, as in this case, the execution sequence is dependent on the task assignment.



**Caution: Do not delete or rename the PLC\_PRG block** (unless you are not using any task configuration; for further information, see the section on task configuration). PLC\_PRG is usually the main program in a single-task program.

## Resources

You need resources to configure and organise your project, and to trace variable values:

- global variables that can be used throughout the entire project
- control configuration to configure your hardware
- task configuration to control your program by means of tasks
- trace recording for the graphical recording of variable values
- watch and receipt managers for displaying and pre-setting variable values

For further information, see the chapter entitled 'The resources'.

## Libraries

It is possible to integrate a range of libraries into your project, whose blocks, data types and global variables can be used in the same way as self-defined ones. The 'standard.lib' library is provided as the default.

For further information, see the chapter entitled 'Library management'.

## Data types

In addition to the standard data types, users may also define their own data types. Structures, enumeration types and references may be created.

For further information, see the sections 'Standard data types' and 'Defined data types' in the appendix.

## Visualization

CP1131 offers a visualization option for viewing your project variables. Visualization allows you to draw geometric elements offline whose shape can then change online, depending on certain variable values.

For further information, see the chapter entitled 'Visualization'.

## The Languages

### Instruction List (IL)

Instruction List (IL) consists of a sequence of instructions. Each instruction begins on a new line and contains an operator and, depending on the type of operation, one or more operands, separated by commas. There may be a *label* identifier in front of the instruction, followed by a colon (:).

A comment must be the last element in a line. Blank lines can be inserted between instructions.

Example:

```
LD      17
ST      lint          (* comment *)
GE      5
JMPC    next
LD      idword
EQ      istruct.sdword
STN     test
next:
```

### Modifiers and Operators in IL

The following operators and modifiers can be used in IL.

Modifiers:

- C in JMP, CAL, RET: The instruction is only executed if the result of the preceding expression is TRUE.
- N in JMPC, CALC, RETC: The instruction is only executed if the result of the preceding expression is FALSE.
- N otherwise: negation of the operand (not of the accumulator)

The following table lists all operators in IL together with their possible modifiers and the respective meanings:

Operator	Modifiers	Meaning
LD	N	Set current result equal to the operand
ST	N	Save the current result at the operand position
S		Set the Boolean operand to TRUE if the current result is TRUE
R		Set the Boolean operand to FALSE if the current result is TRUE
AND	N,(	Bitwise AND
OR	N,(	Bitwise OR
XOR	N,(	Bitwise exclusive OR
ADD	(	Addition
SUB	(	Subtraction
MUL	(	Multiplication
DIV	(	Division

GT	(	>
GE	(	>=
EQ	(	=
NE	(	<>
LE	(	<=
LT	(	<
JMP	CN	Jump to label
CAL	CN	Call function block
RET	CN	Return from calling a function block
)		Evaluate reset operation

A list of all IEC operators is found in the appendix.

Example of an IL program using certain modifiers:

```

LD      TRUE      (*Load TRUE in the accumulator*)
ANDN    BOOL1     (*Execute AND with the negated value of the BOOL1
                  variable*)
JMPC    label     (*If the result was TRUE, jump to the label "label"*)

LDN     BOOL2     (*Save the negated value of *)
ST      RES       (*BOOL2 in RES*)
label:
LD      BOOL2     (*Save the value of *)
ST      RES       (*BOOL2 in RES*)

```

In IL, it is also possible to set brackets after an operation. The value of the bracket is then seen as the operand.

For example:

```

LD      2
MUL     2
ADD     3
ST      Res

```

Here, the value of Res is 7. However, if brackets are set:

```

LD      2
MUL(    2
ADD     3
)       ST      Res

```

Here, the value of Res is 10, as the operation MUL is only evaluated when ")" occurs; the operand for MUL is then calculated as 5.

**Structured Text (ST)**

Structured Text consists of a sequence of instructions which can be executed conditionally as in high-level languages (“IF..THEN..ELSE”) or in loops (WHILE..DO).

Example:

```
IF value < 7 THEN
    WHILE value < 8 DO
        value := value + 1;
    END_WHILE;
END_IF;
```

**Expressions**

An *expression* is a construct that returns a value after being evaluated.

Expressions are composed of operators and operands. An operand could be a constant, a variable, a function call or another expression.

**Evaluation of Expressions**

An expression is evaluated by processing the operators according to certain *linking rules*. The operator with the strongest link is processed first, followed by the operator with the next-strongest link, and so on, until all operators have been processed.

Operators with equal linking strengths are processed from left to right.

The following table lists the ST operators in order of linking strength:

Operation	Symbol	Linking strength
Put in brackets	(Expression)	Strongest link
Function call	Function name (list of parameters)	
Exponentiate	**	
Negate	-	
Complementation	NOT	
Multiply	*	
Divide	/	
Modulo	MOD	
Add	+	
Subtract	-	
Compare	<,>,<=,>=	
Equal to	=	
Not equal to	<>	
Boolean AND	AND	
Boolean XOR	XOR	
Boolean OR	OR	Weakest link

The following are the instructions in ST, listed in a table together with an example:

Instruction type	Example
Assignment	A:=B; CV := CV + 1; C:=SIN(X);
Call of a function block and use of the FB output	CMD_TMR(IN := %IX5, PT := 300); A:=CMD_TMR.Q
RETURN	RETURN;
IF	D:=B*B; IF D<0.0 THEN C:=A; ELSIF D=0.0 THEN C:=B; ELSE C:=D; END_IF;
CASE	CASE INT1 OF 1:    BOOL1 := TRUE; 2:    BOOL2 := TRUE; ELSE BOOL1 := FALSE; BOOL2 := FALSE; END_CASE;
FOR	J:=101; FOR I:=1 TO 100 BY 2 DO IF ARR[I] = 70 THEN J:=I; EXIT; END_IF; END_FOR;
WHILE	J:=1; WHILE J<= 100 AND ARR[J] <> 70 DO J:=J+2; END_WHILE;
REPEAT	J:=-1; REPEAT J:=J+2; UNTIL J= 101 OR ARR[J] = 70 END_REPEAT;
EXIT	EXIT;
Empty instruction	;

### **Instructions in Structured Text**

As the name implies, Structured Text is designed for structured programming, i.e. ST offers predefined structures for the programming of certain frequently-used constructs, such as loops.

Its advantages are a lower incidence of errors and greater program clarity. To illustrate this, we will compare two program sequences in IL and ST with the same meanings:

A loop to calculate powers of two in **IL**:

```
loop:
LD      Counter
EQ      0
JMPC   end

LD      Var1
MUL    2
ST      Var1

LD      Counter
SUB    1
ST      Counter
JMP    loop

end:
LD      Var1
ST      Res
```

The following is the same loop programmed in **ST**:

```
WHILE counter<>0 DO
  Var1:=Var1*2;
  Counter:=Counter-1;
END_WHILE

Res:=Var1;
```

It is obvious that the loop is not only shorter to program in ST, but also much easier to read, particularly if there are nested loops in larger constructs.

The various structures in ST have the following meanings:

### **Assignment Operators**

On the left-hand side of an assignment there is an operand (variable, address), which is assigned to the value of the expression on the right-hand side with the assignment operator :=

Example:

```
Var1 := Var2 * 10;
```

When this line is executed, Var1 has the value of Var2 to the power of 10.

**Calling function blocks in ST**

A function block is called in ST by writing the name of the function block instance and then assigning the required parameter values in brackets. In the following example, a timer is called with assignments for the parameters IN and PT. The result variable Q is then assigned to the variable A. The result variable is addressed as in IL, with the name of the function block followed by a dot and the name of the variable:

```
CMD_TMR(IN := %IX5, PT := 300);
A:=CMD_TMR.Q
```

**RETURN instruction**

The RETURN instruction can be used to end a function, e.g. dependent on a condition.

**IF instruction**

The IF instruction allows you to check a condition and execute an instruction dependent on this condition.

Syntax:

```
IF <Boolean_expression1> THEN
  <IF_instructions>
{ELSIF <Boolean_expression2> THEN
  <ELSIF_instructions1>
.
.
ELSIF <Boolean_expression n> THEN
  <ELSIF_instructions n-1>
ELSE
  <ELSE_instructions>}
END_IF;
```

The part in braces {} is optional. If the result of <Boolean\_expression1> is TRUE, only the <IF\_instructions> are executed and the other instructions are ignored. Otherwise, the Boolean expressions are evaluated in sequence, beginning with <Boolean\_expression2> until one of the expressions produces a TRUE. After that, only the instructions after this Boolean expression and before the next ELSE or ELSIF are evaluated.

If none of the Boolean expressions produces a TRUE, only the <ELSE\_instructions> are evaluated.

Example:

```
IF temp<17
THEN  heating_on := TRUE;
ELSE  heating_on := FALSE;
END_IF;
```

In this case, the heating is turned on if the temperature falls below 17 degrees. Otherwise, it remains turned off.

## **CASE Instruction**

The CASE instruction allows the combination of several conditional instructions in one construct using the same conditional variables.

Syntax:

```
CASE <Var1> OF
<value 1>:    <instruction 1>
<value 2>:    <instruction 2>
...
<value n>:    <instruction n>
ELSE <ELSE instruction>
END_CASE;
```

A CASE instruction is processed according to the following scheme:

- If the variable in <Var1> has the value <value i>, then the instruction <instruction i> is executed.
- If <Var 1> does not have any of the values specified, then the <ELSE instruction> is executed.
- If the same instruction is to be executed for several variable values, these values may be written one after the other, separated by commas, in order to make them conditional to the same instruction.

Example:

```
CASE INT1 OF
1, 5:  BOOL1 := TRUE;
      BOOL3 := FALSE;
2:     BOOL2 := FALSE;
      BOOL3 := TRUE;
ELSE
      BOOL1 := NOT BOOL1;
      BOOL2 := BOOL1 OR BOOL2;
END_CASE;
```

## **FOR Loop**

The FOR loop allows you to program repeated processes.

Syntax:

```
INT_Var :INT;

FOR <INT_Var> := <INIT_VALUE> TO <END_VALUE> {BY <step_size>} DO
  <instructions>
END_FOR;
```

The part in braces {} is optional.

The <instructions> continue to be executed for as long as the counter <INT\_Var> remains smaller than the <END\_VALUE>. This is checked before the execution of the <instructions>, so the <instructions> are never executed if <INIT\_VALUE> is greater than <END\_VALUE>.

Every time that <instructions> is executed, <INT\_Var> is increased by <step\_size>. The step size can have any integer value. If the value is missing, it is set to 1. The loop must therefore terminate, as <INT\_Var> can only increase.

Example:

```
FOR counter:=1 TO 5 BY 1 DO
  Var1:=Var1*2;
END_FOR;
Res:=Var1;
```

Let us assume that the variable Var1 has been preset with the value 1. After the FOR loop, it will have the value 32.

## **WHILE Loop**

The WHILE loop can be used in the same way as the FOR loop, except that the termination condition can be any Boolean expression. This means that a condition is specified, and if it applies, the loop is executed.

Syntax:

```
WHILE <Boolean_expression > DO
  <instructions>
END_WHILE;
```

The <instructions> are executed repeatedly for as long as the <Boolean\_expression> produces TRUE. If <Boolean\_expression> is already FALSE in the first evaluation, then the <instructions> are not executed at all. If <Boolean\_expression> never has the value FALSE, then the <instructions> are repeated endlessly, causing a runtime error.



**Note:** The programmer must ensure that there are no endless loops by changing the condition in the instruction part of the loop, e.g. by increasing or decreasing a counter.

Example:

```
WHILE counter<>0 DO
  Var1 := Var1*2;
  Counter := counter-1;
END_WHILE
```

The WHILE and REPEAT loops are more powerful than the FOR loop to a certain extent, as it is not necessary to know the number of times the loop will be executed before executing the loop. In some cases, therefore, only these two loop types will be used. However, if the number of loop executions is clear, then a FOR loop is preferable, as it does not permit any endless loops.

## REPEAT Loop

The REPEAT loop differs from the WHILE loop in that the termination condition is not checked until after execution of the loop. This means that the loop must be executed at least once, regardless of what the termination condition is.

Syntax:

```
REPEAT
  <instructions>
UNTIL <Boolean_expression>
END_REPEAT;
```

The <instructions> are executed until <Boolean\_expression> produces TRUE.

If <Boolean\_expression> produces TRUE in the first evaluation, then <instructions> is executed once. If <Boolean\_expression> never has the value TRUE, then the <instructions> are repeated endlessly, causing a runtime error.



**Note:** The programmer must ensure that there are no endless loops by changing the condition in the instruction part of the loop, e.g. by increasing or decreasing a counter.

Example:

```
REPEAT
  Var1 := Var1*2;
  Counter := Counter-1;
UNTIL
  Counter=0
END_REPEAT
```

## EXIT Instruction

If the EXIT instruction occurs in a FOR, WHILE or REPEAT loop, the innermost loop is terminated, regardless of the termination condition.

## Sequential Function Chart (SFC)

Sequential Function Chart is a graphical language which allows you to describe the temporal sequence of different actions within a program.

Example of a network in Sequential Function Chart:

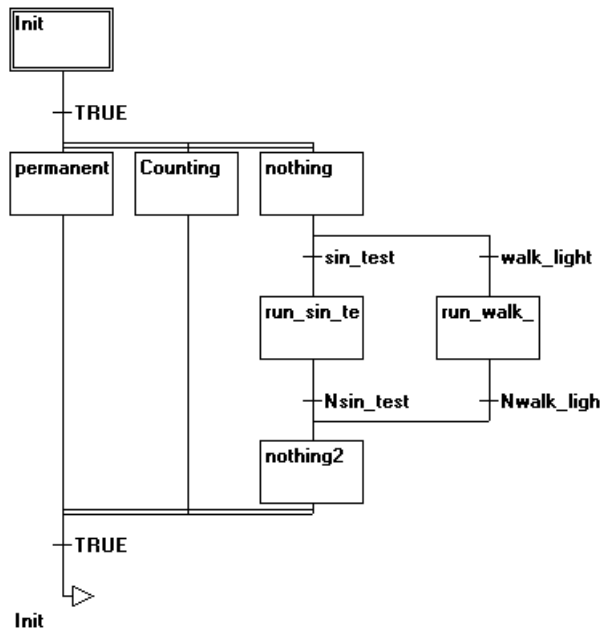


Figure 2\_9: Network in SFC

## Step

A block written in Sequential Function Chart consists of a sequence of steps linked to each other by means of directed links (transitions).

There are two types of steps.

- The simplified form consists of one action and one flag which shows whether the step is active. If the action in a step is implemented, a small triangle appears in the upper right-hand corner of the step.
- An IEC step consists of one flag and one or more assigned actions. The associated actions appear to the right of the step. We will return to this later.

## Action

An action can contain a sequence of instructions in IL or in ST, a quantity of networks in FBD or LD, or an execution structure.

In the simplified steps, an action is always linked to its step. To edit an action, double-click on the step to which the action belongs using the mouse, or select the step and execute the menu command '**Tools**' '**Zoom Action/Transition**'.

Actions for IEC steps are found in the Object Organizer, directly under their SFC block. They are loaded in their editor by double-clicking with the mouse or by pressing the <Enter> key. New actions can be generated using '**Project**' '**Add action**'.

## Input or Output Action

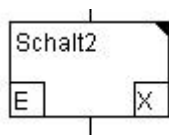
An input and an output action can be added to a step. An input action is only executed once, immediately after the step becomes active. An output action is only executed once, before the step is deactivated.

A step with an input action is identified by an 'E' in the lower left-hand corner, while an output action is identified by an 'X' in the lower right-hand corner.

Input and output actions can be implemented in any language. To edit an input or output action, double-click on the relevant corner in the step.

Input and output actions can only be defined for a simplified step, not for an IEC step.

Example of a step with input and output action:



## Transition / Transition Condition

Transitions are found between each step.

A transition condition can be a Boolean variable, an address, a constant or a sequence of instructions with a Boolean result in any language.

**Active Step**

After calling the SFC block, the action belonging to the initial step (with a double border) is executed first. A step whose action is being executed is considered to be active. If the step is active, the corresponding action is executed once per cycle. In online mode, active steps are displayed in blue.

Each step has a flag which saves the status of the step. The step flag (active or inactive status of the step) is represented by the logical value of a Boolean structure element <StepName>.x. This Boolean variable has the value TRUE when the corresponding step is active, and FALSE when it is inactive. This value is declared implicitly, and can be used in every action and transition of the SFC block.

All actions belonging to active steps are executed in a control cycle. After that, the successor steps to the active steps become active if the transition conditions of the successor steps are TRUE. The steps which are active now are not executed until the next cycle.

**IEC Step**

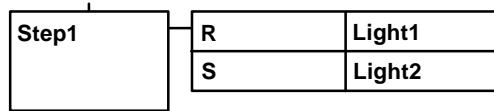
In addition to the simplified steps, the standard-conformant IEC steps are also available in SFC.

Any number of actions can be assigned to an IEC step. The actions of IEC steps are separate from the steps, and can be used several times within their blocks. To do this, they must be associated with the individual steps using the command '**Extra**' '**Associate action**'.

In addition to actions, Boolean variables can also be assigned to steps. The actions and Boolean variables are activated and deactivated by means of qualifiers, sometimes with time delays. As one action may still be active when the next step is being processed, using the qualifier S (Set), for example, simultaneous runs can be achieved.

The associated actions of an IEC step are displayed to the right of the step in a box with two fields. The left-hand field contains the qualifier together with any time constants, and the right-hand box contains the action name.

Example of an IEC step with two actions:



To make it easier to follow the processes, all active actions in online operation are displayed in blue, the same as the active steps. After each cycle, a check is run to determine which actions are active.

Whether the newly-added step is an IEC step depends on whether the menu command '**Extras**' '**Use IEC steps**' is selected.

The actions are found directly under their SFC block in the Object Organizer. New actions can be generated using '**Project**' '**Add action**'.

In order to be able to use IEC steps, you must incorporate the special SFC library lecsfc.lib into your project.

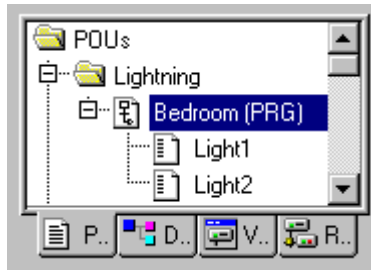


Figure 2\_10: SFC block with actions in the Object Organizer

**Qualifier**

The following qualifiers are available for associating actions to IEC steps:

N	Non-stored	The action is active as long as the step
R	overriding Reset	The action is deactivated
S	Set (Stored)	The action is activated and remains active until a reset
L	time Limited	The action is activated for a set time
D	time Delayed	The action is active after a set time, provided that the step is still active
P	Pulse	The action is executed once only, if the step is active
SD	Stored and time De- layed	The action is activated after a set time and remains active until a reset
DS	Delayed and Stored	The action is activated after a set time, provided that the step is still active, and remains active until a reset
IL	Stored and time Limited	The action is activated for a set period of time

**Alternative Branch**

Two or more branches in SFC can be defined as alternative branches. Each alternative branch must begin and end with a transition. Alternative branches can contain parallel branches and other alternative branches. An alternative branch begins with a horizontal line (alternative start) and ends with a horizontal line (alternative end) or with a jump.

If the step preceding the alternative start line is still active, the first transition of each alternative branch is evaluated from left to right. The first transition from the left whose transition condition has the value TRUE is opened and the successor steps are activated (see active step).

**Parallel Branch**

Two or more branches in SFC can be defined as parallel branches. Each parallel branch must begin and end with a step. Parallel branches can contain alternative branches or other parallel branches. A parallel branch begins with a double line (start of parallel) and ends with a double line (end of parallel) or a jump.

If the step preceding the parallel start line is still active and the transition condition after this step has the value TRUE, the first steps of all parallel branches become active (see active step). These branches are now processed together in parallel. The step after the parallel end line becomes active when all preceding steps are active, and the transition condition before this step returns the value TRUE.

**Jump**

A jump is a link to a step whose name is specified under the jump symbol. Jumps are required because it is not permitted to create links leading upwards or crossing over each other.

**Function Block Diagram (FBD)**

Function Block Diagram is a graphical programming language. It works with a list of networks, with each network containing a structure which in turn represents a logical or arithmetic expression, the call of a function block, a jump or a return instruction.

The following is an example of a network in Function Block Diagram as it could typically appear in CP1131:

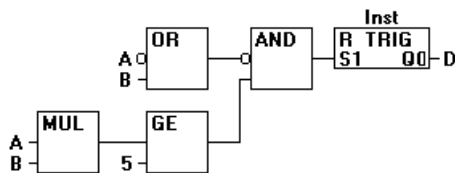


Figure 2\_11: Network in Function Block Diagram

**Ladder Diagram (LD)**

Ladder Diagram is also a graphical programming language which operates on a similar principle to that of an electrical circuit.

On the one hand, Ladder Diagram is suitable for the construction of logical circuit mechanisms, but on the other hand, it is also possible to create networks, as in FBD. For this reason, LD is particularly suitable for controlling the calling of other blocks. We will return to this topic later.

Ladder Diagram consists of a sequence of networks. A network is limited to the left and right-hand sides by a left and right-hand vertical power line. In between is a circuit plan constructed from contacts, coils and connection lines.

The left-hand side of each network consists of a series of contacts which relay the status "OPEN" or "CLOSED" from left to right. These states correspond to the Boolean values TRUE and FALSE. There is a Boolean variable for each contact. If this variable is TRUE, the status is relayed from left to right via the connection line. Otherwise, the right-hand connection receives the value CLOSED.

The following is an example of a network in Ladder Diagram, as it could appear in CP1131:

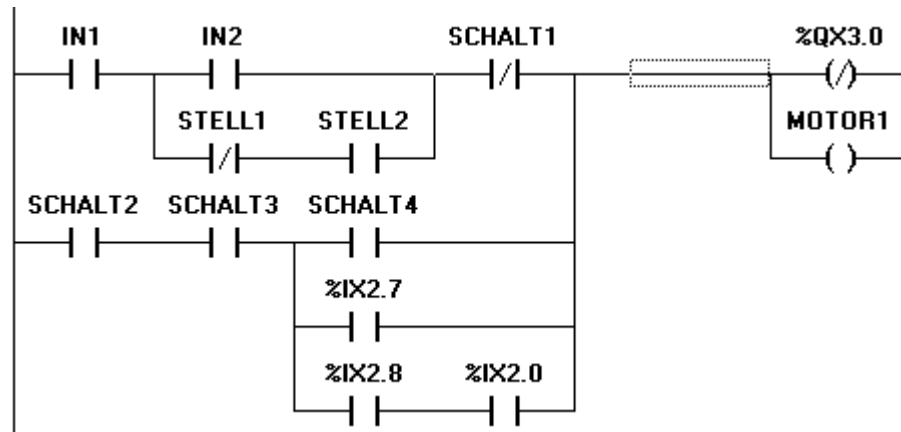


Figure 2\_12: Network in Ladder Diagram consisting of contacts and coils

### Contact

The left-hand side of each network in LD consists of a network of *contacts* (contacts are represented by two parallel lines: | |), which relay the status “open” or “closed” from left to right.

These states correspond to the Boolean values TRUE and FALSE. There is a Boolean variable for each contact. If this variable is TRUE, the state is relayed from left to right via the connection line. Otherwise, the right-hand connection receives the value “closed”.

Contacts can be switched in parallel. To do this, *one* of the parallel branches must relay the value “Open”, so that the parallel branch relays the value “Open”. Alternatively, the contacts are switched in sequence. To do this, *all* contacts must transfer the status “Open”, so that the last contact relays the status “Open”. This corresponds to an electrical parallel or series connection.

A contact can also be negated. This is indicated by the forward slash in the contact symbol: |/. The value of the line is relayed if the variable is FALSE.

### Coil

The right-hand side of a network in LD contains any number of coils, represented by brackets:( ). These can only be switched in parallel. A coil relays the value of the connections from left to right, and copies it to a corresponding Boolean variable. The input line can have the value OPEN (which corresponds to the Boolean value TRUE) or CLOSED (which corresponds to FALSE).

Contacts and coils can also be negated (in this example, the contact SCHALT1 and the coil %QX3.0 are negated). If a coil is negated (as indicated by a forward slash in the coil symbol: (/)), it copies the negated value to the corresponding Boolean variable. If a contact is negated, it only switches through if the corresponding Boolean variable is FALSE.

**Function Blocks in Ladder Diagram**

In addition to contacts and coils, you can also enter function blocks and programs. These must have one input and one output in the network with Boolean values, and can be used in the same locations as contacts, i.e. on the left-hand side of the LD network.

**Set/Reset Coils**

Coils can also be defined as set or reset coils. A set coil (recognisable by an 'S' in the coil symbol: **(S)**) never overwrites the value TRUE in the corresponding Boolean variables. This means that if the variable is set once to TRUE, it remains as TRUE.

A reset coil (recognisable by an 'R' in the coil symbol: **(R)**) never overwrites the value FALSE in the corresponding Boolean variables. This means that if the variable is set once to FALSE, it remains as FALSE.

**LD as FBD**

When using LD, it often happens that you want to use the result of the contact circuit to control other blocks. You can then store the result in a global variable with the help of coils, and it is then used in another location. You can also integrate any call directly into your LD network. To do this, you must introduce a block with an EN input.

Such blocks are usually operands, functions, programs or function blocks which have an additional input labelled as EN. The EN input is always of the type BOOL and has the following meaning: the block with the EN input is evaluated if EN has the value TRUE.

An EN block is switched in parallel to the coils whereby the EN input is connected to the connection line between the contacts and the coils. If the information OPEN is transported over this line, this block is evaluated as normal.

Networks such as those in FBD can be created on the basis of such an EN block.

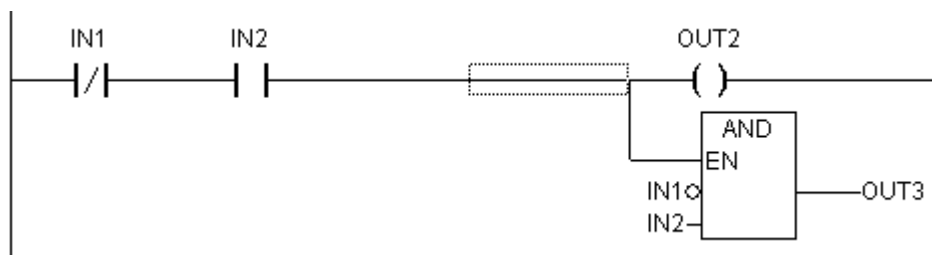


Figure 2\_13: Part of a LD network with an EN block

---

## **Debugging, Online Functionalities**

### **Trace recording**

Trace recording offers the option of recording the value sequence of variables, dependent on the 'trigger event'. This is the rising or falling edge of a previously-defined Boolean variable (the trigger variable). **CP1131** allows the recording of up to 20 variables. Up to 500 values of each variable can be recorded.

### **Debugging**

The debugging functions of **CP1131** make it easier to locate errors.

To allow debugging, the command '**Project**' '**Options**' must be executed, and the item 'Debugging' must be selected from the dialog box that appears below 'Compile options'.

### **Breakpoint**

A breakpoint is a location in a program at which processing is interrupted. This makes it possible to look at the values of variables at a particular part of the program.

Breakpoints can be set in all editors. In text editors, breakpoints are set at line numbers, in FBD and LD they are set at network numbers, and in SFC, they are set at steps.

### **Single Step**

Single step means:

- in IL: execute the program as far as the next CAL, LD or JMP command
- in ST: execute the next instruction
- in FBD, LD: execute the next network
- in SFC: execute the action to the next step

Step-by-step processing allows you to check the logical correctness of your program.

### **Single Cycle**

If single cycle is selected, processing is interrupted after each cycle.

### **Change Values Online**

During operation, variables can be set once to a particular value (write value) or they are rewritten with a particular value at the end of each cycle (force).

### **Monitoring**

The current values from the controller of the variable declarations visible on the screen are read and displayed on an ongoing basis.

**CP1131** monitors all variables visible on the screen in *online mode*. In addition, variables can be collected in the watch and receipt manager, where they are available for viewing at a glance.

### **Simulation**

In simulation, the control program generated is processed not in the control unit, but on the computer on which **CP1131** is also running. All online functions are available. This means that you can test the logical correctness of your program without using control hardware.

---

## **The Standard**

The IEC 1131-3 standard is an international standard for programming languages of programmable controllers.

The programming languages implemented in **CP1131** meet the requirements of this standard.

According to this standard, a program consists of the following elements:

- structures
- blocks
- global variables

Processing of a CP1131 program begins with the special block PLC\_PRG. The PLC\_PRG block can call other blocks.

## Let's Write a Short Program

### Controlling a Set of Traffic Lights

We will now write a short sample program. The project is a small traffic light system to control two sets of traffic lights at an intersection. Both sets have alternate red and green phases, and in order to help prevent accidents, they will also have amber and amber/red phases. The latter phases will be shorter than the former.

This example will show you how to represent time-dependent programs using the IEC 1131-3 languages, how to edit the various languages of the standard using **CP1131**, and how to link them without problems. Finally, you will also get to know and value simulation using **CP1131**. The project 'Lights.pro' can be found in the '**SAMPLE**' directory after successful installation.

### Generating Blocks

Starting is easy: simply start **CP1131** and select '**File**' '**New**'.

The name of the first block is already preset to PLC\_PRG in the first dialog box that appears. You should retain this name. The type of block must always be a program, as every project requires a program of this name. In our example, we will select Sequential Function Chart (SFC) as the language for this block.

We will now generate two more objects using the command '**Project**' '**Object**' '**Add**' from the menu bar or the context-sensitive menu (by clicking with the right-hand mouse button in the Object Organizer): one function block in the language Function Block Diagram (FBD) called LIGHTS, and one block called WAIT, also of the type function block, which we want to program in Instruction List (IL).

### What does LIGHTS do?

In the LIGHTS block, we will allocate the individual traffic light phases to the traffic lights, i.e. we will ensure that the red lamp comes on in the red phase and in the amber/red phase, that the amber lamp comes on in the amber and amber/red phases, etc.

### What does WAIT do?

In WAIT, we will program a simple timer which will receive the duration of the phase in milliseconds as input and return TRUE as output as soon as the time has elapsed.

### What does PLC PRG do?

Finally, PLC\_PRG will link everything together to ensure that the correct light comes on at the correct time, and for the correct duration.

### "LIGHTS" Declaration

We will first concentrate on the LIGHTS block. In the declaration editor, declare a variable with the name STATUS of the type INT as an input variable (between the keywords VAR\_INPUT and END\_VAR). STATUS will have five possible states, one for each of the traffic light phases green, amber, amber/red, red and off.

Our traffic light has four corresponding outputs, namely RED, AMBER, GREEN and OFF. When we declare these four variables, the declaration part of our function block LIGHTS will look like this:

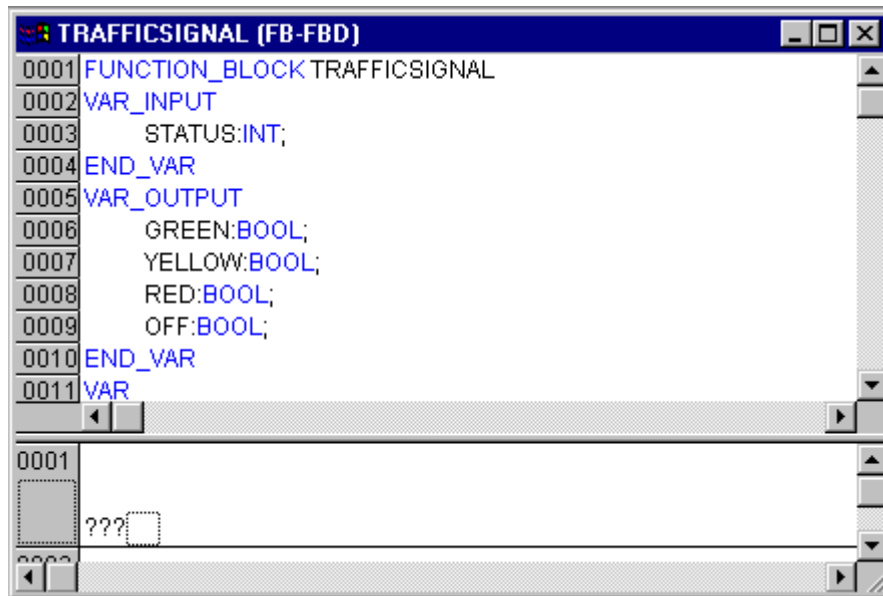
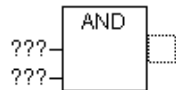


Figure 3\_1: LIGHTS function block, declaration part

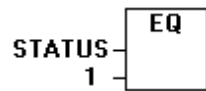
**“LIGHTS” Body**

We can now determine the values of the output variables from the input STATUS of the block. To do this, go to the body of the block. Click in the left-hand field beside the first network (the grey field with the number 1). You have now selected the first network. Now select the menu item **‘Insert’ ‘Operator’**.

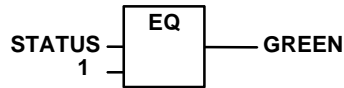
A box containing the operator AND and two inputs is inserted into the first network:



Click on the text AND with the mouse pointer and change the text to EQ. Select the three question marks at the top-most of the inputs and enter the variable STATUS. Then select the lower three question marks and overwrite them with the number 1. You will obtain the following network:



Now click on a point behind the EQ box. The output of the EQ operation is now selected. Execute **‘Insert’ ‘Assign’**. Change the three question marks ??? to GREEN. You have now created a network with the following design:



STATUS is compared with 1, and the result is assigned to GREEN. This network therefore switches to GREEN if the preset status value is 1.

We need three more networks for the other traffic light colors and for OFF. These are generated using the command **‘Insert’ ‘Network (after)’**. These networks should be set up as shown in the example. The finished block now looks like this:

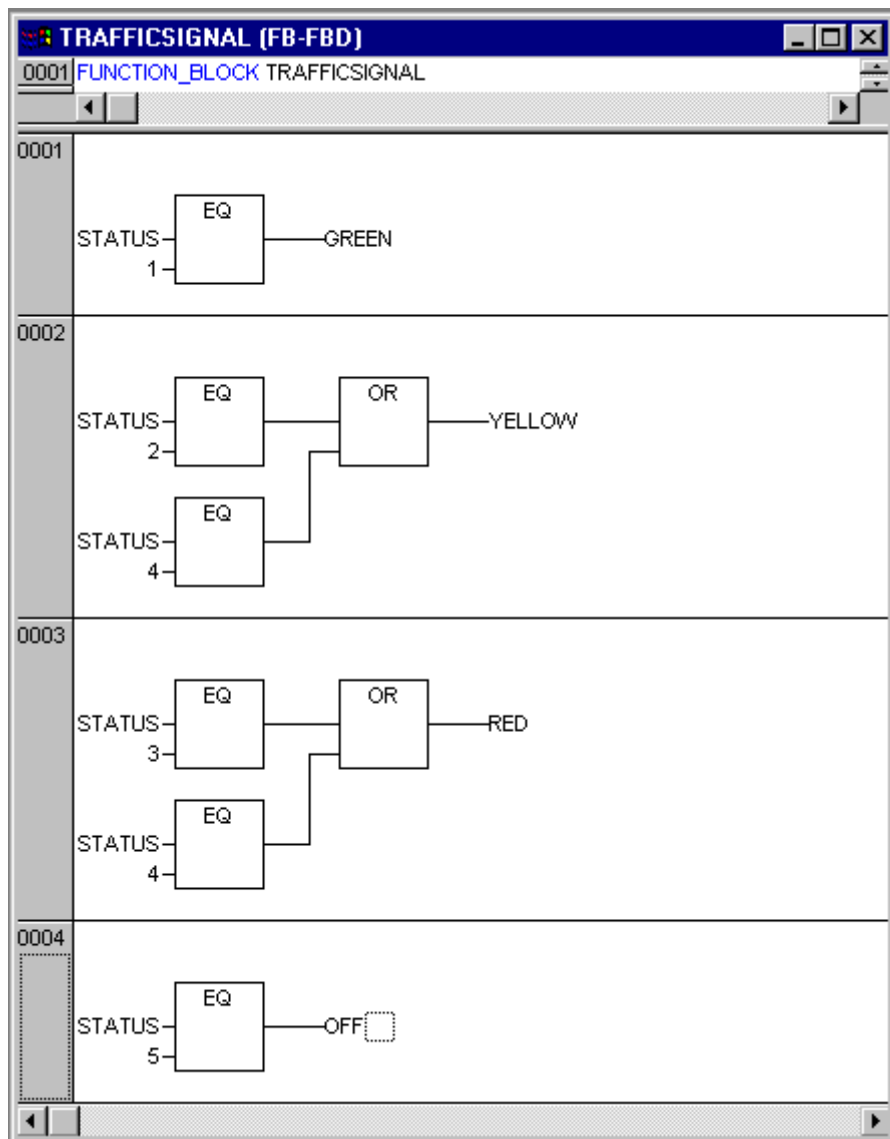


Figure 3\_2: LIGHTS function block, instruction part

To insert another operator in front of an operator, you must select the point where the input to which you want to attach the operator leads into the box.

Now execute **'Insert' 'Operator'**. The rest of the procedure for creating this network is the same as for the first network.

Our first block is now finished. LIGHTS will control the traffic light color required depending on the input of the STATUS value.

### Linking Standard.lib

For the timer in the WAIT block, we require a block from the standard library. Open the Library Manager using **'Window' 'Library Manager'**. Select **'Insert' 'Additional library'**. The dialog for opening files appears. Select 'standard.lib' from the list of libraries.

### "WAIT" Declaration

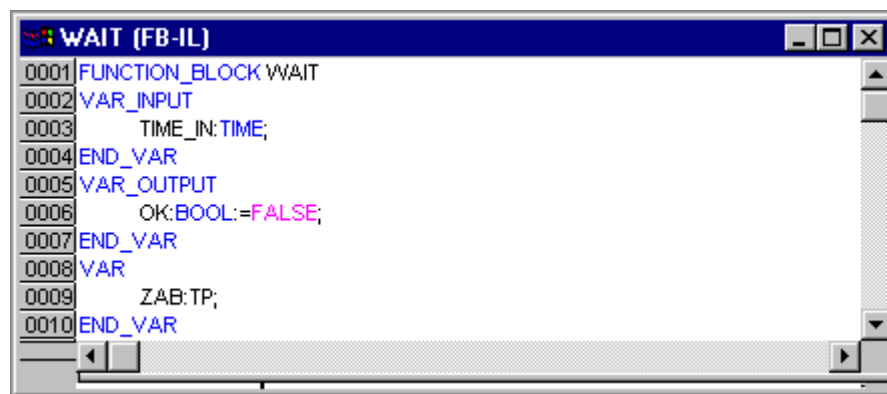
We will now turn to the WAIT block. This is to become a timer which we will use to specify the length of each traffic light phase. Our block receives a variable TIME of the type TIME as an input variable, and returns a Boolean value, which we will name OK. This will be TRUE when the required time has elapsed. We will preset this value with FALSE by inserting " := FALSE " at the end of the declaration (but before the semi-colon).

For our purposes we require the block TP, a pulse generator. This has two inputs (IN, PT) and two outputs (Q, ET). TP now does the following:

As long as IN is FALSE, ET is 0 and Q is FALSE. As soon as IN supplies the value TRUE, the time in milliseconds is incremented in the output ET. When ET reaches the value PT, ET stops incrementing. Meanwhile, Q continues to supply TRUE as long as ET is less than PT. As soon as the value PT is reached, Q supplies FALSE again. A brief description of all the blocks in the standard library can be found in the appendix.

To be able to use the block TP in the WAIT block, we must create a local instance of TP. To do this, we declare a local variable TEL (for time elapsed) of the type TP (between the keywords VAR and END\_VAR).

The declaration part of WAIT now looks like this:



```

0001 FUNCTION_BLOCK WAIT
0002 VAR_INPUT
0003     TIME_IN: TIME;
0004 END_VAR
0005 VAR_OUTPUT
0006     OK: BOOL := FALSE;
0007 END_VAR
0008 VAR
0009     ZAB: TP;
0010 END_VAR

```

Figure 3\_3: Function block WAIT, declaration part

### "WAIT" Body

To implement the required timer, the body of the block must be programmed as follows:

```

0001 FUNCTION_BLOCK WAIT
0002 LD ZAB.Q
0003 JMPC mark
0004
0005 CAL ZAB(IN:=FALSE)
0006 LD TIME_IN
0007 ST ZAB.PT
0008 CAL ZAB(IN:=TRUE)
0009 JMP end
0010
0011 mark:
0012 CAL ZAB
0013 end:
0014 LDN ZAB.Q
0015 ST OK
0016 RET

```

Figure 3\_4: Function block WAIT, instruction part

A check is first carried out to determine whether Q is already set to TRUE (i.e. whether counting has already taken place). In this case, we do not change anything in the allocation of TEL. Instead, we call the function block TEL without input (in order to check whether the time has already elapsed).

Otherwise, we set the variable IN in TEL to FALSE, thereby simultaneously setting ET to 0 and Q to FALSE. All variables are now set to the required initial status. We now save the time required from the variable TIME to the variable PT, and call TEL using IN:=TRUE. In the function block TEL, the variable ET is now incremented until it reaches the value TIME. Q is then set to FALSE.

The negated value of Q is saved to OK after each run of WAIT. OK supplies TRUE as soon as Q is FALSE.

The timer is now finished. Next we must combine our two function blocks, WAIT and LIGHTS, in the main program PLC\_PRG.

**“PLC PRG”: First Project Stage**

First we declare the variables we need. These are two instances of the function block LIGHTS (LIGHTS1, LIGHTS2) and one of the type WAIT (DEL as in delay). PLC\_PRG now looks like this:

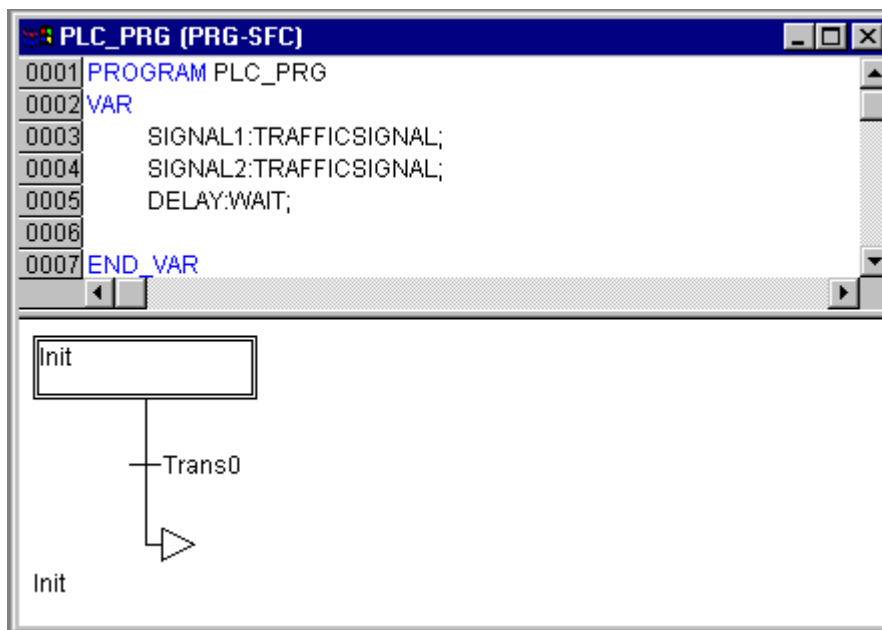


Figure 3\_5: PLC\_PRG program, first project stage, declaration part

**Creating a SFC Diagram**

The initial diagram of a block in SFC consists of an action “Init” of a subsequent transition “Trans0” and a return to Init. We will now expand slightly on this.

We will first define the structure of the diagram before we program the individual actions and transitions. First of all, we require a step for each trafficlight phase. These are inserted by selecting Trans0 and then selecting ‘Insert’ ‘Step transition (after)’. Repeat this process three times.

If you click directly on the name of a transition or step, it is selected and you can change it. Name the first transition after Init “TRUE”, and all other transitions “DEL.OK”.

Therefore, the first transition always switches through, and then all others switch through if DEL in OK outputs TRUE, i.e. when the input time has elapsed.

The steps receive the names Switch1, Green2, Switch2, Green1 from top to bottom, with Init obviously retaining its name. “Switch” should always mean an amber phase, while in Green1, LIGHTS1 is green, and in Green2, LIGHTS2 is green. Finally, change the return address of Init to Switch1. If you have done all this correctly, the diagram should look like this:

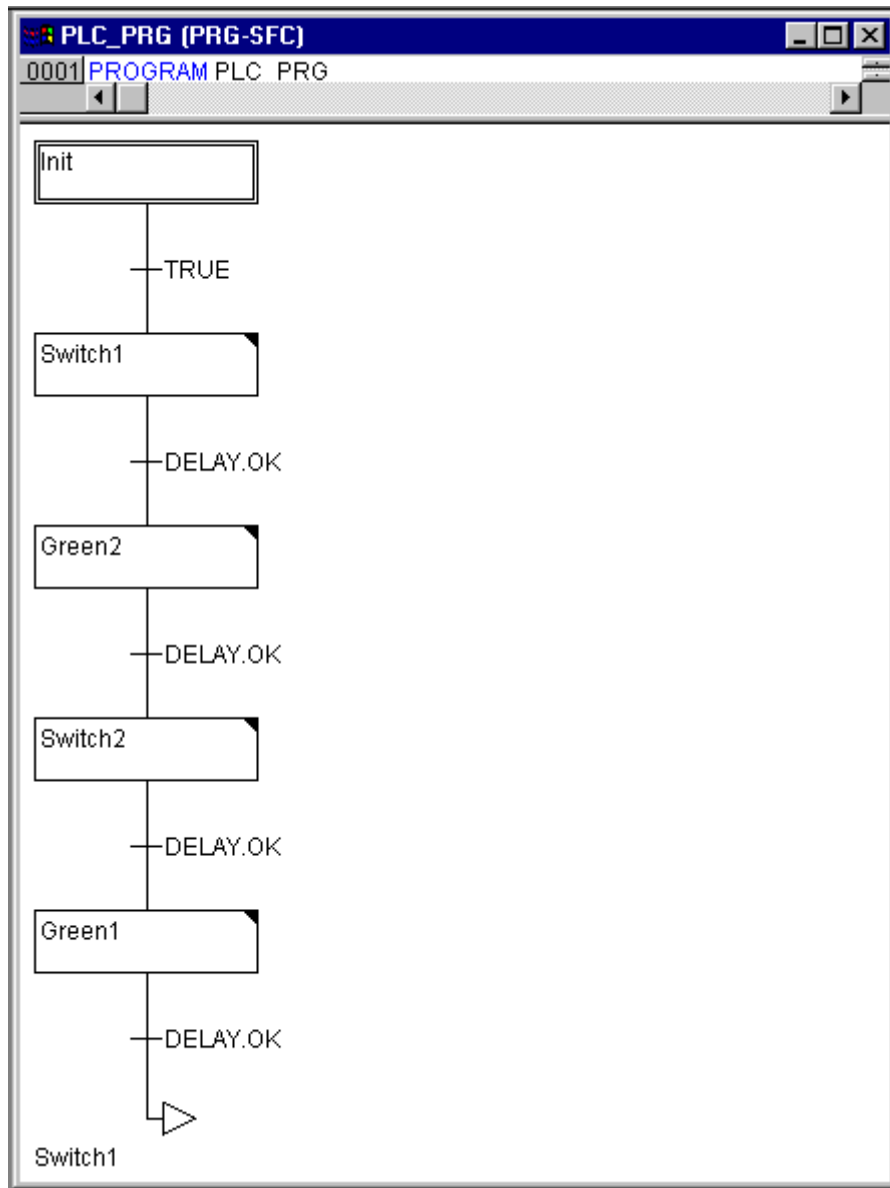


Figure 3\_6: PLC\_PRG program, first project stage, instruction part

We must now program the individual steps. If you double-click on the field of a step, a dialog appears for opening a new action. In our case, we will use IL (Instruction List) as the language.

**Actions and Transition Conditions**

In the action for the step **Init**, the variables are initialised. The STATUS of LIGHTS1 should be 1 (green). The status of LIGHTS2 should be 3 (red). The Init action then looks like this:

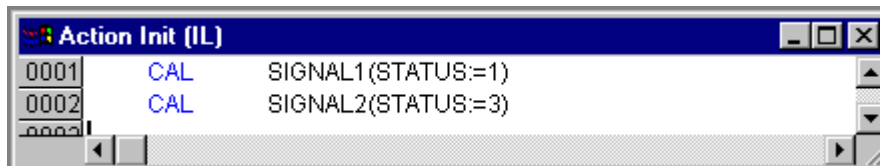


Figure 3\_7: Init action

In **Switch1**, the STATUS of LIGHTS1 switches to 2 (amber) and the status of LIGHTS2 switches to 4 (amber/red). In addition, a delay time of 2,000 milliseconds is now defined. The action now looks like this:

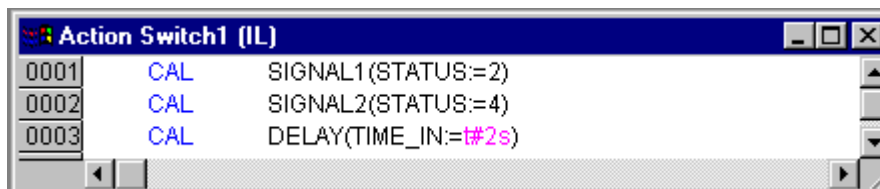


Figure 3\_8: Switch1 action

In **Green2**, LIGHTS1 is red (STATUS:=3), LIGHTS2 is green (STATUS:=1) and the delay time is set to 5,000.

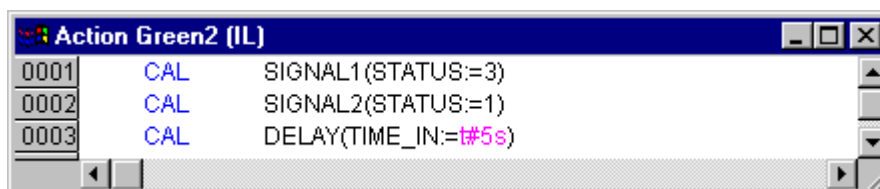


Figure 3\_9: Green2 action

In **Switch2**, the STATUS of LIGHTS1 switches to 4 (amber/red), and the status of LIGHTS2 switches to 2 (amber). A delay time of 2,000 milliseconds is now defined.

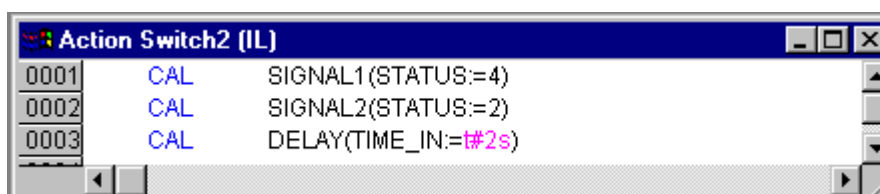


Figure 3\_10: Switch2 action

In **Green1**, LIGHTS1 is green (STATUS:=1), LIGHTS2 is red (STATUS:=3), and the delay time is set to 5,000.

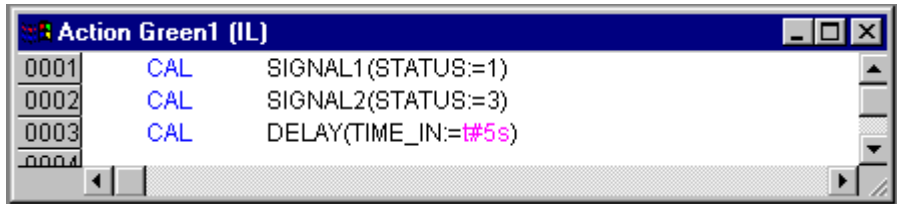


Figure 3\_11: Green1 action

This completes the first project phase of our program. You can now compile it and test it using the simulation option.

**PLC PRG: Second Project Stage**

In order to have at least one alternative branch in our diagram, and to allow our traffic lights to be switched off at night, we will now integrate a counter into our program which will switch off the traffic lights after a certain number of traffic light cycles.

First we need a new variable COUNTER of the type INT. We declare this variable in the usual way in the declaration part of PLC\_PRG, and initialise it in Init using 0.

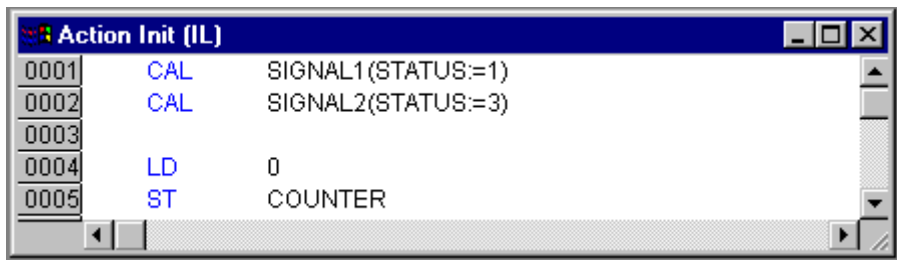


Figure 3\_12: Init action, second stage

Now select the transition after Switch1 and insert a step and a transition after it. Select the newly-created transition and insert an alternative branch to the left of it. Insert a step and a transition after the left-hand transition. Now insert a jump to Switch1 after the newly-created transition.

Name the newly-created parts as follows: the top-most of the two new steps should be called "Count", and the lower one should be called "Off". From top to bottom and left to right, the transitions are called END, TRUE and DEL.OK. The newly-created part should therefore look like the part outlined in black here.

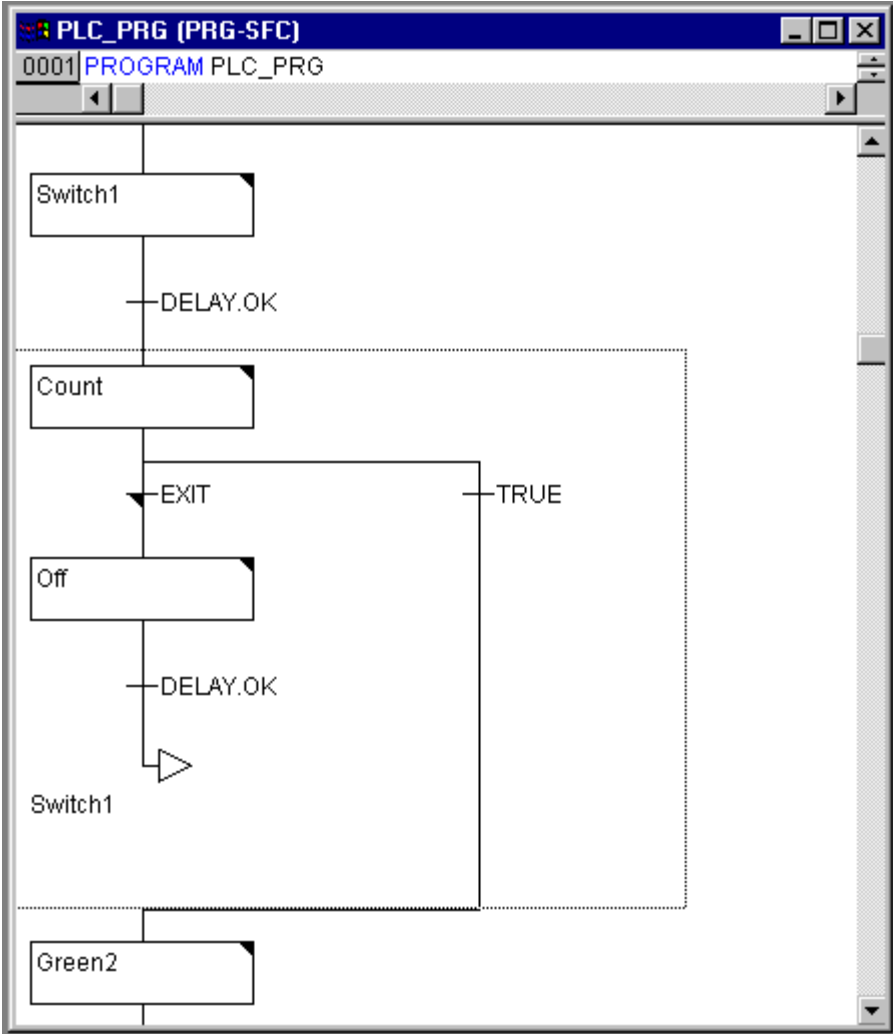


Figure 3\_13: Traffic lights

There are now two new actions and one new transition condition to be implemented. All that happens in the step Count is that COUNTER is increased by one:

```
0001 LD COUNTER
0002 ADD 1
0003 ST COUNTER
```

Figure 3\_14: Count action

The transition END checks whether the counter is greater than a certain number, let's say 7:



Figure 3\_15: END transition

At Off, the status of both traffic lights is set to 5 (OFF), the COUNTER is reset to 0 and a delay time of 10 seconds is defined:

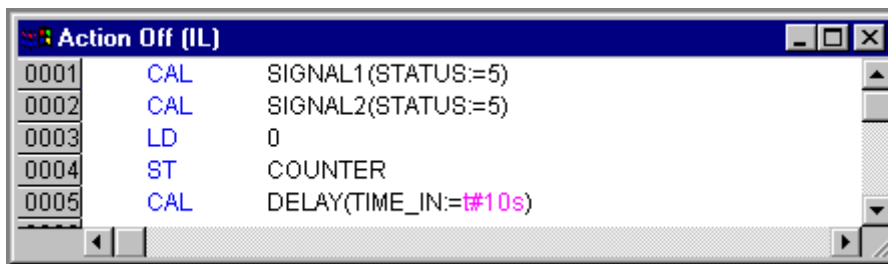


Figure 3\_16: Off action

## The Result

Where our set of traffic lights is installed, it is night-time after seven cycles, the traffic lights switch off for ten seconds, then it is daytime again, the traffic lights switch on again, and the whole cycle is repeated from the beginning.

## Traffic Light Simulation

Now we will test our program. To do this, we must compile it ('Project' 'Rebuild all') and load it ('Online' 'Login' followed by 'Online' 'Download'). If you now execute 'Online' 'Run', you can trace the time sequence of the individual steps in your main program. The window of the PLC\_PRG block has now become more of a monitor window. A double-click on the plus sign in the declaration editor opens the variable display and you can view the values of the individual variables.

---

## Visualising a Set of Traffic Lights

The visualization functionality of **CP1131** quickly and easily brings project variables to life. A more detailed description of visualization can be found in the chapter entitled 'Visualization'. We will now draw two sets of traffic lights to illustrate the switching process.

### Creating a New Visualization


To create a visualization, you must first select the area **Visualization** in the Object Organizer. To do this, click on the left-hand side of the lower edge of the window where the **blocks** are found, on the tab with this icon  and then on the name **Visualization**. If you now execute the command '**Project**' '**Add object**', a dialog opens.



Figure 3\_17: Dialog for opening a new visualization

Enter any meaningful name here. When you confirm the dialog by clicking **OK**, a window opens for you to create your new visualization.

### Inserting an Element in a Visualization

The best way to visualise our traffic lights is to proceed as follows:

- Enter the command '**Insert**' '**Ellipse**' and try to draw a smallish circle (diameter 2cm). To do this, click in the edit field and drag the circle while holding down the left-hand mouse button.
- Now double-click on the circle. The dialog for editing visualization elements opens.
- Select the category **Variables** and enter the text PLC\_PRG.LIGHTS1.RED in the field **Change color**. This addresses the variable RED to the function block instance LIGHTS1 of the block PLC\_PRG.

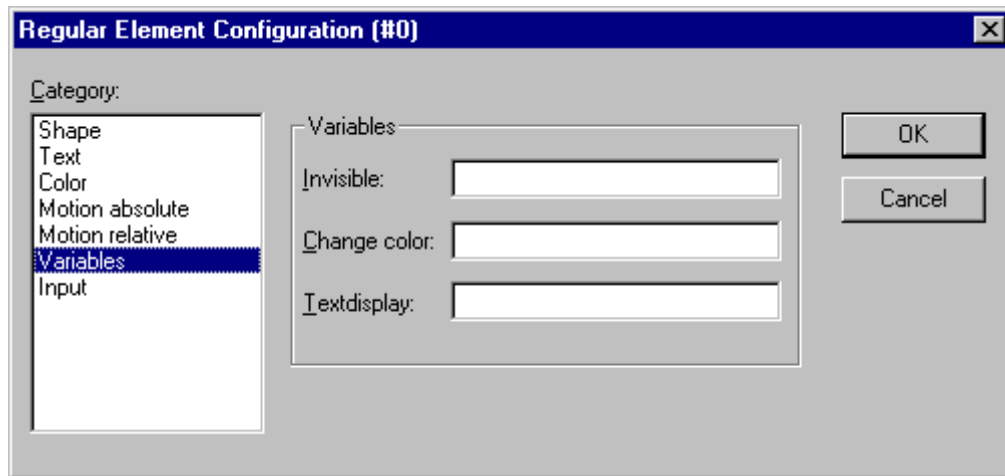


Figure 3\_18: Visualization dialog: Variables

- Then select the category **Colors** and click on the **Inside** button in the **Colors** area. Select as neutral a color as possible, such as black.
- Now click on the **Inside** button in the area **Alarm color** and select a red that corresponds as closely as possible to traffic light red.

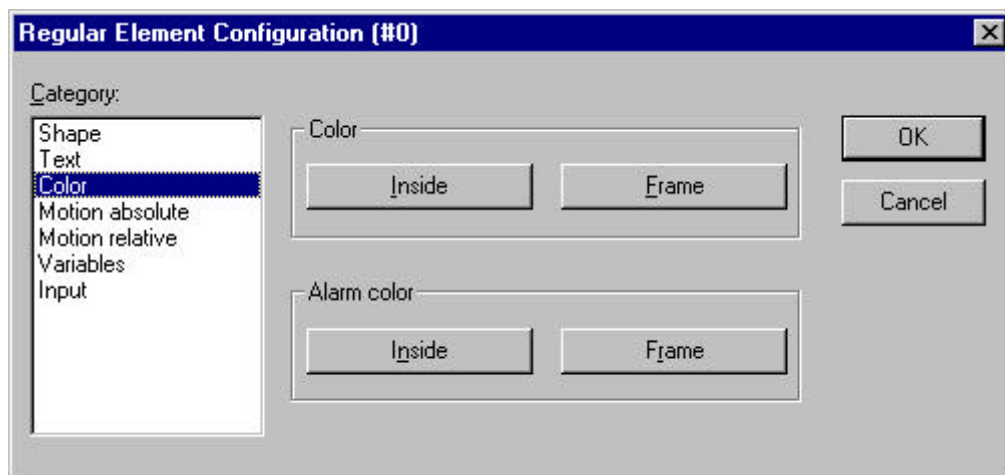


Figure 3\_19: Dialog for configuring visualization elements (Colors category)

The circle created is black in normal status, and if the variable RED of LIGHTS1 is TRUE, its color changes to red. We have now created the first light in the first set of traffic lights.

### The other Traffic Lights

Now enter the commands **'Edit' 'Copy'** (<Ctrl>+<C>) followed by **'Edit' 'Paste'** twice (<Ctrl>+<V>). You now have two more circles of the same size on top of the first circle. You can move the circles by clicking on them and moving them to the required positions while holding down the left-hand mouse button. For our purposes, the required positions should be in a vertical row on the left-hand side of the editor window. Double-clicking on one of the two lower circles opens the configuration dialog again. Enter the following variables in the **Change color** field for the corresponding circle:

For the middle circle:            PLC\_PRG.LIGHTS1.AMBER

For the lower circle:            PLC\_PRG.LIGHTS1.GREEN

Now select the corresponding colors (amber or green) for the circles in the category **Colors** and in the area **Alarm color**.

### The Traffic Light Housing

Now enter the command **'Insert' 'Rectangle'**, and insert a rectangle to enclose the circles in the same way as you inserted the three circles. Select as neutral a color as possible for the rectangle and enter the command **'Extras' 'Send to back'**, to make the circles visible again.

If simulation mode<sup>1</sup> has not yet been activated, you can activate it using the command **'Online' 'Simulation Mode'**.

By starting the simulation using the commands **'Online' 'Login'** and **'Online' 'Run'**, you can trace the color change of the first traffic lights.

### The Second Set of Traffic Lights

The easiest way to create the second traffic lights is to copy all the components of the first traffic light. To do this, select all the elements of the first traffic lights and copy them (as you copied the lights for the first traffic lights) using the commands **'Edit' 'Copy'** and **'Edit' 'Paste'**. All you have to do now is change the text "LIGHTS1" to "LIGHTS2" in the relevant visualization dialog to complete visualization of the second traffic lights.

---

<sup>1</sup> Simulation mode is active if a check mark (✓) is displayed in the 'Online' menu in front of the menu item 'Simulation'.

### Text in the Visualization

To complete the visualization, you must insert two more flat rectangles below the traffic lights.

In the visualization dialog, you must now set white for the **Frame** in the category **Colors** and type "Traffic lights 1" or "Traffic lights 2" in the **Content** field of the category **Text**. Your visualization now looks like this:

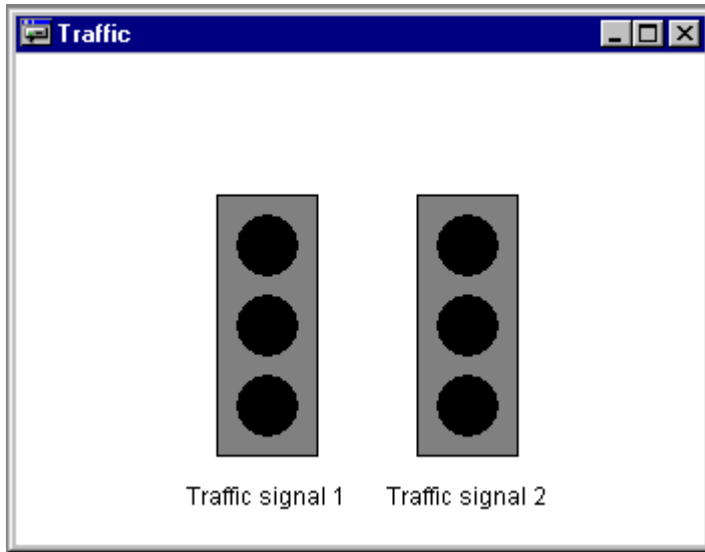


Figure 3\_20: Visualization for the sample project 'Traffic lights'.

## A Detailed Look at Components

### The Main Window

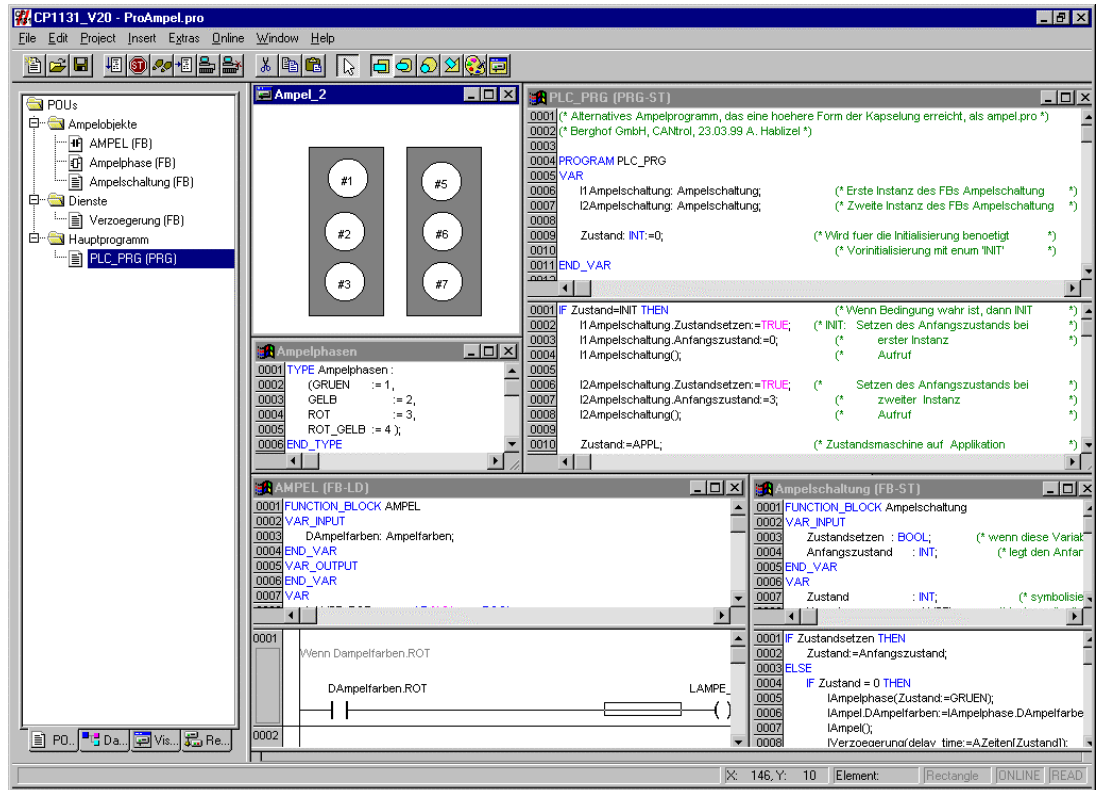


Figure 4\_1: The main window

The following elements are found in the main window of **CP1131** (from top to bottom):

- the menu bar
- the tool bar (optional), with buttons for faster execution of menu commands
- the Object Organizer with tabs for blocks, data types, visualizations and resources
- a vertical screen split bar between the Object Organizer and the workspace of CP1131
- the workspace which contains the editor windows
- the message window (optional)
- the status bar (optional) with information on the current status of the project

## Menu Bar

The menu bar is found at the upper edge of the main window. It contains all the menu commands.

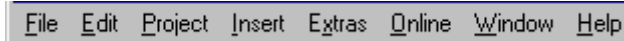


Figure 4\_2: Menu bar

## Tool Bar

The tool bar allows you to select a menu command faster by

clicking on an icon. The selection of icons available adapts automatically to whichever window is active.

The command is only executed if the mouse button is pressed and released over the icon. If you hold the mouse pointer over an icon in the tool bar for a short time, the name of the icon is displayed in a tooltip.

Display of the tool bar is optional (see 'Project' 'Options', category 'Desktop').



Figure 4\_3: Tool bar with icons

## Object Organizer

The Object Organizer is always found on the left-hand side of CP1131. Below it are found four tabs with icons for the four object types **Blocks**, **Data types**, **Visualizations** and **Resources**. To switch between the various object types, click on the corresponding tab with the mouse or use the left or right arrow key.

To find out how to use the objects in the Object Organizer, see the section entitled 'Objects: creating, deleting, etc.'.

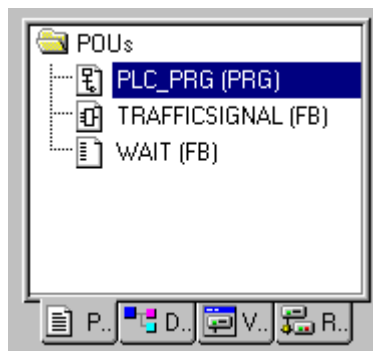


Figure 4\_4: Object Organizer

## Screen Split Bar

The screen split bar is the border between two non-overlapping windows. In **CP1131**, there are screen split bars between the Object Organizer and the workspace of the main window, between the interface (declaration part) and the implementation (instruction part) of blocks, and between the workspace and the message window.

If you position the mouse pointer on the screen split bar, you can move the screen split bar by moving the mouse while holding down the left-hand mouse button.

Remember that the screen split bar always remains at its absolute position, even if the window size changes. If the screen split bar seems to have disappeared, simply maximise your window.

## Workspace

The workspace is found on the right-hand side of the **CP1131** main window. All editors for objects and library management are opened in this area.

A description of the editors can be found in the chapters entitled '**The editors in CP1131**', '**The resources**', '**Visualization**' and '**Library management**'.

All the commands for window management are found under the menu item '**Window**'.

## Message Window

The message window is found below the workspace in the main window, separated by a screen split bar.

It contains all the messages from the most recent compilation, test or comparison process.

When you double-click on a message in the message window using the mouse, or press the <Enter> key, the editor opens with the object. The relevant line of the object is selected. You can switch quickly between the error messages using the commands '**Edit**' '**Next error**' and '**Edit**' '**Previous error**'.

Displaying the message window is optional (see '**Window**' '**Messages**').

## Status Bar

The status bar, which is found at the lower edge of the **CP1131** main window, displays information about the current project and about menu commands.

If an instruction is relevant, the term appears on the right in the status bar in black type. Otherwise, it is in grey type.

If you are working in online mode, the word **Online** is displayed in black type. If you are working in offline mode, it is displayed in grey type.

In online mode, the status bar indicates whether you are running a simulation (**SIM**), the program is being processed (**RUN**), a breakpoint is set (**BP**) or variables are being forced (**FORCE**).

In text editors, the line and column numbers of the current cursor position are specified (e.g. **Ln:5, Col:11**).

If the mouse pointer is in a visualization, the current **X** and **Y positions** of the cursor are specified in pixels relative to the upper left-hand corner of the picture. If the mouse pointer is on an **element**, or if an element is being processed, its number is specified. If you have selected an element for insertion, this is also displayed (e.g. **rectangle**).

When a menu command has been selected but not yet executed, a short description appears in the status bar.

Displaying the status bar is optional (see '**Project**' '**Options**', category '**Desktop**').

### **Context-Sensitive Menu**

**Shortcut: <Shift>+<F10>**

Instead of using the menu bar to execute a command, you can use the right-hand mouse button. The menu displayed when the button is pressed contains the most frequently-used commands for a selected object or for the active editor. The selection of commands provided adapts automatically to the active window.

---

## **Options**

Of course, you will only have one view with regard to **CP1131**, on the other hand, CP1131 allows you to configure your view of the main window. Furthermore, you can also make other settings using the command '**Project**' '**Options**'. Unless an alternative is specified, all settings made here are saved in the file "cp1131\_v20.ini", and are restored the next time **CP1131** is activated.

### **'Project' 'Options'**

This command opens the dialog for setting the options. The options are divided into various categories. Select the category required in the left-hand side of the dialog by clicking with the mouse or by using the arrow keys, and then change the options on the right-hand side.

The following categories are available:

- Load & Save
- User Information
- Editor
- Desktop
- Colors
- Directories
- Build
- Passwords

## Load & Save

If you select this category, the following dialog is displayed:

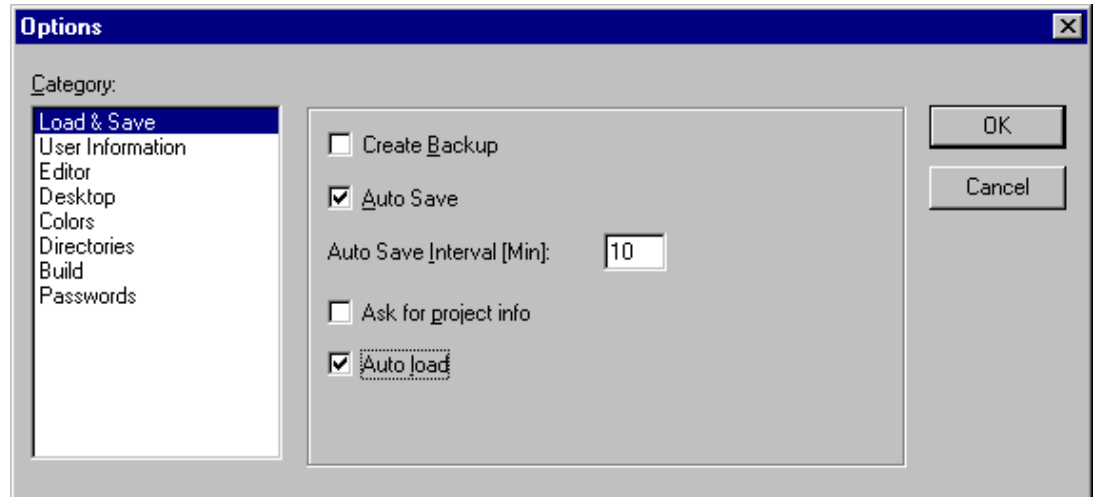


Figure 4\_5: Options dialog for the category 'Loading & saving'

When an option is activated, a check mark (✓) appears in front of the option.

If you select the option **Create Backup**, CP1131 saves the old file to a backup file with the extension ".bak" each time the file is saved. This means that you can always recover the version before the last save.

If you select the option **Auto Save**, your project will be saved as you work on it to a temporary file with the extension ".asd" at the time intervals specified by you (**Auto Save interval**). This file is deleted when the program is ended normally. However, if CP1131 ends "abnormally" for some reason (e.g. due to a power failure), the file is not deleted. The next time the file is opened, the following message is displayed:

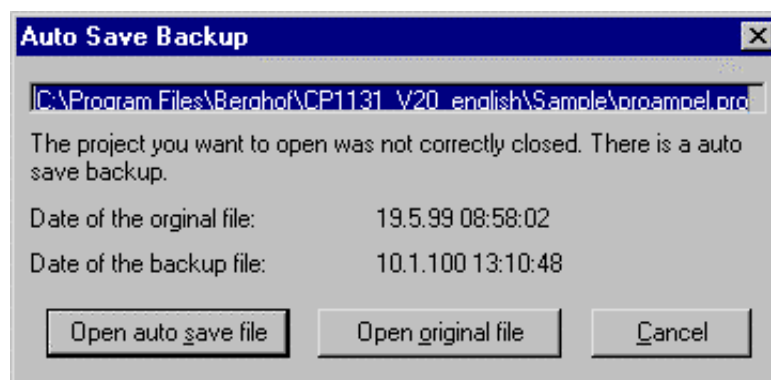


Figure 4\_6: Message if there is a backup file in existence

You can now decide whether you want to open the original file or the backup file. If you select the option **Ask for project info**, project information is called up automatically whenever a new project is saved, or a project is saved under a new name. Project information can be viewed and edited using the command 'Project' 'Project information'.

If you select the option **Auto Load**, the last project opened is loaded automatically the next time **CP1131** is started. A project can also be loaded at startup of **CP1131** by specifying a project in the command line.

### User Information

If you select this category, the following dialog is displayed:

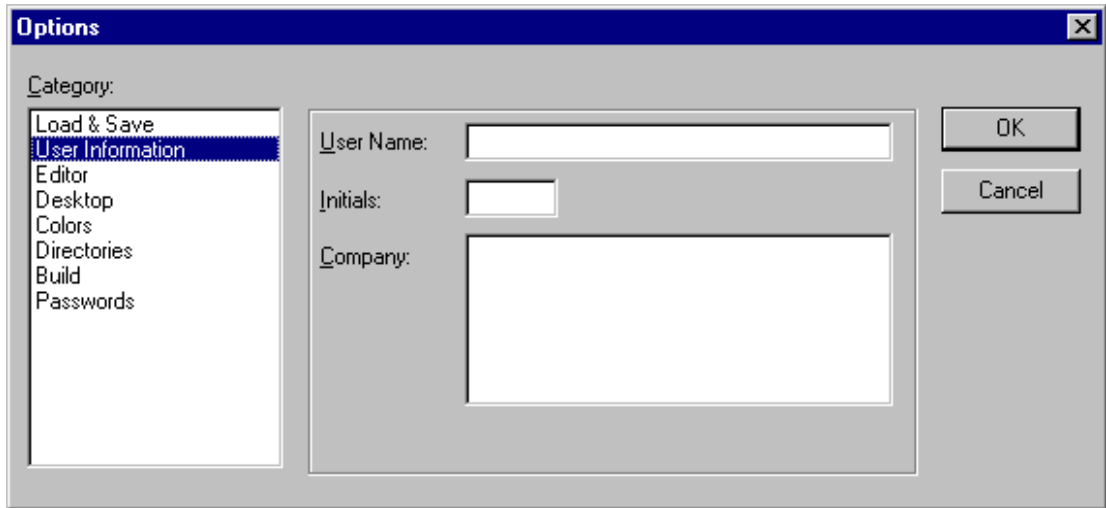


Figure 4.7: Options dialog for the category 'User information'

User information consists of the user **name**, their **initials** and the **company** for which they are working. Each of the entries can be changed, and they are saved with the project.

### Editor

If you select this category, the following dialog is displayed:

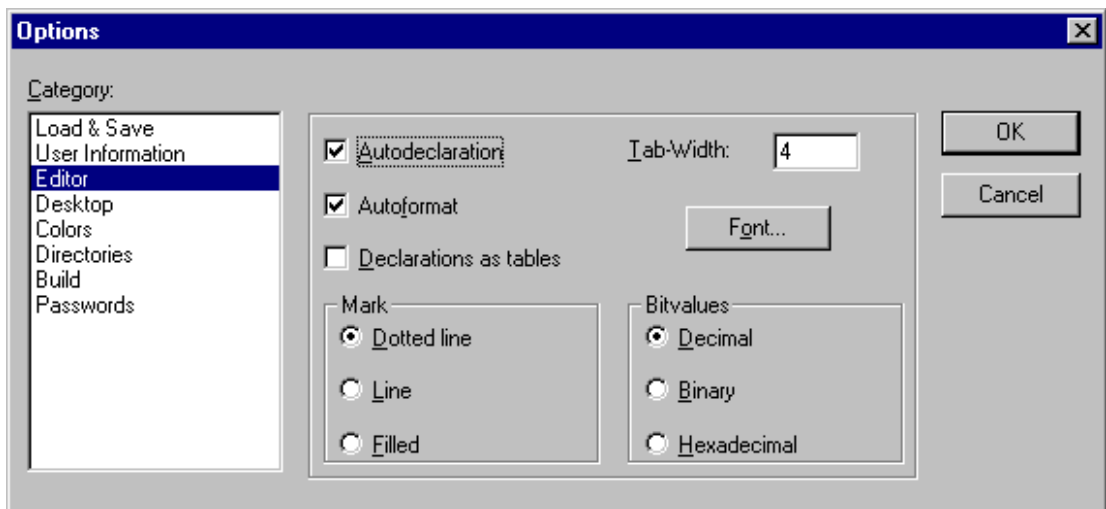


Figure 4.8: Options dialog for the category 'Editor'

When an option is activated, a check mark (✓) appears in front of the option.

You can select the following settings for the editors:

- Autodeclaration
- Autoformat
- Declaration as tables
- Tab width
- Font
- Display of selection (Mark)
- Display of bit values

### **Autodeclaration**

If the option **Autodeclaration** has been selected, a dialog appears in all editors after entry of a variable which has not yet been declared. It can be used to declare the variable.

### **Autoformat**

If the option **Autoformat** has been selected, **CP1131** carries out automatic formatting in the Instruction List editor and in the declaration editor. If a line is omitted, the following formatting is carried out:

- operators in lowercase are displayed as uppercase
- tabulators are inserted to create a uniform column division.

### **Declarations as Tables**

If the option **Declarations as table** has been selected, you can edit variables as a table instead of using the usual declaration editor. This table is arranged like a card index containing tabs for input, output, local and input/output variables. The fields **Name**, **Address**, **Type**, **Initial** and **Comment** are available.

### **Tab Width**

In the **Tab width** field, you can specify the width at which a tabulator is to be displayed in the editors. The default setting is four characters wide, with the character width being dependent on the font set.

### **Font**

By pressing the **Font** button, you can select the font in all **CP1131** editors. The size of the font is the basic unit for all drawing operations. Selection of a larger font increases the size of the output and also the printout in each **CP1131** editor.

After you have selected the command, the dialog opens for selection of the font type, style and size.

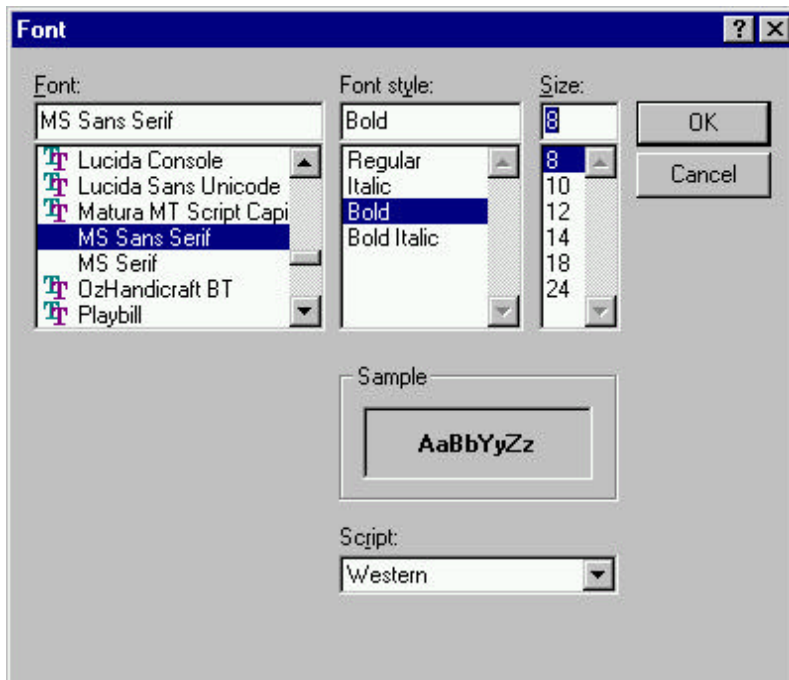


Figure 4\_9: Dialog for setting the font

## Mark

With the **Mark** option, you can choose whether to display the current selection in your graphical editors as a dotted rectangle (**Dotted line**), a rectangle with a solid **line** or a filled-in rectangle (**Filled**). In the latter case, the selection is displayed in reverse video.

The active selection is preceded by a dot (•).

## Bit Values

With the **Bit values** option, you can choose whether binary data (type Byte, WORD, DWORD) is to be displayed in **decimal**, **hexadecimal** or **binary** form during monitoring.

The active selection is preceded by a dot (•).

**Desktop**

If you select this category, the following dialog is displayed:

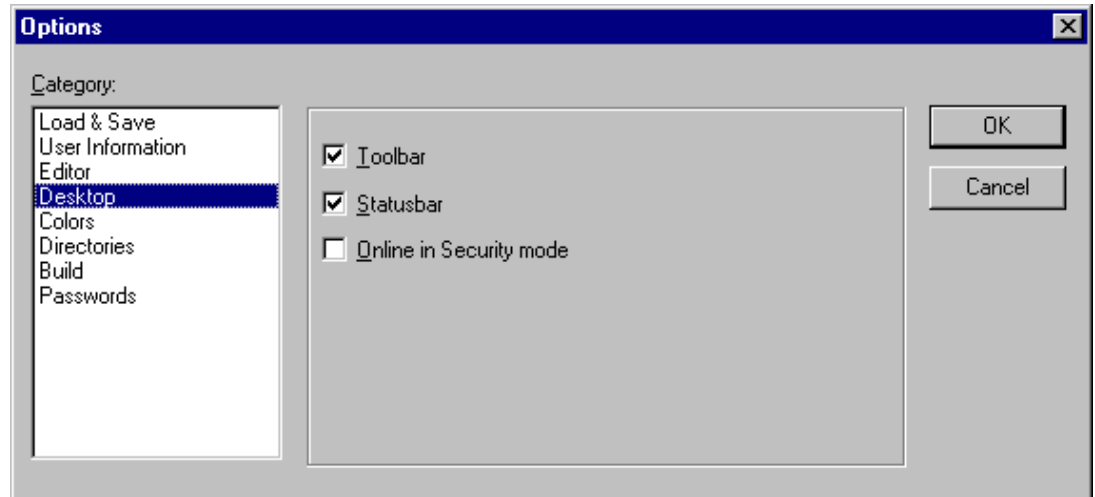


Figure 4\_10: Options dialog for the category 'Desktop'

If the option **Tool bar** has been selected, the tool bar with the buttons for faster selection of menu commands is displayed below the menu bar. If the option **Status bar** has been selected, the status bar is displayed at the lower edge of the **CP1131** window.

If the option **Online operation in security mode** has been selected, a dialog asking you to confirm execution of the command appears when the commands **'Run'**, **'Stop'**, **'Reset'**, **'Breakpoint at'**, **'Single cycle'**, **'Write values'**, **'Force values'** and **'End forcing'** are selected in online mode. This option is saved with the project.

**Colors**

If you select this category, the following dialog is displayed:

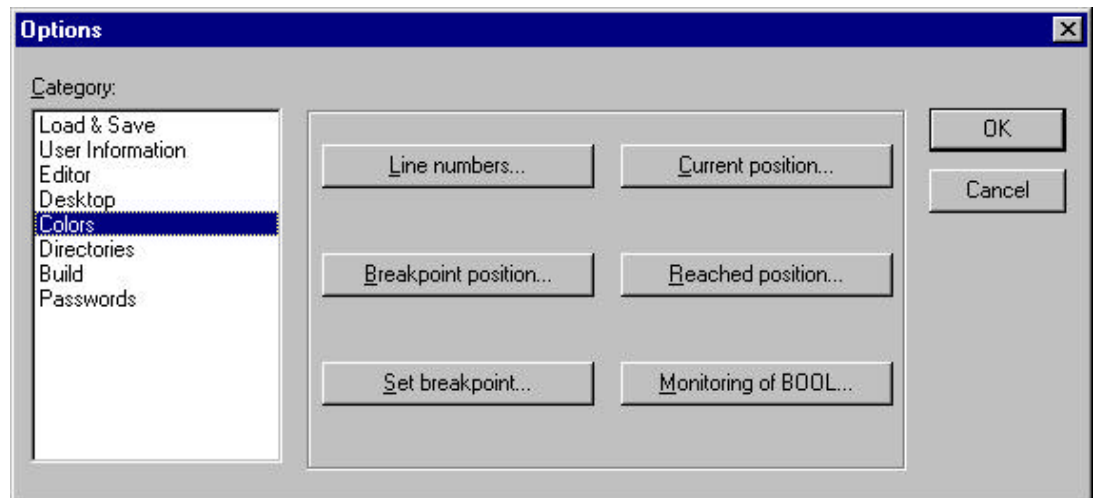


Figure 4\_11: Options dialog for the category 'Colors'

It is possible to edit the preset color settings in **CP1131**. You can choose whether to change the color settings for **line numbers** (presetting: light grey), **breakpoint positions** (dark grey), **set breakpoints** (light blue), **current position** (red), **reached positions** (green) or for **monitoring of Boolean values** (blue).

When you have selected one of the buttons specified, the dialog opens for input of colors.



Figure 4\_12: Dialog for setting colors

### Directories

If you select this category, the following dialog is displayed:

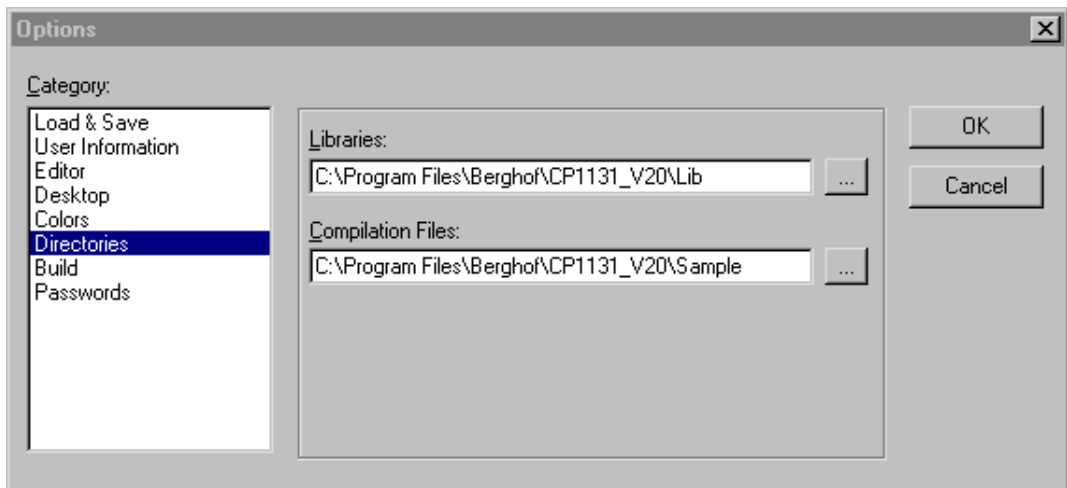


Figure 4\_13: Options dialog for the category 'Directories'

In the input fields **Libraries** and **Compilation files**, you can specify the directories from which **CP1131** is to retrieve the libraries or compilation files. When you press the button (...) after a field, the dialog for selection of a directory opens.

## Build

If you select this category, the following dialog is displayed:

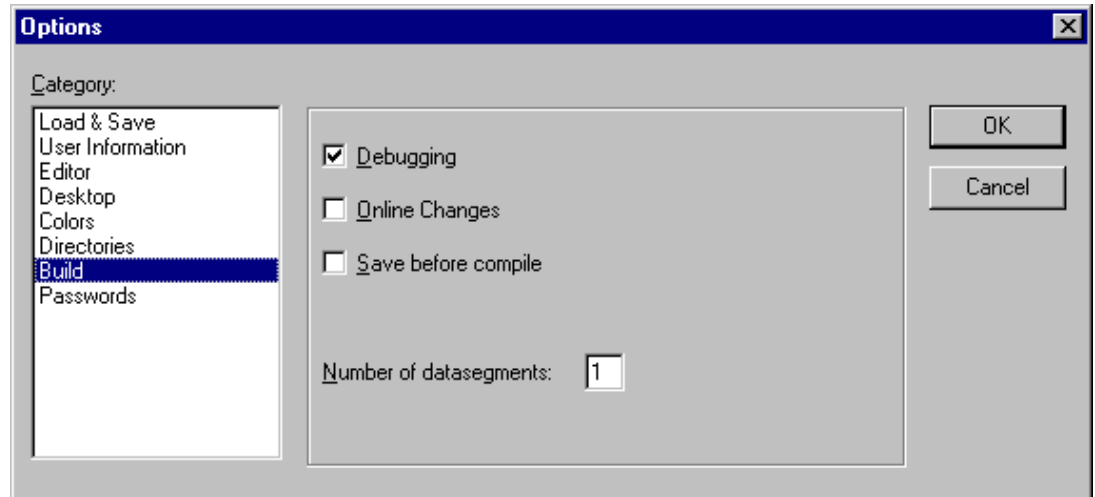


Figure 4\_14: Options dialog for the category 'Build'

If the option **Debugging** is selected, the code can become much more extensive. Selection of this option allows the generation of additional debugging codes. This is necessary in order to use the debugging functions provided by **CP1131**. Deselection of this option allows faster execution and a smaller range of codes. This option is saved with the project.

If the option **Online Changes** is selected, your project can be edited in online mode. The next time you compile the project, only the blocks which have changed are loaded to the controller (see '**Project**' '**Compile**').

If the option **Save before compile** is selected, your project is saved before each compilation.

By specifying the **Number of data segments**, you can define how much space is to be reserved in the controller for your project data. If the following message appears during compilation: "The global variables need too much memory. Increase the number of segments in '**Project**' '**options**' '**build**' you should increase the number of data segments.

These options are saved with the project.

## Passwords

If you select this category, the following dialog is displayed:

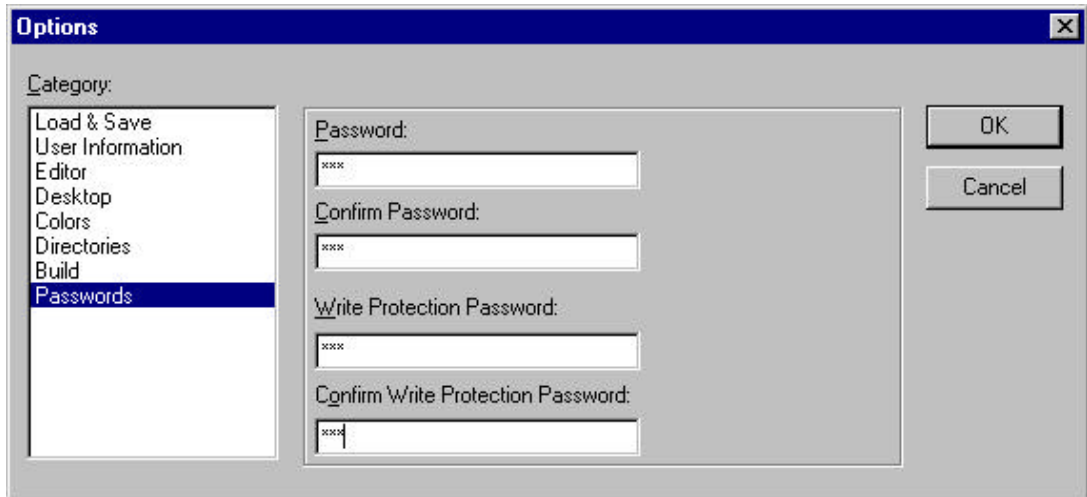


Figure 4\_15: Options dialog for the category 'Passwords'

To protect your files from unauthorised access, **CP1131** offers the option of controlling the opening or editing of your file by means of a password.

Enter your chosen password in the field **Password**. An asterisk (\*) appears in the field for each letter typed. You must type the same word again in the field **Confirm password**. Close the dialog by clicking **OK**. If the following message is displayed:

“The password and its confirmation do not match”,

you have made a typing error in one of the entries. It is best to repeat both entries until the dialog closes without any message.

If you save the file now and open it again, a dialog appears requesting you to enter the password. This project does not open until you have entered the correct password. If the wrong password is entered, **CP1131** outputs the message:

“The password is incorrect.”

In addition to protecting the file from being opened, a password can also be used to protect the file from being edited. To do this, you must type an entry in the field **Write protection password**, and confirm this entry in the field underneath.

A write-protected project can be opened without entry of a password by pressing the **Cancel** button when **CP1131** requests you to enter the write-protection password during opening of a file. The project can now be compiled, loaded to the controller, simulated, and so on, but not edited in any way.

It is of course important to note both passwords carefully yourself. If you forget one of the passwords, please contact the manufacturer of your controller.

The passwords are saved with the project.

In order to create differentiated access rights, you can define user groups (see '**Project**' '**Object**' '**Access rights**' and '**Project**' '**User Group Passwords**').

## Managing projects

Commands relating to an entire project are found under the menu items **'File'** and **'Project'**. Some of the commands under **'Project'** use objects, and are therefore described in the section **'Objects: creating, deleting, etc.'**.

### 'File' 'New'

Icon: 

This command creates an empty project with the name 'Unnamed'. This name must be changed when the project is saved.

### 'File' 'Open'

Icon: 

This command opens an existing project. If a project has already been opened and edited, **CP1131** asks whether this project is to be saved.

The dialog for opening a file is displayed and a project file with the extension **".pro"** or a library file with the extension **".lib"** must be selected. The file must already exist - it is not possible to create a project using the command **'Open'**.

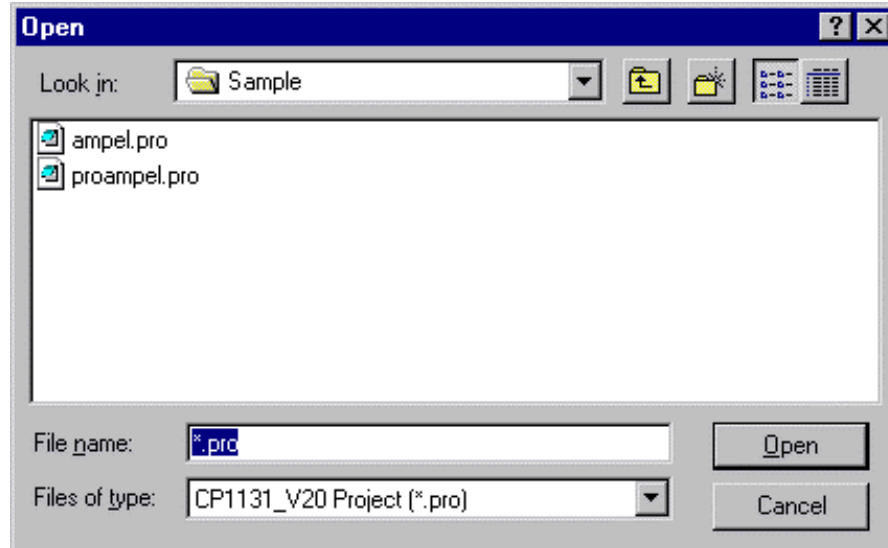


Figure 4\_16: Standard dialog for opening a file in CP1131

The most recently opened projects are listed under the **'File'** command. When you select one of them, this project is opened.

If passwords or user groups have been defined for the project, a dialog appears for entry of the password.

**'File' 'Close'**

This command closes the project that is currently open. If the project has been edited, **CP1131** asks whether these changes are to be saved.

If the project to be saved bears the name "Unnamed", a name must be defined (see '**File**' '**Save as**').

**'File' 'Save'**

Icon:  **Shortcut: <Ctrl>+<S>**

This command saves the project if it has been edited.

If the project to be saved bears the name "Unnamed", a name must be defined (see '**File**' '**Save as**').

**'File' 'Save as'**

Using this command, the current project can be saved in another file or as a library. The original project file remains unchanged.

After the command has been selected, the dialog for saving appears. You must either select an existing **file name**, or enter a new file name and select the required **file type**.

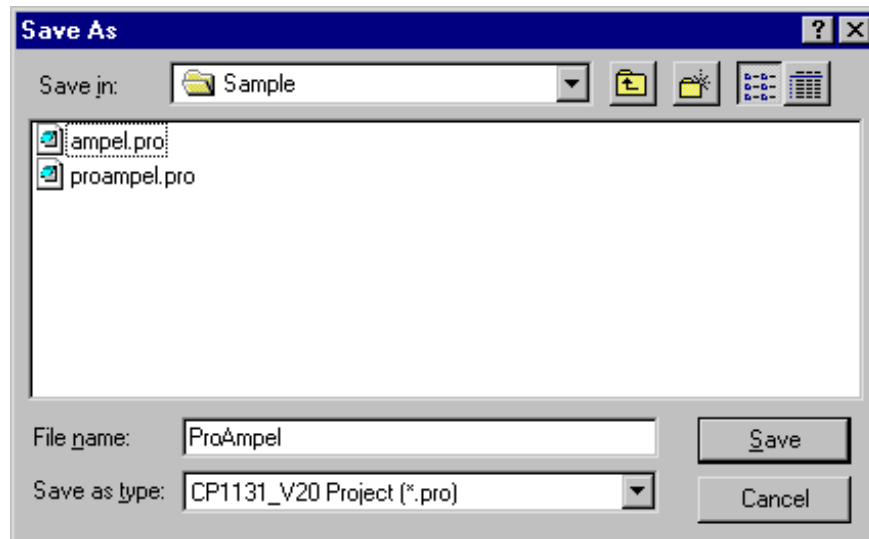


Figure 4\_17: Dialog for 'Save as'

If the project is only to be saved under a new name, select the file type **CP1131\_V20 Project (\*.pro)**.

If you select the file type **Project Version 1.5 (\*.pro)**, the current project is saved as though it had been created in Version 1.5. This means that data specific to Version 2.0 may be lost. However, the project can be further processed in Version 1.5.

You can also save the current project as a library in order to be able to use it in other projects. Select the file type **Internal library (\*.lib)**, if you have programmed your blocks in **CP1131**.

Select the file type **External library (\*.lib)** if you want to integrate blocks that have been implemented in other programming languages (such as C). This causes another file to be saved with the file name of the library, but with the extension “\*.h”. This file is structured as a C header file with the declarations of all blocks, data types and global variables.

Then click on **OK**. The current project is saved in the specified file. If the new file name already exists, you are asked whether this file is to be overwritten.

When being saved as a library, the entire project is compiled. If an error occurs, you are informed that a correct project is required to generate a library. The project is then not saved as a library.

**‘File’ ‘Print’**

**Shortcut: <Ctrl>+<P>**

This command prints the contents of the active window.

After the command has been selected, the dialog for printing is displayed. Select the required option or configure the printer and then click **OK**. The active window is printed.

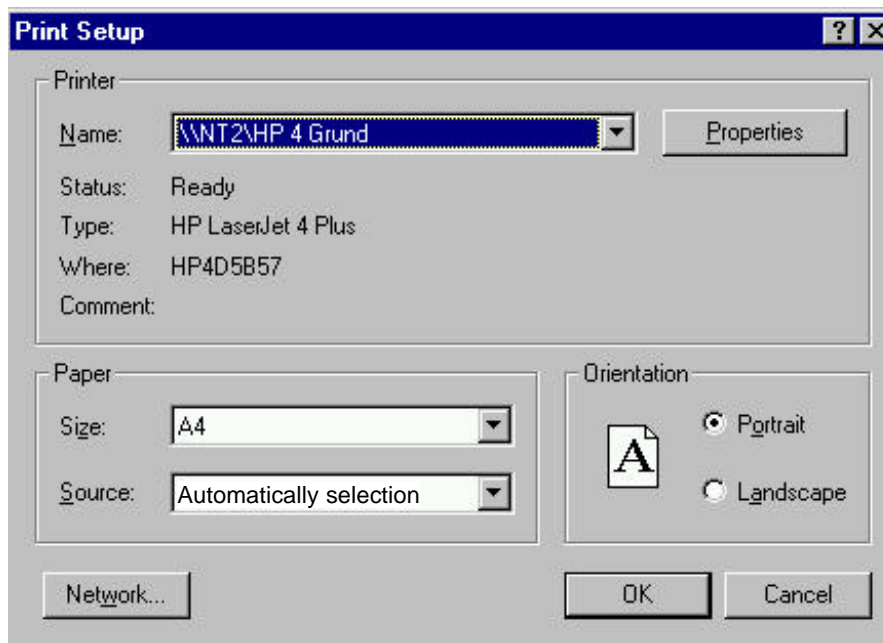


Figure 4\_18: Dialog for printing

In the dialog for printing, you can use Document Project to select the print range (either **All** or a range of explicitly-specified **pages**). When an object is printed, it is printed out in full. You can specify the **number of copies** and route the output to a file.

The **Properties** button opens the dialog for the printer setup.

You can define the layout of your printout using the command **‘File’ ‘Printer Setup’**.

During printing, you are informed of the number of pages already printed in a dialog box. If you close this dialog box, printing stops after the next page.

To document your entire project, use the command **'Project' 'Document'**.

If you want to create a document template for your project, open a global variable list and use the command **'Extras' 'Make Docuframe file'**..

### **'File' Printer Setup'**

You can use this command to define the layout of the printed pages. The following dialog is now opened:

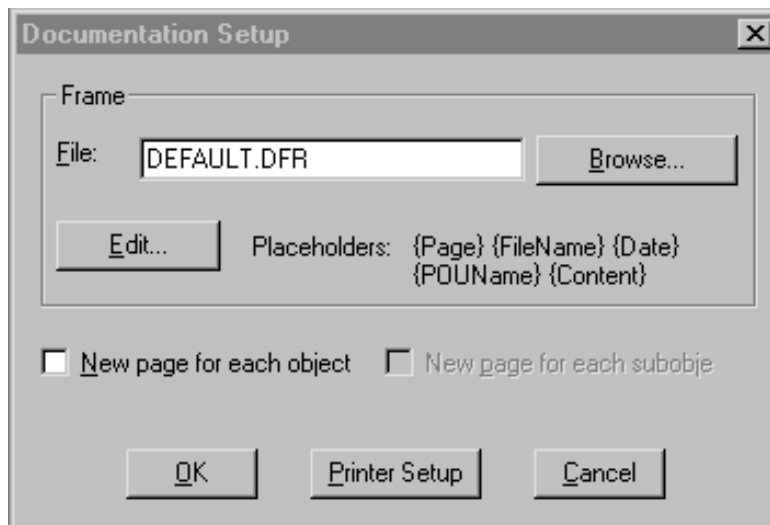


Figure 4\_19: Dialog for setting the page layout of the documentation

In the **File** field you can enter the name of the file in which the page layout is to be saved with the extension ".dfr". The default file setting for saving of the template is DEFAULT.DFR.

If you want to change an existing layout, you can browse through the directory tree to find the required file using the **Browse** button.

You can also select whether to start a **new page for each object** and for **each sub-object**. The **Setup** button opens the printer setup.

When you click on the **Edit** button, a template appears for setting the page layout. Here you can define page numbers, dates, file and block names. You can also position graphics on the page and define the text area in which the documentation is to be printed.

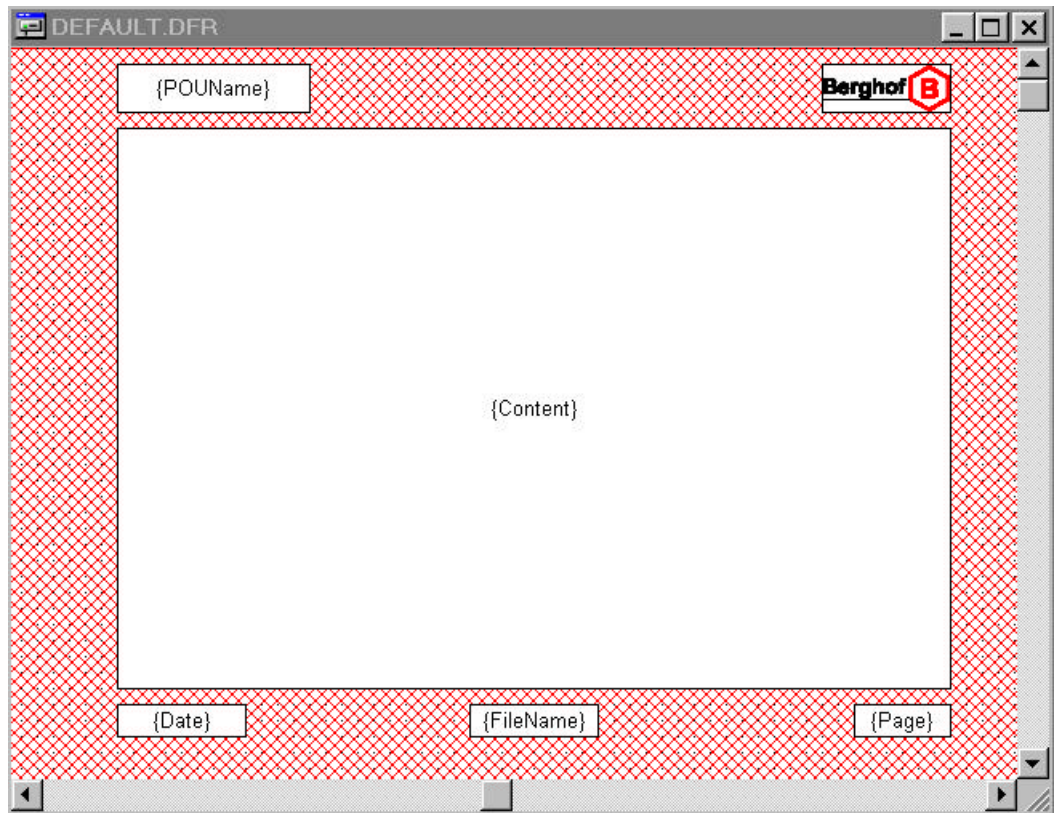


Figure 4\_20: Window for inserting placeholders into the page layout

By selecting the menu item **'Insert' 'Placeholder'** followed by one of the five placeholders (**Page**, **POUName**, **File name**, **Date**, **Contents**), you can insert a placeholder into the layout by dragging a rectangle<sup>2</sup>. These are replaced in the printout by the following:

Command	Placeholder	Effect
Page	{Page}	The current page number appears here in the print-out.
Block name	{POUName}	The name of the current block appears here.
File name	{FileName}	The name of the project appears here.
Date	{Date}	The current date appears here.
Contents	{Content}	The contents of the block appear here.

You can also insert a bitmap graphic (such as a company logo) into the page using **'Insert' 'Bitmap'**. After selecting the graphic, you must also drag a rectangle over the layout using the mouse. Other visualization elements can also be inserted (see the chapter entitled **'Visualization'**).

If the template has been changed, **CP1131** asks whether these changes are to be saved before closing the window.

<sup>2</sup> A rectangle is dragged on the layout by moving the mouse diagonally while holding down the left-hand mouse button.

### 'File' 'Exit'

**Shortcut:** <Alt>+<F4>

This command terminates **CP1131**. If a project is open, it is closed as described in '**File**' '**Save**'.

### 'Project' 'Check'

You can use this command to test the static correctness of your program. If an error occurs, it is reported in the message window in the same way as during compilation of the program.

Unlike the command '**Rebuild all**', no code is generated. You can simply log into the program afterwards, without having to download the program again.

### 'Project' 'Build'

This command compiles *all changed* blocks. When the program is downloaded, only changed blocks are transferred to the controller. The rest of the program remains unchanged in the controller.



**Note:** The '**Build**' command is only supported if **CP1131** has online change functionality (menu: Online\Codegenerator Options\Memory). If it does not, '**Build**' has the same effect as '**Rebuild all**'.

If the changes are extensive, use the functionality of '**Project**' '**Record changes**'.

**Online Change** functionality means that parts of a program can be exchanged (transferred to the controller) without interrupting the controller. All data is retained as far as possible. To write the program changed using Online Change permanently to the flash memory, however, the CPU must be brought to a "stop". Only then can the entire program be saved to the flash memory.



**Warning:** If you execute '**Build**' twice in sequence, without transferring the program to the controller in between, the error message: "Changes are incompatible" appears.

The program must then be recompiled ('Rebuild all') and transferred to the controller in full.

### 'Project' 'Rebuild all'

This command compiles *all* blocks. The message window is opened and outputs the progress of the compilation process and any errors that occur.

A list of all error messages can be found in the appendix.

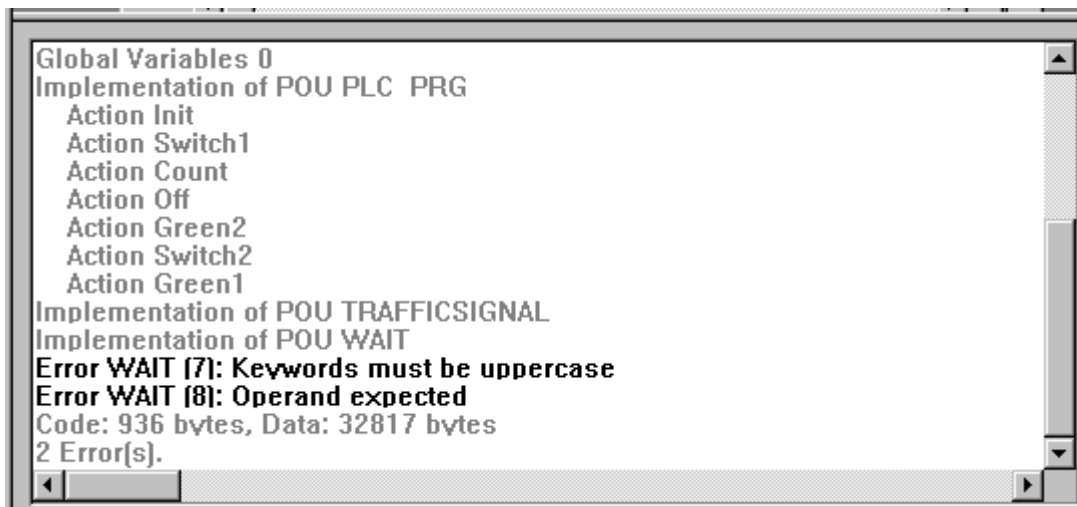


Figure 4\_21: Message window of a project with three blocks and two error messages

When the command **'Online' 'Login'** is used, the command **'Rebuild all'** is executed automatically if the project has been changed since the last compilation.

If the option **Save before compile** is selected in the **'Build'** category of the options dialog, the project is saved before compilation.



**Note:** The cross-references are created during compilation and are not saved in the project. In order to use the commands **'Show call tree'**, **'Show cross-reference'** and **'Show unused variables'**, the project must be recompiled after loading and after a change.

### **'Project' 'Document'**

This command allows you to print the documentation for your entire project. The following elements are part of the complete documentation:

- the POU,
- the contents of the documentation,
- the data types,
- the visualizations,
- the resources (retain variables, global variables, variable configuration, trace recording, controller configuration, task configuration, watch and receipt managers)
- the call trees of blocks and data types, and
- the cross-reference list.

For the last two elements, the project must have been compiled without errors.

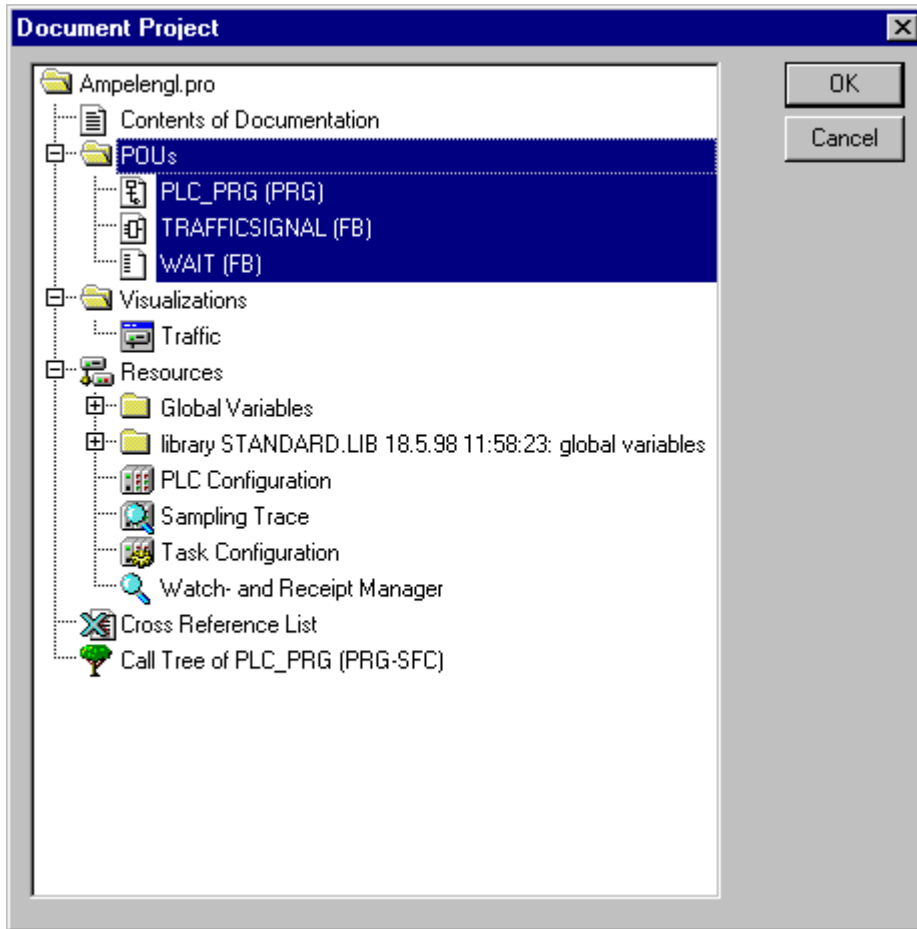


Figure 4\_22: Dialog for project documentation

The areas selected in the dialog that are highlighted in blue are printed out.

If you want to select the entire project, select the name of your project in the first line.

However, if you only want to select a single object, click on the corresponding object or move the dotted rectangle to the required object using the arrow keys. Objects with a plus sign in front of their icons are organisation objects, which contain other objects. When you click on the plus sign, the organisation object is opened. It can be closed again by clicking on the minus sign that now appears. If you select an organisation object, all relevant objects are also selected. By holding down the <Shift> key, you can select a sequence of objects. By holding down the <Ctrl> key, you can select several individual objects.

When you have made your selection, click on **OK**. A dialog appears for printing. The layout of the pages to be printed can be defined using 'File' 'Printer Setup'.

### 'Project' 'Export'

**CP1131** offers the option of exporting or importing projects. This allows you to exchange programs between different IEC programming systems.

Up to now, there has been a standardised exchange format for blocks in IL, ST and SFC (the Common Elements Format of IEC 1131-3). **CP1131** has its own storage format for blocks in LD and FBD and the other objects, as there is no textual format for this in IEC 1131-3. The selected objects are written to an ASCII file.

Blocks, data types, visualizations and resources can be exported. When you have made your selection in the dialog window (selection is the same as that described for '**Project' 'Document'**), click on **OK**. The dialog for saving files appears. Specify a file name with the extension ".exp".

### 'Project' 'Import'

Select the required export file in the dialog that appears for opening files.

The data is imported to the current project. If an object of the same name already exists in the project, a dialog box appears with the question "Do you want to replace it?". If the answer is **Yes**, the object in the project is replaced by the object from the import file. If the answer is **No**, the name of the new object receives an underscore and a counter number ("\_0", "\_1", ...) as an extension. If the answer is **Yes, all** or **No, all**, this is done for all objects.

The import is recorded in the message window.

### 'Project' 'Compare'

Using this command, you can compare the opened project with another project. For example, if you want to know where you made changes in the current project before saving it, you can compare the current version of the project with the previous version saved.

After selection of this command, the dialog for opening files is displayed. Find the project you want to compare with the current project. When you press **OK**, the result of the comparison is displayed in the message window. All objects of the selected project are listed, followed by the change to the object in brackets. Five messages are possible:

- "unchanged": the object has not been changed
- "deleted": the object no longer exists in the current project
- "implementation changed": the instruction part of the block has been changed
- "interface changed": the declaration part of the object has been changed
- "interface and implementation changed": both the instruction part and the declaration part of the block have been changed.

The first change in this object is selected by double-clicking on a message.

### 'Project' 'Merge'

Using this command, you can copy objects (blocks, data types, visualizations and resources) from other projects into your project.

When the command is executed, the standard dialog for opening files is displayed first. If you select a file there, a dialog opens in which you can select the required objects. Selection is the same as that described in **'Project' 'Document'**.

If an object of the same name already exists in the project, the name of the new object receives an underscore and a counter number ("\_1", "\_2" ...) as an extension.

### 'Project' 'Project Info'

You can store information relating to your project under this menu item. The following dialog box opens when the command is executed:

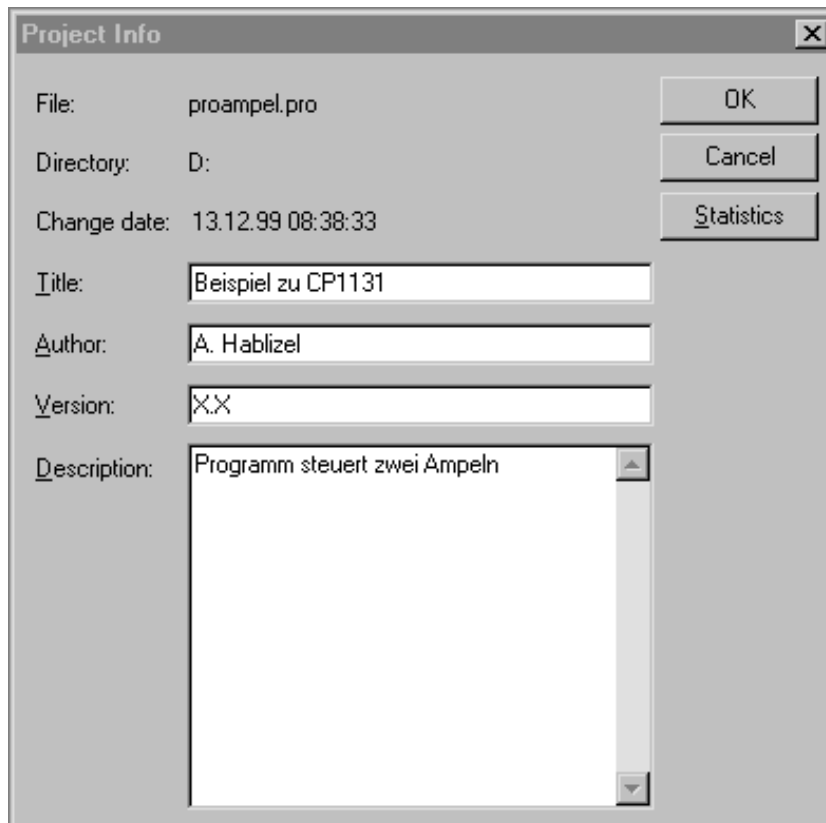


Figure 4\_23: Dialog for entering project information

The following specifications are displayed under project information:

- file name
- directory path
- the time of the last change (last modified)

These specifications cannot be changed.

You can also add the following specifications of your own:

- the name of the project,
- the name of the author,
- the version number and
- a description of the project.

These specifications are optional. You can obtain statistical information on the project by pressing the **Statistics** button.

This contains the specifications from the project information, together with the number of blocks, data types, and local and global variables as they were recorded during the last compilation.

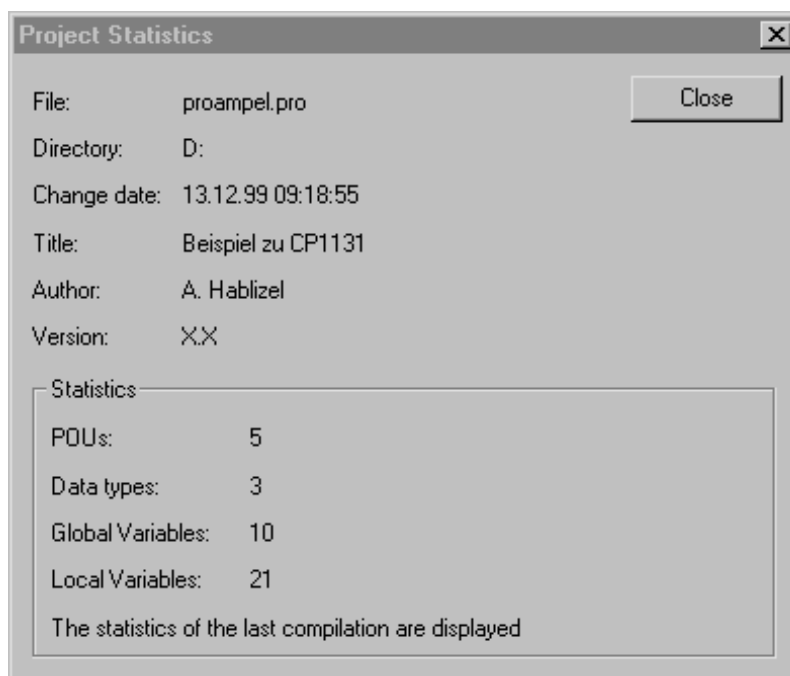


Figure 4\_24: Example of project statistics

If you select the **Ask for project inof** option in the category **Load & Save** in the options dialog, the project information is called up automatically when a new project is saved or when a project is saved under another name.

### **'Project' 'Global Search'**

Using this command, you can search for occurrences of a text in blocks, data types or the objects of the global variables. When the command is executed, a dialog opens in which you can select the required objects. Selection is the same as that described in '**Project** **Document**'.

When you have confirmed the selection by clicking **OK**, the search dialog is displayed. If a text is found in an object, the object is loaded in the corresponding editor and the location is displayed. The display of the text found and the '**Search**' and '**Continue search**' processes are the same as for the command '**Edit**' **Find**'.

**‘Project’ ‘Global Replace’**

Using this command, you can search for the occurrence of a text in blocks, data types or the objects of the global variables, and replace this text with another. Operation and execution are the same as for ‘Project’ ‘Global find’ and ‘Edit’ ‘Replace’.

**‘Project’ ‘Register Changes’**

This command is required if major changes must be made to a project but the controller cannot be stopped (online change).

Copy the project, make your changes and test the changes. Use the command ‘Project’ ‘Compare’ to compare the two projects. With the command ‘Register Changes’, all differences between the current project and the comparison project are recorded. The ‘Build’ command can then be used to compile the changed blocks. When the program is downloaded, only the changed blocks are transferred to the controller. The rest of the program remains unchanged in the controller.

**User Groups**

Up to eight groups with different access rights to blocks, data types, visualizations and resources can be set up in CP1131. Access rights can be defined for individual objects or for all objects. When you open a project, you must do so as a member of a particular user group. You must enter a password as authorisation to open the project.

The user groups are numbered from 0 to 7, with the group 0 having administration rights, i.e. only members of the group 0 may define passwords and access rights for all groups or objects.

When a new project is created, all passwords are empty at first. So long as no password has been defined for the group 0, you are automatically a member of group 0 when you open the project.

If a password for the user group 0 is defined during loading of the project, entry of a password is requested from *all* groups before opening of the project. The following dialog is displayed:

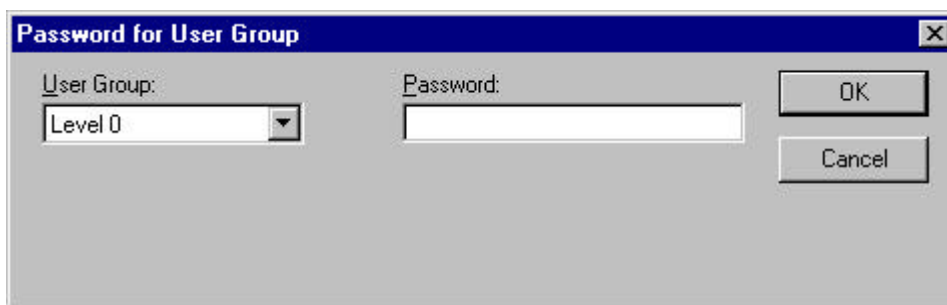


Figure 4\_25: Dialog for password entry

In the combobox **User Group** on the left-hand side of the dialog, enter the group to which you belong, and on the right-hand side, enter the corresponding **Password**. Press **OK**. If the password does not match the password saved, the following message is displayed:

“The password is incorrect.”

The project will not open until you have entered the correct password.

### **‘Project’ ‘User Group Passwords**

This command opens the dialog for password assignment for user groups. It can only be executed by members of group 0. When the command is executed, the following dialog opens:

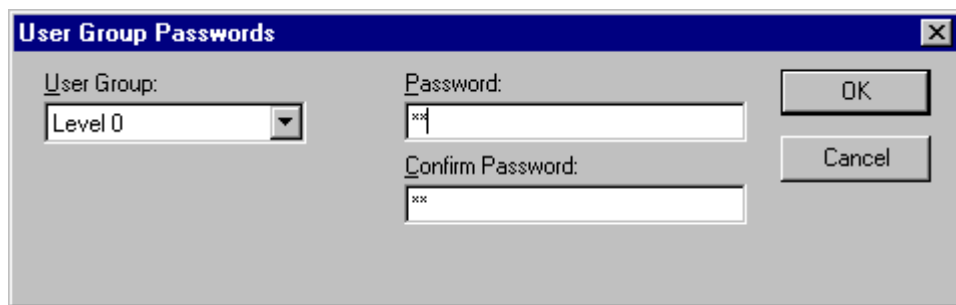


Figure 4\_26: Dialog for password allocation

You can select the group in the left-hand combobox **User Group**. Enter the password required for the group in the **Password** field. An asterisk (\*) appears in the field for each letter typed. You must type the same word again in the **Confirm password** field. Close the dialog after each password entry by clicking **OK**. If the message:

“The password and its confirmation do not match”

appears, you have made a typing error in one of the two entries. It is best to repeat both entries until the dialog closes without any message.

If necessary, you can allocate a password to the next group by calling the command again.

## Objects: Creating, Deleting, etc.

The following section describes how to use objects and what tools are available to help you maintain an overview of the project (folders, call trees, cross-reference lists, etc.).



### Object

The following are considered to be “objects”: blocks, data types, visualizations and resources (access variables, global variables, variable configurations, trace recording, control configurations, task configurations and watch and receipt managers). The folders used for structuring your project are partly implied. All the objects in a project are found in the Object Organizer.

If you hold the mouse pointer over a block in the Object Organizer for a short time, the type of block (program, function or function block) is displayed in a tooltip.

### Folder

To retain an overview of large projects, you should group your blocks, data types, visualizations and global variables into folders.

Folders can be nested any number of times. If there is a plus sign in front of the closed folder icon , this folder contains objects and/or other folders. The folder is opened by clicking on the plus sign and the subordinate objects are displayed. By clicking on the minus sign  which now appears, the folder is closed again. The **‘Expand node’** and **‘Collapse node’** commands in the context-sensitive menu have the same functionality.

You can use Drag&Drop to move objects and even folders within their object type. To do this, select the object and move it to the required location while holding down the left-hand mouse button.



**Note:** Folders do not influence the program in any way. They simply help you to structure your project more clearly.



Figure 4\_27: Example of folders in the Object Organizer

### **'New Folder'**

This command is used to insert a new folder as an organisation object. If a folder is selected, the new folder is inserted below this folder. Otherwise, it is inserted on the same level.

The context-sensitive menu of the Object Organizer which contains this command appears if an object or the object type is selected and you press the right-hand mouse button or <Shift>+<F10>.

### **'Expand Node' 'Collapse Node'**

When the **'Expand'** command is executed, all objects subordinate to the selected object are displayed. When **'Collapse'** is executed, they disappear again.

The same can be done with folders by double-clicking or pressing the <Enter> key.

The context-sensitive menu of the Object Organizer which contains these commands appears when the object or object type is selected and the right-hand mouse button or <Shift>+<F10> is pressed.

### **'Project' 'Delete Object'**

**Shortcut: <Del>**

This command removes the currently-selected object (block, data type, visualization or global variables) or folder with subordinate objects from the Object Organizer, thereby deleting it from the project. As a safety measure, you are asked to confirm the deletion before it is carried out.

If the editor window was open, it is closed automatically.

If you use the command 'Edit' 'Cut' for deletion, the object is moved to the clipboard.

### **'Project' 'Add Object'**

**Shortcut: <Insert>**

This command is used to create a new object. The type of object (block, data type, visualization or global variables) depends on the tab selected in the Object Organizer.

Enter the name of the new object in the dialog box displayed. Remember that the name of the object must not be already in use by another object.

If the object is a block, the type of block (program, function or function block) and the language in which it is to be programmed must also be selected.

After confirmation of the entry, the input window corresponding to the object is displayed.

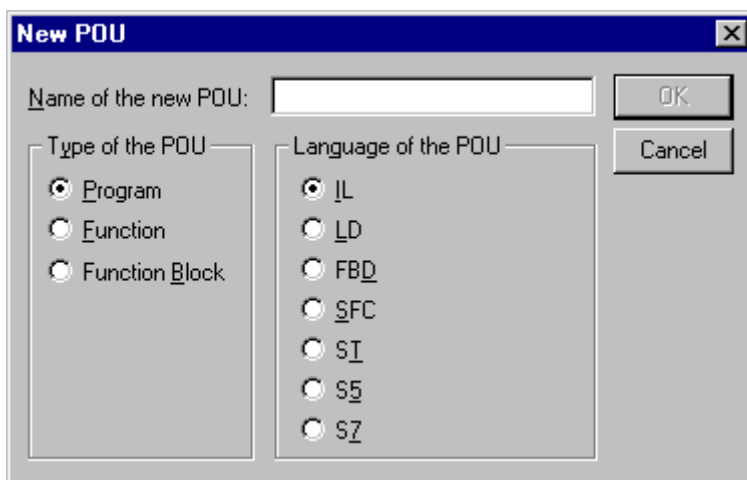


Figure 4\_28: Dialog for creating a new block

On the other hand, if the command **'Edit' 'Paste'** is used, the object is pasted from the clipboard and no dialog appears.

### **'Project' 'Rename Object'**

**Shortcut: <Spacebar>**

This command is used to rename the currently-selected object or folder. Remember that the name of the object must not be already in use.

If the editing window of the object is open, its title changes automatically when the object is renamed.

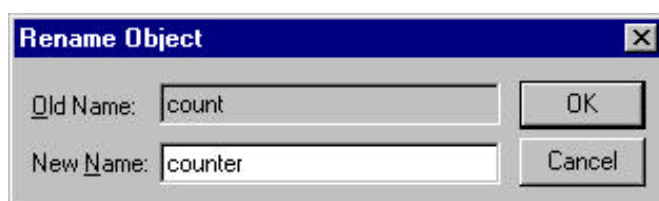


Figure 4\_29: Dialog for renaming a block

### **'Project' 'Convert Object'**

This command can only be used with blocks. You can convert blocks in the languages ST, FBD, LD and IL into one of the three languages IL, FBD or LD.

In order to do this, the program must be compiled. Select the language into which you want to convert and assign a new name to the new block. Remember that the new name of the block must not be already in use by another block. You can then press **OK** and the new block is added to your block list.

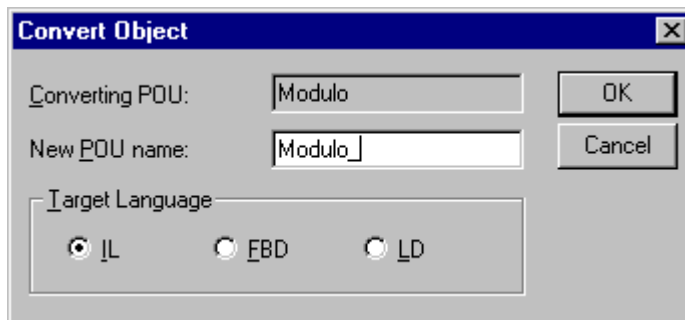


Figure 4\_30: Dialog for converting a block

### 'Project' 'Copy Object'

This command copies a selected object and saves it under a new name. Enter the name of the new object in the dialog displayed. Remember that the name of the object must not be already in use.

On the other hand, if the command '**Edit**' '**Copy**' is used, the object is copied to the clipboard and no dialog is displayed.

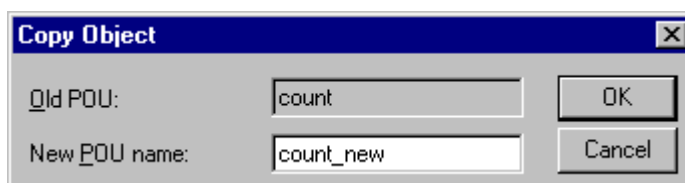


Figure 4\_31: Dialog for copying a block

### 'Project' 'Open Object'

**Shortcut: <Enter> key**

This command loads an object selected in the Object Organizer in the relevant editor. If there is already a window with this object open, it comes into focus, i.e. it is placed in the foreground and can now be edited.

There are two other ways of editing an object:

- by double-clicking with the mouse on the required object or
- by typing the first letter of the object name in the Object Organizer. This opens a dialog displaying all objects of the object type set beginning with this letter which are available for selection. Select the required object and click on the Open button to load the object to its editing window. In the case of the object type 'resources', this option is only supported for global variables.

The last option is particularly useful in projects containing a large number of objects.

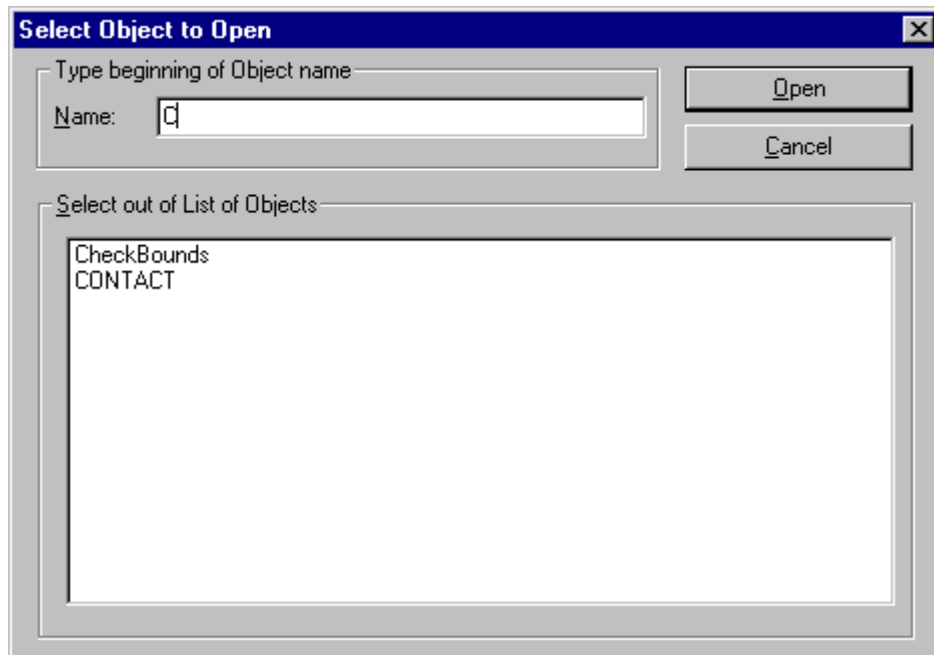


Figure 4\_32: Dialog for selecting the object to be opened

### **'Project' 'Object Access Rights'**

This command opens the dialog for assigning access rights to the various user groups. The following dialog is displayed:

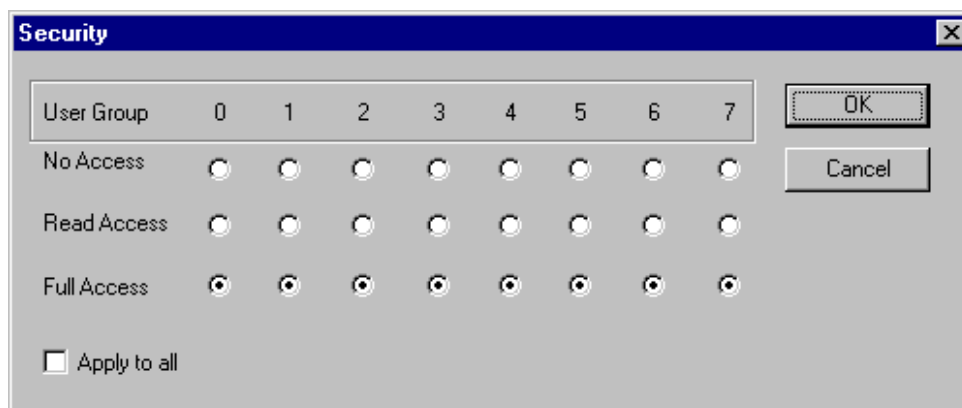


Figure 4\_33: Dialog for assigning access rights

Members of the working group 0 can now assign access rights individually to each user group. Three settings are possible:

- No access: The object cannot be opened by a member of the user group.
- Read access: The object can be opened by a member of the user group for reading, but it cannot be edited.
- Full access: The object can be opened and edited by a member of the user group.

The settings relate either to the currently selected object in the Object Organizer, or, if the option **Apply to all** is selected, to all the blocks, data types, visualizations and resources in the project.

Assignment to a user group occurs when the project is being opened by means of a request for a password, provided that a password has been assigned to the user group 0.

### **'Project' 'View Instance'**

This command can be used to open and display individual instances of function blocks. The function block whose instance is to be opened must first be selected in the Object Organizer before you execute this command. You can now select the required instance of this function block in the dialog that appears.



**Note:** Instances cannot be opened until after login (the project has been compiled correctly and transferred to the controller using **'Online' 'Login'**).

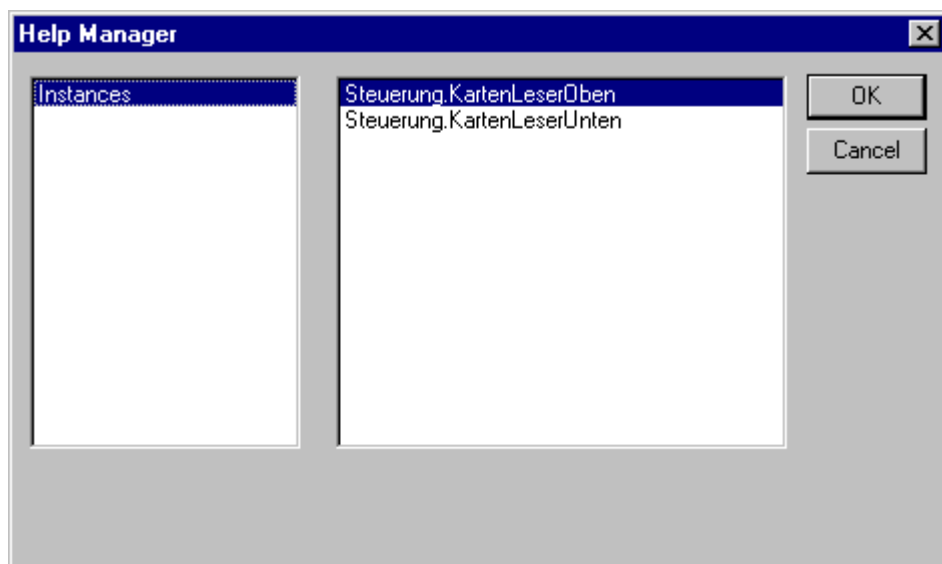


Figure 4\_34: Dialog for opening an instance

### 'Project' 'Show Call Tree'

This command opens a window in which the call tree of the object selected in the Object Organizer is displayed. The project must be already compiled. The call tree contains both calls of blocks and references to the data types used.

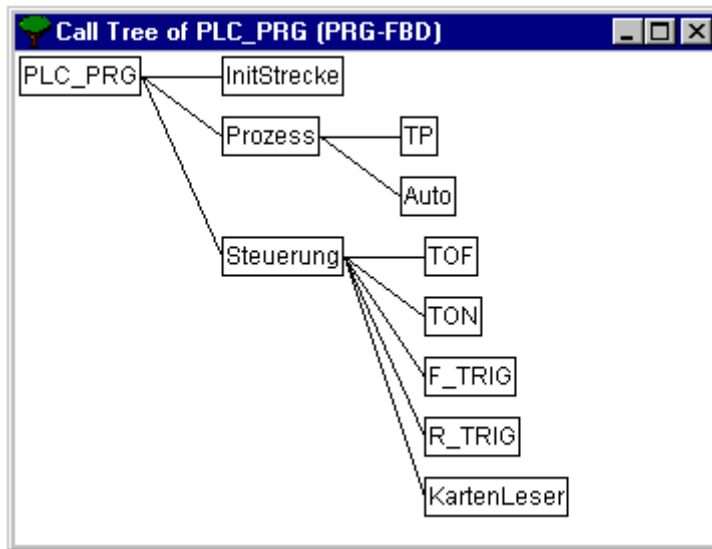


Figure 4\_35: Example of a call tree

### 'Project' 'Show Cross-Reference'

This command opens a dialog which permits output of all locations where a variable, address or block is used. The project must be already compiled.

First select the category **Variable**, **Address** or **Block**, and then enter the name of the required element. By clicking on the **Cross-references** button, you obtain a list of all locations where the element is used. In addition to output of the block and the line or network number, you are also informed whether read or write access is available at the location, and whether the variable is local or global.

If you select a line in the cross-reference list and press the **Go to** button, or double-click on the line, the block is displayed in its editor at the corresponding location. In this way, you can jump to all usage locations without having to carry out searches.

In order to simplify handling, you can use the **To the message window** button to transfer the current cross-reference list to the message window and switch from there to the relevant block.

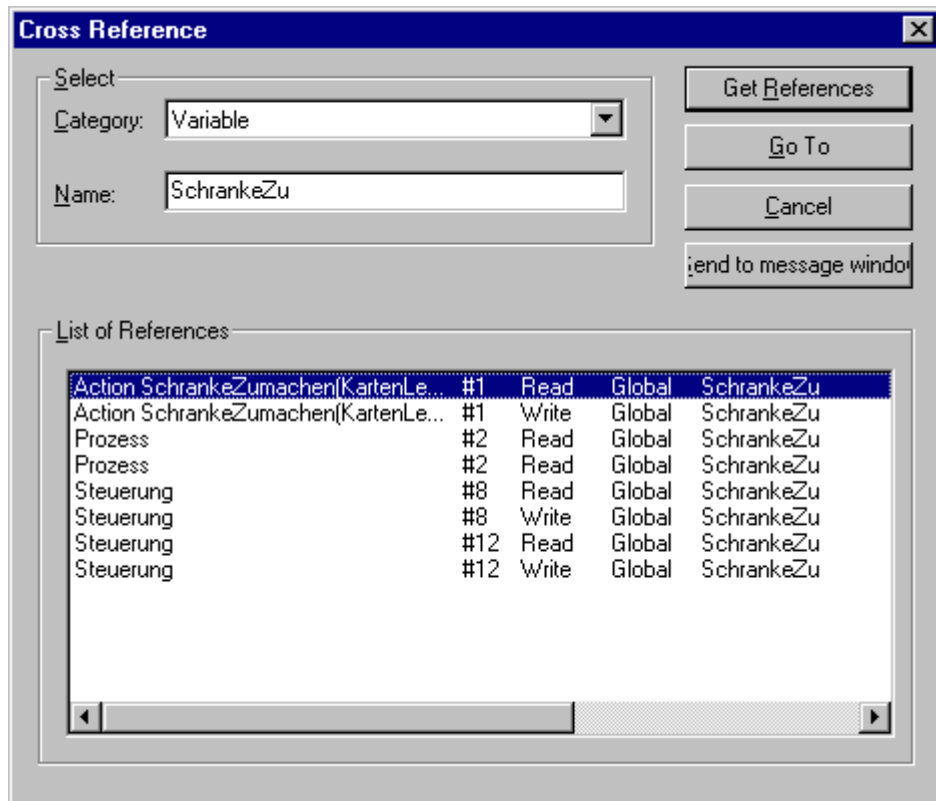


Figure 4\_36: Dialog and example of a cross-reference list

**'Project' 'Show Unused Variables'**

This command outputs a list of all variables that were declared in the project but were not used anywhere. The project must be already compiled.

If there are no unused variables in your project, this is reported. Otherwise, the following window opens:

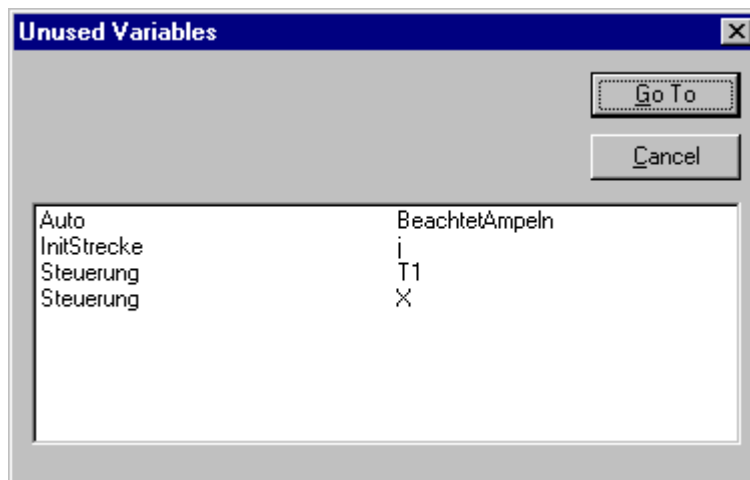


Figure 4\_37: Unused variables in a project

If you select a variable and press the **Go to** button or double-click on the variable, you are switched to the object where the variable was declared.

### 'Extras' 'Previous Version'

This command allows you to reset the object you are currently editing to the status at the last save. The status restored is that retained either at the last manual save ('File' 'Save') or at the automatic save, depending on which version is more recent.

---

## General editing functions

All of the following commands are available in the editors, and some of them are available in the Object Organizer. They are all found under the menu item 'Edit'.

### 'Edit' 'Undo'

**Shortcut:** <Ctrl>+<Z>

This command undoes the last action executed in the currently opened editor window or in the Object Organizer.

Repeated execution of this command undoes all actions back to the time the window was opened. This applies to all actions in the editors for blocks, data types, visualizations and global variables and in the Object Organizer.

'Edit' 'Redo' allows you to re-execute the action you have just undone.

### 'Edit' 'Redo'

**Shortcut:** <Ctrl>+<Y>

This command allows you to redo the action you have just undone ('Edit' 'Undo') in the currently opened editor window or in the Object Organizer.

You can execute the 'Redo' command as often as you have executed the 'Undo' command.



**Note:** The commands 'Undo' and 'Redo' relate to the current window. Each window has its own action list. If you want to undo actions executed in several windows, you must activate the corresponding window. When undoing or redoing actions in the Object Organizer, it must be in focus.

## 'Edit' 'Cut'

Icon: 

**Shortcut:** <Ctrl>+<X> or <Shift>+<Del>

This command moves whatever is currently selected from the editor to the clipboard. The selection disappears from the editor.

In the Object Organizer, this depends on the selected object, as not all objects can be deleted, e.g. the control configuration.

Note that not all editors support cutting, and the tool may be limited in some editors.

The type of selection depends on the relevant editor:

In the text editors (IL, ST, declarations), the selection is a list of characters.

In the FBD and LD editors, the selection is a quantity of networks, each marked by a dotted rectangle in the network figures field, or a box containing all previous lines, boxes and operands.

In the SFC editor, the selection is part of a sequence of steps, surrounded by a dotted rectangle.

To paste the contents of the clipboard, use the command **'Edit' 'Paste'**. In the SFC editor, you can also use the commands **'Tools' 'Insert parallel branch (right)'** and **'Tools' 'Insert after'**.

To insert a selection into the clipboard without deleting it, use the command **'Edit' 'Copy'**.

To delete a selected area without changing the clipboard, use the command **'Edit' 'Delete'**.

## 'Edit' 'Copy'

Icon:  **Shortcut:** <Ctrl>+<C>

This command copies the current selection from the editor to the clipboard. The contents of the editor window therefore remain unchanged.

In the Object Organizer, this depends on the selected object, as not all objects can be copied, e.g. the control configuration.

Note that not all editors support copying, and the tool may be limited in some editors.

The same rules as those described under **'Edit' 'Cut'** apply to selection types.

## 'Edit' 'Paste'

Icon:  **Shortcut: <Ctrl>+<V>**

This command inserts the contents of the clipboard into the current position in the editor window. This command can only be executed in the graphical editors if the insertion maintains a correct structure.

In the Object Organizer, the object is inserted from the clipboard.

Remember that insertion is not supported by all editors, and the tool may be limited in some editors.

The current position has different definitions, depending on the type of editor:

In the text editors (IL, ST, declarations), the current position is the position of the flashing cursor (a small vertical line that can be positioned by means of a mouse click).

In the FBD and LD editors, the current position is the first network with a dotted rectangle in the network number range. The contents of the clipboard are inserted before this network. If a partial structure is copied, this is inserted before the selected element.

In the SFC editor, the current position is defined as the selection surrounded by a dotted rectangle. The contents of the clipboard are inserted before the selection, depending on the selection and the contents of the clipboard, or in a new branch (parallel or alternative) to the left of the selection.

In SFC, the commands '**Tools' 'Insert parallel branch (right)'** and '**Tools' 'Insert after'** can also be used to insert the contents of the clipboard.

## 'Edit' 'Delete'

**Shortcut: <Del>**

This command deletes the selected area from the editor window. The contents of the clipboard remain unchanged.

In the Object Organizer, this depends on the selected object, as not all objects can be cut, e.g. the control configuration.

The same rules as those described under '**Edit' 'Cut'** apply to selection types.

In the Library Manager, the selection is the currently selected library name.

## 'Edit' 'Find'

Icon: 

This command allows you to search for a particular text location in the current editor window. The **'find'** dialog opens. It remains open until the **Cancel** button is pressed.

You can enter the character string to be located in the **Find what** field.

You can also select whether you want to search for the text **as a whole word** or as part of a word, whether **upper/lowercase** is to be observed in the search and whether the search is to be carried out **upwards** or **downwards** in relation to the current cursor position.

The **Find next** button starts the search. It begins at the selected position and continues in the search direction selected. When the text location is found, it is selected. If the text location is not found, this is reported. The search can be executed several times in sequence until the start or end of the contents of the editor window is reached.

Note that the text found may be covered by the **'Find'** dialog.

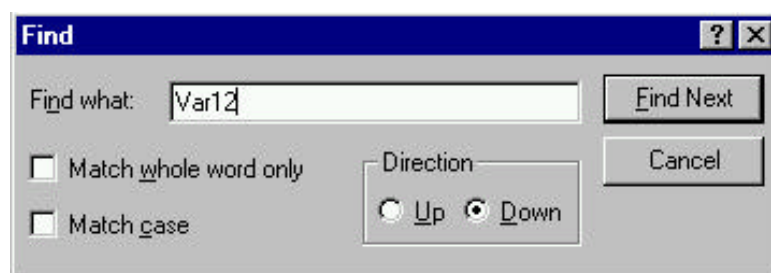


Figure 4\_38: Dialog for finding text

## 'Edit' 'Find Next'

Icon:  Shortcut: <F3>

This command executes a search command using the same parameters as those used in the last execution of the command **'Edit' 'Find'**.

## 'Edit' 'Replace'

This command allows you to search for a certain text location as with the command **'Edit' 'Find'** and replace it with another. After you have selected the command, the find and replace dialog opens. This dialog remains open until the **Cancel** or **Close** button is pressed.

The **Replace** button replaces the current selection by the text in the field **Replace with**.

The **Replace all** button replaces all occurrences of the text in the **Find what** field with the text in the **Replace with** field. After the replacement process, a message reports how many times the text was replaced.

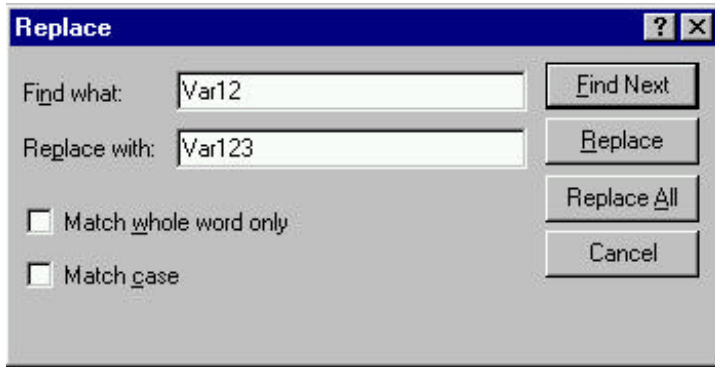


Figure 4\_39: Dialog for find and replace

### 'Edit' 'Input Assistant'

**Shortcut: <F2>**

This command provides you with a dialog for selection from possible inputs at the current cursor position in the editor window. Select the category of input required in the left-hand window, select the required input in the right-hand column and confirm your selection by clicking **OK**. Your selection is inserted at this position.

The categories offered are dependent on the current cursor position in the editor window, i.e. on what can be entered at this position (e.g. variables, operators, blocks, conversions, etc.).

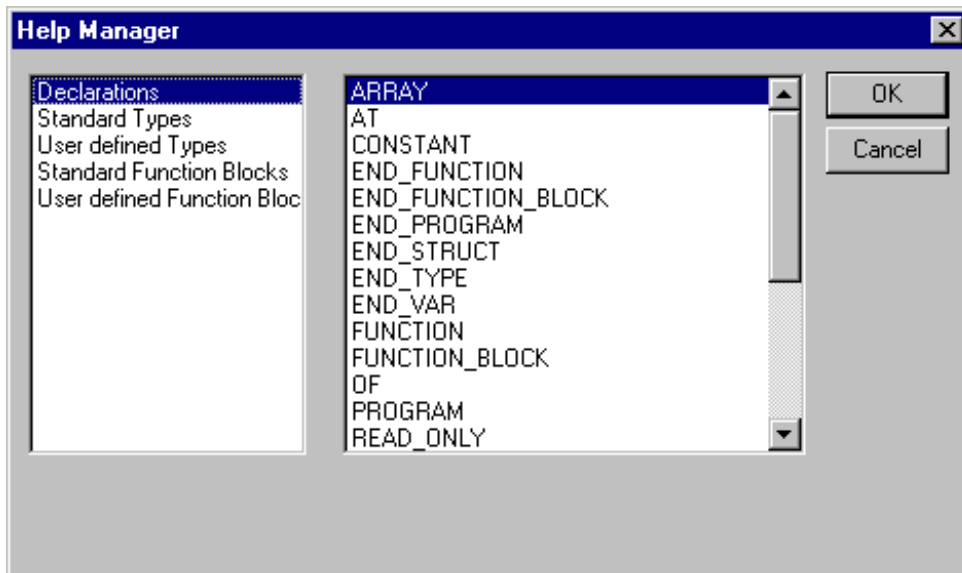


Figure 4\_40: Dialog for input help

Multi-level variable names are required at some positions (e.g. in the watch list). The dialog for input help first contains a list of all blocks plus a single dot for the global variables. There is a dot after each block name. Double-clicking or pressing the <Enter> key opens the list of variables for a selected block. Instances and data types can also be displayed if necessary. The selected variable is imported by pressing **OK**.

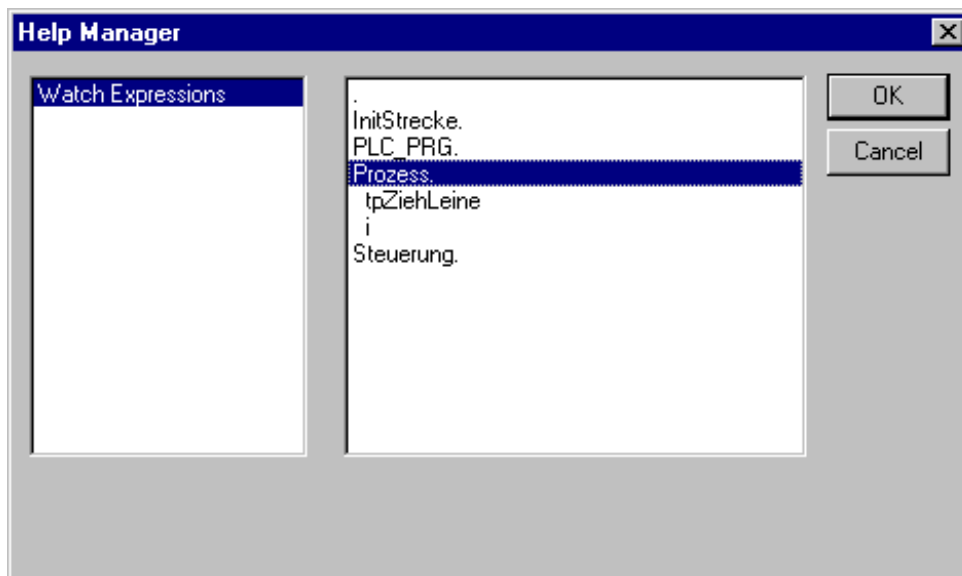


Figure 4\_41: Dialog for input help in the case of multi-level variable names



**Note:** Some entries (e.g. global variables) are not updated in the input help until after a compilation.

### 'Edit' 'Next Error'

**Shortcut:** <F4>

If the compilation of a project contains errors, this command can be used to display the next error. The corresponding editor window is activated, and the location of the error is marked. At the same time, the corresponding error message is displayed in reverse video in the message window.

### 'Edit' 'Previous Error'

**Shortcut:** <Shift>+<F4>

If the compilation of a project contains errors, this command can be used to display the previous error. The corresponding editor window is activated, and the location of the error is selected. At the same time, the corresponding error message is displayed in reverse video in the message window.

---

## General Online Functions

The online commands available are found under the menu item '**Online**'. Execution of individual commands depends on which editor is active.

Online commands are not available until after login.

### 'Online' 'Login'

Icon: 

This command connects the programming system to the controller (or starts the simulation program) and switches to online mode.

If the current project has not been compiled since it was opened or last edited, it is now compiled (as in '**Project**' '**Rebuild all**'). If errors occur during compilation, **CP1131** does not switch to online mode.

After successful login, all online functions are available (provided that the corresponding settings have been entered in '**Options**' category **Build**). The current values of all variable declarations visible on the screen are monitored.

Use the command '**Online**' '**Logoff**' to switch from online mode back to offline mode.

### If the System Reports

the following error:

"Incorrect interface initialisation"

check whether the parameters set in '**Online**' '**Communication options**' match those of your controller.

You should check in particular whether the interfaces (CAN, serial, Ethernet, modem) are correct and the node number of the block is set (for example, if you have set COM1, the cable must be physically connected to COM1).

If the system reports the following error:

"The program has been changed! Do you want to load the new program?"

the current project in the editor does not match the program currently loaded in the controller (or the simulation program that has been started). Monitoring and debugging are therefore impossible. You can now select "No", log off again and open the correct project, or select "Yes" to load the current project to the controller.

### 'Online' 'Logoff'

Icon: 

The connection to the controller is terminated or the simulation program is ended, and you are switched to offline mode.

Use the command '**Online**' '**Login**' to switch back to online mode.

### 'Online' 'Download'

This command loads the compiled project to the controller.

### 'Online' 'Run'

Icon:  Shortcut: <F5>

This command starts processing of the user program in the controller or in the simulation.

The command can be executed immediately after the command '**Online**' '**Load**', after the user program has been stopped in the controller using the command '**Online**' '**Stop**', when the user program is at a breakpoint or when a single cycle has been executed.

### 'Online' 'Stop'

Icon: 

This command stops processing of the user program in the controller or in the simulation between two cycles.

Use the command '**Online**' '**Run**' to continue processing of the programming.

### 'Online' 'Reset'

If you have initialised the variables with a particular value, this command resets them to the initialised values. All other variables are set to a standard initialisation (for example, integer numbers are set to 0). For safety purposes, **CP1131** asks you to confirm the overwrite before all variables are overwritten.

Use the command '**Online**' '**Run**' to restart processing of the program.

### 'Online' 'Toggle Breakpoint'

Icon:  **Shortcut: <F9>**

This command sets a breakpoint at the current position in the active window. If a breakpoint is already set at the current position, it is deleted.

The position at which a breakpoint can be set depends on the language in which the block in the active window has been written.

In the text editors (IL, ST), the breakpoint is set at the line in which the cursor is found if this line is a breakpoint position (recognisable by the dark-grey color of the line number field). You can also click on the line number field to set or delete a breakpoint in the text editors.

In FBD and LD, the breakpoint is set at the currently selected network. You can also click on the network number field to set or delete a breakpoint in the FBD or LD editor.

In SFC, the breakpoint is set at the currently selected step. You can also use <Shift> with a double-click to set or delete a breakpoint in SFC.

If a breakpoint has been set, the line number field, the network number field or the step is displayed on a light blue background.

When a breakpoint is reached during processing of a program, the program stops and the corresponding field is displayed on a red background. To continue to the program, use the commands '**Online**' '**Run**', '**Online**' '**Single step to**' or '**Online**' '**Single step over**'.

### 'Online' 'Breakpoint Dialog'

This command opens a dialog for editing breakpoints throughout the project. The dialog displays all currently-set breakpoints.

To set a breakpoint, select a block in the combobox **Block**, and in the combobox **Location**, select the line or network where you want to set the breakpoint. Then press the **Add** button to add the breakpoint to the list.

To delete a breakpoint, select the breakpoint to be deleted in the list of breakpoints set and press the **Delete** button.

The **Delete all** button deletes all breakpoints.

To go to the location in the editor where a certain breakpoint is set, select the corresponding breakpoint in the list of breakpoints set and press the **Go to** button.

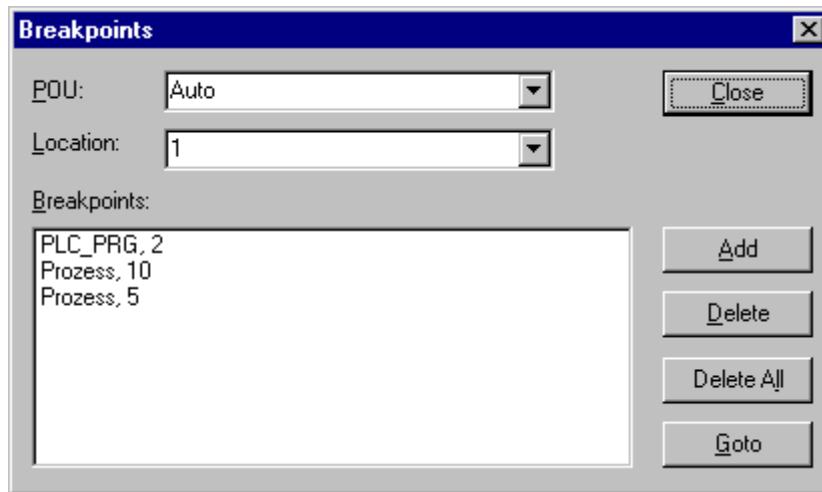


Figure 4\_42: Dialog for editing breakpoints

### 'Online' 'Step Over'

Icon:  **Shortcut: <F10>**

This command executes a single step in such a way that when blocks are called, there is no stop until after they have been processed. In SFC, a complete action is processed.

If the current instruction is the calling of a function or function block, the function or function block is executed in full. Use the command '**Online' 'Step In'**', in order to reach the first instruction of a function or function block called.

When the last instruction has been reached, the program continues to the next instruction of the calling block.

### 'Online' Step In'

**Shortcut: <F8>**

A single step is processed in such a way that in block calls, processing stops before execution of the first instruction of the block. You are switched to a called block if necessary.

If the current position is a call of a function or function block, the command continues to the first instruction of the block called.

In all other situations, the command acts in the same way as '**Online' 'Step over'**'.

### 'Online' 'Single cycle'

**Shortcut: <Ctrl>+<F5>**

This command executes a single control cycle and stops after this cycle. The command can be repeated continually in order to proceed in single cycles. The single cycle ends with execution of the command '**Online' 'Run'**'.

### 'Online' 'Write Values' or 'Force Values'

**Shortcut:** <Ctrl>+<F7> (write values)

**Shortcut:** <F7> (force values)

To change the value of a single-element variable, a double-click must first be executed in the line in which the variable is declared using the mouse, or the <Enter> key must be pressed. The new variable value can then be entered in the dialog box that appears. In the case of Boolean variables, the value is toggled without the dialog being displayed. The new value is displayed in red.

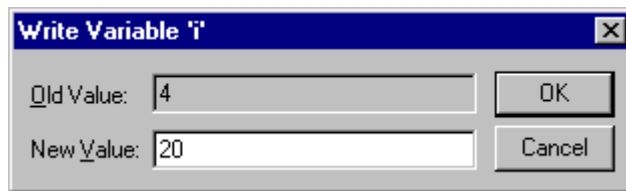


Figure 4\_43: Dialog for writing a new variable value

The new value is not written to the controller yet.

The values of a sequence of variables can now be set to a single value and then written together (cycle-consistently) to the controller.

With '**Write Values**', the values are written once and can be overwritten again immediately.

With '**Force values**', the values are written after each cycle until this process is stopped using '**Release Force**'.

### 'Online' 'Release Force'

**Shortcut:** <Shift>+<F7>

The command ends the forcing of variables in the controller. All forced variables change their values normally again.

If no values are available for forcing, the command has no effect.

### 'Online' 'Show Callstack'

You can start this command when the simulation stops at a breakpoint. A dialog is output containing a list of the blocks currently found in the callstack.

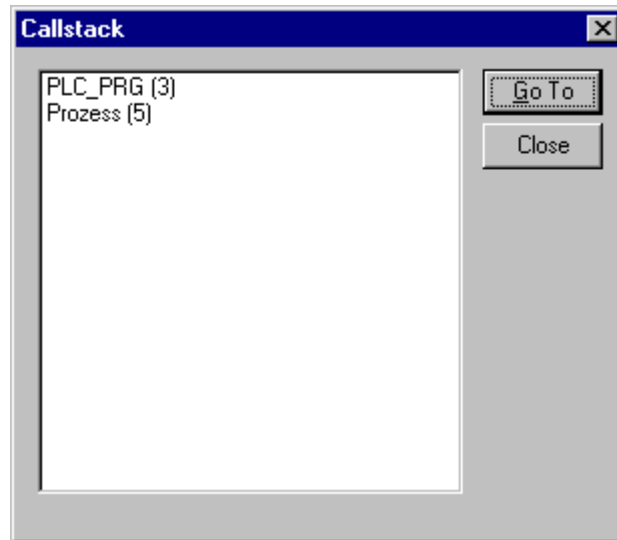


Figure 4\_44: Example of a call hierarchy

The first block is always PLC\_PRG, so processing begins here.

The last block is always the block in which processing is currently taking place.

After one of the blocks has been selected and the **Go to** button has been pressed, the selected block is loaded to a window and the line or network in which processing is taking place is displayed.

### 'Online' 'Display Flow Control'

If **Display Flow Control** has been selected, a check mark (✓) appears in front of the menu item. After that, every line or network executed during the previous control cycle is selected.

The line number field or the network number field of the line or network executed is displayed in green. In the IL editor, another field is inserted at the left-hand edge of each line in which the current contents of the accumulator are displayed. In the graphical editors of Function Block Diagram and Ladder Diagram, another field is inserted in all connection lines that are not transporting Boolean values. If these inputs and outputs are occupied, then the value being transported over the connection line is displayed in this field. Connection lines that only transport Boolean values are colored blue. If you transport TRUE, the flow of information can be traced on an ongoing basis.

### **'Online' 'Simulation Mode'**

If **Simulation Mode** is selected, a check mark (✓) appears in front of the menu item.

In simulation mode, the user program runs on the PC. This mode is used to test the project. Communication between the PC and the simulation uses the Windows message mechanism. The use of function blocks that are hardware-specific and that are supplied with CP1131 in special libraries cannot deliver the required result in simulation mode.

If the program is not in simulation mode, the program runs on the controller. Communication between the PC and the controller usually runs over the interfaces described under **'Online' 'Communication options'**.

The status of this flag is saved with the project.

### **'Online' 'Codegenerator Options'**

See the CP1131 system manual.

### **'Online' 'Communication Options'**

The parameters for transfer over the various interfaces can be entered in a dialog. It is important to ensure that these parameters correspond with those set in the controller.

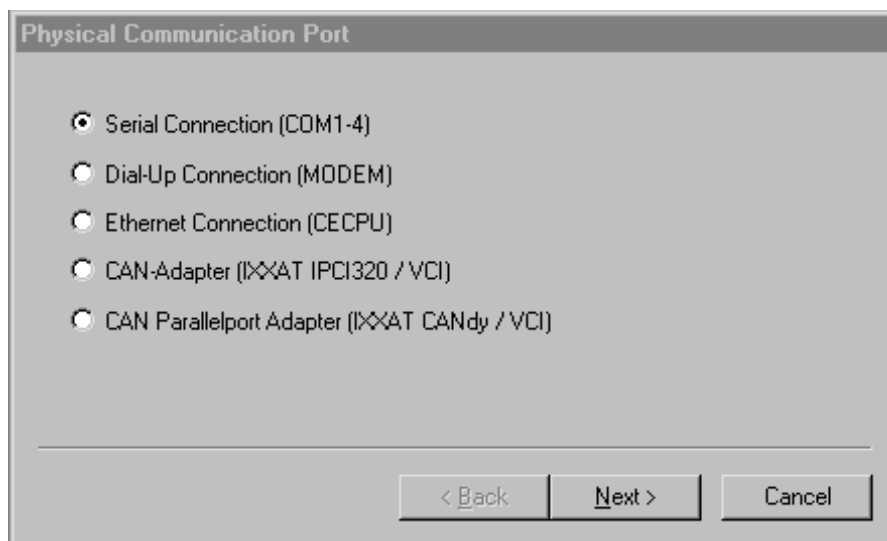


Figure 4\_45: Dialog for setting the communication parameters

Further information on the individual items can be found in the system manual.

### **'Online' 'Set Node ID'**

The node number of the CPU to be programmed can be set here. The project is loaded to the CPU identified in this way.

---

## **Arrange windows**

The menu item **Window** contains all the commands required for window management. This includes commands for arranging your windows automatically, as well as commands for opening the Library Manager and for switching between the open windows. At the end of the menu, there is a list of all open windows in the sequence in which they were opened. A mouse click on the corresponding entry switches you to the required window. A check mark (✓) appears in front of the active window.

### **'Windows' 'Tile Vertically'**

This command arranges all the windows in the workspace side by side, in such a way that they fill the workspace without overlapping.

### **'Windows' 'Tile Horizontal'**

This command arranges all the windows in the workspace one under another, in such a way that they fill the workspace without overlapping.

### **'Windows' 'Cascade'**

This command arranges all the windows one after another in a cascade.

### **'Windows' 'Arrange Symbols'**

This command arranges all the minimised windows in the workspace in a row at the lower edge of the workspace.

### **'Windows' 'Close All'**

This command closes all open windows in the workspace.

### **'Windows' 'Messages'**

**Shortcut: <Shift>+<Esc>**

This command opens or closes the message window containing the messages from the last compilation, testing or comparison process.

If the message window is open, a check mark (✓) appears in front of the command in the menu.

## The Help Tool

If you encounter any problems while using **CP1131**, there is an online help tool provided to sort them out. It contains all the information found in this manual.

### 'Help' 'Contents and Index'

This command opens the help topics window.

The tab **Contents** contains the table of contents. The books can be opened and closed by double-clicking or by pressing the relevant buttons. When you double-click on a selected topic or press the **Display** button, information on the topic is displayed in the main help window or in the keyword window.

Click on the tab **Index** to search for a particular keyword, and click on the **Find** tab to carry out a full text search. Follow the instructions in the tabs.

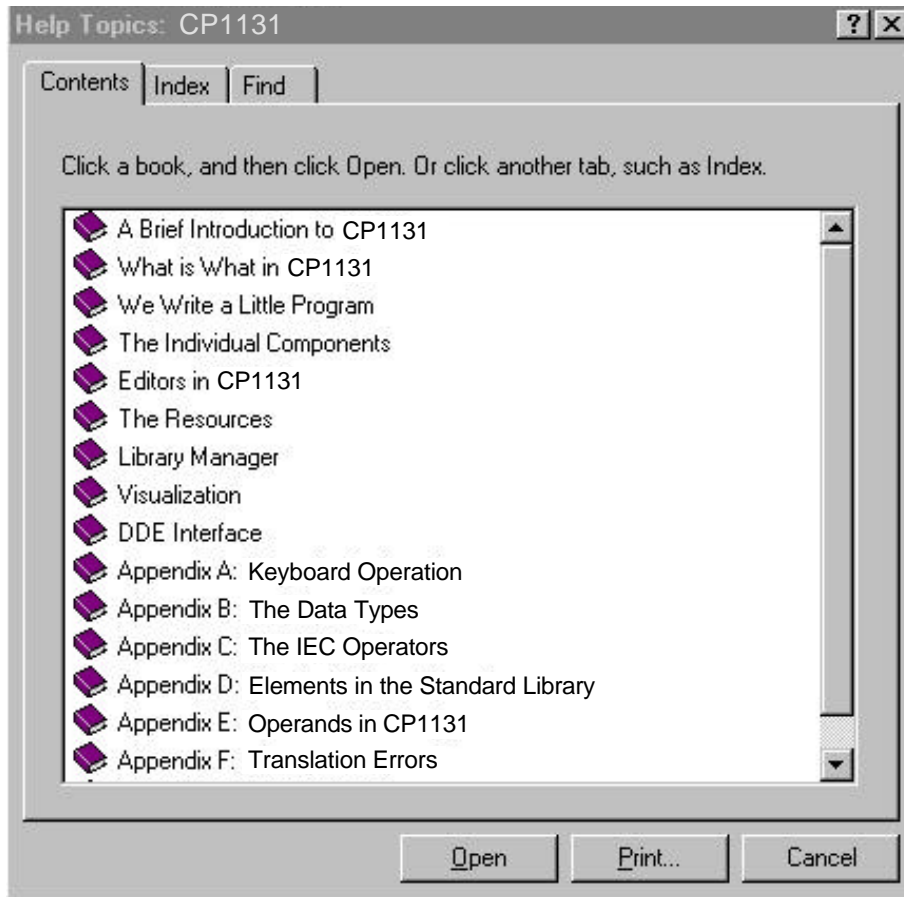


Figure 4\_46: Help topics window

## Main Help Window

The main help window displays topics with subordinate keywords.

The following buttons are available:

- **Help topics** opens the help topics window.
- **Back** displays the help entry that was displayed last.
- **Print** opens the dialog for printing.
- **<<** displays the help entry immediately before the current entry.
- **>>** displays the next help entry after the current entry.

The following menu commands can also be used:

- You can print out the current help entry using **'File' 'Print topic'**.
- By executing the command **'Edit' 'Copy'**, the selected text is copied to the clipboard. From there, you can insert the text into other applications and continue to use it there.
- By executing the command **'Edit' 'Annotate'**, a dialog opens. The left-hand side of the dialog contains an edit field in which you can record an annotation on the help page.

The right-hand side contains buttons to save the text, to cancel the application, to delete the annotation, to copy a selected text to the clipboard and to paste a text from the clipboard.

If you have made an annotation on a help entry, a small green paperclip appears in the top left. Clicking on the paperclip opens the dialog containing the annotation made.

- If you want to annotate one page of the help text, you can set a bookmark at the page. To do this, select the command **'Bookmark' 'Define'**. A dialog appears in which you can enter a new name (the presetting is the name of the page) or delete an old bookmark. If bookmarks have been defined, this is displayed in **the 'Bookmark'** menu. Selection of this menu item brings you to the required page.
- Under **'Options'** you can set whether the help window is to appear on top or not on top or according to the default.
- The command **'Display previous topics'** under **'Options'** displays a selection window containing the previously-displayed help topics. These are displayed by double-clicking on the entry.
- Depending on your requirements, you can select the **'font size'** small, normal or large under **'Options'**.
- If **'Options' 'Use system colors'** is selected, help is displayed in the system colors instead of in the set colors.

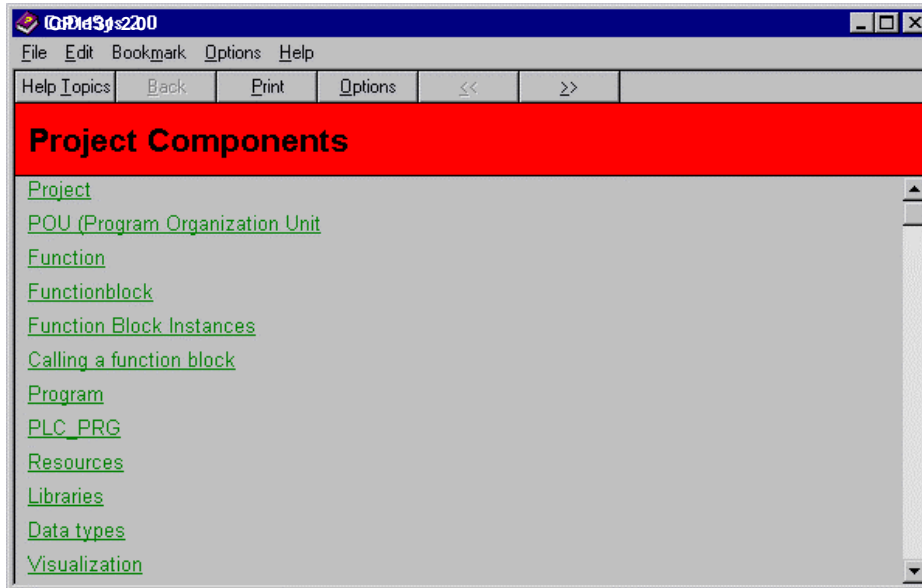


Figure 4\_47: Main help window

## Keyword Window

The keyword window explains menu commands, terms or actions. The keyword window is always on the interface by default, unless the option '**Help in background**' is set in the help main window.

The following buttons are available:

- **Help topics** opens the help topics window.
- **Back** displays the help entry that was displayed last.
- **Print** opens the dialog for printing.
- **<<** displays the help entry immediately before the current entry.
- **>>** displays the next help entry after the current entry.

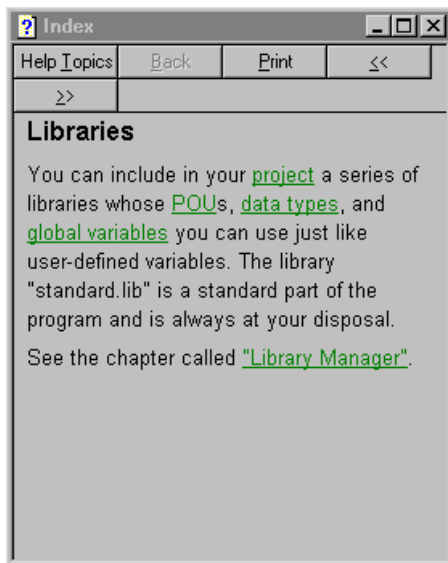


Figure 4\_48: Keyword window

**Context-Sensitive Help****Shortcut: <F1>**

You can press the <F1> key in an active window, a dialog or over a menu command. In the case of menu commands, help for the currently-highlighted command is displayed.

You can also select text (e.g. a keyword or a standard function) in order to display the help information for it.

Blank page

## The Editors in CP1131

### The Declaration Editor

Declaration editors are used in the variable declaration of blocks and global variables, for data type declaration and in the watch and receipt manager.

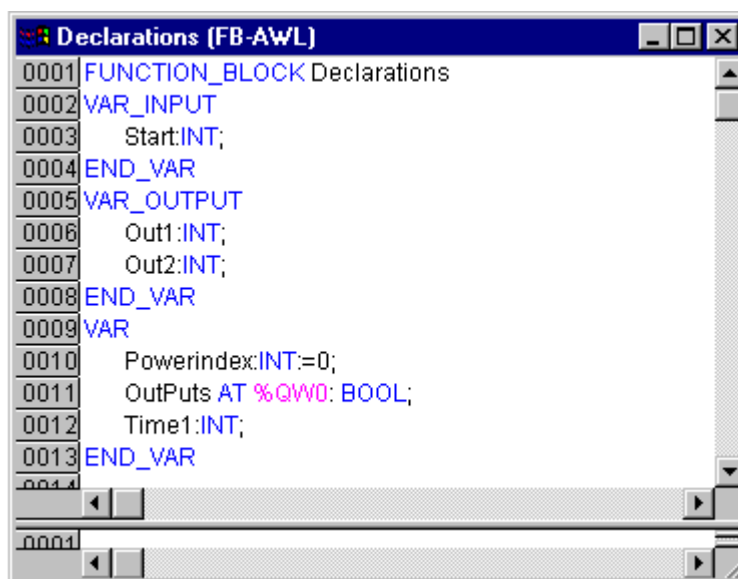
The variable declaration is supported by syntax coloring.

All editors for blocks consist of a declaration part and a body. These are separated by a screen split bar which can be moved if necessary by clicking it with the mouse and moving it upwards or downwards, while holding down the mouse button.

The most important commands are found in the context-sensitive menu (right-hand mouse button or <Ctrl>+<F10>).

#### Declaration Part

All variables used only in that block are declared in the declaration part of the block. These can be input variables, output variables, input/output variables, local variables, retentive variables or constants. The declaration syntax is based on the IEC1131-3 standard. The following is an example of a correct variable declaration in the **CP1131** editor:



```
0001 FUNCTION_BLOCK Declarations
0002 VAR_INPUT
0003     Start:INT;
0004 END_VAR
0005 VAR_OUTPUT
0006     Out1:INT;
0007     Out2:INT;
0008 END_VAR
0009 VAR
0010     Powerindex:INT:=0;
0011     OutPuts AT %QW0: BOOL;
0012     Time1:INT;
0013 END_VAR
```

Figure 5\_1: Declaration editor

### Input Variables

All variables used as input variables of a block are declared between the keywords **VAR\_INPUT** and **END\_VAR**. This means that the value of the variables can be specified in the call at the call position.

Example:

```
VAR_INPUT
  in1:INT;    (* first input variable *)
END_VAR
```

### Output Variables

All variables used as output variables of a block are declared between the keywords **VAR\_OUTPUT** and **END\_VAR**. This means that these values are returned to the calling block. They can be requested there and used further.

Example:

```
VAR_OUTPUT
  out1:INT;   (* first output variable *)
END_VAR
```

### Input/output Variables

All variables used as input and output variables of a block are declared between the keywords **VAR\_IN\_OUT** and **END\_VAR**.

**Warning:** In these variables, the value of the variable supplied is changed directly ("supply as pointer"). For this reason, the input value for such a variable cannot be a constant.

Example:

```
VAR_IN_OUT
  inout1:INT; (* first input/output variable *)
END_VAR
```

### Local Variables

All local variables of a block are declared between the keywords **VAR** and **END\_VAR**. These have no outward connection, which means that they cannot be accessed from outside.

Example:

```
VAR
  loc1:INT;   (* first local variable*)
END_VAR
```

### Retentive Variables

Retentive variables are identified by the keyword **RETAIN**. These variables retain their values, even after a power failure. The next time the program is started, processing continues with the saved values. An application example would be an operation time counter, which continues counting after a power failure.

All other variables are re-initialised, either with their initialised values or with the standard initialisations.

Example:

```
VAR RETAIN
  ret1:INT;    (* first retentive variable *)
END_VAR
```

### Constants

Constants are identified by the keyword **CONSTANT**. They can be declared locally or globally.

Syntax:

```
VAR CONSTANT
  <identifier>:<type> := <initialisation>;
END_VAR
```

Example:

```
VAR CONSTANT
  con1:INT:=12;    (* first constant *)
END_VAR
```

A list of possible constants can be found in the appendix.

### Keywords

Keywords must be written in uppercase in all editors. Keywords must not be used as variable names.

### Variable Declaration

A variable declaration has the following syntax:

```
<identifier> {AT <address>};<type> {:= <initialisation>;}
```

The parts in braces {} are optional.

Identifiers of variables must not contain any spaces, they must not be declared twice, and they must not be the same as keywords. There is no case sensitivity in variables, i.e. VAR1, Var1 and var1 all refer to the same variable. Underscores are significant in identifiers, e.g. "A\_BCD" and "AB\_CD" are interpreted as different identifiers. Several underscores in sequence at the start of an identifier or within an identifier are not permitted. The first 32 characters are significant.

All variable declarations and data type elements can contain initialisations. They use the assignment operator “ := ”. These initialisations are constants for variables of elementary types. The default initialisation for all declarations is 0.

Example:

```
var1:INT:=12;    (* integer variable with initial value 12*)
```

If you want to link a variable directly to a particular address, you must declare the variable using the keyword **AT**.

For faster input of declarations, use the shortcut mode.

In function blocks, variables can also be specified with incomplete address specifications. An entry must be made in the variable configuration in order to use such variables in a local instance.

### **AT Declaration**

If you want to link a variable directly to a particular address, you must declare the variable using the keyword **AT**. The advantage of this procedure is that a more meaningful name can be given to the address, and if an input or output signal must be changed, it need only be changed at one position (in the declaration).

Remember that it is not possible to obtain write access to variables placed at an input. Another limitation is that AT declarations can only be made for local and global variables, not for input and output variables of blocks.

Examples:

```
switch_heating7 AT %QX0.0: BOOL;  
lightbarrier_impulse AT %IX7.2: BOOL;  
storage AT %MX2.2: BOOL;
```

### **'Insert' 'Declaration Keywords'**

This command opens a list of all keywords that can be used in the declaration part of a block. After a keyword has been selected and the selection has been confirmed, the words is inserted at the current cursor position.

The list can also be obtained by calling the input help and selecting the category **Declarations**.

### 'Insert' 'Types'

This command displays a selection of possible types for a variable declaration. The list can also be obtained by calling the input help.

The types are divided into the following categories:

- standard types                      BOOL, BYTE, etc.
- defined types                       structures, enumerator types, etc.
- standard function blocks       for instance declarations
- defined function blocks       for instance declarations

**CP1131** supports all standard types in the IEC1131-3 standard:

Examples of the use of the different types can be found in the appendix.

### Syntax Coloring

Visual support is provided in the text editors and in the declaration editor during implementation and variable declaration. Errors are avoided or quickly detected, as the text is displayed in color.

An unclosed comment which comments out instructions is detected immediately; there are no typing errors in keywords, etc.

The following colors are used:

- Blue                      Keywords
- Green                     Comments
- Pink                      Boolean values (TRUE/FALSE)
- Red                       Incorrect input (e.g. invalid time constant, keyword in lowercase, etc.)
- Black                     Variables, constants, assignment operators, etc.

### Shortcut Mode

The **CP1131** declaration editor offers the option of using shortcut mode. This is activated by ending a line with <Ctrl>+<Enter> key.

The following shortcuts are supported:

- All identifiers up to the last identifier in a line become variable identifiers in the declaration.
- The type of declaration is determined by the last identifier in the line. The following applies in this case:

B or BOOL	produces	BOOL
I or INT	produces	INT
R or REAL	produces	REAL
S or STRING	produces	STRING

- If no type could be defined by these rules, the type is BOOL and the last identifier is not used as a type (example 1).
- Depending on the type of declaration, each constant becomes an initialisation or string length (examples 2 and 3).
- An address (as in %MD12) is expanded by the AT ... attribute (example 4).
- A text after a semi-colon (;) becomes a comment (example 4).
- All other characters in the line are ignored (such as the call character in example 5).

Examples:

Shortcut	Declaration
A	A: BOOL;
A B I 2	A, B: INT := 2;
ST S 2; a string	ST: STRING(2); (* a string *)
X %MD12 R 5; real number	X AT %MD12: REAL := 5.0;(* real number *)
B !	B: BOOL;

### Declaring Automatically

If the option **Autodeclaration** has been selected, a dialog appears in all editors after input of a variable which has not yet been declared. This can be used to declare the variable.

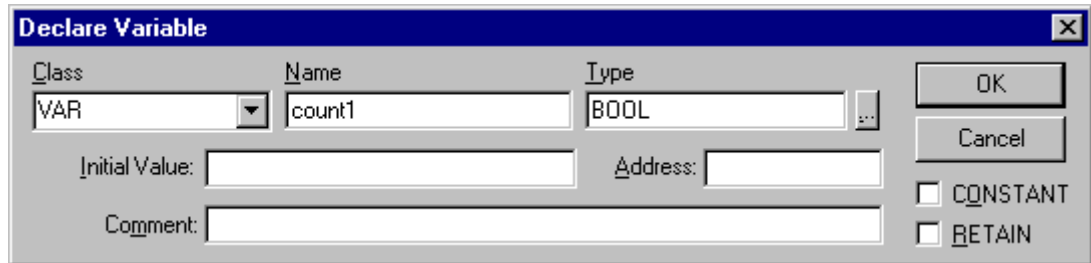


Figure 5\_2: Dialog for variable declaration

Using the combobox **Class**, define the variable as a local variable (**VAR**), an input variable (**VAR\_INPUT**), an output variable (**VAR\_OUTPUT**), an input/output variable (**VAR\_INOUT**) or a global variable (**VAR\_GLOBAL**).

Using the options **CONSTANT** and **RETAIN**, you can define whether the variable is a constant or a retentive variable.

The **Name** field has been preset with the variable name entered in the editor, and the **Type** field has been preset with BOOL. The ... button displays a dialog with input help for selection from all possible types.

In the **Initial value** field, you can assign a value to the variable. Otherwise, the default initial value is used.

Using the **Address** field, you can link a variable to the address (AT declaration). A **comment** can be entered if necessary.

When **OK** is pressed, the variable is entered in the corresponding declaration editor.

**Line Numbers in the Declaration Editor**

In offline mode, a single click on a special line number selects the entire text line.

In online mode, a single click on a particular line number opens or closes the variable in this line, if it is a structured variable.

**Declarations as Table**

If the option **Declarations as tables** is set in the **Editor** category of the Options dialog, the declaration editor is displayed in table form. You can select the various variable types as tabs in the same way as a card index, and edit the variables.

The following fields are displayed for each variable:

- Name: Enter the identifier of the variable.
- Address: Enter the address of the variable, if it has one (AT declaration).
- Type: Enter the type of variable (if instancing a function block, enter the function block).
- Initial: Enter any initialisation of the variable (in accordance with the assignment operator " := ").
- Comment: Enter a comment here.

You can easily switch between both display types of the declaration editor. In online mode, there is no difference in the display of the declaration editor.

	VAR	VAR_INPUT	VAR_OUTPUT	VAR_IN_OUT	CONSTANT
	Name	Address	Type	Initial	Comment
0001	I1 Ampelschaltung		Ampelschaltung		Erste Instanz de
0002	I2 Ampelschaltung		Ampelschaltung		Zweite Instanz i
0003	Zustand		INT	0	Wird fuer die In

Figure 5\_3: Declaration editor as table

**'Insert' 'New Declaration'**

This command allows you to enter a new variable in the declaration table of the declaration editor. If the cursor is currently positioned in a field of the table, the new variable is inserted in front of this line. Otherwise it is appended at the end of the table. You can also append a new declaration at the end of the table by pressing the right arrow key or the tab key in the last field of the table.

You obtain a variable with the presetting **'Name'** in the **Name** field and the presetting **'Bool'** in the **Type** field. You should change these values to the required values. The name and type are sufficient for a full variable declaration.

## Declaration Editors in Online Mode

The declaration editor becomes a monitor window in online mode. There is a variable in each line, followed by an equals sign (=) and the value of the variable. If the variable is not defined at this point, three question marks appear (???).

There is a plus sign in front of each multi-element variable. The variable is opened by pressing the <Enter> key or double-clicking on the variable. In the example below, the structure Lights1 is opened:

```

+---AMPEL1
  |---.STATUS = 3
  |---.GRUEN = FALSE
  |---.GELB = FALSE
  |---.ROT = TRUE
  |---.AUS = FALSE
  
```

An open variable lists all your components in sequence. A minus sign appears in front of the variable. Another double-click, or pressing the <Enter> key closes the variable and the plus sign is displayed again.

Pressing the <Enter> key or double-clicking on a single-element variable opens the dialog for writing a variable. The current value of the variable can be changed here. No dialog appears for Boolean variables, as they are toggled.

The new value turns red and remains unchanged. If the command '**Online**' '**Write values**' is executed, all variables are set to the selected values and displayed in black again.

If the command '**Online**' '**Force values**' is executed, all variables are set to the selected values until the command '**End forcing**' is executed.

## Comment

User comments must be enclosed within the special character sequences "(" and ")".

Comments are permitted at any position in all text editors. This includes all declarations, the languages IL and ST, and self-defined data types.

Comments on every network can be entered in FBD and LD. To do this, locate the network on which you would like to comment and activate '**Insert**' '**Comment**'.

In SFC, you can enter comments on the step in the dialog for editing step attributes. Nested comments are not permitted.

If you hold the mouse pointer over a variable for a short time in online mode, the type and any comment on the variable are displayed in a tooltip.

## The Text Editors

The text editors (the Instruction List editor and the Structured Text editor) in **CP1131** have all the typical functionalities of Windows text editors.

Implementation in the text editors is supported by syntax coloring.

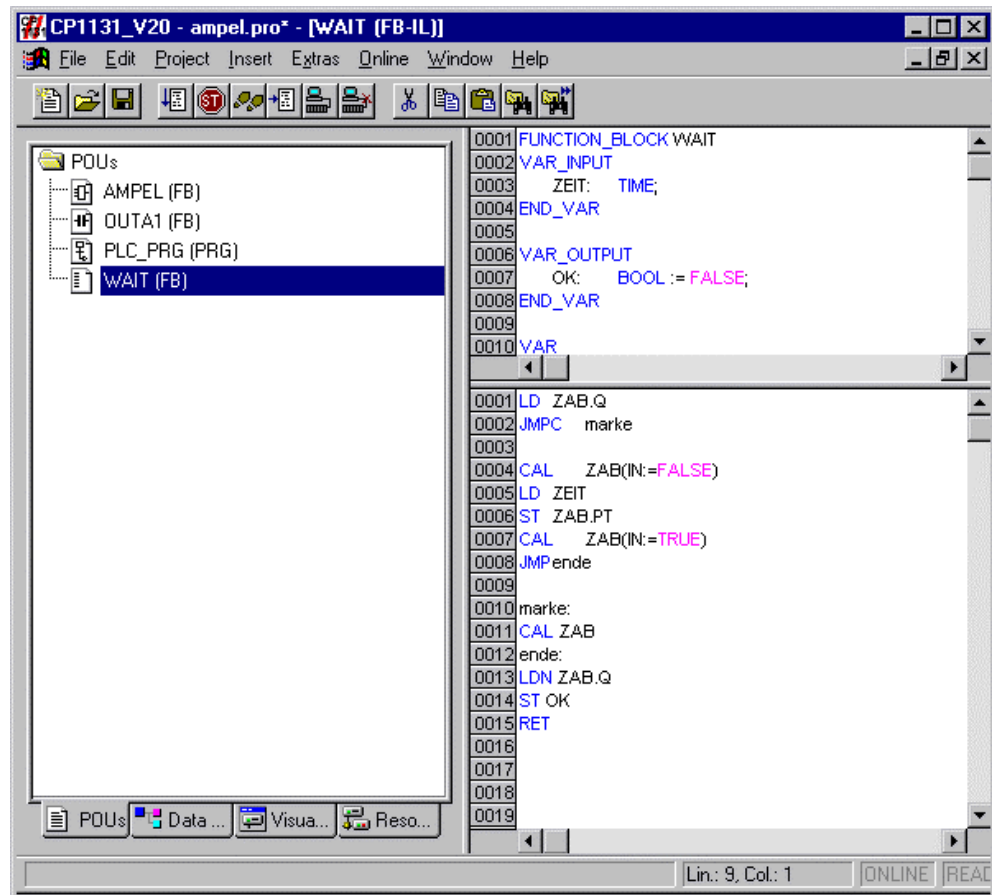


Figure 5\_4: Text editor for Instruction List and Structured Text

The most important commands are found in the context-sensitive menu (right-hand mouse button or <Ctrl>+<F10>). The following menu commands are used especially by the text editors:

### **'Insert' 'Operator'**

This command displays all operators available in the current language in a dialog.

If one of the operators is selected and the list is closed using **OK**, the selected operator is then inserted at the current cursor position.

### **'Insert' 'Operand'**

This command displays all variables in a dialog. You can select whether to display a list of global, local or system variables.

If one of the operands is selected and the dialog is closed using **OK**, the selected operand is inserted at the current cursor position.

### **'Insert' 'Function'**

This command displays all functions in a dialog. You can select whether to display a list of user-defined functions or standard functions. If one of the functions is selected and the dialog is closed using **OK**, the selected function is inserted at the current cursor position. If the option **With arguments** is selected in the dialog, the required input and output variables for the function are included.

### **'Insert' 'Function Block'**

This command displays all function blocks in a dialog. You can select whether to display a list of user-defined function blocks or standard function blocks.

If one of the function blocks is selected and the dialog is closed using **OK**, the selected function block is inserted at the current cursor position. If the option **With arguments** is selected in the dialog, the required input and output variables of the function block are included.

## **The Text Editors in Online Mode**

The online functions in the editors are breakpoint setting and single-step processing (stepping). Together with monitoring, they provide the user with the debugging functionality of a modern Windows high-level language debugger.

In online mode, the text editor window is split into two vertical sections. The left-hand side of the window contains the normal program text, while the right-hand side displays the variables whose values are changed in the relevant line.

The display is the same as in the declaration part. This means that if the controller is running, the current values of the relevant variables are displayed. Structured values (arrays, structures or instances of function blocks) are indicated by a plus sign in front of the identifier.

The variable is opened and closed by clicking on the plus sign or pressing the <Enter> key. If you hold the mouse pointer over a variable for a short time, the variable type and a comment on the variable are displayed in a tooltip.

### 'Extras' 'Monitoring Options'

This command allows you to configure your monitoring window. In the text editors, the window is divided in two during monitoring. The program is displayed on the left, while on the right, all variables in the corresponding program line are monitored.

You can set the **width** of the monitoring area in the text window and the **distance** between two monitoring variables in a line. The distance specification 1 corresponds to one line height in the selected font.

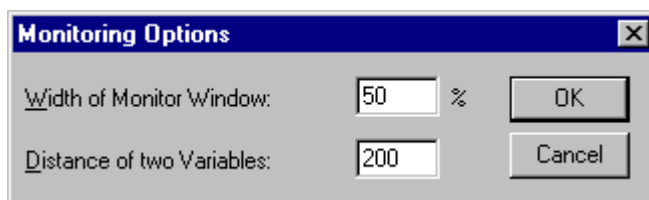


Figure 5\_5: Monitoring options dialog

### Breakpoint Positions

As several IL lines are combined in one C code line internally in **CP1131**, breakpoints cannot be set in each line. Breakpoint positions are all positions in the program at which variable values can change or at which the program flow branches (exception: function calls - here it may be necessary to set a breakpoint in the function). It is not practical to set a breakpoint at the intermediate positions, as none of the data will have changed since the previous breakpoint position.

The following breakpoint positions are therefore used in IL:

- the beginning of the block
- each LD, LDN (or if an LD is found directly after a label, on this)
- each JMP, JMPC, JMPCN
- each label
- each CAL, CALC, CALCN
- each RET, RETC, RETCN
- the end of the block

The following breakpoint positions are used in Structured Text:

- each assignment
- each RETURN and EXIT instruction
- lines in which conditions are evaluated (WHILE, IF, REPEAT)
- the end of the block

The line number field is displayed in dark grey at breakpoint positions.

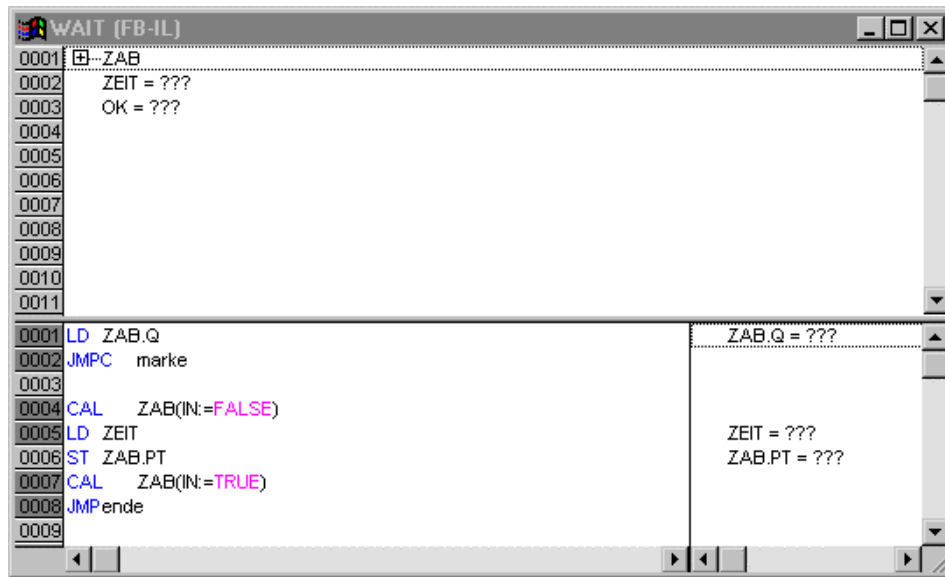


Figure 5\_6: IL editor with possible breakpoint positions (darker number fields)

### How is a Breakpoint Set?

To set a breakpoint, the user clicks on the line number field of the line in which a breakpoint is to be set using the mouse. If the selected field is a breakpoint position, the color of the line number field changes from dark grey to light blue and the breakpoint is activated in the controller.

### Deleting Breakpoints

To delete a breakpoint, you click on the line number field of the line containing the breakpoint to be deleted.

Breakpoints can also be set and deleted using the menu ('**Online**' '**Breakpoint on/off**'), the function key <F9> or the icon in the tool bar.

### What Happens at a Breakpoint?

When the controller reaches a breakpoint, the section containing the corresponding line is displayed on the screen. The line number field of the line at which the controller is stopped is displayed in red. The controller contains the processing of the user program.

If the program is stopped at a breakpoint, processing can be continued using '**Online**' '**Start**'.

Alternatively, '**Online**' '**Single step over**' or '**Single step to**' can be used to proceed as far as the next breakpoint position. If the instruction where the controller is currently stopped is a CAL command, or if there is a function call in the lines up to the next breakpoint position, this is skipped using '**Single step over**', while '**Single step to**' branches to the block called.

### Line Numbers of the Text Editor

The line numbers of the text editor specify the number of each text line in an implementation of a block.

In offline mode, a single click on a specific line number selects the entire text line.

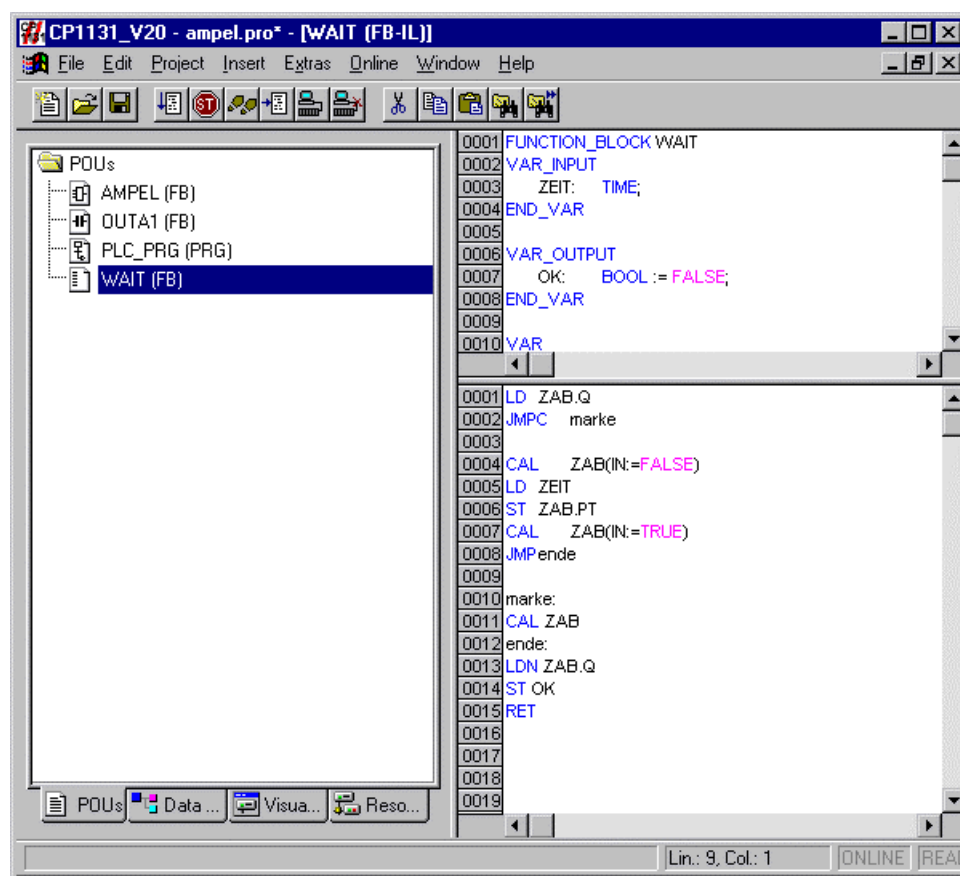
In online mode, the background color of the line number indicates the breakpoint status of each line:

- dark grey: this line is a possible position for a breakpoint
- light blue: a breakpoint has been set in this line
- red: program processing is at this point.

In online mode, a single mouse click switches the breakpoint status of this line.

### The Instruction List Editor

This is what a block written in IL looks like under the corresponding **CP1131** editor:



Figure

5\_7: IL editor

All editors for blocks consist of a declaration part and a body. These are separated by a screen split bar.

The Instruction List editor is a text editor with the usual functionalities of Windows text editors. The most important commands are found in the context-sensitive menu (right-hand mouse button or <Ctrl>+<F10>).

For information on the IL editor in online mode, see the section entitled 'The text editors in online mode'. For information on the language, see the section entitled 'Instruction lists (IL)'.

### Program Check

The command 'Online' 'Display Flow Control' in the IL editor inserts another field on the left-hand side of each line to display the contents of the accumulator.

### The Structured Text Editor

This is what a block written in ST looks like under the corresponding CP1131 editor:

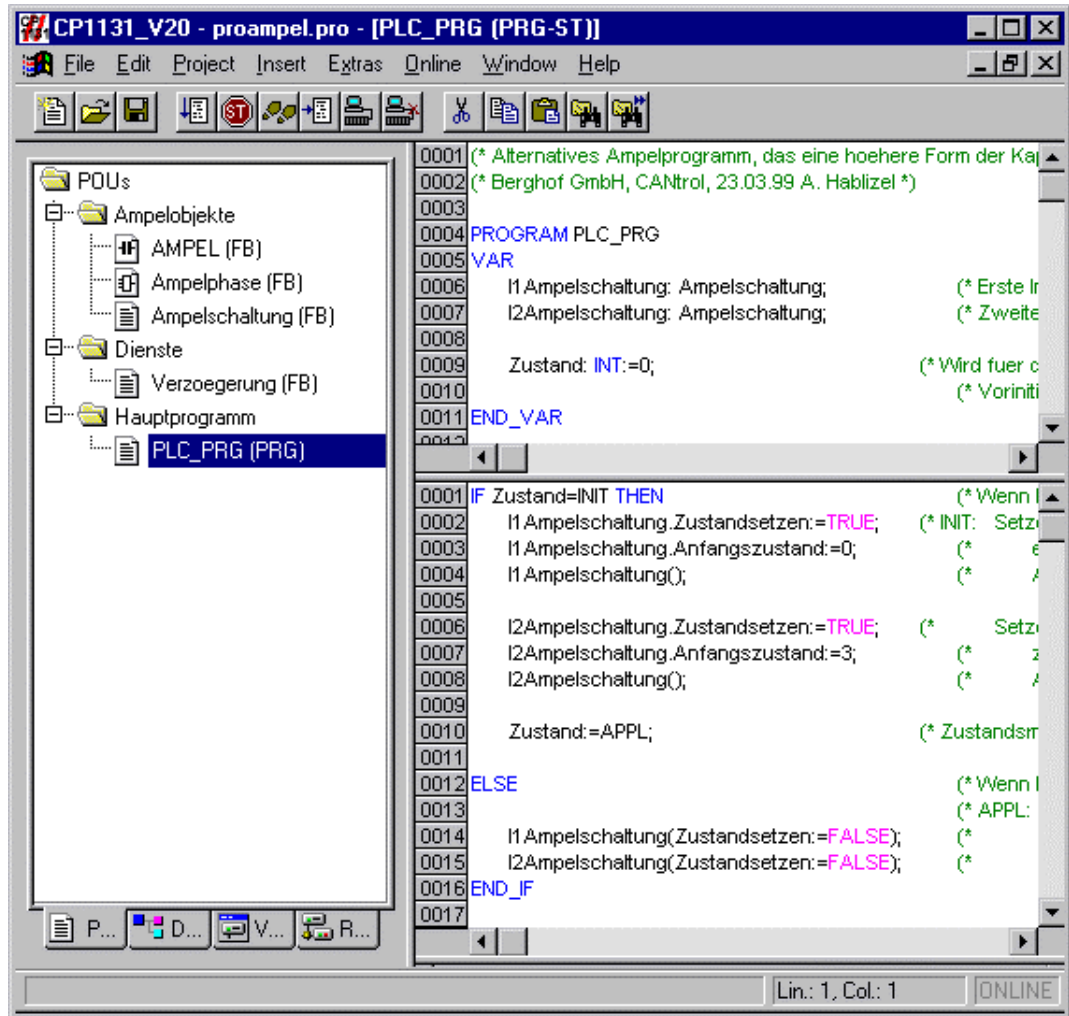


Figure 5\_8: Editor for Structured Text

All editors for blocks consist of a declaration part and a body. These are separated by a screen split bar.

The Structured Text editor is a text editor with the typical functionalities of Windows text editors. The most important commands are found in the context-sensitive menu (right-hand mouse button or <Ctrl>+<F10>).

For information on the ST editor in online mode, see the section entitled 'The text editors in online mode'. For information on the language, see the section entitled 'Structured Text (ST)'.

---

## The Graphical Editors

The editors in the two graphical languages, LD and FBD, have many similarities. Their similarities are summarised in the following paragraphs. The Sequential Function Chart editor is different, and is described later in the section entitled '**The Sequential Function Chart editor**'.

### Jump Labels

There is a jump label in each network which can sometimes be empty. This label is edited by clicking on the first line of the network right beside the network number. A label can now be entered, followed by a colon.

### Network Comments

A multi-line comment can be assigned to each network. In '**Etras**' '**Options**', the maximum number of lines to be made available for a network comment can be entered in the **Maximum comment size** field (the preset value is 4), as well as the number of lines to be generally released for a comment (**Minimum comment size**). If, for example, 2 is set here, there are two empty comment lines at the beginning of each network after the label line. The preset value is 0, and the advantage of this is that it allows more networks to fit into the screen area.

If the minimum comment size is greater than 0, you can create a comment by simply clicking in the comment line and entering the comment. Otherwise, it is necessary to first select the network for which the comment is to be entered and insert a comment line using '**Insert**' '**Comment**'. Unlike program text, comments are displayed in grey.

### 'Insert' 'Network (after)' or 'Insert' 'Network (before)'

**Shortcut:** <Ctrl>+<T> (network after)

To insert a new network in the FBD or LD editor, either the command '**Insert**' '**Network (after)**' or '**Insert**' '**Network (before)**' is selected, depending on whether the new network is to be inserted before or after the current network. The current network is changed by clicking on the network number with the mouse. This is indicated by a dotted rectangle under the number. The entire range of networks between the current network and the network clicked is selected by pressing the <Shift> key and clicking with the mouse.

### The Network Editors in Online Mode

In the FBD and LD editors, breakpoints can only be set at networks. The network number field of a network at which a breakpoint has been set is displayed in blue. Processing stops before the network at which the breakpoint is found. The network number field is now displayed in red. In single-step processing (stepping), you jump from network to network.

All values are monitored at the inputs and outputs of network blocks.

Program checking is started using the menu command '**Online**' '**Display Flow Control**'. This allows you to view the current values transported in the networks via the connection lines. If the connection lines are not transporting Boolean values, the value is displayed in a separately-inserted field. If the lines are transporting Boolean values, they are coloured blue when transporting TRUE. This allows tracing of the flow of information during the controller run.

If you hold the mouse pointer over a variable for a short time, the variable type and the comment on the variable are displayed in a tooltip.

## The Function Block Diagram Editor

This is what a block written in FBD looks like under the corresponding **CP1131** editor:

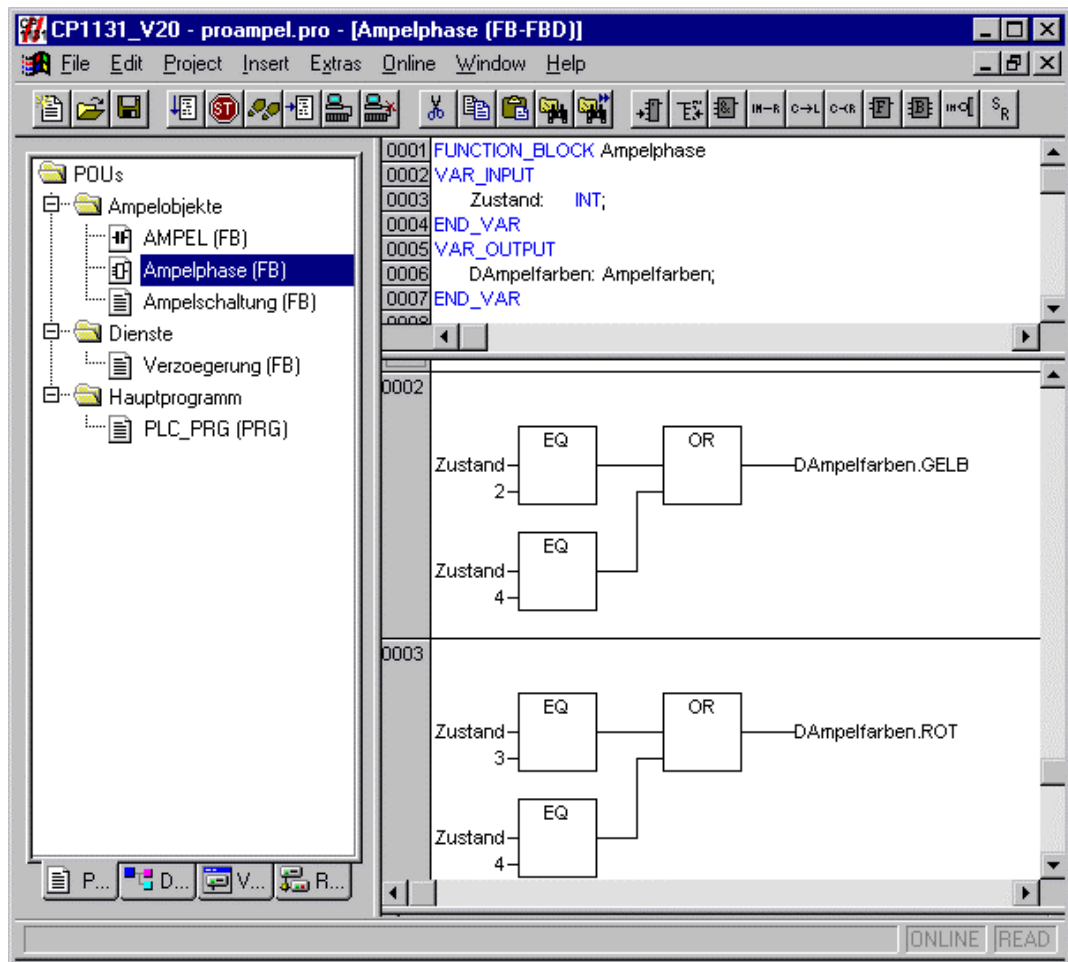


Figure 5\_9: Editor for Function Block Diagram

The Function Block Diagram editor is a graphical editor. It uses a list of networks, with each network containing a structure representing a logical or arithmetic expression, a call of a function block, a jump or a return instruction.

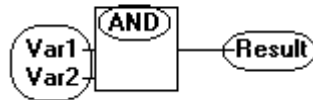
The most important commands are found in the context-sensitive menu (right-hand mouse button or <Ctrl>+<F10>).

**Cursor Positions in FBD**

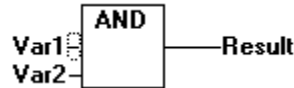
Every text is a possible cursor position. The selected text is displayed on a blue background and can now be edited.

Otherwise, the current cursor position is indicated by a dotted rectangle. The following is a list of all possible cursor positions, together with an example:

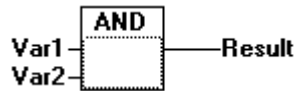
- 1) every text field (possible cursor positions are framed in black):



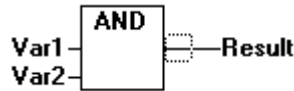
- 2) every input:



- 3) every operator, function or function block:



- 4) outputs, if followed by an assignment or a jump:



- 5) The line cross-over above an assignment, a jump or a return instruction:



- 6) After the right-most object in any network ("last cursor position", this is also the cursor position if a network has been selected):



- 7) the line cross-over immediately before an assignment:



## How to Set the Cursor

The cursor can be set at a certain position by clicking with the mouse or by using the keyboard.

With the help of the arrow keys, you can jump to the next cursor position in the selected direction. All cursor positions, including the text fields, can be reached in this way. If the last cursor position is selected, the <up> and <down> arrow keys can be used to select the last cursor position of the previous or the subsequent network. An empty network contains only three question marks “???”. The last cursor position is selected by clicking after these.

## ‘Insert’ ‘Assign’

Icon:  **Shortcut: <Ctrl>+<A>**

This command inserts an assignment.

Depending on the position selected, the assignment is inserted immediately before the selected input (cursor position 2), immediately after the selected output (cursor position 4), immediately before the selected line cross-over (cursor position 5) or at the end of the network (cursor position 6).

The text entered “???” for an inserted assignment can then be selected and replaced by the variable to which the assignment is to be made. This can also be done using the input help. To add another assignment to an existing assignment, use the command ‘Insert’ ‘Output’.

## ‘Insert’ ‘Jump’

Icon:  **Shortcut: <Ctrl>+<L>**

This command inserts a jump.

Depending on the position selected, the jump is inserted immediately before the selected input (cursor position 2), immediately after the selected output (cursor position 4), immediately before the selected line cross-over (cursor position 5) or at the end of the network (cursor position 6).

The text entered “???” for an inserted jump can then be selected and replaced by the jump label to which the jump is to be made.

## ‘Insert’ ‘Return’

Icon:  **Shortcut: <Ctrl>+<R>**

This command inserts a RETURN instruction.

Depending on the position selected, the RETURN instruction is inserted immediately before the selected input (cursor position 2), immediately after the selected output (cursor position 4), immediately before the selected line cross-over (cursor position 5) or at the end of the network (cursor position 6).

### 'Insert' 'Operator'

Icon:  **Shortcut: <Ctrl>+<O>**

This command inserts an operator. It is inserted in accordance with the selection position.

If an input is selected (cursor position 2), the operator is inserted before this input. The first input of this operator is linked to the branch to the left of the selected input. The output of the new operator is linked to the selected input.

If an output is selected (cursor position 4), the operator is inserted after this output. The first input of the operator is linked to the selected output. The output of the new operator is linked to the branch with which the selected output was linked.

If an operator, a function or a function block is selected (cursor position 3), the old element is replaced by the new operator. Insofar as is possible, the branches are linked as they were before the replacement. If the old element had more inputs than the new one, the unlinkable branches are deleted. The same applies to the outputs.

If a jump or a return is selected, the operator is inserted before this jump or return. The first input of the operator is linked to the branch to the left of the selected element. The output of the operator is linked to the branch to the right of the selected element.


If the last cursor position of a network is selected (cursor position 6), the operator is inserted after the last element. The first input of the operator is linked to the branches to the left of the selected position.

The inserted operator is always an AND. This can be converted to any other operator by selecting and overwriting the text. The required operator can also be selected from the list of supported operators using the input help. If the new operator has another minimum number of inputs, these are added. If the new operator has a smaller maximum number of inputs, the last inputs are deleted together with the branch before them.

All inputs of the operator that could not be linked receive the text "???". This text must be clicked on and changed to the required constant or variable.

### 'Insert' 'Function' or 'Insert' 'Function block'

Icon:  **Shortcut: <Ctrl>+<F>** (function)

Icon:  **Shortcut: <Ctrl>+<B>** (function block)

This command inserts a function or a function block. It is inserted in accordance with the selected position. Firstly, the input help dialog opens with all functions or function blocks available for selection.

Insertion, as well as the assignment of inputs and outputs, is the same as for the command **'Insert' 'Operator'**. If there is a branch to the right of an inserted function block, this is allocated to the first output. Otherwise, the outputs remain unassigned.

**'Insert' 'Input'**

Icon:  **Shortcut: <Ctrl>+<U>**

This command inserts an operator input. The number of inputs is variable for many operators (e.g. ADD can have two or more inputs).

To extend such an operator by one input, the input before which another input is to be inserted (cursor position 1), or the operator itself (cursor position 3), if a bottom-most input is to be appended, is selected.

The inserted input has the text "???". This text must be clicked on and changed to the required constant or variable. This can also be done using the input help.

**'Insert' 'Output'**

Icon: 

This command adds an additional assignment to an existing assignment. This functionality is used to create "assignment combs", i.e. the assignment of the value currently on the line to several variables.

If the line cross-over above an assignment (cursor position 5) or the immediately preceding output (cursor position 4) is selected, another assignment is appended after the existing assignments.

If the line cross-over directly before an assignment is selected (cursor position 4), another one is inserted before this assignment.

The inserted output contains the text "???". This text must be clicked on and changed to the required variable. This can also be done using the input help.

**'Extras' 'Negate'**

Icon:  **Shortcut: <Ctrl>+<N>**

This command can be used to negate inputs, outputs, jumps or RETURN instructions. The icon for negation is a small circle on a connection.

If an input is selected (cursor position 2), this input is negated.

If an output is selected (cursor position 4), this output is negated.

If a jump or a return is selected, the input of this jump or return is negated.

A negation can be deleted by repeating the negation.

**'Extras' 'Set/Reset'**

Icon: 

This command can be used to define outputs as set or reset outputs. A grid with a set output is displayed as [S] and a grid with a reset output is displayed as [R].

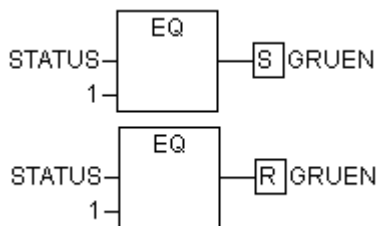


Figure 5\_10: Set/reset outputs in FBD

A *set output* is set to TRUE if the corresponding grid supplies TRUE. The output now contains this value, even if the grid jumps back to FALSE.

A *reset output* is set to FALSE if the corresponding grid supplies TRUE. The output retains its value, even if the grid jumps back to FALSE.

Repeated execution of the command switches the output between set, reset and normal output.

**'Extras' 'Zoom'**

**Shortcut: <Alt>+<Enter> key**

This command loads a selected block to its editor (cursor position 3).

If the block is from a library, the Library Manager is called and the corresponding block is displayed.

**Cutting, Copying, Inserting and Deleting in FBD**

The commands **'Cut'**, **'Copy'**, **'Paste'** and **'Delete'** are found under the menu item **'Edit'**.

If a line cross-over is selected (cursor position 5), the underlying assignments, jumps or RETURN instructions are cut, deleted or copied.

If an operator, a function or a function block is selected (cursor position 3), the selected object itself as well as all branches at the inputs are cut, deleted or copied, with the exception of the first branch.

Otherwise, the entire branch before the cursor position is cut, deleted or copied.

After copying or cutting, the deleted or copied part is stored in the clipboard and can be inserted any number of times.

To do this, the insertion point must first be selected. Valid insertion points are inputs and outputs.

If an operator, a function or a function block has been loaded to the clipboard, (remember that in this case, all branches apart from the first one are also in the clipboard), the first input is connected with the branch before the insertion point.

Otherwise, the entire branch before the insertion point is replaced by the contents of the clipboard.

In every case, the last element inserted is linked to the branch to the right of the insertion point.



**Note:** The following problem can be resolved by cutting and pasting: a new operator is inserted into the middle of a network. The branch to the right of the operator is now linked to the first input, but must be linked to the second input. You select the first input and execute **Edit** **Cut**. You then select the second input and execute **Edit** **Paste**. The branch now hangs at the second input.

### **Function Block Diagram in Online Mode**

In Function Block Diagram, breakpoints can only be set at networks. If a breakpoint has been set at a network, the network number field is displayed in blue. Processing stops before the network in which the breakpoint is found. The network number field is now red. You are jumped from network to network by stepping (single-step).

The current value is displayed for each variable present as an input on a network element (function, program, function block instance or operator).

A double-click on a variable opens the dialog for writing a variable. Here it is possible to change the current value of the variable. No dialog is displayed for Boolean variables, as they are toggled.

The new value is red and remains unchanged. If the command **Online** **Write values** is executed, all variables are set to the selected values and the display reverts to black.

The program check is started using the menu command **Online** **Program check**. This allows you to view the current values to be transported in the networks over the connection lines. If the connection lines do not transport any Boolean values, the value is displayed in a separately-inserted field. If the lines transport Boolean values, they are colored blue when they are transporting TRUE. In this way it is possible to trace the flow of information during the controller run.

If you hold the mouse pointer over a variable for a short time, the variable type and a comment on the variable are displayed in a tooltip.

### The Ladder Diagram Editor

This is what a block written in LD looks like under the **CP1131** editor:

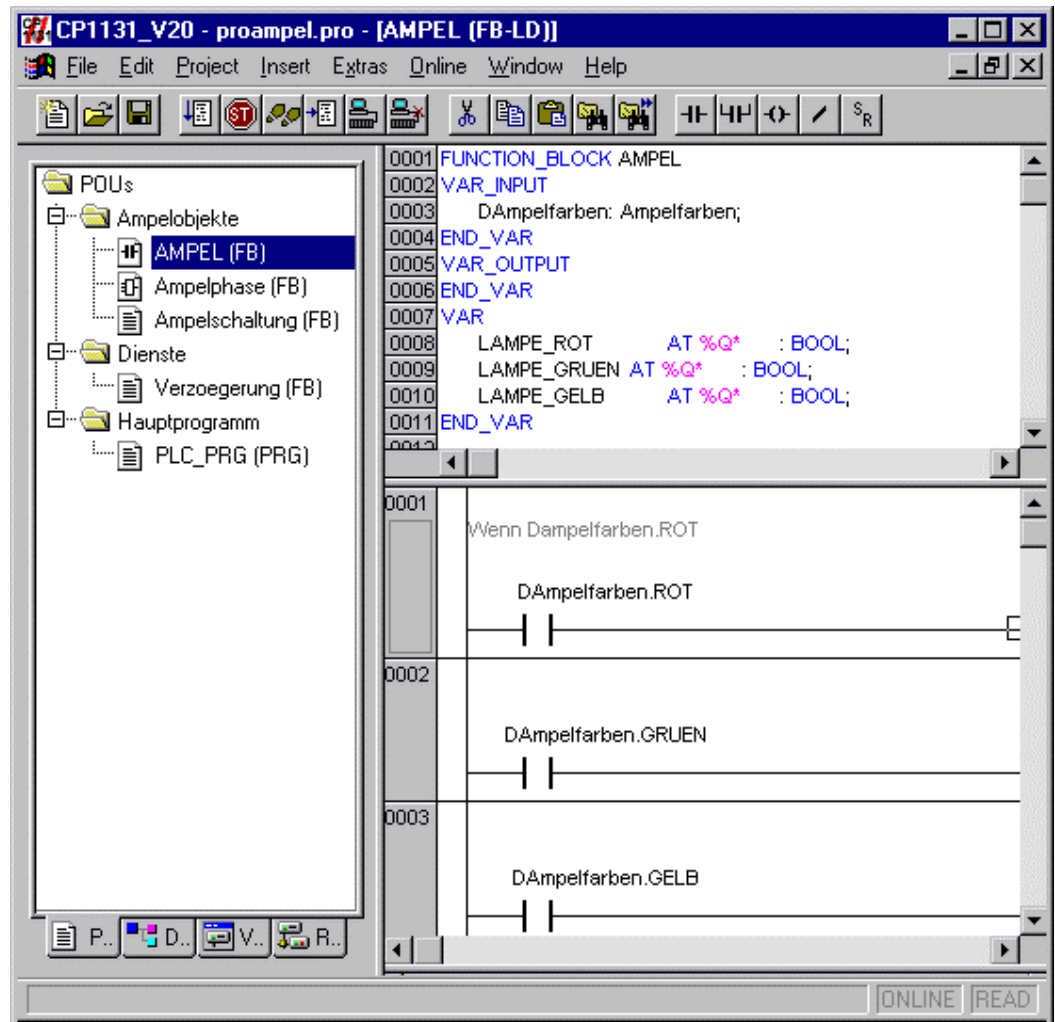


Figure 5\_11: Block in Ladder Diagram

All editors for blocks consist of a declaration part and a body. These are separated by a screen split bar.

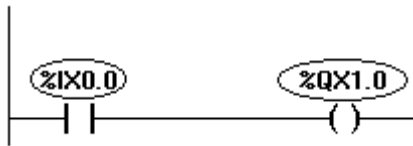
The LD editor is a graphical editor. The most important commands are found in the context-sensitive menu (right-hand mouse button or <Ctrl>+<F10>).

For information on the elements, see the section entitled 'Ladder Diagram (LD)'.

**Cursor Positions in the LD Editor**

The following positions can be cursor positions, with function block and program calls being handled in the same way as contacts. Blocks with EN inputs and other blocks linked to them are handled in the same way as in FBD. Information on editing these network parts can be found in the section entitled 'The Function Block Diagram editor' explaining the FBD editor.

- 1. every text field (possible cursor positions are framed in black)



- 2. every contact or function block



- 3. every coil



- 4. the connection line between the contacts and the coils



The following menu commands are used especially by Ladder Diagram:

**'Insert' 'Contact'**

Icon:  Shortcut: <Ctrl>+<O>

This command inserts a contact before the selected position in the network in the LD editor.

If the selected position is a coil (cursor position 3), or the connection line between the contacts and the coils (cursor position 4), the new contact is serial connected to the existing contact circuit.

The contact is preset with the text "???". You can click on this text and change it to the required variable or constant. This can also be done using the input help.

### **'Insert' 'Parallel Contact'**

Icon:  Shortcut: <Ctrl>+<R>

This command inserts a contact parallel to the selected position in the network in the LD editor.

If the selected position is a coil (cursor position 3), or the connection between the contacts and the coils (cursor position 4), the new contact is connected parallel to the overall existing contact circuit.

The contact is preset with the text "???". You can click on this text and change it to the required variable or constant. This can also be done using the input help.

### **'Insert' 'Function Block'**

Shortcut: <Ctrl>+<B>

This command opens a dialog for selecting a function block or program. You can choose between self-defined or standard blocks.

The selected block is inserted in accordance with the same rules as for a contact, with the first input of the block being placed on the input connection, and the first output being placed on the output connection. For this reason, these variables must be of the type BOOL. All other inputs and outputs of the block contain the text "???". These presettings can be changed to other constants, variables and addresses. This can also be done using the input help.

### **'Insert' 'Coil'**

Icon:  Shortcut: <Ctrl>+<L>

This command inserts a coil parallel to the existing coils in the LD editor.

If the selected position is the connection between the contacts and the coils (cursor position 4), the new coil is inserted as the last one. If the selected position is a coil (cursor position 3), the new coil is inserted directly over it.

The coil is preset with the text "???". You can click on this text and change it to the required variable. This can also be done using the input help.

### **Blocks with EN Inputs**

If you want to use your LD network to control calls of other blocks, you must insert a block with an EN input. This type of block is connected parallel to the coils. You can further develop the network as in Function Block Diagram on the basis of this block. The commands for inserting an EN block can be found under the menu item **'Insert' 'Insert at blocks'**.

An operator, a function block or a function with an EN input behaves in the same way as the corresponding block in Function Block Diagram, except that its execution is controlled by the EN input. This input is connected at the connection line between coils and contacts. If this connection transports the information "Open", the block is evaluated.

Once a block with an EN input has been created, it can be used to create a network as in Function Block Diagram. This means that data from the usual operators, functions and function blocks can flow into an EN block, and an EN block can transport data to the usual blocks of this type.

Therefore, if you want to program a network in the LD editor in the same way as in FBD, you need only insert an EN operator into a new network first, and then you can continue to develop your network from this block as in the FBD editor. A network created in this way behaves the same as the corresponding network in FBD.

### **'Insert' 'Operator with EN'**

This command inserts an operator with an EN input into an LD network.

The selected position must be the connection between the contacts and the coils (cursor position 4) or a coil (cursor position 3). The new operator is inserted parallel to the coils and below them, and initially has the identifier AND. You can change this to whatever identifier you wish. This can also be done using the input help.

### **'Insert' 'Function Block with EN'**

This command inserts a function block with an EN input into an LD network.

The selected position must be the connection between the contacts and the coils (cursor position 4) or a coil (cursor position 3). The new function block is inserted parallel to the coils and below them. In the input help dialog that is then displayed, you can select whether you want to enter a self-defined or a standard function block.

### **'Insert' 'Function with EN'**

This command inserts a function with an EN input into an LD network.

The selected position must be the connection between the contacts and the coils (cursor position 4) or a coil (cursor position 3). The new function is inserted parallel to the coils and below them. In the input help dialog that is then displayed, you can select whether you want to enter a self-defined or a standard function.

### **'Insert' 'Insert at Blocks'**

This command allows you to insert other elements at an already-inserted block (including a block with an EN input). The commands under this menu item can be executed at the same cursor positions as the corresponding commands in Function Block Diagram.

**Input** adds a new input at the block.

**Output** adds a new output at the block.

**Operator** adds a new operator at the block, whose output is created at the selected input.

**Assign** adds an assignment at the selected input or output.

**Function** adds a function at the selected input.

**Function block** adds a function block at the selected input.

### **'Insert' 'Jump'**

This command inserts a jump in parallel at the end of the existing coils in the LD editor. If the incoming line supplies the value "Open", the jump is executed at the label indicated. The selected position must be the connection between the contacts and the coils (cursor position 4) or a coil (cursor position 3). The jump is preset with the text "???". You can click on this text and change it to the required jump label.

### **'Insert' 'Return'**

This command inserts a RETURN instruction in parallel at the end of the existing coils in the LD editor. Processing of the block in this network is terminated when the incoming line supplies the value "Open".

The selected position must be the connection between the contacts and the coils (cursor position 4) or a coil (cursor position 3).

### **'Extras' 'Paste after'**

This command inserts the contents of the clipboard as a serial contact after the selection point in the LD editor. This command is only possible if the contents of the clipboard and the selected position are networks of contacts.

### **'Extras' 'Paste below'**

**Shortcut: <Ctrl>+<U>**

This command inserts the contents of the clipboard as a parallel contact under the selection point in the LD editor. This command is only possible if the contents of the clipboard and the selected position are networks of contacts.

### **'Extras' 'Paste above'**

This command inserts the contents of the clipboard as a parallel contact over the selection point in the LD editor. This command is only possible if the contents of the clipboard and the selected position are networks of contacts.

### **'Extras' 'Negate'**

Icon:  **Shortcut: <Ctrl>+<N>**

This command negates a contact, a coil, a jump or RETURN instruction or an input or output of EN blocks at the current cursor position (cursor positions 2 and 3).

An oblique (*/*) or (*|/*) appears between the round brackets of the coil or between the straight lines of the contact. In jumps, returns and inputs or outputs of EN blocks, a small circle appears on the connection as in the FBD editor.

The coil now writes the negated value of the input connection to the corresponding Boolean variable. A negated contact switches the status of the input to the output if the corresponding Boolean variable supplies the value FALSE.

If a jump or a return is selected, the input of this jump or return is negated. A negation can be deleted by repeating the negation.

### **'Extras' 'Set/Reset'**

If you execute this command on a coil, you receive a set coil. This type of coil never overwrites the value TRUE in the corresponding Boolean variables. This means that if the value of this variable is set to TRUE once, it remains at TRUE permanently. A set coil is indicated by an "S" in the coil icon.

If you execute this command a second time, you obtain a reset coil. This type of coil never overwrites the value FALSE in the corresponding Boolean variables. This means that if the value of this variable is set to FALSE once, it remains at FALSE permanently. A reset coil is indicated by an "R" in the coil icon.

If you execute this command several times, this coil switches from set to reset to normal coil.

### **Ladder Diagram in Online Mode**

In online mode, all contacts and coils in Ladder Diagram with the status "Open" are colored blue, and all lines over which "Open" is transported are also colored blue. The values of the corresponding variables are displayed at the inputs and outputs of function blocks.

Breakpoints can only be set on networks; with stepping, you are jumped from network to network.

If you hold the mouse pointer over a variable for a short time, the variable type and a comment on the variable are displayed in a tooltip.

## The Sequential Function Chart Editor

This is what a block written in SFC looks like under the **CP1131** editor:

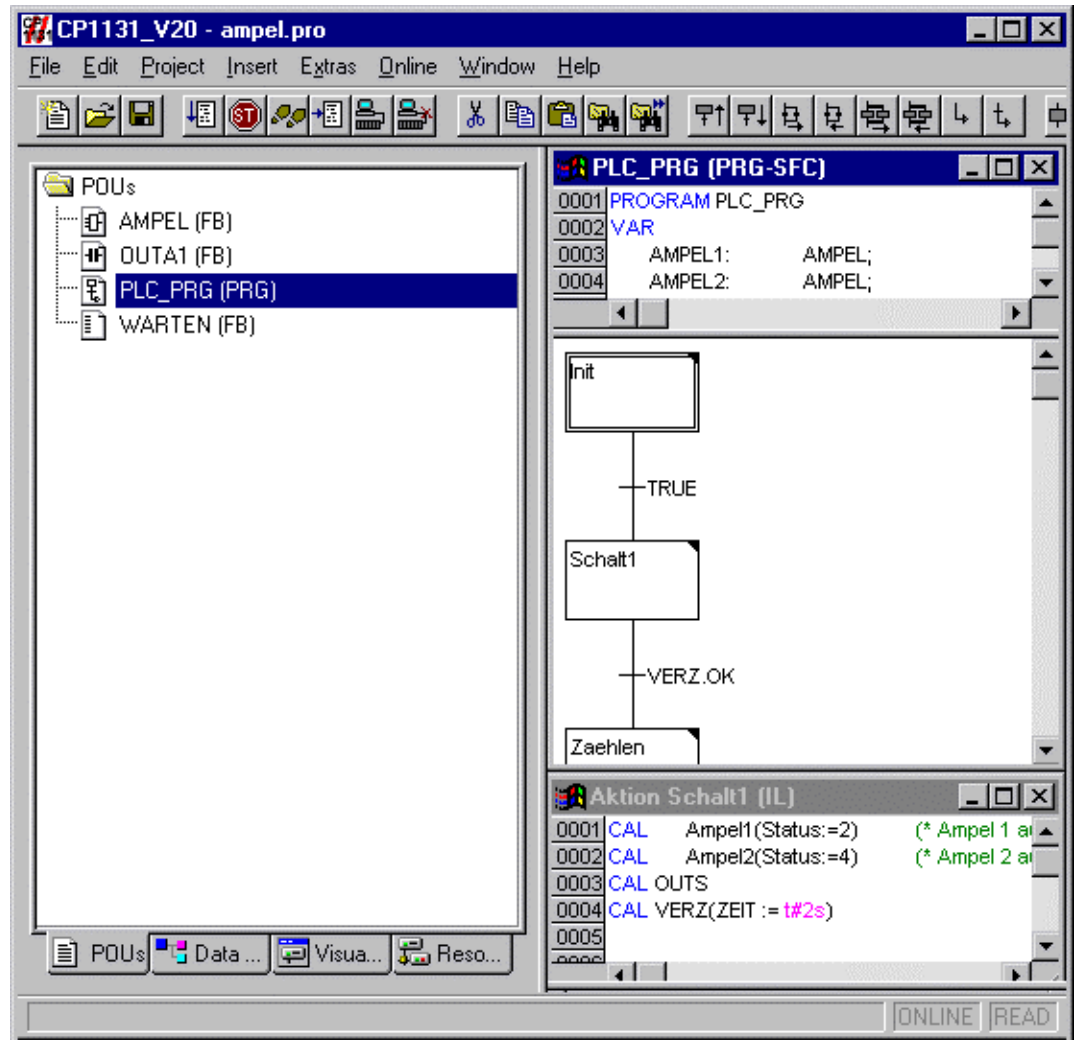


Figure 5\_12: Sequential Function Chart editor with an opened action

All editors for blocks consist of a declaration part and a body. These are separated by a screen split bar. The SFC editor is a graphical editor. The most important commands are found in the context-sensitive menu (right-hand mouse button or <Ctrl>+<F10>).

For information on Sequential Function Chart, see the section entitled '**Sequential Function Chart (SFC)**'. The following menu items allow the Sequential Function Chart editor to cope with its particular features:

## **Select Blocks in SFC**

A selected block is a group of SFC elements surrounded by a dotted rectangle. (In the example above, the step Switch1 is selected.)

An element (step, transition or jump) can be selected by placing the mouse pointer on the element and pressing the left-hand mouse button or by using the arrow keys. To select a group of several elements, press the <Shift> key in addition to an already-selected block, and select the element in the lower left or right-hand corner of the group. The resulting selection is the smallest related group of elements containing both these elements.

Remember that all commands can only be executed if they do not contravene the language conventions.

### **'Insert' 'Step Transition (before)'**

Icon:  Shortcut: <Ctrl>+<T>

This command inserts a step followed by a transition in front of the selected block in the SFC editor.

### **'Insert' 'Step Transition (after)'**

Icon:  Shortcut: <Ctrl>+<E>

This command inserts a step followed by a transition after the first transition in the selected block in the SFC editor.

### **'Insert' 'Alternative Branch (right)'**

Icon:  Shortcut: <Ctrl>+<A>

This command inserts an alternative branch as a right branch of the selected block in the SFC editor. The selected block must begin and end with a transition. The new branch then consists of one transition.

### **'Insert' 'Alternative Branch (left)'**

Icon: 

This command inserts an alternative branch as a left branch of the selected block in the SFC editor. The selected block must begin and end with a transition. The new branch then consists of one transition.

### **'Insert' 'Parallel Branch (right)'**

Icon:  Shortcut: <Ctrl>+<L>

This command inserts a parallel branch as a right branch of the selected block in the SFC editor. The selected block must begin and end with a step. The new branch then consists of one step.

**'Insert' 'Parallel Branch (left)'**Icon: 

This command inserts a parallel branch as a left branch of the selected block in the SFC editor. The selected block must begin and end with a step. The new branch then consists of one step.

**'Insert' 'Jump'**Icon:  Shortcut: <Ctrl>+<U>

This command inserts a jump at the end of the branch to which the selected block belongs in the SFC editor. The branch must be an alternative branch.

**'Insert' 'Transition Jump'**Icon: 

This command inserts a transition followed by a jump at the end of the selected branch in the SFC editor. The branch must be a parallel branch.

**'Insert' 'Add Entry Action'**

This command allows you to add an input action to a step. An input action is only executed once, immediately after the step becomes active. The input action can be implemented in any language.

A step with an input action is indicated by an 'E' in the lower left-hand corner.

An input action cannot be defined as an IEC step.

**'Insert' 'Add Exit Action'**

This command allows you to add an output action to a step. An output action is only executed once, before the step is deactivated. The output action can be implemented in any language.

A step with an output action is indicated by an 'X' in the lower right-hand corner.

An output action cannot be defined as an IEC step.

**'Extras' 'Insert Parallel Branch (right)'**

This command inserts the contents of the clipboard as a right-hand parallel branch of the selected block. The selected block must begin and end with a step. The contents of the clipboard must also be an SFC block which begins and ends with a step.

**'Extras' 'Insert After'**

This command inserts the SFC block into the clipboard after the first step or first transition of the selected block (normal copying inserts it in front of the selected block). This is only executed if the resulting SFC structure is correct according to the language standards.

**‘Extras’ ‘Zoom Action/Transition’**

**Shortcut: <Alt>+<Enter> key**

The action of the first step of the selected block or the transition body of the first transition of the selected block is loaded to the editor in the language in which it was written. If the action or the transition body is empty, the language in which it is to be written must be selected.

**‘Extras’ ‘Clear Action/Transition’**

This command allows you to delete the actions of the first step of the selected block or the transition body of the first transition of the selected block.

If only the action, the input action or the output action in a step is implemented, this is deleted using the command. Otherwise, a dialog appears in which you can select which action or actions are to be deleted.

If the cursor is located in an action of an IEC step, only this association is deleted. If an IEC step with an associated action is selected, this association is deleted. In an IEC step with several actions, a selection dialog is displayed.

**‘Extras’ ‘Step Attributes’**

This command opens a dialog in which you can edit the attributes of the selected step.

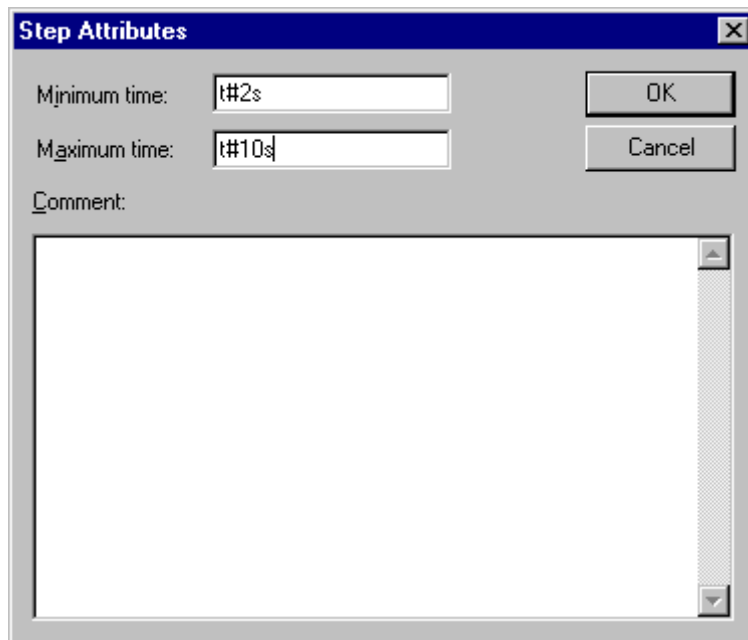
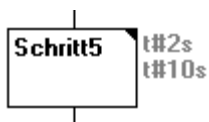


Figure 5\_13: Dialog for editing step attributes

You can make three different entries in the step attributes dialog. Under **Minimum time**, enter the minimum time duration for processing of this step. Under **Maximum time**, enter the maximum time duration for processing of the step. Remember that the entries must be of the type **TIME**. The notation described in Appendix C is therefore applicable.

Under **Comment**, you can enter a comment on the step. Using **'Tools' 'Options'**, you can set whether the comments or the time settings for your steps is to be displayed in the SFC editor. Either the comment or the time settings are then displayed on the right beside the step.

If the maximum time is exceeded, SFC flags are set. These can be queried by the user.



The example shows a step whose execution is to take at least two seconds and no more than ten seconds. In online mode the display also shows how long the step has been active for, in addition to these two times.

## **SFC Flags**

If a step is active for longer than the time specified in its attributes in SFC, certain specific flags are set. In addition, you can also define other variables to control the program in Sequential Function Chart. In order to use the flags, you must declare them somewhere, either globally or locally, as output or input variables.

**SFCEnableLimit:** This special variable is of the type BOOL. If it is TRUE, time overruns in the steps are recorded in SFCError. Otherwise, time overruns are ignored.

**SFCInit:** This variable is also of the type BOOL. If this variable is TRUE, Sequential Function Chart is reset to the Init step. The Init step remains active for as long as the variable remains TRUE. Processing of the block does not continue normally until SFCInit is set to FALSE again.

**SFCQuitError:** A variable of the type BOOL. For as long as this variable is TRUE, processing of the Sequential Function Chart is stopped. Any time overrun in the SFCError variable is reset. When the variable is reset to FALSE, all existing times in the active steps are reset.

**SFCError:** This Boolean variable is set if a time overrun occurs in an SFC diagram.

**SFCErrorStep:** This variable is of the type String. If a time overrun occurs, the name of the step that caused the time overrun is stored in this variable.

**SFCErrorPOU:** If a time overrun occurs, this variable of the type String receives the name of the block in which the time overrun occurred.

If a time overrun occurs once and the SFCError variable is not reset, no further time overruns are recorded.

**'Extras' 'Times Overview'**

This command opens a window in which you can edit the time settings of your SFC steps:

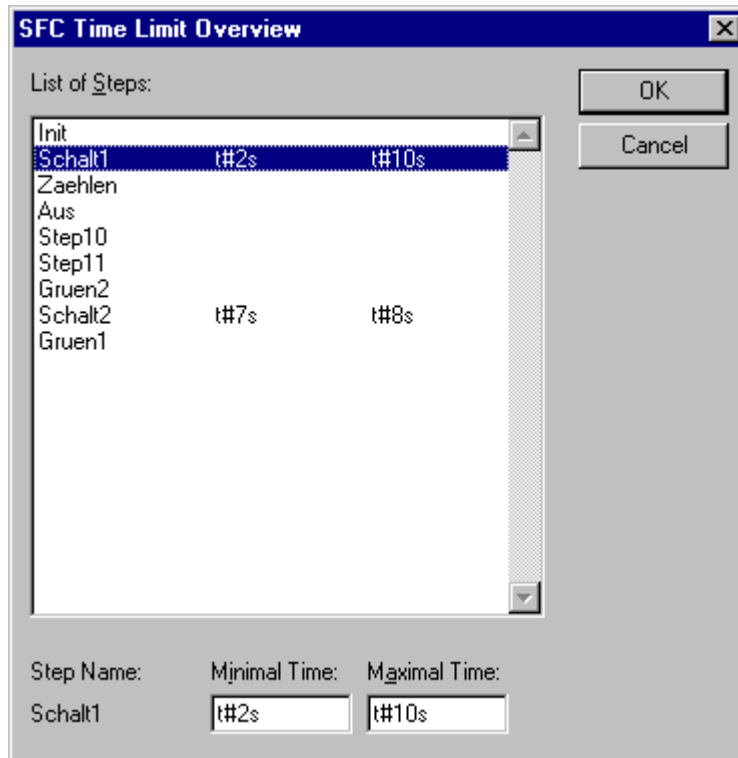


Figure 5\_14: Time limit overview for an SFC block

All the steps in your SFC block are displayed in the time limit overview. If you have specified a time limit for a step, this is displayed to the right of the step (lower limit followed by upper limit). You can also edit the time limits. To do this, click on the required step in the overview. If the **step name** is then displayed at the bottom of the window, go to the **Minimum time** or **Maximum time** field and enter the required time limit there. All changes are saved by closing the window using **OK**.

In the example, steps 2 and 6 have a time limit. Switch1 lasts for a minimum of two and a maximum of ten seconds. Switch2 lasts for a minimum of seven and a maximum of eight seconds.

**'Tools' 'SFC Overview'**

This command provides a reduced display of the active SFC block.

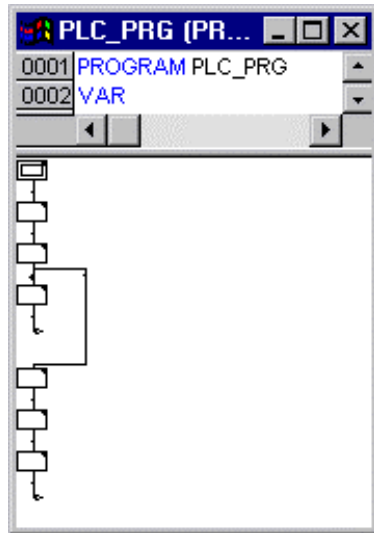


Figure 5\_15: SFC overview

A check mark appears in front of the menu item. For better orientation, display the names of the steps, transitions and jumps in tooltips, by placing the mouse pointer on an element. To switch to normal SFC view, set a mark, and switch to there by pressing the <Enter> key or by executing the command again.

**'Extras' 'Options'**

This command opens a dialog which you can set various options for your SFC block.

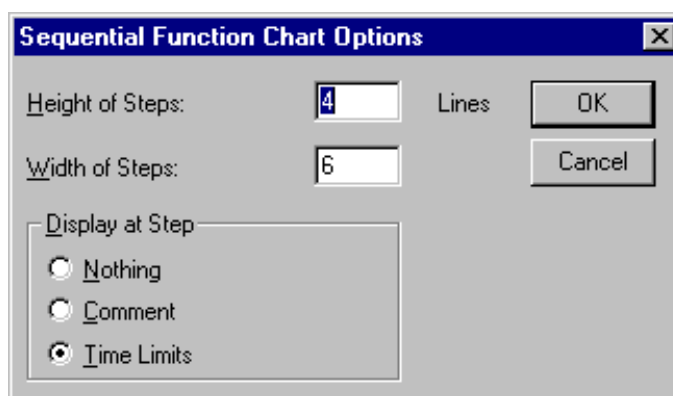


Figure 5\_16: Dialog for Sequential Function Chart Options

You can make five entries in the SFC options dialog. Under **Height of Steps**, you can enter how many lines high an SFC step in your SFC editor is to be. The default setting is four. Under **Width of Steps**, you can enter how many columns wide a step is to be. The default setting is six. You can also set which **Display at Step** is to be used. You have three options: you can display **Nothing**, the **Comment** or **Time Limits**. The latter two are displayed as they were entered under **'Extras' 'Step Attributes'**.

**‘Extras’ ‘Associate Action’**

This command allows you to associate actions and Boolean variables to IEC steps.

Another two-part box is added on the right beside the IEC step for the association of an action. The left-hand field is preset with the qualifier ‘N’ and the name ‘Action’. Both pre-settings can be changed. You can use the input help for the change.

New actions for IEC steps are created in the Object Organizer for an SFC block using the command ‘Project’ ‘Add action’.

**‘Extras’ ‘Use IEC Steps’**



If this command is activated (indicated by a check mark (✓) in front of the menu item and the “pressed” icon in the tool bar), IEC steps are inserted instead of simplified steps when inserting step transitions and parallel branches.

**‘Project’ ‘Add Action’**

This command generates an action for the selected SFC block in the Object Organizer which can be used in the IEC steps of this block. You select the action name and the language in which the action is to be implemented in the dialog that appears.

The new action is added to the Object Organizer under its SFC block. A plus sign now appears in front of the SFC block. A single click on the plus sign displays the action objects, and a minus sign appears in front of the block. A second click on the minus sign causes the actions to disappear, and the plus sign appears again. This can also be done using the context-sensitive menu commands ‘Expand node’ and ‘Collapse node’.

An action is loaded to the editor for editing by double-clicking on the action or by pressing the <Enter> key.

**Sequential Function Chart in Online Mode**

In the SFC editor, the currently-active steps are displayed as blue steps in online operation (black in the example). If you have set it under ‘Tools’ ‘Options’, time monitoring is displayed in addition to the steps. A third time specification appears under the upper and lower limits specified by you, from which you can read the length of time for which the step has already been active.



In the above diagram, the step represented has already been active for 8 seconds and 410 milliseconds. It must have been active for at least 7 minutes before the step is exited.

A breakpoint can be set on a step using ‘Online’ ‘Toggle Breakpoint’. Processing stops before execution of this step. The step with the breakpoint is light blue.

If several steps within a parallel branch are active, the active step whose action is to be processed next is displayed in red.

If IEC steps have been used, all active actions are displayed in blue during online operation.

SFC also support step-by-step processing ('Online' 'Step Over'). This means that you are always jumped to the next step whose action is being executed.

'Online' 'Step in' allows you to step to actions or transitions. All debugging functionalities of the corresponding editor are available to the user within transitions or actions.

If you hold the mouse pointer over a variable for a short time, the variable type and a comment on the variable are displayed in a tooltip.

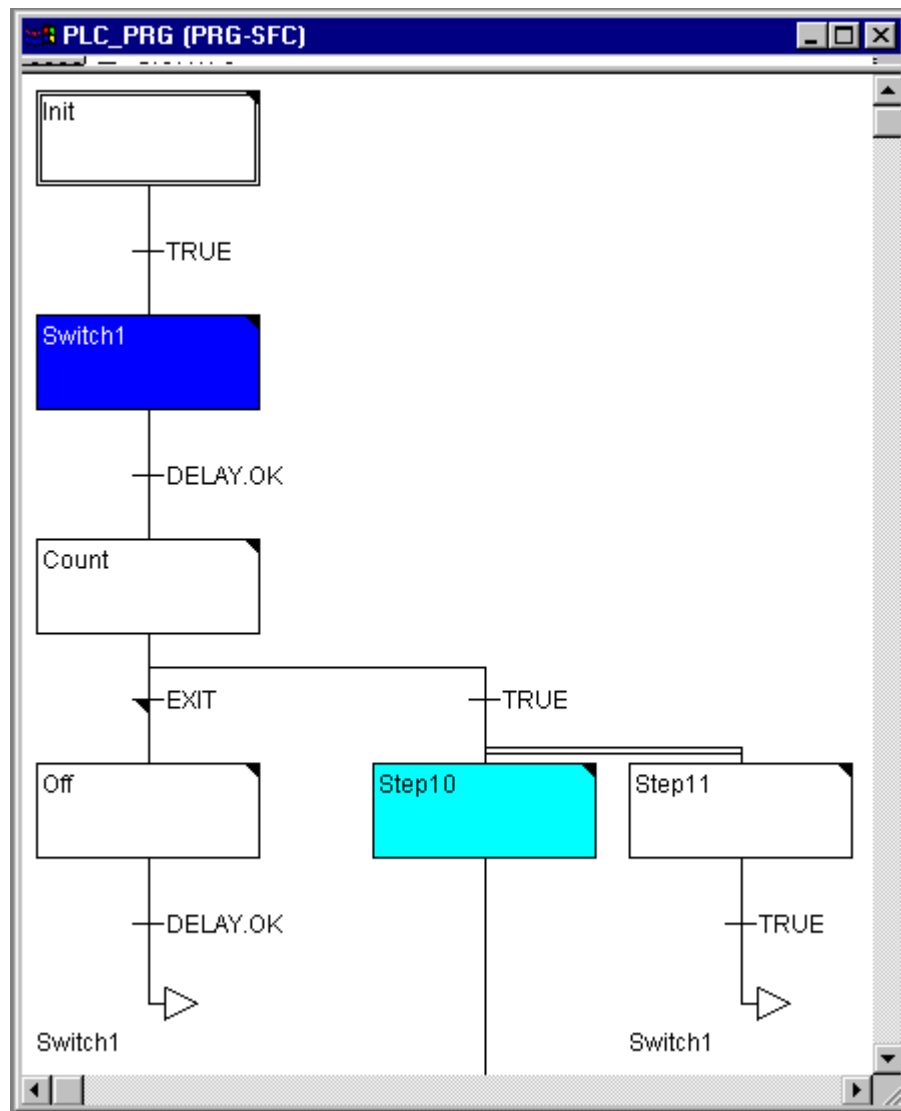


Figure 5\_17: Sequential Function Chart in online mode with one active step (Switch1) and one breakpoint (Step10)

Blank page

## The Resources

### Overview of the Resources

The **Resources** tab in the Object Organizer contains the following objects for configuring and organising your project and for tracing variable values:

- Global variables that can be used throughout the project
- Controller configuration for configuring your hardware
- Task configuration for controlling your program using tasks
- Trace recording for the graphical tracing of variable values
- Watch and receipt manager for displaying and pre-assigning variable values

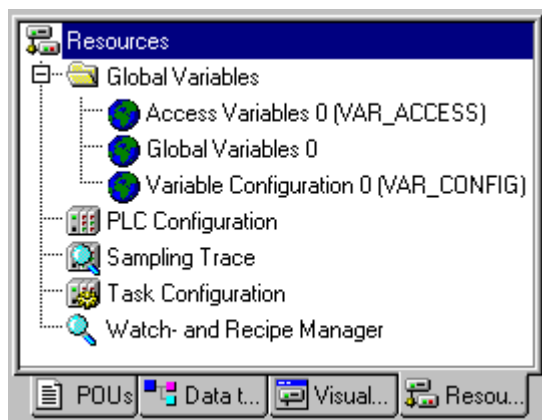


Figure 6\_1: Resources

## Global Variables

### Editing Global Variables

In the Object Organizer, the **Resources** tab in the **Global variables** folder contains two objects as standard (the pre-assigned names of the objects are enclosed in brackets):

- global variable list (Global\_Variables)
- variable configuration (Variable\_Configuration)

All variables defined within these objects are known throughout the project.


If the Global variables folder is not opened (as indicated by a plus sign in front of the folder), you can open it by double-clicking or by pressing the <Enter> key in the line.

Select the corresponding object. The **'Open object'** command opens a window containing the global variables that have already been defined. The editor works in the same way as the declaration editor.

## Multiple Variable Lists


Normal global variables (**VAR\_GLOBAL**) and variable configurations (**VAR\_CONFIG**) must be defined in separate objects.

If you have declared a large number of global variables, and you want to structure your global variable list better, you can create additional variable lists.

In the Object Organizer, select the **Global variables** folder or one of the existing  objects with global variables and execute the command 'Project' 'Add Object'. Enter a suitable name for the object in the dialog box that appears. This creates another object with the keyword **VAR\_GLOBAL**.

If you would prefer to have an object with a variable configuration, change the keyword accordingly to **VAR\_CONFIG**.

## Global Variables

Normal global variables are made available in an object in the **Resources** tab under the  **Global\_variables** object. The object can be renamed, and other objects can be created for global variables.

The global variables editor works in the same way as the declaration editor.

Global variables can be used throughout the project.

Syntax:

```
VAR_GLOBAL
  (* variable declarations *)
END_VAR
```

Retentive global variables have the additional keyword **RETAIN**.

Syntax:

```
VAR_GLOBAL RETAIN
  (* variable declarations *)
END_VAR
```

Global constants have the additional keyword **CONSTANT**.

Syntax:

```
VAR_GLOBAL CONSTANT
  (* variable declarations *)
END_VAR
```


## Variable Configuration

In function blocks, variables that are defined between the keywords **VAR** and **END\_VAR** may contain incompletely-specified addresses for inputs and outputs. Incompletely-specified addresses are marked with an asterisk.

Example:

```
FUNCTION_BLOCK locio
VAR
    loci AT %I*: BOOL := TRUE;
    loco AT %Q*: BOOL;
END_VAR
```

Two local I/O variables are defined here, one local in (%I\*) and one local out (%Q\*).

If you want to configure local I/Os, the  **Variable\_configuration** object is provided as standard in the **Resources** tab in the Object Organizer for variable configuration. The object can be renamed and other objects can be created for the variable configuration. The editor for variable configuration works in the same way as the declaration editor.

Variables for local I/O configuration must be placed between the keywords **VAR\_CONFIG** and **END\_VAR**.

The name of this type of variable consists of a full instance path in which the individual block and instance names are separated by dots. The declaration must contain an address whose class (input/output) corresponds to the incompletely-specified address (%I\*, %Q\*) in the function block. The data type must also correspond with the declaration in the function block.

Configuration variables whose instance path is invalid because the instance does not exist are identified as errors. Correspondingly, an error is also reported if no configuration exists for an instance variable. To obtain a complete list of all required configuration variables, the menu command '**All instance paths**' in the '**Insert**' menu can be used.

Example: A program contains the following definition of function blocks:

```
PROGRAM PLC_PRG
VAR
    Hugo: locio;
    Otto: locio;
END_VAR
```

A correct variable configuration looks like this:

```
VAR_CONFIG
    PLC_PRG.Hugo.loci AT %IX1.0 : BOOL;
    PLC_PRG.Hugo.loco AT %QX0.0 : BOOL;
    PLC_PRG.Otto.loci AT %IX1.0 : BOOL;
    PLC_PRG.Otto.loco AT %QX0.3 : BOOL;
END_VAR
```

### 'Insert' 'All Instance Paths'

This command generates a **VAR\_CONFIG - END\_VAR** block containing all instance paths in the project. Declarations that already exist are not re-inserted, in order to retain existing addresses. This menu item is found in the variable configuration window if the project has been compiled ('**Project**' '**Compile all**').

### Document Template

If you must document a project several times, possibly with comments in English and in another language, or if you want to document several similar projects that use the same variable names, you can considerably reduce the amount of work involved by using the command '**Extras**' '**Make Docuframe File**'.

You can load and edit the file created in any text editor. The file begins with the line **DOKUFILE**, and this is followed by a list of project variables. There are three lines before each variable: a **VAR** line which indicates that a new variable is coming, a line containing the name of the variable and then an empty line. This line can now be replaced by a comment on the variables. Variables that you do not want to document can simply be deleted from the text. You can create any number of document templates for your project.

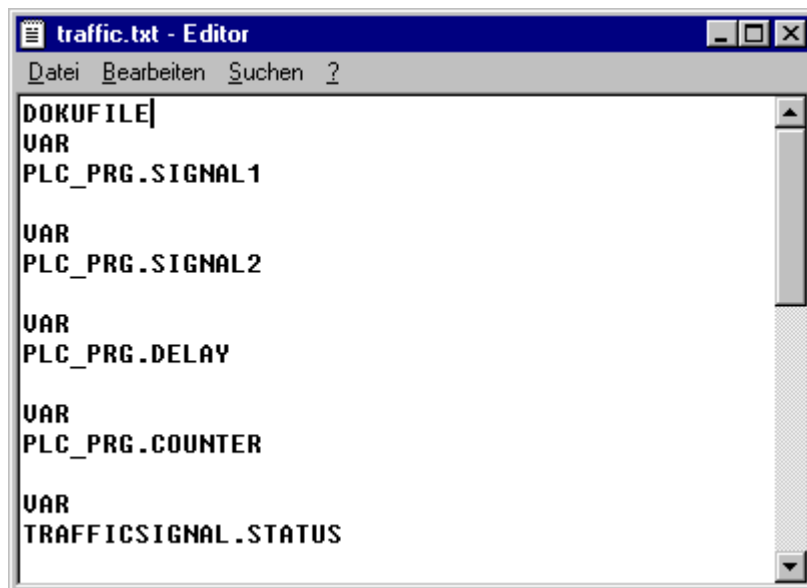


Figure 6\_2: Windows editor with document template

To use a document template, enter the command '**Extras**' '**Link Docu File**'. If you now document the entire project or print a part of your project, the comment you created in the document template is inserted in the program text for all variables. However, this comment only appears in the printout.

### **'Extras' 'Make Docuframe File'**

This command creates a document template. The command is available when a global variables object is selected.

A dialog opens for saving files under a new name. The extension \*.txt is entered in the field for the **file name**. Select any name. A text file is now created listing all variables in your project.

### **'Extras' Link Docu File'**

This command selects a document template.

The dialog for opening files opens. Select the required document template and press **OK**. If you now document the entire project, or print part of your project, the comment you created in the document template is inserted in the program text for all variables. However, this comment only appears in the printout.

To create a document template, use the command **'Extras' 'Make Docuframe File'**.


---

## **PLC Configuration**

The PLC Configuration is dependent on the hardware to be configured. For this reason, only the basic operation of the **CP1131** hardware configuration is described here.

Generally speaking, only digital inputs/outputs can currently be declared with the controller configuration. Analog inputs/outputs must be addressed using a corresponding library function (qaiolib library).

Only digital inputs/outputs of the CPU modules CEDIO 16/16-0,5 and CDIO 16/16-0,5, as well as the inputs/outputs of the digital expansion component QDIO 16/16-0,5 can be addressed with the controller configuration.

The  PLC Configuration is found as an object in the **Resources** tab in the Object Organizer. The hardware for which the opened project was created must be described using the controller configuration editor. The amount and location of the inputs and outputs are important for program creation. **CP1131** uses this description to check whether the IEC addresses used in the program actually exist in the hardware.

Furthermore, you can use the PLC Configuration editor to assign identifiers to inputs/outputs.

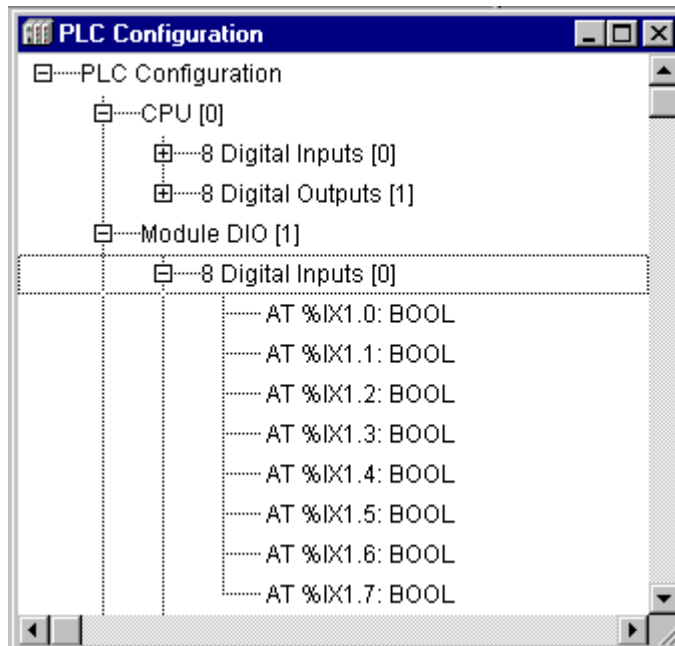


Figure 6\_3: PLC Configuration of a CPU module with a connected QDIO 16/16-0,5 CANtrol expansion module.

Symbolic names can be assigned to inputs and outputs. After the symbolic name comes the IEC address with which this input or output can be accessed.

### Working in the Controller Configuration

When a new project is created, a minimum PLC Configuration is provided.

- To select elements, click on the corresponding element using the mouse or move the dotted rectangle to the required element using the arrow keys.
- At the top of the controller configuration are the words "Hardware configuration". Elements preceded by a plus sign are organisation elements, and contain sub-elements. To open them, select the element and double click on the plus sign or press the <Enter> key. Open elements (indicated by a minus sign in front of the element) can be closed in the same way.
- If the cursor is positioned on an element, you can place an editing frame around the name by double-clicking on the entry or by pressing the <space bar>. You can then change the identifier of the input/output.
- To store information on an input/output block use the command **'Extras' 'Edit Entry'**.
- The command **'Insert' 'Insert Element'** inserts your chosen element in front of the selected element.
- The command **'Insert' 'Append Subelement'** adds your chosen element to the selected element as the last sub-element.
- The most important commands are found in the context-sensitive menu (right-hand mouse button or <Ctrl>+<F10>).

### **'Extras' 'Edit Entry'**

**Shortcut:** <Enter> key

This command opens a dialog box. You can store additional information on an input/output block in the controller configuration. (Example: an input block is a temperature regulator, and the information consists of regulatory parameters or a comment on a simple digital input).

You can also obtain the dialog box by selecting the required element and double-clicking on it.

### **'Insert' 'Insert Element'**

This command inserts your chosen element into the controller configuration in front of the selected element.

### **'Insert' 'Append Subelement'**

This command adds your chosen element to the selected element in the controller configuration as the last sub-element.

## **PLC Configuration in Online Mode**


In online mode, the controller configuration displays the states of the controller's inputs and outputs. If a Boolean input or output has the value **'TRUE'**, the box in front of the input or output is displayed in blue.

The Boolean inputs can be toggled with a mouse click. For other elements, a dialog appears for entry of the new value. The new value is set in the controller as soon as you confirm it using **OK**.

---

## **Task Configuration**

As well as declaring the special program PLC\_PRG, you can also control processing of your project using task management.

The  Task configuration is an object in the **Resources** tab in the Object Organizer. For this, you specify a sequence of tasks in the task editor. The task declaration consists of the name of the task, an entry for the priority that the task is to have, and an entry for the condition under which the task is to be executed. This condition can be either a time interval after which the task is to be executed or a global variable that causes an execution when it has a rising edge.

For each task, you can now specify a sequence of programs to be started by the task. If the task is executed in the current cycle, these programs are processed for one cycle.

The task configuration is displayed in the following format:

- the first line contains the text **'Task configuration'**
- indented below this is a sequence of task entries (with name, priority, interval and event)
- below each task entry is a sequence of program calls.

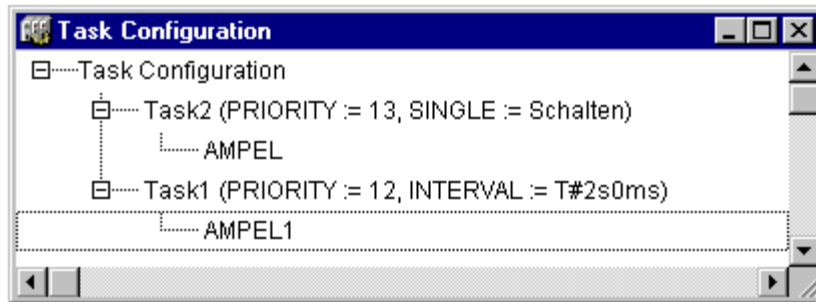


Figure 6\_4: Example of a task configuration

In this example of a task configuration, Task2 has a lower priority than Task1. However, Task1 is only executed every two seconds (the entry under Event is ignored). Therefore, in this task configuration, Task1 is executed every two seconds, and Task2 can be executed in between, provided that the global variable “Switch” has a rising edge.

### Which Task is Processed?

The following rules apply to the execution of the task:

- The task whose condition is valid is processed. This means the task whose specified time has elapsed or after a rising edge of its condition variable.
- If several tasks have a valid condition, the task with the highest priority is executed.
- If several tasks have a valid condition and an equal priority, the task with the longest queuing time is processed.
- The most important commands are found in the context-sensitive menu (right-hand mouse button or <Ctrl>+<F10>).

### Working in the Task Configuration

- At the top of the task configuration are the words “Task configuration”. If the words are preceded by a plus sign, the list is closed. To open it, double click on the plus sign or press the <Enter> key. A minus sign now appears. The list can be closed by double-clicking on the minus sign.
- A list of program calls is attached to each task. You can open and close this list also.
- The command ‘Insert’ ‘Insert Task’ inserts a task.
- The command ‘Insert’ ‘Append Program call’ inserts a program call for a task.
- The command ‘Extras’ ‘Edit Entry’ allows you to edit the task properties or the program call, depending on the element selected.
- Clicking on a task or program name or pressing the <space bar> sets an editing frame around the name. You can then edit the name directly in the task editor.

### 'Insert' 'Insert Task' or 'Insert' 'Add Task'

This command adds a new task to the task configuration.

If a task has been selected, the command **'Insert task'** is available. The new task is inserted before the cursor. If the words **'Task configuration'** are selected, the command **'Add task'** is available and the new task is added to the end of the list.

The dialog for definition of task properties opens.

Figure 6\_5: Dialog for defining task properties

You enter the required attributes in the dialog: **name**, **priority** (a number between 0 and 31, where 0 has the highest priority and 31 has the lowest) and the **interval** after which the task is to be restarted or a variable which causes execution of the task after a rising edge (in the **Event** field). The **Select...** button opens the input help for selection from the declared variables.

If there is an entry in both Interval and in the variables, only the time interval is taken into account as an execution condition. If no entry is made in either of the fields, only the priority is taken into account in the calculation. This means that the task can be considered to be executable in every cycle, and the only reason for non-execution is if another task with a higher priority is also executable.

### 'Insert' 'Append Program Call' or 'Insert' 'Insert Program Call'

These commands open a dialog for entry of a program call for a task in the task configuration.

With **'Insert program call'**, the new program call is inserted before the cursor, and with **'Append program call'** it is added to the end of the existing list.

Figure 6\_6: Dialog for entry of a program call

Enter a valid program name for your project in the field, or use the **Select** button to select a valid program name from the input help. If the selected program requires input variables, these are specified in the usual way from the declared type (e.g. prg(invar:=17)).

**'Extras' 'Edit Entry'**

Depending on the element selected, this command opens either the dialog for defining the task properties (see '**Insert task**') or the dialog for entering the program call (see '**Append program call**') in the task configuration.

If the cursor is positioned on a task entry and there is still no list of program calls at the task entry, you can open the dialog for defining the task properties by double-clicking on the entry or by pressing the <Enter> key.

If the cursor is positioned on an entry for a program call, you can open the dialog for editing the program entry by double-clicking on the entry.

An editing frame is set around the name by clicking on a task or program name or by pressing the <space bar>. You can then edit the identifier directly in the task editor.

**'Extras' 'Define Debug Task'**


This command allows you to define a task for debugging in online mode in the task configuration. The text [DEBUG] is displayed after the defined task.

Debugging functionalities then relate to this task only, i.e. the program only stops at a breakpoint if the program is run by the task set.

## Trace Recording

Trace recording means that the value sequence of variables is traced for a set period of time. These values are written to a ring memory known as the trace buffer. When the memory becomes full, the “oldest” values at the beginning of the memory are overwritten. Up to 20 variables can be recorded at the same time. A maximum of 500 values can be recorded for each variable.

As the size of the trace buffer in the controller has a fixed value, less than 500 values may be recorded if the variables are very numerous or very wide-ranging (DWORD). Example: if 10 WORD variables are to be recorded and the memory in the controller is 5,000 bytes long, 250 values of each variable can be recorded.

To record a trace, open the  **Trace recording** object in the **Resources** tab of the Object Organizer. You must then enter the trace variables to be recorded (see ‘Extras’ **Trace configuration**). Recording of variable values begins after you have sent the configuration to the controller using **Define trace** and started recording (**Start trace**). **Read trace** reads out the values traced and displays them graphically in the form of curves.

### ‘Extras’ ‘Trace Configuration’

This command displays the dialog for entry of the variables to be traced, as well as various trace parameters for the trace recording.

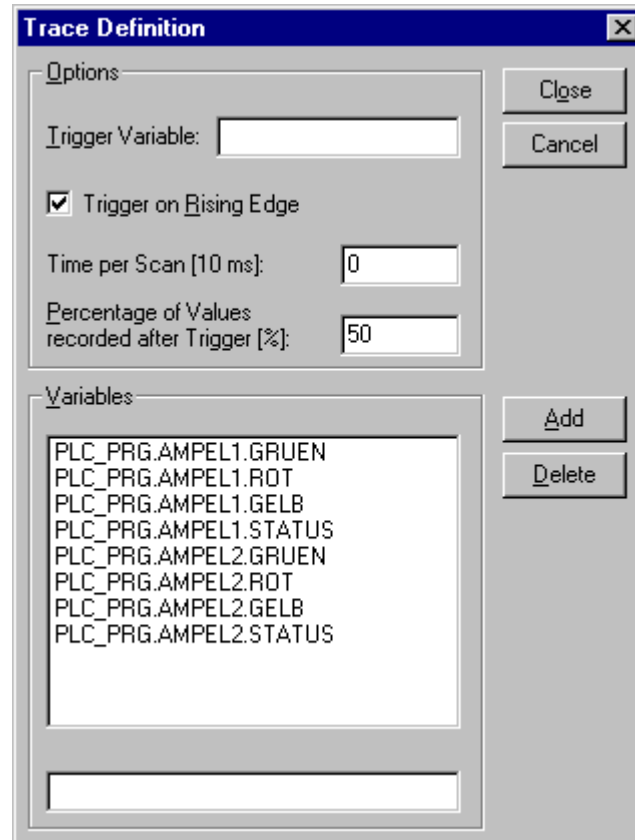


Figure 6\_7: Dialog for trace configuration

At first, the list of **variables** to be traced is empty. To add a variable, it must be entered in the field under the list. It can then be added to the list using the **Insert** button or the <Enter> key. You can also use the input help.

A variable is deleted from the list by selecting it and then pressing the **Delete** button. In the **Sampling rate** field you can enter the time between two tracings in milliseconds. The presetting "0" means one scan process per cycle.

A Boolean variable can be entered in the **Trigger variable** field. You can also use the input help. The variable describes the condition for terminating the trace. If **Rising edge** is selected in the trace definition (as in the example above), the trigger event intervenes after a rising edge of the trigger variables. Otherwise, it intervenes after a falling edge.

A certain number of values is recorded after the trigger event, and then the trace is terminated. This number can be entered as a percentage in the field **Percentage of Values recorded after Trigger**.

If the **Trigger variable** field is empty, trace recording must be terminated explicitly ('Extras' 'Stop trace').

### 'Extras' 'Define Trace'

Icon: 

This command loads the trace configuration created to the controller. It is retained there until the condition for terminating the trace has been fulfilled, or recording is stopped ('Extras' 'Stop trace').

### 'Extras' 'Start Trace'

Icon: 

This command starts trace recording in the controller.

### 'Extras' 'Read Trace'

Icon: 

This command reads the current trace buffer from the controller and the values of the selected variables are displayed.

### 'Extras' 'Auto Read'

This command causes the current trace buffer to be read automatically from the controller, and the values are displayed as they are obtained.

If the trace buffer is to be read automatically, a check mark (✓) appears in front of the menu item.

**'Extras' 'Stop Trace'**



This command stops trace recording in the controller. In order to restart recording, the trace definition must be loaded and the trace must be restarted.

**Selecting the Variables to be Displayed**

The comboboxes on the right beside the window for displaying the curves contain all trace variables defined in the trace configuration. If a variable is selected from the list, it is output in the corresponding color after a trace buffer is read (Var 0 green, etc.). Variables can also be selected if curves have already been output.

Up to eight variables can be monitored simultaneously in the trace window.

**Displaying the Trace Recording**

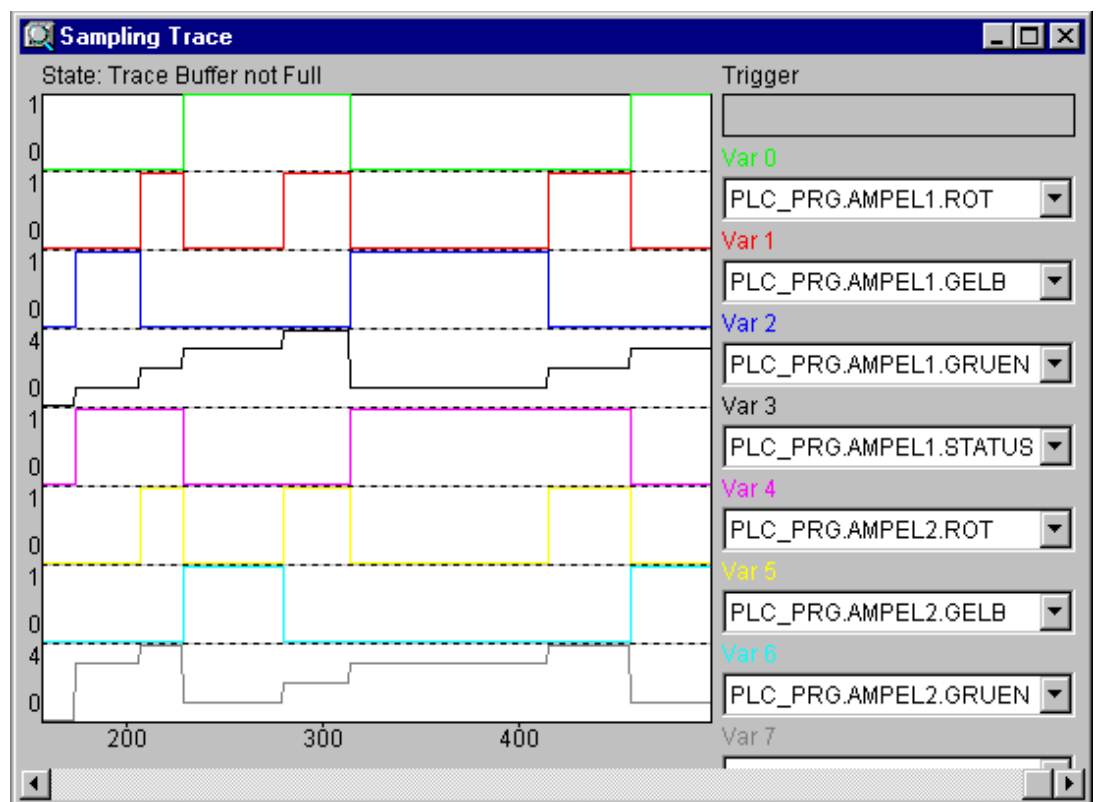


Figure 6\_8: Trace recording of 8 different variables without trigger

If a trace buffer has been loaded, the values of all variables to be displayed are read out and displayed. If no sampling rate has been set, the X axis is labelled with the consecutive number of the value recorded. This status display of the trace window (first line) shows whether the trace buffer is full, and when the trace will be finished.

If a value has been specified for the sampling rate, the X axis specifies the time of the value. The "oldest" value recorded is assigned the time 0. The above example displays the values for the previous 25 seconds.

The Y axis is labelled with integer values. The scale is adapted to ensure that the lowest and highest values fit onto the screen. In the example, Var5 has the lowest value 6 and the highest value 11, and the scale on the left edge is adjusted accordingly.

If the trigger condition has been fulfilled, a horizontal line is output at the interface between the values before and after the trigger condition.

A memory that has been read is retained until the project is changed or the system is exited.

**'Extras' 'Cursor Mode'**

This command causes a vertical line to appear in the trace recording. This line can be moved to the right and left using the mouse or the arrow keys. The speed of movement can be increased by pressing <Ctrl>+<left> or <Ctrl>+<right>.

The current x position of the cursor can be read in the graphic window. The value of the relevant variable is displayed beside Var0,.Var1, ... ,VarN.

A cursor is also displayed when the mouse pointer is located in the graphic window and the left-hand mouse button is pressed.

**'Extras' 'Multi-Channel'**

This command allows you to switch between single-channel and multi-channel display of the trace recording. If multi-channel display is selected, a check mark (✓) appears in front of the menu item.

The presetting is multi-channel display. The display window is split up to eight times, depending on the number of curves to be displayed. The maximum and minimum values are output at the edge for each curve.

In single-channel display, all curves are displayed and overlaid with the same scale factor. This is useful for displaying deviations between curves.

**'Extras' 'Y Scaling'**

This command allows you to change the preset Y scale of a curve in the trace display.

Enter the number of the required curve in the dialog (**channel**), together with the new highest value (**Max. Y value**) and the new lowest value (**Min. Y value**) on the Y axis.

The dialog can also be obtained by double-clicking on a curve. The channel and the previous values are preset.

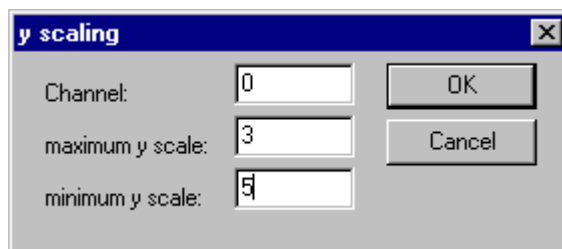


Figure 6\_9: Dialog for setting the Y scale

### 'Extras' 'Stretch'

Icon: 

This command allows you to enlarge the values output during trace recording. The start position is set using the horizontal scroll bar. If you enlarge several times in succession, the trace section displayed in the window becomes progressively shorter.

This command is the opposite of 'Extras' 'Compress'.

### 'Extras' 'Compress'

Icon: 

This command allows you to compress the values output during trace recording, i.e. this command allows you to view the progression of trace variables over a larger time span. This command can be executed several times in succession.

This command is the opposite of 'Extras' 'Stretch'.

### 'Extras' 'Save Trace'

This command allows you to save a trace recording. A dialog opens for saving a file. The file name receives the extension "\*.trc".

The saved trace recording can be reloaded using 'Extras' 'Load Trace'.

### 'Extras' 'Load Trace'

This command allows you to reload a saved trace recording. The dialog for opening a file appears. Select the required file with the extension "\*.trc".

A trace recording can be saved using 'Extras' 'Save Trace'.

### 'Extras' 'Trace in ASCII File'

This command allows you to save a trace to an ASCII file. The dialog for saving a file opens. The file name receives the extension "\*.txt". The values are stored in the file in accordance with the following scheme:

```
CP1131 Trace
D:\CP1131\PROJECTS\LIGHTS.PRO
Cycle PLC_PRG.TIMER PLC_PRG.LIGHT1
0 2 1
1 2 1
2 2 1
.....
```

If no sampling rate has been set in the trace configuration, the cycle is placed in the first column, i.e. one value per cycle is recorded. In other cases, the time at which the variables were saved from the start of recording is entered in milliseconds.

The corresponding trace variable values are saved in the subsequent columns. The values are separated by spaces.


The relevant variable names are displayed in sequence one after another in the third line (PLC\_PRG.TIMER, PLC\_PRG.LIGHT1).

## Watch and Receipt Manager

### Watch and Receipt Manager

The watch and receipt manager allows you to display the values of variables you have found. The watch and receipt manager also allows you to preset variables with certain values and transfer them all to the controller in one action (**Write receipt**). Current controller values can also be read into and stored in the watch and receipt manager as pre-settings (**Read receipt**). These functions are useful for setting and recording control parameters.

All watch lists generated (**Insert** **New watch list**) are displayed in the left-hand column of the watch and receipt manager and can be selected by clicking with the mouse or by using the arrow keys. The corresponding variables are displayed in the right-hand area of the watch and receipt manager.

To use the watch and receipt manager, open the  **Watch and receipt manager** object in the **Resources** tab in the Object Organizer.

### Watch and Receipt Manager in Offline Mode

You can generate a number of watch lists in the watch and receipt manager in *offline mode* using the command **Insert** **New watch list**.

To enter the variables to be monitored, you can call a list of all variables using the input help or enter the variables using the keyboard with the following notation:

<block\_name>.<variable\_name>

There is no block name for global variables. They begin with a dot. The variable name can have several levels. Addresses can be entered directly.

Example of a multi-level variable:

PLC\_PRG.Instance1.Instance2.Structure.Componentname

Example of a global variable:

.global1.component1

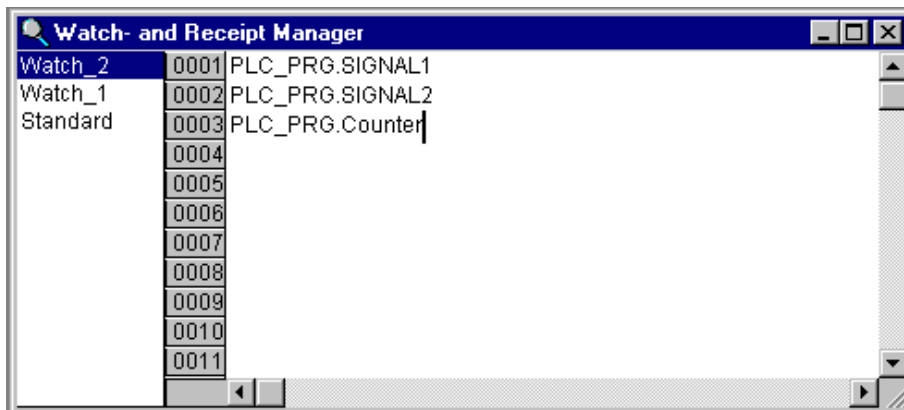


Figure 6\_10: Watch and receipt manager in offline mode

The variables in the watch list can be preset with constant variables, i.e. these values can be written to the variables in online mode using the command **'Extras' 'Write receipt'**. To do this, the constant value must be allocated to the variable using :=.

Example:

```
PLC_PRG.TIMER:=50
```

In this example, the PLC\_PRG.TIMER variable is preset with the value 6.

### **'Insert' 'New Watch List'**

This command inserts a new watch list into the watch and receipt manager. Enter the required name of the watch list in the dialog displayed.

### **'Extras' 'Rename Watch List'**

This command changes the name of a watch list in the watch and receipt manager.

Enter the new name of the watch list in the dialog that appears.

### **'Extras' 'Save Watch List'**

This command saves a watch list. A dialog opens for saving a file. The file name is preset with the name of the watch list and receives the extension "\*.wtc".

The saved watch list can be reloaded using **'Extras' 'Load watch list'**.

### **'Extras' 'Load Watch List'**

This command reloads a saved watch list. The dialog for opening a file is displayed. Select the required file with the extension "\*.wtc". You can rename the watch list in the dialog that appears. The presetting is the file name with no extension.

A watch list can be saved using **'Extras' 'Save watch list'**.

## **Watch and Receipt Manager in Online Mode**

The values of the variables entered are displayed in online mode.

Structured values (arrays, structures or instances of function blocks) are indicated by a plus sign in front of the identifier. The variable can be opened or closed by clicking on the plus sign with the mouse or by pressing the <Enter> key.

To enter a new variable, you can switch off the display using the command **'Extras' 'Monitoring active'**. After the variables have been entered, you can reactivate the display of values using the same command.

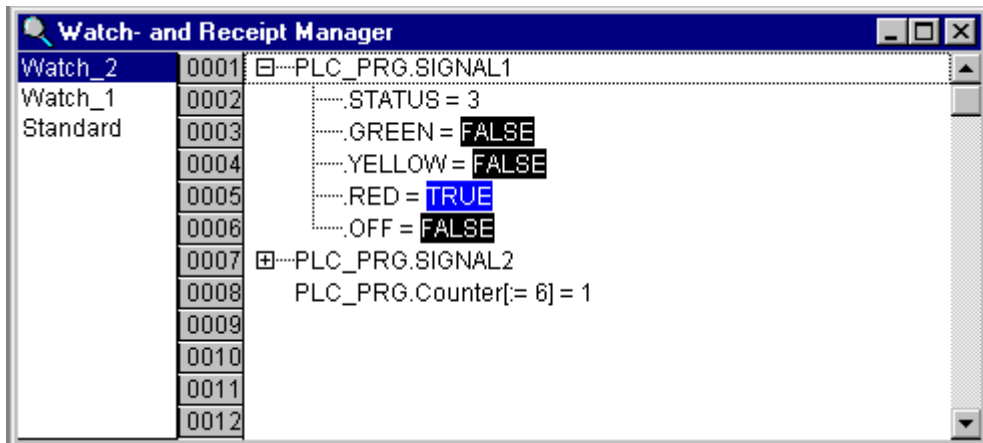


Figure 6\_11: Watch and receipt manager in online mode

Variables can be preset with constant values in *offline mode* (by entering := <value> after the variables). These values can then be written to the variables in *online mode* using the command 'Extras' 'Write Receipts'.

The preset variable values can be replaced with the current values using the command 'Extras' 'Read Receipt'.



**Note:** Only the values of one watch list selected in the watch and receipt manager are loaded.

### **'Extras' 'Monitoring Active'**

This command switches the display of the watch and receipt manager on or off in online mode. If the display is active, a check mark (✓) appears in front of the menu item.

To enter a new variable or preset a value (see offline mode), the display must be switched off using the command. After the variables have been entered, you can reactivate the display using the same command.

### **'Extras' 'Write Receipts'**

This command can be used to write the preset values to the variables in the watch and receipt manager in *online mode* (see offline mode).

**'Extras' 'Read Receipt'**

This command replaces the presettings of the variables (see offline mode) with the current variable values in *online mode* of the watch and receipt manager.

Example:

```
PLC_PRG.Timer [:= <current value>] = <current value>
```

**Force values**

You can also **'Force values'** and **'Write values'** in the watch and receipt manager. When you click on the relevant variable value, a dialog opens in which you can enter the new value of the variable. Changed variables are displayed in red in the watch and receipt manager.

Blank page

## Library Management

### Library Manager

The Library Manager shows all libraries linked to the current project. The blocks, data types and global variables of the libraries can be used in the same way as self-defined blocks, data types and global variables. The Library Manager is opened using the command '**Window** Library Manager'.

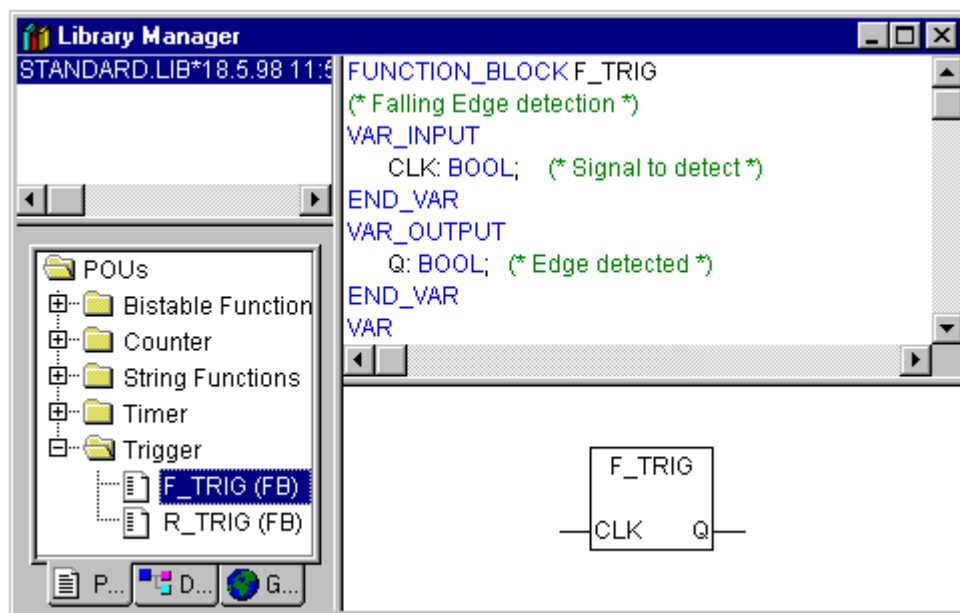


Figure 7\_1: Library Manager

Here we see the parameters of the F\_TRIG block from the library Standard.lib.

### Using the Library Manager

The window of the Library Manager is separated into three or four areas by screen split bars. The libraries linked to the project are listed in the upper left-hand area. In the area below this are listed the **blocks**, **data types** or **global variables** of the library selected in the upper area, depending on the tab selected.

Folders are opened and closed by double-clicking on the line or by pressing the <Enter> key. A plus sign is displayed in front of closed folders, and a minus sign is displayed in front of opened ones.

If a block is selected by clicking with the mouse or using the arrow keys, the declaration of the block appears at the top of the right-hand area, and the graphical representation appears at the bottom as a black box with inputs and outputs.

For data types and global variables, the declaration is displayed in the right-hand area of the Library Manager.

### **Standard Library**

The 'standard.lib' library is the default library available. It contains all the functions and function blocks required by IEC1131-3 as standard blocks for an IEC programming system. The difference between a standard function and an operator is that the operator is known implicitly to the programming system, while the standard blocks must be linked to the project as a library (standard.lib).

The code for these blocks is provided as a C library and is an integral part of **CP1131**.

### **User-Defined Libraries**

If a project is to be fully compiled free of errors, it can be stored in a library using the command '**Save as**' in the '**File**' menu. The project itself remains unchanged. It then becomes available under the name entered, in the same way as the standard library.

### **'Insert' 'Additional Library'**

This command allows you to link another library to your project.

In the dialog for opening a file, select the required library with the extension "'.lib"'. The library is now listed in the Library Manager and you can use the objects in the library in the same way as self-defined objects.

### **Deleting a Library**

You can delete a library from a project and from the Library Manager using the command '**Edit**' '**Delete**'.

## Visualization

### Creating a Visualization

#### Visualization

In order to illustrate your project variables, you have the option of creating a visualization. This visualization can be used to draw geometric elements offline, which can then change their color or shape online in response to certain variable values. For example, you can represent the growth behaviour of a variable in the form of a bar chart. You can also make entries for the program using the mouse and the keyboard.

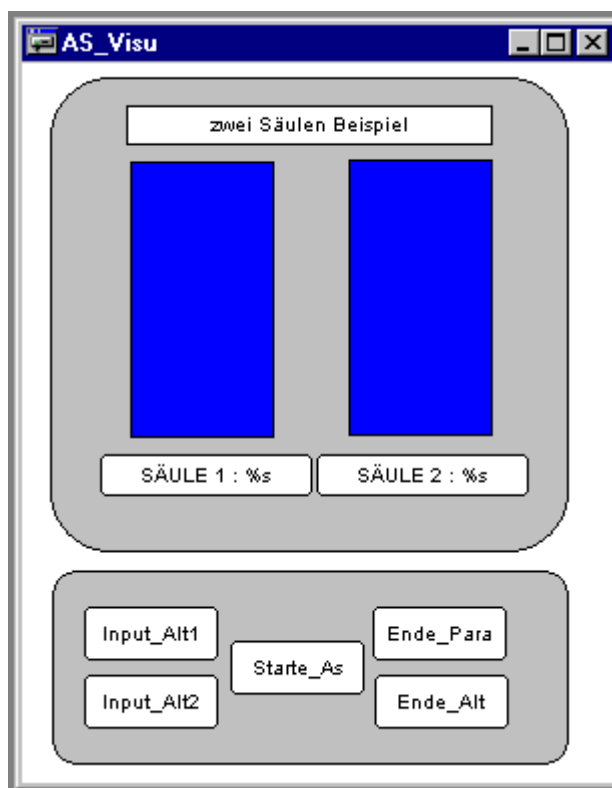



Figure 8\_1: Example of a visualization

#### Creating a visualization

To create a visualization, you must select the tab  **Visualization** in the Object Organizer.








You create a new visualization object by using the commands **'Project' 'Object' 'Add'**. A window opens for you to enter the name of the new visualization. When you have entered a valid name, you can close the dialog using **OK**. A window opens in which you can edit the new visualization.


## Inserting Visualization Elements

### Inserting Visualization Elements

You can insert four different geometric shapes, as well as bitmaps and existing visualizations, into your visualization.

The *geometric shapes* available are rectangles, rounded rectangles, ellipses/circles and polygons.

Go to the menu item **'Insert'** and select one of the commands  **'Rectangle'**,  **'Rounded rectangle'**,  **'Ellipse'**,  **'Polygon'**,  **'Bitmap'** or  **'Visualization'**. A check mark appears in front of the selected command. You can also use the icons in the tool bar. An element selected from the tool bar appears as a pressed button (e.g. ).

If you now move the mouse to the edit window, you will see that the mouse pointer is identified with the corresponding icon (e.g. ). Click wherever you want your element to start and drag the pointer while holding down the left-hand mouse button until the element reaches the required size.

If you want to create a polygon, first click on the position of the first corner of the polygon using the mouse, and then click on the other corners. A double-click on the last corner closes the polygon and completes the drawing.

You should also note the status bar and how it switches between select mode and insert mode.

### 'Insert' 'Rectangle'

Icon: 

This command inserts a rectangle as an element into your current visualization. (For information on operation, see the section entitled **'Inserting visualization elements'**.)

### 'Insert' 'Rounded Rectangle'

Icon: 

This command inserts a rectangle with rounded corners as an element into your current visualization. (For information on operation, see the section entitled **'Inserting visualization elements'**.)

### 'Insert' 'Ellipse'

Icon: 

This command inserts an ellipse or circle as an element into your current visualization. (For information on operation, see the section entitled **'Inserting visualization elements'**.)

### **'Insert' 'Polygon'**



This command inserts a polygon as an element into your current visualization. (For information on operation, see the section entitled '**Inserting visualization elements**').

### **'Insert' 'Bitmap'**



This command inserts a bitmap as an element into your current visualization. (For information on operation, see the section entitled '**Inserting visualization elements**').

Drag an area to the required size by holding down the left-hand mouse button. The dialog for opening a file appears. When you have selected the required bitmap, it is inserted in the area you have drawn.

### **'Insert' 'Visualization'**



This command inserts an existing visualization into your current visualization. (For information on operation, see the section entitled '**Inserting visualization elements**').

Drag an area to the required size by holding down the left-hand mouse button. A list of existing visualizations opens. When you have selected the required visualization, it is inserted into the area you have drawn.

---

## **Using Visualization Elements**

### **Selecting Visualization Elements**

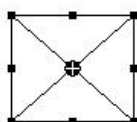
To select an element, click on it using the mouse.

To select several elements, hold down the <Shift> key and click on the relevant elements in sequence using the mouse, or draw a window over the elements to be selected by holding down the left-hand mouse button.

To select all elements, use the command '**Extras**' '**Select all**'.

### **Changing Visualization Elements**

You can select an element which has already been inserted by clicking on the element with the mouse. A black square now appears at each corner of the element (in the case of ellipses, they are at the corners of the surrounding rectangle). With the exception of polygons, other squares appear at the centres of the element edges, between the corner points.



The pivot (centre point) of a selected element is also displayed. When a movement or angle is set, the element rotates around this point. The pivot is displayed as a small black circle containing a white cross (⊕). You can move the pivot by holding down the left-hand mouse button.

The size of the element can be changed by clicking on one of the black squares and moving to the new outline while holding down the left-hand mouse button.

If a *polygon* is selected, each corner can be moved individually. When the <Ctrl> key is held down, an additional corner point is inserted at the corner point which can be moved by dragging the mouse. A corner point can be deleted by pressing the <Shift> + <Ctrl> keys.

### **Moving Visualization Elements**

One or more selected elements can be moved by pressing the left-hand mouse button or the arrow keys.

### **Copying Visualization Elements**

You can copy one or more selected elements using the 'Edit' 'Copy' command, the key combination <Ctrl> + <C> or the corresponding Copy icon, and insert it using 'Edit' 'Paste'.

Another option is to select the elements and click again in one element using the mouse, while holding down the <Ctrl> key. The copied elements can now be moved away from the original ones while holding down the left-hand mouse button.

### **Switching between Select Mode and Insert Mode**

After you have inserted a visualization element, you are switched back automatically to select mode. If you now want to insert another element of the same type, you can select the corresponding command in the menu or select the icon from the tool bar again.

You can also switch quickly between select and insert mode by pressing the <Ctrl> key and the right-hand mouse button simultaneously.

When you are in insert mode, the corresponding icon appears at the mouse pointer and the name is displayed in black in the status bar.

### **Status Bar in the Visualization**

In a visualization, the current **X** and **Y position** of the mouse pointer relative to the upper left-hand corner of the screen is always specified in pixels in the status bar. If the mouse pointer is positioned on an element, or if an element is being edited, its number is also specified. If you have selected an element for insertion, this is also displayed (e.g. **rectangle**).

## Configuring Visualization Elements

### 'Extras' 'Configure'

This command opens the dialog for configuring the selected visualization element.

The configuration dialog can also be obtained by double-clicking on the element.

Select a category from the left-hand area of the dialog and in the right-hand area, fill in the required specifications.

Various categories are offered for selection, depending on the visualization element selected:

- |                   |  |
|-------------------|--|
| • Shape           | Rectangle, rounded rectangle, ellipse          |
| • Text            | All  |
| • Color           | Rectangle, rounded rectangle, ellipse, polygon |
| • Motion absolute | All  |
| • Motion relative | All except polygon                             |
| • Variables       | All  |
| • Input           | All  |
| • Bitmap          | Bitmap   |
| • Visualization   | Visualization                                  |

### Shape

In the **Shape** category of the dialog for configuring visualization elements, you can choose between **rectangle**, **rounded rectangle** and **ellipse**. The shape changes within the measurements already defined.

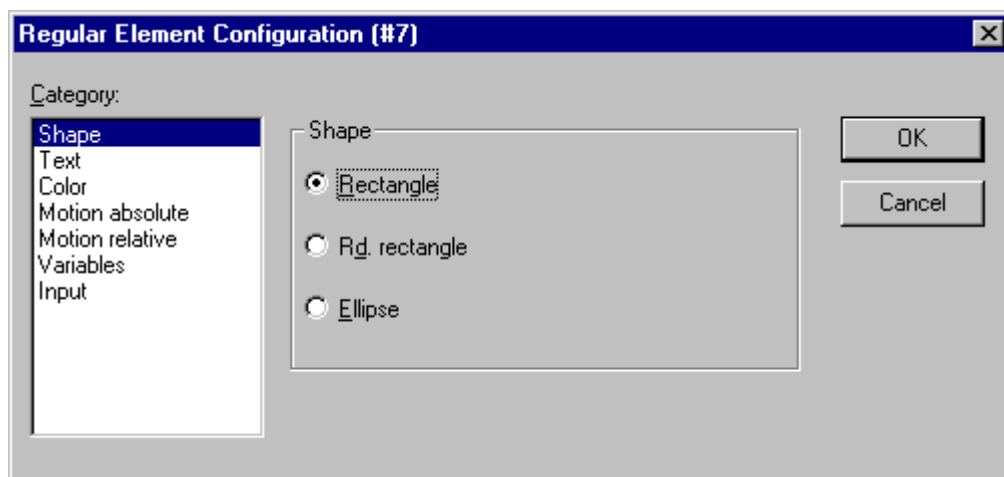


Figure 8\_2: Dialog for configuring visualization elements (Shape category)

**Text**

You can define a text for the element in the **Text** category of the dialog for configuring visualization elements.

Enter the text in the **Content** field. If you enter “%s” in the text, this position is replaced in online mode by the value of the variables in the **Text output** field of the category **Variables**.

This text appears in the element in the direction specified in the element: **horizontal right, centred or left** and **horizontal top, centred or bottom**.

The **Font** button displays the dialog for selecting the font. Select the required font and confirm the dialog using **OK**. The **Default font** button selects the font set under ‘**Project**’ ‘**Options**’. If it is changed there, this font is displayed in all elements except for elements for which a different font has been explicitly specified using the **Font** button.

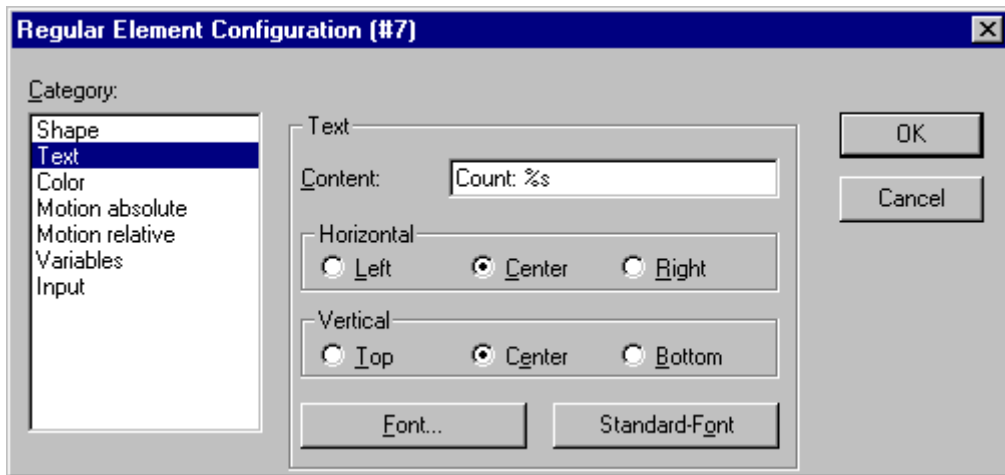


Figure 8\_3: Dialog for configuring visualization elements (Text category)

## Colors

In the **Colors** category of the dialog for configuring visualization elements, you can select basic colors and alarm colors for the inside and the frame of your element.

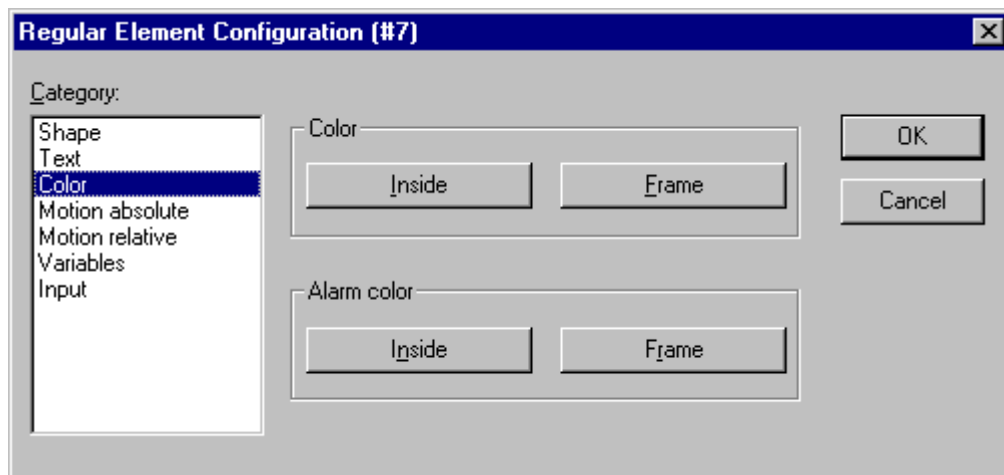


Figure 8\_4: Dialog for configuring visualization elements (Colors category)

If you enter a Boolean variable in the **Color change** field in the **Variables** category, the element is displayed in the **color** set, provided that the variable is FALSE. If the variable is TRUE, the element is displayed in its **Alarm color**.



**Note:** The color change function does not become active until the controller is in *on-line mode*.

If you want to change the color of the frame, press the **Frame** button instead of **Inside**. The dialog for selection of the color opens in both cases.

You can select the required color shade here from the basic colors and the colors you have defined yourself. You can change the self-defined colors by pressing the **Define colors** button.

## Motion Absolute

In the **Motion absolute** category of the dialog for configuring visualization elements, you can enter variables in the **X** and **Y Offset** fields to move the element in the X or Y direction, in accordance with the variable value set. A variable in the **Scale** field changes the size of the element linear to the variable value.

A variable in the **Angle** field causes the element to rotate around its pivot in accordance with the variable value set (positive value = mathematically positive = clockwise direction). The value is evaluated in degrees. In the case of polygons, every point rotates, which means that the polygon turns. In the case of all other elements, the object rotates, but the top edge always remains upwards.

The pivot is displayed when the element is clicked on once, and is displayed as a small, black circle with a white cross (⊕). You can move the pivot by holding down the left-hand mouse button.

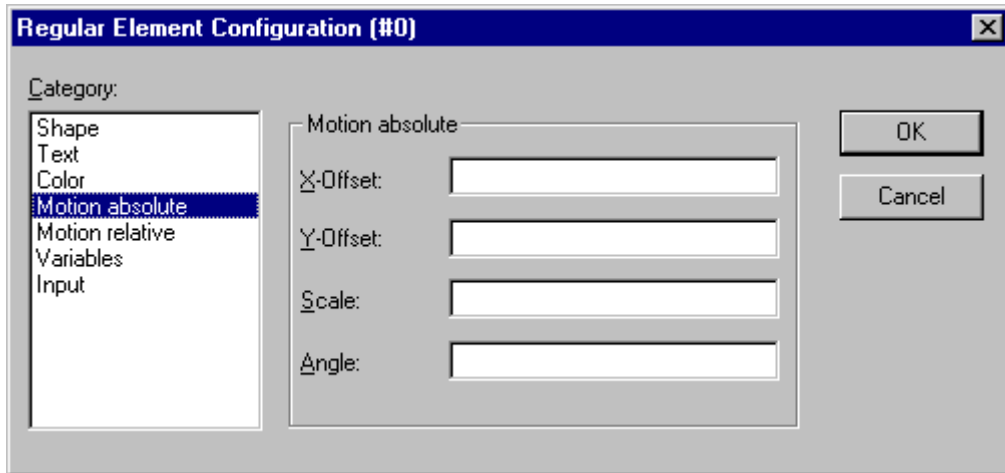


Figure 8\_5: Dialog for configuring visualization elements (Absolute movement category)

### Motion Relative

In the **Motion relative** category of the dialog for configuring visualization elements, you can allocate variables to the individual **edges** of the element. The corresponding element edges then move in accordance with the variables set. The easiest way to enter variables in the fields is to use the input help.

The four entries specify the four sides of your element. The basic position of the edges is always on zero, and a new variable value in the corresponding column moves the edge by this value in pixels. The variables entered should therefore be of the type INT.



**Note:** Positive values move the horizontal edges downwards and vertical edges to the right.

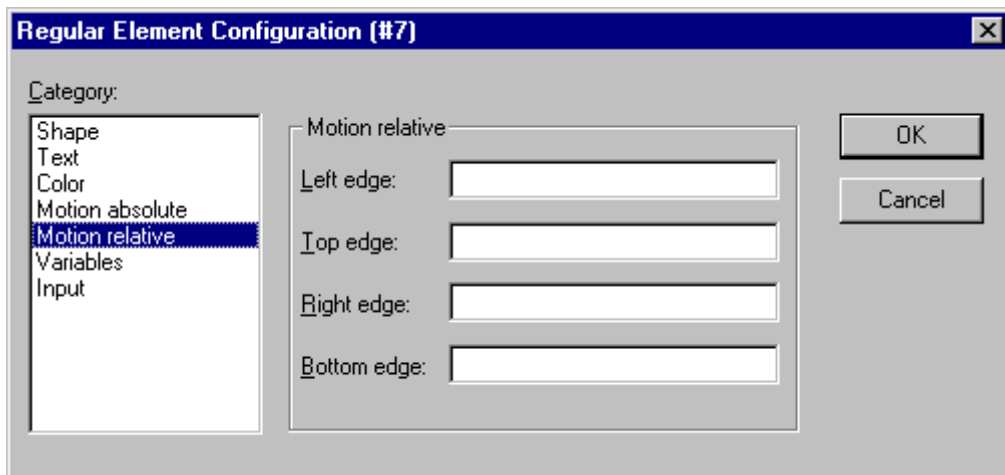


Figure 8\_6: Dialog for configuring visualization elements (Relative movement category)

## Variables

In the **Variables** category of the dialog for configuring visualization elements, you can specify the variables to describe the status of the visualization element. The easiest way to enter variables in the field is to use the input help.

In the **Invisible** and **Change color** fields, you can specify Boolean variables to initiate certain actions in accordance with their values. If the variable in the **Invisible** field has the value FALSE, the visualization element is visible. If the variable has the value TRUE, the element is invisible.

If the variable in the **Change color** field has the value FALSE, the visualization element is displayed in its basic color. If the variable is TRUE, the element is displayed in its alarm color.

In the **Text display** field, you can specify a variable whose value is output if you have entered “%s” in the **Content** field of the **Text** category in addition to the text. “%s” is replaced in online mode by the value of the variables in **Text output**.

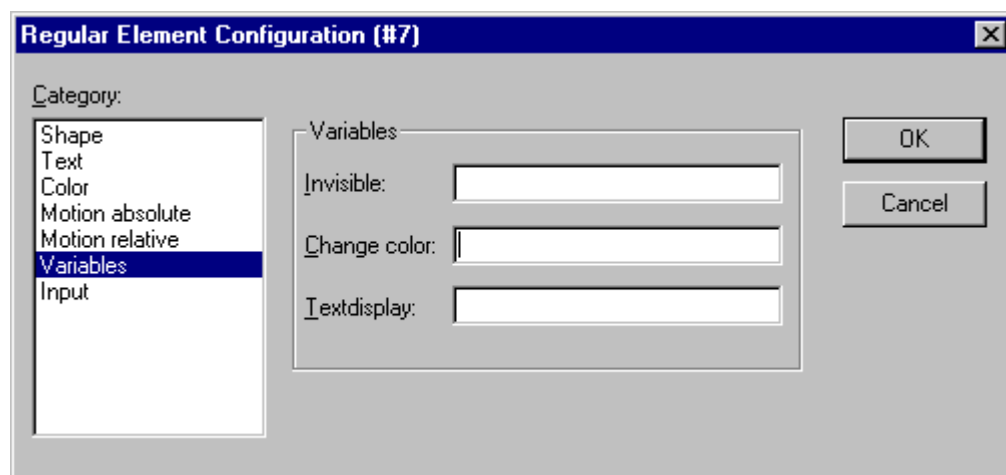


Figure 8\_7: Dialog for configuring visualization elements (Variables category)

## Input

In the **Input** category of the dialog for configuring visualization elements, you can specify whether entries can be made with the mouse and the keyboard in online mode. The field **No entries** is selected by default, which means that nothing happens when you click on the visualization element with the mouse.

If you select the **Toggle variable** field, each mouse click toggles the value of the variable specified in the input field behind. The value of the Boolean variable changes from TRUE to FALSE with one mouse click, and then back to TRUE with the next, and so on.

If you select the **Zoom to Vis...** field, you must specify the name of a visualization object in the same project in the subsequent field. In online mode, a mouse click switches you to the element in the window of the required visualization. If you select the option **and toggle variable**, the variable specified in the **Toggle variable** field is entered.

If you select the option **Text input of the variable 'Text display'**, you can allocate a value to a variable by means of this visualization element in online mode. The value is written to the variable in the **Text output** field of the **Variables** category. If you click on the element in online mode, an edit frame appears into which you can enter the new value of the variable using the keyboard. The value is imported by pressing the <Enter> key.

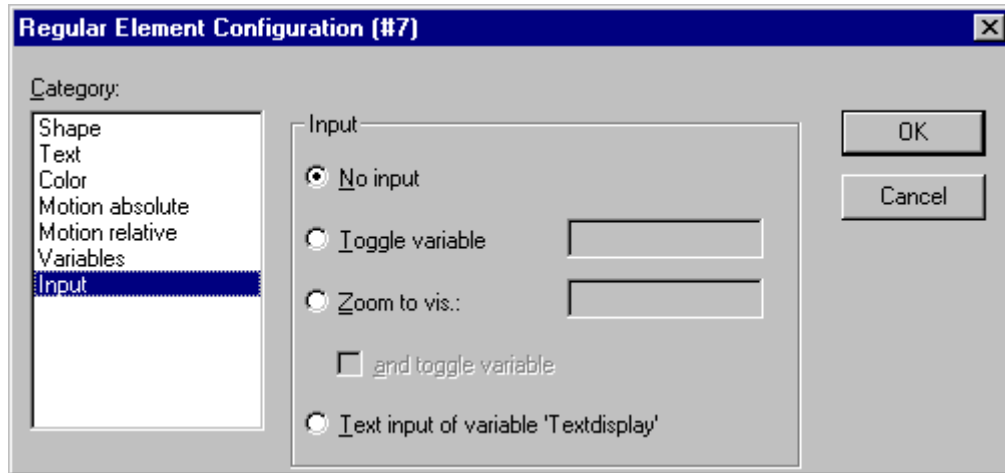


Figure 8\_8: Dialog for configuring visualization elements (Entry category)

## **Bitmap**

In the **Bitmap** category of the dialog for configuring visualization elements, you can specify options for a bitmap.

Specify the bitmap file with its path in the **Bitmap** field. The ... button opens the standard Windows dialog for browsing and you can select the required bitmap there.

All other specifications relate to the **frame** of the bitmap.

The settings **Anisotropic**, **Isotropic** and **Fixed** allow you to specify how the bitmap is to behave in relation to a change in the size of the frame. **Anisotropic** means that the bitmap is as big as its frame and can therefore be distorted by a change in size. **Isotropic** means that the proportions of the bitmap are retained after a change in size, i.e. the relations between length and width are retained. If you select **Fixed**, the bitmap is displayed in the original size, regardless of the frame.

If the **Cut** option is selected, the **Fixed** setting means that only the section of the bitmap enclosed in the frame is displayed.

If you select the **Draw** option, the frame is displayed in the color selected using the **Color** and **Alarm color** buttons in the color dialogs. The alarm color is only displayed if the variable specified in the **Change color** field of the **Variables** category is TRUE.

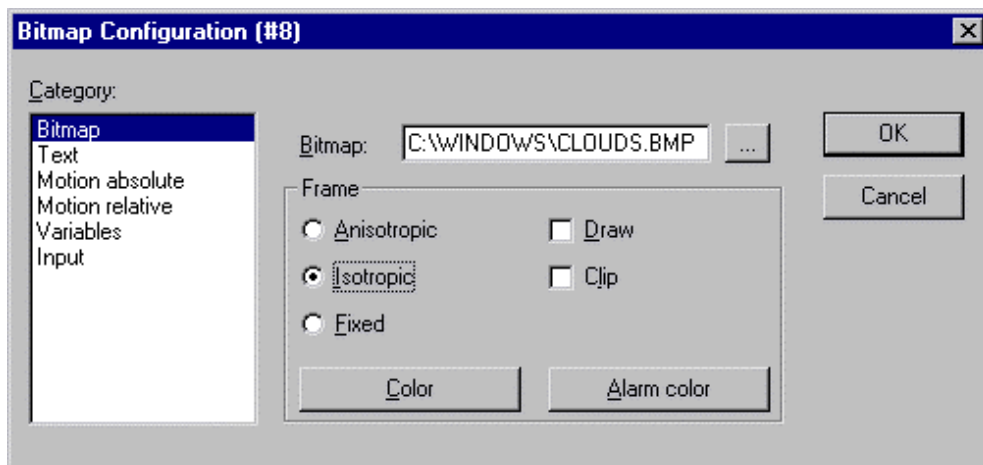


Figure 8\_9: Dialog for configuring visualization elements (Bitmap category)

### Visualization

In the **Visualization** category of the dialog for configuring visualization elements, you can specify the options for using a visualization as an element in another visualization. Enter the object name of the visualization in the **Visualization** field. The ... button opens a dialog containing the visualizations available in this project. All visualizations can be specified, with the exception of the current one. All other specifications relate to the **Frame** of the visualization.

If you select the **Draw** option, the frame is displayed in the color selected using the **Color** and **Alarm color** buttons in the color dialogs. The alarm color is only displayed if the variable specified in the **Variables** category in the **Change color** field is TRUE. If you select **Isotropic**, the proportions of the visualization remain unchanged, even when the size is changed, i.e. the relations between length and width remain the same. Otherwise, the visualization can be distorted.

If the option **Crop** is selected, only the original section of the visualization is displayed in online mode. If, for example, an object moves outside the original display, it is cropped and may disappear from the field of view of the visualization.

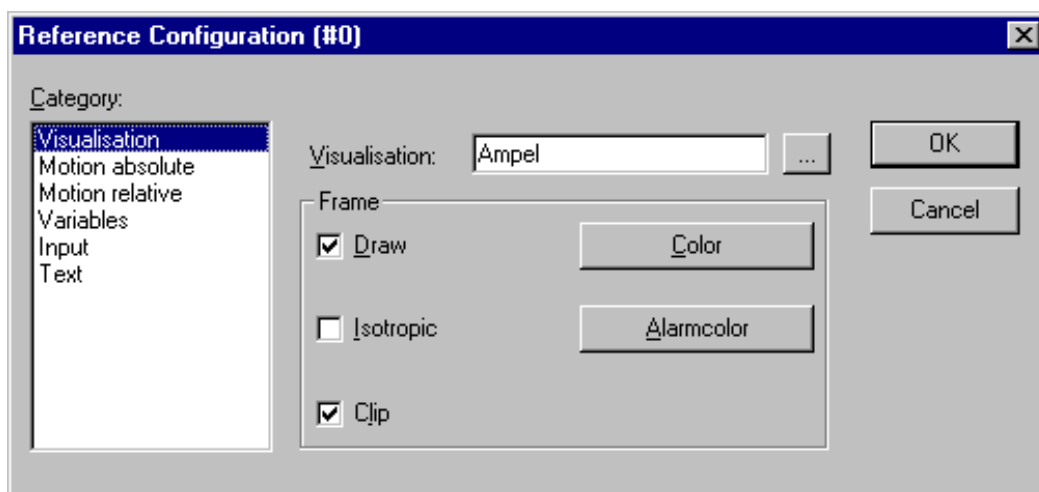


Figure 8\_10: Dialog for configuring visualization elements (Visualization category)

---

## **Other Functions for Visualization Elements**

### **'Extras' 'Send to Front'**

This command brings the selected visualization elements into the foreground.

### **'Extras' 'Send to Back'**

This command places the selected visualization elements in the background.

### **'Extras' 'Select Background Bitmap'**

This command opens the dialog for selecting files. Select a file with the extension "\*.bmp". The selected bitmap appears in the background of your visualization.

The bitmap can be deleted using the command '**Extras' 'Delete background bitmap'**'.

### **'Extras' 'Clear Background Bitmap'**

This command deletes the bitmap in the background of the current visualization.

You can use the command '**Extras' 'Select background bitmap'**' to select a bitmap for the current visualization.

### **'Extras' 'Align'**

This command allows you to align several selected visualization elements.

The following alignment options are available:

- Left: the left edges of all elements are aligned with the leftmost element
- the same applies to Right / Top / Bottom
- Horizontal center: the center points of all elements are aligned with the horizontal center
- Vertical centre: the center points of all elements are aligned with the vertical center

### **'Extras' 'Select All'**

This command selects all visualization elements in the current visualization object.

**'Extras' 'Elementlist'**

This command opens a dialog listing all visualization elements, together with their **number**, **type** and **position**. The position relates to the X and Y positions of the top left and bottom right corners of the element.

When you select one or more entries, the corresponding elements in the visualization are selected for optical checking and if necessary, the display is scrolled to the section in which the elements are found.

The **To front** button brings selected visualization elements into the foreground. The **To behind** button places them in the background.

The **Delete** button deletes the selected visualization elements.

**Undo** and **Redo** can be used to cancel or restore the changes made as in **'Edit' 'Undo'** and **'Edit' 'Redo'**. The processes can be monitored in the dialog.

To exit the dialog, confirm your changes using **OK**.

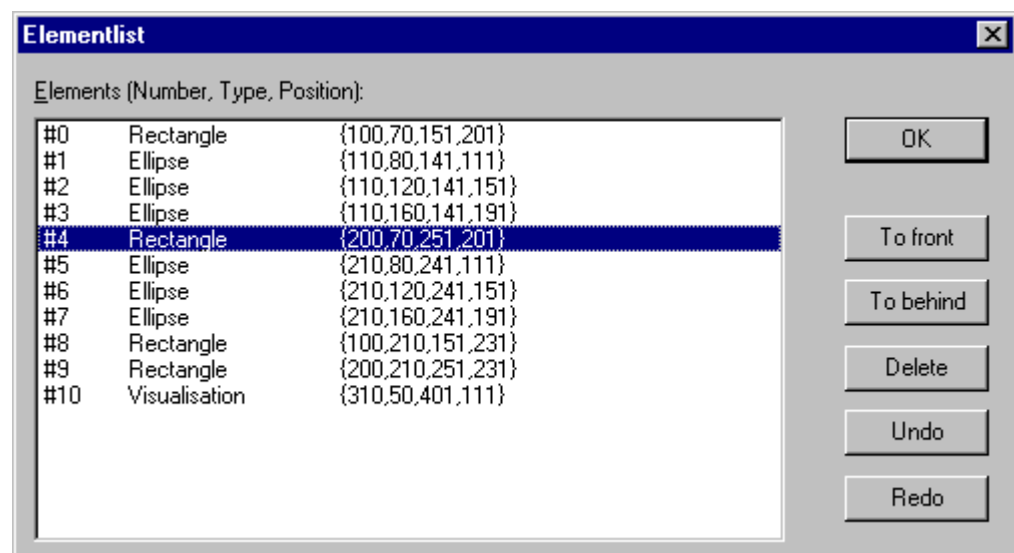


Figure 8\_11: Dialog for elementlist

**'Extras' 'Settings'**

This command allows you to make various settings concerning the visualizations in the dialog that opens.

In the **Grid** category, you can define whether the grid dots are to be **Visible** in offline mode. The distance between visible dots is at least 10, even if the size specified is smaller. In this case, only the grid dots appear at a distance in a multiple of the specified size. If the item **Active** is selected, the elements are snapped to the grid dots when drawing and moving. The distance between the grid dots is specified in the **Size** category.

In the **Zoom** field of the **Display** category, specify a number between 10 and 500% to increase or decrease the display of the visualization. If **Elementnumbers** is selected, the numbers of the elements in each visualization element are displayed in offline mode.

**Auto-Scrolling** in the **Frame** category means that when drawing or moving a visualization element, the visible area of the visualization window moves automatically, even if the window frame is reached. If **Best fit in online mode** is selected, the entire visualization with all elements is displayed in the window in online mode, regardless of how big the window is. If **Include background bitmap** is also selected, the bitmap is included in the adaptation calculation. Otherwise, only the elements are considered.

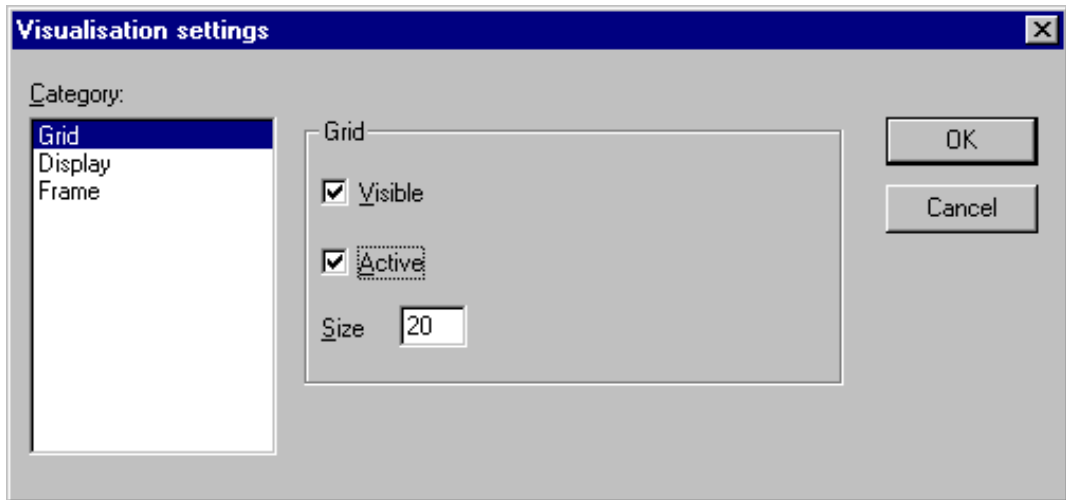


Figure 8\_12: Settings dialog for visualizations (Grid category)

## DDE Interface

### DDE Interface

**CP1131** has a DDE (dynamic data exchange) interface for reading data. This allows it to make available the contents of control variables and IEC addresses of other applications which also have DDE interfaces.

### Activating the DDE Interface

The DDE interface is activated as soon as you Login to the controller (or the simulation).

### General Accessing of Data

A DDE request has three parts:

1. the name of the program (in this case CP1131),
2. the file name and
3. the variable name to be read.

Name of the program: **CP1131**

File name: Full path of the project from which you are to read  
(C:\BERGHOF\CANtro\CP1131\_V20\SAMPLE\PROLIGHTS.pro).

Variable name: The name of a variable as it is specified in the watch and receipt manager.

### Which Variables can be Read?

All addresses and variables can be read. Variables and addresses are entered in the same way as they are entered in the watch and receipt manager.

Examples:

%IX0.1	(* reads the input 0.1 *)
PLC_PRG.TEST	(* reads the variable TEST of the block PLC_PRG *)
.GlobVar1	(* reads the global variable GlobVar1 *)

### Linking Variables to WORD

To obtain the current value of the TEST variable from the block PLC\_PRG in Microsoft WORD by means of the DDE interface, any field must be entered in WORD ('Insert' 'Field'), such as the date. If you now click on the field using the right-hand mouse button and select the command 'Display field function', you can change the field function to the required text. In our example, the result would look like this:

```
{DDEAUTO CP1131_V20 "C:\Berghof\CANtro\CP1131_V20\Sample\ProLights.pro" "PLC_PRG.i1Lightscircuit.ilights.Lamp_red"}
```

Click again on the field using the right-hand mouse button and select the command "Update field". The required variable contents appear in the text.

### **Linking Variables to EXCEL**

To allocate a variable to a cell in Microsoft EXCEL, the following must be entered in EXCEL:

```
=CP1131_V20|'C:\Berghof\CANtrol\CP1131_V20\Sample\ProLights.pro'  
  PLC_PRG.i1Lightscircuit.ilights.Lamp_red
```

In 'Edit' 'Links', the following is entered for this link:

```
Type:          CP1131_V20  
Source file:    C:\Berghof\CANtrol\CP1131_V20\Sample\ProLights.pro  
Element: PLC_PRG.i1Lightscircuit.ilights.Lamp_red
```

### **Addressing Variables using InTouch**

For your project, define a DDE access name <AccessName> with the application name CP1131\_V20 and the DDE topic name C:\Berghof\CANtrol\CP1131\_V20\Sample\ProLights.pro.

You can now define variables of the type DDE with the access name <AccessName>. The name of the variable should be entered again as the item name (e.g. PLC\_PRG.i1Lightscircuit.ilights.Lamp\_red).

## Appendix A: Keyboard Operation

### Operation

In order to operate CP1131 using only the keyboard, you require certain commands that cannot be found in the menu.

- With the function key <F6>, you can switch between the declaration part and the instruction part within an opened block.
- With <Alt>+<F6>, you can switch from an open object to the Object Organizer, and from there to the message window, if it is open. If a Find dialog is open, you can use <Alt>+<F6> to switch from the Object Organizer to the Find dialog and from there to the object.
- With the <Tab> key, you can jump between the input fields and the buttons in the dialogs.
- With the arrow keys, you can move through the tabs and the objects within the Object Organizer and the Library Manager.

All other actions can be initiated using the menu commands or the shortcuts displayed after the menu commands. <Shift>+<F10> displays the context-sensitive menu containing the most frequently-used commands for a selected object or for the active editor.

### Key Combinations

The following is an overview of all key combinations and function keys:

<b>General operation</b>	
Switch between the declaration part and the instruction part of a block	<F6>
Switch between the Object Organizer, the object and the message window	<Alt>+<F6>
Context-sensitive menu	<Shift>+<F10>
Shortcut mode for declarations	<Ctrl>+<Enter> key
Switch from the message in the message window to the position in the editor	<Enter> key
Open and close multi-level variables	<Enter> key
Open and close folders	<Enter> key
Change tabs in the Object Organizer and the Library Manager	<Arrow> keys
Jump to the next item in dialogs	<Tab>
Context-sensitive help	<F1>

<b>General commands</b>	
'File' 'Save'	<Ctrl>+<S>
'File' 'Print'	<Ctrl>+<P>
'File' 'Exit'	<Alt>+<F4>
'Project' 'Delete object'	<Del>
'Project' 'Insert object'	<Insert>
'Project' 'Rename object'	<Space bar>
'Project' 'Edit object'	<Enter> key
'Edit' 'Undo'	<Ctrl>+<Z>
'Edit' 'Redo'	<Ctrl>+<Y>
'Edit' 'Cut'	<Ctrl>+<X> or <Shift>+<Del>
'Edit' 'Copy'	<Ctrl>+<C>
'Edit' 'Paste'	<Ctrl>+<V>
'Edit' 'Delete'	<Del>
'Edit' 'Find next'	<F3>
'Edit' 'Input help'	<F2>
'Edit' 'Next error'	<F4>
'Edit' 'Previous error'	<Shift>+<F4>
'Online' 'Start'	<F5>
'Online' 'Breakpoint on/off'	<F9>
'Online' 'Single step over'	<F10>
'Online' 'Single step to'	<F8>
'Online' 'Single cycle'	<Ctrl>+<F5>
'Online' 'Write values'	<Ctrl>+<F7>
'Online' 'Force values'	<F7>
'Online' 'End forcing'	<Shift>+<F7>
'Window' 'Messages'	<Shift>+<Esc>

<b>FBD editor commands</b>	
'Insert' 'Network (after)'	<Shift>+<T>
'Insert' 'Assignment'	<Ctrl>+<A>
'Insert' 'Jump'	<Ctrl>+<L>
'Insert' 'Return'	<Ctrl>+<R>
'Insert' 'Operator'	<Ctrl>+<O>
'Insert' 'Function'	<Ctrl>+<F>
'Insert' 'Function block'	<Ctrl>+<B>
'Insert' 'Input'	<Ctrl>+<U>
'Extras' 'Negation'	<Ctrl>+<N>
'Extras' 'Zoom'	<Alt>+<Enter> key
<b>LD editor commands</b>	
'Insert' 'Network (after)'	<Shift>+<T>
'Insert' 'Contact'	<Ctrl>+<O>
'Insert' 'Parallel contact'	<Ctrl>+<R>
'Insert' 'Function block'	<Ctrl>+<B>
'Insert' 'Coil'	<Ctrl>+<L>
'Extras' 'Insert under'	<Ctrl>+<U>
'Extras' 'Negation'	<Ctrl>+<N>
<b>SFC editor commands</b>	
'Insert' 'Step transition (before)'	<Ctrl>+<T>
'Insert' 'Step transition (after)'	<Ctrl>+<E>
'Insert' 'Alternative branch (right)'	<Ctrl>+<A>
'Insert' 'Parallel branch (right)'	<Ctrl>+<L>
'Insert' 'Jump' (SFC)	<Ctrl>+<U>
'Extras' 'Zoom action/transition'	<Alt>+<Enter> key
Switch from SFC overview back to editor	<Enter> key

<b>Operating the control configuration</b>	
Open and close organisation elements	<Enter> key
Set edit frame around the name	<Space bar>
'Extras' 'Edit entry'	<Enter> key
<b>Operating the task configuration</b>	
Set edit frame around the task or program name	<Space bar>

## Appendix B: The Data Types

### Standard Data Types

#### Data Types

Users can avail of standard data types and self-defined data types for programming. Each identifier is assigned a data type, which defines how much memory is reserved and which values correspond to the memory contents.

#### BOOL

Variables of the type **BOOL** can assume the truth values TRUE and FALSE. 8 bits of memory are reserved.

#### Integer Data Types

Integer data types include **BYTE**, **WORD**, **DWORD**, **SINT**, **USINT**, **INT**, **UINT**, **DINT**, **UDINT**.

The various number types cover different number ranges. The following limits apply to integer data types:

Type	Lower limit	Upper limit	Memory
BYTE	0	255	8 bits
WORD	0	65535	16 bits
DWORD	0	4294967295	32 bits
SINT:	-128	127	8 bits
USINT:	0	255	8 bits
INT:	-32768	32767	16 bits
UINT:	0	65535	16 bits
DINT:	-2147483648	2147483647	32 bits
UDINT:	0	4294967295	32 bits

This means that information may be lost during type conversion from large to small types.

#### REAL / LREAL

**REAL** and **LREAL** are known as floating point types. They are required when using rational numbers. The memory reserved is 32 bits for REAL and 64 bits for LREAL.

## **STRING**

A variable of the type **STRING** can contain any character string. The size specification for reservation of memory in the declaration relates to characters, and can be made in round or square brackets. If no size is specified, a default size of 80 characters is assumed.

Example of a string declaration:

```
str:STRING(35):='This is a string';
```

## **Time Data Types**

The data types **TIME**, **TIME\_OF\_DAY** (abbreviated to **TOD**), **DATE** and **DATE\_AND\_TIME** (abbreviated to **DT**) are handled internally in the same way as **DWORD**.

In **TIME** and **TOD**, the time is specified in milliseconds, and in **TOD**, it is calculated from 00:00.

In **DATE** and **DT**, the time is specified in seconds, calculated from 1 January 1970 at 00:00.

The time data format for assigning values is described in the section entitled '**Constants**'.

---

## Defined Data Types

### Array

One, two and three-dimensional fields (arrays) of elementary data types are supported. Arrays can be defined in the declaration part of a block and in the global variable lists.

Syntax:

```
<field_name>:ARRAY [<ll1>..

```

ll1 and ll2 specify the lower limit of the field range, while ul1 and ul2 specify the upper limit. The limit values must be integers.

Example:

```
Card game : ARRAY [1..13, 1..4] OF INT;
```

Initialisation of arrays:

Either all elements of an array are initialised or none are initialised.

Examples of initialisations of arrays:

```
arr1 : ARRAY [1..5] OF INT := 1,2,3,4,5;
arr2 : ARRAY [1..2,3..4] OF INT := 1,3(7);           (* short for 1,7,7,7 *)
arr3 : ARRAY [1..2,2..3,3..4] OF INT := 2(0),4(4),2,3; (* short for 0,0,4,4,4,4,2,3 *)
```

Components of arrays are obtained in a two-dimensional field with the following syntax:

```
<field_name>[Index1,Index2]
```

Example:

```
Cardgame[9,2]
```

### Pointer

The addresses of variables or function blocks for the runtime of a program are stored in pointers.

Pointer declarations have the following syntax:

```
<identifier>: POINTER OF <data type/function block>;
```

A pointer can indicate any data type or function block, including self-defined ones.

The address of a variable or function block is assigned to the pointer with the address operator ADR.



A pointer is dereferenced using the contents operator “^” after the pointer identifier.

Example:

```
pt:POINTER TO INT;
var_int1:INT := 5;
var_int2:INT;
pt := ADR(var_int1);
var_int2:= pt^; (* var_int2 is now 5 *)
```

## Enumeration Type

An enumeration type is a self-defined data type consisting of a quantity of string constants. These constants are called enumeration values.

The enumeration values are known throughout the project, even if they have been declared locally in a block. It is best to create your enumeration types as  objects in the Object Organizer under the tab  **Data types**. They begin with the keyword TYPE and end with END\_TYPE.

Syntax:

```
TYPE <identifier>:(<Enum_0> ,<Enum_1>, ...,<Enum_n>);
END_TYPE
```

The <identifier> can assume one of the enumeration values and is initialised with the first one. The values are compatible with integers, i.e. they can be used to execute operations as with INT. A number x can be assigned to the <identifier>. If the enumeration values are not initialised, counting begins at 0. When initialising, remember that the initial values ascend. The validity of the number is checked for the runtime.

Example:


```
LIGHTS: (red, amber, green:=10); (*red has the initial value 0, amber 1, green 10 *)
LIGHTS:=0; (* lights has the value red*)
FOR i:= red TO green DO
  i := i + 1;
END_FOR;
```

The same enumeration value cannot be used twice.

Example:

```
LIGHTS: (red, amber, green);
COLOR: (blue, white, red);
Error: red cannot be used for LIGHTS and COLOR.
```

## Structures

Structures are stored as objects in the Object Organizer under the tab  **Data types**. They begin with the keyword TYPE and end with END\_TYPE.

Structure declarations have the following syntax:

```
TYPE <structure_name>:
STRUCT
  <variable_declaration 1>
  .
  .
  <variable_declaration n>
END_STRUCT
END_TYPE
```

<Structure name> is now a type known throughout the project, and can be used in the same way as a standard data type.

Nested structures are permitted. The only limitation is that variables cannot be set to addresses (AT declaration is not permitted).

Example of a structure definition with the name 'Polygon':

```
TYPE Polygon:
STRUCT
  Start:ARRAY [1..2] OF INT;
  Point1:ARRAY [1..2] OF INT;
  Point2:ARRAY [1..2] OF INT;
  Point3:ARRAY [1..2] OF INT;
  Point4:ARRAY [1..2] OF INT;
  End:ARRAY [1..2] OF INT;
END_STRUCT
END_TYPE
```

Components of structures are accessed using the following syntax:


```
<structure_name>.<component_name>
```

If, for example, we have a structure with the name "week", which contains a component with the name "Monday", we can obtain the component as follows:

```
Week.Monday
```

## References

The self-defined data type '**Reference**' is used to generate an alternative name for a variable, constant or function block.

Create your references as objects in the Object Organizer under the tab  **Data types**. They begin with the keyword TYPE and end with END\_TYPE.

Syntax:

```
TYPE <identifier>: <assignment expression>;  
END_TYPE
```

Example:

```
TYPE message:STRING[50];  
END_TYPE;
```

## Appendix C: The IEC Operators

### The IEC Operators

CP1131 supports all IEC operators. Unlike the standard functions, these are known implicitly throughout the project. In the block implementation, operators are used in the same way as functions. The following is a list of all operators supported.

### Arithmetic Operators

#### ADD

Addition of variables of the type BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL or LREAL.

Two TIME variables can also be added. The total is also a time (e.g. t#45s + t#50s = t#1m35s is valid).

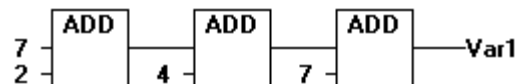
Example in IL:

```
LD 7
ADD 2,4,7
ST var1
```

Example in ST:

```
var1 := 7+2+4+7;
```

Example in FBD:



#### MUL

Multiplication of variables of the type BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL or LREAL.

Example in IL:

```
LD 7
MUL 2,4,7
ST var1
```

Example in ST:

```
var1 := 7*2*4*7;
```

Example in FBD:



**SUB**

Subtraction of a variable of the type BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL or LREAL from another variable of one of these types.

A TIME variable can also be subtracted from another TIME variable. The result is another TIME variable. Remember that negative TIME variables are not defined.

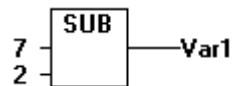
Example in IL:

```
LD 7
SUB 8
ST var1
```

Example in ST:

```
var1 := 7-2;
```

Example in FBD:

**DIV**

Division of a variable of the type BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL or LREAL by another variable of one of these types.

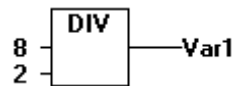
Example in IL:

```
LD 8
DIV 2
ST var1
```

Example in ST:

```
var1 := 8/2;
```

Example in FBD:

**MOD**

Modulo division of a variable of the type BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL or LREAL by another variable of one of these types. This function supplies the integer remainder of the division as the result.

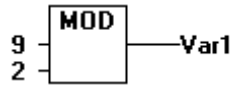
Example in IL:

```
LD 9
MOD 2
ST var1      (* result is 1 *)
```

Example in ST:

```
var1 := 9 MOD 2;
```

Example in FBD:



## INDEXOF

This function supplies the internal index of a block as the result.

Example in ST:

```
var1 := INDEXOF(block2);
```

## SIZEOF

This function supplies the number of bytes required by the specified data type as the result.

Example in IL:

```
arr1:ARRAY[0..4] OF INT;
var1:=INT;
LD arr1
SIZEOF
ST var1 (* result is 10 *)
```

Example in ST:

```
pt := ADR(pt^)^ + SIZEOF(INT);
```

---

## Bitstring Operators

### AND

Bitwise AND of bit operands. The operands should be of the type BOOL, BYTE, WORD or DWORD.

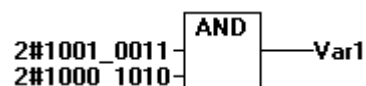
Example in IL:

```
var1 :BYTE;
LD 2#1001_0011
AND 2#1000_1010
ST var1 (* result is 2#1000_0010 *)
```

Example in ST:

```
var1 := 2#10 01_0011 AND 2#1000_1010
```

Example in FBD:



**OR**

Bitwise OR of bit operands. The operands should be of the type BOOL, BYTE, WORD or DWORD.

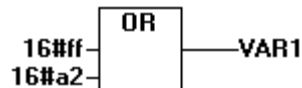
Example in IL:

```
var1 :BYTE;
LD 2#1001_0011
OR 2#1000_1010
ST var1 (* result is 2#1001_1011 *)
```

Example in ST:

```
Var1 := 2#1001_0011 OR 2#1000_1010
```

Example in FBD:

**XOR**

Bitwise XOR of bit operands. The operands should be of the type BOOL, BYTE, WORD or DWORD.

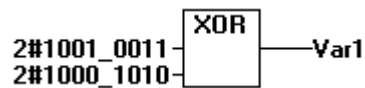
Example in IL:

```
Var1 :BYTE;
LD 2#1001_0011
XOR 2#1000_1010
ST Var1 (* result is 2#0001_1001 *)
```

Example in ST:

```
Var1 := 2#1001_0011 XOR 2#1000_1010
```

Example in FBD:

**NOT**

Bitwise NOT of a bit operand. The operand should be of the type BOOL, BYTE, WORD or DWORD.

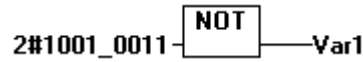
Example in IL:

```
Var1 :BYTE;
LD 2#1001_0011
NOT
ST Var1 (* result is 2#0110_1100 *)
```

Example in ST:

```
Var1 := NOT 2#1001_0011
```

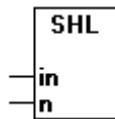
Example in FBD:



## Bit Shift Operators

The following operators are represented with a diagram as FBD operators.

### SHL



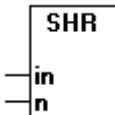
Bitwise left-shift of an operand: A:= SHL (IN, N)

A, IN and N should be of the type BYTE, WORD or DWORD. IN is moved to the left by N bits, and filled in from the right with zeroes.

Example:

```
LD 1
SHL 1
STVar1 (* result is 2 *)
```

### SHR



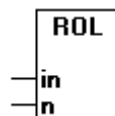
Bitwise right-shift of an operand: A:= SHR (IN, N)

A, IN and N should be of the type BYTE, WORD or DWORD. IN is moved to the right by N bits, and filled in from the left with zeroes.

Example:

```
LD 32
SHL 2
STVar1 (* result is 8 *)
```

### ROL



Bitwise left-rotation of an operand: A:= ROL (IN, N)

A, IN and N should be of the type BYTE, WORD or DWORD. IN is moved by one bit position to the left N times, with the leftmost bit being re-inserted from the right.

Example:

```
Var1 :BYTE;
LD 2#1001_0011
ROL 3
ST Var1 (* result is 2#1001_1100 *)
```

## ROR



Bitwise right-rotation of an operand: A:= ROR (IN, N)

A, IN and N should be of the type BYTE, WORD or DWORD. IN is moved by one bit position to the right N times, with the rightmost bit being re-inserted from the left.

Example:

```
Var1 :BYTE;
LD 2#1001_0011
ROR 3
ST Var1 (* result is 2#0111_0010 *)
```

---

## Selection Operators

All selection operations can be executed on variables. For the sake of clarity, we will limit the following examples to constants as operators.

### SEL

Binary selection.

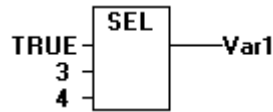
```
OUT := SEL(G, IN0, IN1) means:
OUT := IN0 if G=FALSE;
OUT := IN1 if G=TRUE.
```

IN0, IN1 and OUT can have any type, while G must be of the type BOOL. The result of the selection is IN0, if G is FALSE, and IN1 if G is TRUE.

Example in IL:

```
LD TRUE
SEL 3,4
ST Var1 (* result is 4 *)
LD FALSE
SEL 3,4
ST Var1 (* result is 3 *)
```

Example in FBD:



**MAX**

Maximum function: supplies the greater of two values.

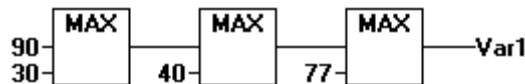
OUT := MAX(IN0, IN1)

IN0, IN1 and OUT can be of any type.

Example in IL:

```
LD 90
MAX 30
MAX 40
MAX 77
ST Var1 (* result is 90 *)
```

Example in FBD:



**MIN**

Minimum function. Supplies the smaller of two values.

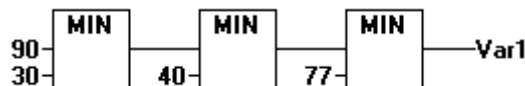
OUT := MIN(IN0, IN1)

IN0, IN1 and OUT can be of any type.

Example in IL:

```
LD 90
MIN 30
MIN 40
MIN 77
ST Var1 (* result is 30 *)
```

Example in FBD:



**LIMIT**

Limiting

OUT := LIMIT(Min, IN, Max) means:

OUT := MIN (MAX (IN, Min), Max)

Max is the upper limit and Min is the lower limit for the result. If the value IN exceeds the upper limit Max, LIMIT supplies Max. If IN falls below Min, the result is Min.

IN and OUT can be of any type.

Example in IL:

```
LD 90
LIMIT 30,80
ST Var1 (* result is 80 *)
```

## MUX

Multiplexer

OUT := MUX(K, IN0,...,INn) means:

OUT := IN<sub>K</sub>.

IN0, ...,INn and OUT can be of any type. K must be of the type BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT or UDINT. MUX selects the Kth value from a quantity of values.

Example in IL:

```
LD 0
MUX 30,40,50,60,70,80
ST Var1 (* result is 30 *)
```

---

## Comparison Operators

### GT

Greater than

A Boolean operator with the result TRUE if the first operand is greater than the second. The operands can be of the type BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, TIME, DATE, TIME\_OF\_DAY, DATE\_AND\_TIME or STRING.

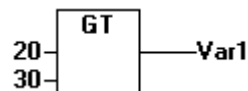
Example in IL:

```
LD 20
GT 30
ST Var1 (* result is FALSE *)
```

Example in ST:

```
VAR1 := 20 > 30 > 40 > 50 > 60 > 70;
```

Example in FBD:



**LT**

Less than

A Boolean operator with the result TRUE if the first operand is less than the second. The operands can be of the type BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, TIME, DATE, TIME\_OF\_DAY, DATE\_AND\_TIME or STRING.

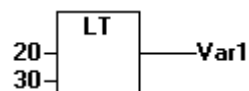
Example in IL:

```
LD 20
LT 30
ST Var1 (* result is TRUE *)
```

Example in ST:

```
VAR1 := 20 < 30;
```

Example in FBD:

**LE**

Less than or equal to

A Boolean operator with the result TRUE if the first operand is less than or equal to the second operand. The operands can be of the type BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, TIME, DATE, TIME\_OF\_DAY, DATE\_AND\_TIME or STRING.

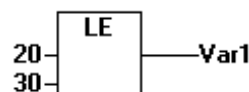
Example in IL:

```
LD 20
LE 30
ST Var1 (* result is TRUE *)
```

Example in ST:

```
VAR1 := 20 <= 30;
```

Example in FBD:

**GE**

Greater than or equal to

A Boolean operator with the result TRUE, if the first operand is greater than or equal to the second operand. The operands can be of the type BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, TIME, DATE, TIME\_OF\_DAY, DATE\_AND\_TIME or STRING.

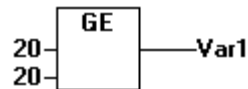
Example in IL:

```
LD 60
GE 40
ST Var1 (* result is TRUE *)
```

Example in ST:

```
VAR1 := 60 >= 40;
```

Example in FBD:



## **EQ**

Equality

A Boolean operator with the result TRUE, if the operands are equal. The operands can be of the type BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, TIME, DATE, TIME\_OF\_DAY, DATE\_AND\_TIME or STRING.

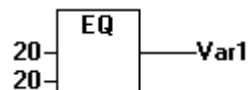
Example in IL:

```
LD 40
EQ 40
ST Var1 (* result is TRUE *)
```

Example in ST:

```
VAR1 := 40 = 40;
```

Example in FBD:



## **NE**

Inequality

A Boolean operator with the result TRUE, if the operands are not equal. The operands can be of the type BOOL, BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL, LREAL, TIME, DATE, TIME\_OF\_DAY, DATE\_AND\_TIME or STRING.

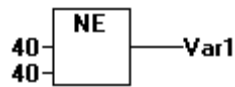
Example in IL:

```
LD 40
NE 40
ST Var1 (* result is FALSE *)
```

Example in ST:

```
VAR1 := 40 <> 40;
```

Example in FBD:




---

## Address Operators

### ADR

Address function

ADR supplies the address of its argument in a DWORD. This address can be sent to manufacturer functions and handled there in the same way as a pointer or assigned to a pointer within the project.

Example in IL:

```
LD var1
ADR
ST var2
man_fun1
```

### Content Operator

A pointer is dereferenced using the content operator “^” after the pointer identifier.

Example in ST:

```
pt:POINTER TO INT;
var_int1:INT;
var_int2:INT;
pt := ADR(var_int1);
var_int2:=pt^;
```

---

## Call Operator

### CAL

Call of a function block.

The instance of a function block is called in IL using CAL. The assignments of the input variables of the function block follow the name of the function block instance, in round brackets.

Example: Call of the instance *Inst* of a function block with assignment of the input variables *Par1* and *Par2* to 0 or TRUE.

```
CAL INST(PAR1 := 0, PAR2 := TRUE)
```

Blank page

## Appendix D: Elements in the Standard Library

### Type Conversion Functions

It is not permitted to convert implicitly from a “large” type to a “small” type (e.g. from INT to BYTE or from DINT to WORD). If you wish to do this, you must use special type conversion functions. Generally speaking, it is possible to convert from any elementary type to any other elementary type.

Syntax:

```
<elem.type1>_TO_<elem.type2>
```

#### **BOOL TO Conversions**

Conversion from the type BOOL to any other type:

In the case of number types, the result is 1 if the operand is TRUE and 0 if the operand is FALSE.

With the type STRING, the result is 'TRUE' or 'FALSE'.

Examples in ST:

```
i:=BOOL_TO_INT(TRUE);           (* result is 1 *)
str:=BOOL_TO_STRING(TRUE);      (* result is 'TRUE' *)
t:=BOOL_TO_TIME(TRUE);          (* result is T#1ms *)
tof:=BOOL_TO_TOD(TRUE);         (* result is TOD#00:00:00.001 *)
dat:=BOOL_TO_DATE(FALSE);       (* result is D#1970-01-01 *)
dandt:=BOOL_TO_DT(TRUE);        (* result is DT#1970-01-01-00:00:01 *)
```

#### **TO BOOL Conversions**

Conversion from a type to the type BOOL:

The result is TRUE if the operand is not equal to 0. The result is FALSE if the operand is equal to 0.

For the type STRING, the result is TRUE if the operand is 'TRUE'. Otherwise, the result is FALSE.

Examples in ST:

```
b := BYTE_TO_BOOL(2#11010101);  (* result is TRUE *)
b := INT_TO_BOOL(0);             (* result is FALSE *)
b := TIME_TO_BOOL(T#5ms);       (* result is TRUE *)
b := STRING_TO_BOOL('TRUE');    (* result is TRUE *)
```

## **Conversions between Integer Number Types**

Conversion from one integer number type to another number type:

Information may be lost during type conversion from large to small types. If the number to be converted exceeds the range limit, the first bytes in the number are ignored.

Example in ST:

```
si := INT_TO_SINT(4223); (* result is 127 *)
```

If you save the integer number 4223 (16#107f in hexadecimal display) in a SINT variable, then this contains the number 127 (16#7f in hexadecimal display).

Example in IL:

```
LD 2
INT_TO_REAL
MUL 3.5
```

## **REAL TO/LREAL TO Conversions**

Conversion from the type REAL or LREAL to another type:

The number is rounded up to or down to the nearest integer value and converted to the relevant type. Exceptions are the types STRING, BOOL, REAL and LREAL.

Information may be lost during type conversion from large to small types.

Example in ST:

```
i := REAL_TO_INT(1.5); (* result is 2 *)
```

```
j := REAL_TO_INT(1.4); (* result is 1 *)
```

Example in IL:

```
LD 2.7
REAL_TO_INT
GE %MW8
```

## **TIME TO/TIME OF DAY Conversions**

Conversion from the type TIME or TIME\_OF\_DAY to another type:

The time is saved internally in a DWORD in milliseconds (for TIME\_OF\_DAY from 00:00). This value is converted.

Information may be lost during type conversion from large to small types.

For the type STRING, the result is the time constant.

Examples in ST:

```
str :=TIME_TO_STRING(T#12ms);           (* result is 'T#12ms' *)
dw:=TIME_TO_DWORD(T#5m);               (* result is 300000 *)
si:=TOD_TO_SINT(TOD#00:00:00.012);    (* result is 12 *)
```

### **DATE TO/DT TO Conversions**

Conversion from the type DATE or DATE\_AND\_TIME to another type:

The date is saved internally in a DWORD in seconds from 1 January 1970. This value is converted.

Information may be lost during type conversion from large to small types.

For the type STRING the result is the date constant.

Examples in ST:

```
b :=DATE_TO_BOOL(D#1970-01-01);        (* result is FALSE *)
i :=DATE_TO_INT(D#1970-01-15);         (* result is 29952 *)
byt :=DT_TO_BYTE(DT#1970-01-15-05:05:05); (* result is 129 *)
str:=DT_TO_STRING(DT#1998-02-13-14:20); (* result is
'DT#1998-02-13-14:20' *)
```

### **STRING TO Conversions**

Conversion from the type STRING to another type:

The operand of the type STRING must have a valid value to the target type. Otherwise, the result is 0.

Examples in ST:

```
b :=STRING_TO_BOOL('TRUE');           (* result is TRUE *)
w :=STRING_TO_WORD('abc34');          (* result is 0 *)
t :=STRING_TO_TIME('T#127ms');        (* result is T#127ms *)
```

### **TRUNC**

Conversion from the type REAL to the type INT. Only the amount of the integer portion of the number is taken.

Information may be lost during type conversion from large to small types.

Examples in ST:

```
i:=TRUNC(1.9); (* result is 1 *)
i:=TRUNC(-1.4); (* result is 1 *)
```

Example in IL:

```
LD 2.7
TRUNC
GE %MW8
```

---

## **Numeric Functions**

### **ABS**

Supplies the absolute value of a number. ABS(-2) supplies 2.

### **SQRT**

Supplies the square root of a number.

### **LN**

Supplies the natural logarithm of a number.

### **LOG**

Supplies the logarithm of the base 10 of a number.

### **EXP**

Supplies the exponential function of a number.

### **SIN**

Supplies the sine of a number.

### **COS**

Supplies the cosine of a number.

### **TAN**

Supplies the tangent of a number.

### **ASIN**

Supplies the arc sine (inverse function of sine) of a number.

### **ACOS**

Supplies the arc cosine (inverse function of cosine) of a number.

### **ATAN**

Supplies the arc tangent (inverse function of tangent) of a number.

**EXPT**

Exponentiation of one variable by another:

$$\text{OUT} = \text{IN1}^{\text{IN2}}$$

OUT; IN1 and IN2 can be of the type BYTE, WORD, DWORD, INT, DINT or REAL.

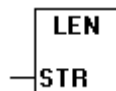
Example in IL:

```
LD 7
EXPT 2
ST var1 (* result is 49 *)
```

Example in ST:

```
var1 := 7 EXPT 2;
```

---

**String Functions****LEN**

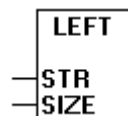
Outputs the length of a string.

Example in IL:

```
LD 'SUSI'
LEN
ST Var1 (* result is 4 *)
```

Example in ST:

```
Var1 := LEN ('SUSI');
```

**LEFT**

Left supplies a left initial string of a string.

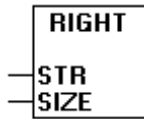
LEFT (STR, SIZE) means: take the first SIZE character from the left in the STR string.

Example in IL:

```
LD 'SUSI'
LEFT 3
ST Var1 (* result is 'SUS' *)
```

Example in ST:

```
Var1 := LEFT ('SUSI',3);
```

**RIGHT**

Right supplies a right initial string of a string.

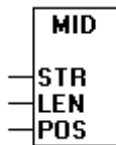
RIGHT (STR, SIZE) means: take the first SIZE character from the left in the STR string.

Example in IL:

```
LD 'SUSI'
RIGHT 3
ST Var1 (* result is 'USI' *)
```

Example in ST:

```
Var1 := RIGHT ('SUSI',3);
```

**MID**

Mid supplies a substring of a string.

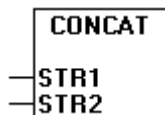
MID (STR, LEN, POS) means: retrieve the LEN character from the STR string, beginning with the character at the position POS.

Example in IL:

```
LD 'SUSI'
MID 2,2
ST Var1 (* result is 'US' *)
```

Example in ST:

```
Var1 := MID ('SUSI',2,2);
```

**CONCAT**

Concatenation (joining) of two strings.

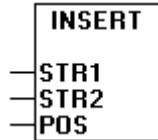
Example in IL:

```
LD 'SUSI'
CONCAT 'WILLI'
ST Var1 (* result is 'SUSIWILLI' *)
```

Example in ST:

```
Var1 := CONCAT ('SUSI','WILLI');
```

**INSERT**



INSERT inserts another string from a certain position in a string.

INSERT(STR1, STR2, POS) means: insert STR2 into STR1 after POSth position.

Example in IL:

```
LD 'SUSI'
INSERT 'XY',2
ST Var1 (* result is 'SUXYSI' *)
```

Example in ST:

```
Var1 := INSERT ('SUSI','XY',2);
```

**DELETE**



DELETE deletes a substring from a certain position in a string.

DELETE(STR, L, P) means: delete character L from STR, beginning with the Pth.

Example in IL:

```
LD ' SUXYSI'
DELETE 2,2
ST Var1 (* result is 'SUSI' *)
```

Example in ST:

```
Var1 := DELETE ('SUXYSI',2,2);
```

**REPLACE**



REPLACE replaces a substring of a string with another string.

REPLACE(STR1, STR2, L, P) means: replace the character L from STR1 with STR2, beginning with the Pth character.

Example in IL:

```
LD 'SUXYSI'
REPLACE 'K',2,2
ST Var1 (* result is 'SKYSI' *)
```

Example in ST:

```
Var1 := REPLACE ('SUXYSI','K',2,2);
```

## FIND



FIND locates a substring in a string.

FIND(STR1, STR2) means: Find the position of the first character of the first occurrence of STR2 in STR1. If STR2 does not occur in STR1, then OUT := 0 applies.

Example in IL:

```
LD 'SUXYSI'
FIND 'XY'
ST Var1 (* result is 3 *)
```

Example in ST:

```
Var1 := FIND ('SUXYSI','XY');
```

---

## Bistable Function Blocks

### SR



Set bistable function block dominantly:

Q1 = SR (SET1, RESET) means:

$$Q1 = (\text{NOT RESET AND } Q1) \text{ OR SET1}$$

Q1, SET1 and RESET are of the type BOOL.

### RS



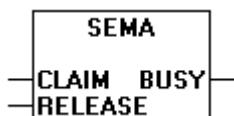
Reset bistable function block:

$Q1 = RS(\text{SET}, \text{RESET1})$  means:

$Q1 = \text{NOT RESET1 AND } (Q1 \text{ OR SET})$

$Q1$ , SET and RESET1 are of the type BOOL.

## SEMA



A software semaphore (interruptible).

$\text{BUSY} = \text{SEMA}(\text{CLAIM}, \text{RELEASE})$  means:

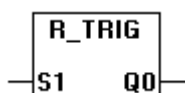
```
BUSY := X;
IF CLAIM THEN X:=TRUE;
ELSIF RELEASE THEN BUSY := FALSE; X:= FALSE;
END_IF
```

X is an internal BOOL variable, which is initialised with FALSE. BUSY, CLAIM and RELEASE are of the type BOOL.

If SEMA is called and BUSY is TRUE, this means that SEMA has already been pre-assigned (SEMA was called with CLAIM = TRUE). If BUSY is FALSE, SEMA has not yet been called, or it has been released (call with RELEASE = TRUE).

## Edge Detection

### R\_TRIG

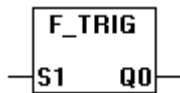


Detector for a rising edge.

```
FUNCTION_BLOCK R_TRIG
VAR_INPUT
    S1 : BOOL;
END_VAR
VAR_OUTPUT
    Q0 : BOOL;
END_VAR
VAR
    M : BOOL := FALSE;
END_VAR
    Q0 := S1 AND NOT M;
    M := S1;
END_FUNCTION_BLOCK
```

As long as the input variable S1 supplies FALSE, the output will be Q0 and the dummy variable M will be FALSE. As soon as S1 supplies TRUE, Q0 will first supply TRUE and then M is set to TRUE. This means that for all subsequent calls of the function, Q0 will continue to supply FALSE until S1 has a falling edge and then a rising edge again.

## F\_TRIG



Detector for a falling edge.

```

FUNCTION_BLOCK F_TRIG
VAR_INPUT
    S1: BOOL;
END_VAR
VAR_OUTPUT
    Q0: BOOL;
END_VAR
VAR
    M: BOOL := FALSE;
END_VAR
    Q0 := NOT S1 AND NOT M;
    M := NOT S1;
END_FUNCTION_BLOCK

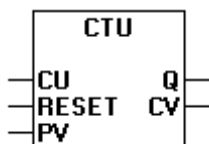
```

As long as the input variable S1 supplies TRUE, the output will be Q0 and the dummy variable M will be FALSE. As soon as S1 supplies FALSE, Q0 will first supply TRUE and then M is set to TRUE. This means that for all subsequent calls of the function, Q0 will continue to supply FALSE until S1 has a rising edge and then a falling edge again.

---

## Counters

### CTU



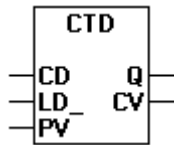
Up-counter:

CU, RESET and Q are of the type BOOL, while PV and CV are of the type INT.

If RESET is TRUE, the count variable CV is initialised with 0. If CU has a rising edge from FALSE to TRUE, the function block CV will be increased by 1, for as long as CV remains smaller than PV (i.e. if no overflow is caused).

Q supplies TRUE if CV is greater than or equal to the upper limit PV.

**CTD**



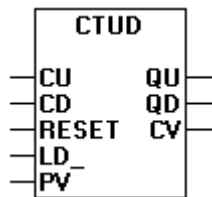
Down-counter:

CD, LD\_ and Q are of the type BOOL, while PV and CV are of the type INT. (LD\_ means 'not LD', as otherwise it would be a keyword).

If LD\_ is TRUE, the count variable CV is initialised with the upper limit PV. If CD has a rising edge from FALSE to TRUE, CV is reduced by 1, as long as CV is greater than 0 (if no underflow is caused).

Q supplies TRUE if CV is less than or equal to 0.

**CTUD**



Up- and down-counter.

CU, CD, RESET, LD\_, QU and QD are of the type BOOL, while PV and CV are of the type INT.

If RESET applies, the count variable CV is initialised with 0. If LD\_ applies, CV is initialised with PV.

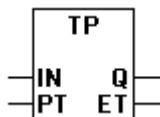
If CU has a rising edge from FALSE to TRUE, CV is increased by 1 for as long as CV does not cause an overflow. If CD has a rising edge from FALSE to TRUE, CV is reduced by 1 for as long as CV does not cause an overflow.

QU supplies TRUE if CV is greater than or equal to PV.

QD supplies TRUE if CV is less than or equal to 0.

## Timer

### TP



Pulse generator.

TP(IN, PT, Q, ET) means:

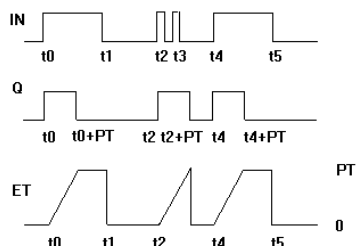
IN and PT are input variables of the type BOOL or TIME. Q and ET are output variables of the type BOOL or TIME. If IN is FALSE, the outputs are FALSE or 0.

As soon as IN is TRUE, the time in milliseconds is incremented until the value is equal to that in PT, and then it remains the same.

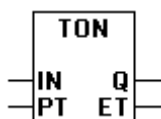
Q is TRUE if IN is TRUE and ET is less than or equal to PT. Otherwise it is FALSE.

Q then supplies a signal for the time specified in PT.

Graphical display of the time sequence of TP:



### TON



Timer on delay.

TON(IN, PT, Q, ET) means:

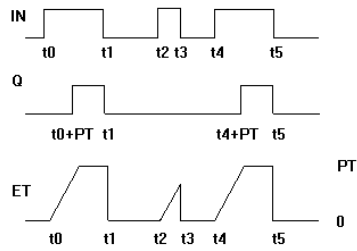
IN and PT are input variables of the type BOOL or TIME. Q and ET are output variables of the type BOOL or TIME. If IN is FALSE, the outputs are FALSE or 0.

As soon as IN is TRUE, the time in milliseconds is incremented in ET until the value is equal to that in PT, and then it remains equal.

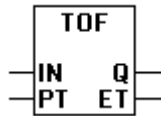
Q is TRUE if IN is TRUE and ET is equal to PT. Otherwise, it is FALSE.

Q then has a rising edge when the time specified in PT in milliseconds has elapsed.

Graphical display of the time behaviour of TON:



**TOF**



Timer off delay.

TOF(IN, PT, Q, ET) means:

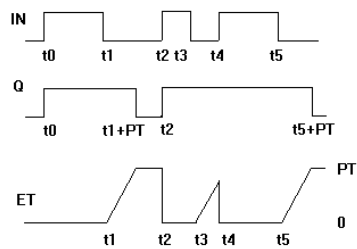
IN and PT are input variables of the type BOOL or TIME. Q and ET are output variables of the type BOOL or TIME. If IN is TRUE, the outputs are TRUE or 0.

As soon as IN is FALSE, the time in milliseconds is incremented in ET until the value is equal to that in PT, and then it remains equal.

Q is FALSE if IN is FALSE and ET is equal to PT. Otherwise, it is TRUE.

Q then has a falling edge when the time specified in PT in milliseconds has elapsed.

Graphical display of the time response of TOF:



## Appendix E: Operands in CP1131

### Operands

Constants, variables, addresses and possibly function calls may occur as operands in CP1131.

### Constants

#### BOOL Constants

BOOL constants are the truth values TRUE and FALSE.

#### TIME Constants

TIME constants can be declared in **CP1131**. These are used in particular to operate the timer in the standard library. A TIME constant consists of a leading "t" or "T" (or "time" or "TIME" in their complete forms) and a hash symbol "#".

These are followed by the actual time declaration, which can consist of days ("d"), hours ("h"), minutes ("m"), seconds ("s") and milliseconds ("ms"). Time specifications must be entered in order of size (d before h before m before s before ms).

Examples of correct TIME constants in an ST instruction:

TIME1 := T#14ms;

TIME1 := T#100S12ms; (\*overrun in the highest component is permitted\*)

TIME1 := t#12h34m15s;

The following are incorrect:

TIME1 := t#5m68s; (\*overrun in a lower level\*)

TIME1 := 15ms; (\*T# missing\*)

TIME1 := t#4ms13d; (\*incorrect sequence of time specifications\*)

#### DATE Constants

This type allows you to make date specifications. A DATE constant is declared with a leading "d", "D", "DATE" or "date" followed by "#".

Any date can then be entered in the sequence day-month-year.

Examples:

DATE#1996-05-06

d#1972-03-29

### TIME OF DAY Constants

This type allows you to save times. A TIME\_OF\_DAY declaration begins with "tod#", "TOD#", "TIME\_OF\_DAY#" or "time\_of\_day#", followed by a time entered as follows: hour:minute:second. Seconds can be specified as real numbers, meaning that fractions of seconds can also be used.

Examples:

```
TIME_OF_DAY#15:36:30.123
tod#00:00:00
```

### DATE AND TIME Constants

Date constants and times can also be combined in DATE\_AND\_TIME constants. DATE\_AND\_TIME begin with "dt#", "DT#", "DATE\_AND\_TIME#" or "date\_and\_time#". The date specification is followed by a dash and then the time.

Examples:

```
DATE_AND_TIME#1996-05-06-15:36:30
dt#1972-03-29-00:00:00
```

### Number Constants

Number values can occur as binary numbers, octal numbers, decimal numbers or hexadecimal numbers. If an integer value is not a decimal number, its base, followed by a hash (#), must be inserted before the integer constant. The digit values for the numbers 10 to 15 as hexadecimal numbers are specified in the usual way with the letters A-F.

Underscores are permitted within a number value.

Example:

```
14          (decimal number)
2#1001_0011 (binary number)
8#67        (octal number)
16#A        (hexadecimal number)
```

These number values can be of the type BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL or LREAL.

Implicit conversions from "larger" to "smaller" types are not permitted. This means that a DINT variable cannot be used as an INT variable without any further action. The type conversion functions in standard.lib are used for this purpose (see the section entitled '**Type conversions**' in the appendix).

**REAL/LREAL Constants**

REAL and LREAL constants can be specified as decimal fractions and in exponential notation.

Example:

7.4  
1.64e+009

**STRING Constants**

A string is any sequence of characters. STRING constants are enclosed in single inverted commas. Spaces can also be entered, and are treated in the same way as all other characters.

In character strings, the combination of the dollar sign (\$) followed by two hexadecimal digits are interpreted as the hexadecimal representation of the eight-bit character code. Furthermore, when they occur in a character string, combinations of two characters beginning with a dollar sign are interpreted as follows:

\$\$	Dollar sign
\$'	Inverted comma
\$L or \$l	Line indent
\$N or \$n	New line
\$P or \$p	Page break
\$R or \$r	Line break
\$T or \$t	Tab

Examples:

'w1Wüß?'  
'Mary and Peter'  
':-)'

**Variables**

Variables are declared either locally in the declaration part of a block or in the global variable lists.

It is important to remember that variable identifiers must not contain spaces, they must not be declared twice, and they must not be the same as keywords. Variables are not case-sensitive, which means that VAR1, Var1 and var1 all signify the same variable. Underscores are significant in variables, e.g. "A\_BCD" and "AB\_CD" are interpreted as different identifiers. A sequence of underscores at the start of an identifier or within an identifier are not permitted. The first 32 characters are significant.

Variables can be used wherever they are permitted by the declared type. The variables available can be called using the input help.

## **Access to Variables of Arrays, Structures and Blocks**

Components of two-dimensional arrays are accessed using the following syntax:

<fieldname>[Index1, Index2]

Variables of structures are accessed using the following syntax:

<structurename>.<variablename>

Variables of function blocks and programs are accessed using the following syntax:

<blockname>.<variablename>

---

## **Addresses**

### **Address**

Individual memory cells can be displayed directly using special character strings. These consist of the concatenation of the per cent character "%", an area prefix, a prefix for the size and one or more natural numbers separated by spaces.

The following area prefixes are supported:

I	Input
Q	Output
M	Marker

The following size prefixes are supported:

X	Single bit
None	Single bit
B	Byte (8 bits)
W	Word (16 bits)
D	Double word (32 bits)

Examples:

%QX0.0	Output bit 0 of DIO Bank 0
%IW4	Input word 4
%QB7	Output byte 7
%MD48	Double word at memory position 48 in the marker

The validity of the address depends on the program's current controller configuration.

## **Markers**

All supported sizes can be used to access the marker.

For example, the address %MD48 would address byte numbers 192, 193, 194 and 195 in the marker area ( $48 * 4 = 192$ ). The first byte is the byte number 0.

It is also possible to access words, bytes and even bits. For example, %MX5.0 accesses the first bit in the fifth word (bits are usually stored word-by-word).

4 kilobytes of marker memory is provided in CP1131 (MB0 ... MB4095). Accesses outside this area are not permitted.

---

## **Functions**

In the ST, a function call can also occur as an operand.

Example:

```
Result := Fct(7) + 3;
```

## Appendix F: Translation Errors

### Error Messages

Here you will find the error messages displayed by the parser (*italics*), and their possible causes listed in alphabetical order:

*“A bit address may only have BOOL variables”*

Change the declaration type to BOOL, or change the address to another format.

*“A comma is not permitted after ’”*

Delete the comma.

*“ADR requires a variable and not an expression or constant as a parameter”*

Replace the expression or constant with a variable.

*“A function block call must begin with the name of an instance”*

Enter the name of the required instance, or undo the call of the function block.

*“A function name is not permitted here”*

Replace the function call with a variable or a constant.

*“A jump must have exactly one label”*

Change the jump destination to a defined label.

*“A maximum of 4 address fields are permitted”*

Delete the superfluous address fields.

*“An address must appear after ‘AT’”*

Enter a valid address after the keyword AT, or change the keyword AT.

*“An array of arrays may not be set at an address”*

Undo the address assignment.

*“An array of strings may not be set at an address”*

Undo the address assignment.

*“An expression is required”*

An expression must be entered in this position.

*“A number is required after ‘+’ or ‘-’”*

Change the word after the + or - to a valid constant.

*“A number is required after ‘,’”*

Delete the comma, or add another number.

*“A structure variable must appear before ‘.’”*

The identifier to the left of ‘.’ is not a structure variable or an instance of a function block. Change the identifier to a structure variable or to an instance of a function block, or delete the stop and the identifier to its right.

*“At least one instruction is required”*

Enter an instruction.

*“Block has incorrect ending: add ST or delete last expression.”*

The block ends with an incomplete expression. Complete the expression or delete it.

*“Block <name> is not in the project”*

Define a block with the name <name> using the menu commands **‘Project’ ‘Add Object’** or change <name> to the name of a defined block.

*“Block <name> requires exactly <number> inputs”*

Check the number of input variables required by this block and remove the superfluous ones or add any extra variables that may be required.

*“CAL, CALC or CALN require a function block instance as operand”*

Declare an instance of the function block that you want to call.

*“Closing bracket without corresponding opening bracket”*

Delete the closing bracket or add an opening bracket.

*“Comment in IL only permitted at line end”*

Move the comment to the end of the line.

*"<component> is not a component of <variable>"*

If the variable is a structure, change <component> to one of the components declared in this structure.

If the variable is an instance of a function block, change <component> to an input or output parameter that is declared in this function block.

*"END\_STRUCT or identifier required"*

A structure definition must end with the keyword END\_STRUCT.

*"END\_VAR or identifier required"*

Enter a valid identifier or END\_VAR at the beginning of the declaration line.

*"EXIT is only permitted within a loop"*

Delete EXIT.

*"Function <name> has too few parameters"*

Add the missing parameters.

*"Function <name> has too many parameters"*

Delete the superfluous parameters.

*"<identifier> is not a function"*

Change <identifier> to one of the functions from the libraries linked to the project, or to a function declared in the project.

*"Identifier required"*

Enter a valid identifier at the beginning of the declaration line.

*"IF and ELSIF require a Boolean expression as a condition"*

Change the expression to an expression with a result of type BOOL.

*"Illegal time constants"*

Check that you have entered a correct time constant, and rectify any errors. The following errors might occur:

t or # missing at the start.

A time specification appears twice (e.g.: t#4d2d).

The times appear in the wrong order.

Incorrect identification of time unit (no d, h, m, s or ms).

*"IL operator required"*

Change the first word in the line to a valid operator or a valid function.

*"Incompatible types: <Type1>  
cannot be converted to <Type2>"*

Check the required operator types (look for the operator in your Help file), and change the type of the variable that caused the error to a permitted type, or choose another variable.

*"Incorrect initial value"*

Enter a constant (constant) corresponding to the type in the declaration as lower range value.

*"[<index>]' is only permitted for array variables"*

Declare the identifier before the bracket as an array, or change it to a declared array variable.

*"INI operator requires a function block  
instance or a data block instance"*

Change the operand to the instance of a function block by declaring the operand as a function block or by using a function block that has already been declared, or use the instance of a data block.

*"Integer or symbolic constant required"*

Only integers or symbolic constants are permitted as the condition of a CASE event. Change the incorrect condition.

*"Invalid address: <Address>"*

Check your controller configuration to see which addresses are permitted, and replace the addresses with permitted addresses, or change the controller configuration.

*"Invalid characters follow the valid step name:  
'<name>'"*

Delete the superfluous characters.

*"Invalid type for parameter <name> of  
<FBName>:  
<Type1> cannot be converted to <Type2>"*

Use a variable of type <Type2> for the assignment to the parameter <name> or change the type of the assigned variable to <Type1>.

*"Invalid type for the input parameter <name>:  
<Type1> cannot be converted to <Type2>"*

A value with the invalid type <Type2> is assigned to the variable <name>. Change the variable or the constant to a variable or constant with the type <Type1>.

*“Invalid type in parameter <Parameter> of <block>:  
<Type1> cannot be converted to <Type2>”*

Check the required type of the parameter <Parameter> for the block <block>. Change the type of the variable that provoked the error to <Type2>, or select another variable with <Type2>.

*“Jump at non-defined step: <name>”*

Change <name> to the name of an existing step, or add a step with name <name>.

*“Jump label <label > is not defined”*

Define a label with the name <LabelName> or change <LabelName> to a defined label.

*“Jump/return is only permitted on the right-hand side of a network”*

Delete the jump/return that is not permitted in this position.

*“Jump/return requires a boolean input”*

The result of the previous instructions is not of the type BOOL. Add an operator or a function with a result of type BOOL.

*“Keywords must be entered in upper case”*

Change the case of the keyword.

*“LD required”*

Only the instruction “LD” is permitted in this line.

*“Multi-dimensional arrays may not be set at addresses”*

Undo the address assignment.

*“Multiple definition of the jump label <label>”*

Delete one of the defined jump labels.

*“Multiple underscores in the identifier”*

Delete one of the underscores in the identifier.

*“<Name> is not a function block”*

Replace <name> with a valid name of a function block.

*"<Name> is not an input parameter of the called function block"*

Check the input variables of the called function block and change <name> to one of these variables.

*"<Name> must be a declared instance of the function block <FBName>"*

Change the text of the instance of the function block <name> (initialised with "Instance") to an identifier of a valid instance declaration of a function block.

*"'N' modifier requires an operand of type BOOL"*

Delete the N and explicitly negate the operand with the NOT operator.

*"No instance was specified for the call of the function block <name>"*

Change the text of the instance of the function block <name> (initialised with "Instance") to the identifier of a valid instance declaration.

*"No jump labels between bracketed expressions"*

Delete the label or the brackets.

*"Non-permissible characters follow a valid watch expression"*

Delete the superfluous characters.

*"No \*.obj can be found"*

Turn on simulation mode.

*"Not enough memory"*

Save to quit the system. Quit Windows, restart and attempt the translation again.

*"NOT requires a Boolean operand"*

Change the operand to an operand of type BOOL.

*"No write access to <name>"*

Change <name> to a variable with write permission.

*"Number, ELSE or END\_CASE required"*

A CASE statement was concluded incorrectly. Enter the keyword END\_CASE.

*"<Number> operands are too few for <operator>. At least <number > are required"*

Check how many operands the operator <operator> requires and add the missing operands.

*"<Number> operands are too many for <operator>. Exactly <number> are required"*

Check how many operands the operator <operator> requires and delete the superfluous operands.

*"Only VAR and VAR\_GLOBAL may be set at addresses"*

Copy the declaration to a VAR or VAR\_GLOBAL area.

*"Operand required"*

Add another operand.

*"Operands of ST, STN, S, R must be variables with write access"*

Replace the first operand with a variable with write permission.

*"<Operator> in brackets is not permissible"*

This operator is not permitted between brackets. Either delete the brackets or the operator.

*"Operator is not extensible.  
Delete superfluous operands"*

Check the number of operands for this operator and delete superfluous operands.

*"Parameter <Number> is not of the correct type"*

Check the type of the operand with regard to the number <number> of this operator, this function, or this function block. Change the type of the variables that provoked the error to a permitted type, or choose another variable with permitted type.

*"Several declarations with the same identifier <name>"*

Rename one of the identifiers with the same name.

*"Step names are repeated: '<name>'"*

Change one of the names.

*"<string> is not an operator"*

Change <string> to a valid operator.

*"The block name in the declaration name does not match that in the object list"*

Give your block a new name using the menu command **'Project' 'Rename Object'**, or change the block's name in its declaration part. The name must appear directly after the key words PROGRAM, FUNCTION or FUNCTIONBLOCK.

*"The counter in the FOR instruction is not a variable with write access"*

Change the variable to a variable with write permission.

*"The counter of the FOR instruction must be of type INT"*

Change the variable to a variable of type INT.

*"The expression before an operator has to produce a result of type BOOL"*

The result of the previous instruction is not of the type BOOL. Add an operator or a function with the result BOOL.

*"The expression in the index of an array has to have a result of type INT"*

Change the index to a constant or a variable of type INT.

*"The incrementation value of the FOR instruction must be of type INT"*

Change the variable to a variable of type INT.

*"The selector of the CASE instruction must be of type INT"*

Change the selector to a selector of type INT.

*"The starting value of the FOR instruction must be of type INT"*

Change the variable to a variable of type INT.

*"The step name is not a permissible identifier: '<name>'"*

Change the identifier <name> to a valid identifier.

*“The upper range limit of the FOR instruction must be of type INT”*

Change the variable to a variable of type INT.

*“Too few array indexes”*

Check the number of indexes (1, 2 or 3) for which the array was declared and add the missing indexes.

*“Too many array indexes”*

Check the number of indexes (1, 2 or 3) for which the array was declared and delete the superfluous ones.

*“Too many identifiers”*

You must remember to limit your number of identifiers as it is not possible to have more than 64,000 identifiers.

*“Type identifier required”*

Enter a valid type after the identifier in the declaration.

*“Unexpected end”*

In the declaration part: Enter the keyword END\_VAR at the end of the declaration part.

In the text editor: Enter instructions that will terminate the last instruction sequence (e.g. ST).

*“Unexpected end of bracketed expression”*

Enter a closing bracket.

*“Unknown type: <string>”*

Change <string> to a valid type.

*“Unknown variable or address”*

This watch variable is not declared in the project. Press <F2> for help on declared variables.

*“UNTIL requires a boolean expression as a condition”*

Change the expression to an expression with a result of type BOOL.

*“Variable <name> has not been declared”*

Declare this variable in the declaration part of the block or in the global variable list.

*“VAR, VAR\_INPUT, VAR\_OUTPUT  
or VAR\_INOUT required”*

The first line after the name of a block must contain one of these keywords.

*“WHILE requires a boolean  
expression as a condition”*

Change the expression to an expression with a result of type BOOL.