

CyPro User Manual

rev. 33

(applies to CyPro v2.6.4 and later)



© 1998-2011 Cybrotech Ltd

Cybrotech Ltd
14 Brinell Way, Harfreys Industrial Estate
Great Yarmouth, Norfolk, Nr31 OLU - UK
tel: +44 (0)1157 149 991
www.cybrotech.co.uk
info@cybrotech.co.uk

Index

Index.....	1
Introduction.....	4
System overview.....	4
Hardware requirements.....	5
Installation.....	5
Communication.....	6
Serial connection.....	6
TCP/IP connection.....	7
User interface.....	8
Main window.....	8
Standard toolbar.....	8
Program toolbar.....	8
Communication toolbar.....	9
Project tree.....	9
Status bar.....	9
Menu.....	10
File.....	10
Edit.....	11
Format.....	11
View.....	11
Project.....	12
Program.....	12
Tools.....	13
Edit window.....	13
Online monitor.....	14
Identify Modules.....	15
Data manager.....	16
Multisend.....	17
Programming.....	18
Hardware.....	18
Expansion modules.....	18
Hardware setup.....	19
Variables.....	19
Naming convention.....	19
Allocation.....	19
Retentive variables.....	21
EE variables.....	22
I/O variables.....	23
Internal variables.....	24
Timers.....	25
Pulse timer.....	25
On-delay timer.....	26
Counters.....	26
Visibility in alc file.....	26
Refresh processing.....	27
Scan overrun.....	27
Instruction list.....	28
Structured text.....	29
Assignment statements.....	29
Expressions.....	29
Operators.....	29
Expression evaluation.....	30
Data type conversions.....	30
Multiline expressions.....	31
Conditional statements.....	31
if...then...else.....	31
case...of.....	32

Iteration statements	33
for...do	33
while...do	33
Operator panel	35
General	35
Print functions	35
OP keys	37
Masks	38
Handling masks from plc program	41
Serial ports	43
General	43
Free-programmable mode	43
Port selection	43
Prepare transmit message	43
Transmit	44
Receive	45
Parse received message	47
High speed counter	48
General	48
One phase counting	48
Two phase counting, single precision	49
Two phase counting, quad-precision	49
High speed counting	50
High speed action	50
Zero reset	53
Zero detect	53
Real-time clock	54
General	54
I/O variables	54
Networking	55
Ethernet connection	55
Connection options	56
1. Direct connection	57
2. Local area connection	57
3. Wide area connection using static IP or DynDNS	57
4. Wide area connection using relay server	59
Ethernet sockets	60
1. Periodic 1s	60
2. Periodic 10s	60
3. On-request	60
4. On-change	61
Socket examples	61
Event-driven action	61
Synchronized value	61
MODBUS connection	62
Additional features	63
NAD alias	63
Password protection	64
Command-line options	65
CyBro tutorial	67
Your first CyBro program	67
Step one: define problem	67
Step two: define hardware	67
Step three: allocate variables	67
Step four: write code	68
Step five: send and run	68
Step six: new frontiers	69
Appendix	71
Data types summary	71
Elementary	71

Input/Output	71
Timer	71
Internal variables	72
Instruction list summary	73
Instructions	73
Allowed type conversions	74
Allowed type combinations	74
Structured text summary	75
Operators	75
Flow control	75
Edge detect functions	76
Cast functions	76
Display functions	76
Com2 functions	77
High speed counter functions	77
Special functions	78
Character set	79
Keyboard shortcuts	80
Common	80
Text editor	80

Introduction

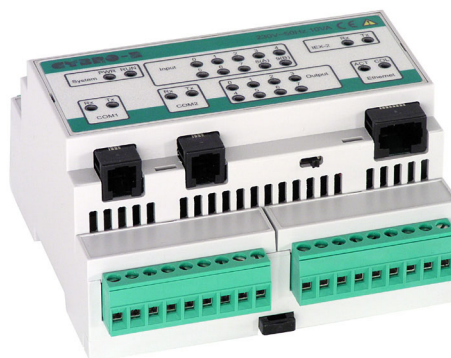
System overview

CyPro is a software package for programming CyBro-2 controllers. It runs under Microsoft Windows 2000/XP/Vista/Win7, and also under Linux/Wine emulator. CyPro is fully featured IDE (integrated development environment) containing editor, compiler and on-line monitor.



CyBro-2 is the generic name for a group of controllers, based on the same technology. All controllers are compatible, although some features may be different.

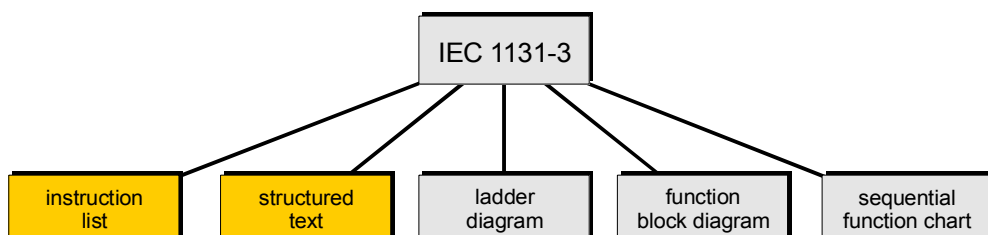
Each CyBro-2 is labeled with unique 6-digit serial number, also used as communication network address (NAD).



CyBro-2 has four independent communication ports:

COM1	A-bus, Modbus RTU, or freely programmable serial port
COM2	A-bus, Modbus RTU, or freely programmable serial port
ETH	A-bus and Modbus TCP
IEX-2	IEX-2/CAN and A-bus

CyPro is based on IEC 1131-3, and implements instruction list and structured text programming, extended with several visual tools.



Hardware requirements

Any PC capable of running MS Windows XP is sufficient. CyPro occupies about 8Mb on disc.

To connect CyBro and PC, standard serial (RS232) or Ethernet port is required. USB-to-serial converters are generally supported, although some devices may not work.

Installation

To install CyPro, start the installation archive and follow the instructions. Recommended install directory is C:\Program Files\CyPro. If older versions are needed, it is recommended to append version number to the installation directory, e.g. C:\Program Files\CyPro v201.



Installation does the following:

- unpack files into specified directory
- create start menu group and icons
- set association to .cyp file type
- set CyPro language and communication port

No file is copied to windows directory. No system files are replaced or changed. Default directory for user files is C:\Program Files\CyPro\Project, although it is recommended to keep projects in \MyDocuments directory.

To upgrade CyPro, install new release into the same directory, without uninstalling previous version. User settings will be preserved.

Once CyPro is upgraded, it is required to also upgrade the CyBro system software (kernel). To do this, start **Tools/Kernel Maintenance**, load and send **kernel.bin**.

CyPro and kernel version should always match. CyPro release always comes with kernel file.

To uninstall CyPro, start **Control Panel, Add/Remove Programs**, select CyPro and press **Add/Remove** button.




Communication

CyBro-2 has two serial (RS-232) and one Ethernet port. All ports are independent, and may operate at the same time.

To establish communication between CyPro and CyBro, the following steps should be made:

- connect CyBro
- open new project
- adjust communication settings (serial or Ethernet)
- open **Hardware setup (F5)**
- start **Autodetect (Alt-A)**

If communication is not established, check the cable indicator icon on the status bar:

-  Communication port is not available. The reason may be a wrong port number, or the port may already be used by another application.
-  Communication port is open, but the cable is not detected. The reason may be a wrong port number, or communication cable. Connect cable to another DB-9 connector, or use another cable.
-  Cable is properly detected. If communication is still not working, check network address and connection.

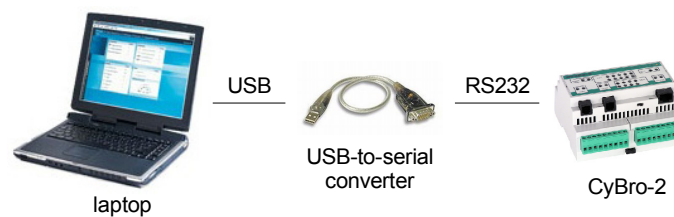
If more than one CyBro is connected through the serial interface, NAD autodetection will not work. In that case, please enter the network address of each CyBro manually.

If the option **Synchronize program with PLC** is activated, commands **Start** and **Monitor** will automatically compile and send the current project to the CyBro.

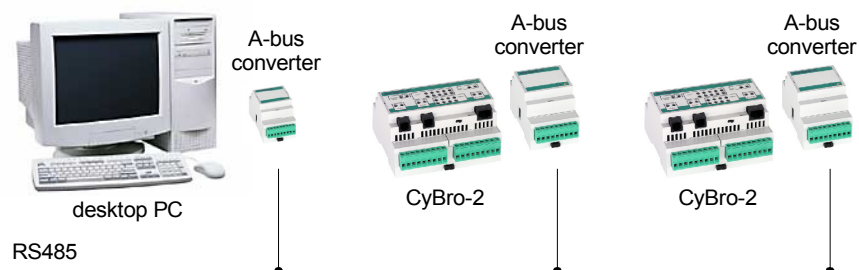
To connect PC and CyBro, a number of connection options are available.

Serial connection

Serial RS232 connection using serial cable or USB-to-serial converter:



Serial RS-485 connection using CAD-232-A2 converters:

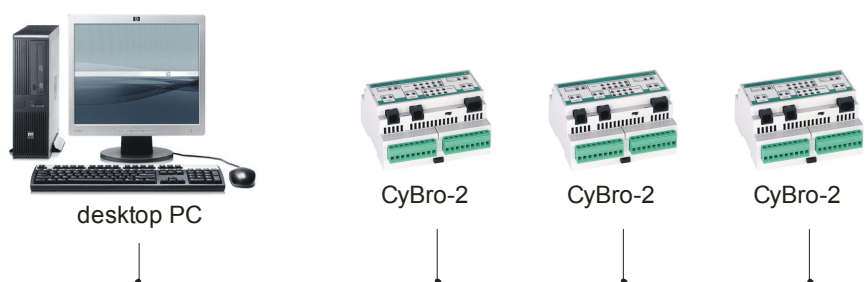


TCP/IP connection

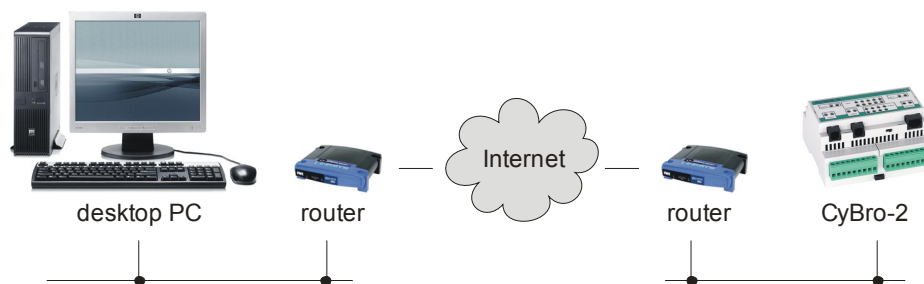
Direct connection:



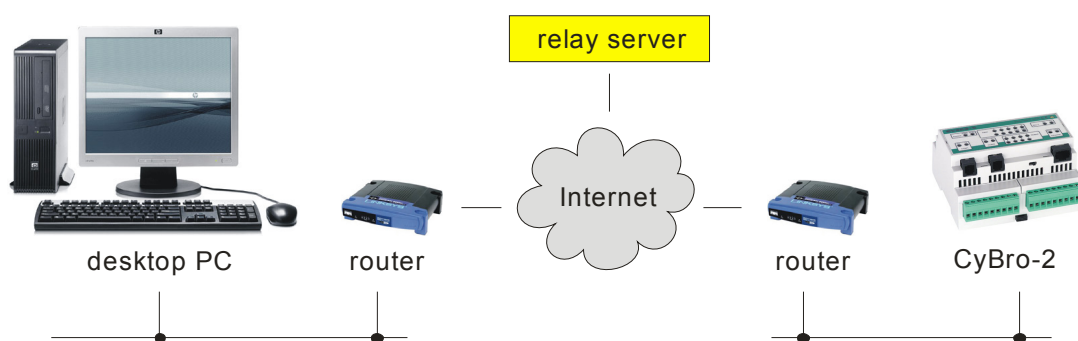
Local area network connection:



Wide area connection through Internet:



Wide area connection through Internet using relay server:

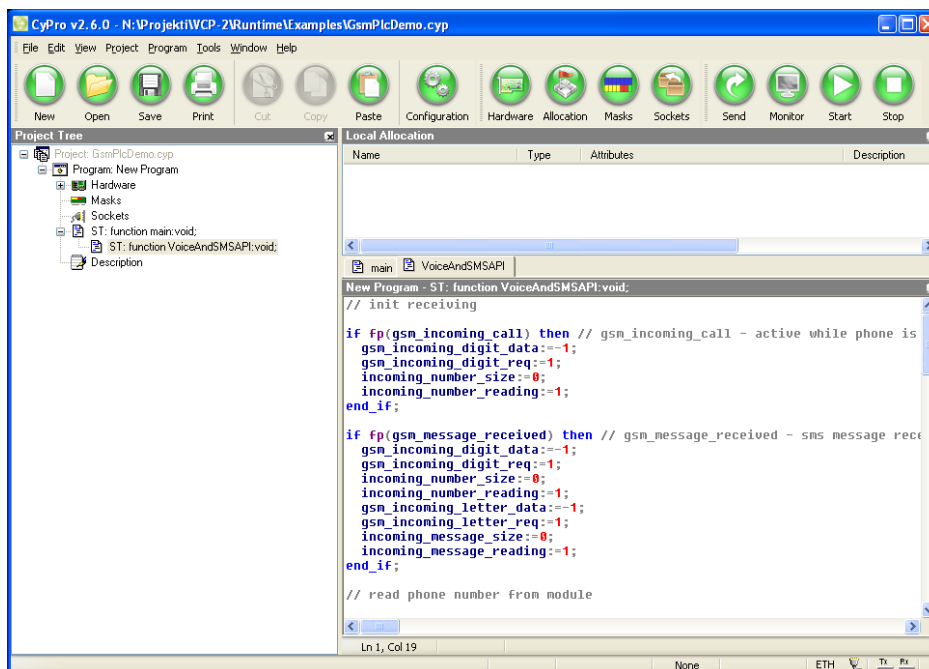


For more details how to setup the network, check Networking section.

User interface


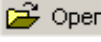


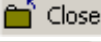

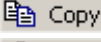
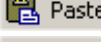
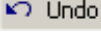
Main window

CyPro consists of editor, toolbars and status bar. Default screen setup is shown below:







Each component can be docked or floating. To undock, drag the component by the left vertical line over the edit area. To dock it again, drag window to main window border.


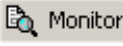
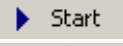
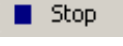
Standard toolbar

- | | | |
|-------------------------------------------------------------------------------------|-------|----------------------------------------------------------------------------------------------|
|  | New | Create a new empty project |
|  | Open | Open an existing project (Ctrl-O) |
|  | Save | Save current project (Ctrl-S) |
|  | Print | Print current project (Ctrl-P) |
|  | Close | Close current project |
|  | Cut | Remove the selection and place it on the clipboard (Ctrl-X) |
|  | Copy | Copy the selection onto the clipboard (Ctrl-C) |
|  | Paste | Insert the content of the clipboard at the cursor, replacing any selection (Ctrl-V) |
|  | Undo | Undo the most recent editing action (Ctrl-Z) |



Program toolbar

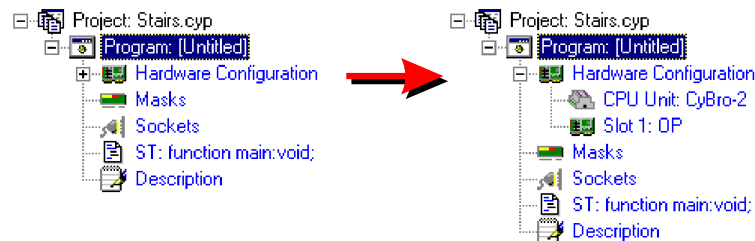
- | | | |
|-------------------------------------------------------------------------------------|------------|------------------------------------------------------------|
|  | Hardware | Open the Hardware Setup dialog box (F5) |
|  | Allocation | Open the Allocation Editor dialog box (F6) |
|  | Masks | Open the Mask List editor (F7) |
|  | Sockets | Open the Socket List editor (F8) |

Communication toolbar

	Send the current project to the CyBro (F9)
	Open the on-line Variable Monitor (F10)
	Start CyBro program (F11)
	Stop CyBro program and turn off all outputs (F12)

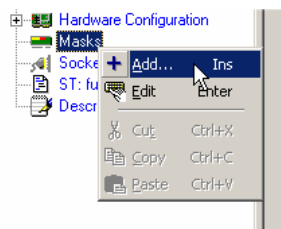
Project tree

Project tree displays all the project parts hierarchically. Left click on  will expand the tree, and allow a more detailed view. To collapse tree node click .






Right clicking any component opens its context sensitive pop-up menu. Depending on type, it is possible to **Add**, **Edit**, **Delete** or change **Properties** of the selected component.

For example, to define a new mask right click on **Masks** and choose **Add**.



Status bar

Status bar shows various information about communication and connected CyBro. Default status bar position is at the bottom of the main window.

Modified	192.168.1.4	6512	Stop	ETH 	26 ms	 
project status	IP address	A-bus address	PLC status	com port status	delay	Tx/Rx indicators

System message (on the left) show result of preceding operation.

Project status is displayed if the current project is not saved. It reflects changes in source, allocation, mask, socket, data manager or monitor list.

IP address shows CyBro IP address.

A-bus address shows CyBro A-bus address. Right click to select another or enter a new address.

PLC status shows:

Off-line	CyBro is not responding.
Run	CyBro is on-line and running.
Stop	CyBro is on-line, stopped. Outputs are inactive and program is not executing.
Pause	CyBro is on-line, paused. Outputs remain active, but program is not executing.
Error	CyBro is on-line, some error occurred. Error codes are listed in the appendix. To clear the error press Stop .
Loader	CyBro is on-line, but system software (kernel) is not active. Start Kernel Maintenance and send a new kernel.
Busy	Unexpected messages are received.

Com port status indicates whether communication cable is properly connected:



OK

communication port is ok, but cable is not connected

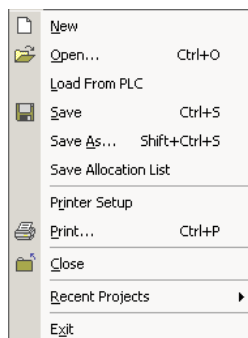
communication port is already used by another application

Delay shows time between sent and received message, in milliseconds.

Communication indicators show activity, green is transmit (Tx), red is receive (Rx).












Menu

File







New	Create a new project
Open	Open an existing project
Load From PLC	Load project from the PLC
Save	Save the current project
Save As	Save the current project under a new name
Save Allocation List	Save allocation list to use with CyBro OPC or UniOP Designer
Printer Setup	Select printer and printer options
Print	Print the current project
Close	Close the current project
Recent Projects	Open recently opened project
Exit	Exit

Edit

	Undo	Ctrl+Z
	Redo	Shift+Ctrl+Z
	Cut	Ctrl+X
	Copy	Ctrl+C
	Paste	Ctrl+V
	Delete	
	Select All	Ctrl+A
	Find...	Ctrl+F
	Find Next	F3
	Find Previous...	Shift+F3
	Replace...	Ctrl+R
	Go to Line Number...	Ctrl+G
	Properties...	Alt+Enter

Undo	Cancel last action
Redo	Cancel last Undo operation
Cut	Delete the selection and put it on the clipboard
Copy	Copy the selection onto the clipboard
Paste	Insert text from the clipboard to the insertion point
Delete	Delete the selection
Select All	Select a whole document
Find	Find the specified text
Find Next	Find the next occurrence of the specified text
Find Previous	Find the previous occurrence of the specified text
Replace	Find the specified text and replace it
Go to Line Number	Move insertion point to the specific line in the text
Properties	Show properties of the selected project module

Format

	Indent Block	Shift+Ctrl+I
	Unindent Block	Shift+Ctrl+U
	Comment/Uncomment Code	Shift+Ctrl+C
	Insert Identifier	Ctrl+Space

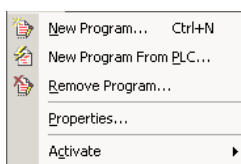
Indent Block	Indent block of code/text
Unindent Block	Unindent block of code/text
Comment/Uncomm. Code	Comment/Uncomment selected part of code
Insert Identifier	Open the list of identifiers to insert in the code

View

<input checked="" type="checkbox"/>	Project Tree	Ctrl+Alt+T
<input checked="" type="checkbox"/>	Local Allocation Editor	Ctrl+Alt+A
<input checked="" type="checkbox"/>	Editor Tabs	
<input type="checkbox"/>	Compiler Messages	Ctrl+Alt+C
<input checked="" type="checkbox"/>	Standard Toolbar	
<input checked="" type="checkbox"/>	Program Toolbar	
<input checked="" type="checkbox"/>	Communication Toolbar	

Project Tree	Show Project Tree
Local Allocation Editor	Show Local Allocation Editor
Editor Tabs	Show Editor Tabs
Compiler Messages	Show Compiler Messages
Standard Toolbar	Show Standard Toolbar
Program Toolbar	Show Program Toolbar
Communication Toolbar	Show Communication Toolbar

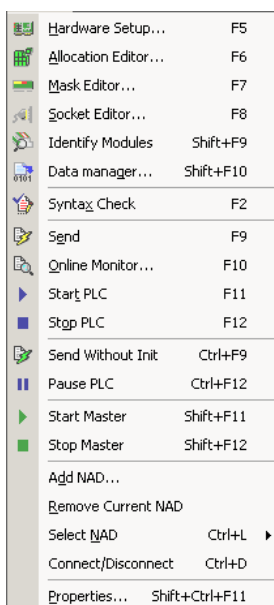
Project



New Program
New Program From PLC
Remove Program
Properties

Create **New Program** in the current project
Load program from PLC into the current project
Remove program from the current project
Show properties of the current project

Program



Hardware Setup
Allocation Editor
Mask Editor
Socket Editor
Identify Modules
Data Manager

Open **Hardware Setup** dialog box
Open **Allocation Editor** dialog box
Open **Mask List** editor
Open **Socket List** editor
Identify IEX modules and individual inputs/outputs
Backup/restore a set of plc variables to/from PC

Syntax Check

Check the current source for errors

Send
Online Monitor
Start PLC
Stop PLC

Send current program to CyBro
Read/write plc variables on-line
Start CyBro program
Stop CyBro program and turn off all outputs

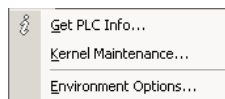
Send Without Init
Pause PLC

Send current program to CyBro, keep outputs and plc variables
Pause CyBro program, keep outputs active

Add NAD
Remove Current NAD
Select NAD
Connect/Disconnect
Activate
Properties

Add new network address to the current program
Remove current NAD from the current program
Select current network address for the active program
Connect/Disconnect communication port
Activate program
Show program properties

Tools

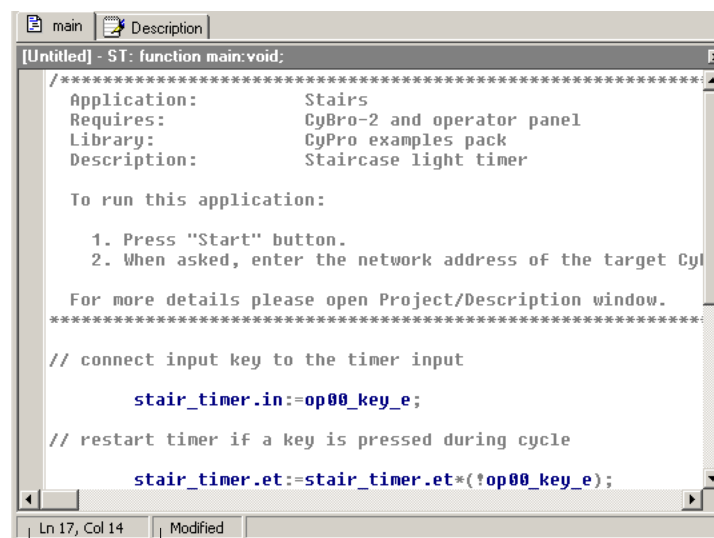


Get PLC Info	Retrieve various PLC information
Kernel Maintenance	Upload a new kernel to CyBro
Environment Options	Change configuration options of CyPro environment
Communication Monitor	optional, A-bus monitor tool, enabled by password

Edit window

Edit window is used to type and edit PLC program. Each window represents a single function.

Instructions may be written in both available languages, instruction list and structured text. They cannot be mixed in the same window. To change current language, select **Edit/Properties**.



```

[Untitled] - ST: function main: void;
/*****
Application:      Stairs
Requires:        CyBro-2 and operator panel
Library:         CyPro examples pack
Description:     Staircase light timer

To run this application:

1. Press "Start" button.
2. When asked, enter the network address of the target Cy

For more details please open Project/Description window.
*****/
// connect input key to the timer input
    stair_timer.in:=op00_key_e;

// restart timer if a key is pressed during cycle
    stair_timer.et:=stair_timer.et*(!op00_key_e);

```

Cursor position File status

Editor content is dynamically syntax highlighted. Variables, constants, functions and other language elements are displayed in different colors. To select different color scheme or customize colors, use **Tools/Environment Options/Colors**.

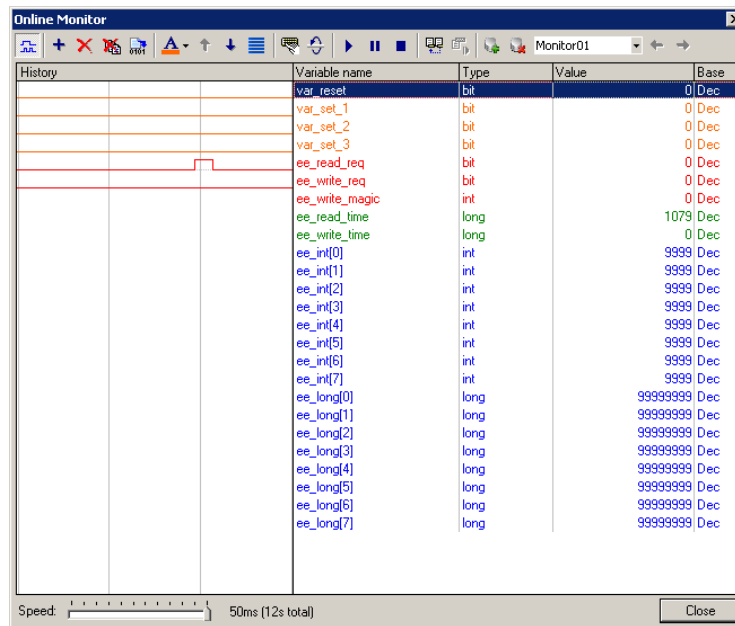
Code helper is useful feature of the structure text editor. To display a list of allocated variables and available functions, press **Ctrl-Space**.

CyPro editor allows multiple level of undo operation. Deleted or changed text can be restored by pressing **Ctrl-Z** key combination, even after other completed operations. The number of edit actions that can be undone is limited only by the available memory space, and is usually quite sufficient.

To observe structure of .cyp file, open an example file.

Online monitor

Online monitor is a dialog box designed to display current values of CyBro variables. To open it, select **Program/Online Monitor**, click **Monitor** icon on communication toolbar or press key **F10**.

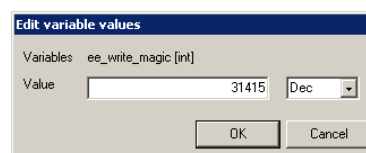


To insert variables click **Add** button or press **Insert** key. Dialog box **Variable Insert** will appear. Select desired variables and press **OK**. To select continuous block of variables, press **Shift** key. Multiple selections are possible when **Control** key is pressed. To rearrange list, select variable and click **Move Up** or **Move Down**, or press **Control** and move variable with **Up** or **Down** (arrow) keys.

Variable monitor allows unlimited number of sets. To quick access first five press **Alt-1** to **Alt-5**.

Monitor values are updated approximately every 100ms. History scrolling speed can be changed using **Speed** slider.

To change variable value double-click it, press the **Edit Variable properties** button or select **Properties** option in the right-click menu.

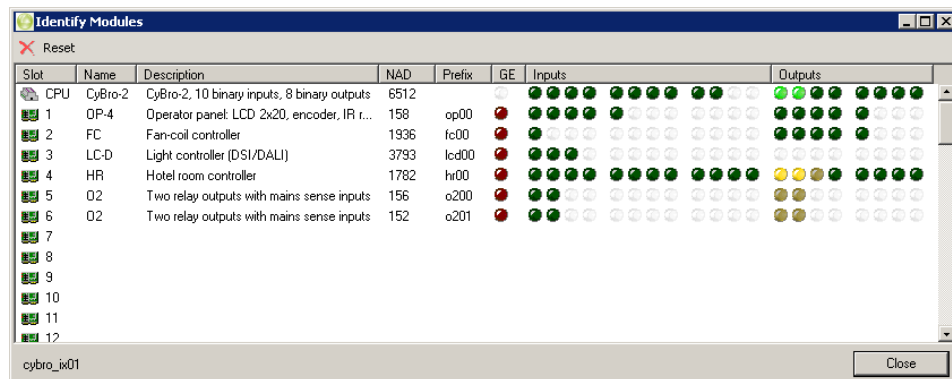


Enter new value and press **OK**. The entered value is sent to CyBro and then read back, so the monitor displays the actual value. Note that if CyBro is running and the program changes the variable, its value will be immediately rewritten.

To toggle value of a bit variable, press **Space** key.

Identify Modules

Identify Modules is a tool used to identify IEX modules and individual inputs/outputs. Because of change/reset operation, tool may be used by a single person.



Each LED represents a single digital input or output. When mouse is positioned over LED, signal name is shown in the bottom left corner.

Input/output LED colors are defined according to the following table:

LED	current level	changed
	0	no
	1	no
	0	yes
	1	yes

General error (GE) condition is defined as:

LED	description
	module is operating properly
	error, module is not operative

In case of general error, use on-line monitor to find the problem cause (module timeout error, bus error or program error).

To identify an unknown input:

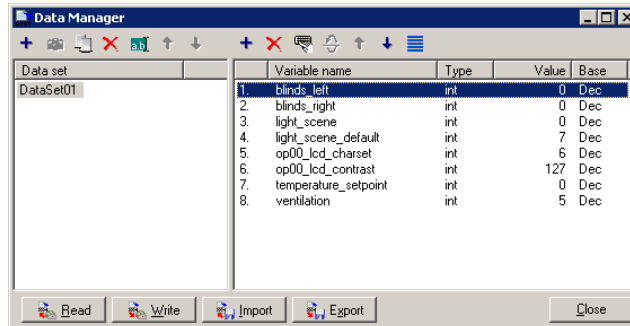
1. Reset "Identify Modules".
2. Walk to unknown switch and press for a second.
3. Walk back and search the yellow LED.

To identify an unknown output:

1. Click output LED and check outputs to activate.

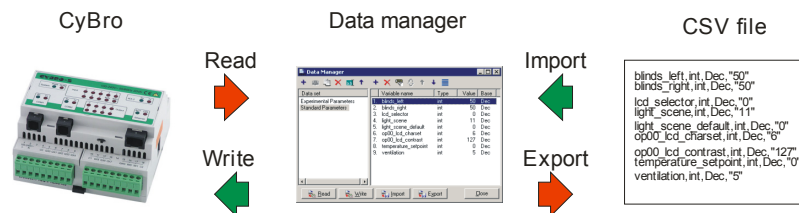
Data manager

Data manager is a tool used to transfer data between PLC and PC. It allows user to store plc configuration or to export data to for statistic or graphic analysis. Variables are organized by data sets. Data set is user-defined list of plc variables. It contains variables and their respective values.



The following commands are available:

Read	Read variables from plc to data manager.
Write	Write variables from data manager to plc.
Import	Import variables from csv file to data manager.
Export	Export variables from data manager to csv file.



The comma-separated values (or CSV) file is a text-only file that stores tabular data. Each row contains variable name, type, base and value.

```

blinds_left,int,Dec,"50"
blinds_right,int,Dec,"50"
light_scene,int,Dec,"11"
light_scene_default,int,Dec,"0"
op00_lcd_charset,int,Dec,"6"
op00_lcd_contrast,int,Dec,"127"
temperature_setpoint,int,Dec,"0"
ventilation,int,Dec,"5"

```

A single csv file contains one data set. File name is equal to data set name. Csv format is supported by virtually all spreadsheet and database programs. Number of variables in data set is not limited. For a large number of variables, reading and writing may last a few seconds.

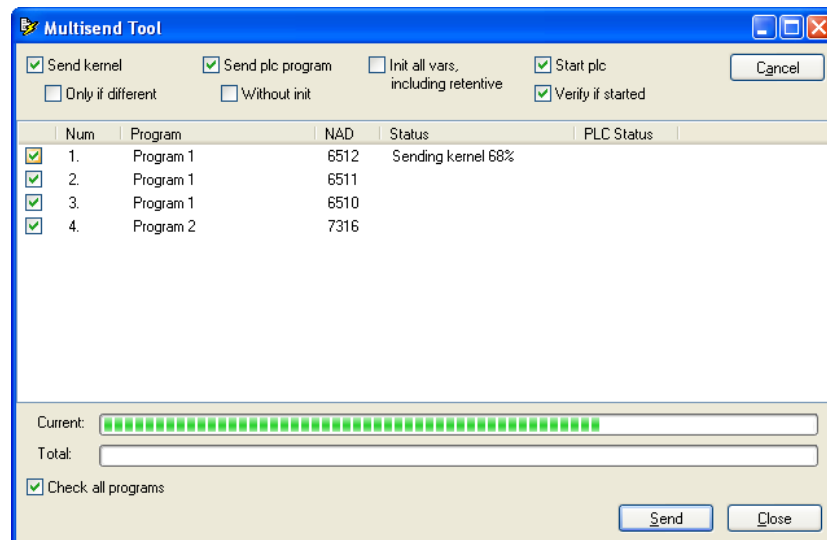
To open a set of plc variables into MS Excel, do the following:

- Create **New** data set
- **Add** variables to data set
- **Read** values from PLC
- **Export** to csv file
- **Open** MS Excel and **import** csv file

Similar procedure may be used in the opposite direction.

Multisend

Multisend is tool which allows updating multiple controllers at once.



All programs in project, and all NAD's for program are listed.

It is optional to send program either without initialization (only if allocation is not changed), with a standard initialization (retentive variables are preserved), or with forced initialization (all variables are initialized, including retentives).

Option "**Check all programs**" will verify all programs by reading back and comparing to original.

Programming

Hardware

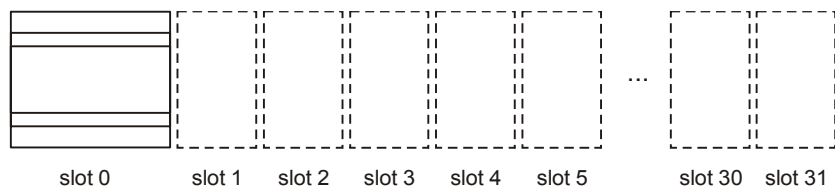
Expansion modules

CyBro-2 can be expanded with various IEX-2 expansion units. For the complete list of available modules, please check the hardware manual.



IEX-2 expansion units

Each IEX module occupies one “slot”. Slot is not a physical device - it is a placeholder, logical entity used to address an expansion.



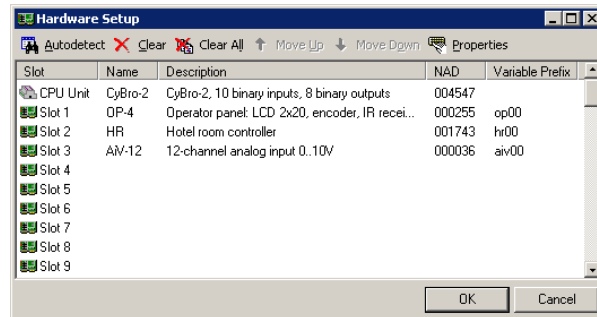
Each slot has a slot number, numbered from 0 to 31. Slot 0 is occupied with local CyBro-2 inputs and outputs.

Each IEX-2 module has unique 21-bit address, there are no DIP switches. Autodetect will sort modules according to type and address, in ascending order. Variable prefixes are assigned in the same order.

Hardware setup

The first step is setting up hardware configuration.

To perform automatic hardware detection press **Autodetect** button. If autodetect is not responding, enter CyBro network address and try again.



After autodetection, Hardware Setup will display list of connected modules. Press **OK** to accept the configuration and create auto-allocated i/o variables.

Variables

Naming convention

Variable name can be any string, containing letters, digits and underlines; provided that the first character is not a digit. **Maximum length is 32 characters**. Names are not case sensitive, so "Valve" and "valve" are the same variable. National characters (like ß, ä, ü, ë, đ, č, ć, đ, š, ž...) are not supported.

Examples of valid names:

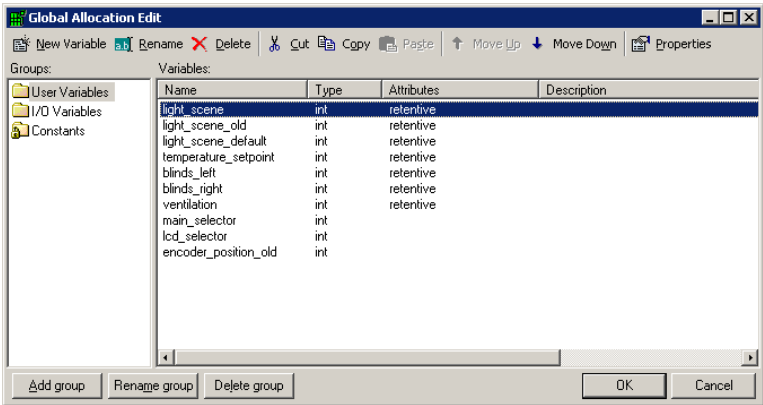
```
cnt
track5a
caret_position
valve_open_req
MaximumWaterLevel
```

Variable name should not match any IEC-1131-3 keyword.

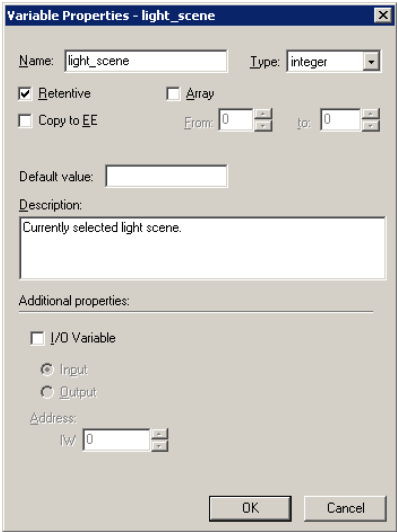
Allocation

According to IEC-1131-3, memory cannot be accessed by address. Each variable should have unique name and a strictly defined type. Same is valid for inputs and outputs.

Variables are allocated using **Global Allocation Edit**:



To insert a new variable choose group where a new variable will be stored, and select **New Variable**. **Insert New Variable** form will appear.



Variable name should be entered. If name is incorrect, button OK will not be available.

Summary of data types:

type	size (bytes)	range
bit	1	0..1
int	2	-32768..32767
long	4	-2147483648..2147483647
real	4	-1e38..1e38
word	16	-

Bit, **int**, **long** and **real** are basic data types.

Bit is a single boolean variable with only two possible values, zero or one. It should be used for flags, logical equations, logical states and similar. **In bit** and **out bit** are both bit type. The result of the comparison instructions is also a bit type value.

Int is a 16-bit signed number. It should be used for counting, encoding states, fixed point arithmetic and similar.

Long is a 32-bit signed value. It should be used when numbers bigger than 32,767 are expected.

Real is a floating point number. Floating point instructions typically consume three times as much memory as integer, and processing of the floating point numbers is about ten times slower than integers.

Word is an array of 16 bits. It doesn't represent a value - it is a handy method to perform logical operations to 16 bits simultaneously. Arithmetic operations are not applicable for word variables.

In bit, out bit, in word and **out word** variables represent physically connected binary (bit) and analog (integer) signals.

Structures (timers and counters) are structured data types, consisting of several dedicated fields.

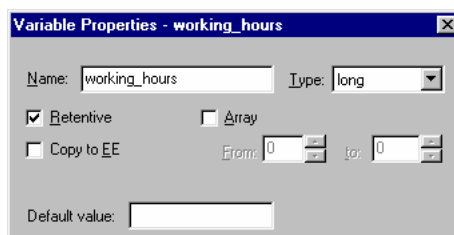
Constants are used to represent a value that will not be changed. As an example, $\text{Pi}=3.14$ can be defined for trigonometrical calculations. Data types do not apply for constants.

Type conversion instructions are also available, but should be used with caution, because they are likely to cause errors. Limited number of type conversions indicates proper planning and good programming style.

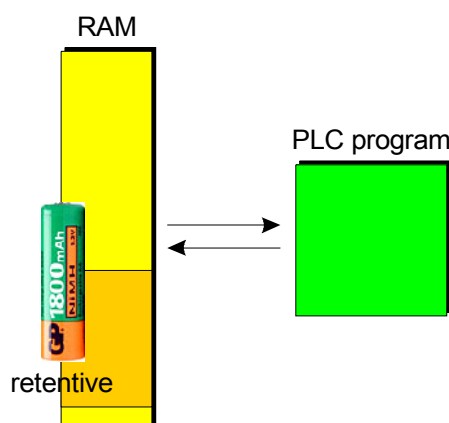
Retentive variables

Retentive variables retain their value when power supply goes down, but also when PLC is stopped/started.

To make variable retentive, set the retentive flag in the global allocation dialog. Retentive flag is set for each variable individually.



Both retentive and non-retentive variables reside in RAM, but non-retentives are automatically cleared by kernel during power-up and program start.



Number of retentive variables is not limited. If required, the whole PLC memory may be retentive.

Data retention time of retentive variables is specified in CyBro hardware manual. When power is off for a time period longer than specified, content of retentive memory may be lost.

System bit **retentive_fail** indicates that retentive memory is lost. It is set automatically after power-on, and clear next time PLC is started. Note that **retentive_fail** means retentive memory is certainly

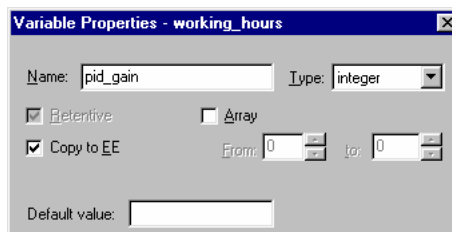
lost. If only a partial lost of memory occur, it may not be indicated. To ensure 100% memory protection, user should calculate checksum/crc protection.

Sending a new program to PLC will clear variables. If allocation list is not changed, retentive variables will be preserved. To send program without clearing variables, use command Send Without Init.

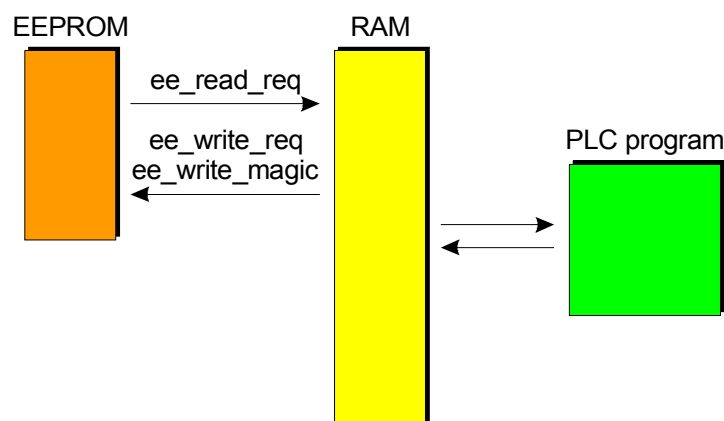
EE variables

Variables that need to be preserved for a long period should be stored in EEPROM memory.

To make variable permanent, set copy to EE flag in the global allocation dialog. Copy to EE flag should be defined for each variable individually.



EE variables resides in RAM memory as all other retentive and non-retentive variables, but they also have a copy in EEPROM. Because of this, they are used by PLC program exactly the same way as other variables, but in addition, reading and writing to EE is available.



To read all variables from EE to RAM, set bit `ee_read_req`. Bit will be automatically cleared when copy is finished. Depending on number of EE variables, copy process may last from few milliseconds up to 3-4 seconds.

To write all variables from RAM to EE, set `ee_write_magic` to 31415 and set `ee_write_req`. When copy is finished, both variables will be cleared. Depending on number of EE variables, write process may last from few milliseconds up to 5-6 seconds. The purpose of magic is to protect EE memory from accidental writing.

To check how system works, open the example program `EEReadWriteDemo.cyp`.

Only a complete EE memory can be read or written, there is no method to read or write single variable.

EE variables should not be accessed by PLC program during both read and write operation. Reading and writing should not be activated at the same time. Clearing bits `ee_read_req` and `ee_write_req` during reading or writing may lead to unexpected results and should be avoided.

All EE variables are automatically read on power-up. Variables may be used when reading is finished, after the `ee_read_req` goes to zero.

Important or sensitive variables may also be stored in EE. EE memory is much more safe against accidental damage caused by electromagnetic spikes. If RAM content is damaged, all chances are that EEPROM is preserved.

Total number of EE variables is limited by physical memory size, and it is specified by hardware manual. To check memory usage, open **PLC Info** dialog box, tab **PLC Program, Total EE size**.

EEPROM data retention time is 100 years. Endurance is about 1 million write cycles.

I/O variables

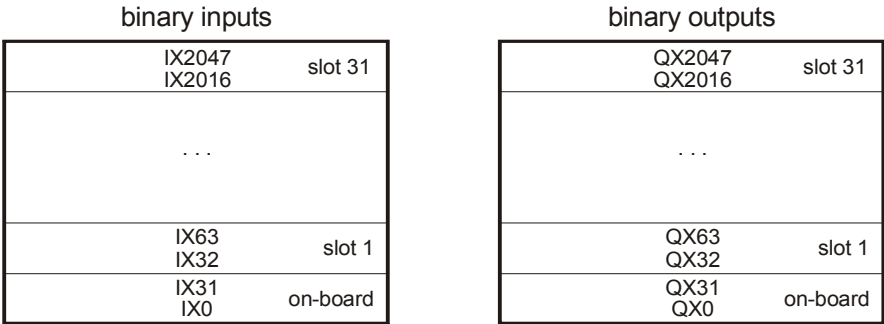
I/O variables are used to access physical inputs and outputs. Hardware setup automatically allocates I/O variables.

Although it is possible to change i/o names, it's advisable to use default names. When hardware setup is changed, renamed variable may not be placed correctly.

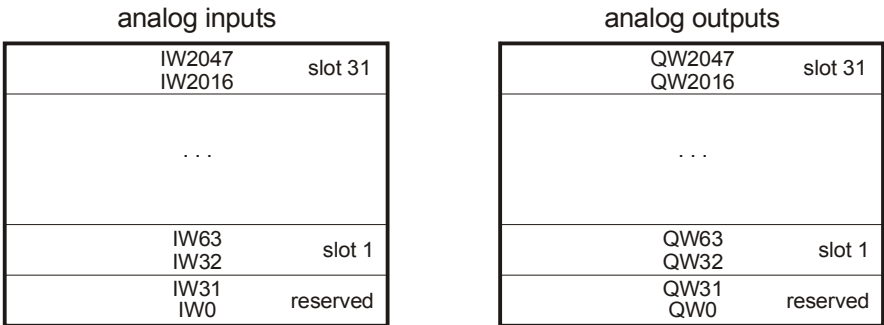
CyBro-2 uses four address spaces for I/O access, two binary and two analog:

IX	input bit
QX	output bit
IW	input word
QW	output word

Binary inputs and outputs are allocated respectively, starting from the ix0 as the first physical input and qx0 as the first physical output. Remaining space is reserved for expansion cards.



Analog I/O cards have 32 words reserved for each slot. Single channel cards use only the first reserved word. Slot 0 is reserved, so the first available slot is 1. **In word** and **out word** variables are both integer type.



Input and output variables are auto-allocated, and their names are in the form:

nnnxx_varname

where nnn is module type name (e.g. bio for Bio-24, op for OP-2), xx is module ordinal number, starting from 00 (i.e. third operator panel is 02), and varname is the name of the variable.

For example, I/O variable `key_f` related to the key F of the first operator panel will be allocated as `op00_key_f`.

Internal variables

CyBro-2 internal variables are allocated through I/O address space. Each variable has a specific function.

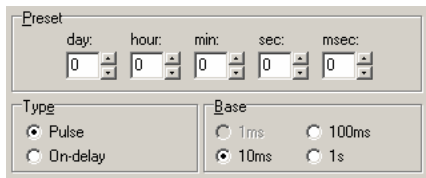
<code>first_scan</code>	Active only during first scan. May be used to initialize variables.
<code>scan_overrun</code>	Indicates that scan timeout occurred. If program cycle lasts longer than 50ms, execution will be interrupted.
<code>clock_10ms</code>	10ms system clock, 5ms high and 5ms low.
<code>clock_100ms</code>	100ms system clock, 50ms high and 50ms low.
<code>clock_1s</code>	One second system clock, 500ms high and 500ms low.
<code>clock_10s</code>	Ten second system clock, 5s high and 5s low.
<code>clock_1min</code>	One minute system clock, 30s high and 30s low.
<code>retentive_fail</code>	Indicates that retentive memory is no longer valid, because the power supply was down for a too long period.
<code>all_outputs_off</code>	If active, all CyBro binary outputs will immediately go off.
<code>all_inputs_off</code>	If active, all CyBro binary inputs are disconnected, retaining the last value.
<code>no_input_filter</code>	If active, local 5ms input filter is turned off.
<code>rtc_read_req</code>	Set to read current time/date from RTC. Will automatically reset when finished.
<code>rtc_write_req</code>	Set to write current time/date to RTC. Will automatically reset when finished.
<code>ee_read_req</code>	Set to read all permanent variables from EEPROM. It is automatically set during power-up. Will automatically reset when finished.
<code>ee_write_req</code>	Set to write all permanent variables to EEPROM. Will automatically reset when finished.
<code>ee_write_magic</code>	Used as EEPROM write protection. Set to 31415 to enable writing. Will automatically reset when writing is finished.
<code>scan_time</code>	Execution time of last scan in milliseconds.
<code>scan_time_max</code>	Maximal scan execution time encountered.
<code>scan_frequency</code>	Number of scan cycles per second. Zero if program stopped.

rtc_sec
 rtc_min
 rtc_hour
 rtc_weekday
 rtc_date
 rtc_month
 rtc_year

Real-time clock data.

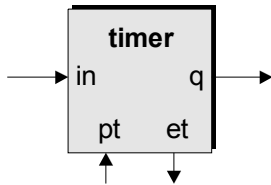
Timers

Timers are special structured variable types, used to determine time interval. To define a new timer variable, open **Insert New Variable** dialog box, choose timer type, enter name, adjust preset, type and timer base, and press **OK**.



Timer base is a period in which the timer is incremented, i.e. the time resolution of the timer. The base should be equal or longer than the program scan time, therefore 10ms base is not suitable for a long CyBro program.

Timer may be represented as the function block with two inputs and two outputs:



Similarly, the timer variable consists of four fields. Each field is an elementary data type. Fields are:

name	direction	type	description
in	input	bit	input
q	output	bit	output
pt	input	long	preset time
et	output	long	elapsed time

To use timer from the CyBro program, the following syntax applies:

```
<timer name>.<field>
```

For example, to set the preset of the wash_timer to 15 seconds (assuming that base is 100ms):

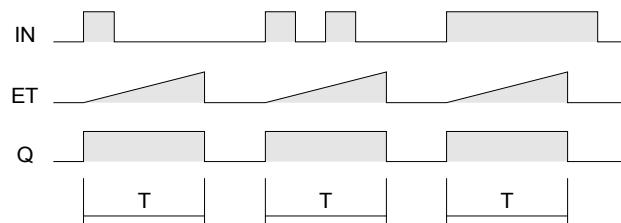
```
wash_timer.pt:=150;
```

Elapsed time of the wash_timer will start at 0 and increment every 100ms until it reaches 150.

Pulse timer

Timer output is activated immediately after the rising edge of input signal. After the specified time, the output will go off. Changes of input signal during pulse do not affect output.

The picture shows typical pulse timer operation:

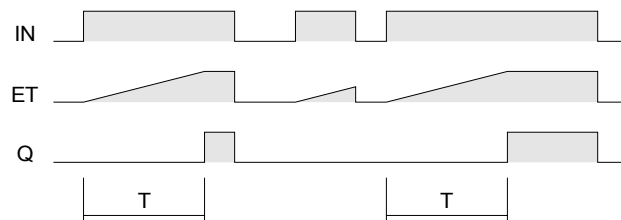


Typical application is staircase light timer.

On-delay timer

When input is activated, timer starts counting. After specified time output activates and stays high until input goes low. Available fields are the same as pulse timer.

The picture shows typical on-delay timer operation:



Typical application is star-delta switching for accelerating three-phase motors.

Counters

Counter type is depreciated, and recommended not to use.

Visibility in alc file

Various tools are able to read/write CyBro variables, like CyBroComServer, CybroMiniScada, CyBroScheduler and others.

Each variable can be marked as:

- User** exported to alc file as "user variable", visible in all tools, including tools intended mainly for end users (e.g. CyBroScheduler)
- System** exported to alc file as "system variable", visible only in tools normally used by system administrators (e.g. CyBroComServer, CyBroConfigTool, CyBroKissLogger, CyBroMiniScada...)
- Hidden** not exported to alc file, invisible for all external tools (except CyPro)

Leave automatically allocated I/O variables marked as System.

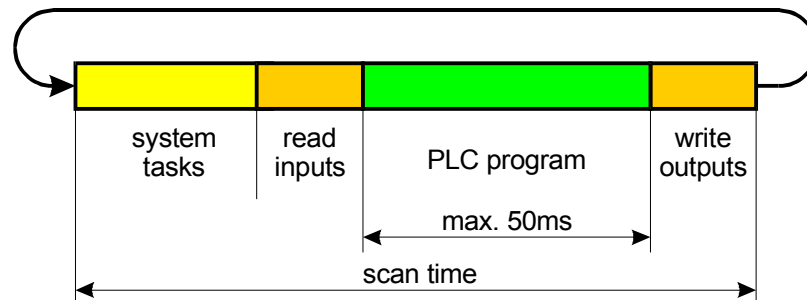
Refresh processing

CyBro implements refresh processing system - inputs are sampled immediately before, and outputs are refreshed immediately after the task execution. Even if input changes during task processing, input value remains stable during scan.

Although refresh processing is slower, it makes programming easier.

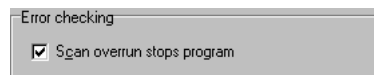
Scan overrun

Scan time is defined as a time needed to complete one program cycle (scan). Program cycle consists of system processing and PLC (user) program.



If execution time of PLC program exceeds 50ms, CyBro enters the scan overrun error state and stops program execution. Error code is displayed on the status bar.

In final version of PLC program, checking for a scan overrun error is usually disabled. To do this, uncheck checkbox **Scan overrun stops program** located at **Program Properties** dialog box, tab PLC.



In that case, program longer than 50ms will be interrupted, and started from the beginning. System variable `scan_overrun` will be set, but program continues operation without error.

Instruction list

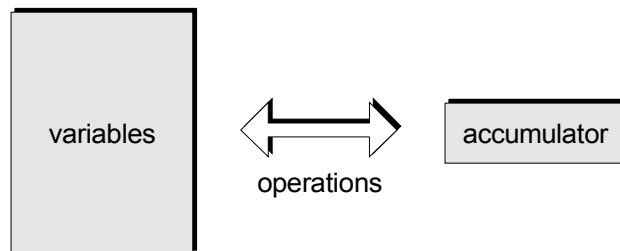
Instruction list is a low level language with structure similar to assembly language. Each line of instruction list code consists of four parts: label, instruction, operand and comment.

```
cnt_beg:    ld      product_count    // counting products
```

|
|
|
|

label
instruction
operand
comment

Computation model of instruction list program consists of allocated variables and accumulator. All arithmetic and logic operations are performed on the accumulator.



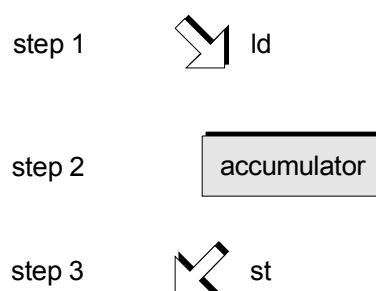
Unlike classical assembly language, accumulator may contain value of any type. Type is determined by the operand. If operand type is not unique, type is determined from subsequent instructions. Type tracking panel shows type for each line.

Once the accumulator is loaded, type cannot be changed until the result of the operation is stored. Exceptions are the comparison instructions and instructions for changing type.

Single operand instructions generally use accumulator as operand. In some cases, it is possible to execute the instruction directly on the variable.

The accumulator is always the first operand in the instructions with two operands. Second operand may be constant or variable. The result of the operation is also stored in accumulator.

Typical arithmetical or logical sequence should load value into accumulator (step 1), perform operation (step 2), and store result (step 3).



Each instruction is written in a separate line.

For a list of available functions, check appendix.

Structured text

Structured text is a high level language with syntax somewhat similar to Pascal, but specifically developed for industrial control applications.

Assignment statements

Assignment statements are used to store value in variable. An assignment statement has the following general format:

```
variable := expression;
```

The assigned value should be equal or lower data type than the variable.

Expressions

Expressions are used to calculate values derived from other variables and constants. Expression always produces a value of a particular data type. An expression may involve one or more constants, variables, operators or functions. Using expressions, CyBro can perform complex arithmetic computations involving nested parenthesis and/or different data types.

Examples:

```
y_position:=5;
down_timer.pt:=15000;
circle_area:=r*r*3.14;
case_counter:=case_counter+1;
volts:=amps*ohms;
start:=(oil_press and steam and pump) and not emergency_stop;
valid_value:=(value = 0) or ((value > 10) and (value <= 60));
```

Operators

CyBro supports a number of arithmetic and logical operators, listed in the following table:

operator	alias	unary	binary	bit	word	int	long	real	result type
+			•			•	•	•	same
-		•	•			•	•	•	same
*			•			•	•	•	same
/			•			•	•	•	same
mod	%		•			•	•		same
not	!	•		•	•				same
and	&		•	•	•				same
or			•	•	•				same
xor			•	•	•				same
=	==		•	•	•	•	•	•	bit
<>	!=		•	•	•	•	•	•	bit
<			•			•	•	•	bit
<=			•			•	•	•	bit
>			•			•	•	•	bit
>=			•			•	•	•	bit
:=			•	•	•	•	•	•	same

Operators are sorted by precedence. Some operators have alias which may be used instead of standard mnemonic.

Expression evaluation

Expressions are evaluated in a particular order depending on precedence of the operators and other sub-expressions. Parenthesized expressions have the highest precedence. Operators of the highest precedence are evaluated first, followed by lower precedence operators, down to the lowest. Operators of the same precedence are evaluated left to right.

Consider following example:

```
Speed1 := 50.0;
Speed2 := 60.0;
Press  := 30.0;
Rate   := Speed1/10 + Speed2/20 - (Press + 24)/9;
```

Evaluation order is:

```
Speed1/10 = 5
Speed2/20 = 3
Press+24  = 54
54/9     = 6
5+3      = 8
8-6      = 2
Rate     = 2
```

To change evaluation order add brackets:

```
Rate:=Speed1/10+Speed2/ (20- (Press+24) /9) ;
```

The expression (20-(Press+24)/9) has higher precedence and will be evaluated before its value is used as a divisor for Speed2.

The value for Rate in this case will be:

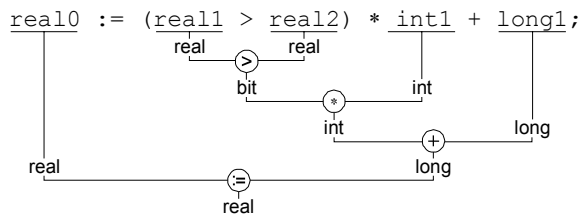
```
= 5 + 60 / (20 - 6)
= 5 + 60 / 14
Rate = 9.286
```

Data type conversions

Type conversions are performed automatically, but only lower-to-higher type conversions are valid:

bit → integer → word → longint → real

In the following example, a number of conversions will be performed.



If both arguments are integer, result is also integer, even if assigned to a real variable.

```
i := 25
r := i/10; //result is r=2
```

To get correct floating point result, at least one operator should be floating point.

To correct previous example, constant 10 may be written as 10.0:

```
i := 25
r := i/10.0; //result is r=2.5
```

Same result is obtained casting one operator to real:

```
i := 25
r := real(i)/10; //result is r=2.5
```

Multiline expressions

It is possible to write a multiline expression, but every line must end with an operator. Following example shows how expression can be divided into several lines.

```
heater_on := (heater_temperature < 600) and
             ((mode = MANUAL) and start_pressed) or
             ((mode = AUTO) and heater_on_request)) and
             not emergency_stop;
```

Conditional statements

Conditional statements provides restricted execution of statement blocks, depending whether particular condition is true or not.

if...then...else

Block of statements can be evaluated and executed depending on the value returned by a boolean expression using **if...then** construction.

This takes general form:

```
if <boolean expression> then
  <statements>;
end_if;
```

The boolean expression can be any expression that returns a TRUE or FALSE boolean result, e.g. the state of a single bit variable or a complex expression involving numerous variables.

Alternative statements can be executed using the **if...then...else** construction in the form:

```
if <boolean expression> then
  <statements>;
elsif <boolean expression> then
  <statements>;
else
  <statements>;
end_if;
```

Examples of conditional execution:

```
if collision then
  speed:=0;
  brakes:=ON;
end_if;

if (gate=closed) and (pump=on) and (temp>200) then
  control_state:=active;
else
  control_state:=hold;
  pump_speed:=10;
end_if;
```

if...then and **if...then...else** constructions can be nested within other conditional statements to create more complex conditional statements.

```
if flow_rate>230 then
  if flame_size>4 then
    fuel:=4000;
  else
    fuel:=2000;
  end_if;
else
  fuel:=1000;
end_if;
```

Further statements can be conditionally executed within the **if...then** using the **elsif** construction, which has the general form:

```
if <boolean expression> then
  <statements>
elsif <boolean expression> then
  <statements>
else
  <statements>
end_if;
```

Any number of additional **elsif** sections can be added to the **if...then** construction.

```
if a>b then
  d:=1;
elsif a=b+2 then
  d:=2;
elsif a=b-3 then
  d:=4;
else
  d:=3;
end_if;
```

case...of

The case statement may provide a readable alternative to deeply nested if conditionals. It consists of an selector expression and a list of statement blocks, each preceded by one possible expression value. Value of selector expression must be ordinal (boolean, integer or longint) and may be the result of another complex expression.

The set of statements that have a constant value that matches the value of the expression are executed. If no match is found, the statements preceded by **else** will be executed.

The case construction has the following form:

```
case <expression> of
  <value1>: <statements>;
  <value2>: <statements>;
  <value3>: <statements>;
else
  <statements>;
end_case;
```

Example:

```

case material_type of
  1: speed:=5;
  2: speed:=20;
  3: speed:=25;
     fan:=ON;
  4: speed:=30;
     fan:=ON;
  5: speed:=50;
     fan:=ON;
     water:=ON;
else
  speed:=0;
end_case;

case alarm_bit of
  TRUE: speed:=0;
  FALSE: speed:=MAX_SPEED;
end_case;

```

Iteration statements

Iteration statements are provided for situations where it is necessary to repeat one or more statements a number of times, depending on the state of the particular variable or condition.

Iteration statements should be used carefully in order to avoid endless loops, which will cause scan overrun error.

for...do

The **for...do** construction allows a set of statements to be repeated depending on the value of an iteration variable. This is an integer or long integer variable which is used to count the statements executions. Iteration variable is incremented by 1 at the end of **for...do** loop.

This construction takes the general form:

```

for <var>:=<expression> to <expression> do
  <statements>;
end_for;

```

Before iteration takes place, its variable is tested whether it has reached the final value. After leaving **for...do** construction, iteration value will contain the final expression value.

The statements within for...do loop should not contain fp or fn instructions.

Example:

```

for i:=0 to 19 do
  channel[i]:=TRUE;
end_for;

for t:=lo_value-1 to hi_value*2 do
  tank_number:=t;
  state[t]:=t/2;
end_for;

```

while...do

The **while...do** construction allows one or more statements to be executed while a particular boolean expression remains true. The boolean expression is tested prior to executing the statements. If it is false, the statements within the **while...do** will not be executed.

This construction takes the general form:

```
while <expression> do
  <statements>;
end_while;
```

The statements within while...do loop should not contain fp or fn instructions.

Example:

```
while value<(max_value-10) do
  value:=value+position;
end_while;
```

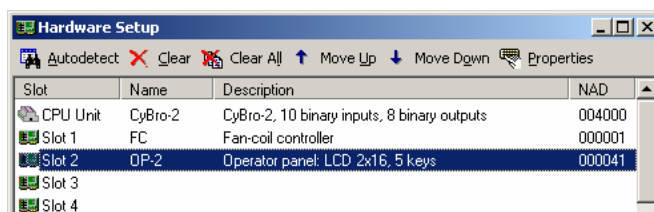
While...do loop is rarely used in a typical PLC program.

Operator panel

General

Operator panel is the optional external device connected to the CyBro-2 via the IEX-2 bus. OP provides LCD display and a few keys readable from the PLC program.

For a proper operation, OP has to be defined in the **Hardware Setup** dialog box. Hardware setup is saved together with project.



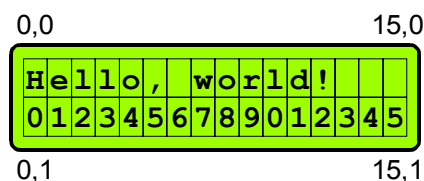
To program operator panel, the following tools are available:

- Print functions Structured text functions typed in the PLC program. Used to display strings and values.
- OP keys Bit variables readable from PLC program. Represents operator panel keys.
- Masks Visual tool for programming operator panel, used to enter parameters. Capable of entering integer values, decimal values and values represented by strings. Parameters may be hierarchically organized.

Print functions

Print functions are structured text functions used to display text messages and values.

First parameter is slot number where display appears in the hardware setup. Two following parameters of all functions are x and y coordinates. They are used to set display position. Print origin is in the upper left corner.



Printing outside visible range may produce unexpected results.

Print functions are:

```
dclr(slot:int);
```

Display clear.

Clears the whole display. Requires no parameters.

```
dprnc(slot:int, x:int, y:int, c:char);
```

Display print ASCII character.

Prints a single character on the specified coordinates. Character may be entered directly ('A'), as ASCII constant (65), or using an integer variable which represents the ASCII value. Values from 0 to 255 are allowed.

```
dprns(slot:int, x:int, y:int, str:string);
```

Display print string.

Prints character string on the specified coordinates. The string is an array of characters enclosed in the single quotes. It may contain any characters from the extended ASCII set (character codes 32 to 255).

```
dprnb(slot:int, x:int, y:int, c0:char, c1:char, value:bit);
```

Display print binary value.

Prints first or second ASCII character on the specified coordinates, depending on the bit value. If the value is false the first character is printed, otherwise the second.

```
dprni(slot:int, x:int, y:int, w:int, zb:bit, value:int);
```

Display print integer value.

Prints integer value to the specified coordinates. Parameter w defines printing width. For example, if w is 4, valid print range is from -999 to 9999. Parameter zb is binary and stands for a zero blanking. If zb is 1 leading zeroes are not printed.

```
dprnl(slot:int, x:int, y:int, w:int, zb:bit, value:long);
```

Display print long value.

Prints long value to the specified coordinates. Parameter w defines printing width. For example, if w is 6, valid print range is from -99999 to 999999. Parameter zb is binary and stands for a zero blanking. If zb is 1 leading zeroes are not printed.

```
dprnr(slot:int, x:int, y:int, w:int, dec:int, value:real);
```

Display print real value.

Prints real value to the specified coordinates. Parameter w defines printing width, and parameter dec defines number of printed decimals. For example, if w is 6 and dec is 2, valid print range is from -99.99 to 999.99. Zero blanking is always on.

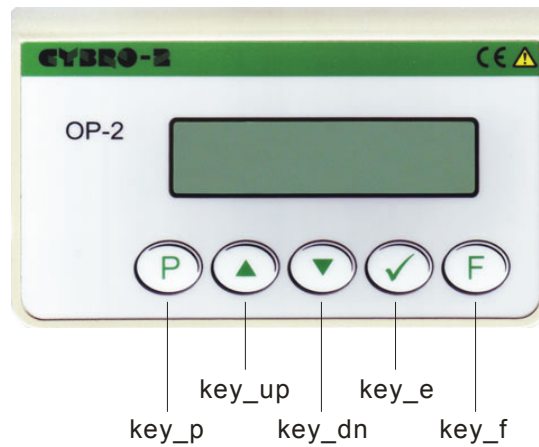
Each function parameter (except string for dprns) may be a constant, a variable or an expression. This may be used to create animated displays, as in the following example:

```
dclr(1);
dprns(1,0,0,'Moving...');
dprns(1,x,1,'o');
x:=(x+fp(clock_100ms))%16;
```

CyBro-2 may handle multiple operator panels.

OP keys

OP keys are accessible from PLC program as input variables:



Key **P** is usually used to invoke and exit mask, so it's not available for PLC program (reading is zero). However, if no entry point is defined, it behaves the same as other keys. In such case, mask may be invoked by writing mask number to `op00_next_mask`.

When mask is active, keys **up**, **dn** and **E** are also not available (reading is zero). Key **F** is always available.

Keys don't have autorepeat. Key variable is true as long as the key is pressed. After the key is released, it becomes false.

Any two (or more) keys may be pressed simultaneously. This may be used to initiate a special function. In the following example, pressing **up** and **dn** simultaneously resets `product_count`.

```
if fp(key_up and key_dn) then
    product_count:=0;
end_if;
```

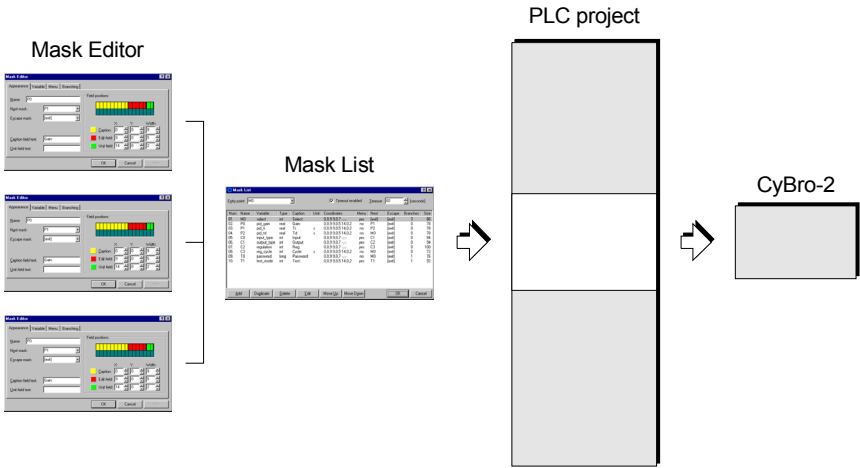
Key variables are allocated automatically when OP is defined in Hardware Setup.

Masks

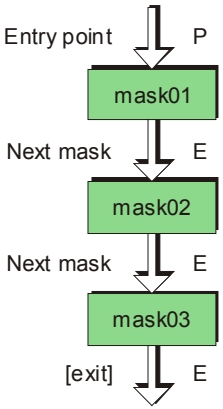
Mask system is easy-to-use visual tool for programming operator terminal.

Mask is a container for a variable that will be edited. Masks are transferred to the CyBro together with PLC code.

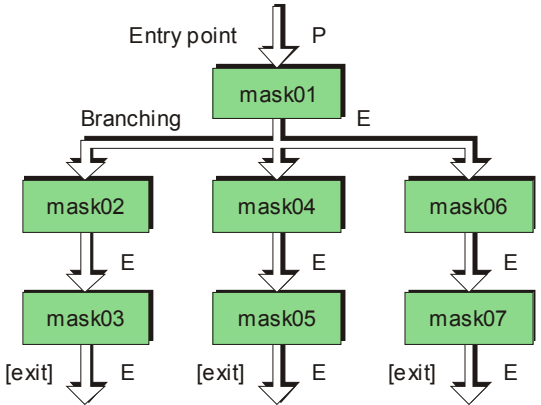
User creates a new mask or edits the existing one by using **Mask Editor**. Created masks are listed in the **Mask List**. Masks are integral part of the PLC project, they are saved on the disc and transferred to the CyBro.



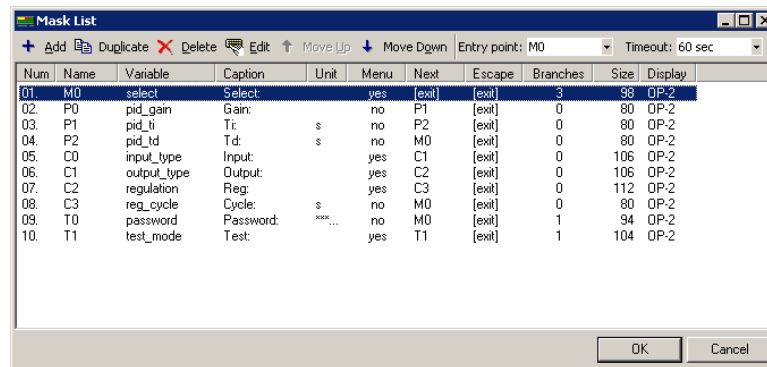
When user presses **P**, CyBro sends first mask to the OP. Pressing **E** advances it to the next mask.



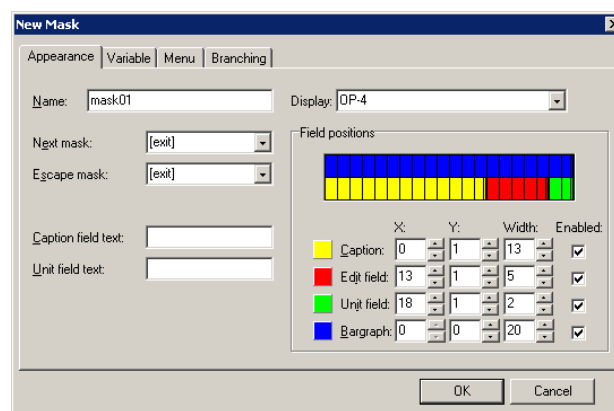
By the use of branching features, masks may be hierarchically organized:



To start working with masks, press **Masks** button on the standard toolbar or press key **F7**. **Mask List** dialog box will appear.



To create a new mask click **Add** or press **Insert** key. **Mask Editor** dialog box will appear.



Name is a unique string identifier that identifies a particular mask.

Next mask defines a mask that becomes active after **E** key is pressed.

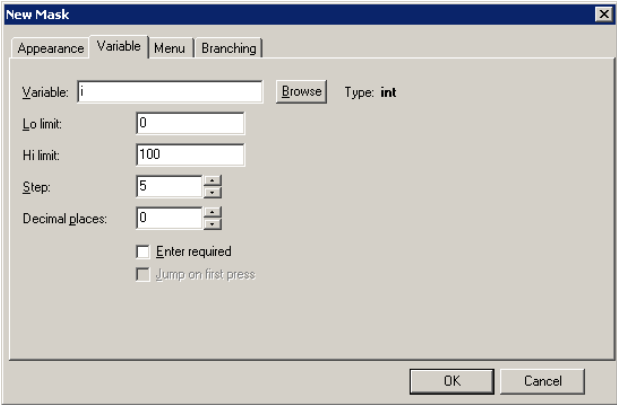
Escape mask defines a mask that becomes active after **P** key is pressed. Usually, this key is used to exit from mask.

Caption field is a short string that will appear on the display to identify the currently edited variable. Caption position is represented by the yellow rectangle. To move the caption, drag the rectangle into the desired position. To resize caption, drag the right edge of the rectangle.

Edit field is a display area in which the value of edited variable is displayed. It is represented by the red rectangle. Edit field should have enough space for editing variable in the desired range. To move and resize field, drag it like the caption.

Unit field is a short string, similar to caption. Unit field is represented with green rectangle, and it is commonly used for displaying engineering units.

Bargraph is a semi-graphic horizontal progress bar. Few different styles are available. To use bargraph, both low and high limits should be defined.



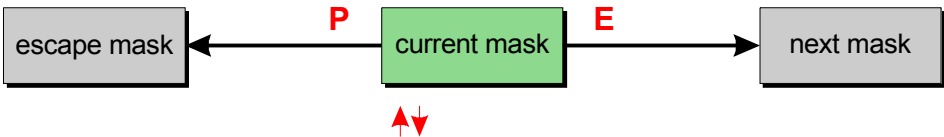
Lo limit and Hi limit define allowed range.

Step defines a value for which the variable will be changed for a single key press.

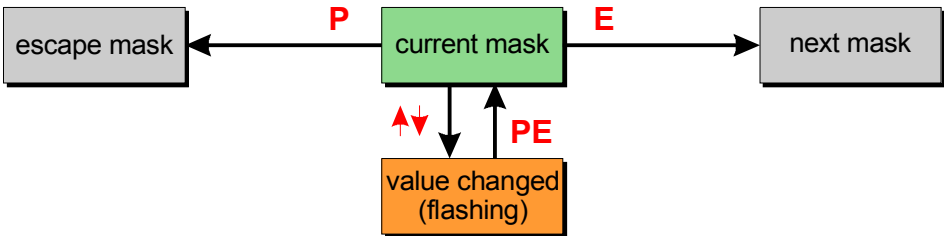
Decimal places may be used for real as well as for integer and long variables. In the former case, only the display is fractional (e.g. for decimal places=1, value 254 is shown as 25.4).

Enter required and Jump on first press define method to operate with navigation keys (P, E). Three combinations are available:

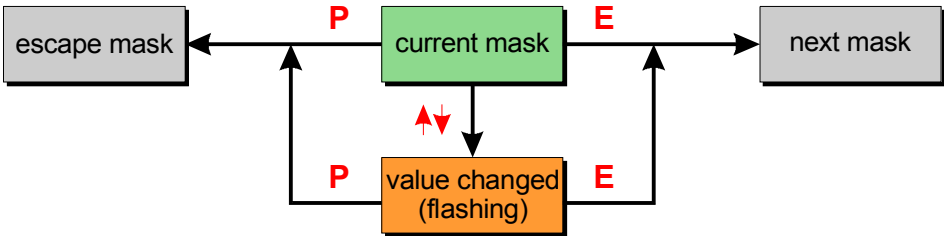
Enter required: no



Enter required: yes
Jump on first press: no

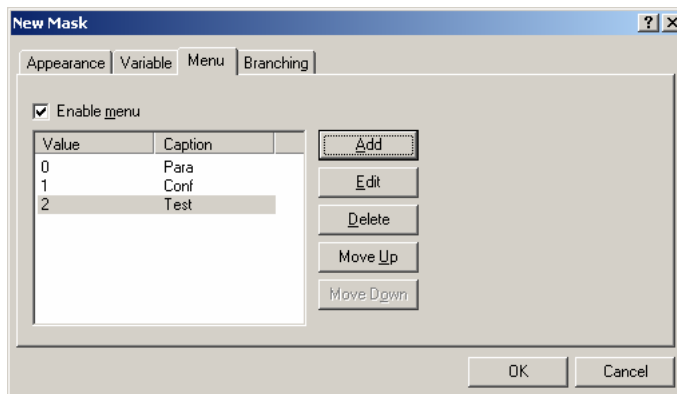


Enter required: yes
Jump on first press: yes



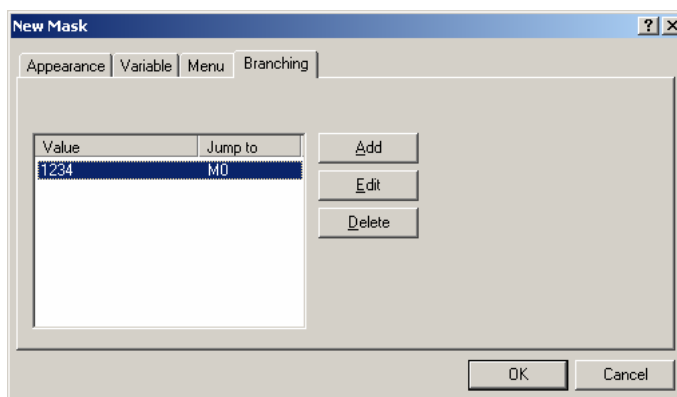
If enter required is false, changed value will be sent to CyBro immediately after up or dn key is pressed. If enter required is true, changed value will be sent to CyBro only when E key is pressed. To indicate that change is not confirmed, changed value will flash.

Variable may be entered as menu rather than as numerical value. To define menu entries, run **Mask Editor**, click **Menu** tab and **Add** as many items as needed.



When executing CyBro program, the display will show items by name, and variable `product_type` will take value 0, 1 or 2.

Branching tab provides branching onto different masks according to the entered value. This can be used to organize parameters into various parameter sets, but also for a password protected parameters.



Active mask takes control of all panel keys except the **F** key, so it is not possible to use them from CyBro program at the same time. Mask fields are displayed “over” the user display. After exiting mask, display content is restored.

If mask is too large to fit into operator panel it will not be activated, and it will operate like an empty mask. Mask size is displayed in **Mask List** dialog box. Available operator panel mask memory is displayed in the **Hardware Setup** dialog box. To decrease mask size reduce number of menu entries or reduce edit field width. Reducing caption and unit field width may also save few bytes.

Only one mask can be activated at the time.

Handling masks from plc program

CyBro program can get currently active mask number by reading variable `current_mask`. When `current_mask` is zero, no mask is active.

Program may force execution of a certain mask by writing to variable `next_mask`. After the mask is sent, `next_mask` will be set to -1, and `current_mask` will change accordingly.

The following example shows a typical mask transition:

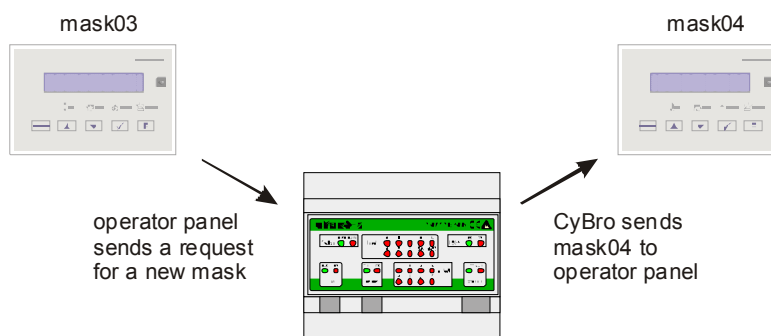


Table shows approximate timings and values for the transition:

		1	2	3		4	
		↓	↓	↓		↓	
mask03 variable	20	20	→ 25	25		25	
current_mask	3	3	3	0		→ 4	
next_mask	-1	-1	-1	→ 4		-1	
		2-3 ms		2-3 ms	50-100ms		

Events are marked by black arrows:

1. User pressed Enter
2. New value sent to CyBro
3. Request for new mask sent to CyBro
4. New mask sent to operator panel and activated

Red arrows mark most important value changes.

The same transition may be initiated by the following plc program:

```
if <condition> then
    op00_next_mask:=4;
end_if;
```

Short gap in **current_mask** value comes from the finite network response time. To check weather no mask is active, program should also check the value of **next_mask**. The following example will properly set the panel begining mask:

```
if op00_current_mask=0 and op00_next_mask=-1 then
    op00_next_mask:=10;
end_if;
```

Both mask control variables may also be accessed through the A-bus.

Serial ports

General

COM1 and COM2 are multi-purpose serial RS232 ports, available for following protocols:

- A-bus slave
- Modbus RTU slave
- free-programmable port

Operation mode is selected by Configuration. It is not possible to change mode from plc program.

Free-programmable mode

As a general-purpose communication port, various serial devices may be connected: sensors, scales, modems, radio links, printers and others.



Communication protocol is determined by the PLC program. Although binary message format is supported, communication functions are most suitable for sending, receiving and parsing plain ASCII messages. Both master and slave operation is supported. Com2 port is full duplex, so data may flow simultaneously in both directions.

Port selection

Before communication commands are applied, port should be selected:

```
com_select(port:int);
```

To select COM1, **port** is 1. To select COM2, **port** is 2.

Selected port may be changed at any time.

Prepare transmit message

Prior to transmission, program should create complete outgoing message. To create a message use display functions `dprnc()`, `dprns()`, `dprnb()`, `dprni()`, `dprnl()` and `dprnr()`.

To "print" to transmit buffer, use 0 as a slot number - transmit buffer appears as display in slot 0.

The X coordinate is transmit buffer position. The Y coordinate should be zero. Maximum message length is 1024 bytes.

For example, to write message "Hello!" enter:

```
dprns(0,0,0,'Hello!');
```

Message length is 6 characters.

Special characters may be entered as a two-character combination. The first character being a backslash ('\'), and second being a one of the following:

combination	ASCII code	hex code
\n	CR LF	0D 0A
\r	CR	0D
\t	TAB	09
\\	\	5C
\nn	any	nn

The last three-character combination may be used to enter a hexadecimal code of any ASCII character. For example, '\41' is equivalent to letter 'A'.

For example, to create the message "Hello!" followed by carriage return and line feed, enter:

```
dprns(0,0,0,'Hello!\n');
```

Message length is 8 characters.

To create message composed of keywords and numerical values use:

```
dprns(0,0,0,'>LEVEL=xxx.xx TEMPERATURE=xx.x ERROR=xx\n');
dprnr(0,7,0,6,2,water_level);
dprnr(0,26,0,4,2,water_temperature);
dprni(0,37,0,2,no,error_code);
```

Note that x-es in the dprns instruction will be overwritten by the subsequent commands.

Function dprnc() is used to enter single-byte binary value.

For example, to create 4-byte message containing letters 'ABC' followed by ESC character (ASCII code 27), enter:

```
dprnc(0,0,0,65);
dprnc(0,1,0,66);
dprnc(0,2,0,67);
dprnc(0,3,0,27);
```

To create 2-byte binary message containing an integer value (most significant byte first), enter:

```
dprnc(0,0,0,value/256);
dprnc(0,1,0,value%256);
```

Using dprns() and dprnc(), any binary message may be created.

Transmit

To send prepared message, use function tx_start():

```
tx_start(char_num:int);
```

Parameter **char_num** is the number of characters to transmit. Transmission always starts from beginning of the buffer.

```
tx_active():bit;
```

Function tx_active() returns current transmission state. When transmission is finished tx_active() falls to zero.

```
tx_count():int;
```

Function `tx_count()` returns the number of characters left. If `tx_count()` is zero and `tx_active()` is still true, that means last character is currently transmitted.

```
tx_stop();
```

To stop transmitting immediately, use function `tx_stop()`. Current character will be finished, but rest of a message will not be transmitted.

Receive

Receiver and transmitter are fully independent.

Maximum length of a received message is 1024 bytes. If more than 1024 characters are received, receiving position rolls-over to the beginning of the receive buffer. In that case, the number of received characters is set to zero.

To start receiving use function `rx_start()`. This function also defines criteria to stop receiving:

```
rx_start(beg_ch:char, end_ch:char, len:int, msg_tout:int, char_tout:int);
```

Parameter **begch** specifies the first character of received message. After the receiving is initiated, any other character will be rejected until **begch** is received. To receive a message without specifying the first character, put a zero instead of **begch**.

Parameter **endch** specifies the last character of a received message. After **endch** is received, reception is stopped. To receive a message without specifying the last character, put a zero instead of **endch**.

Parameter **len** specifies the length of a received message. After the required number of bytes, the reception is stopped. To receive a message of undefined length, set **len** to zero.

Parameter **msg_tout** specifies message timeout in milliseconds. This is the time after which the receiver will quit if no characters are received. Maximum timeout is 32 seconds. To receive without limiting message time, set **msg_tout** to zero.

Parameter **char_tout** specifies the receiving timeout between individual characters. If characters sent by accompanying device are sent continuously without gaps, character timeout may be set to a pretty low value in order to improve the receiver response time. Character timeout should always be greater than the time needed to transmit a single character, concerning selected baud rate and data bits.

Example:

Communication parameters are 1200 bps, 8 bits and no parity. Transmission of one character will be approximately 8ms (start bit + 8 data bits + stop bit = 10bits; 10bits/1200bps=8.3ms). Character timeout should be at least 10ms, although cca. 50ms is more safe, if response time is not critical.

To receive without timeout set **char_tout** to zero.

Message and char timeouts are independent. It is possible to use character timeout, and leave message timeout disabled. The opposite is probably of little use.

A few examples will illustrate usage of the `rx_start()` function:

Receive continuously:

```
rx_start(0,0,0,0,0);
```

Receive message beginning with '>' and ending with CR character:

```
rx_start('>', '\r', 0, 0, 0);
```

Receive message of exactly 12 characters:

```
rx_start(0, 0, 12, 0, 0);
```

Infinitely wait for a first character, but stop receiving 250ms after message is received:

```
rx_start(0, 0, 0, 0, 250);
```

Stop receiving 100ms after the last received character, or if no character is received in 5 seconds:

```
rx_start(0, 0, 0, 5000, 100);
```

Any combination of start and stop criteria is allowed. To receive message up to 80 characters, ending with Ctrl-Z (1A hex) with 10 seconds timeout:

```
rx_start(0, '\1A', 80, 10000, 100);
```

If given stop criteria for is not appropriate, determining the end of message may be done using PLC program. Received message should be analyzed on the fly, and when stop condition is satisfied, function `rx_stop()` may be used to stop receiving:

```
rx_stop();
```

Function `rx_count()` returns the number of received characters. The function may be used whether the receiving is active or not. Function `rx_start()` resets this number to zero.

```
rx_count():int;
```

To check if the receiving is active, use function:

```
rx_active():bit;
```

To check detailed receiving status, use function:

```
rx_status():int;
```

Function `rx_status()` returns one of the following codes:

- 0 - receive active
- 1 - `rx_stop()` executed
- 2 - end character detected
- 3 - requested number of characters received
- 4 - timeout expired

Example:

Incoming message has no terminating character, and the length may vary. Message length is coded binary in the fourth byte. To receive such messages use the following code:

```
if rx_active() and rx_count()>=4 then
  if rx_count()>=rx_bufrd(3) then
    rx_stop();
  end if;
end if;
```


Parse received message

Functions for parsing a received message:

```
rx_bufprd(position:int):int;
```

Returns single character from the given position of the receive buffer. Character is converted to integer value 0..255.

```
rx_strcmp(position:int, str:string):bit;
```

Compares receive buffer with a specified string. If string match, return true.

```
rx_strpos(position:int, str:string):int;
```

Searches specified string in receive buffer. Search starts from given position. If the string is found, function returns position of the first matching character, otherwise return value is -1.

```
rx_strtoi(position:int):int;
```

Returns value of decimal number starting at given position. If character at specified position is space, next character is taken until digit is found. Conversion continues until the first non-digit character is encountered.

```
rx_strtol(position:int):long;
```

Returns value of decimal number starting at given position. If character at a specified position is space, it is skipped until digit is found. Conversion continues until first non-digit is encountered.

```
rx_strtor(position:int):real;
```

Returns real value of decimal number starting from the given position. If character at a specified position is space, the next character is taken until the number is found. Conversion continues until the first non-digit character is encountered.

Example:

Communication is used to set the parameters of the current PLC program. Message may contain keywords (OPEN, CLOSE) or set values (TEMP=25, CYCLE=100), separated by spaces. Incoming message may contain request to set one or more parameters.

```
if rx_strpos(0,'OPEN')<>-1 then
  main_valve=1;
end_if;

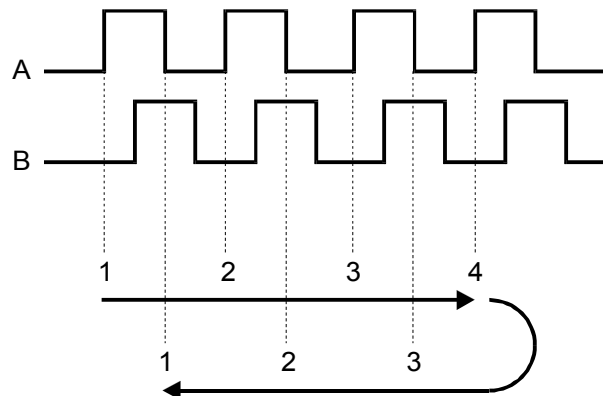
if rx_strpos(0,'CLOSE')<>-1 then
  main_valve=0;
end_if;

position=rx_strpos(0,'TEMP=');
if position<>-1 then
  set_point=rx_strtoi(position+5);
end_if;

position=rx_strpos(0,'CYCLE=');
if position<>-1 then
  cycle_timer.pt=rx_strtol(position+6);
end_if;
```


Two phase counting, single precision

Counting both up and down. Two-phase inputs (A and B) are used.

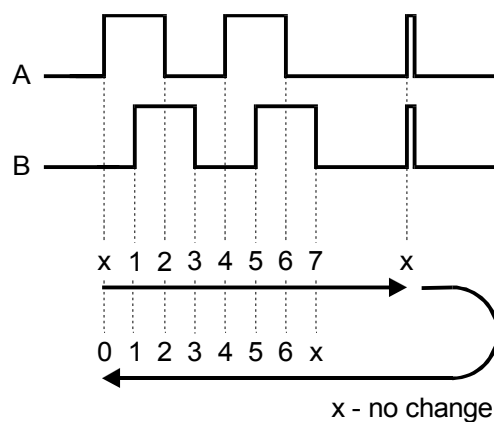


Counter is advanced or retarded at rising edge of A input. Counting resolution is equal to encoder resolution, each encoder pulse increments (or decrements) counter by 1.

Shaft vibration or electromagnetic noise may result in false counting.

Two phase counting, quad-precision

Counting both up and down. Two-phase inputs (A and B) are used. Counter is incremented or decremented at each edge of A and B inputs. Counting resolution is four times the encoder resolution, so each encoder pulse increments (or decrements) counter by four. 500 pulses per rotation encoder will actually increment counter by 2000.



Quad-precision mode also yields glitch removal and ± 1 pulse removal, eliminating improper counting caused by shaft vibration or electromagnetic noise.

High-speed counter and Com2 serial port can not be used at the same time. Function `hsc_start()` will interfere with COM2 serial communication.

After the power-up, counter value is undetermined.

If HSC is not used, high speed inputs A and B may be used as standard binary inputs.

For connections and technical specifications of HSC, please check the hardware manual.

High speed counting

By default, the high-speed counter is stopped. To start counting use function `hsc_start()`.

```
hsc_start(); // start high-speed counter
```

To stop counting use function `hsc_stop()`.

```
hsc_stop(); // stop high-speed counter
```

To determine if HSC is stopped or started, use function `hsc_active()`. If HSC is started, `hsc_active()` returns true, otherwise returns false.

```
if hsc_active() then // check if high-speed counter is running
  activate_drive();
end_if;
```

To read the current position of high-speed counter use function `hsc_read()`. Reading is possible no matter if the counter is active or not. Counter value type is longint, and may be stored in a variable of the same type.

```
current_position:=hsc_read();
```

To set the counter to the desired value use function `hsc_write()`. Usually this is performed only when the counter is stopped, but it is possible to write a new value even if the counter is active and running.

```
hsc_write(12000);
```

As an example, the following line increments the counter by 500 counts.

```
hsc_write(hsc_read()+500);
```

High speed action

High-speed counting is usually used for a precise motion control. The following example shows how to stop the motor connected to `qx000` output, when the counter reaches zero (downward counting).

```
if hsc_read()<=0 then
  qx000:=0;
end_if;
```

Problem that might arise here is the unpredictable response time. If current scan time is 5ms, response time may be anywhere between 0ms and 10ms. For a speed of 1m/s that induces inaccuracy of 10 millimeters.

To prevent this, high speed counter has a hardware capable of performing a very fast action when counter reaches zero. Action is defined as setting or resetting a single binary output. To perform task as in the last example, execute the following line:

```
hsc_set_action(0,qx000);
```

The function `hsc_set_action()` initiates action. When counter reaches zero, the output `qx000` will be deactivated. Also, it is possible to set action in the opposite direction, to activate the output. See example:

```
hsc_set_action(1,brake_out);
```

When counter reaches zero, brakes will be activated.

Action should be initiated only once, so the typical code for activating action is:

```
if fp(start_key) then
  hsc_write(300000);
  hsc_set_action(0,qx000);
  hsc_start();
  qx000:=1; // start motor
end_if;
```

Function `hsc_reset_action()` is provided to cancel initiated action. This is useful when something unexpected happens, for example when alarm condition is detected.

```
if alarm_condition then
  hsc_reset_action();
  hsc_stop();
  qx000:=0;
  qx001:=0;
end_if;
```

Action is performed only once, when counter reaches zero for the first time. When counter reaches zero again, no action will be performed. To check if the action is still pending, use function `hsc_check_action()`. If action is initiated, `hsc_check_action()` returns true. After the action is performed (or canceled), `hsc_check_action()` returns false.

It is not possible to initiate more than one action simultaneously. An attempt to do so will cause incorrect operation. If multiple actions are needed, next action may be initiated after the first action is performed. In applications that need multi-speed control, gear change may also be performed by `if..then` instructions, rather than actions. Usually, this allows enough precision to slow down, with a single high speed action used to stop accurately.

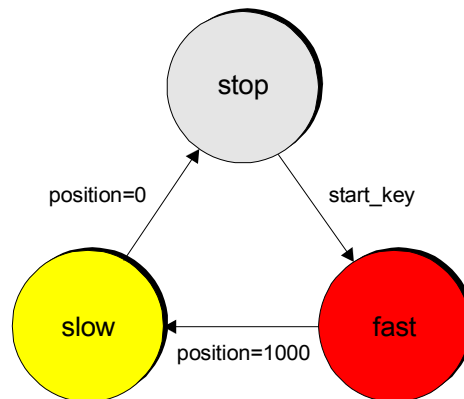
Next example shows a typical two-speed application. Two-speed motor is activated by two binary outputs, `qx000` and `qx001`. Low speed is activated by `qx000`, and high speed is activated by `qx001`.

	stop	slow	fast
qx000	0	1	1
qx001	0	0	1

The example requires motion of 6000 units. For the first 5000 units motor will run fast, then for the next 1000 units it will run slowly, and then it will stop. Slowing down will allow accurate stop position.



Program consists of three main states: stop, fast and slow. Transitions between states are defined by start_key and position.



The actual code may be written on both first or second manner, by the if..then instruction or by the two consecutive actions.

The first way is somewhat shorter and easier to understand:

```

if qx000=0 and qx001=0 then
  // currently stopped
  if fp(start_key) then
    // start full speed and set action to stop after 6000
    hsc_write(6000);
    hsc_set_action(0,qx000);
    hsc_start();
    qx000:=1;
    qx001:=1;
  end_if;
elseif qx000=1 and qx001=1 then
  // currently fast
  if hsc_read()<=1000 then
    // slow down
    qx000:=1;
    qx001:=0;
  end_if;
end_if;

```

The same functionality may be performed in another way, by the two sequentially activated actions. The code is a bit longer and less understandable, but both actions will be performed very accurately.

```

if qx000=0 and qx001=0 then
  // currently stopped
  if fp(start_key) then
    // start full speed and set action to slow down after 5000
    hsc_write(5000);
    hsc_set_action(0,qx001);
    hsc_start();
    qx000:=1;
    qx001:=1;
  end_if;
elseif qx000=1 and qx001=0 then
  // currently slow
  if not hsc_check_action() then
    // continue slow, increment counter by 1000 and stop on zero
    hsc_write(hsc_read()+1000);
    hsc_set_action(0,qx000);
  end_if;
end_if;

```

Zero reset

HSC types A/AB+Z x1 and AB+Z x4 implements an additional high speed input, used to accurately determine absolute position.

Usual function of zero input is to reset high speed counter. By default this is disabled, and may be enabled by the function `hsc_enable_zero()`. If enabled, each inactive-to-active transition on the zero input will reset counter to zero.

To disable zero reset, use function `hsc_disable_zero()`. To check if the zero reset is enabled, use function `hsc_check_zero()`. If the reset is enabled, `hsc_check_zero()` will return true, otherwise it will return false.

Once activated, zero reset is active until explicitly deactivated by the `hsc_disable_zero()` function.

If zero reset is enabled and action is activated, action will be performed immediately after the transition is detected.

Zero detect

The second way of utilizing zero input is provided by the two additional functions, `hsc_detect_zero()` and `hsc_read_zero()`.

Function `hsc_detect_zero()` indicates transition on the zero input. When inactive-to-active transition on zero input is detected, `hsc_detect_zero()` will return true. Next consecutive calls will return false, until the next zero transition is detected again.

That allows simple transition counting:

```
zero_counter:=zero_counter + hsc_detect_zero();
```

When transition is detected, value of the high speed counter is also written to the zero detect register, readable by the function `hsc_read_zero()`. Zero detect register is also 32 bits long.

Zero detect functions may be used only in A/AB+Z x1 and AB+Z x4 modes.

Real-time clock

General

Real-time clock (RTC) is a hardware clock/calendar device. It runs even when the power supply is down. For technical specifications about accuracy and data retention time, please check the hardware manual.

RTC may be synchronized to PC clock each time program is transferred to the PLC. To enable or disable this option, clear check box **Tools/Environment Options/Communication/Synchronize RTC to PC Clock**. RTC can also be adjusted from the PLC program with RTC I/O variables.

I/O variables

To read or set time of the RTC, use:

```
rtc_hour:int;
rtc_min:int;
rtc_sec:int;
```

Value ranges for that variables are:

Hour	0..23
Min	0..59
Sec	0..59

To read or set date of the RTC, use:

```
rtc_year:int;
rtc_month:int;
rtc_date:int;
```

Values are in the range:

Year	2000..2099
Month	1..12
Date	1..31

To read or set day of the week of the RTC, use:

```
rtc_weekday:int;
```

- 0 - Sunday
- 1 - Monday
- 2 - Tuesday
- 3 - Wednesday
- 4 - Thursday
- 5 - Friday
- 6 - Saturday

To set real-time clock, write new time/date to I/O variables and set write request flag:

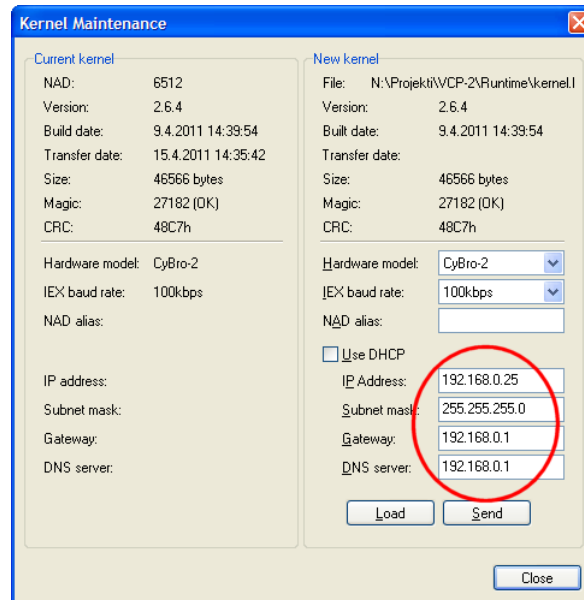
```
rtc_write_req:=1;
```

An example that displays date/time on the operator panel and allows user to adjust them is RtcClock.cyp. The program is located in the Project\Examples directory.

Networking

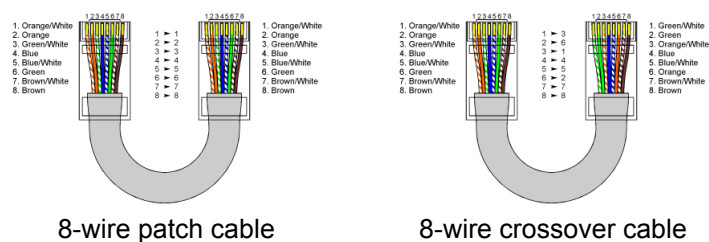
Ethernet connection

CyBro may have a dynamic IP address obtained from DHCP server, or static IP address set with **Kernel Maintenance**. To define static address, turn off checkbox **Use DHCP**, and enter **IP address**, **Subnet mask**, **Gateway**, and **DNS server**. DNS server is needed when push to domain name is used, push to IP address needs no DNS.



After power-on, static IP address is available immediately. Dynamic address needs 3-5 seconds to start in same network as last time, or 30-35 seconds in a new network. If DHCP server is not available, CyBro will have no IP address (0.0.0.0). In such case, **Direct connection** may be used to obtain connection.

CyBro has a standard RJ-45 UTP connector. Baudrate is 10/100M, autodetected.



8-wire patch cable

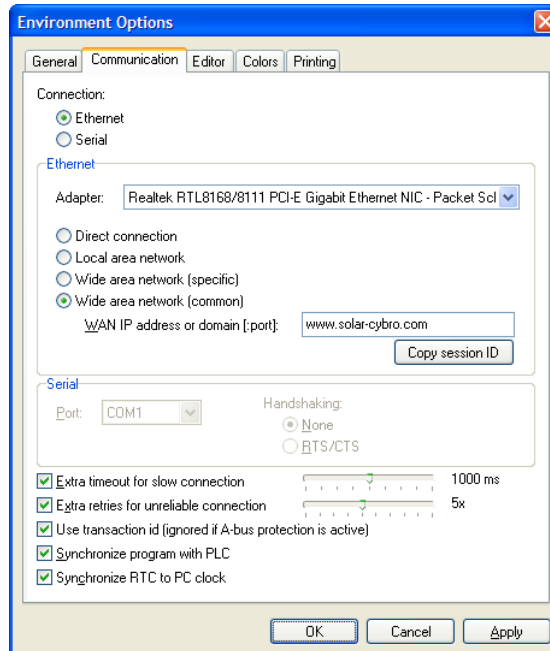
8-wire crossover cable

Connection cable may be patch cable or crossover cable. Cable type is detected automatically.

CyBro has 6-byte MAC address in form 00-CB-00-xx-xx-xx, where last three bytes are serial number (NAD). For example, CyBro 6512 (001970h) has MAC address 00-CB-00-00-19-70.

Connection options

- Direct connection
- Local area network
- Wide area network (specific) - individual address for each controller, stored with project
- Wide area network (common) - one common address for all controllers, stored in registry



When computer has two or more network adapters, the right one should be selected manually.

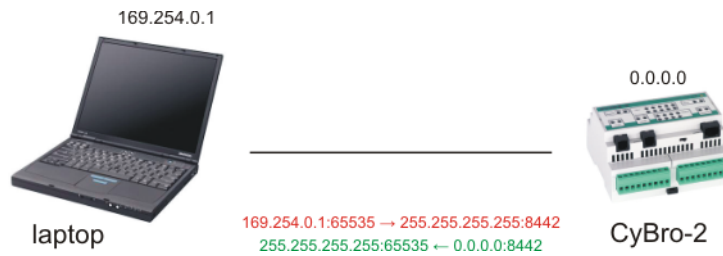
Transaction id adds an unique id to each request/answer message pair, avoiding trouble with delayed messages. It can't be used when A-bus protection (password) is active.

Recommended settings:

	typical rountrip	transaction id off		transaction id on	
		extra timeout	extra retries	extra timeout	extra retries
direct connection	1ms	-	-	-	-
local network connection	2-3ms	-	-	-	-
internet connection	50-100ms	1000ms	3x	200ms	5x
relay connection	100-200ms	2000ms	3x	500ms	5x
HSDPA connection	200-500ms	5000ms	5x	500ms	5-10x
GPRS connection	500-1000ms	20000ms	10x	500-1000ms	5-10x

Transaction id is supported starting with kernel v2.6.4 and loader v2.6. Older kernel/loader will result in no connection. If you have loader v2.5, turn transaction id off, send new kernel, then turn transaction id on.

1. Direct connection

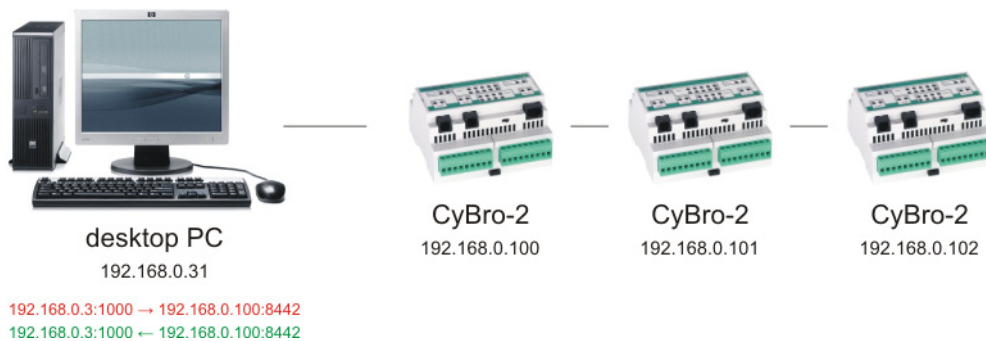


This connection is used when only two devices, PC and CyBro, are connected. Communication is using a limited broadcast address (255.255.255.255:65535).

PC needs a few minutes to set autoconfiguration address (169.254.x.x), in that period connection is not possible.

Direct connection may not work in a local network, because limited broadcast is generally ignored by routers.

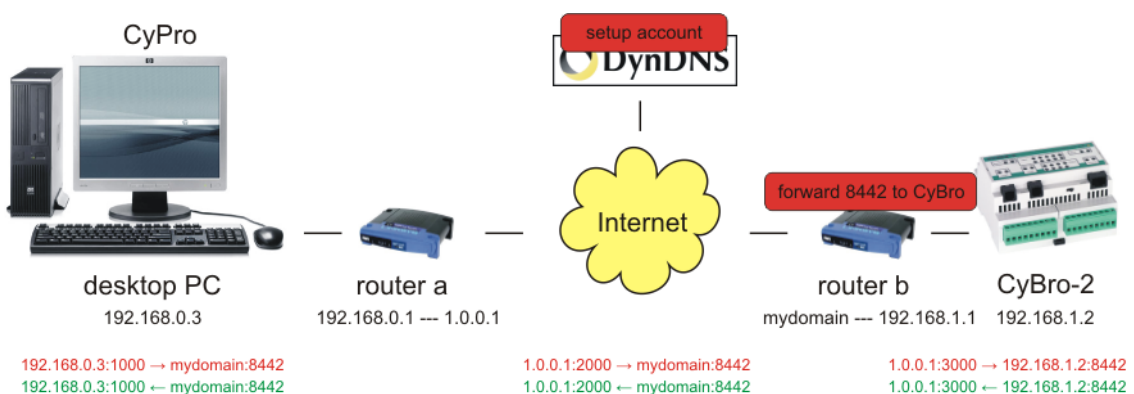
2. Local area connection



This is a typical setup for a small home/office network. All devices belong to same subnet. IP address may be dynamic (using DHCP server), or static.

CyPro will detect each IP address automatically, using direct broadcast (192.168.0.255) as first message.

3. Wide area connection using static IP or DynDNS



This connection provides programming and monitoring over the Internet.

1. CyPro Set Ethernet connection to wide area network, enter router b WAN address (85.10.5.58) or domain name into **Address** field. If needed, use **Extra timeout**, **Extra retries** and **Transaction id** options.

2. Router a No special settings needed.
3. Router b Forward UDP port 8442 to CyBro:



CyBro-2

- **Game or Application Name**

New Name:

- **Game or Application Definition**

A game or application is made of one or more TCP/UDP port ranges. Each incoming port range can be translated into a different internal (local network) port range. Port ranges can be statically assigned to devices or dynamically assigned using an outgoing trigger.

Protocol	Port Range	Translate To ...	Trigger Protocol	Trigger Port
<i>No port maps defined for this game or application.</i>				
UDP	8442 to 8442	8442	Any	<input type="text"/>
<input type="button" value="Add"/>				



Game & Application Sharing

This page summarizes the games and applications defined on your SpeedTouch. Each game or application can be assigned to a device on your local network.

- **Universal Plug and Play**

Universal Plug and Play (UPnP) is a technology that enables seamless operation of a wide range of games and messaging applications.

Use UPnP: Yes

Use Extended Security: No

- **Assigned Games & Applications**

The table below shows the games and applications that are allowed to be initiated from the Internet.

You need to configure such games or applications if you like to act as a game server or share a server located on your local network with other people.

If you are simply a player or simply accessing the Internet, you don't need to configure games or applications.

Game or Application	Device	Log
CyBro-2	CyBro-2-6511	off
uTorrent	athlon64	off
Web Server (HTTP)	athlon64	off

If more than one CyBro is in the local network, one solution may be to assign port 8442 to broadcast address (e.g. 192.168.0.255), but some routers will not support this.

4. CyBro No special settings needed. Either DHCP or static IP may be used.

In a small home/office network connected using ADSL, fixed IP address may not be available, so dynamic DNS service may be used.

5. DNS service Register to a dynamic DNS service, such as www.dyndns.com. Choose a domain. Some domains (e.g. xxx.getmyip.com) are available free of charge.

6. DNS client Configure a dynamic DNS client on a router b:



Dynamic DNS Service

Dynamic DNS can be used to point a fixed host name (e.g. host.a-domain.com) to the public (or WAN) IP address assigned by your Internet Service Provider (typically a dynamic IP address). This allows servers located on your Local Network (configured using Game & Application Sharing) to be accessible using this alias rather than the IP address assigned by your Internet Service Provider.

- **Configuration**

Use DynDNS: Yes

Internet Service: Internet

Username: damirskrijanec

Password: *****

IP address: 85.10.56.170

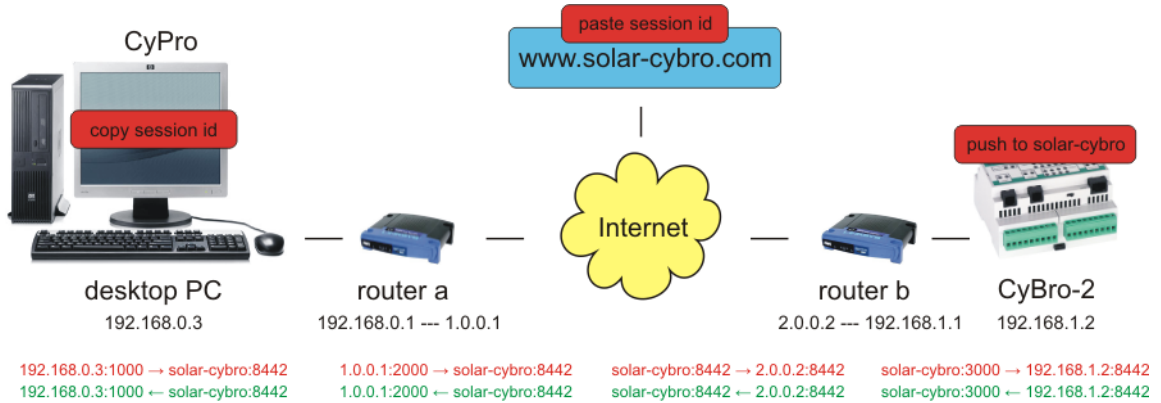
Dynamic DNS service: dyndns

Hostname: damir.getmyip.com (Update successful)

If router has no DynDNS support, install update client on a local PC. If firewall is active, allow outgoing traffic and open UDP port 8442 for incoming connections.

When connection is made, remote operation is same as in a local network. Hardware autodetect, program send, on-line monitor, kernel maintenance - all functions are available.

4. Wide area connection using relay server



This connection provides programming and monitoring over the Internet, using relay server as middle-point. Relay function is implemented in CyBroWebScada v1.1.1.

1. CyBro Configure push message to relay server (e.g. www.solar-cybro.com).
2. CyPro Set connection to Wide area network (common), enter address of relay server (e.g. www.solar-cybro.com), and press "Copy session ID". It is recommended to use **Extra timeout**, **Extra retries** and **Transaction id** options.
3. Server Enable relay, Set new session id and paste your id.

✖ Disable relay Set new session ID Reset

Relay enabled: ✔
 Session ID: 618076182
 Message count Tx: 10
 Message count Rx: 11
 Last message at: 2010-10-02 19:20:46
 Last controller: c6512

User controllers

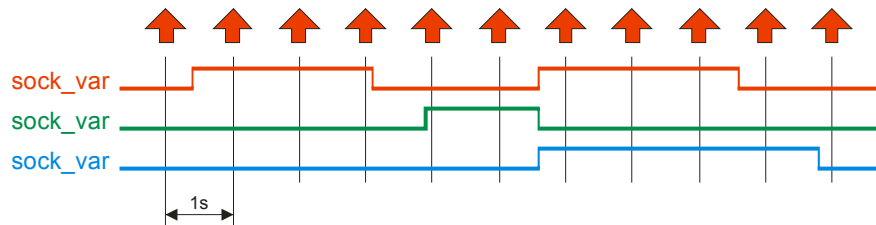
NAD	Plant	In push list	Ping result
c8785	Demo Plant	✔	-
c9859	Primel plant	✔	-
c6512	-	✔	✔

Use **Ping** command to check connection between server and CyBro.

Ethernet sockets

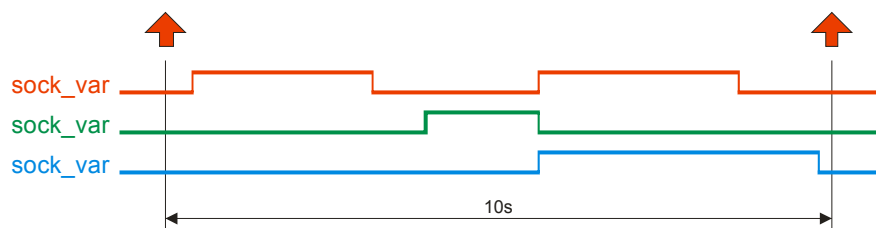
Sockets are used for CyBro-CyBro communication. User has to define a list of variables on both sides, and select mode of communication. Four modes are available:

1. Periodic 1s



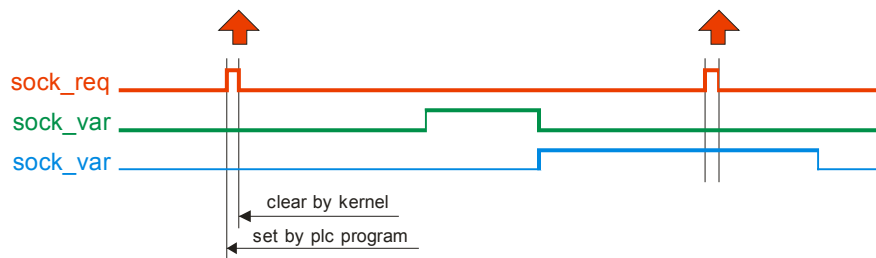
Socket is transmitted periodically, once in a second.

2. Periodic 10s



Socket is transmitted periodically, once in 10 seconds.

3. On-request



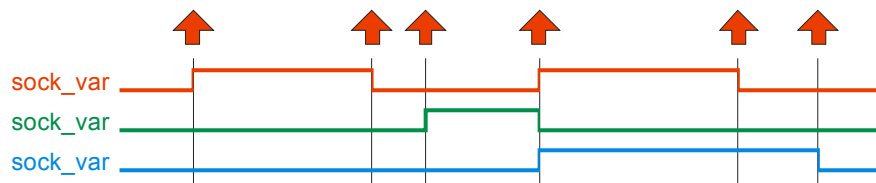
Socket is transmitted on request.

Transmission begins when plc program set the request bit. Kernel responds by clearing the request and sending the socket.

Request bit is the first bit variable in the socket. By convention, request bit is always sent as 1. Received socket will not cause retransmission. Request is automatically cleared after the scan.

To trigger transmission by an external device (e.g. another CyBro), use a second socket. Request variable can not be common, because it is automatically cleared after reception.

4. On-change



Socket is transmitted each time an socket variable is changed. Received socket will not cause retransmission.

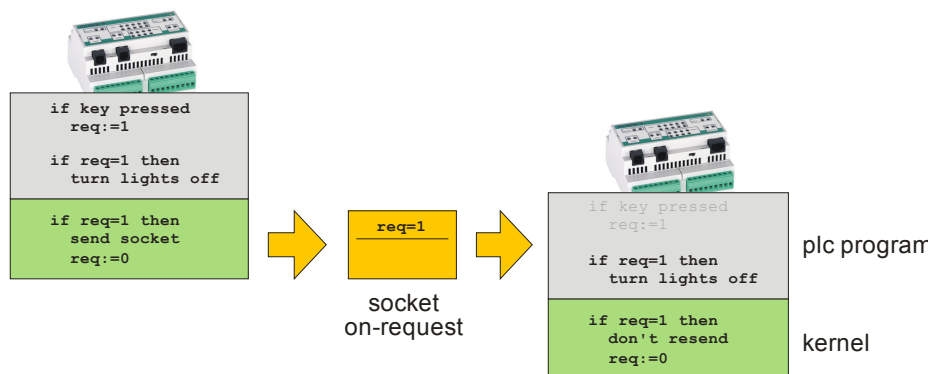
Sockets are transmitted only when plc is running.

Socket examples

Event-driven action

On-request output socket may be used to send a single event to the network. Each controller may trigger the event, each one will receive the event, and the request will reset automatically. Number of controllers is not limited.

The example shows an event to turn all lights off.

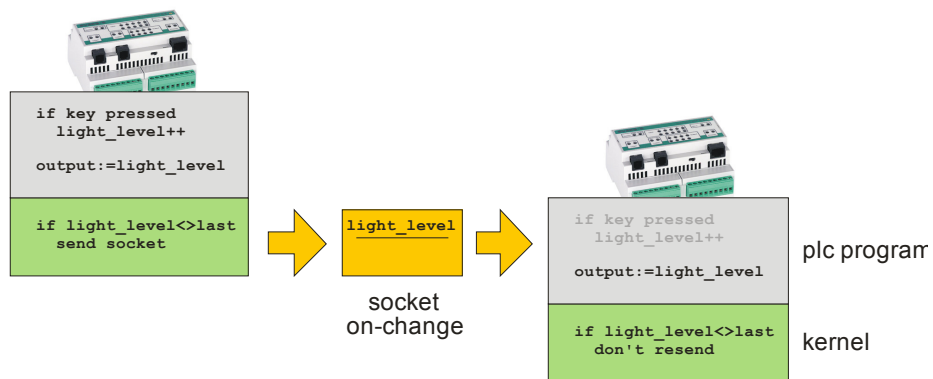


Each CyBro may have the similar program, although local i/o assignment may not be the same. Program structure is simple because request handling is fully automatic - once activated, network take care first to spread the request, and then to reset it.

Synchronized value

On-change output socket may be used to synchronize a value common for multiple controllers. Each controller may modify the value, and each one will receive the last modified value. Number of controllers is not limited.

The example shows a common lightness setting (0-100%) in a large hall.



Each CyBro has the same program, although local i/o assignment may be different. Because of automatic synchronization, program structure is very simple.

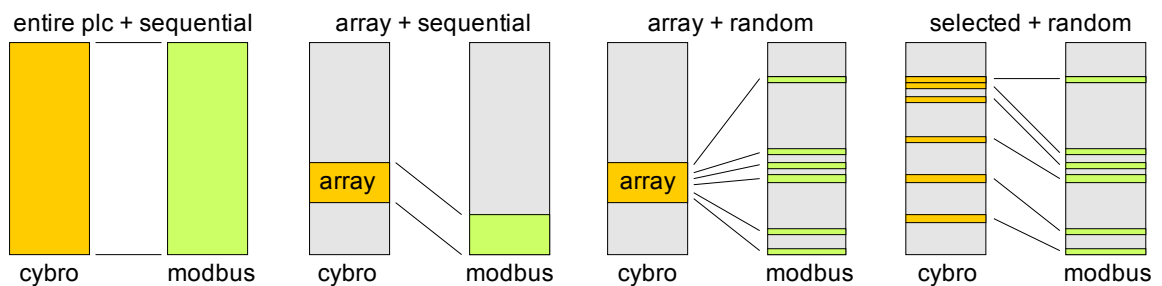
MODBUS connection

Modbus is a serial communication protocol published by Modicon in 1979 for use with programmable logic controllers (PLCs). It has become a de facto standard communication protocol in industry, and is now the most commonly available means of connecting industrial electronic devices. Modbus allows for communication between many devices connected to the same network.

CyBro supports:

- Modbus RTU slave (232/485)
- Modbus TCP slave (Ethernet)

Modbus data model describes how modbus coils and registers are translated to CyBro memory.



Appropriate model allows easy handling of modbus devices.

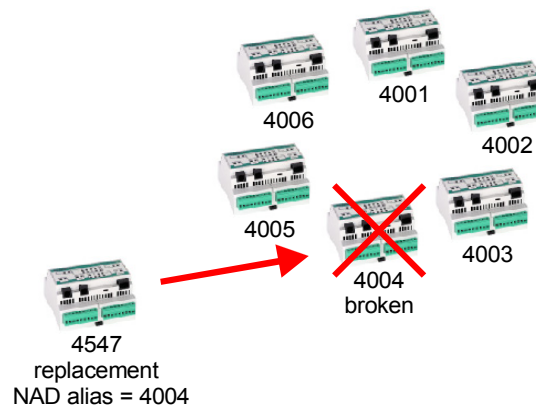
If Modbus master is needed, use InverterModbusDemo.cyp from \Examples.

Additional features

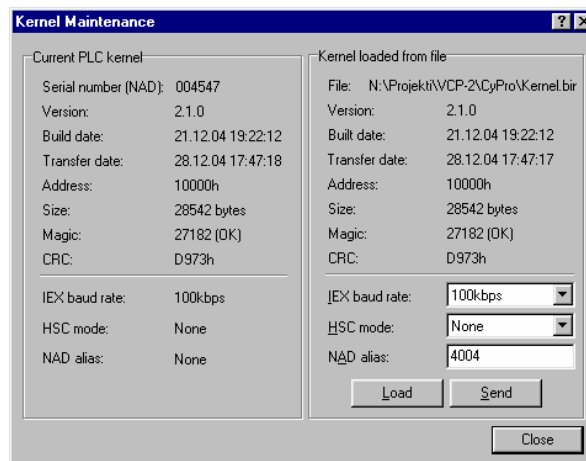
NAD alias

Each Cybro unit has its unique serial number, used also as communication address (NAD). Serial number is permanent and can not be changed.

NAD alias is a second communication address configurable by user. Once set, alias functions as the original NAD. CyBro with alias may be addressed with both serial number and alias.



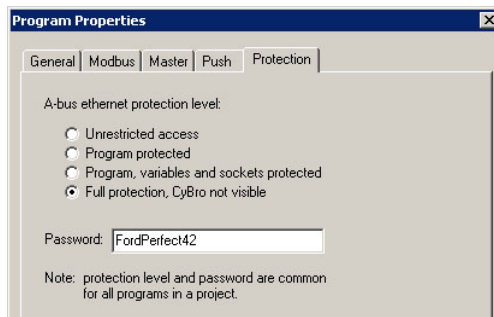
To enter NAD alias, open **Kernel Maintenance**, enter desired alias address and send the kernel.



To determine actual address, open **Get PLC Info** dialog box. Loader tab shows the original NAD (serial number) and Kernel tab shows alias.

Password protection

Beginning with version 2.5.0, CyBro has option to restrict access with password. Depending on selected protection level, protection may extend to a plc program, variables and sockets. For example, if protection level is **Program protected**, anybody can freely read and write variables, but will need password to send a new program.



Password protection is effective only for Ethernet. Serial access is not restricted, even if CyBro is configured for a full protection.

Password may contain any combination of letters and numbers of reasonable length. It is case sensitive. Don't use spaces, special symbols or national characters.

Password is common for all programs in project, it is not possible to define a separate password for each CyBro. Password stored in project file is not secure, so keep project safe.

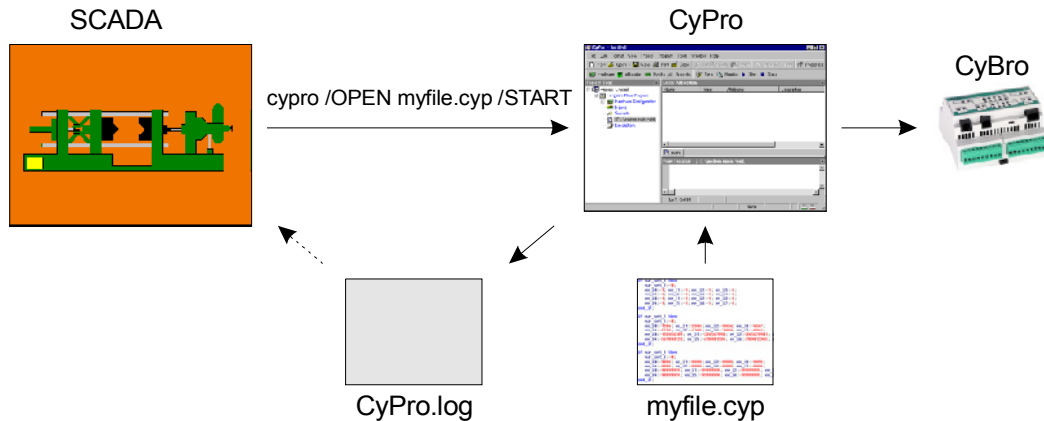
When password is active, communication option **Transaction id** can not be used.

To send a new password, use command **Erase protected program**.

Command-line options

Command-line options may be specified upon starting CyPro. They are used to automatically perform some tasks, such as sending a program to a connected PLC.

Using command-line options, CyPro may be used as external compiler for a shell application.



There are two general styles to use command line options:

```
cypro.exe filename.cyp
cypro.exe /OPTION1 /OPTION2 /OPTION3...
```

First style is used by operating system, when user double-clicks a cyp file.

Command-line options are:

/NEW [filename.cyp]	Create a new project. Filename is optional. If omitted, "untitled" is used instead.
/OPEN filename.cyp	Open existing project with specified filename.
/SAVE	Save project.
/SAVEAS filename.cyp	Save project under specified name.
/EXIT	Exit CyPro.
/NAD number	Select program. If specified NAD exists, that program will be selected, otherwise NAD is appended to current program.
/AUTODETECT	Hardware autodetect.
/START	Compile, send (only if different) and run.
/STARTALL	Start all programs in project.
/STOP	Stop current program.
/SEND	Send current program.
/HIDDEN	Silent operation, do not show any window or dialog box.

Filename may be just a name or a full path. If filename contain spaces, quotas should be used ("my file.cyp"). If an operation requires user input to continue execution, default option will be used

automatically. For example, when autodetect asks a network address, default address (zero) will be used automatically.

When started with command-line options, CyPro creates log file "CyPro.log" that contains all given commands and their results (success or failure). Log file is saved in CyPro directory (c:\Program Files\CyPro\CyPro.log).

If /HIDDEN mode is used, CyPro will automatically exit after last command is executed.

When using command-line options, it is advisable to turn on checkbox **Allow multiple instances** in **Environment Options**. If only a single instance is allowed, and CyPro is already running, command-line requests will be proceeded to the active copy.

Examples:

```
cypro.exe myfile.cyp
```

Start CyPro and open project myfile.cyp.

```
cypro.exe "c:\My Documents\myfile.cyp"
```

Start CyPro and open project myfile.cyp in specified directory. As path may contain spaces, quotas are required.

```
cypro.exe /HIDDEN /OPEN "myfile.cyp" /START /EXIT
```

Start CyPro, open an existing project (myfile.cyp), start PLC (compile, send & run) and exit. Operation is hidden, no window or dialog box will appear. Possible errors are saved in CyPro.log.

```
cypro.exe /HIDDEN /NEW /AUTODETECT /SAVEAS "myfile.cyp" /EXIT
```

Start CyPro, open a new project, start Autodetect, save as myfile.cyp and exit. Operation is hidden, no window or dialog box will appear. Possible errors are saved in CyPro.log.

```
cypro.exe /HIDDEN /NEW /NAD 4000 /AUTODETECT /SAVEAS "myfile.cyp" /EXIT
```

Start CyPro, open a new project, add new NAD, start Autodetect to detect connected IEX-2 modules, save as myfile.cyp and exit. Operation is invisible, no window or dialog box appears. Possible errors are saved in CyPro.log.

```
cypro.exe /HIDDEN /OPEN "myfile.cyp" /AUTODETECT /START /EXIT
```

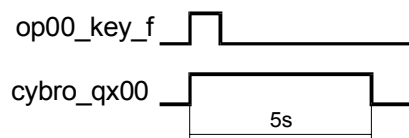
Start CyPro, open an existing project (myfile.cyp), start Autodetect (assuming the project has no defined hardware setup and network address, like CyPro examples), start PLC (compile, send & run) and exit. Original file will remain unchanged. Operation is invisible, no window or dialog box will appear. Possible errors are saved in CyPro.log.

CyBro tutorial

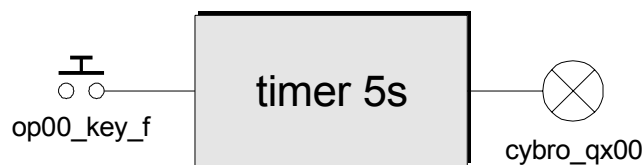
Your first CyBro program

Step one: define problem

In the first example, we will take a very simple task: a timer activated by a key press. By pressing a key, timer will turn the output on for a predefined time period, about 5 seconds.



A key for activating the timer can be any binary input, but it is simpler to use one of the display keys. The output is the first binary output, cybro_qx00.

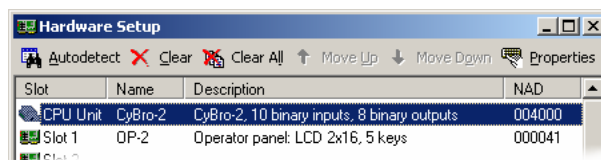


Step two: define hardware

Connect CyBro and PC to local network.

Start CyPro and select **File/New Project** to start a new project. Open **Hardware Setup** and start **Autodetect**. Select your CyBro from the list, and press **OK**.

Autodetect will list connected IEX-2 modules. In our example, it will be CyBro and OP-2.



To accept autodetected hardware press **OK**.

Step three: allocate variables

Our simple project needs one variable of timer type.

Start **Allocation Editor**, and press **Insert** to **Insert New Variable**. Dialog box will appear:

Enter name, choose type, adjust preset value, choose pulse type, select 100ms base and press **OK**.

Step four: write code

PLC code should connect the timer input to the key, and the timer output to the output relay. This can be done by:

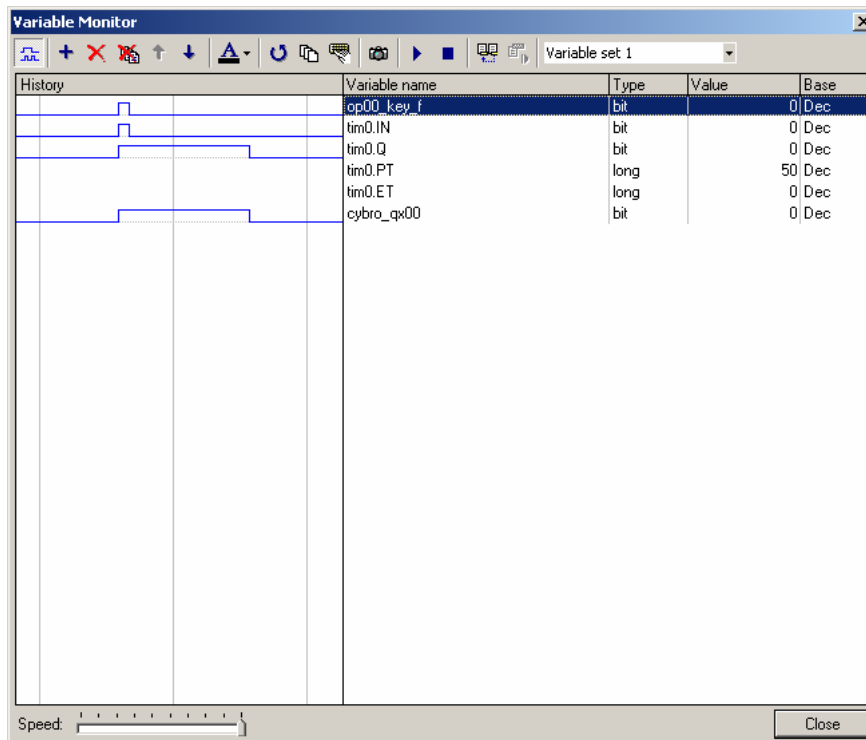
```
tim0.in:=op00_key_f;
cybro_qx00:=tim.q;
```

Step five: send and run

To compile and transfer the program to the CyBro, press **Start** button. Status line indicator will show that the program is running.

PLC program started. Modified 004000 Run

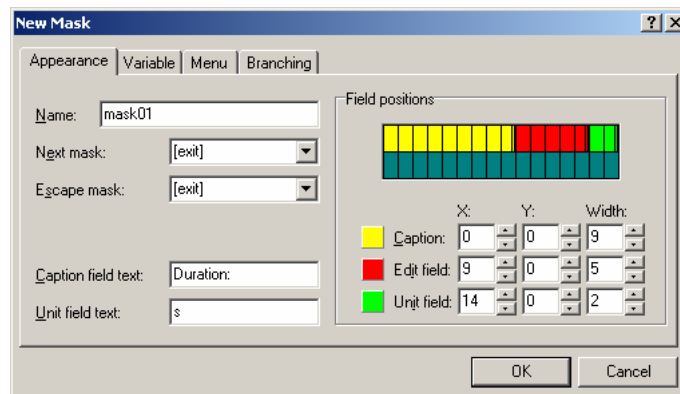
To check operation, start **Variable Monitor**, add allocated variables, and press "F" shortly.



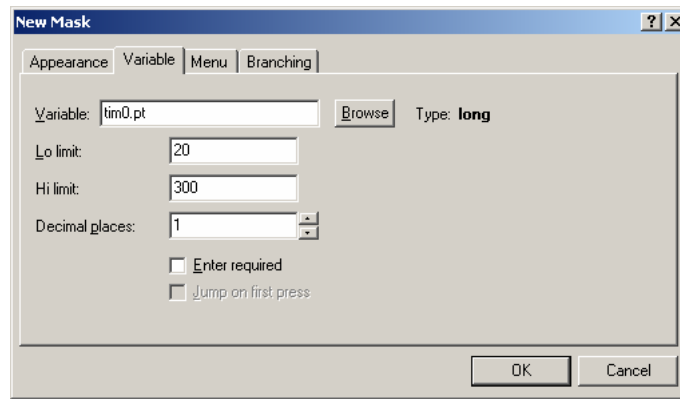
Step six: new frontiers

Another task is to make timer adjustable.

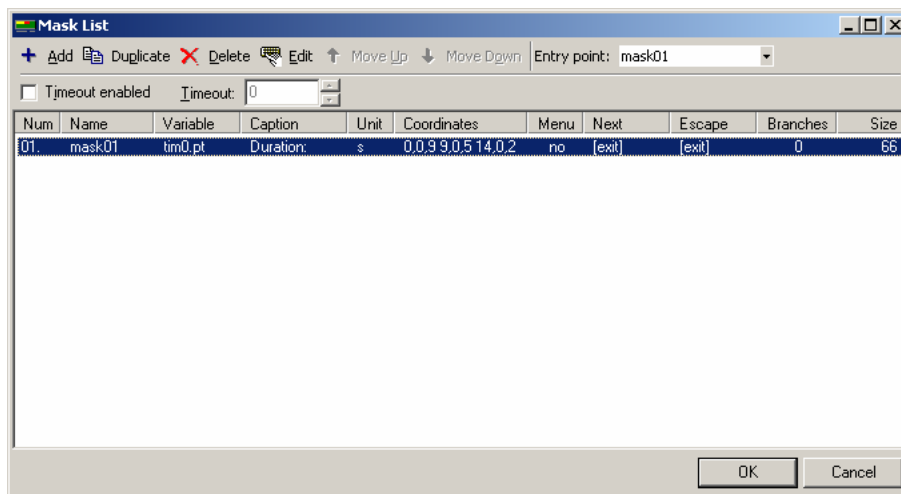
A way to accomplish this is the mask system. Start **Mask Editor** and press **Add** to create a new mask. Enter caption and unit field texts. Other fields may retain default values.



Switch to **Variable** tab, enter tim0.pt variable and adjust boundaries. Note that timer resolution is 100ms, so one decimal place is required to indicate seconds.

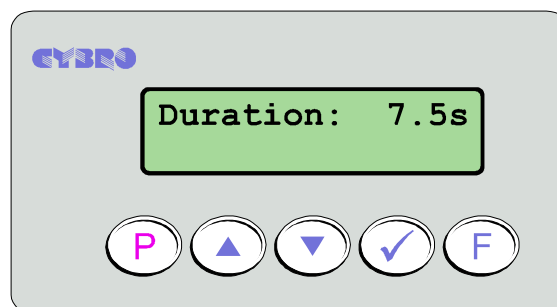


Press **OK**. Set entry point to mask01 and press **OK** again.



Press **Start** button. The program is automatically compiled, transferred and started.

To adjust timer value, press the **P** key, adjust with **up** and **dn** and exit with **P** again.



To activate timer press the **F** key.

Appendix

Data types summary

Elementary

type	width	range
bit	1	0..1
word	16	-
integer	16	-32768..32767
long	32	-2147483648..2147483647
real	32	$-3.4 \times 10^{38} \dots 3.4 \times 10^{38}$

Input/Output

type	width	equal type	description
in bit	1	bit	binary input
out bit	1	bit	binary output
in word	16	integer	analog input
out word	16	integer	analog output

Timer

field	type	direction	description
in	bit	input	input
pt	long	input	preset time
et	long	output	elapsed time
q	bit	output	output

Internal variables

name	type	direction	description
first_scan	bit	read only	active during first scan only
scan_overrun	bit	read only	scan timeout occurred
clock_10ms	bit	read only	10ms clock
clock_100ms	bit	read only	100ms clock
clock_1s	bit	read only	1s clock
clock_10s	bit	read only	10s clock
clock_1min	bit	read only	1min clock
retentive_fail	bit	read/write	indicates that retentive memory has failed
cybro_outputs_off	bit	read/write	if active, all CyBro binary outputs goes off
disconnect_inputs	bit	read/write	if active, inputs may be manipulated manually
no_input_filter	bit	read/write	if active, local 5ms input filter is turned off
rtc_read_req	bit	read/write	request read from RTC
rtc_write_req	bit	read/write	request writing to RTC
ee_read_req	bit	read/write	set to read all EE variables from EEPROM
ee_write_req	bit	read/write	set to write all EE variables to EEPROM
ee_write_magic	int	read/write	set to 31415 to enable writing to EEPROM
scan_time	int	read only	last scan execution time [ms]
scan_time_max	int	read only	max. scan execution time encountered [ms]
scan_frequency	int	read only	number of scans per second
rtc_sec	int	read/write	RTC second
rtc_min	int	read/write	RTC minute
rtc_hour	int	read/write	RTC hour
rtc_weekday	int	read/write	RTC weekday
rtc_date	int	read/write	RTC date
rtc_month	int	read/write	RTC month
rtc_year	int	read/write	RTC year

Instruction list summary

Instructions

Move

ld	move variable or constant to accumulator
ldn	move complement of variable to accumulator
st	move accumulator to variable
stn	move complement of accumulator to variable
set	set accumulator or variable
setc	if condition true set variable
res	clear accumulator or variable
resc	if condition true clear variable

Logic

cpl	complement accumulator or variable
and	logical and accumulator with variable or constant
andn	logical and accumulator with complement of variable or constant
or	logical or accumulator with variable or constant
orn	logical or accumulator with complement of variable or constant
xor	exclusive or accumulator with variable or constant
xorn	exclusive or accumulator with complement of variable or constant
shl	shift left accumulator, set LSB to zero
shr	shift right accumulator, set MSB to zero
rol	rotate left accumulator, copy MSB to LSB
ror	rotate right accumulator, copy LSB to MSB
fp	positive flank, 1 if low-to-high transition detected, 0 otherwise
fn	negative flank, 1 if high-to-low transition detected, 0 otherwise

Arithmetic

neg	change sign of accumulator
add	add variable or constant to accumulator
sub	subtract variable or constant from accumulator
mul	multiply accumulator with variable or constant
div	divide accumulator with variable or constant
mod	remains of dividing accumulator with variable or constant

Compare

eq	test if accumulator equal to value
ne	test if accumulator not equal to value
gt	test if accumulator greater than value
ge	test if accumulator greater or equal value
lt	test if accumulator lower than value
le	test if accumulator lower or equal value

Branch

jmp label	unconditional jump to position indicated by label
jmpc label	jump if condition true
jmpnc label	jump if condition not true
cal subroutine	call subroutine
calc subroutine	call subroutine if condition is true
calnc subroutine	call subroutine if condition is not true

Type conversions:

x-to-y convert acc from type x to type y (**bit**, **word**, **integer**, **long**, **real**)

Allowed type conversions

	bit	word	int	long	real
bit		+	+	+	
word			+	+	
int		+		+	+
long		+	+		+
real			+	+	

Allowed type combinations

	bit	word	int	long	real	acc	const	var
ld	+	+	+	+	+		+	+
ldn	+							+
st	+	+	+	+	+			+
stn	+							+
set	+					+		+
setc	+							+
res	+					+		+
resc	+							+
cpl	+	+				+		+
and	+	+					+	+
andn	+	+					+	+
or	+	+					+	+
orn	+	+					+	+
xor	+	+					+	+
xorn	+	+					+	+
shl		+				+		
shr		+				+		
rol		+				+		
ror		+				+		
fp	+					+		+
fn	+					+		+
neg			+	+	+	+		
add			+	+	+		+	+
sub			+	+	+		+	+
mul			+	+	+		+	+
div			+	+	+		+	+
mod			+	+			+	+
eq	+	+	+	+	+		+	+
ne	+	+	+	+	+		+	+
gt			+	+	+		+	+
ge			+	+	+		+	+
lt			+	+	+		+	+
le			+	+	+		+	+
jmp							+	
jmpc							+	
jmpnc							+	
cal							+	
calc							+	
calnc							+	
x-to-y	+	+	+	+	+	+		
dprnx	+		+	+	+			+

Structured text summary

Operators

operator	alias	unary	binary	bit	word	int	long	real	result
+			•			•	•	•	same
-		•	•			•	•	•	same
*			•			•	•	•	same
/			•			•	•	•	same
mod	%		•			•	•		same
not	!	•		•	•				same
and	&		•	•	•				same
or			•	•	•				same
xor	^		•	•	•				same
=	==		•	•	•	•	•	•	bit
<>	!=		•	•	•	•	•	•	bit
<			•			•	•	•	bit
<=			•			•	•	•	bit
>			•			•	•	•	bit
>=			•			•	•	•	bit
:=			•	•	•	•	•	•	same

Flow control

if...then...else

```

if <expression> then
  <statements>;
elseif <expression> then
  <statements>;
else
  <statements>;
end_if;

```

case...of

```

case <expression> of
  <value1>: <statements>;
  <value2>: <statements>;
  ...
  <valuen>: <statements>;
else
  <statements>;
end_case;

```

for...do

```

for <var>:=<expression> to <expression> do
  <statements>;
end_for;

```

while...do

```

while <expression> do
  <statements>;
end_while;

```

Edge detect functions

positive edge detect

```
fp(b:bit):bit;
```

negative edge detect

```
fn(b:bit):bit;
```

Cast functions

```
int(expression):int;
word(expression):word;
long(expression):long;
real(expression):real;
```

Display functions

clear display

```
dclr(slot:int);
```

print ASCII character

```
dprnc(slot:int, x:int, y:int, c:char);
```

print string

```
dprns(slot:int, x:int, y:int, str:string);
```

print binary value

```
dprnb(slot:int, x:int, y:int, c0:char, c1:char, value:bit);
```

print integer value

```
dprni(slot:int, x:int, y:int, width:int, zeroblank:bit, value:int);
```

print long value

```
dprnl(slot:int, x:int, y:int, width:int, zeroblank:bit, value:long);
```

print real value

```
dprnr(slot:int, x:int, y:int, width:int, dec:int, value:real);
```

Legend:

slot..... slot number
x..... x position (0-left)
y..... y position (0-top)
width..... print width
zeroblank.... suppress leading zero (0-no, 1-yes)
dec..... decimal places
c..... single character
str array of characters
value..... value to print

Com2 functions

port selection

```
com_select(port:int); // 1-COM1, 2-COM2
```

transmit

```
tx_start(char_num:int);
tx_stop();
tx_count():int;
tx_active():bit;
```

receive

```
rx_start(beg_ch:char, end_ch:char, len:int, msg_tout:int, char_tout:int);
rx_stop();
rx_count():int;
rx_active():bit;
rx_status():int;
```

parse received message

```
rx_bufrd(position:int):int;
rx_bufwr(data:int, position:int):int;
tx_bufrd(position:int):int;
tx_bufwr(data:int, position:int):int;
rx_strcmp(position:int, str:string):bit;
rx_strpos(position:int, str:string):int;
rx_strtoi(position:int):int;
rx_strtol(position:int):long;
rx_strtor(position:int):real;
```

High speed counter functions

start/stop counting

```
hsc_start();
hsc_stop();
hsc_active():bit;
```

read/write counter value

```
hsc_read():long;
hsc_write(position:long);
```

set/reset high speed action

```
hsc_set_action(action:bit, variable:bit);
hsc_reset_action();
hsc_check_action():bit;
```

enable/disable counter reset on zero input

```
hsc_enable_zero();
hsc_disable_zero();
hsc_check_zero():bit;
```

detect and read position of zero input

```
hsc_detect_zero():bit;
hsc_read_zero():long;
```

Special functions

read/write current ip address

```
get_ip():long;  
set_ip(ip_address:long, subnet:long, gateway:long, dns_server:long);
```

read current address (equal to serial if no alias, otherwise equal to alias)

```
get_nad():long;
```

read unique serial number

```
get_serial():long;
```


Character set

High Low	0	2	3	4	5	6	7	A	B	C	D	E	F
0	10	0	0	P	'	P	-	9	3	x	p		
1	11	!	1	Q	a	q	.	7	7	4	ä	q	
2	12	"	2	R	b	r	'	ı	ı	ı	ı	ı	ı
3	13	#	3	S	c	s	ı	ı	ı	ı	ı	ı	ı
4	14	\$	4	T	d	t	.	ı	ı	ı	ı	ı	ı
5	15	%	5	U	e	u	.	ı	ı	ı	ı	ı	ı
6	16	&	6	F	v	f	ı	ı	ı	ı	ı	ı	ı
7	17	'	7	G	w	g	.	ı	ı	ı	ı	ı	ı
8		(8	H	x	h	.	ı	ı	ı	ı	ı	ı
9)	9	I	y	i	.	ı	ı	ı	ı	ı	ı
A		*	:	J	z	j	.	ı	ı	ı	ı	ı	ı
B		+	:	K	ı	k	.	ı	ı	ı	ı	ı	ı
C		,	<	L	ı	l	.	ı	ı	ı	ı	ı	ı
D		-	=	M	ı	m	.	ı	ı	ı	ı	ı	ı
E		.	>	N	ı	n	.	ı	ı	ı	ı	ı	ı
F		/	?	0	_	o	.	ı	ı	ı	ı	ı	ı

To enter characters not supported by your keyboard, press **Alt** key, on the numeric keypad type character code preceded by 0, and release **Alt**. Character code should be expressed in decimal. Num lock should be on.

Example:

According to character table, symbol "°" (degrees centigrade) has hex code DFh. Converting value to decimal gives 223 (DFh = D0h + 0Fh = 16*13 + 15 = 223).

To enter centigrade symbol:

- make sure num lock is on
- press **Alt**
- press consecutively **0223**
- release **Alt**

Because of different character sets, character "ß" appears instead of "°", but it will show correctly on LCD.

```
dprns(1,0,0,'T=xx.xßC');
dprnr(1,2,0,4,1,iw000*0.1);
```



Codes 0..7 are reserved for bar-graph or Latin-2 characters.

Keyboard shortcuts

Common

F1		Help
F2		Syntax check
F5		Hardware setup
F6		Allocation editor
F7		Mask editor
F8		Socket editor
F9		Send
F10		Variable monitor
Ctrl-F10		Data manager
F11		Start PLC program
F12		Stop PLC program
Ctrl-O		Open
Ctrl-S		Save
Ctrl-Shift-S		Save As
Ctrl-P		Print project
Ctrl-D		Connect/disconnect communication port
Ctrl-L		Select NAD
Ins		Insert (context sensitive)
Delete		Delete (context sensitive)
Ctrl-Up		Move item up
Ctrl-Dn		Move item down
Ctrl-Tab		Next window
Ctrl-Shift-Tab		Previous window
Ctrl-F4		Close window
Alt-F4		Exit program

Text editor

Ctrl-space		Insert variable or function
Ctrl-Z	Alt-Backspace	Undo
Shift-Ctrl-Z		Redo
Ctrl-X	Shift-Del	Cut
Ctrl-C	Ctrl-Insert	Copy
Ctrl-V	Shift-Insert	Paste
Ctrl-A		Select all
Ctrl-F		Find
F3		Find next
Ctrl-R		Replace
Ctrl-G		Go to line
Ctrl-Shift-I		Indent block
Ctrl-Shift-U		Unindent block
Ctrl-Shift-C		Comment/uncomment selection