

**Introduction to Development Enviroment**

**CP1131**

**V.1.0**

**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

**CANtrol® //** is a registered trademark of Berghof Automationstechnik GmbH

#### **General Information on this Manual**

Content:

This manual describes the introduction to CP1131. The product-related information contained herein was up to date at the time of publication of this manual.

Completeness:

This manual is complete only in conjunction with the user manual entitled

‘Introduction  
to CANtrol Automation System’

and the product-related hardware or software user manuals required for the particular application.

Standards:

The CANtrol automation system, its components and its use are based on International Standard IEC 61131 Parts 1 to 4 (EN 61131 Parts 1 to 3 and Supplementary Sheet 1).  
Supplementary Sheet 1 of EN 61131 (IEC 61131-4) entitled ‘User Guidelines’ is of particular importance for the user.

Order numbers:

Please see the relevant product overview in the ‘Introduction to CANtrol Automation System’ manual for a list of available products and their order numbers.

Ident. No.: 2801020

You can reach us at our headquarters at:

Berghof 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 Automationstechnik GmbH is DIN EN ISO 9001 certified



blank page

**Contents**

**GENERAL INSTRUCTIONS ..... 7**

**Hazard Categories and Indications ..... 7**

**Qualified users ..... 7**

**Use as Prescribed..... 8**

**INTRODUCTION TO PROGRAMMING WITH CP1131 ..... 9**

**About this Manual..... 9**

**Hardware and Software Requirements..... 9**

**The CDIO 16/16-0,5 module ..... 10**

**Installation ..... 11**

    Installing the Software on the Programming Unit ..... 11

    Installing the CNW - CANtrol Node Wizard ..... 11

    Installing CP1131..... 11

    Hardware Installation ..... 12

    Connecting the CDIO 16/16-0,5 ..... 12

    Installing the Programming Unit ..... 12

    Peer-to-peer Connection ..... 12

    Configuring the Serial Interface ..... 13

    CNW: Configuring the Serial Interface ..... 13

    CP1131: Configuring the Serial Interface ..... 14

    Configuring the Node ID of the CANtrol Module ..... 15

    CNW: Configuring the Node ID ..... 15

    CP1131: Configuring the Node ID ..... 18

    Project-specific Configuration ..... 19

    Configuring CP1131 ..... 19

    Configuring Libraries..... 19

    Configuring the Editors ..... 19

    Configuring to Project-specific Hardware ..... 20

    Completing the Controller Configuration ..... 21

    Hardware Configuration of the CDIO 16/16-0,5 ..... 23

    Connecting the 'Traffic Light' LEDs ..... 23

    Assigning Digital Outputs to the Variable Names..... 23

**The PLC Sample Project under CP1131 ..... 24**

    POU Types in CP1131 ..... 24

    How Can the IEC 61131-3 Programming Languages be Used? ..... 25

    Example of the Use of LD..... 26

    Example of the Use of FBD ..... 27

Example of the Use of IL.....	28
Example of the Use of ST.....	29
Example of the Use of SFC.....	30
<b>The 'Lights.pro' Sample Project.....</b>	<b>32</b>
Starting the 'Lights.pro' Sample Project.....	32
The <i>TrafficLight</i> Function Block.....	34
The <i>TrafficLightPhase</i> Function Block.....	36
The <i>delay</i> Function Block.....	37
The <i>TrafficLightSwitch</i> Function Block.....	38
The <i>PLC_PRG</i> Program.....	40
Project Structure.....	41
<b>Starting the Sample Projects on the CDIO 16/16-0,5.....</b>	<b>43</b>
<b>DIGITAL INPUTS/OUTPUTS.....</b>	<b>47</b>
Grouping of Inputs/Outputs.....	47
Schematic Diagram of Input/Output Grouping.....	48
Without Grouping.....	49
<b>Digital Inputs, Positive-Switching.....</b>	<b>50</b>
Block diagram of input.....	50
Digital Inputs Data.....	51
<b>Digital Outputs, Positive-Switching.....</b>	<b>53</b>
Block diagram of output.....	53
Digital Outputs Data.....	54
Overload Reaction of Digital Outputs.....	55
<b>ANNEX.....</b>	<b>57</b>
<b>Environmental Protection.....</b>	<b>57</b>
Emissions.....	57
Disposal.....	57
<b>Maintenance / Upkeep.....</b>	<b>57</b>
<b>Repairs / Customer Service.....</b>	<b>57</b>
Warranty.....	57
<b>Nameplate.....</b>	<b>58</b>
<b>Addresses and Literature.....</b>	<b>59</b>
Addresses.....	59
Standards / Literature.....	59

## General Instructions

### Hazard Categories and Indications

The indications described below are used in connection with safety instructions you will need to observe for your own personal safety and the avoidance of damage to property.

These instructions are emphasised by bordering and/or shading and a bold-printed indication, their meaning being as follows:

<b>DANGER!</b>	means that death, severe physical injury or substantial damage to property <u>will occur</u> on failure to take the appropriate precautions.
----------------	--

<b>Warning !</b>	means that death, severe physical injury or substantial damage to property <u>may occur</u> on failure to take the appropriate precautions.
------------------	---

<b>Caution</b>	means that minor physical injury or damage to property may occur on failure to take the appropriate precautions.
----------------	--

**Note:**  
provides important information on the product or refers to a section of the documentation which is to be particularly noted.

### Qualified users

Qualified users within the meaning of the safety instructions in this documentation are trained specialists who are authorised to commission, earth and mark equipment, systems and circuits in accordance with safety engineering standards and who as project planners and designers are familiar with the safety concepts of automation engineering.

---

## **Use as Prescribed**

CANtrol is a modular automation system based on the CANbus, intended for industrial control applications within the medium to high performance range.

CANtrol is designed for use within Overvoltage Category I (IEC 364-4-443) for the controlling and regulating of machinery and industrial processes in low-voltage installations in which the rated supply voltage does not exceed 1,000 VAC (50/60 Hz) or 1,500 VDC.

Qualified project planning and design, proper transport, storage, installation, use and careful maintenance are essential to the flawless and safe operation of CANtrol.

CANtrol may only be used within the scope of the data and applications specified in the present documentation and associated user manuals.

### **CANtrol is to be used only as follows:**

- as prescribed,
- in technically flawless condition,
- without arbitrary or unauthorised changes and
- exclusively by qualified users

The regulations of the German professional and trade associations, the German technical supervisory board (TÜV), the VDE (Association of German electricians) or other corresponding national bodies are to be observed.

### **Safety-oriented (fail-safe) systems**

Particular measures are required in connection with the use of SPC in safety-oriented systems. If an SPC is to be used in a safety-oriented system, the user ought to seek the full advice of the SPC manufacturer in addition to observing any standards or guidelines on safety installations which may be available.

<b>Warning!</b>	As with any electronic control system, the failure of particular components may result in uncontrolled and/or unpredictable operation. All types of failure and the associated fuse systems are to be taken into account at system level. The advice of the SPC manufacturer should be sought if necessary
-----------------	--

## Introduction to Programming with CP1131

### About this Manual

<b>Contents</b>	The aim of this manual is to give users a simple introduction to the CANtrol automation system in conjunction with the CP1131 development environment. The hardware used is the CPU module CDIO 16/16-0,5. All the steps required to get started, such as module configuration, initial program download, etc., are described using the example of a simple PLC sample project.
<b>Implementation</b>	The PLC sample project is implemented using programming languages in accordance with standard IEC 61131-3. To familiarise users with CP1131, the example uses structural elements such as function blocks, data structures and enumeration types.
<b>Introduction</b>	Comprehensive information on the CANtrol automation system, such as correct use, operating conditions, system structure, etc. can be found in the user manual <i>'Introduction to CANtrol Automation System'</i>

### Hardware and Software Requirements

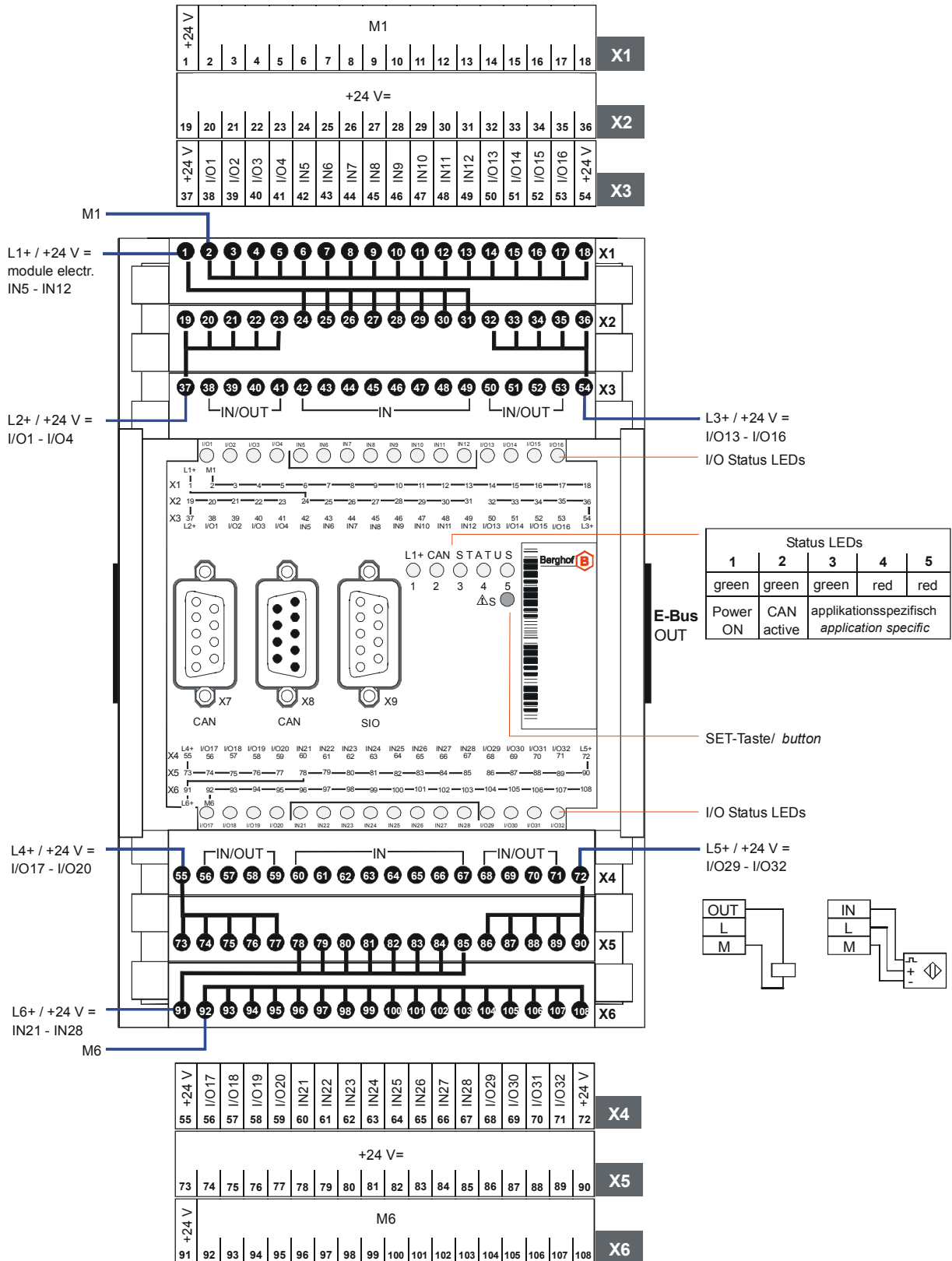
<b>Hardware</b>	<p>The following hardware is required to implement the PLC sample project:</p> <ul style="list-style-type: none"><li>• 1 power supply, rated voltage 24 VDC</li><li>• 1 CANtrol CDIO 16/16-0,5 module with current IEC 61131-3 firmware</li><li>• 1 serial programming cable (crossed RX/TX)</li><li>• 1 PC with Win95/WinNT operating system and free serial interface as the programming device (PADT)</li><li>• Optional: 2 x 3 LEDs in the traffic light colours red (2), amber (2) and green (2)</li></ul>
-----------------	---

**Note:**

Power must be supplied to the CANtrol automation system using **safety extra low voltage (SELV) in accordance with EN 6 1131-2.**

<b>Software</b>	<p>The software required for the sample project consists of the current versions of the following:</p> <ul style="list-style-type: none"><li>• CP1131</li><li>• CANtrol Node Wizard (CNW)</li><li>• 'Lights.pro' sample project (contained in the directory: ...\\SAMPLE of the CP1131 installation)</li></ul>
-----------------	--

The CDIO 16/16-0,5 module



2VF100004DG00.cdr

---

## Installation

### Installing the Software on the Programming Unit

Software used Before you can begin, the CANtrol software tools CNW and CP1131 must be installed on the programming unit/PADT (PC with Win95/WinNT).

### Installing the CNW - CANtrol Node Wizard

Installation Execute the program *SETUP.EXE* contained on diskette #1. Set the installation directory in the dialog that appears. You can use the default installation directory ...\*Berghof* \*CNW*, or specify your own.

CNW overview The CNW (CANtrol Node Wizard) can be used to configure CANtrol CPU and remote modules from the programming unit via a suitable programming cable. The following options are possible:

- CAN interface,
- serial interface,
- modem,
- Ethernet interface.

The CNW can also be used to configure the node numbers and the CAN baud rate of CAN channel 0 (X7/X8).

The CNW provides user guidance when downloading new firmware to CPU modules.

**Note:**

In this example, the CNW is only used to modify the node number of the CPU via the serial interface.

### Installing CP1131

Installation Execute the program *SETUP.EXE* contained on diskette #1. Set the installation directory in the dialog that appears. You can use the default installation directory ...\*Berghof* \*CP1131*, or specify your own.

CP1131 Overview The CP1131 development environment allows you to control and regulate simple and complex processes alike. You can test your program offline using the simulation environment integrated into CP1131. In a subsequent step, the PLC program created is executed on the target hardware. On completion, the project is compiled and prepared for the target processor. In this step, an automatic error detector integrated into CP1131 identifies potential errors. The compiler cannot detect logical programming errors. The user can identify these by using the 'Single Step' and 'Single Cycle' functions, for example. You also have the option of integrating both variable values ('forcing values') and new lines of code ('online changes') into the running project online without having to stop the program.

**Note:**

Only standard commands can be tested in simulation mode. The simulation does not support hardware components.

## Hardware Installation

Contents This section describes the electrical connection to the CANtrol CDIO 16/16-0,5 module and the correct configuration of the serial interface between the programming unit/PADT (PC) and CANtrol.

### Connecting the CDIO 16/16-0,5

Please read the section entitled 'Digital Inputs/Outputs' in this manual before attempting connection.

This section also specifies the designations and arrangement of the power supply connection terminals.

**Note:**

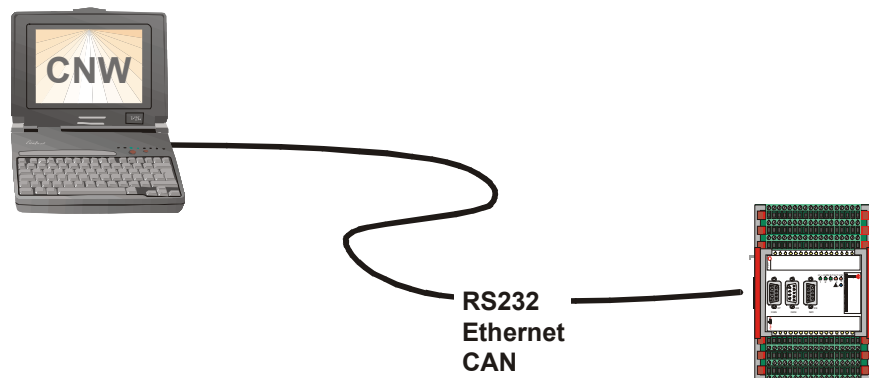
Power must be supplied to the CANtrol automation system using **safety extra low voltage (SELV) in accordance with EN 6 1131-2.**

### Installing the Programming Unit

Serial interface To install the PC, simply configure the serial interface on a free COM port correctly. A peer-to-peer connection between this COM port of the PC and the serial connection (X9) of the CANtrol module is then established via the programming cable. The COM port interface used must be reported to the CNW and CP1131.

### Peer-to-peer Connection

**Konfiguration von Node ID und Baudrate mit Peer-to-Peer Verbindung**  
**Configuration of the node-ID and baudrate with peer-to-peer connection**



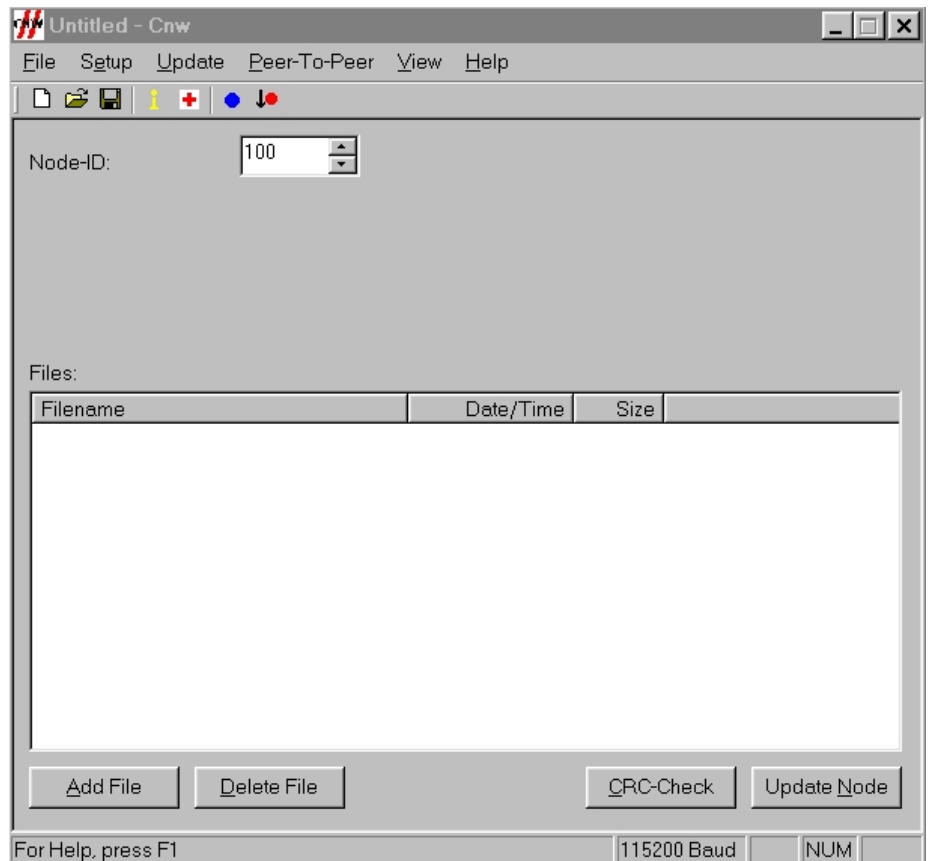
2VF100023DG00.cdr

## Configuring the Serial Interface

### CNW: Configuring the Serial Interface

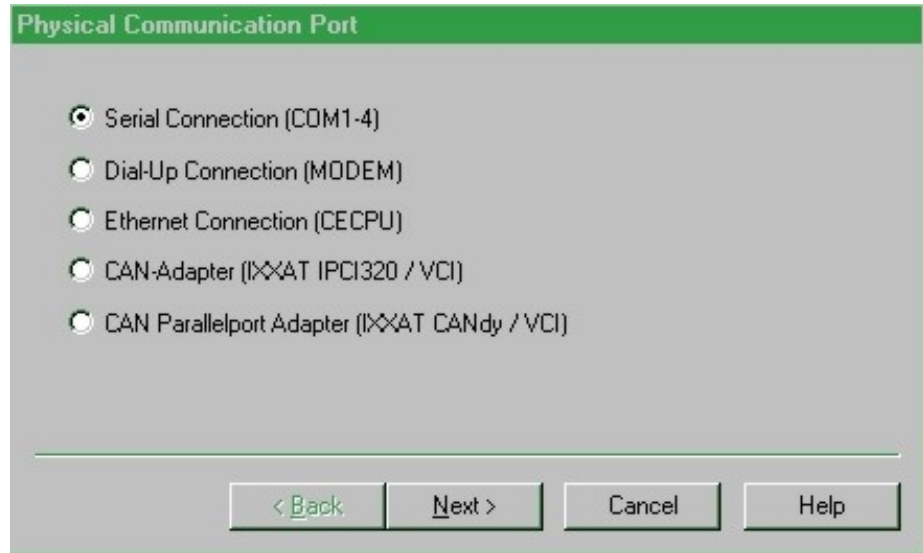
Set up COM port      To configure the serial interface correctly:

- Start the CNW by double-clicking its icon.  
The following window appears.



Initial CNW window

- Open the *Setup* menu and select *Physical Port*.  
*Select Serial Connection (COM1-4)*.



CNW serial connection

- Click *Next*.  
The four specified COM interfaces are now displayed.  
Make your selection by clicking one of these four interfaces.
- Activate your selection by clicking *Finish*.

### **CP1131: Configuring the Serial Interface**

Set up COM port

To configure the serial interface correctly:

- Start CP1131 by double-clicking its icon
- Open the Online menu and select *Communication Options*.

The dialog box that is displayed is the same as the window in the CNW.  
Please ensure that you select the same COM interface as in the CNW.

## Configuring the Node ID of the CANtrol Module

### CNW: Configuring the Node ID

**Preparation** Ensure that no other program accesses the reserved CANtrol programming interface while you are working with the CNW.

**Note:**

Quit all active programs in the programming interface before attempting to use the CNW.

**Node number** The CNW can be used to configure the node numbers (node ID) of CANtrol CPU and remote modules. This is absolutely essential to ensure uninterrupted intercommunication, because in a closed CAN network, node numbers can occur only once. The node numbers can also be used to create a modular structure for the CAN network.

CANtrol is designed for 'bridge operation'. This means that spatially distributed CANtrol control cells connected to each other via CAN channel 0 can be programmed from any location. The relevant module is addressed exclusively via its unique node number. The module to be programmed does not have to be identical to the module to which the programming unit (PADT) is connected. The bridge functions without any user intervention.

**Module identification** Module identification is performed as follows:

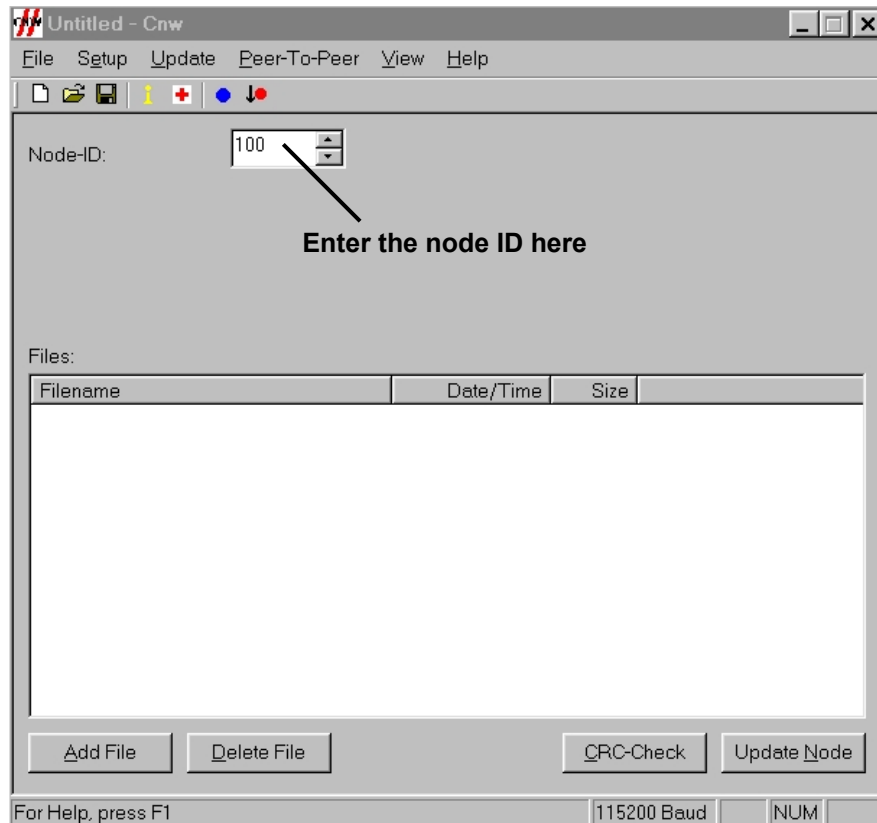
- Start the CNW by double-clicking its icon. The initial CNW window appears.
- Select *Identify Node* in the *Peer-To-Peer* menu. The *Node-Information* window is displayed.

Node identification of a CANtrol module

The current data from the module that is linked directly with the programming unit via the cable connection (hence 'peer-to-peer') is displayed.

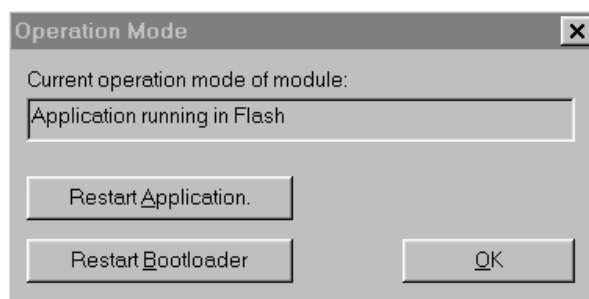
## Define target module

Using the node number, you define the 'target module' (cell controller) for communication between the programming unit and the CANtrol system.  
To activate the specified module as the target module, enter the node ID displayed in the previous window.



Entering the node ID

- Select the menu *Update/Firmware Operation Mode*.  
The following window is displayed:

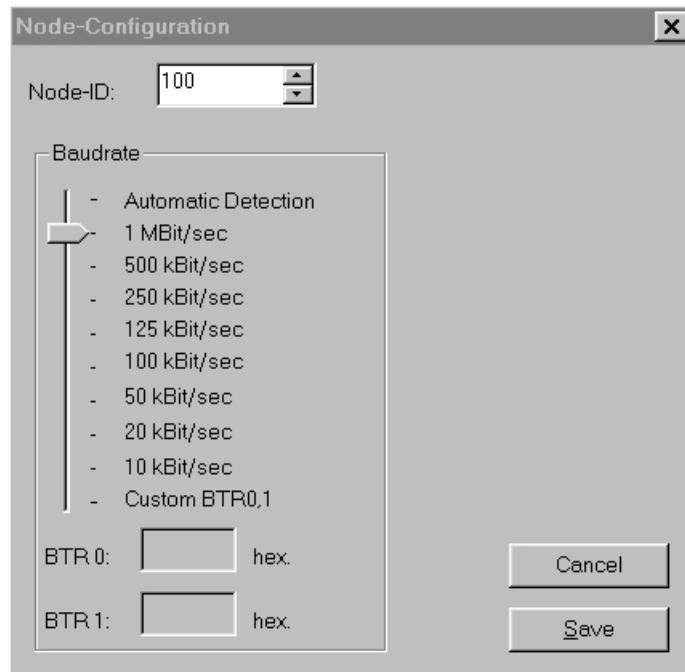


Select Bootloader/Application menu

- Click *Restart Bootloader*. The two red LEDs on the front of the module (LEDs 4+5) must flash alternately (flash on/off ratio 1:1).  
The module can now be configured. This status is called 'bootloader'.
- Once you have booted the module, close the menu by clicking *OK*.

Configuration

Select Configure Node in the Peer-To-Peer menu. The following window is displayed:



Configuring a CANtrol module

You can now configure the peer-to-peer module.

- Enter the required node ID of the relevant peer-to-peer module and specify the baud rate of CAN channel 0 (X7/X8).

**Note:**

You can modify the specified node ID of the peer-to-peer module here as required.

Save

Click Save to transfer these settings to the module and store them in the flash memory there.

When you have successfully completed the configuration process, a message appears on screen asking you to restart the module. The changes made to the configuration only take effect after the module has been restarted.

Restart

To restart the module, select the menu Uppdate/Firmware Operation-Mode again. The menu described previously is displayed.

To implement the required changes, you must now click Restart Application. This exits 'bootloader' mode and the CP1131 runtime system, and the application (if available) is launched.

You can then exit the CNW.

**Note:**

The values specified previously take effect when you launch the application.

One consequence of this is that, immediately after the module is restarted, the node ID specified in the Node ID field is no longer correct. The CNW will therefore display question marks on-screen shortly afterwards to inform you that communication can no longer be established using the specified node ID. Confirm this message with OK.

**CP1131: Configuring the Node ID**

## Preparation

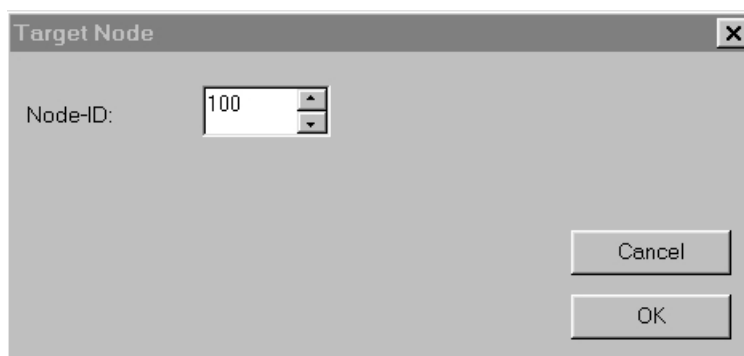
Ensure that the login mode is disabled and the development environment is offline. You can verify this as follows using the Onl~~ine~~ menu: If the menu item *Logout* is greyed out (cannot be selected), the development environment is offline. If not, you must select *Logout*. Alternatively, check the status bar at the bottom right of your screen; it contains the word *Online*. If *Online* is greyed out, you are offline.

## Set node ID

You can now set the node ID in CP1131. To do this, select the menu option *Set Node ID* from the Onl~~ine~~ menu.

**Note:**

You must enter the same node number here as was specified with the CNW.



Setting the node ID in CP1131

## Project-specific Configuration

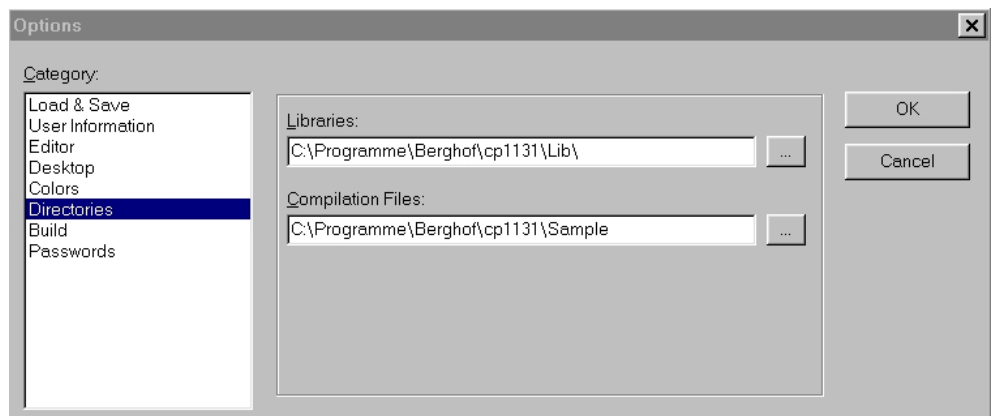
Before commencing with the project, various additional settings must be configured for the CP1131 development environment. Specifically, these are:

- configuring the libraries
- configuring the editors,
- configuring to project-specific hardware.

## Configuring CP1131

### **Configuring Libraries**

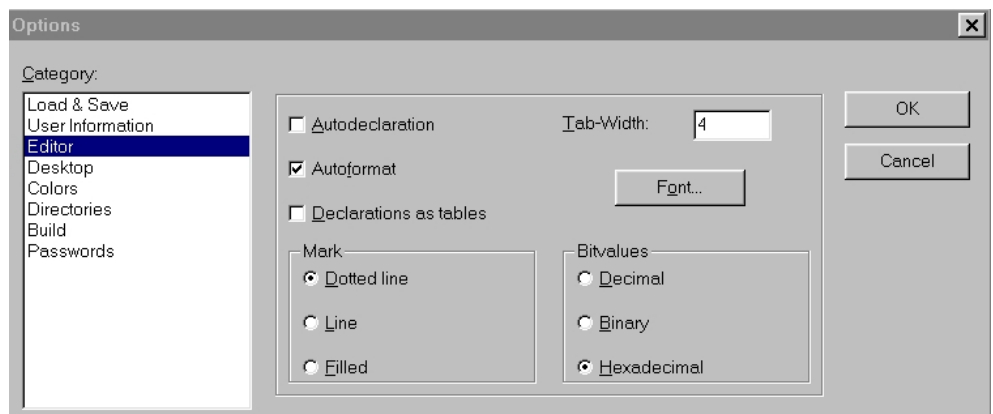
The directory paths of the CP1131 libraries can be configured using the menu *Project/Options*. The screenshot shows the possible settings for the standard libraries supplied with CP1131. Modifications can be typed in directly or set using the browse function.



Configuring the CP1131 library directory paths

### **Configuring the Editors**

The default editor properties can be reconfigured for this PLC sample project. The settings are configured using the menu *Project/Options*. The screenshot shows the required settings.

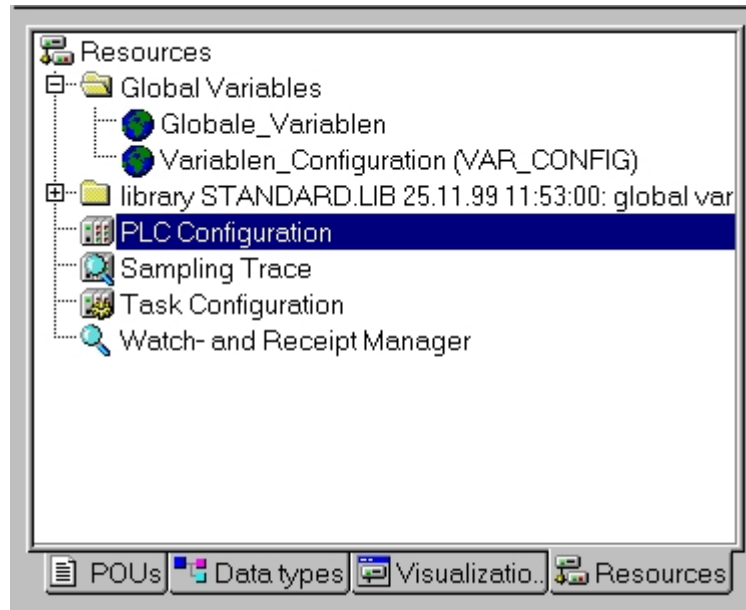


Configuring the CP1131 editor options

## Configuring to Project-specific Hardware

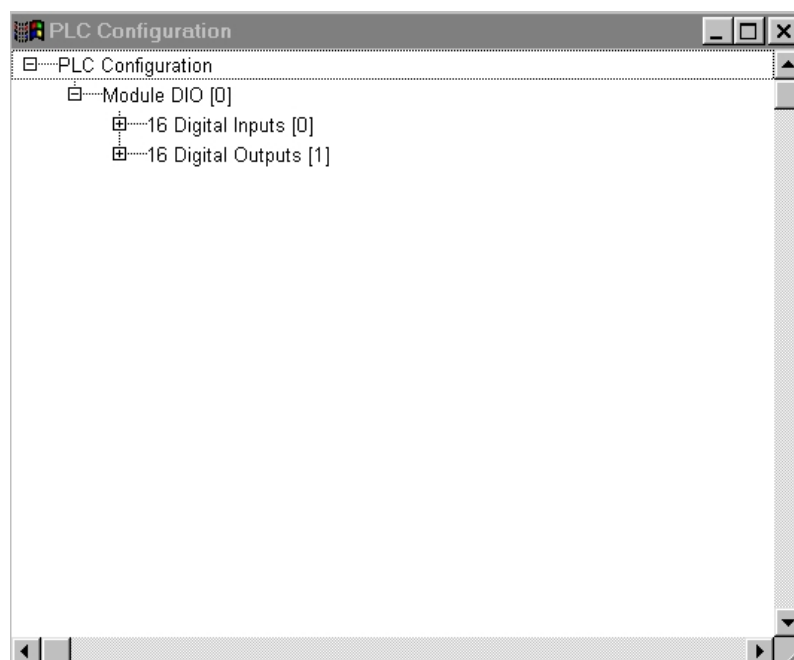
When starting a new project, the hardware used must be reported to the CP1131 development environment.

This is done using the Object Organizer. The Object Organizer is located in the left pane of the development environment. This lists the individual submodules (POUs, Data types, Visualizations, Resources) in a similar way to an index card system. Changes to the hardware configuration are implemented using the menu item Resources in the Object Organizer (see screenshot).



The Object Organizer

Select the Resources tab in the Object Organizer, and double-click the menu item PLC Configuration. The following menu appears.



Configuring the CP1131 editor options

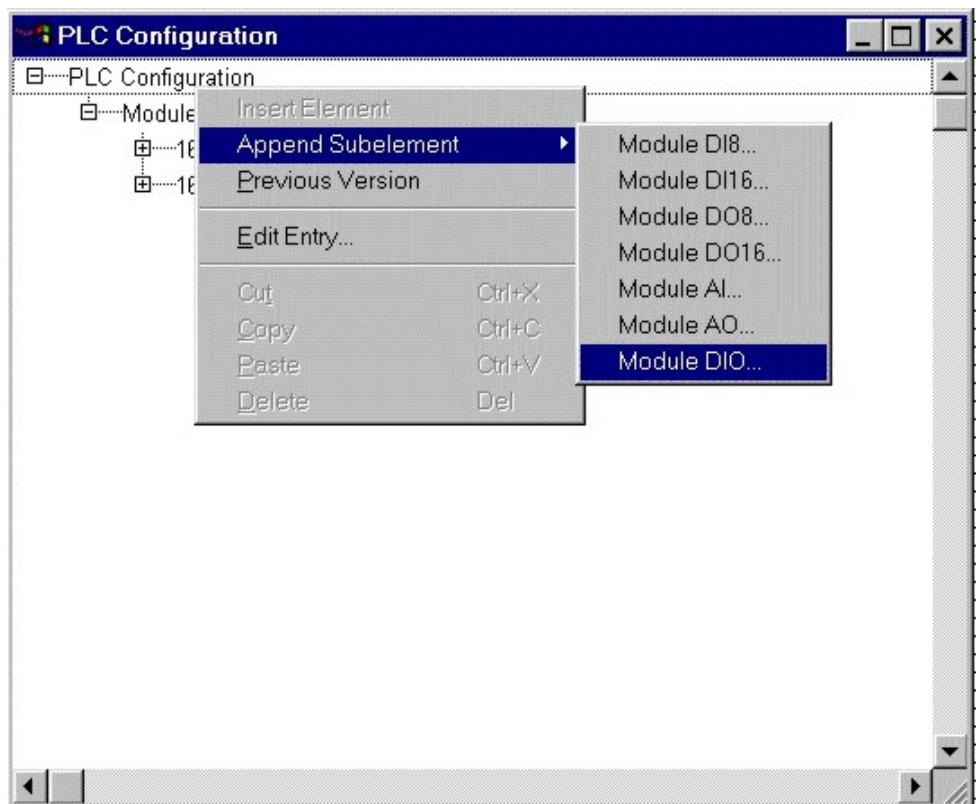
**Completing the Controller Configuration**

The hardware configuration displayed in the previous screenshot is incomplete, as only half of the total 32 digital I/Os are registered.

With this configuration, only 16 of the CDIO 16/16-0,5's digital inputs or outputs can be accessed. The 16 digital inputs and 16 digital outputs displayed should be considered as superposed on top of each other.

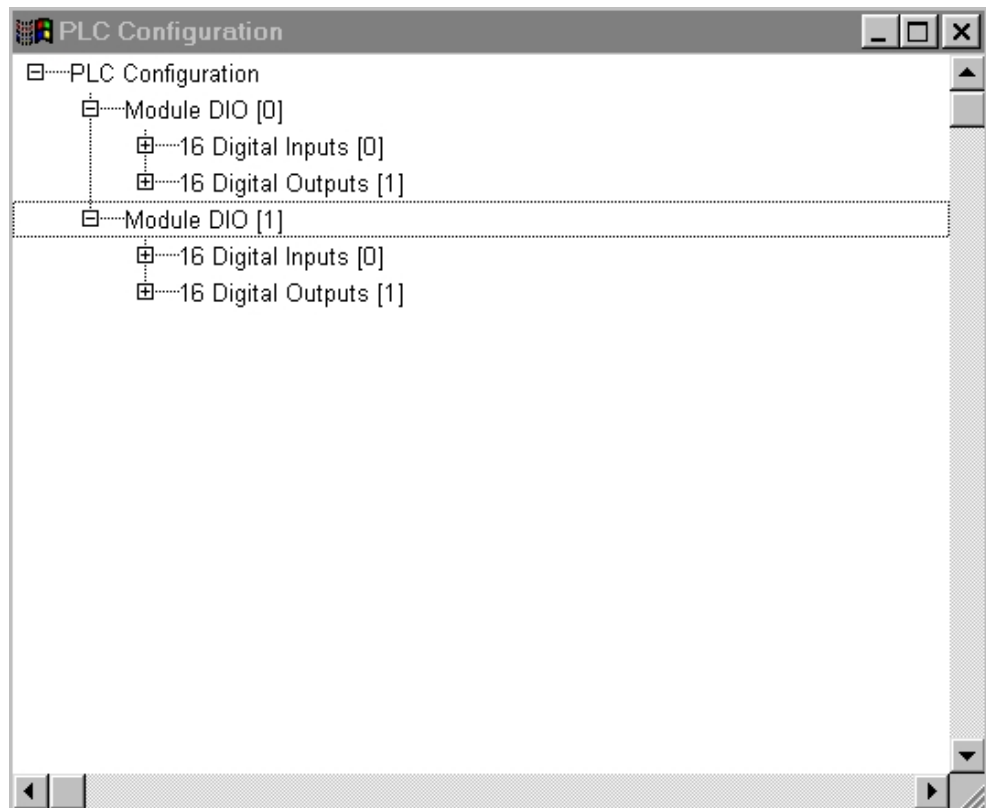
To access all 32 digital I/Os of the CDIO 16/16-0,5, an additional I/O module has to be added to provide the extra 16 digital I/Os.

To do this, select the line PLC Configuration. Right-click with the mouse to add an additional 16-bit digital I/O bank. The required setting is shown in the screenshot below:



Adding another 16-bit digital I/O bank

When you have finished, the following hardware configuration is displayed.



General overview of the hardware configuration

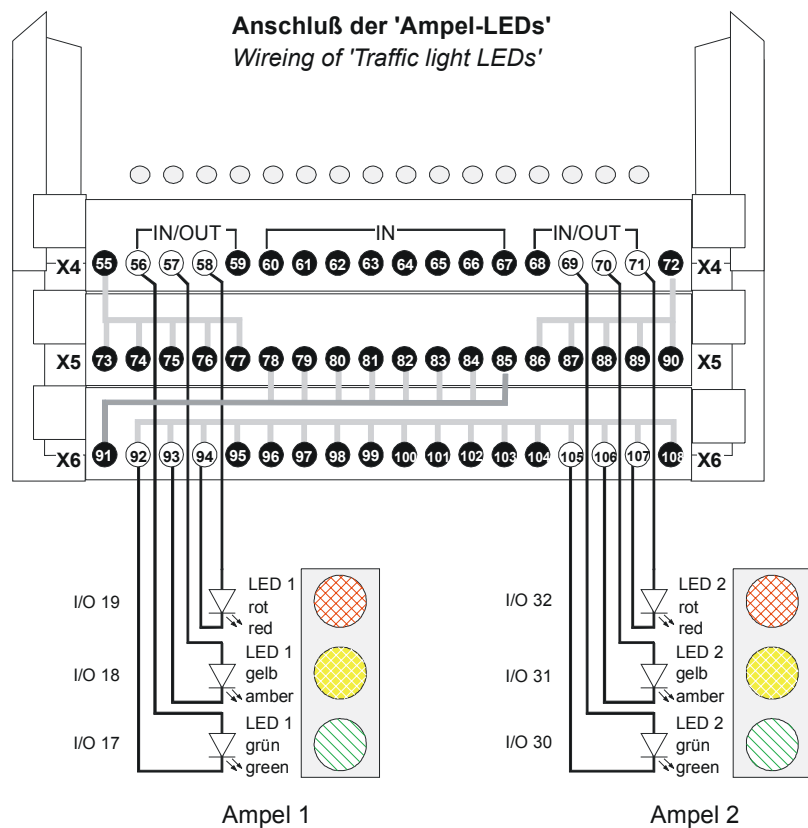
This hardware configuration allows all 32 digital I/Os of the CDIO 16/16-0,5 to be incorporated into the programming for the PLC sample project. The bank CPU [0] displayed here corresponds to the 16 'upper' I/Os (I/O 1–16) of the CDIO module, and the bank Module DIO [1] the 16 'lower' I/Os (I/O 17-32) of the CDIO module.

**Hardware Configuration of the CDIO 16/16-0,5**

**Connecting the 'Traffic Light' LEDs**

The switching status of the digital outputs is always displayed by the status LEDs of the CDIO module.

To illustrate what we mean, in this project we propose to connect two 'traffic lights' consisting of coloured (red, amber, green) 24 V LEDs to the outputs.



2VF100024DG00.cdr

**Assigning Digital Outputs to the Variable Names**

An allocation table is required to assign the digital outputs of the CDIO 16/16-0,5 to the settings specified previously in CP1131.

For the PLC sample project 'Lights.pro', the following allocation table applies:

Connection terminal number	I/O designation	CP1131 variable	Description
X4.56	I/O17	%QX1.0	LED 1 green
X4.57	I/O18	%QX1.1	LED 1 amber
X4.58	I/O19	%QX1.2	LED 1 red
X4.69	I/O30	%QX1.13	LED 2 green
X4.70	I/O31	%QX1.14	LED 2 amber
X4.71	I/O32	%QX1.15	LED 2 red

## The PLC Sample Project under CP1131

The CP1131 development system contains the five programming languages in accordance with IEC 61131-3 (LD, FBD, IL, ST, SFC).

These programming languages are used to develop task-specific types of program module.

These are called PROGRAM ORGANISATION UNITS (POUs).

POUs are used to modularise a project. A project consists, among other things, of a collection of these POUs.

### POU Types in CP1131

Depending on the purpose of the modules to be defined, three different types of POUs can be used.

The interface between the POUs is uniquely identified by the input and output parameters.

These POU types can be used:

- functions
- function blocks (FBs)
- programs

#### Functions

A function can be understood as an aggregate of specific instructions that have only one return value and depend directly on the input parameters.

Internal help states can be utilised by the user. However, these internal states are reset to a defined value when the function is exited.

Example of a function:

Output = 10 \* Input\_1 + (Input\_2 - Input\_3) \* Input\_4

#### Function block

One relevant factor differentiating a function block from a function is the possibility of forming instances from a function block. This means a function block can be used several times, even in different parts of a project.

Another difference between a function block and a function is that several return values are possible in a function block and the internal states (help states) are retained when an instance is called again. This makes it possible to generate dependencies of return values that do not depend only on the input parameters.

There is no dependency between the internal states of the individual instances.

A function block can call functions and instances of function blocks.

Example of a function block:

Output = 10 \* (Input - Dummy)  $\wedge$  Dummy = 2 \* Output

#### **Note:**

The output parameter *Output* depends on both the input parameter *Input* and the internal state *Dummy*. The state *Dummy* is not visible for the POU calling the instance due to the declaration of variables.

To guarantee determinism, the internal states must be static.

Program	<p>The difference between a program and a function block is that programs cannot be 'instanced'. A program is always considered 'global' in a project. Programs can call functions and instances of function blocks. A program can be called by another program or function block.</p> <p>Instances of a FB or functions can be called from FBs and programs.</p>
Program or FB?	<p>The question of whether to use a program or a function block depends primarily on the required project structure.</p> <p>If large parts of the project belong together and are only instanced once, it makes sense to use the POU type <i>program</i>.</p> <p>A function block should always be used in cases where several instances (copies) of the POU are to be used in the project.</p> <p>The choice between program and function block also depends on the project hierarchy. Subordinate hierarchical levels should preferably be formed using FBs, and higher hierarchical levels with programs.</p> <p>A program should not be called from an instance of a function block.</p>

### **How Can the IEC 61131-3 Programming Languages be Used?**

IEC 61131-3

Standard IEC 61131-3 defines the following five PLC programming languages:

'English' name	'German' name
<b>Ladder Diagram (LD)</b>	<b>Kontaktplan (KOP)</b>
<b>Function Block Diagram (FBD)</b>	<b>Funktionsplan (FUP)</b>
<b>Instruction List (IL)</b>	<b>Anweisungsliste (AWL)</b>
<b>Structured Text (ST)</b>	<b>Strukturierter Text (ST)</b>
<b>Sequential Function Chart (SFC)</b>	<b>Ablaufsprache (AS)</b>

The differences between the languages are structural.

While IL and ST are classic programming languages containing line-oriented instructions, (like Assembler, BASIC, PASCAL or C), FBD and LD are graphical programming languages that are characterised by simple line (LD) or block diagrams (FBD).

Sequential Function Chart (SFC) is also a graphical programming language. However, unlike FBD and LD, SFC can be used to create state machines (step sequence control).

The common feature of all of these programming languages is that they can be used to write functions, function blocks or programs that can in turn call functions, function blocks or programs. These can also be written in other programming languages.

**Example of the Use of LD**

LD

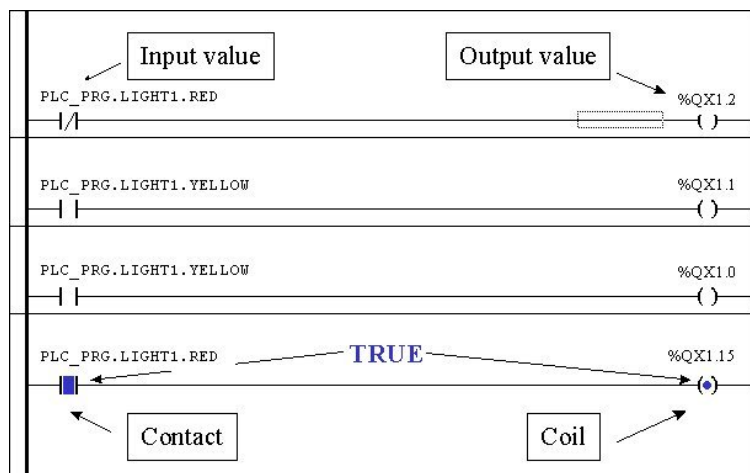
LD is used primarily for Boolean tasks/expressions. POU locks can be implemented easily using LD.

The individual networks are processed from left to right. Contacts are used as input values and coils as output values.

Logical operations such as NOT, NAND, AND, NOR or OR can also be used, as can user-defined function blocks.

Input or output values can be negated by inserting an oblique. S and R functions are also possible.

The advantage of this language is its graphical representation and its simplicity.



Example of LD

**Example of the Use of FBD**

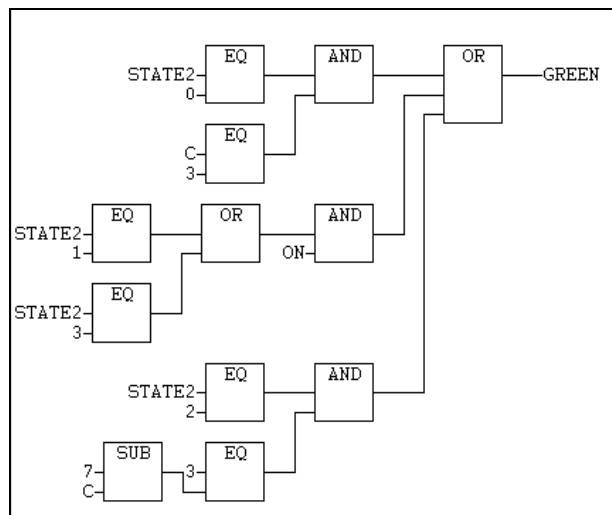
FBD

The process of creating logical and arithmetical networks can be greatly simplified using the PLC language FBD. Whereas logical and arithmetic expressions are not always easy to read in IL and ST, these can be depicted graphically, and therefore transparently, in FBD.

Operations are depicted by a rectangle. Inputs and outputs are shown as solid lines and negations of inputs or outputs as small circles. The operations can of course also be user-defined POU's.

The diagram shows logical links through different modules and the dependencies of the function from left to right.

It shows the dependency of the Boolean expression *GREEN* on the three variables *STATE2*, *C* and *ON*.



Example of FBD

In contrast to the programming languages IL and ST, the formation of the output variables is very transparent here.

**Example of the Use of IL**

IL is extremely widespread in the PLC programming world (e.g. Simatic S5). Its command code is similar to the assembler code used for 80X86 or Z80 processors.

The use of the 'accumulator register' (ACCU) is the outstanding feature of this programming language. The ACCU is a register containing the results of logical or arithmetic operations. Results in variables can also be saved using the ACCU.

The following example is written in IL; comments are between (\* \*) .

LD COUNT	(* load Accu with COUNT	*)
ST LIGHT1.C	(* store Accu at Light1.C	*)
SUB 3	(* Accu = Accu -3	*)
ST LIGHT2.C	(* store Accu at Light2.C	*)
LD STATE	(* load Accu with STATE	*)
ST LIGHT1.STATE2	(* store Accu at Light1.STATE2	*)
ST LIGHT2.STATE2	(* store Accu at Light2.STATE2	*)
LD MODE	(* load Accu with Mode	*)
ST LIGHT1.ON	(* store Accu at Light1.ON	*)
ST LIGHT2.ON	(* store Accu at Light2.ON	*)
CAL LIGHT1	(* call function block Light1	*)
CAL LIGHT2	(* call function block Light2	*)
CAL OUTS	(* call function block Outs	*)
CAL DELAY(T:=t#0.5s)	call function block DELAY with	(*
	(* parameter T=0.5 seconds	*)
	(* data type TIME	*)

**Example of the Use of ST**

ST

Procedural language (high-level language) programmers find IL too unwieldy, and the programming language ST was developed for this reason. In terms of its syntax, ST can be considered as a mix of PASCAL, C and BASIC. ST can be used to define structured programs, function blocks and functions.

The structure of the command set of ST is PASCAL-based. The individual commands can be used to form very complex and therefore powerful expressions. Instead of the jump commands used in IL, the program code can be structured in ST using IF..THEN..ELSE loops or CASE instructions. As a structured programming language, ST does not require direct or indirect jump commands.

Semicolons have an important function in ST. They are used to separate completed commands, but do not end the line. The next command can be written after the semicolon.

**Note:**

In the ST editor, the completion of a command (line) is not signalled by the Enter key (RETURN). Therefore, it is theoretically possible to write large expressions with a number of links over multiple lines. However, to ensure that program listings remain legible, you should only do this to a limited extent.

An example of an ST program is set out below.

```

A:= 10;                (* sets A by 10
                       *)
B:= A MOD X;          (* B stores the rest of the divisio
                       *)
ZAEHLER:=1;           (* sets ZAEHLER by 1
                       *)
REPEAT                (* starts a REPEAT loop          *)
  ZAEHLER:=ZAEHLER*B; (* the product of Zaehler and
                       B is stored in ZAEHLER          *)
  B=B-1;              (* B is decremented          *)
UNTIL B=1             (* condition for a new cycle
                       in the REPEAT loop
                       *)
END_REPEAT;          (* ends REPEAT loop respectively
                       the block          *)
IF ZAEHLER > 5 THEN  (* if condition is true go ahead
                       with the next line
                       *)
  ZAEHLER:=5;        (* limits the value by 5          *)
END_IF;              (* ends IF Block          *)

```

**Example of the Use of SFC**

SFC	<p>There are two ways to use SFC with CP1131.</p> <p>There are SFC steps that comply with IEC 61131-3 (including qualifiers) and some that do not meet this standard.</p> <p>The editor in SFC allows you to combine both types of SFC. SFC is suitable for graphic depiction of state machines using actions and transitions, where every action is assigned to a transition.</p>
Transitions	<p>A transition describes a logical expression that must be true to terminate the current action and start the next action.</p> <p>If the logical expression is false, processing of the current action continues. Transitions can be programmed using any of the IEC 61131-3 programming languages except SFC.</p> <p>There are two ways to program transitions in CP1131.</p> <p>The first option is to write the forwarding condition directly in the flowchart. The transition is depicted as a small horizontal line.</p> <p>The second option is to store the forwarding condition. Any expression can be written in the flowchart.</p> <p>The actual transition is accessed by double-clicking the border of the expression. The programming language is then queried, and the programming language editor appears. A small black triangle on the transition indicates that a condition has been stored.</p> <p><b>Note:</b> To avoid infinite loops when working with SFC, it must be possible to modify the transition condition. This is usually implemented by the action immediately before the transition.</p>
Branches	<p>Branches are used in SFC to call another action depending on the values of the relevant branch transition.</p> <p>If more than one transition is true, the branch with the highest priority is followed. All other branches are disregarded.</p> <p>In CP1131, the first branch (extreme left in the flowchart) has the highest priority, and the last branch (extreme right in the flowchart) the lowest. If the last branch condition is true, it is only enabled if all branch conditions to its left are false.</p> <p>Parallel branches cannot be implemented by means of 'normal' branches with one transition per branch. Parallel branches must be used for parallel branching. These are depicted differently from 'normal' branches (double horizontal line).</p> <p><b>Note:</b> If more than one branch condition is true, the branch condition with the highest priority is always executed. The highest priority is always situated to the left of the branch, the lowest on the right.</p>

Jump

Jumps can be placed immediately after transitions. If the transition condition is true, the jump is executed. The name of the action to which the system required to jump is entered as the jump value.

**Note:** If parallel branches are used in the project and these must also be accessed using jumps, it must be ensured that the action accessed by the jump starts before the parallel branch.

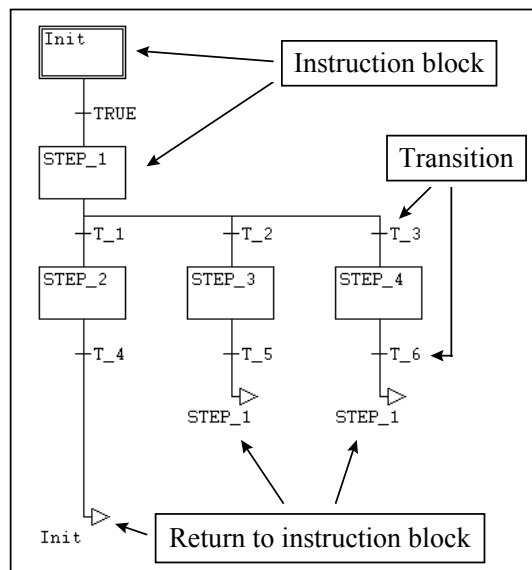
Under no circumstances can an action in the parallel branch be accessed by a jump. If necessary, a dummy action/transition that serves purely as a jump label must be inserted.

If this is not implemented, the sequence may contain defective functions.

Action

An action is a string of executable code. It can be programmed using any language that complies with IEC 61131-3. It is thus also possible to nest SFC code hierarchically. However, attention must be given to transitions to lower hierarchical levels to ensure that different SFC diagrams terminate in the right order. If this is not ensured, a SFC diagram with a lower priority may not necessarily be exited by the action (intended by the programmer) that the user intended (e.g. the init block). To ensure that the main loop cycle is complete, jumps ('return to instruction block') can be used.

SFC can be used if project states are obvious and the program is intended to be executed sequentially. This programming language is very well suited to the implementation of state machines.



Example of the use of SFC

When a SFC sequence such as the one shown above is executed for the first time, the init action is called. An init block such as this can be found in every SFC sequence. Init blocks are distinguished from other actions by a double border.

Continued execution of the program is dependent on the transitions which state which expression is true.

## The 'Lights.pro' Sample Project

This section describes the sample project supplied with CP1131, 'Lights.pro'. With 'Lights.pro', you can see how easy it is to implement a project with CP1131 using the hardware and software described. The 'Lights.pro' project controls two traffic lights that are independent of each other. Care has been taken to ensure that there is no interaction between the traffic lights so that timers used by these are independent of each other.

### Starting the 'Lights.pro' Sample Project

Load the sample project After starting CP1131, select the option *Open* from the *File* menu. In the CP1131 directory...\Sample you will find the sample 'Lights.pro'.

*New project:*

*You can also create a new project at this point. To do this, select the option New from the *File* menu. Then right-click with the mouse in the blue-highlighted folder POU in the left pane of the Object Organizer; a context menu appears. Select Add object to insert any number of POU's. The first POU that you should create is the POU proposed by CP1131 at this point, PLC\_PRG. Use the proposed POU type 'program'. You should also set the required programming language in this window. You can use this menu to create further programs, function blocks or functions in any of the IEC 61131-3 programming languages at any time.*

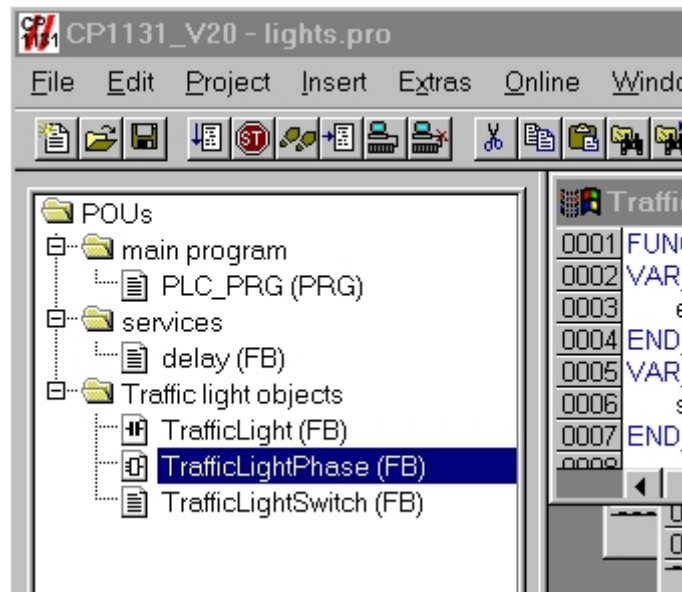
This sample project uses the following POU's:

POU Name	POU Type	Programming language
PLC_PRG	Program	ST
TrafficLight	Function block	LD
TrafficLightPhase	Function block	FBD
TrafficLightSwitch	Function block	ST
delay	Function block	ST

### Object Organizer

CP1131 allows the elements used (data types, modules, etc.) to be structured by folder in the Object Organizer. The Object Organizer is permanently displayed as a separate window at the left edge of the development environment. In the Object Organizer, multiple elements of a category can be combined within the open project. The names of the folders are not important; the elements are known globally throughout the entire project.

The following depiction of the Object Organizer window shows a summary of the POU's from the POU's category contained in the project 'Lights.pro'.



Structuring the POU's with the Object Organizer

You can see the folders main program, services and Traffic light objects that were created for the 'Lights.pro' project. The user determines how POU's or data types (see below) are distributed between folders, and the naming convention for those folders. This ability to structure the folders as required allows projects to be managed very transparently.

Data types

The sample project also uses other data types in addition to the POU's. This table provides an overview of the data types used and their purposes.

Data type name	Data type	Description
TrafficLightColours	Structure	Contains the 3 traffic light colours
TrafficLightPhases	Enumeration	Assignment to int values
States_PLC_PRG	Enumeration	INIT or application

Using these user-definable data types, you can group project variables used e.g. in transfer interfaces, as you wish (structures) or directly name process variables whose values symbolise particular states (enumeration).

You can select existing data types or create user-defined data types using the Object Organizer. To do this, select the corresponding tab at the bottom of the window (2<sup>nd</sup> tab from the left).

### The *TafficLight* Function Block

**TrafficLight** function      The function block *TrafficLight* is added hierarchically to the TrafficLightObjects folder. *TrafficLight* is used to set 3 physical CDIO 16/16-0,5 outputs according to the *sTrafficLightColours* variables provided over the input interface. The variable *sTrafficLightColours* here is derived from the user-defined structure *TrafficLights-Object*.

Instances of *TrafficLight*      To use the function block several times in the same project, as many instances as are required must be created. The variables defined in the function block are then managed with each instance using the 'full-stop operator' (e.g. *sTrafficLightColours.RED*).

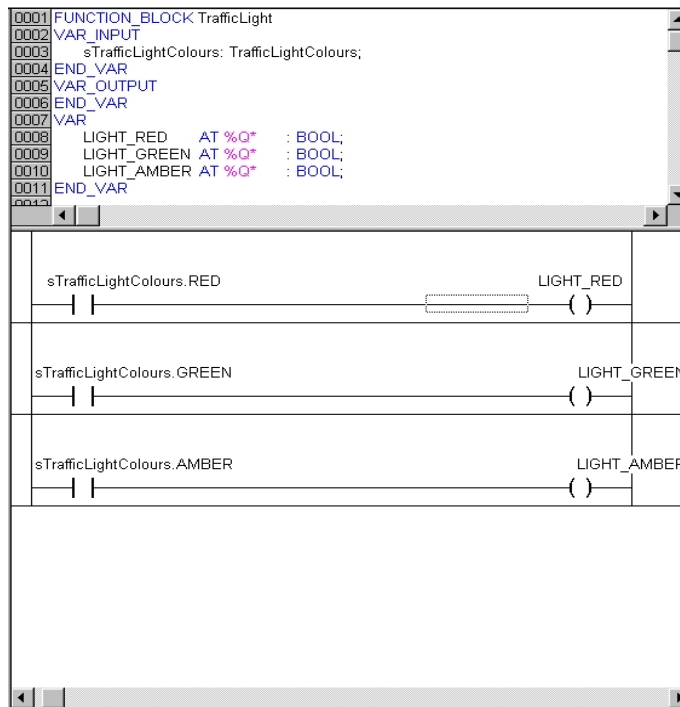
The physical addresses are handled differently. These are managed using %Q in CP1131. If permanent addresses are allocated in the function block (e.g. %QX0.1), these become permanent in all instances, i.e. all instances access the same physical outputs of the CDIO 16/16-0,5.

However, to form function blocks with multiple instances that have access to physical outputs, a specific syntax must be adhered to for the variable declaration. This is shown in the image below.

Physical addresses are assigned using the *Variable\_Configuration* menu.

Variable declaration      As can be seen in the following image, *sTrafficLightColours* is declared as the input variable. The type from which *sTrafficLightColours* was derived is user-defined, and not a standard CP1131 type such as e.g. INT or BYTE.

The variables *LIGHT\_RED*, *LIGHT\_GREEN* and *LIGHT\_AMBER* are declared as internal function block variables and therefore encapsulated. The function block itself is only viewed externally through its input and output variables. Internal variables are not and should not be taken out. Nevertheless, an output is obtained from this function block.



Traffic Light function block programmed in LD

Using structures

As mentioned previously, *sTrafficLightColours* is derived from the user-defined data type *TrafficLightColours*, and has an underlying structure. The structure itself is declared as follows:

```

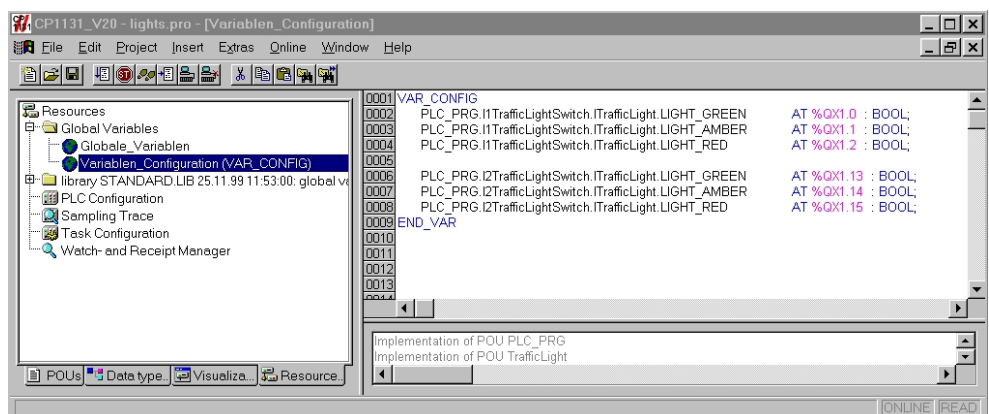
TYPE TrafficLightColours :
STRUCT
    GREEN      : BOOL;
    AMBER      : BOOL;
    RED        : BOOL;
END_STRUCT
END_TYPE
    
```

Access via these variables occurs as shown in the previous image, via the full-stop operator; i.e. the variable *RED* of the structure is accessed with the expression *sTrafficLightColours.RED*. This mode of access can be found at various locations in CP1131. This method can be used inter alia to access the input and output variables of functions, instances or even programs.

Variable\_Configuration

The variable declaration of Boolean variables using the *AT %Q\** command supports the flexible assignment of physical outputs. In a separate Object Organizer menu (Resources), you can permanently link the physical outputs or inputs with the outputs of the individual instances of the function blocks defined in this way. This makes it possible to make changes easily even at a later stage. To ensure that changes to the I/O can only be made at one location, the I/O variables (%Q, %I) of all function blocks must be declared according to this diagram.

'Virtual' outputs of instances are allocated to physical project addresses (CDIO 16/16-0,5 addresses) using the *Variable\_Configuration* menu in the Resources tab in the Object Organizer. The screenshot shows the allocation of virtual outputs to the physical addresses in the sample project 'Lights.pro'.



Variable configuration in 'Lights.pro'

### The *TrafficLightPhase* Function Block

#### *TrafficLightPhase* function

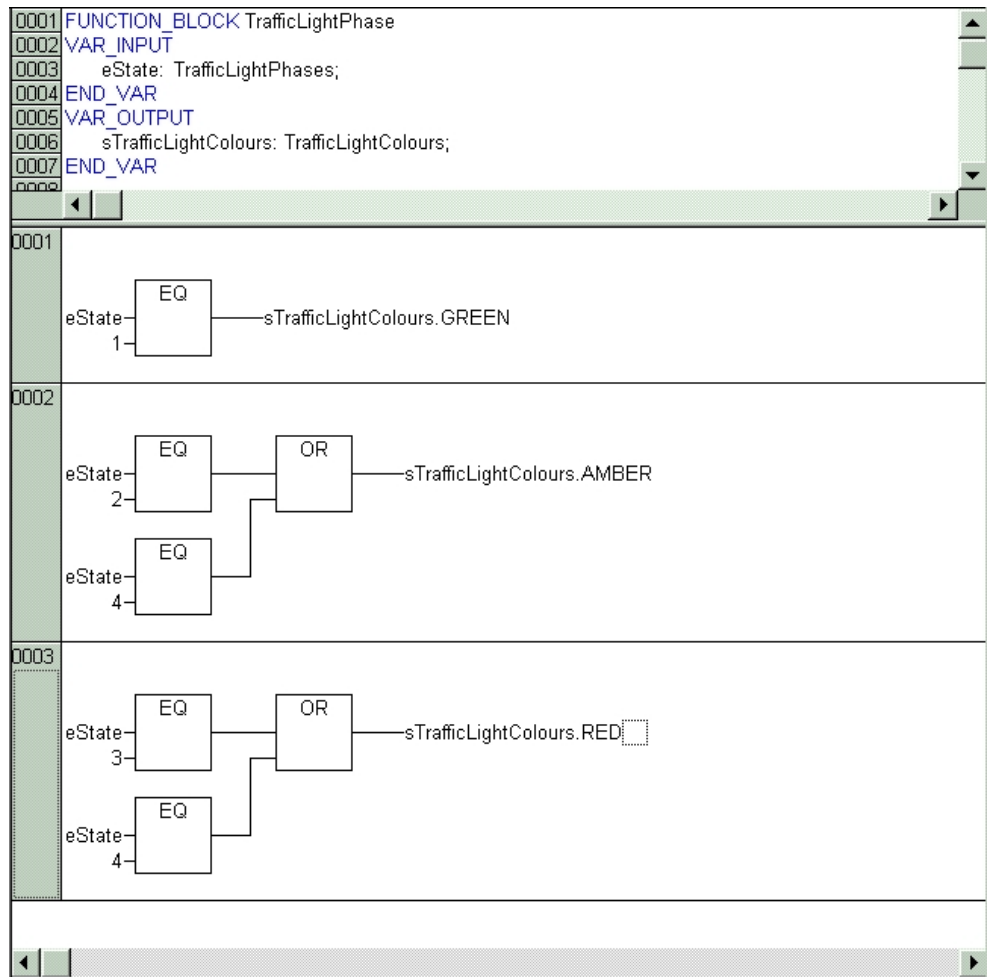
Using a value *eState*, this function block determines the outputs (amber, green, red) of one traffic light. This function is implemented with FBD.

Several outputs can be active as a result of one state. Like the input variable of the function block *TrafficLight* the output itself is issued to an identically named variable *sTrafficLightColours*, which is also derived from the *TrafficLightColours* structure.

The identical variable names do not present a problem, as the scope of the function blocks is only local. In addition, the variables of the function block are accessed via the instances of the function block using the full-stop operator.

#### FBD

The screenshot shows the three networks that determine the individual phases of a traffic light. Only the functions *EQ* and *OR* are used in all three networks. *EQ* compares the input of the block (e.g. in network 1: state and 1) with each other. If they match, the Boolean output value of the block is set accordingly. This output value is directly assigned to the variables *sTrafficLightColours*.{*RED*, *GREEN*, *AMBER*}. Here you can see the mixture of algebraic (*EQ*) and logical (*OR*) functions in a network that is discussed above.



Determining the traffic light colours with the *TrafficLightPhase* FB

## The *delay* Function Block

### *Delay* function

The function block *delay* is used to implement process delays in the project and return a status on the state of the delay. A *delay\_time* is used as the input variable. The Boolean variable *OK* is returned as the output variable.

In 'Lights.pro', this function block is used to implement the lighting interval of the individual traffic light phases. To do this, the instance of the FB must be called by the superordinate instance until the return value 'toggles' *OK*.

In CP1131, a POU does not automatically return values when changed. The instances required by the user must be queried continually until the desired result is obtained.

How this appears and how operation with the superordinate instance continues depends on application. CP1131 does not provide any further suggestions. A POU must therefore only be called if it is needed for the current application.

**Note:** After a single POU call in CP1131, the values linked to it (POU output values) are not updated automatically.

The user must initiate updating explicitly by repeatedly calling the block.

The function block displayed below was programmed in ST.

This function block can be used at many different locations in a project due to its universal (simple) interface. As a result, it is not in the Traffic light objects folder like the other function blocks, but is located under Services

```

0001 FUNCTION_BLOCK delay
0002 VAR_INPUT
0003     delay_time: TIME;
0004 END_VAR
0005 VAR_OUTPUT
0006     OK: BOOL:= FALSE;
0007 END_VAR
0008 VAR
0009     del: TP;
0010 END_VAR
0011
0011 IF del.Q THEN
0012     del();
0013 ELSE
0014     del(IN:=FALSE);
0015     del(IN:=TRUE,PT:=delay_time);
0016 END_IF
0017 OK:=del.Q;

```

delay function block

## The *TrafficLightSwitch* Function Block

The screenshot below shows the code for *TrafficLightSwitch* code, which is written in ST.

It includes the instances of the function blocks *TrafficLight*, *TrafficLightPhase* and *delay* that are listed in the variable declaration within the function block.

This function block controls the cyclical traffic light control process using the input variables *eStartState* and *SetState*. Using *eStartState* and *SetState*, the instance of the *TrafficLightSwitch* function block can be initialised at start-up, so that the required traffic light phase can begin immediately after the traffic light controller is started.

The delay times of the individual traffic light phases are specified in the array *TLTimes*.

```

0001 FUNCTION_BLOCK TrafficLightSwitch
0002 VAR_INPUT
0003   SetState      : BOOL;          (* If this variable equals TRUE, the state of the instance can be changed from outside*)
0004   eStartState   : TrafficLightPhases; (* sets the StartState of the state machine, only valid if SetState equals TRUE *)
0005 END_VAR
0006 VAR
0007   eState        : TrafficLightPhases; (* State of the state machine*)
0008   ITrafficLight : TrafficLight;      (* Instance which switches the lights*)
0009   ITrafficLightPhase: TrafficLightPhase;
0010   Idelay        : delay;
0011   TLTimes       : ARRAY[1..4] OF TIME:=T#5s,T#2s,T#5s,T#2s;
0012 END_VAR

0001 IF SetState THEN
0002   eState:=eStartState;
0003 ELSE
0004   IF eState = GREEN THEN
0005     ITrafficLightPhase(eState:=GREEN);
0006     ITrafficLight.sTrafficLightColours:=ITrafficLightPhase.sTrafficLightColours;
0007     ITrafficLight();
0008     Idelay(delay_time:=TLTimes[eState]);
0009     IF NOT Idelay.OK THEN
0010       eState:=AMBER;
0011     END_IF
0012   END_IF
0013   IF eState = AMBER THEN
0014     ITrafficLightPhase(eState:=AMBER);
0015     ITrafficLight.sTrafficLightColours:=ITrafficLightPhase.sTrafficLightColours;
0016     ITrafficLight();
0017     Idelay(delay_time:=TLTimes[eState]);
0018     IF NOT Idelay.OK THEN
0019       eState:=RED;
0020     END_IF
0021   END_IF
0022   IF eState = RED THEN
0023     ITrafficLightPhase(eState:=RED);
0024     ITrafficLight.sTrafficLightColours:=ITrafficLightPhase.sTrafficLightColours;
0025     ITrafficLight();
0026     Idelay(delay_time:=TLTimes[eState]);
0027     IF NOT Idelay.OK THEN
0028       eState:=RED_AMBER;
0029     END_IF
0030   END_IF
0031   IF eState = RED_AMBER THEN
0032     ITrafficLightPhase(eState:=RED_AMBER);
0033     ITrafficLight.sTrafficLightColours:=ITrafficLightPhase.sTrafficLightColours;
0034     ITrafficLight();
0035     Idelay(delay_time:=TLTimes[eState]);
0036     IF NOT Idelay.OK THEN
0037       eState:=GREEN;
0038     END_IF
0039   END_IF
0040 END_IF

```

Traffic Light Switch function block

Instances-Call	This function block contains the different types of input and output variable allocations with the assistance of instances.
State machines	<p>The function block <i>TrafficLightSwitch</i> essentially symbolises two state machines. The first state machine has two modes:</p> <ul style="list-style-type: none"><li>- Initialisation (<i>SetState = TRUE</i>) and</li><li>- Application (<i>SetState = FALSE</i>).</li></ul> <p>The second state machine is shown in application mode. This is the actual traffic light controller.</p> <p>The second state machine is the more interesting part here.</p>
1 <sup>st</sup> state machine	<p>As already explained above, the first state machine knows both states, initialisation and application mode.</p> <p>In initialisation mode, the start state of the second state machine can be set, so that the start of the traffic light phase can be set by the level that calls the instance of this function block. It is therefore possible to implement multiple instances of the FB with differently set phases. Therefore, for example, the first instance of the FB can display the traffic light colour green immediately after initialisation and a second instance the traffic light colour red.</p> <p>When initialisation has terminated (input variable <i>SetState = FALSE</i>), the second state machine is activated. As this process is determined by the input variable <i>SetState</i> it is only possible to switch from initialisation mode to application mode from outside, i.e. only from the level calling the instance.</p>
2 <sup>nd</sup> state machine	<p>The second state machine only becomes active when initialisation mode is exited. To do this, the input variable must be <i>SetState = FALSE</i>. If this is the case, the lines from 4 to 40 shown in the screenshot at the bottom of the window are active. These lines contain four large <i>IF..END_IF</i> blocks, each of which represents a state. Execution starts from the state <i>eStartState</i>, which was assigned using the first state machine in line 2 during initialisation. Forwarding of states depends on the expression that terminates the state.</p> <p>This is to prevent forwarding of the state from re-calling the state that triggered forwarding.</p> <p>This can be verified using the <i>IF</i> queries expression.</p> <p>The instructions in the <i>IF</i> blocks are very similar to each other. The differences are in the forwarding criteria, delay times (retrieved from the <i>TLTimes</i> array) and the traffic light phases to be depicted. Forwarding itself depends on the delay time of the state.</p> <p>If the value toggles <i>delay.OK</i>, forwarding is executed.</p> <p><b>Note:</b> If a state machine does not behave correctly, check the expression, dependency and next state of the forwarding condition.</p>

### The PLC PRG Program

**PLC\_PRG function** If the CP1131 tasks are not used in *PLC\_PRG*, the main program is used here. The instructions in *PLC\_PRG* can be used to better structure subordinate levels and expand them more easily. In complex projects for which you do not want to use the task management function in CP1131, *PLC\_PRG* can be used to implement a process-dependent scheduler that can be used to call instances of function blocks, functions and programs again. In the 'Lights.pro' example, *PLC\_PRG* is used to manage two instances of the FB *TrafficLightSwitch*.

**State machine** *PLC\_PRG* also contains a small state machine. It is actually used only for initialising the two instances *I1TrafficLightSwitch* and *I2TrafficLightSwitch*. The transfer of the two initialisation values in lines 3 and 7 of *PLC\_PRG* forms the functionality of the FB *TrafficLightSwitch*. When lines 4 and 8 are called, the two instances are called and initialised for the first time. The differing methods for transferring instance variables are illustrated here once again. After both instances have been initialised, forwarding of the *PLC\_PRG* state machine is triggered in line 10. In the next and all subsequent cycles of *PLC\_PRG*, the *ELSE* branch of the *IF* instruction is executed. The relevant transfer values are transferred when both instances are called in lines 14 and 15.

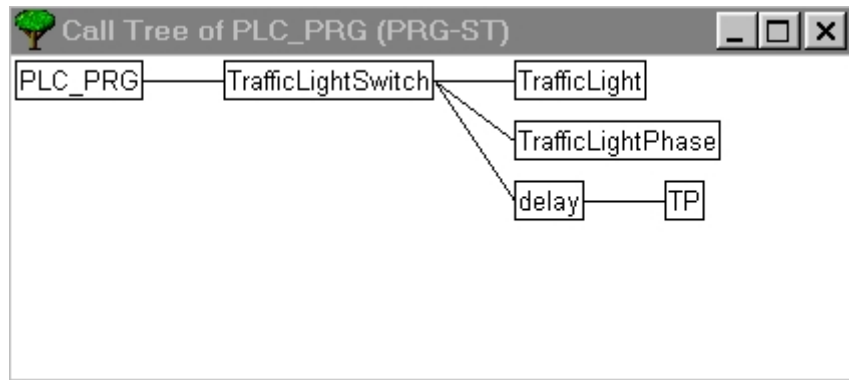
```
0001 (* Alternative traffic light program, which achieves a higher structure of segmenting the program
0002 as the 'normal' traffic light program *)
0003 (* Berghof GmbH, CANtrol, 18.07.00 A. Hablitzel *)
0004
0005 PROGRAM PLC_PRG
0006 VAR
0007   I1TrafficLightSwitch: TrafficLightSwitch; (* First instance of the FB TrafficLightSwitch *)
0008   I2TrafficLightSwitch: TrafficLightSwitch; (* Second instance of the FB TrafficLightSwitch *)
0009
0010   State: States_PLC_PRG:=INIT; (* Only used for initialization *)
0011                               (* sets the value to the enum 'INIT' *)
0012 END_VAR
0013
0001 IF State=INIT THEN (* If condition TRUE then initialization *)
0002   I1TrafficLightSwitch.SetState:=TRUE; (* INIT: sets the beginning for the *)
0003   I1TrafficLightSwitch.eStartState:=GREEN; (* first instance *)
0004   I1TrafficLightSwitch(); (* Call of the instance *)
0005
0006   I2TrafficLightSwitch.SetState:=TRUE; (* INIT: sets the beginning for the *)
0007   I2TrafficLightSwitch.eStartState:=RED; (* second instance *)
0008   I2TrafficLightSwitch(); (* Call of the instance *)
0009
0010   State:=APPL; (* Set state machine to application *)
0011               (* after finishing the initialization *)
0012 ELSE (* If condition FALSE then application *)
0013
0014   I1TrafficLightSwitch(SetState:=FALSE); (* Call first instance *)
0015   I2TrafficLightSwitch(SetState:=FALSE); (* Call second instance *)
0016 END_IF
0017
0018
0019
0020
```

PLC\_PRG written in ST

## Project Structure

### Call tree

With the call tree (menu option: *Project/Show Call Tree*), a graphical output of the project hierarchy is displayed. Here, the roots of the hierarchy are formed by the currently selected POU. If you select e.g. *PLC\_PRG* in the Object Organizer (POU tab), the following diagram is displayed. If you select a different POU, the call tree for that POU is shown.



Call tree of PLC\_PRG

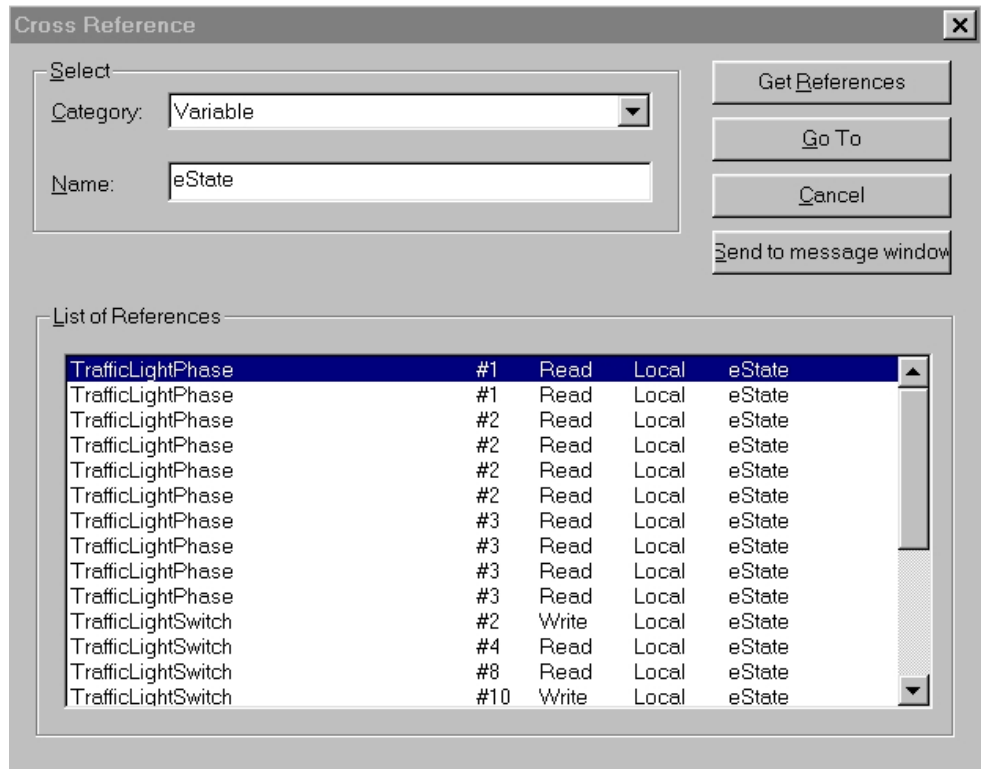
This diagram depicts the hierarchy of calls in the individual POUs. This shows that the POU *TP* is called from the POU *delay*.

The POUs *TrafficLight*, *TrafficLightPhase* and *delay* are called by the POU *TrafficLightSwitch*, which is in turn called by *PLC\_PRG*. The dependencies of each POU can be visualised using this tree.

**Note:** The generation of project information such as *call tree*, *cross-reference list*, *unused variables* only works if the project as compiled is up-to-date. The project is not yet compiled when it is loaded, so these menu options are greyed out and cannot be selected. To ensure that your references are always current, you should compile your project before processing with *call tree*, *cross-reference list*, *unused variables*. To do this, select the menu option *Project/Rebuild All* or press F11.

Cross-reference list

The cross-reference list (menu option *Project>Show Cross Reference...*) allows the user to search the variables, addresses and POUs used in the project according to use. The line number, the type of access and the POU in which the variable, address and POU are used are listed here. The following image is a screenshot:

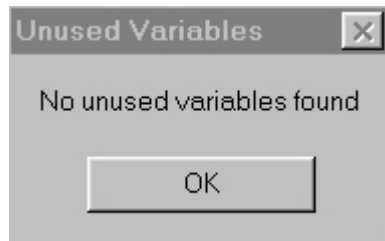


Cross-reference list for the eState variables

Double-click the selection or select *Go To* to edit the selected element in the affected POU. The editor corresponding to the POU concerned then opens automatically.

Unused variables

By selecting the menu option *Project>Show unused Variables...*, the variables declared in a project in any POU but not used can be found. Unused variables are listed. If there are no unused variables, the following message appears:



Unused variables

**Note:** Unused variables in a project consume resources in terms of memory space and internal administration and represent an unnecessary utilisation of project resources. For this reason, unused variables should always be deleted.

---

## **Starting the Sample Projects on the CDIO 16/16-0,5**

- Preparing to download    Once the program analysis is complete, the project can be compiled to generate executable code for the CPU module.  
Select the menu item *Login* from the *Online* menu to instruct the compiler to first check the project for syntactic errors. If compilation is successful and does not contain any errors, the program is loaded into the module after completion of a dialog.  
It is a compulsory requirement at this stage that the node numbers and communications options settings are checked. If the settings are incorrect, CP1131 informs the user by displaying an error message. If there is a correctly installed CANtrol CAN network with an unique baud rate, the specific node address in CP1131 can differ from the node address to which the programming cable is connected. The CANtrol CPU modules are equipped with a bridge function that makes it possible to load projects to modules that are not directly connected by the programming cable.
- Note:**  
If errors occur while downloading, you must ensure that the correct communication connection and node ID are specified. If a project is downloaded via several CPUs using the bridge function, you must ensure that all CAN nodes in application mode have the same baud rate and that the node ID specified in CP1131 exists in the network.  
Another error common at the start is attempting to work with the CNW and CP1131 in parallel. This is not currently possible, as the communications interfaces cannot yet be subdivided.  
If you want to work with CP1131, the CNW must be closed, and vice versa.
- Error messages    CP1131 informs the user of syntax errors. A possible cause of the error is also proposed and number of the line containing the error displayed.  
Click on an error message to display the editor for the affected program sequence. Move on to the next error with F4.  
  
CP1131 cannot perform a check for logical errors. As with all programming languages, this is the responsibility of the user.
- Simulation    In some cases, it may be advisable to run project simulations before the CPU download. CP1131 offers a simulation environment for this purpose in which the projects run online, within the development environment.  
You can specify whether a project runs on the CPU or within the development environment by selecting the menu item *Simulation* from the *Online* menu.  
No CANtrol-specific characteristics can be simulated, e.g. CAN communication, SIO communication, Ethernet communication, STRCNV.LIB, etc.

- Online operation
- If the project is *online*, the editors change.  
In *PLC\_PRG*, for example, the ST editor is subdivided. In the upper section of the window, the values of the declared variables are displayed directly. Existing data types and instances can be expanded by double-clicking with the mouse. This makes it possible to view all relevant data on an object at a glance.
- In the lower section, the source code as it appears offline is displayed on the left, and the variables used in the individual lines are displayed on the right. Thus the process-related data can also be viewed in the source code.
- In online operation, every change in the controller is automatically visible. The only issue is a slight delay in the transfer of the online data via the programming cable, which means that 'rapid' processes cannot be accessed in the development environment online.
- Starting the project
- If you are online, you can start the project in the CPU using the menu option *Online/Run* (only possible if the simulation has been deactivated). The CPU is then in run mode. LED 3 on the CPU is lit continuously to indicate this. If the menu item *Stop* from the *Online* menu is selected, the CPU enters stop mode. In stop mode, LED 4 is illuminated instead of LED 3.
- Debugging instances online
- The individual variables of instances can be identified in the variable declaration of the level at which the instance was created.
- To debug the instance code, the function block from which the instance is derived must be selected in online mode. Then select the menu item *View Instance* from the *Project* menu to display a list of the instances assigned to the function block. The code of the instance can then be viewed online by simply selecting it. If just the corresponding function block itself is selected instead (like selecting, but double-click with the mouse), the source code is displayed but the variable values appear together with question marks. This is because CP1131 does not know which instance to consider.
- Flashing projects
- If projects are intended to run permanently in the CPU (without continual project downloading via CP1131 each time the controller is powered down), the project can be saved to the flash memory if the CPU is in stop mode.
- To do this, select the menu option *Online/Save Application to Flash*. This stores the project permanently in the flash memory.
- If the project is saved immediately after it has been flashed, the project can be viewed with CP1131 without any further downloading after the controller is powered down.
- CP1131 allows you to access a running project in the CPU actively in two ways, either by forcing values or by making online changes.

## Forcing values

The first method is by selecting variables in online mode and changing their values. Double-click any variables to display a dialog in which you can set the values of these variables within their range limits. The values are checked by CP1131 when input is complete.

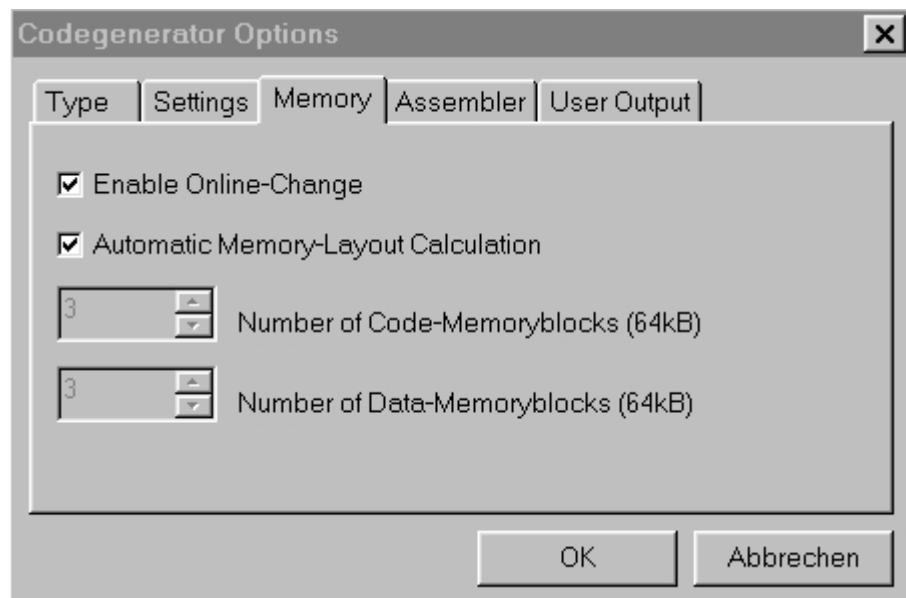
If the specified variable value is correct according to the type convention, it is displayed in red (although the value is not yet in the controller). This makes it possible to go on to modify other variable values. When all changes have been made, the values can be written to the controller.

To do this, use the menu items *Write Values* (the specified values are written once only when the next cycle is run) and *Force Values* (the specified values are written in every cycle) in the *Online* menu. You can terminate continuous writing to the controller by selecting the menu option *Release Force*.

## Online changes

The other way to actively access a running project is by making online changes. With online changes, you can access the code directly, not just the variable values of the running project.

To do this, modify the setting in the menu option *Online/Codegenerator Options* so that it corresponds to the image below.



Code generator setting

The code can only be modified during operation with this setting. The changes have to be implemented in offline mode in CP1131.

The next time you log in using *Online/Login*, the difference resulting from the changes is displayed in a CP1131 dialog and the change is incorporated into the CP1131 code during operation.

## Breakpoint

Breakpoints can be set in a running or stopped application. The locations at which the breakpoints are set are displayed in blue. If the CPU comes to a breakpoint, the corresponding flag in the project changes in the display from blue to red. This is also indicated by an LED change on the CPU cover. The red LED (4) is lit continuously and the green LED (3) blinks during this state.

You can set or delete breakpoints using the F5/F9 keys or the *Online* menu. You can also step through a project with single steps (*Online/Single Step In*) or with a single cycle (*Online/Single Cycle*).

blank page

## Digital Inputs/Outputs

Outputs may also be connected to inputs without additional external load.

### Grouping of Inputs/Outputs

The grouping facility permits formation of groups, separate power circuits, emergency off circuits, etc. as and when required.

Inputs/outputs can be supplied in groups as

- 2 input groups and
- 4 output / input groups

The **modular electronic circuit** for C modules is supplied together with input group 2 (Group 2) over connection terminals 1 (L1+) and 2 (M1).

The modular electronic circuit must be supplied with power in **any** cases, otherwise the modules will be inoperable.

Supply must be provided directly (unswitched) from the supply unit.

#### **Inputs**

Inputs (sensors) must be supplied directly from the supply unit.

Do not conduct the sensor supply through switched circuits.

#### **Outputs**

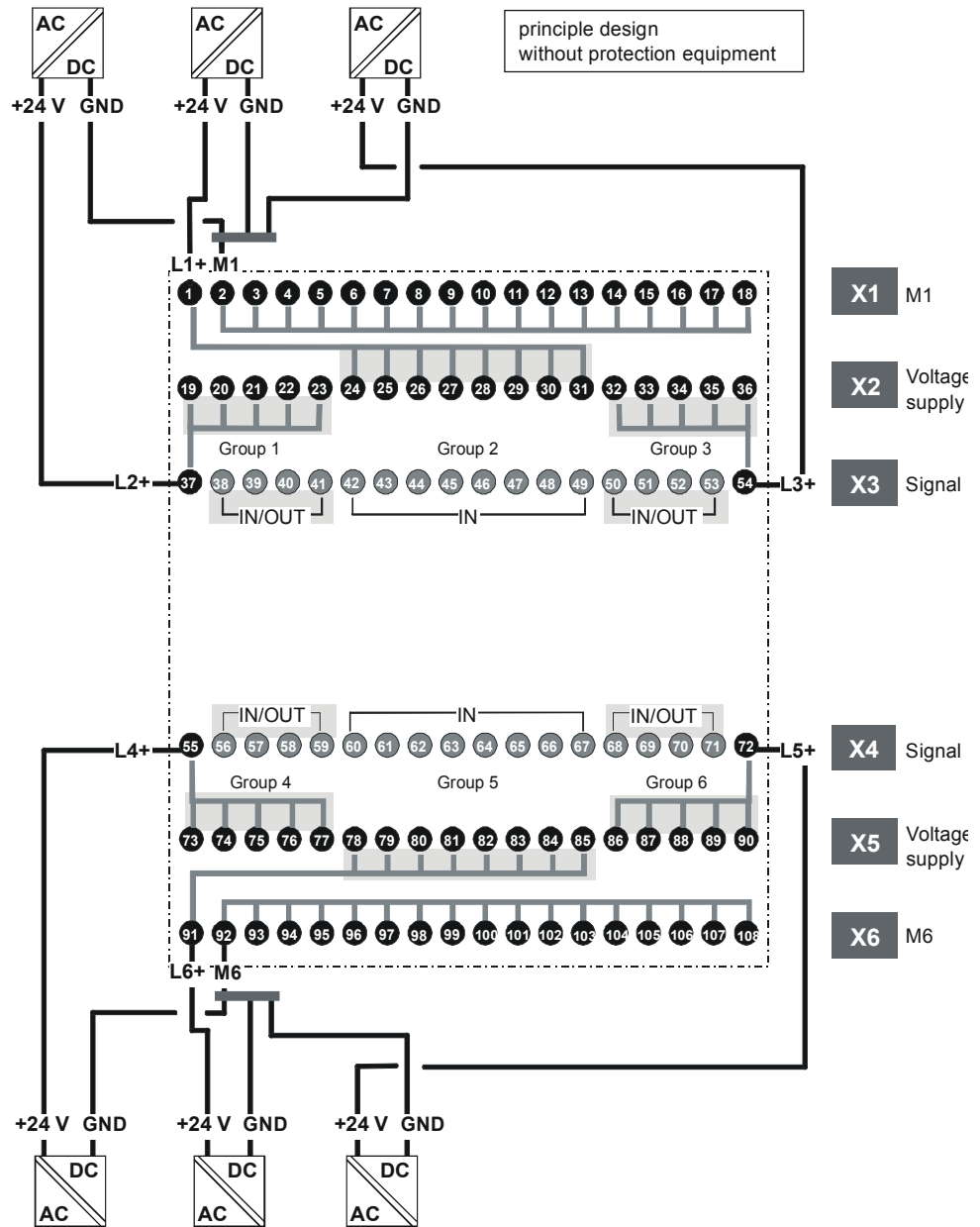
Output groups may be supplied through upstream switch elements (emergency off, manual switches, etc.).

#### **Feedback**

Always make sure the sensors are each supplied from the same power source as the module's associated I/O group.

Otherwise, when group power supply is disconnected, connected sensors could produce a feedback over the output transistors. This could destroy the module!

**Schematic Diagram of Input/Output Grouping**

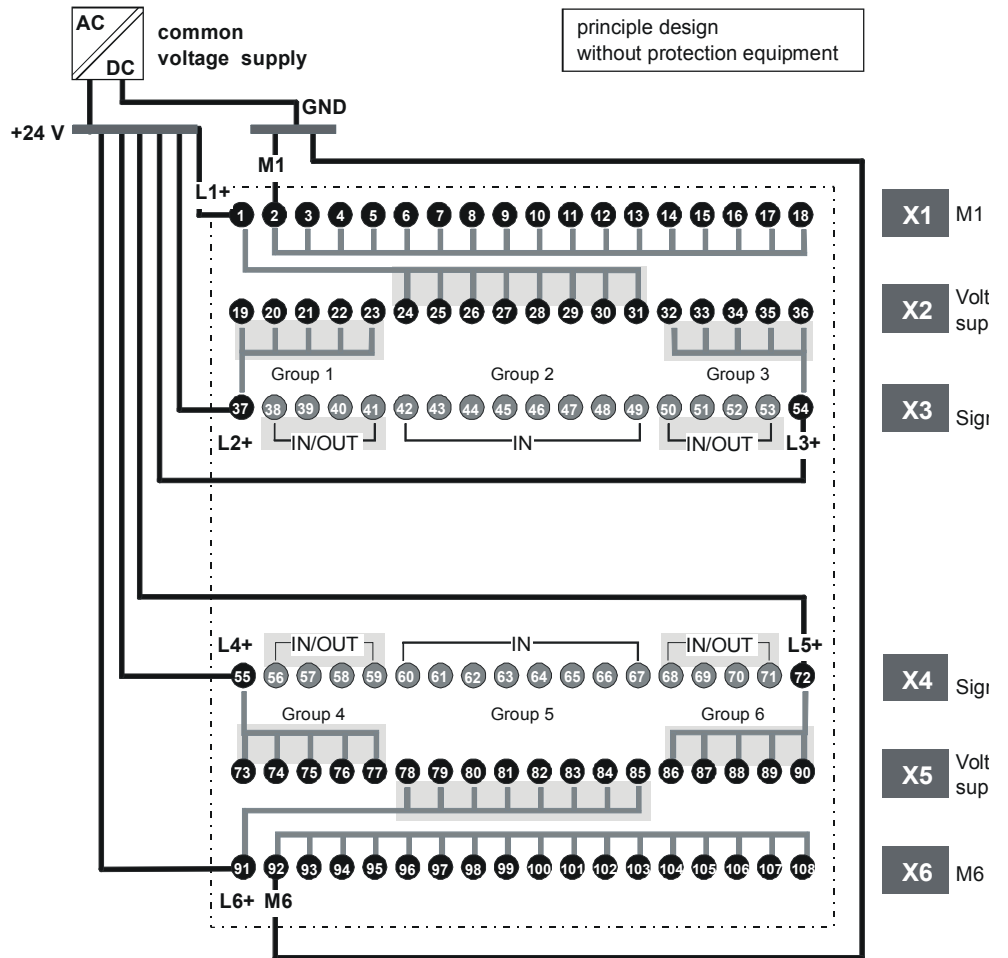


Group 1	IN / OUT 1-4	<i>Bemessungsspannung für erhöhte Isolation nach</i> <i>Rated voltage for increased isolation defined by</i> EN 61131-2 0 ... 50 V
Group 2	IN 5-12	
Group 3	IN / OUT 13-16	
Group 4	IN / OUT 17-20	
Group 5	IN 21-28	
Group 6	IN / OUT 29 -32	

2VF100007DG00.cdr

**Without Grouping**

Wird auf die Gruppenbildung bei der Spannungsversorgung verzichtet, sind vom Anwender die im folgenden Bild dargestellten Verbindungen herzustellen.  
 Without grouping of the voltage supply, the user has to build the following connection.



2VF100008DG00.cdr

### Digital Inputs, Positive-Switching

The digital inputs are positive-switching type 1 inputs for 3-conductor sensors. They are designed for input voltages of 24 V nominal. The inputs are transmitted cyclically to the CPU. An open input is interpreted as static 0 (LOW).

#### Pulse recognition and interference suppression

Inputs are read cyclically. Pulses < 100 µs are hardware suppressed. The sampling interval can be parameterised by software. The shortest possible sampling interval is 250 µs.

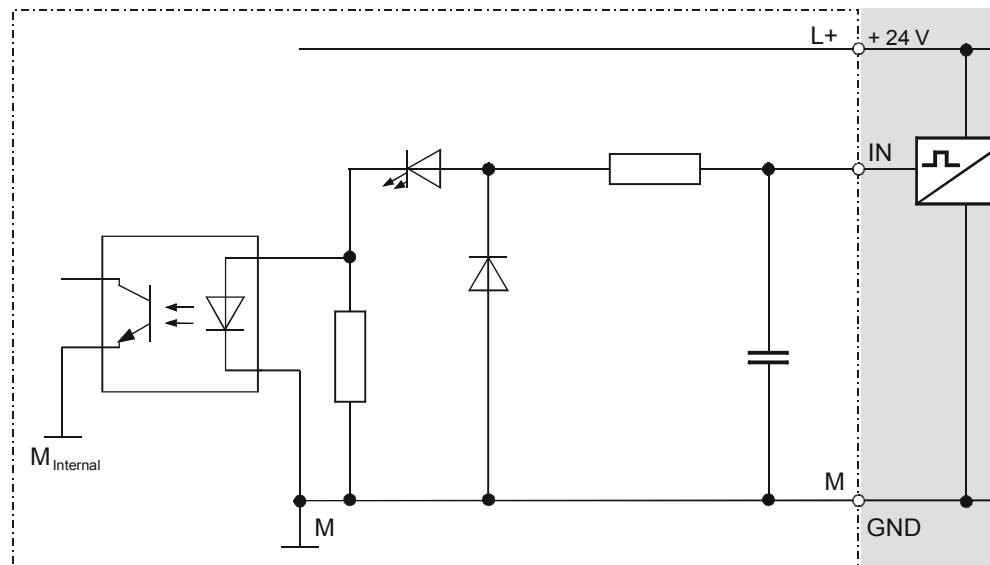
If pulses are to be detected reliably they must be longer than the sampling interval stipulated by software.

Multiple sampling can be programmed in order to suppress spurious pulses. Sampling interval and multiple sampling (filtering) can be activated in groups of 32 inputs each.

**Note:**  
This function is available only for C applications at present. The filter is permanently set to conform to IEC 61131.

**Operating status** The status of each input is indicated by a yellow operating status LED on the front panel of the module. The LEDs are spatially assigned to the supply terminals. An LED lights when its associated input is activated (logical 1 / HIGH).

### Block diagram of input

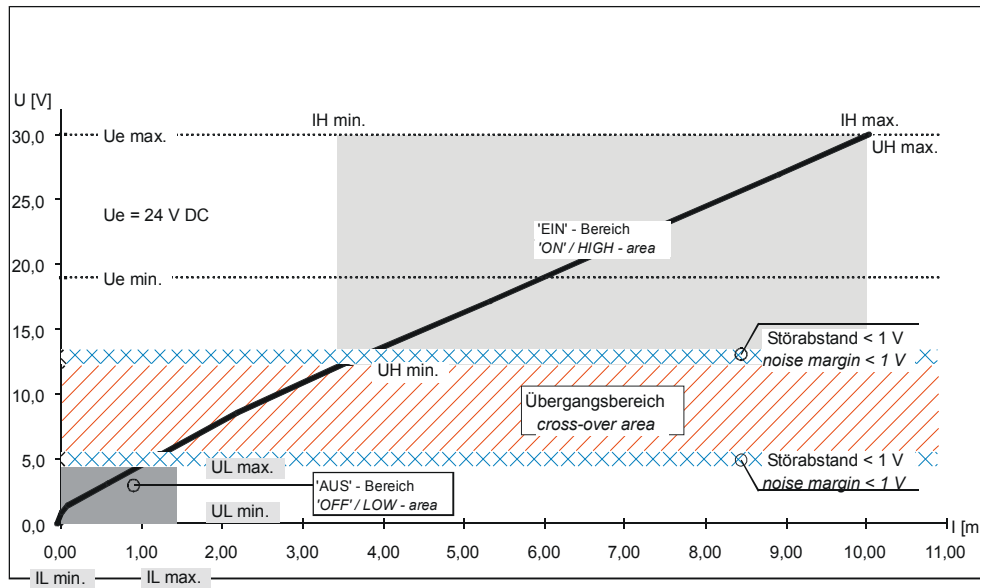


2VF100009DG00.cdr

**Digital Inputs Data**

<b>Module data</b>	
Number of inputs	16 (max. 32)
Line lengths:	<p>Allow for voltage drop when choosing conductor cross-section, otherwise no restrictions in practice.</p> <p>Observe all relevant local regulations and the requirements of EN 6 1131-3.</p> <p>Please consult manufacturer regarding lightning hazard</p>
<p>in switchgear cabinet</p> <p>dedicated I.v. wiring</p>	
Rated load voltage L+ Reverse voltage protection	24 VDC (SELV) yes
Electrical isolation	yes (optical isolator) in groups
Status display	yes, yellow LED for each input
Alarms	definable according to software
Input delay	parameterisable by software
Input capacitance	< 10 nF

Digital-input operating areas



Eingangsspannung (DC) der externen Stromversorgung <i>Input voltage (DC) of extern power supply</i>		
$U_e$	24 V	Bemessungsspannung / <i>rated voltage</i>
$U_{e \text{ max.}}$	30 V	oberer Grenzwert / <i>upper limit</i>
$U_{e \text{ min.}}$	19,2 V	unterer Grenzwert / <i>lower limit</i>
Grenzwerte für '1' Signal für die 'EIN'-Bedingung <i>Limit for '1' signal for the 'ON'-condition</i>		
$U_{H \text{ max.}}$	30,0 V	obere Spannungsgrenze / <i>upper voltage limit</i>
$I_{H \text{ max.}}$	10,0 mA	obere Stromgrenze / <i>upper current limit</i>
$U_{H \text{ min.}}$	13,5 V	untere Spannungsgrenze / <i>lower voltage limit</i>
$I_{H \text{ min.}}$	3,5 mA	untere Stromgrenze / <i>lower current limit</i>
Grenzwerte für '0' Signal für die 'AUS'-Bedingung <i>Limit for '0' signal of the 'OUT'-condition</i>		
$U_{L \text{ max.}}$	5,5 V	obere Spannungsgrenze / <i>upper voltage limit</i>
$I_{L \text{ max.}}$	1,5 mA	obere Stromgrenze / <i>upper current limit</i>
$U_{L \text{ min.}}$	0 V	untere Spannungsgrenze / <i>lower voltage limit</i>
$I_{L \text{ min.}}$	0 mA	untere Stromgrenze / <i>lower current limit</i>

2VF100010DG00.cdr

## Digital Outputs, Positive-Switching

**Warning !**

The module can be destroyed by overvoltages > 32 V and / or feedback.  
Risk of fire!

Each digital output is also usable as an input. See description under 'Digital Inputs' if using as input.

**Outputs**

The outputs are of positively-switching 24 volt type (two-conductor). Maximum output current per output is 500 mA. The outputs have a common earth (GND) when operating in groups. Power is supplied separately from the supply for the modular electronic circuit (see 'Connection Assignment'). The outputs switch automatically to '0' (LOW) if there is no available data link to the CPU or if the module's internal supply is insufficient.

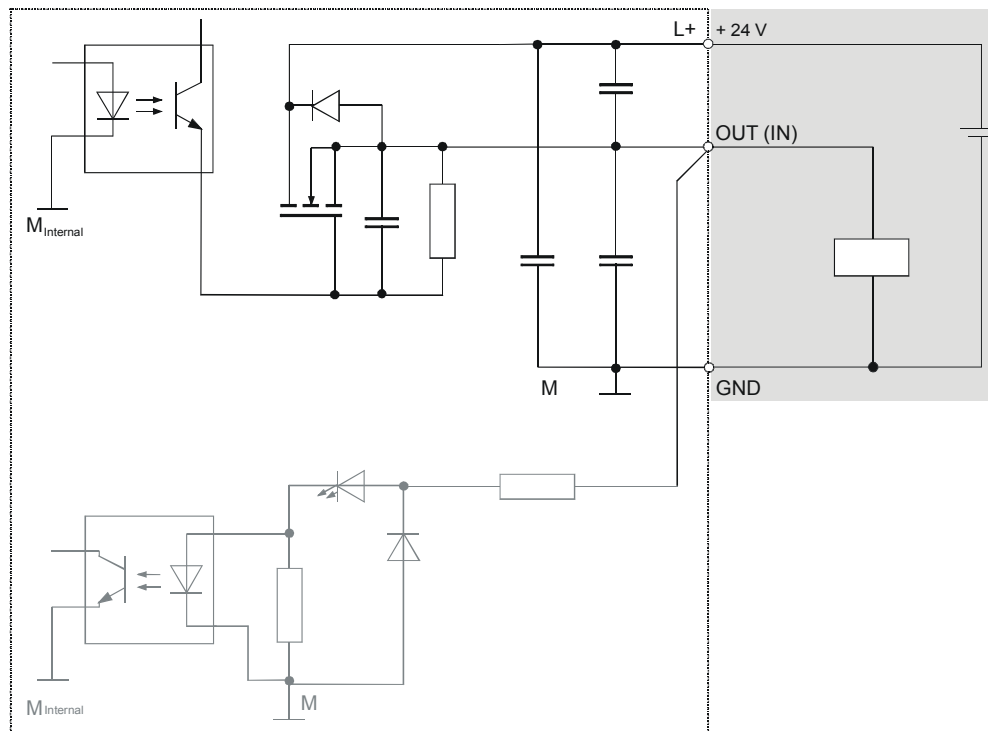
**Protected output**

All outputs are protected by an incorporated current-limiting circuit and a thermal overload protection circuit. If overloaded, the affected output switches off. The output can be re-activated by program on elimination of the overload and thermal cooling. A high-speed de-excitation feature having a terminal voltage of 50 V protects all outputs against induced voltage peaks under inductive loads. The overload protection of non-involved outputs may respond prematurely if feedback or high-speed de-excitation give rise to thermal loads.

**Operating status**

The status of each output is indicated by a yellow operating status LED on the front panel of the module. The LEDs are spatially assigned to the supply terminals. A LED lights when its associated output is activated, logical '1' (HIGH).

**Block diagram of output**



2VF100011DG00.cdr

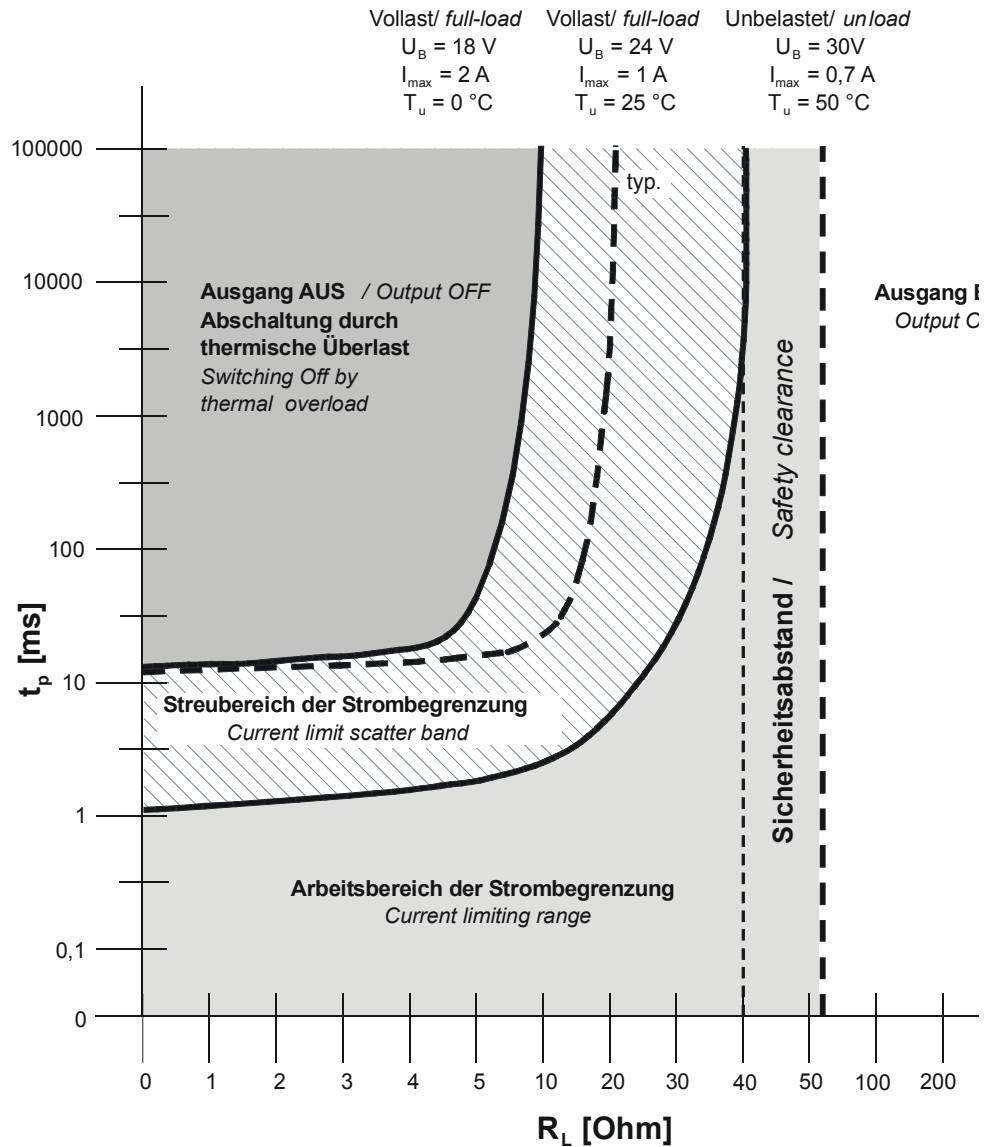
**Digital Outputs Data**

<b>Module data</b>	
Number of outputs	16 semiconductor outputs in 4 groups
Type of outputs	semiconductor, non-holding
Suppressor circuit for inductive loads	high-speed de-excitation 50 V terminal voltage (typical) to + 24 V
Power loss due to de-excitation	max. 0.5 watts per output max. 4 watts per module
Status display	yes, yellow LED for each output
Diagnostic function	yes, switching state can be read back at pin
<b>Load connection</b>	
Total loading (100%)	8 A (16 x 0,5 A)
Overload protection	yes, in event of thermal overload Responding of thermal overload protection may influence adjoining outputs
Short-circuit protection <sup>1)</sup> response threshold	yes, electronic current-limiting feature, min. 0.5 A, typically 0.9 A
1) Current is limited electronically. Responding of the short-circuit protection feature produces thermal overload and trips the thermal overload protection circuit..	
Output delay for '0' to '1' for '1' to '0'	max. 0,5 ms max. 0,5 ms
Output capacitance	< 20 nF
Rated voltage	+24 VDC
Voltage drop (at rated current)	< 0,5 V
Rated current for '1' signal	0,5 A
Leakage current for '0' signal	max. 0,1 mA
Total current of all outputs	max. 8 A (16 x 0,5)
Total current per group (horizontal mounting on vertical mounting plate)	max. 2 A (4 x 0,5)
Lamp load (+24 VDC)	max. 6 watts
Connection of two outputs in parallel to provide logic operation to increase performance	allowed not allowed
<b>Insulation resistance</b>	
Rated voltage	0 V <U <sub>e</sub> <50 V
Test voltage up to 2,000 m altitude	500 VDC

**Overload Reaction of Digital Outputs**

**Überlast-Verhalten der digitalen Ausgänge**

Overload-reaction of digital output



Innerhalb des Streubereichs der Strombegrenzung ist das Verhalten der Strombegrenzung undefiniert.  
*Within the current-limit scatter band the reaction of current limiting is undefined.*

2VF100021DG00.cdr

**Note:**

It is not possible to know for certain within the current limit scatter band whether the response will be to disconnect or to return to the working range. As a result, this state should be avoided!

The output may be re-activated by program following elimination of the overload and thermal cooling.

blank page

## Annex

---

### Environmental Protection

#### Emissions

CANtrol modules produce no harmful emissions when used as prescribed.

#### Disposal

CANtrol modules may be returned against an all-inclusive charge to Berghof Automationstechnik GmbH once they have completed their service life. We then arrange for the modules to be re-cycled.

---

### Maintenance / Upkeep

**Warning !**

Do not insert, apply, detach or touch connections when in operation! Destruction or malfunctioning may otherwise occur.  
Disconnect all incoming supplies before working on CANtrol modules; including those of connected peripherals such as externally supplied sensors, programming devices, etc.  
All ventilation openings must always be left unobstructed!

CANtrol modules are maintenance-free when used as prescribed.  
Clean only with a dry, non-linting cloth. Do not use cleaning agents!

---

### Repairs / Customer Service

**Warning !**

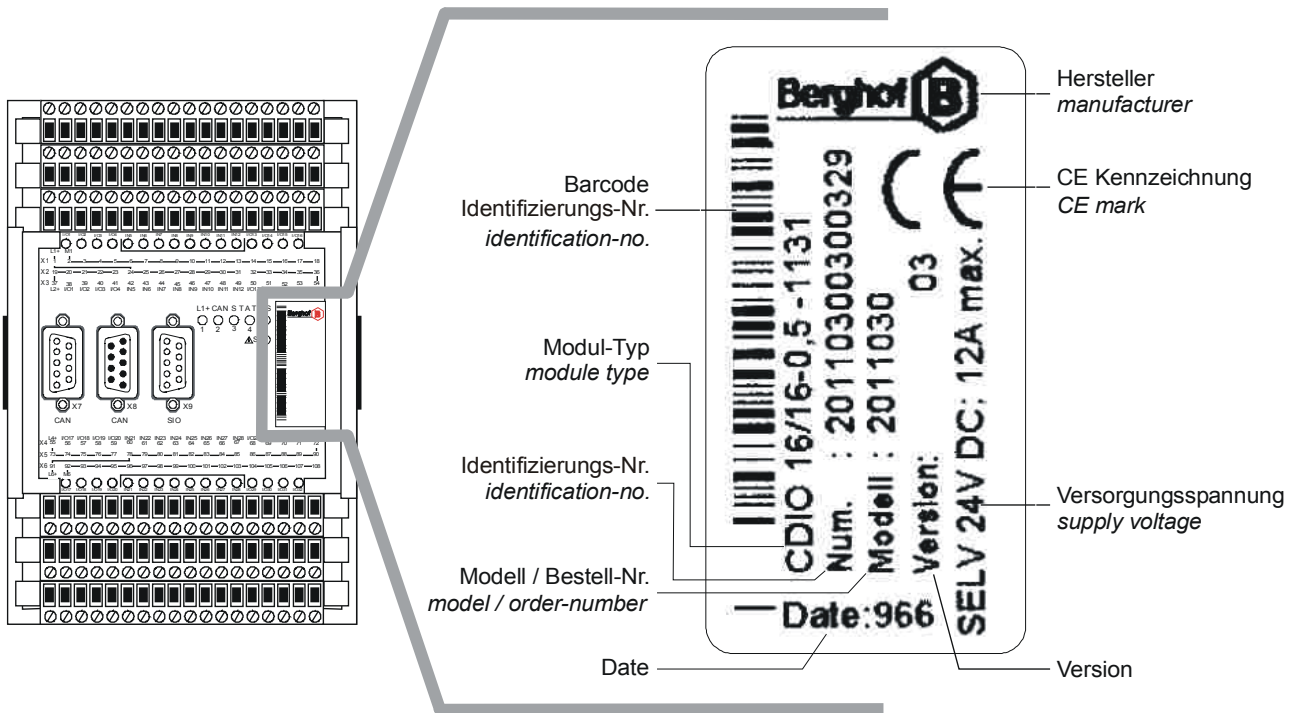
Repair work may only be carried out by Berghof Automationstechnik GmbH or their authorised service representatives

#### Warranty

Statutory warranty cover provided. Cover invalidated if device / product subjected to unauthorised attempted repairs or other intervention.

**Nameplate**

**Erklärungen zum Typenschild (Beispiel CDIO 16/16-0,5)**  
*nameplate descriptions (example CDIO 16/16-0,5)*



2VF100056DG01.cdr

- Module type** Plain-text name of module (z.B. CDIO 16/16-0,5).
- Num** This is the module's identification number. It begins with the leading digit '2', followed by the model No. The version supplied is specified by the next two digits. The last six digits make up the individual module number, which the production department issues in serial form as each module is produced.
- Model** When ordering a module, indication of this number will suffice. The module then supplied will be in the prevailing current hardware and software version.
- Version** defines the ex-factory version of module supplied.
- Date** internal coding.
- Barcode** identification number represented in 2/5 Standard Code.

**Note:**  
 The "Version" field (version supplied) specifies the ex-works condition of the module as delivered. When replacing a module, the user can use the CNW Tool to read the current software version off the module supplied and then reload 'his own' project-specific software version if necessary. You should file 'old' software levels (SW version, node ID, baud rate, etc.) in your project documentation for this purpose.

---

## Addresses and Literature

### Addresses

<b>CiA</b>	CAN in Automation; international manufacturers' and users' organisation for users of CAN in automation:  CiA - CAN in Automation e.V. Am Weichselgarten 26 D-91058 Erlangen  e-mail: <a href="mailto:headquarters@can-cia.de">headquarters@can-cia.de</a> <a href="http://www.can-cia.de">http://www.can-cia.de</a>
<b>DIN-EN Standards</b>	Beuth Verlag GmbH 10772 Berlin or VDE-Verlag GmbH 10625 Berlin
<b>IEC Standards</b>	VDE Verlag GmbH 10625 Berlin or search on Internet <a href="http://www.iec.ch/">http://www.iec.ch/</a>

### Standards / Literature

<b>IEC61131-1/EN61131-1</b>	Stored-program controllers Part 1: General information
<b>IEC61131-2/EN61131-2</b>	Stored-program controllers Part 2: Equipment requirements and tests
<b>IEC61131-3/EN61131-3</b>	Stored-program controllers Part 3: Programming languages
<b>IEC61131-4/EN61131B11</b>	Stored-program controllers Supplementary sheet 1: User guidelines
<b>EN 50081 Part 1+2</b>	EMC law: emitted interference
<b>EN 50082 Part 1+2</b>	EMC law: noise immunity
<b>ISO/DIS 11898</b>	Draft International Standard: Road vehicles - Interchange of digital information - Controller Area Network (CAN) for high-speed communication.
<b>EN 954-1</b>	Safety of machinery: Safety-related parts of controllers (Part 1)
<b>Literature</b>	A variety of specialist publications on the subject of the CANbus is available from specialist dealers and through the CiA user organisation.

**Note:**

For further literature details please contact our Technical Support.

blank page