Scripting language for PP2CAN

User manual

Compiler and interpreter version 3.02

CANLAB s.r.o. · www.canlab.cz

Contents

Introduction 5
Purpose of scripts 5
First script 7
Script structure 8
Script types 8
Variables and objects 9
Named constants 9
Writing a program 11
Characters and lines 11
Identifiers 11
Keywords 11
Comments 11
Statements and the semicolon 12
Blocks and braces 12
Data types and constants 13
Writing constants 13
Escape sequences 14
Hexadecimal and binary notation 14
Implicit conversions 15
Type cast 15
Expressions and operators 17
Assignment 17
Statements 19
if, else if, else 19
while 19
for 19
break and continue 20
switch 20
return 21
Stopping the script 21
Functions 22
User functions 22
Sections func_0 to func_9 23
Built-in functions 24
Conversion functions 24
String functions 27

Output 28
Arithmetic and mathematical operations 30
CAN interface 33
Date and time 33
Display and control of variables 34
Global variables 36
Other functions 37
Built-in constants 38
Objects 44
Object assignment 44
Adding to a container 53
Object comparison 54
obj_can_msg 54
obj_lin 58
obj_rs232 59
obj_vector_int 60
obj_vector_double 62
obj_deque_int 64
obj_deque_double 65
obj_timepoint 66
obj_csvfile 67
obj_cfgfile 68
obj_logfile 69
obj_datagrid 69
obj_dataset 70
obj_datagraph 71
obj_datagauge 72
obj_datahistogram 73
obj_socket_server 75
obj_socket_client 76
obj_pid 77
obj_memory 77
obj_dllfnc 79
obj_uds 81
obj_j1939 83
Transfer of long data (UDS, J1939) 85
Transmission 85
Reception 85

obj_j1939 protocols 85
Example: UDS read DID (0x22) 85
Example: J1939 transmission by RTS/CTS and BAM 86
LIN 88
Interpreter environment 89
Compiler 90
Warnings 90
Common error messages 90
Binary format 91
Example and test scripts 92
Self-checking tests 92
Examples and script pairs 92
List of scripts 92

Introduction

Purpose of scripts

Scripts in PP2CAN allow quick creation of custom CAN communication handling: reacting to received messages, sending data periodically, watching combinations of states from several messages and warning about them, configuring devices over CAN, communicating through UDS or J1939, working with LIN, the serial port, TCP/UDP and files. The scripting language is based on C and extended with built-in functions and objects for CAN messages, data containers, display windows and files.

The tool for writing and running scripts is in the Additional tools toolbar, item CAN Script N. Up to 4 scripts can be active at the same time; they can share data through global variables.

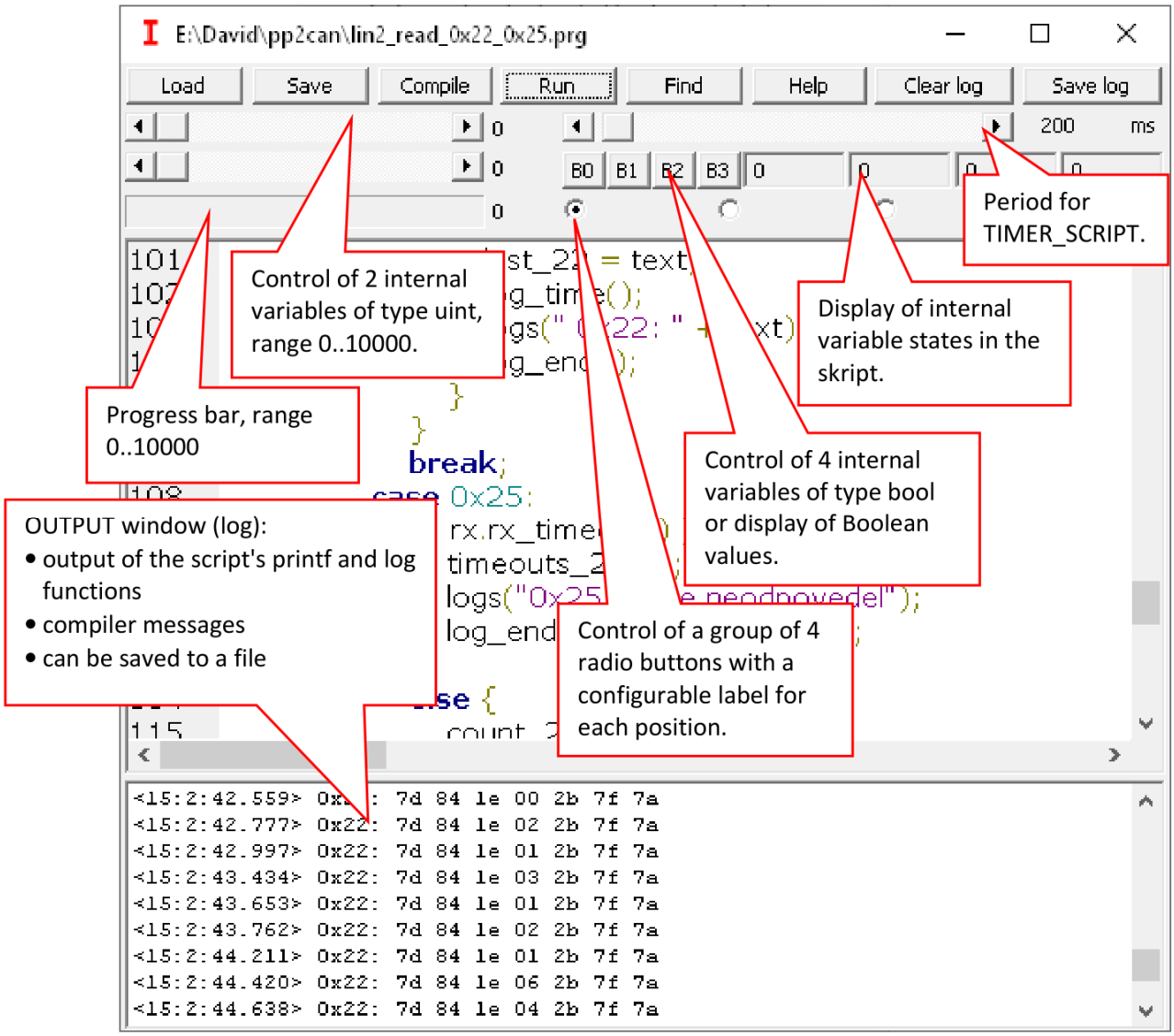


Fig. 1: CAN script window

Window element	Meaning
Load, Save	Loading and saving the script source text (.prg).
Compile	Compiling the script. Errors or the message Script ready are printed to the OUTPUT window.
Run	Starting and stopping the compiled script.
Help	Help window describing the language, functions and objects.
Clear log, Save log	Clearing and saving the content of the OUTPUT window.

Window element	Meaning
Period (ms)	Period of timer calls of the script (TIMER_SCRIPT and AND_TIMER scripts).
Two sliders	Values 0 to 10000 readable by ui_get_val0 and ui_get_val1.
Progress bar	Indicator 0 to 10000 set by ui_set_p.
Buttons B0 to B3	Four logical values (ui_get_b0 ... ui_set_b3).
Four value fields	Display of double values by ui_set_d0 to ui_set_d3.
Radio buttons	Group of 4 radio buttons with labels (ui_get_r, ui_set_r, ui_set_name_r).
OUTPUT window	Output of the print and log functions and compiler messages; can be saved to a file.

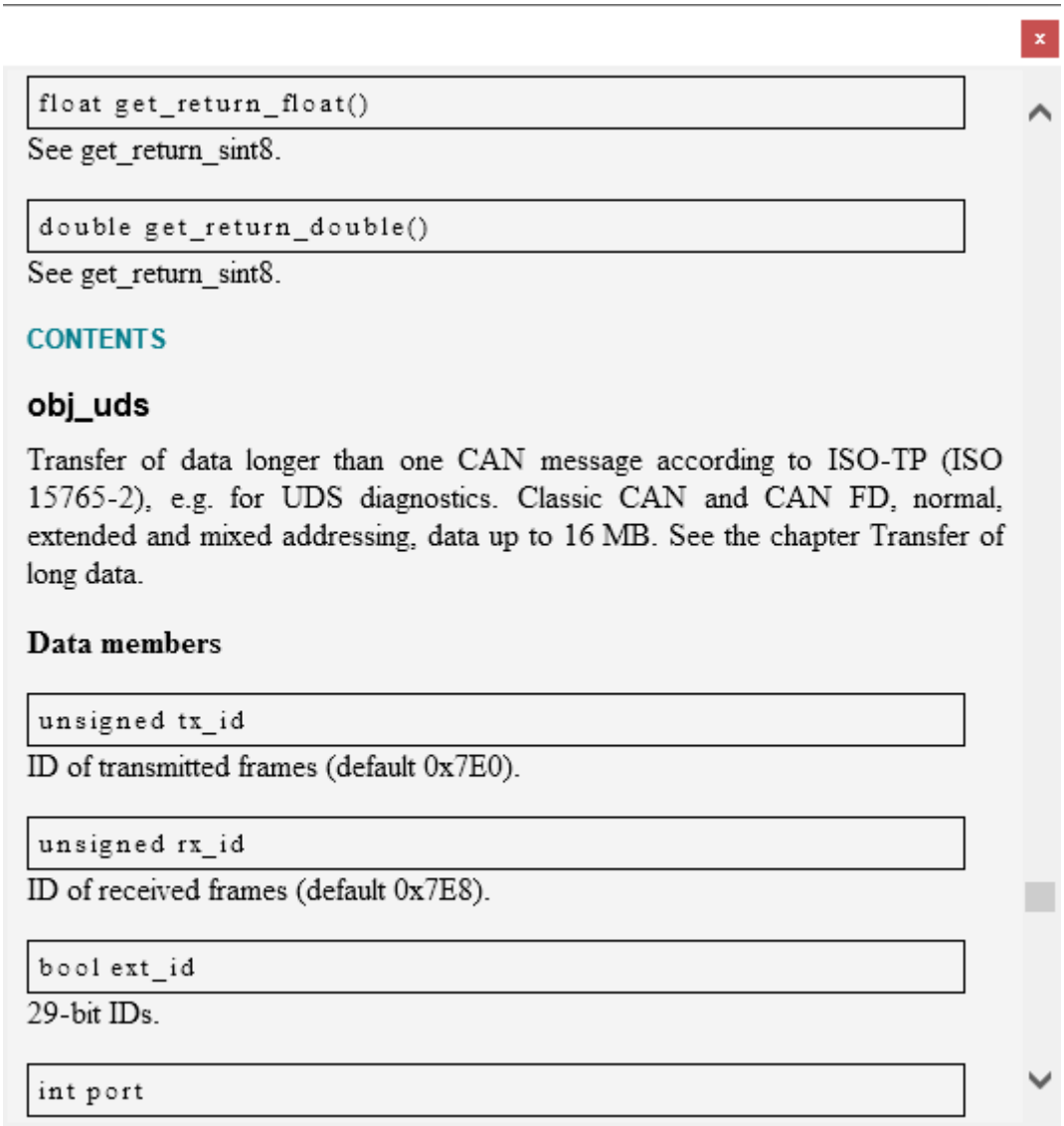


Fig. 2: CAN script help window

Use Ctrl+F to search in the help window. Scripts can be tried without a converter: with no interface connected, PP2CAN works in V2CAN mode (virtual loopback) in which sent messages come back as received.

The manual comes with a set of commented example and test scripts (chapter Example and test scripts). Notes "Example:" in the individual chapters refer to the related scripts. The manual describes the compiler and interpreter version 3.02.

First script

The following script sends a message with identifier 0x123 every 100 ms containing a 16-bit counter and prints every tenth transmission to the OUTPUT window.

```
script:
    type = TIMER_SCRIPT;        // body is called periodically
end
variables:
    unsigned counter;
end
objects:
    obj_can_msg msg;
end
init:
    counter = 0;
    msg.id1 = 0x123;
    msg.size = 2;
    ui_set_timer_script_period(100);
end
body:
    counter++;
    msg.set_uint16(0, counter);
    msg.send();
    if( counter % 10 == 0 ) {
        logs("Sent: ");
        logu(counter);
        log_endl();
    }
end
```

- The Compile button compiles the script; the OUTPUT window shows Script ready or a list of errors with line numbers.
- The Run button starts the script: the init section is executed and then the body section is called periodically.
- Pressing Run again stops the script (the shutdown section is executed if present).

Script structure

A script is a text file (extension .prg) divided into sections. Every section starts with its name followed by a colon and ends with the keyword end. The script section is mandatory; the other sections are used as needed. The order of the sections does not matter, but it is good practice to put the declarations of variables and objects first. Every section may appear only once in a script (otherwise the error Section "init" is defined more than once); there can be any number of user functions.

Section	Meaning
`script:`	Script properties, mainly the type: type = TIMER_SCRIPT;
`variables:`	Declarations of global variables and named constants (const).
`objects:`	Declarations of objects (CAN message, vector, window, file ...).
`init:`	Code executed once when the script starts.
`body:`	Main code, executed on every call of the script.
`shutdown:`	Code executed once when the script stops.
`func_0: to func_9:`	Simple functions without parameters, called by func(N);
`function ... end`	User function with parameters and a return value.

Skeleton of a script with all common sections:

```
script:
    type = TIMER_SCRIPT;
end

variables:
    const int LIMIT = 100;          // named constant
    int counter = 0;                // variable with an initial value
    double sum;
end

objects:
    obj_can_msg msg;
end

function double average(double a, double b)
    return (a + b) / 2;
end

init:
    msg.id1 = 0x100;
end

body:
    counter++;
    sum += 0.5;
    if( counter >= LIMIT ) {
        stop();
    }
end

shutdown:
    logs("Finished");
    log_endl();
end
```

Script types

The script type determines when PP2CAN calls the body section.

Type	Call of the body section
`CAN_FUNCTION`	For every received CAN message; the message is read by the load() method of obj_can_msg.

Type	Call of the body section
TIMER_SCRIPT	Periodically according to the period in the script window or ui_set_timer_script_period().
CAN_FUNCTION_AND_TIMER	For received CAN messages and periodically; distinguished by is_can_function_call() and is_timer_call().
DATA_GENERATOR_SCRIPT	Called by the PP2CAN data generator for the generated message.
LIN_FUNCTION	For every received LIN message; the message is read by the load() method of obj_lin.
LIN_FUNCTION_AND_TIMER	For received LIN messages and periodically; distinguished by is_lin_function_call() and is_timer_call().

Variables and objects

Variables are declared in the variables section and objects in the objects section, always in the form type name1, name2;. They are global: they are valid in all sections and functions and keep their values between calls of the body section. When the script starts, variables are 0, strings are empty and logical variables are false.

```
variables:
    int i, j;
    double voltage;
    string name;
end

objects:
    obj_can_msg rx_msg, tx_msg;
    obj_vector_int buffer;
end
```

Variables can be initialized directly in the declaration. The initializer can be any expression, including functions and previously declared variables and constants. Global variables are initialized at the beginning of the init section in the order of their declarations.

```
variables:
    int a = 5, b = a * 2, c;        // c = 0
    double gain = 0.5;
    string title = "ECU " + i2str(a);
    bool enabled = true;
end
```

The type of the initializer must be assignable to the variable according to the same rules as in the assignment statement (chapter Implicit conversions). Objects always start empty; their setup belongs to the init section.

Named constants

Named constants are defined in the variables section with the keyword const (constants cannot be defined in user functions, but global constants are valid there). A constant has a type and a value that cannot be changed, and it does not occupy a variable. In the script it behaves like a literal: it can be used in expressions, in case labels and case ranges, in variable initializers and in the definition of other constants.

```
variables:
    const int      MAX_NODES = 16;
    const int      OFFSET    = -40;
    const unsigned MASK      = 0xFF00;
    const char     SEPARATOR = ';';
    const double   GAIN      = 2.5e-3;
    const string   PREFIX    = "ECU_";
    const bool     DEBUG     = false;
    const int64    BIG       = -9000000000;
    const int      LAST_NODE = MAX_NODES;      // constant from a constant
    int node_count = MAX_NODES / 2;            // variable initialized by a constant
end
```

- The value of a constant must be a literal, a literal with unary minus or another constant of a matching type, within the range of the type. Otherwise the compiler reports Constant "X" must be initialized by a literal (or another constant) of a matching type within its range.
- A constant cannot be assigned. MAX_NODES = 5; is the error Constant "MAX_NODES" cannot be used here (a constant cannot be assigned).
- It is recommended to write constant names in capitals to distinguish them from variables.

> *Examples: T25_initialization_const.prg, T33_constants.prg*

Writing a program

This chapter describes the rules of the source text: characters, identifiers, keywords, comments, statements and blocks. The language is based on C, so most rules match C; the differences are mentioned in the text.

Characters and lines

The source is plain text in the Windows-1250 encoding (the Windows default encoding for Central European languages; for English text it is the same as ASCII). Spaces, tabs and line ends (CR LF or LF) only separate symbols and can be used freely: a statement can span several lines and one line can contain several statements. Indentation has no meaning for the program.

```
if( a > 0 ) { b = 1; c = 2; }        // several statements on one line

total = first_value
      + second_value
      + third_value;                 // one statement on several lines
```

Characters with diacritics can be used in strings and comments; use only English letters in identifiers. Line numbers in compiler messages correspond to the lines of the source file.

Identifiers

Names of variables, constants, objects and functions start with a letter or an underscore and can contain English letters, digits and the underscore. Upper and lower case are distinguished: count, Count and COUNT are three different names.

- An identifier must not be a keyword, the name of a built-in function (Cannot declare variable "sin", an identifier of the same name already exists) or the name of a built-in or named constant.
- A name can be used only once: a variable, object, constant and function must not have the same name.
- A local variable and a function parameter hide a global variable of the same name.

Keywords

Keywords are written in lower case and cannot be used as identifiers.

Group	Keywords
sections	`script variables objects init body shutdown func_0 … func_9 function end`
types	`bool char unsigned int unsigned64 int64 float double string` (aliases unsigned32, int32)
declarations	`const static ref void type`
statements	`if else while for switch case default break continue return`
values	`true false`
script types	`CAN_FUNCTION TIMER_SCRIPT CAN_FUNCTION_AND_TIMER DATA_GENERATOR_SCRIPT LIN_FUNCTION LIN_FUNCTION_AND_TIMER`
objects	object type names obj_... (chapter Objects)

Comments

A // comment lasts to the end of the line, a /* */ comment can span several lines. Comments cannot be nested. Inside a string or a character constant the characters // and /* do not start a comment.

```
// comment to the end of the line
/* comment over
   several lines */
x = 1;   /* comment inside a line */  y = 2;
s = "http://pp2can";               // // in a string is not a comment
```

Statements and the semicolon

Every simple statement (assignment, function call, return, break, continue) ends with a semicolon. A semicolon alone is an empty statement and does nothing; a semicolon after the closing brace of a block (if(a) { ... };) is therefore not an error.

Assignment is a statement in this language, not an expression: a = b = 0; and if((a = f()) > 0) cannot be written. A call of a function or method with a return value can stand alone as a statement; the result is then discarded.

```
a = 0; b = 0;                // instead of a = b = 0;
v.pop_back();                // returns the removed element, it is discarded
;                            // empty statement
```

Blocks and braces

A block is a sequence of statements in braces { }. The bodies of if, else, while and for are always blocks, even with a single statement (unlike C, if(a) b = 1; cannot be written). A block can also be empty: { }.

A block can also be used on its own wherever a statement can stand, most often as a branch of the switch statement (chapter switch). A standalone block does not create a new scope of variables; it only helps readability.

The placement of braces does not matter. All of the following forms are equivalent:

```
if( x < 0 ) {                // brace on the same line (K&R)
    r = -1;
} else {
    r = 1;
}

if( x < 0 )                  // braces on separate lines (Allman)
{
    r = -1;
}
else
{
    r = 1;
}

if(x<0){r=-1;}else{r=1;}     // everything on one line
```

- There can be a line end or a comment between } and else.
- Closing braces of nested blocks can follow each other: }}}.
- Every opening brace must have its closing brace. The compiler reports a missing } as Expected symbol "}" on the line where the block had to end at the latest (usually at the end of the section) and an extra } as Symbol "}" caused a syntax error.
- Sections and user functions are delimited by the word end, not by braces. The compiler reports a missing end as Expected symbol "end" at the beginning of the next section.

> *Examples: T30_braces_layout.prg, N14_if_without_braces.prg, N18_missing_closing_brace.prg*

Data types and constants

Type	Size	Range / meaning
`bool`	1 byte	true, false
`char`	8 bits	0 to 255 (byte or character)
`unsigned`	32 bits	0 to 4294967295 (alias unsigned32)
`int`	32 bits	-2147483648 to 2147483647 (alias int32)
`unsigned64`	64 bits	0 to 18446744073709551615
`int64`	64 bits	-9223372036854775808 to 9223372036854775807
`float`	32 bits	floating-point number, about 7 significant digits
`double`	64 bits	floating-point number, about 15 significant digits
`string`	max. 1023 characters	text string; a longer result is truncated

The char type is unsigned and is used for bytes and characters. The string type is a value, not a reference: the assignment s2 = s1; creates a copy of the text and a later change of s1 does not affect s2.

Writing constants

Form	Type	Example
integer number	unsigned, above 32 bits unsigned64	100, 4294967296
floating-point number (with a dot or an exponent)	float	1.5, 2.5e-3, 1.0e10, 1e3, 2E-3
0x..., hex(...)	char up to 0xFF, unsigned up to 0xFFFFFFFF, otherwise unsigned64	0xFF, 0x18EA0B66, hex(ABC)
0b..., bin(...)	as hexadecimal	0b1010, bin(11110000)
character in apostrophes	char	'A', '\n', '\x1B'
string in quotes	string	"text", "a\tb", "\"", "\\"
`true, false`	bool	

- A negative number is written with the unary minus: -1, -2.5. The constant 1 is of type unsigned, the expression -1 is of type int. The compiler evaluates expressions with constants (e.g. -1, -0x10, -(-5)) already at compile time.
- Floating-point constants are of type float. For a calculation with full double precision it is enough that one operand is double, e.g. (double)1 / 3.
- When a floating-point constant is converted to double (assignment to a double variable, an expression with double), the exact value of the written number is used, not the value rounded to float: double d = 0.1; gives exactly 0.1. A float variable keeps float precision.
- The exponent can be written with a lower or upper case E, with or without a sign, and also for a number without a decimal dot: 1e3 is the floating-point number 1000.0.
- An integer constant 0 to 255 can be assigned directly to a char variable or passed as a char parameter (c = 0;). Other unsigned values require a cast to char.
- Built-in constants are available in scripts, e.g. M_PI, M_INT32_MAX, COLOR_RED (chapter Built-in constants), as well as user named constants const (chapter Named constants).

> *Examples: T01_data_types_constants.prg, T24_escapes_literals.prg, T33_constants.prg*

Escape sequences

Special characters in character constants and strings are written with a backslash:

Form	Character	Code
\n	new line	10
\r	carriage return	13
\t	tab	9
\0	null character (only in a character constant)	0
\\	backslash	92
\'	apostrophe	39
\"	quote	34
\xHH	character with the given code, 1 or 2 hexadecimal digits	0 to 255

```
c = '\x1B';                  // ESC
rs.send_string("AT\r");      // command terminated by CR
s = "\x41\x42C";             // "ABC"
s = "Path: C:\\data\\log.csv";
```

A string ends with a null character, so \0 and \x00 cannot be used in a string (they are omitted in a string). Data with zero bytes belong to the objects obj_vector_int, obj_can_msg or obj_memory.

A backslash followed by another character (e.g. \U in "c:\Users") is not an escape sequence: the backslash and the character stay in the string and the compiler prints a warning. Up to version 3.01 both were silently omitted, so "c:\Users\data.csv" became "c:sersata.csv". In file paths write \\ or a forward slash / (Windows accepts it as well).

Hexadecimal and binary notation

An integer value can be written in hexadecimal or binary in two ways: with a prefix (0x..., 0b...) or as hex(...) and bin(...). The case of hexadecimal digits does not matter. The length is not limited to a byte; a constant can have up to 64 bits.

```
c  = 0xA5;                    // char
c  = bin(10100101);           // char, the same value 0xA5
u  = 0x1234ABCD;              // unsigned
u  = hex(DEADBEEF);           // unsigned
u  = 0b1010 | 0x80;           // bit operations with constants
ul = 0x123456789ABCDEF0;      // unsigned64
i  = -0x10;                   // int -16
msg.set_ID29(0x18FEF100);     // 29-bit identifier
```

Constant value	Constant type	Can be assigned to
0x00 to 0xFF	char	char, unsigned, int, unsigned64, int64, float, double
0x100 to 0xFFFFFFFF	unsigned	unsigned, int, unsigned64, int64, float, double
above 0xFFFFFFFF	unsigned64	unsigned64, int64, double

- Binary constants follow the same rules according to their value (0b11111111 is char, 0b100000000 is unsigned).
- Assigning a larger type to a smaller one is a compile error: neither c = 0x1FF; nor u = 0x100000000; is allowed. The value can be narrowed by a cast: c = (char)0x1FF;.
- Assignment to a signed type takes the value bitwise: i = 0xFFFFFFFF; gives -1, l = 0x8000000000000000; gives the smallest int64.
- Values can also be printed in hexadecimal and binary: logc_hex, logc_bin, logu_hex, logu_bin, logu64_hex, logu64_bin; the functions u2hex and u642hex convert them to text.

- A string with a hexadecimal number is converted to a value by from_hex("1A2B"); hexadecimal data are accepted by the methods data_from_hex (obj_can_msg) and from_hex_string (obj_vector_int).

Implicit conversions

The types are ordered bool < char < unsigned < int < float < unsigned64 < int64 < double. In an expression with operands of different types the smaller type is converted to the larger one automatically; float with a 64-bit integer gives double and int with unsigned64 gives int64. char can be converted to string automatically.

Assignment, initialization in a declaration and passing a parameter to a function allow only a conversion to a larger type. A conversion to a smaller type (e.g. double to int, int to unsigned) is the compile error Operator "=" has operands of wrong type and requires a cast.

Conversion	Result
`bool → char`	character '1' (code 49) for true, character '0' (code 48) for false – see the exception below
`bool → unsigned, int, 64-bit types, float, double`	1 for true, 0 for false
`char → unsigned, int, ...`	value 0 to 255
`unsigned → int`	bitwise: a value above 2147483647 becomes negative
`int → int64, double`	signed value
`int → unsigned64`	bitwise as in C: -1 gives 18446744073709551615
`integer → float`	float has only a 24-bit mantissa: 16777217 is stored as 16777216
`char → string`	one-character string

> *Exception bool → char: converting a logical value to char gives the character '1' or '0' (code 49 or 48), not the number 1 or 0 as with the other types. This applies to the assignment c = b; and to the cast (char)b. For a numeric value convert to int first: c = (char)(int)b; or i = b;.*
>
> *Caution: when int is compared with unsigned, both sides are converted to int. An unsigned value above 2147483647 then becomes negative. The solution is a comparison in int64: (int64)u > i. Examples: T03_implicit_conversions.prg, T32_assign_data.prg*

Type cast

The form (type)expression converts a value to the given type, also to a smaller one. A cast has a higher priority than the arithmetic operators: (int)d * 2 means ((int)d) * 2.

- An integer to a smaller integer: the upper bits are cut off, e.g. (char)300 = 44, (char)-1 = 255, (int)4294967297 = 1.
- A signed integer to an unsigned type and vice versa: bitwise, (unsigned)-1 = 4294967295, (int)4294967295 = -1.
- A floating-point number to an integer: the fractional part is truncated toward zero, (int)2.99 = 2, (int)-3.9 = -3; values out of range are limited to the minimum or maximum of the type, (int)1e20 = 2147483647, NaN gives 0.
- To bool: the result is true if the value is not zero; (bool)0.001 is true.
- bool to char: the character '1' or '0' (the exception described in chapter Implicit conversions).
- Only char can be cast to string: (string)c. Numbers are converted to text by i2str, d2str and other functions (chapter Conversion functions).

```
c = (char)i;              // lowest byte
i = (int)d;               // truncation of the fractional part
l = (int64)u * 4;         // calculation in 64 bits
d = (double)a / b;        // division in double
```

```
u = (unsigned)i;           // bitwise, -1 = 4294967295
```

> *Examples: T04_explicit_casts.prg, T32_assign_data.prg*

Expressions and operators

An expression consists of variables, constants, calls of functions with a return value, data members and methods of objects, operators and parentheses. The operators in the table are ordered from the highest priority; operators of the same priority are evaluated from left to right.

Operators	Meaning	Operand types
( )	parentheses, function call	
- ! ~ (type)	unary minus, logical negation, bitwise negation, cast	minus: numbers; !: bool or number; ~: integers
* / %	multiplication, division, remainder	% integers only (char, unsigned, int, unsigned64, int64)
+ -	addition and string concatenation, subtraction	numbers; + also string and char
<< >>	bit shift left and right	integers
< > <= >=	comparison	numbers, characters, strings
== !=	equality, inequality	all basic types, objects of the same type
&	bitwise AND	integers
^	bitwise exclusive OR (XOR)	integers
|	bitwise OR	integers
&& ||	logical AND and OR (the same priority)	bool or numbers

- The operators && and || have the same priority and are evaluated from left to right: a || b && c means (a || b) && c, not a || (b && c) as in C. Always use parentheses when combining them; the compiler warns about a mixed form without parentheses.
- The operators && and || are evaluated with short circuit: if the result is known from the left operand (false && ..., true || ...), the right operand is not evaluated and a function in it is not called.
- Integer division truncates toward zero (-7 / 2 = -3), the remainder has the sign of the dividend (-7 % 2 = -1). Integer division by zero prints the message Division by zero and the result is 0.
- Integer types wrap around on overflow (two's complement), e.g. M_INT32_MAX + 1 = -2147483648, unsigned 0 - 1 = 4294967295.
- The calculation is done in the type of the operands: char + char is calculated in 8 bits, unsigned * unsigned in 32 bits. For a larger range cast one operand, e.g. (int64)a * b.
- For the operators ~, << and >> a char is extended to int, so c << 8 does not lose the upper bits. The right shift is arithmetic for signed types (-16 >> 2 = -4) and logical for unsigned types.
- A logical value (bool) is not a number: it cannot be used with + - * /; convert it to int. On the other hand a number can be used as a condition in if, while and for and as an operand of the logical operators ! && ||: it is true if it is not zero.
- Strings are compared alphabetically (by character codes) and case-sensitively.
- The comparison == and != can also be used for two objects of the same type (obj_can_msg, obj_lin, containers, obj_memory); see chapter Object comparison.

> *Examples: T02_arithmetic.prg, T06_bitwise_operators.prg, T07_comparison_logic.prg, T20_expressions_priority.prg*

Assignment

Form	Meaning
`x = expression;`	assignment

Form	Meaning
`x += expression; -= *= /= %=`	compound assignment: x = x + expression etc.
`x &= expression; |= ^= <<= >>=`	compound bitwise assignment
`x++; x--; ++x; --x;`	increment and decrement by 1
`obj += value; obj += obj2;`	adding elements to a container (chapter Adding to a container)

- Assignment is a statement, not an expression (a = b = 0; cannot be written).
- The type of the expression must be convertible to the variable without loss (chapter Implicit conversions), otherwise a cast is needed.
- In a compound assignment the result is converted back to the type of the variable, so char c; c += 1; works.
- Compound assignment also works for data members of objects (msg.size += 2;) and strings (s += "abc"; s += c;). Concatenation beyond 1023 characters truncates the result.
- x++ and x-- can be used as a statement and in the header of a for loop, ++x and --x only as a standalone statement.
- Assignment between objects (msg = vec;) copies or converts data; the supported combinations are in chapter Object assignment.

> *Examples: T05_compound_assignment.prg, T32_assign_data.prg*

Statements

Statements are executed one after another in the order they are written. Simple statements end with a semicolon, the bodies of if, else, while and for are always blocks in braces (chapter Blocks and braces).

Statement	Use
`if, else if, else`	conditional execution
`while`	loop with the condition at the beginning
`for`	loop with initialization, condition and step
`switch`	selection of a branch by a value
`break`	end of a loop or of the switch statement
`continue`	next pass of a loop
`return`	return from a user function
`{ ... }`	block of statements
`;`	empty statement

if, else if, else

```
if( temperature > 90 ) {
    prints("ALARM");
}
else if( temperature > 75 ) {
    prints("WARNING");
}
else {
    prints("OK");
}
```

The condition is an expression of type bool or a number; a number (integer or floating-point) is true if it is not zero, as in C: if(n) means if(n != 0). A string or an object cannot be a condition. There can be any number of else if branches, else is optional.

while

```
i = 1;
while( i <= 10 ) {
    sum += i;
    i++;
}
```

The condition (bool or a number, as for if) is tested before every pass; if it is not met at the beginning, the body is not executed at all. An endless loop is written as while(true) { ... } and is ended by break or return. The language has no do ... while loop; use while(true) with break at the end of the body.

for

```
for( i = 0, j = 10; i < j; i++, j-- ) {
    logi(i);
    log_endl();
}
```

The header has three parts separated by semicolons: initialization, condition and step. The first and the third part can contain only assignments (also compound, ++ and --) separated by commas; a variable declaration in the header (for(int i = 0; ...)) is not allowed. The condition is mandatory. The step is executed after every pass of the body, also after the continue statement.

break and continue

break ends the innermost loop (while, for) or switch statement and execution continues after it. continue jumps to the next pass of the innermost loop: in while to the condition test, in for to the step and the condition test.

```
for( i = 0; i < 100; i++ ) {
    if( data.at(i) == 0 ) { continue; }    // skip zero values
    if( data.at(i) < 0 ) { break; }        // end of data
    sum += data.at(i);
}
```

- break inside switch ends only the switch, not the loop around it; break in a loop inside a case branch ends only that loop.
- continue inside a switch that is in a loop continues with the next pass of the loop.
- break outside a loop and switch is the error Statement break can be used only in a loop or switch, continue outside a loop the error Statement continue can be used only in a loop.

> *Example: T23_break_continue.prg*

switch

The switch statement selects a branch by the value of an expression. The expression can be an integer or floating-point number, a character or a string; the values after case must be of the same kind.

```
switch( cmd ) {
    case "start", "run":
        state = 1;
        break;
    case "stop":
        state = 0;
        break;
    default:
        prints("unknown command");
        break;
}
```

- Several values separated by commas and ranges from..to (both limits included) can follow case: case 1..5, 10:. A range can be used for numbers, characters ('a'..'z') and strings (alphabetically). Values and limits can be const constants.
- Every case and default branch must end with break, return (in a function) or continue (in a switch inside a loop), or with a statement that itself always ends with one of them (an if with else in which both branches end with return). The language has no fall-through to the next branch; a missing break is the error Expected symbol "break". This also applies to the last branch.
- A label without statements shares the branch with the following label: case 1: case 2: ... means the same as case 1, 2:.
- A branch can be a block: case 1: { ... } break; or case 1: { ...; break; }. A break inside the block of a branch ends the switch.
- The default branch is executed when no case value matches. It can be anywhere (at the beginning, in the middle or at the end), at most once in a switch, and it is optional; without it nothing is executed when no value matches.
- Strings are compared as a whole and case-sensitively. Floating-point numbers are compared exactly; use a range for intervals.
- An empty switch(x) { } is allowed.

```
function string grade(int points)
    switch( points ) {
        case 90..100:  return "A";
        case 75..89:   return "B";
        case 50..74:   return "C";
        case 0..49:    return "F";
        default:       return "?";
    }
```

```
end

switch( key ) {
    case 'a'..'z', 'A'..'Z':
    {
        letters++;
        break;
    }
    case '0'..'9':
        digits++;
        break;
    default:
        break;
}
```

> *Examples: T08_control_flow.prg, T35_numeric_conditions.prg, T17_switch_details.prg, T22_switch_return_if_break.prg, T27_case_ranges.prg, T31_switch_flow.prg*

return

return expression; ends a user function and returns a value, return; ends a function without a return value. return can be used only in a user function (not in the init, body, shutdown or func_N sections); elsewhere it is the error Statement return can be used only in a function. A section ends at the end of its code or by the function stop().

Stopping the script

The function stop() ends the script (the shutdown section is executed). The interpreter also stops the script when the time limit of a section or the nesting depth of functions is exceeded (chapter Interpreter environment).

> *Example: T21_body_calls.prg*

Functions

User functions

```
function double average(double a, double b)
    return (a + b) / 2;
end

function divide(int x, int y, ref int quotient, ref int remainder)
    quotient = x / y;
    remainder = x % y;
end

function int next_id()
    static int counter = 1000;      // initialized only once
    counter++;
    return counter;
end
```

- Definition: function [type | void] name([ref] type parameter, ...), the body ends with the word end. Without a type or with void the function has no return value.
- A function can be anywhere in the file and can be called before its definition. A function with a value is called in an expression or as a standalone statement (the result is discarded), a function without a value as a standalone statement.
- Parameters are passed by value and converted to the parameter type as in an assignment. A ref parameter is an output parameter: after the return its value is copied back, so the argument must be a variable of the same type. Parameter names must not be keywords (e.g. type).
- Local variables are declared at the beginning of the body, before the first statement: [static] type name [= expression], ...;. Without an initializer they are 0 on every call; with an initializer the expression is evaluated on every call and can use the parameters.
- A static variable keeps its value between calls; its initializer is executed only once, at the beginning of the init section.
- Local names hide global variables of the same name.
- return expression; returns a value, return; ends a function without a value. A function with a return value should end with return on every path; otherwise the result is undefined and the compiler prints a warning.
- Functions can call other functions up to a depth of 10; recursion (also indirect) is not allowed and the interpreter stops the script when it occurs.
- Parameters are of basic types. Objects are global and a function uses them directly by name.

```
function string mode_name(int m)
    switch( m ) {
        case 0:  return "off";
        case 1:  return "eco";
        default: return "?";
    }
end

function int first_over(int limit)
    int k = 0;
    while( true ) {                 // left only by return
        k++;
        if( k * k > limit ) { return k; }
    }
end
```

> *Examples: T09_user_functions.prg, T19_edge_cases.prg, T25_initialization_const.prg, T29_discard_results.prg*

Sections func_0 to func_9

Simple functions without parameters and return value working with global variables. They are called by the statement func(N);, where N can also be a variable. Nesting is limited to 10 levels.

```
body:
    func(0);
end

func_0:
    counter++;
end
```

> *Example: T10_func_sections.prg*

Built-in functions

Every function is given with its declaration in C syntax: result type (void = no result), name and parameter types. A function with a result is used in an expression or as a standalone statement (the result is then discarded), a function without a result as a standalone statement.

Conversion functions

Conversions between types and between numbers and strings, working with bits and bytes. Conversions between numeric types can also be done by a cast (type)expression.

`string b2str(bool value)`

Returns the string true or false according to value.

`string c2str(char value)`

Converts a character to a string.

`string i2str(int value)`

Converts an integer to a string.

`string i2str_radix(int value, int radix)`

Converts an integer to a string in the given base 2 to 36 (digits above 9 are lower case letters). In base 10 a negative number is written with a sign, in other bases as a 32-bit two's complement: i2str_radix(-1, 16) = "ffffffff". For an invalid base it returns an empty string.

`string u2str(unsigned value)`

Converts an unsigned integer to a string.

`string f2str(float value)`

Converts a floating-point number to a string with six decimal places, e.g. 0.250000.

`string d2str(double value)`

Converts a floating-point number to a string with six decimal places, e.g. d2str(-0.5) = "-0.500000".

`string i642str(int64 value)`

Converts an int64 number to a string.

`string u642str(unsigned64 value)`

Converts an unsigned64 number to a string.

`string u2hex(unsigned value, int digits)`

Converts a number to hexadecimal notation in upper case, padded with zeros on the left to digits digits (0 to 8; 0 = no padding): u2hex(0x1A, 4) = "001A".

`string u642hex(unsigned64 value, int digits)`

As u2hex for a 64-bit number (0 to 16 digits).

`int str2i(string value)`

Converts an integer written in a string to its value. Leading spaces and a sign are accepted, reading stops at the first character that is not a digit (str2i("15kg") = 15); a text without a number gives 0. A value out of range is limited: str2i("3000000000") = 2147483647.

`unsigned str2u(string value)`

Converts an integer written in a string to an unsigned value. A negative number is converted bitwise (str2u("-1") = 4294967295), a value above the range gives 4294967295.

`char str2c(string value)`

Converts an integer written in a string to a char value (the lowest byte: str2c("300") = 44).

int64 str2i64(string value)

Converts an integer written in a string to an int64 value.

unsigned64 str2u64(string value)

Converts an integer written in a string to an unsigned64 value.

bool str2b(string value)

Converts a logical value written in a string. The first character decides: 1, y, Y, a, A, t, T means true, anything else false. Therefore str2b(b2str(x)) == x, and the forms yes, ano and true are accepted.

float str2f(string value)

Converts a floating-point number written in a string to its value.

double str2d(string value)

Converts a floating-point number written in a string (with a decimal dot, optionally an exponent: "1.5e3") to its value; a text without a number gives 0.

unsigned from_hex(string value)

Converts a hexadecimal number written in a string (e.g. "1A2B", the case does not matter, the prefix 0x is allowed) to its value. More than 8 digits give 4294967295, an invalid text 0.

int truncf(float value)

Returns the integer part of a floating-point number (truncation toward zero). A value out of the int range is limited to -2147483648 or 2147483647, NaN gives 0.

int truncd(double value)

See truncf.

int roundf(float value)

Rounds a floating-point number to the nearest integer; a half is rounded up (2.5 to 3, -2.5 to -2). Range limitation as for truncf.

int roundd(double value)

See roundf.

char i2c(int value)

Converts int to char (lowest byte).

char u2c(unsigned value)

Converts unsigned to char (lowest byte).

unsigned i2u(int value)

Converts int to unsigned.

char i642c(int64 value)

Converts int64 to char (lowest byte).

char u642c(unsigned64 value)

Converts unsigned64 to char (lowest byte).

int i642i(int64 value)

Converts int64 to int (lowest 32 bits).

unsigned u642u(unsigned64 value)

Converts unsigned64 to unsigned (lowest 32 bits).

char bcd2dec(char value)

Converts a BCD byte to its value (e.g. 0x25 to 25).

`char dec2bcd(char value)`

Converts a value 0 to 99 to a BCD byte.

`char get_bfu(unsigned value, int byte_index)`

Returns the given byte (index 0 to 3, 0 = lowest) of an unsigned word; for another index it returns 0.

`char get_bfi(int value, int byte_index)`

Returns the given byte (index 0 to 3) of an int word; for another index it returns 0.

`char bitsetc(char value, int index)`

Returns the value with bit index set to 1. Variants: bitseti (int), bitsetu (unsigned), bitseti64 (int64), bitsetu64 (unsigned64).

`int bitseti(int value, int index)`

See bitsetc.

`unsigned bitsetu(unsigned value, int index)`

See bitsetc.

`int64 bitseti64(int64 value, int index)`

See bitsetc.

`unsigned64 bitsetu64(unsigned64 value, int index)`

See bitsetc.

`char bitclrc(char value, int index)`

Returns the value with bit index cleared to 0. Variants: bitclri, bitclru, bitclri64, bitclru64.

`int bitclri(int value, int index)`

See bitclrc.

`unsigned bitclru(unsigned value, int index)`

See bitclrc.

`int64 bitclri64(int64 value, int index)`

See bitclrc.

`unsigned64 bitclru64(unsigned64 value, int index)`

See bitclrc.

`bool bitestc(char value, int index)`

Returns true if bit index is 1. Variants: bitesti, bitestu, bitesti64, bitestu64.

`bool bitesti(int value, int index)`

See bitestc.

`bool bitestu(unsigned value, int index)`

See bitestc.

`bool bitesti64(int64 value, int index)`

See bitestc.

`bool bitestu64(unsigned64 value, int index)`

See bitestc.

> *Examples: T12_string_conversions.prg, T14_bit_functions_bcd.prg, T18_more_functions.prg, T26_new_functions.prg, T34_number_text.prg*

String functions

Character positions in a string are numbered from 0. Functions that change a string (str_insert, str_erase, str_replace, str_remove_...) change the variable passed as the first parameter directly.

int str_length(string str)

Returns the length of the text.

int str_find(string str, string substr)

Returns the position of substr in str, or -1 if not found.

int str_find_from(string str, int startpos, string substr)

Like str_find, the search starts at startpos.

int str_find_first_of(string str, int startpos, string charset)

Returns the position of the first occurrence of any character of charset from startpos, otherwise -1.

int str_find_first_not_of(string str, int startpos, string charset)

Returns the position of the first character from startpos that is not in charset, otherwise -1.

int str_replace(string str, int startpos, int count, string source)

Replaces count characters of str starting at startpos by source. Returns 1 if the replacement was done, or 0 if startpos is outside the string.

char str_get(string str, int pos)

Returns the character at pos, or 0 if the position is outside the string.

void str_insert(string str, int pos, char character)

Overwrites the character at pos of str; the position must be inside the string.

void str_erase(string str, int startpos, int length)

Erases length characters from startpos.

int str_row(string text, string row, int pos)

Copies the line of text starting at pos (up to the end-of-line character) to row. Returns the position of the next line, or -1 if there is none. Suitable for walking through a multi-line text.

int str_item(string text, string item, int pos, char separator)

Like str_row, items are separated by separator (e.g. a comma).

After the last line (item) str_row and str_item return the length of the text; only the next call returns -1 and leaves row (item) unchanged. Walk the text with a loop:

```
pos = 0;
while( pos >= 0 && pos < str_length(text) ) {
    pos = str_item(text, item, pos, ';');
    ...
}
```

void str_remove_nonprintable(string str)

Removes characters outside the ASCII range 0 to 127.

void str_remove_diacritics(string str)

Replaces characters with diacritics (Windows-1250 encoding) by characters without them.

string str_sub(string str, int pos, int length)

Returns the substring from position pos with length characters; length < 0 means to the end. A position and length outside the string are limited, the result is at most an empty string.

void str_upper(string str)

Converts the letters of the string to upper case (ASCII characters only).

void str_lower(string str)

Converts the letters of the string to lower case (ASCII characters only).

void str_trim(string str)

Removes spaces, tabs and line ends at the beginning and at the end of the string.

```
s = "  ECU_12;OK  ";
str_trim(s);                          // "ECU_12;OK"
name = str_sub(s, 0, str_find(s, ";"));   // "ECU_12"
str_lower(name);                      // "ecu_12"
```

> *Examples: T11_strings.prg, T12_string_conversions.prg, T26_new_functions.prg*

Output

The print functions write a value to the OUTPUT window immediately. The log functions write to an internal text buffer (max. 512 characters) whose content is written to the OUTPUT window by log_endl().

void prints(string value)

Prints a string.

void printb(bool value)

Prints true or false.

void printc(char value, bool type)

Prints the value as a character (if type is true) or as a number 0 to 255.

void printi(int value)

Prints an int value.

void printu(unsigned value)

Prints an unsigned value.

void printi64(int64 value)

Prints an int64 value.

void printu64(unsigned64 value)

Prints an unsigned64 value.

void printf(float value)

Prints a floating-point number.

void printd(double value)

Prints a floating-point number.

void print_endl()

Prints an empty line.

void logs(string value)

Appends a string to the buffer.

void logb(bool value)

Appends true or false to the buffer.

void logc(char value, bool type)

Appends the value as a character (if type is true) or as a number 0 to 255.

void logc_hex(char value)

Appends a byte in hexadecimal, always two digits (00 to FF).

`void logc_bin(char value)`

Appends a byte in binary as the character b and 8 bits, e.g. 15 as b00001111.

`void logi(int value)`

Appends an int value.

`void logu(unsigned value)`

Appends an unsigned value.

`void logu_hex(unsigned value)`

Appends an unsigned value in hexadecimal.

`void logu_bin(unsigned value)`

Appends an unsigned value in binary (b and 32 bits).

`void logi64(int64 value)`

Appends an int64 value.

`void logu64(unsigned64 value)`

Appends an unsigned64 value.

`void logu64_hex(unsigned64 value)`

Appends an unsigned64 value in hexadecimal.

`void logu64_bin(unsigned64 value)`

Appends an unsigned64 value in binary (b and 64 bits).

`void logf(float value)`

Appends a floating-point number.

`void logd(double value)`

Appends a floating-point number.

`void log_time()`

Appends the current local time as <hh:mm:ss.mmm>, e.g. <09:05:03.007>. Suitable at the beginning of an event record.

`void log_endl()`

Writes the buffer to the OUTPUT window and clears it.

`void log_endl_color(char color)`

Like log_endl, the text is shown in colour 0 black, 1 red (COLOR_RED), 2 green (COLOR_GREEN), 3 blue (COLOR_BLUE).

`void log_clr()`

Clears the buffer.

`int log_length()`

Returns the current length of the text in the buffer.

`int log_max()`

Returns the maximum length of the text in the buffer.

`void clear_output()`

Clears the OUTPUT window.

> *Examples: T18_more_functions.prg*

Arithmetic and mathematical operations

Trigonometric functions work in radians. The bit functions (shl, shr, and, or, xor) give the same result as the operators << >> & | ^.

```
char shlc(char value, int count)
```

Returns the value shifted left by count bits. Variants: shli (int), shlu (unsigned), shlu64.

```
int shli(int value, int count)
```

See shlc.

```
unsigned shlu(unsigned value, int count)
```

See shlc.

```
unsigned64 shlu64(unsigned64 value, int count)
```

See shlc.

```
char shrc(char value, int count)
```

Returns the value shifted right by count bits. Variants: shri, shru, shru64.

```
int shri(int value, int count)
```

See shrc.

```
unsigned shru(unsigned value, int count)
```

See shrc.

```
unsigned64 shru64(unsigned64 value, int count)
```

See shrc.

```
char andc(char a, char b)
```

Returns the bitwise AND. Variants: andi, andu, andu64.

```
int andi(int a, int b)
```

See andc.

```
unsigned andu(unsigned a, unsigned b)
```

See andc.

```
unsigned64 andu64(unsigned64 a, unsigned64 b)
```

See andc.

```
char orc(char a, char b)
```

Returns the bitwise OR. Variants: ori, oru, oru64.

```
int ori(int a, int b)
```

See orc.

```
unsigned oru(unsigned a, unsigned b)
```

See orc.

```
unsigned64 oru64(unsigned64 a, unsigned64 b)
```

See orc.

```
char xorc(char a, char b)
```

Returns the bitwise XOR. Variants: xori, xoru, xoru64.

```
int xori(int a, int b)
```

See xorc.

unsigned xoru(unsigned a, unsigned b)

See xorc.

unsigned64 xoru64(unsigned64 a, unsigned64 b)

See xorc.

double sin(double a)

Sine, parameter in radians.

double cos(double a)

Cosine, parameter in radians.

double tan(double a)

Tangent, parameter in radians.

double sinh(double a)

Hyperbolic sine.

double cosh(double a)

Hyperbolic cosine.

double tanh(double a)

Hyperbolic tangent.

double asin(double a)

Arc sine (result in radians).

double acos(double a)

Arc cosine (result in radians).

double atan(double a)

Arc tangent (result in radians).

double log(double a)

Natural logarithm.

double log10(double a)

Decimal logarithm.

double exp(double a)

Exponential function e^a.

double sqrt(double a)

Square root.

double pow(double a, double b)

a raised to the power of b.

double poly(double x, double a, double b, double c)

Value of the polynomial a*x*x + b*x + c.

double linear_interpolation(double x0, double y0, double x1, double y1, double x)

Linear interpolation of the value at x between the points [x0, y0] and [x1, y1].

double deg2rad(double a)

Converts degrees to radians.

double rad2deg(double a)

Converts radians to degrees.

```
char minc(char x1, char x2)
```

Returns the smaller of two values. Variants: mini, minu, mini64, minu64, minf, mind.

```
int mini(int x1, int x2)
```

See minc.

```
unsigned minu(unsigned x1, unsigned x2)
```

See minc.

```
int64 mini64(int64 x1, int64 x2)
```

See minc.

```
unsigned64 minu64(unsigned64 x1, unsigned64 x2)
```

See minc.

```
float minf(float x1, float x2)
```

See minc.

```
double mind(double x1, double x2)
```

See minc.

```
char maxc(char x1, char x2)
```

Returns the bigger of two values. Variants: maxi, maxu, maxi64, maxu64, maxf, maxd.

```
int maxi(int x1, int x2)
```

See maxc.

```
unsigned maxu(unsigned x1, unsigned x2)
```

See maxc.

```
int64 maxi64(int64 x1, int64 x2)
```

See maxc.

```
unsigned64 maxu64(unsigned64 x1, unsigned64 x2)
```

See maxc.

```
float maxf(float x1, float x2)
```

See maxc.

```
double maxd(double x1, double x2)
```

See maxc.

```
int minusi(int value)
```

Returns the value with the opposite sign.

```
double minusd(double value)
```

Returns the value with the opposite sign.

```
int random(int min, int max)
```

Returns a random number in the range min to max-1 (uniformly in the whole int range). If max is less than or equal to min, it returns min.

```
int absi(int value)
```

Absolute value of an integer (absi(-2147483648) stays -2147483648).

```
double absd(double value)
```

Absolute value of a floating-point number.

```
double floor(double a)
```

Rounding down to an integer (floor(-1.5) = -2.0).

`double ceil(double a)`

Rounding up to an integer (ceil(-1.5) = -1.0).

`double fmod(double a, double b)`

Remainder of a / b with the sign of a: fmod(7.5, 2) = 1.5.

`double atan2(double y, double x)`

Angle of the point [x, y] from the positive x axis in radians (-π to π); unlike atan(y / x) it distinguishes all quadrants and works also for x = 0.

`int clampi(int value, int min, int max)`

Limits the value to the interval <min, max>.

`double clampd(double value, double min, double max)`

Limits a floating-point value to the interval <min, max>.

> *Examples: T13_math_functions.prg, T14_bit_functions_bcd.prg, T26_new_functions.prg*

CAN interface

`unsigned x2can_tec()`

Returns the TEC (Transmit Error Counter) register of the CAN controller. Reading must be enabled in the PP2CAN Options.

`unsigned x2can_rec()`

Returns the REC (Receive Error Counter) register of the CAN controller. Reading must be enabled in the PP2CAN Options.

`unsigned x2can_txbl()`

Returns the number of CAN messages waiting for transmission.

`unsigned x2can_rxbl()`

Returns the number of received CAN messages waiting for processing.

`void x2can_reset(int value)`

Sets the CAN bit rate: 0 10k, 1 20k, 2 33.3k, 3 50k, 4 62.5k, 5 83.3k, 6 100k, 7 125k, 8 250k, 9 500k, 10 1M.

`bool lin_config(int port, int mode, unsigned speed, int version)`

Configures LIN port 1 or 2 of the USB2LIN converter: mode LIN_MASTER, LIN_SLAVE or LIN_SCAN, speed in Bd (e.g. 19200) and version LIN_VER_13 or LIN_VER_20. Returns true on success; with a wrong parameter or another converter it prints a message and returns false.

Date and time

The functions return the local time, the _utc variants the Coordinated Universal Time (UTC). Use the function tick_ms() or the obj_timepoint object to measure time intervals.

`int time_h()`

Current hour. Variant time_utc_h.

`int time_m()`

Current minute. Variant time_utc_m.

`int time_s()`

Current second. Variant time_utc_s.

`int time_ms()`

Current milliseconds. Variant time_utc_ms.

int time_utc_h()

See time_h.

int time_utc_m()

See time_m.

int time_utc_s()

See time_s.

int time_utc_ms()

See time_ms.

int date_y()

Current year. Variant date_utc_y.

int date_m()

Current month. Variant date_utc_m.

int date_d()

Current day. Variant date_utc_d.

int date_utc_y()

See date_y.

int date_utc_m()

See date_m.

int date_utc_d()

See date_d.

string time_str()

Current local time as the text "hh:mm:ss.mmm".

string date_str()

Current local date as the text "yyyy-mm-dd".

unsigned64 tick_ms()

Monotonic time in milliseconds from an unspecified origin. It does not change when the system time is changed, so it is suitable for measuring intervals: t0 = tick_ms(); ... dt = tick_ms() - t0;

unsigned64 unix_time()

Number of seconds since 1 January 1970 00:00 UTC.

> *Examples: T16_time_timepoint.prg, T26_new_functions.prg*

Display and control of variables

The script window can display 4 double values and 4 bool values, control a progress bar (0 to 10000), read two sliders (0 to 10000), 4 buttons and a group of 4 radio buttons. The functions have the prefix ui (user interaction). A joystick and the key state can be read as well.

bool ui_get_b0()

Returns the state of button 0. Variants ui_get_b1, ui_get_b2, ui_get_b3.

bool ui_get_b1()

See ui_get_b0.

bool ui_get_b2()

See ui_get_b0.

bool ui_get_b3()

See ui_get_b0.

void ui_set_b0(bool state)

Sets the state of button 0 (e.g. for signalling). Variants ui_set_b1 to ui_set_b3.

void ui_set_b1(bool state)

See ui_set_b0.

void ui_set_b2(bool state)

See ui_set_b0.

void ui_set_b3(bool state)

See ui_set_b0.

int ui_get_r()

Returns the selected radio button (0 to 3).

void ui_set_r(int state)

Selects a radio button (0 to 3).

void ui_set_name_r(int radio_button, string name)

Sets the label of the given radio button (0 to 3); another index is ignored.

void ui_set_p(unsigned state)

Sets the progress bar (0 to 10000).

void ui_set_d0(double state)

Displays a value in field 0. Variants ui_set_d1 to ui_set_d3.

void ui_set_d1(double state)

See ui_set_d0.

void ui_set_d2(double state)

See ui_set_d0.

void ui_set_d3(double state)

See ui_set_d0.

unsigned ui_get_val0()

Returns the value of slider 0 (0 to 10000).

unsigned ui_get_val1()

Returns the value of slider 1 (0 to 10000).

void ui_set_val0(unsigned value)

Sets the value of slider 0.

void ui_set_val1(unsigned value)

Sets the value of slider 1.

void ui_set_timer_script_period(unsigned period)

Sets the period of timer calls of the script in ms. Call it in the init section; the control is updated automatically.

unsigned ui_get_timer_script_period()

Returns the timer call period of the script.

int ui_joy_x(int joystick_id)

Returns the X axis position of a joystick (0 first, 1 second; constants JOYSTICK1, JOYSTICK2). Variants for the axes y, z, r, u, v. The range depends on the joystick.

int ui_joy_y(int joystick_id)

See ui_joy_x.

int ui_joy_z(int joystick_id)

See ui_joy_x.

int ui_joy_r(int joystick_id)

See ui_joy_x.

int ui_joy_u(int joystick_id)

See ui_joy_x.

int ui_joy_v(int joystick_id)

See ui_joy_x.

bool ui_joy_b(int joystick_id, int button_number)

Returns the state of the given joystick button.

int ui_joy_get_num()

Returns the number of joysticks supported by the system.

bool ui_key_state(int key)

Returns true if the key given by a virtual key code (constants VK_...) is pressed.

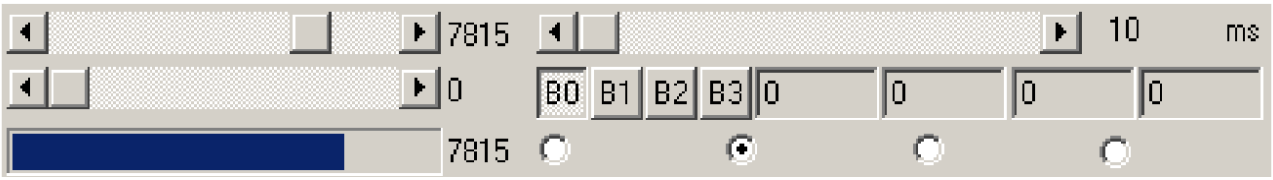


Fig. 3: Controls of the script window

> *Example: D01_ui_controls.prg*

Global variables

PP2CAN can run 4 scripts at the same time. They can cooperate through shared arrays of 128 elements of the types bool, int, unsigned, double and string (index 0 to 127).

bool global_bool_get(unsigned index)

Reads a global bool variable.

int global_int_get(unsigned index)

Reads a global int variable.

unsigned global_unsigned_get(unsigned index)

Reads a global unsigned variable.

double global_double_get(unsigned index)

Reads a global double variable.

string global_string_get(unsigned index)

Reads a global string variable.

void global_bool_set(unsigned index, bool value)

Writes a global bool variable.

void global_int_set(unsigned index, int value)

Writes a global int variable.

void global_unsigned_set(unsigned index, unsigned value)

Writes a global unsigned variable.

void global_double_set(unsigned index, double value)

Writes a global double variable.

void global_string_set(unsigned index, string value)

Writes a global string variable.

> *Examples: T15_global_variables.prg, P01a_global_producer.prg, P01b_global_consumer.prg*

Other functions

void stop()

Stops the script.

void func(int index)

Runs the section func_0 to func_9 with the given number.

void wait(unsigned value)

Pauses the script for value milliseconds. Use it only when necessary; with a lot of data it may cause congestion.

unsigned get_run_counter()

Returns the number of executions of the body section.

void msg_box(string value)

Shows a MessageBox with the given text.

void msg_beep()

Short sound signal.

string file_dialog_load(string fileext)

Opens a file selection dialog for loading with the extension fileext and returns the path (a space if cancelled).

string file_dialog_save(string fileext)

Opens a file selection dialog for saving and returns the path.

void command(string cmd, string output)

Runs the operating system command cmd and stores its text output in output.

string work_dir()

Working directory of PP2CAN without the trailing separator (e.g. "e:\pp2can"). Files given by a relative path (obj_csvfile, obj_logfile, obj_cfgfile ...) are looked up there. Requires the interpreter 3.02.

string script_dir()

Directory of the compiled script (.prg) without the trailing separator. The value is determined by the compiler at compile time (it is a constant and needs no new interpreter); useful for files stored next to the script: csv.read(script_dir() + "/data.csv"); Nothing may be written between the parentheses.

bool is_can_function_call()

Returns true if the script was started by a received CAN message (CAN_FUNCTION_AND_TIMER).

bool is_timer_call()

Returns true if the script was started by the timer.

bool is_lin_function_call()

Returns true if the script was started by a received LIN message (LIN_FUNCTION_AND_TIMER).

> *Examples: T21_body_calls.prg*

Built-in constants

Constants can be used in a script instead of values. The names of built-in constants cannot be used for variables. User constants are defined in the variables section with the keyword const (chapter Named constants).

Characters and strings

Name	Value
ENDL	(unsigned char)'\n'
TAB	(unsigned char)'\t'
ENDLS	"\n"
TABS	"\t"

Colours for log_endl_color

Name	Value
COLOR_RED	(unsigned char)0x01
COLOR_GREEN	(unsigned char)0x02
COLOR_BLUE	(unsigned char)0x03

Joystick indexes

Name	Value
JOYSTICK1	(int)0
JOYSTICK2	(int)1

Mathematical constants

Name	Value
M_PI	(double)3.14159265358979323846
M_PI_2	(double)1.57079632679489661923
M_PI_4	(double)0.78539816339744830962
M_E	(double)2.71828182845904523536
M_SQRT2	(double)1.41421356237309504880
M_SQRT1_2	(double)0.70710678118654524401

Type ranges

Name	Value
M_INT16_MIN	(int)-32768
M_INT16_MAX	(int)32767
M_INT32_MIN	(int)-2147483648
M_INT32_MAX	(int)2147483647
M_INT64_MIN	LLONG_MIN
M_INT64_MAX	LLONG_MAX
M_UINT16_MIN	(unsigned int)0
M_UINT16_MAX	(unsigned int)65535

Name	Value
M_UINT32_MIN	(unsigned int)0
M_UINT32_MAX	(unsigned int)4294967295
M_UINT64_MIN	0
M_UINT64_MAX	ULLONG_MAX
M_FLOAT_MIN	(float)1.175494351e-38
M_FLOAT_MAX	(float)3.402823466e+38
M_DOUBLE_MIN	(double)2.2250738585072014e-308
M_DOUBLE_MAX	(double)1.7976931348623158e+308

LIN

Name	Value
LIN_MASTER	(int)0
LIN_SLAVE	(int)1
LIN_SCAN	(int)2
LIN_VER_13	(int)13
LIN_VER_20	(int)20
LIN_PERIOD_OFF	(unsigned int)0
LIN_ONE_SHOT	(unsigned int)0xFFFF

obj_uds addressing

Name	Value
ISOTP_NORMAL	(int)0
ISOTP_EXTENDED	(int)1
ISOTP_MIXED	(int)2

Error codes of obj_uds and obj_j1939

Name	Value
TP_OK	(int)0
TP_ERR_SEQUENCE	(int)1
TP_ERR_UNEXPECTED	(int)2
TP_ERR_SIZE	(int)3
TP_ERR_ABORT	(int)4
TP_ERR_CONFIG	(int)5
TP_ERR_FRAME	(int)6

Calling conventions for obj_dllfnc

Name	Value
ABI_STDCALL	(unsigned int)0
ABI_THISCALL	(unsigned int)1
ABI_FASTCALL	(unsigned int)2
ABI_MS_CDECL	(unsigned int)3

Name	Value
ABI_PASCAL	(unsigned int)4
ABI_REGISTER	(unsigned int)5

Parameter types for obj_dllfnc

Name	Value
PARAM_VOID	(unsigned int)0
PARAM_UINT8	(unsigned int)1
PARAM_SINT8	(unsigned int)2
PARAM_UINT16	(unsigned int)3
PARAM_SINT16	(unsigned int)4
PARAM_UINT32	(unsigned int)5
PARAM_SINT32	(unsigned int)6
PARAM_UINT64	(unsigned int)7
PARAM_SINT64	(unsigned int)8
PARAM_FLOAT	(unsigned int)9
PARAM_DOUBLE	(unsigned int)10
PARAM_POINTER	(unsigned int)11

Virtual key codes for ui_key_state

Name	Value
VK_LBUTTON	(int)0x01
VK_RBUTTON	(int)0x02
VK_CANCEL	(int)0x03
VK_MBUTTON	(int)0x04
VK_BACK	(int)0x08
VK_TAB	(int)0x09
VK_CLEAR	(int)0x0C
VK_RETURN	(int)0x0D
VK_SHIFT	(int)0x10
VK_CONTROL	(int)0x11
VK_MENU	(int)0x12
VK_PAUSE	(int)0x13
VK_CAPITAL	(int)0x14
VK_ESCAPE	(int)0x1B
VK_CONVERT	(int)0x1C
VK_NONCONVERT	(int)0x1D
VK_ACCEPT	(int)0x1E
VK_MODECHANGE	(int)0x1F
VK_SPACE	(int)0x20
VK_PRIOR	(int)0x21
VK_NEXT	(int)0x22

Name	Value
VK_END	(int)0x23
VK_HOME	(int)0x24
VK_LEFT	(int)0x25
VK_UP	(int)0x26
VK_RIGHT	(int)0x27
VK_DOWN	(int)0x28
VK_SELECT	(int)0x29
VK_PRINT	(int)0x2A
VK_EXECUTE	(int)0x2B
VK_SNAPSHOT	(int)0x2C
VK_INSERT	(int)0x2D
VK_DELETE	(int)0x2E
VK_HELP	(int)0x2F
VK_0	(int)0x30
VK_1	(int)0x31
VK_2	(int)0x32
VK_3	(int)0x33
VK_4	(int)0x34
VK_5	(int)0x35
VK_6	(int)0x36
VK_7	(int)0x37
VK_8	(int)0x38
VK_9	(int)0x39
VK_A	(int)0x41
VK_B	(int)0x42
VK_C	(int)0x43
VK_D	(int)0x44
VK_E	(int)0x45
VK_F	(int)0x46
VK_G	(int)0x47
VK_H	(int)0x48
VK_I	(int)0x49
VK_J	(int)0x4A
VK_K	(int)0x4B
VK_L	(int)0x4C
VK_M	(int)0x4D
VK_N	(int)0x4E
VK_O	(int)0x4F
VK_P	(int)0x50
VK_Q	(int)0x51
VK_R	(int)0x52

Name	Value
VK_S	(int)0x53
VK_T	(int)0x54
VK_U	(int)0x55
VK_V	(int)0x56
VK_W	(int)0x57
VK_X	(int)0x58
VK_Y	(int)0x59
VK_Z	(int)0x5A
VK_LWIN	(int)0x5B
VK_RWIN	(int)0x5C
VK_APPS	(int)0x5D
VK_SLEEP	(int)0x5F
VK_NUMPAD0	(int)0x60
VK_NUMPAD1	(int)0x61
VK_NUMPAD2	(int)0x62
VK_NUMPAD3	(int)0x63
VK_NUMPAD4	(int)0x64
VK_NUMPAD5	(int)0x65
VK_NUMPAD6	(int)0x66
VK_NUMPAD7	(int)0x67
VK_NUMPAD8	(int)0x68
VK_NUMPAD9	(int)0x69
VK_MULTIPLY	(int)0x6A
VK_ADD	(int)0x6B
VK_SEPARATOR	(int)0x6C
VK_SUBTRACT	(int)0x6D
VK_DECIMAL	(int)0x6E
VK_DIVIDE	(int)0x6F
VK_F1	(int)0x70
VK_F2	(int)0x71
VK_F3	(int)0x72
VK_F4	(int)0x73
VK_F5	(int)0x74
VK_F6	(int)0x75
VK_F7	(int)0x76
VK_F8	(int)0x77
VK_F9	(int)0x78
VK_F10	(int)0x79
VK_F11	(int)0x7A
VK_F12	(int)0x7B
VK_F13	(int)0x7C

Name	Value
VK_F14	(int)0x7D
VK_F15	(int)0x7E
VK_F16	(int)0x7F
VK_F17	(int)0x80
VK_F18	(int)0x81
VK_F19	(int)0x82
VK_F20	(int)0x83
VK_F21	(int)0x84
VK_F22	(int)0x85
VK_F23	(int)0x86
VK_F24	(int)0x87
VK_NUMLOCK	(int)0x90
VK_SCROLL	(int)0x91
VK_LSHIFT	(int)0xA0
VK_RSHIFT	(int)0xA1
VK_LCONTROL	(int)0xA2
VK_RCONTROL	(int)0xA3
VK_LMENU	(int)0xA4
VK_RMENU	(int)0xA5
VK_BROWSER_BACK	(int)0xA6
VK_BROWSER_FORWARD	(int)0xA7
VK_BROWSER_REFRESH	(int)0xA8
VK_BROWSER_STOP	(int)0xA9
VK_BROWSER_SEARCH	(int)0xAA
VK_BROWSER_FAVORITES	(int)0xAB
VK_BROWSER_HOME	(int)0xAC
VK_VOLUME_MUTE	(int)0xAD
VK_VOLUME_DOWN	(int)0xAE
VK_VOLUME_UP	(int)0xAF
VK_MEDIA_NEXT_TRACK	(int)0xB0
VK_MEDIA_PREV_TRACK	(int)0xB1
VK_MEDIA_STOP	(int)0xB2
VK_MEDIA_PLAY_PAUSE	(int)0xB3
VK_LAUNCH_MAIL	(int)0xB4
VK_LAUNCH_MEDIA_SELECT	(int)0xB5
VK_LAUNCH_APP1	(int)0xB6
VK_LAUNCH_APP2	(int)0xB7

Objects

Objects group data and operations on them (methods). Data members and methods are accessed with a dot.

```
objects:
    obj_can_msg msg;
end
init:
    msg.id1 = 0x100;          // data member
    msg.size = 8;
    msg.send();               // method without a return value
    n = msg.get_uint16(0);    // method with a return value
end
```

The next_frame and receive methods of obj_uds and obj_j1939 take an obj_can_msg object as a parameter.

Object	Purpose
obj_can_msg	CAN and CAN FD message: building, sending, receiving, signals
obj_lin	LIN message and LIN table item (USB2LIN)
obj_uds, obj_j1939	transfer of long data: ISO-TP (UDS), J1939-21/22
obj_vector_int, obj_vector_double	array of numbers with algorithms (sorting, statistics, CRC, Base64)
obj_deque_int, obj_deque_double	double-ended queue (FIFO, stack)
obj_memory	memory block (e.g. for DLL functions)
obj_timepoint	measuring time intervals
obj_rs232	serial port
obj_socket_server, obj_socket_client	TCP and UDP communication
obj_csvfile, obj_cfgfile, obj_logfile	CSV files, configuration (ini) files and text logs
obj_datagrid, obj_dataset, obj_datagraph, obj_datagauge, obj_datahistogram	windows for displaying and entering values, graph, gauge, histogram
obj_pid	PID controller
obj_dllfnc	calling a function of a DLL

Object assignment

The assignment a = b; between objects copies or converts data. The compiler knows all supported combinations of types; another combination (e.g. msg = 5; or pid = vec;) is reported as the error Unsupported assignment: "obj_pid" cannot be assigned from "obj_vector_int". The assignment in the header of a for loop is checked in the same way. Since version 3.02 a string can be assigned to objects and objects to a string.

The tables list all supported combinations. Combinations with double containers (except assignments between containers, graph, table, CSV and log), with a string, a queue to obj_can_msg and obj_datagrid and a queue from obj_csvfile and obj_dataset require the interpreter 3.02 (format V4); the compiler points this out with a note.

Vector or queue as the source

Source	Destination	Behaviour
obj_vector_int	obj_vector_int	The content of the destination vector is replaced by the elements of the source. The source vector does not change.
obj_vector_int	obj_vector_double	The content of the destination vector is replaced by the elements of the source. The source vector does not change.
obj_vector_int	obj_deque_int	The elements of the source are appended to the end of the destination queue. The source vector does not change.

Source	Destination	Behaviour
obj_vector_int	obj_deque_double	The elements of the source are appended to the end of the destination queue. The source vector does not change.
obj_vector_int	obj_can_msg	Data bytes 0 to n-1, where n is the number of elements (max. 64); size = n, over 8 bytes fd = true is set. ID, rtr and port do not change. The values are truncated to 8 bits. The source vector does not change.
obj_vector_int	obj_lin	Data bytes and size = the number of elements (max. 8). The identifier does not change. The values are truncated to 8 bits. The source vector does not change.
obj_vector_int	obj_memory	The block is allocated for the number of elements and the elements are copied as bytes. The values are truncated to 8 bits. The source vector does not change.
obj_vector_int	obj_uds	Data to send (replace the previous data). The values are truncated to 8 bits. The source vector does not change.
obj_vector_int	obj_j1939	Data to send (replace the previous data). The values are truncated to 8 bits. The source vector does not change.
obj_vector_int	obj_rs232	The elements are sent as bytes. If the port is not open, nothing is sent. The values are truncated to 8 bits. The source vector does not change.
obj_vector_int	obj_socket_server	The elements are sent as bytes to all connected clients (TCP); data longer than 1024 bytes are sent completely. A server that is not open or is open as UDP sends nothing. The values are truncated to 8 bits. The source vector does not change.
obj_vector_int	obj_socket_client	The elements are sent as bytes to the server (TCP and UDP); data longer than 1024 bytes are sent completely. Without a connection nothing is sent. The values are truncated to 8 bits. The source vector does not change.
obj_vector_int	obj_datagraph	The values are added to line 0 of the graph and the graph is redrawn. The graph draws integers. If the window is not open, nothing happens. The source vector does not change.
obj_vector_int	obj_datagrid	The first 10 elements to cells 0 to 9 of the table, the window is redrawn. If the window is not open, nothing happens. The source vector does not change.
obj_vector_int	obj_csvfile	The values are written to the active row of the table (the table grows as needed) and the active row moves to the next one. The source vector does not change.
obj_vector_int	obj_logfile	Writes one line, values separated by commas. If the file is not open (set_name), nothing is written. The source vector does not change.
obj_vector_int	string	The elements as characters (bytes 0 to 255) up to the first zero value, at most 1023 characters. The source vector does not change.
obj_vector_double	obj_vector_int	The content of the destination vector is replaced by the elements of the source. The fractional part is truncated. The source vector does not change.
obj_vector_double	obj_vector_double	The content of the destination vector is replaced by the elements of the source. The source vector does not change.
obj_vector_double	obj_deque_int	The elements of the source are appended to the end of the destination queue. The fractional part is truncated. The source vector does not change.
obj_vector_double	obj_deque_double	The elements of the source are appended to the end of the destination queue. The source vector does not change.

Source	Destination	Behaviour
obj_vector_double	obj_can_msg	Data bytes 0 to n-1, where n is the number of elements (max. 64); size = n, over 8 bytes fd = true is set. ID, rtr and port do not change. The values are truncated to integers and to 8 bits. The source vector does not change.
obj_vector_double	obj_lin	Data bytes and size = the number of elements (max. 8). The identifier does not change. The values are truncated to integers and to 8 bits. The source vector does not change.
obj_vector_double	obj_memory	The block is allocated for the number of elements and the elements are copied as bytes. The values are truncated to integers and to 8 bits. The source vector does not change.
obj_vector_double	obj_uds	Data to send (replace the previous data). The values are truncated to integers and to 8 bits. The source vector does not change.
obj_vector_double	obj_j1939	Data to send (replace the previous data). The values are truncated to integers and to 8 bits. The source vector does not change.
obj_vector_double	obj_rs232	The elements are sent as bytes. If the port is not open, nothing is sent. The values are truncated to integers and to 8 bits. The source vector does not change.
obj_vector_double	obj_socket_server	The elements are sent as bytes to all connected clients (TCP); data longer than 1024 bytes are sent completely. A server that is not open or is open as UDP sends nothing. The values are truncated to integers and to 8 bits. The source vector does not change.
obj_vector_double	obj_socket_client	The elements are sent as bytes to the server (TCP and UDP); data longer than 1024 bytes are sent completely. Without a connection nothing is sent. The values are truncated to integers and to 8 bits. The source vector does not change.
obj_vector_double	obj_datagraph	The values are added to line 0 of the graph and the graph is redrawn. The graph draws integers, the fractional part is truncated. If the window is not open, nothing happens. The source vector does not change.
obj_vector_double	obj_datagrid	The first 10 elements to cells 0 to 9 of the table (as decimal numbers), the window is redrawn. If the window is not open, nothing happens. The source vector does not change.
obj_vector_double	obj_csvfile	The values are written to the active row of the table (the table grows as needed; format %f) and the active row moves to the next one. The source vector does not change.
obj_vector_double	obj_logfile	Writes one line, values separated by commas (format %f). If the file is not open (set_name), nothing is written. The source vector does not change.
obj_vector_double	string	The elements as characters (bytes 0 to 255) up to the first zero value, at most 1023 characters. The values are first truncated to integers. The source vector does not change.
obj_deque_int	obj_vector_int	The content of the destination vector is replaced by the elements of the source. The source queue is emptied.
obj_deque_int	obj_vector_double	The content of the destination vector is replaced by the elements of the source. The source queue is emptied.
obj_deque_int	obj_deque_int	The elements of the source are appended to the end of the destination queue. The source queue is emptied.
obj_deque_int	obj_deque_double	The elements of the source are appended to the end of the destination queue. The source queue is emptied.

Source	Destination	Behaviour
obj_deque_int	obj_can_msg	As many elements as the size of the message (DLC, max. 64) are taken from the front of the queue and written to bytes 0, 1, ... If the queue is shorter, all of it is taken and the remaining data bytes of the message do not change. size, ID, rtr, fd and port do not change; with size = 0 or rtr = true nothing is taken. The values are truncated to 8 bits. The elements taken are removed from the queue, the rest stays.
obj_deque_int	obj_lin	As many elements as the size of the message (max. 8) are taken from the front of the queue. If the queue is shorter, all of it is taken and the remaining data bytes do not change. size and the identifier do not change. The values are truncated to 8 bits. The elements taken are removed from the queue, the rest stays.
obj_deque_int	obj_memory	The block is allocated for the number of elements and the elements are copied as bytes. The values are truncated to 8 bits. The source queue is emptied.
obj_deque_int	obj_uds	Data to send (replace the previous data). The values are truncated to 8 bits. The source queue is emptied.
obj_deque_int	obj_j1939	Data to send (replace the previous data). The values are truncated to 8 bits. The source queue is emptied.
obj_deque_int	obj_rs232	The elements are sent as bytes. If the port is not open, nothing is sent. The values are truncated to 8 bits. The source queue is emptied. If the port is closed, the queue stays unchanged.
obj_deque_int	obj_socket_server	The elements are sent as bytes to all connected clients (TCP); data longer than 1024 bytes are sent completely. A server that is not open or is open as UDP sends nothing. The values are truncated to 8 bits. The source queue is emptied. If the server is closed or UDP, the queue stays unchanged.
obj_deque_int	obj_socket_client	The elements are sent as bytes to the server (TCP and UDP); data longer than 1024 bytes are sent completely. Without a connection nothing is sent. The values are truncated to 8 bits. The source queue is emptied. If the client is not connected, the queue stays unchanged.
obj_deque_int	obj_datagraph	The values are added to line 0 of the graph and the graph is redrawn. The graph draws integers. If the window is not open, nothing happens. The source queue is emptied. If the window is closed, the queue stays unchanged.
obj_deque_int	obj_datagrid	At most 10 elements are taken from the front of the queue and written to cells 0 to 9 of the table, the window is redrawn. The elements taken are removed from the queue, the rest stays. If the window is closed, the queue stays unchanged.
obj_deque_int	obj_csvfile	The values are written to the active row of the table (the table grows as needed) and the active row moves to the next one. The source queue is emptied.
obj_deque_int	obj_logfile	Writes one line, values separated by commas. If the file is not open (set_name), nothing is written. The source queue is emptied. If the file is not open, the queue stays unchanged.
obj_deque_int	string	The elements as characters (bytes 0 to 255) up to the first zero value, at most 1023 characters. The characters taken and the terminating zero are removed from the queue, the rest stays for the next assignment.
obj_deque_double	obj_vector_int	The content of the destination vector is replaced by the elements of the source. The fractional part is truncated. The source queue is emptied.

Source	Destination	Behaviour
obj_deque_double	obj_vector_double	The content of the destination vector is replaced by the elements of the source. The source queue is emptied.
obj_deque_double	obj_deque_int	The elements of the source are appended to the end of the destination queue. The fractional part is truncated. The source queue is emptied.
obj_deque_double	obj_deque_double	The elements of the source are appended to the end of the destination queue. The source queue is emptied.
obj_deque_double	obj_can_msg	As many elements as the size of the message (DLC, max. 64) are taken from the front of the queue and written to bytes 0, 1, ... If the queue is shorter, all of it is taken and the remaining data bytes of the message do not change. size, ID, rtr, fd and port do not change; with size = 0 or rtr = true nothing is taken. The values are truncated to integers and to 8 bits. The elements taken are removed from the queue, the rest stays.
obj_deque_double	obj_lin	As many elements as the size of the message (max. 8) are taken from the front of the queue. If the queue is shorter, all of it is taken and the remaining data bytes do not change. size and the identifier do not change. The values are truncated to integers and to 8 bits. The elements taken are removed from the queue, the rest stays.
obj_deque_double	obj_memory	The block is allocated for the number of elements and the elements are copied as bytes. The values are truncated to integers and to 8 bits. The source queue is emptied.
obj_deque_double	obj_uds	Data to send (replace the previous data). The values are truncated to integers and to 8 bits. The source queue is emptied.
obj_deque_double	obj_j1939	Data to send (replace the previous data). The values are truncated to integers and to 8 bits. The source queue is emptied.
obj_deque_double	obj_rs232	The elements are sent as bytes. If the port is not open, nothing is sent. The values are truncated to integers and to 8 bits. The source queue is emptied. If the port is closed, the queue stays unchanged.
obj_deque_double	obj_socket_server	The elements are sent as bytes to all connected clients (TCP); data longer than 1024 bytes are sent completely. A server that is not open or is open as UDP sends nothing. The values are truncated to integers and to 8 bits. The source queue is emptied. If the server is closed or UDP, the queue stays unchanged.
obj_deque_double	obj_socket_client	The elements are sent as bytes to the server (TCP and UDP); data longer than 1024 bytes are sent completely. Without a connection nothing is sent. The values are truncated to integers and to 8 bits. The source queue is emptied. If the client is not connected, the queue stays unchanged.
obj_deque_double	obj_datagraph	The values are added to line 0 of the graph and the graph is redrawn. The graph draws integers, the fractional part is truncated. If the window is not open, nothing happens. The source queue is emptied. If the window is closed, the queue stays unchanged.
obj_deque_double	obj_datagrid	At most 10 elements are taken from the front of the queue and written to cells 0 to 9 of the table (as decimal numbers), the window is redrawn. The elements taken are removed from the queue, the rest stays. If the window is closed, the queue stays unchanged.
obj_deque_double	obj_csvfile	The values are written to the active row of the table (the table grows as needed; format %f) and the active row moves to the next one. The source queue is emptied.

Source	Destination	Behaviour
obj_deque_double	obj_logfile	Writes one line, values separated by commas (format %f). If the file is not open (set_name), nothing is written. The source queue is emptied. If the file is not open, the queue stays unchanged.
obj_deque_double	string	The elements as characters (bytes 0 to 255) up to the first zero value, at most 1023 characters. The values are first truncated to integers. The characters taken and the terminating zero are removed from the queue, the rest stays for the next assignment.

Vector or queue as the destination

The rows with another container as the source are in the previous table.

Source	Destination	Behaviour
obj_can_msg	obj_vector_int	The content of the vector is replaced by the data bytes of the message (size bytes). The message does not change.
obj_can_msg	obj_vector_double	The content of the vector is replaced by the data bytes of the message (size bytes). Values 0 to 255. The message does not change.
obj_can_msg	obj_deque_int	Appended to the end of the queue: the data bytes of the message (size bytes). The message does not change.
obj_can_msg	obj_deque_double	Appended to the end of the queue: the data bytes of the message (size bytes). Values 0 to 255. The message does not change.
obj_lin	obj_vector_int	The content of the vector is replaced by the data bytes of the message (size bytes). The message does not change.
obj_lin	obj_vector_double	The content of the vector is replaced by the data bytes of the message (size bytes). Values 0 to 255. The message does not change.
obj_lin	obj_deque_int	Appended to the end of the queue: the data bytes of the message (size bytes). The message does not change.
obj_lin	obj_deque_double	Appended to the end of the queue: the data bytes of the message (size bytes). Values 0 to 255. The message does not change.
obj_memory	obj_vector_int	The content of the vector is replaced by the bytes of the block. The block does not change.
obj_memory	obj_vector_double	The content of the vector is replaced by the bytes of the block. Values 0 to 255. The block does not change.
obj_memory	obj_deque_int	Appended to the end of the queue: the bytes of the block. The block does not change.
obj_memory	obj_deque_double	Appended to the end of the queue: the bytes of the block. Values 0 to 255. The block does not change.
obj_uds	obj_vector_int	The content of the vector is replaced by the received data. The received data stay in the object.
obj_uds	obj_vector_double	The content of the vector is replaced by the received data. Values 0 to 255. The received data stay in the object.
obj_uds	obj_deque_int	Appended to the end of the queue: the received data. The received data stay in the object.
obj_uds	obj_deque_double	Appended to the end of the queue: the received data. Values 0 to 255. The received data stay in the object.
obj_j1939	obj_vector_int	The content of the vector is replaced by the received data. The received data stay in the object.
obj_j1939	obj_vector_double	The content of the vector is replaced by the received data. Values 0 to 255. The received data stay in the object.

Source	Destination	Behaviour
obj_j1939	obj_deque_int	Appended to the end of the queue: the received data. The received data stay in the object.
obj_j1939	obj_deque_double	Appended to the end of the queue: the received data. Values 0 to 255. The received data stay in the object.
obj_rs232	obj_vector_int	The content of the vector is replaced by the received bytes; they are taken out of the receive buffer. No received data, a closed port or no connection: the vector has length 0.
obj_rs232	obj_vector_double	The content of the vector is replaced by the received bytes; they are taken out of the receive buffer. Values 0 to 255. No received data, a closed port or no connection: the vector has length 0.
obj_rs232	obj_deque_int	Appended to the end of the queue: the received bytes; they are taken out of the receive buffer. No received data, a closed port or no connection: the queue stays unchanged.
obj_rs232	obj_deque_double	Appended to the end of the queue: the received bytes; they are taken out of the receive buffer. Values 0 to 255. No received data, a closed port or no connection: the queue stays unchanged.
obj_socket_server	obj_vector_int	The content of the vector is replaced by the received bytes; they are taken out of the receive buffer. No received data, a closed port or no connection: the vector has length 0.
obj_socket_server	obj_vector_double	The content of the vector is replaced by the received bytes; they are taken out of the receive buffer. Values 0 to 255. No received data, a closed port or no connection: the vector has length 0.
obj_socket_server	obj_deque_int	Appended to the end of the queue: the received bytes; they are taken out of the receive buffer. No received data, a closed port or no connection: the queue stays unchanged.
obj_socket_server	obj_deque_double	Appended to the end of the queue: the received bytes; they are taken out of the receive buffer. Values 0 to 255. No received data, a closed port or no connection: the queue stays unchanged.
obj_socket_client	obj_vector_int	The content of the vector is replaced by the received bytes; they are taken out of the receive buffer. No received data, a closed port or no connection: the vector has length 0.
obj_socket_client	obj_vector_double	The content of the vector is replaced by the received bytes; they are taken out of the receive buffer. Values 0 to 255. No received data, a closed port or no connection: the vector has length 0.
obj_socket_client	obj_deque_int	Appended to the end of the queue: the received bytes; they are taken out of the receive buffer. No received data, a closed port or no connection: the queue stays unchanged.
obj_socket_client	obj_deque_double	Appended to the end of the queue: the received bytes; they are taken out of the receive buffer. Values 0 to 255. No received data, a closed port or no connection: the queue stays unchanged.
obj_csvfile	obj_vector_int	The content of the vector is replaced by the values of the active row of the table (always size_c() elements, missing cells = default value, also converts 0x1A and a decimal comma), the active row moves to the next one. After the last row: the vector has length 0.
obj_csvfile	obj_vector_double	The content of the vector is replaced by the values of the active row of the table (always size_c() elements, missing cells = default value, also converts 0x1A and a decimal comma), the active row moves to the next one. After the last row: the vector has length 0.
obj_csvfile	obj_deque_int	Appended to the end of the queue: the values of the active row of the table (always size_c() elements, missing cells = default value, also converts 0x1A and a decimal comma), the active row moves to the next one. After the last row: the queue stays unchanged.

Source	Destination	Behaviour
obj_csvfile	obj_deque_double	Appended to the end of the queue: the values of the active row of the table (always size_c() elements, missing cells = default value, also converts 0x1A and a decimal comma), the active row moves to the next one. After the last row: the queue stays unchanged.
obj_dataset	obj_vector_int	The content of the vector is replaced by the 10 values entered in the window. If the window is not open: the vector has length 0.
obj_dataset	obj_vector_double	The content of the vector is replaced by the 10 values entered in the window. If the window is not open: the vector has length 0.
obj_dataset	obj_deque_int	Appended to the end of the queue: the 10 values entered in the window. If the window is not open: the queue stays unchanged.
obj_dataset	obj_deque_double	Appended to the end of the queue: the 10 values entered in the window. If the window is not open: the queue stays unchanged.
string	obj_vector_int	The content of the vector is replaced by the characters of the string as bytes 0 to 255 (without the terminating zero). Empty string: the vector has length 0.
string	obj_vector_double	The content of the vector is replaced by the characters of the string as bytes 0 to 255 (without the terminating zero). Empty string: the vector has length 0.
string	obj_deque_int	Appended to the end of the queue: the characters of the string as bytes 0 to 255 (without the terminating zero). Empty string: the queue stays unchanged.
string	obj_deque_double	Appended to the end of the queue: the characters of the string as bytes 0 to 255 (without the terminating zero). Empty string: the queue stays unchanged.

String and objects

A string to a container and a container to a string are in the previous tables. The string is a variable of type string or a string constant (e.g. client = "Ping";).

Source	Destination	Behaviour
string	obj_can_msg	The characters as data bytes, size = the length of the string (max. 64), over 8 bytes fd = true is set. ID, rtr and port do not change.
string	obj_lin	The characters as data bytes, size = the length of the string (max. 8).
string	obj_memory	The block is allocated for the length of the string and the characters are copied (without the terminating zero).
string	obj_rs232	The characters are sent (without the terminating zero). If the port is not open, nothing is sent.
string	obj_socket_server	The characters are sent to all connected clients (TCP). A server that is not open or UDP sends nothing.
string	obj_socket_client	The characters are sent to the server (TCP and UDP). Without a connection nothing is sent.
obj_can_msg	string	The data bytes (size bytes) as characters up to the first zero byte, at most 1023 characters. The object does not change.
obj_lin	string	The data bytes (size bytes) as characters up to the first zero byte, at most 1023 characters. The object does not change.
obj_memory	string	The data bytes (size() bytes) as characters up to the first zero byte, at most 1023 characters. The object does not change.

Other objects

Source	Destination	Behaviour
obj_can_msg	obj_can_msg	Copy of the whole message (ID, flags, port, data).
obj_lin	obj_can_msg	Data, size and id1 = LIN identifier (standard message, not RTR).
obj_can_msg	obj_lin	Data, size (max. 8) and the identifier id1 & 0x3F.
obj_lin	obj_lin	Copy of the LIN message.
obj_memory	obj_can_msg	The bytes of the block (max. 64), size by the size of the block, over 8 bytes fd = true.
obj_can_msg	obj_memory	The block is allocated for size bytes and the data bytes are copied.
obj_lin	obj_memory	The block is allocated for size bytes and the data bytes are copied.
obj_memory	obj_memory	Copy of the block.
obj_uds	obj_can_msg	The next frame to send; if there is none, the message does not change.
obj_j1939	obj_can_msg	The next frame to send; if there is none, the message does not change.
obj_can_msg	obj_uds	Passing a received frame.
obj_can_msg	obj_j1939	Passing a received frame.
obj_can_msg	obj_rs232	Sends the data bytes of the message (size bytes). If the port is not open, nothing is sent.
obj_can_msg	obj_socket_server	Sends the data bytes of the message to all clients (TCP).
obj_can_msg	obj_socket_client	Sends the data bytes of the message to the server; nothing without a connection.
obj_timepoint	obj_timepoint	Copy of the time marks.
obj_pid	obj_pid	Copy of the settings and the state of the controller.

- Assigning an object to itself (v = v;, q = q;, msg = msg;) changes nothing.
- After an assignment the inner iterator of the destination container points to the first element; get() right after v = w; returns the first element.
- Comparison ==, != does not change any object.
- Values going to bytes (messages, memory, transports, ports, string) are truncated to 8 bits: 256 gives 0, -1 gives 255. Double containers are first truncated to integers (65.9 gives 65).

```
csv.read("profile.csv");
row = csv;                      // 1st row of the table
while( row.size() > 0 ) {
    ...
    row = csv;                  // next row
}

msg.size = 8;
msg = rx_queue;                 // 8 bytes from the queue, the rest stays
client = "Ping";                // sending a string
answer = server;                // received data (no data: length 0)
text = answer;                  // vector to a string
```

> *Examples: C06_object_assignment.prg, C07_object_assignment_matrix.prg, C09_object_assignment_new.prg, C10_object_assign_edge.prg, T38_socket_vector.prg, T39_assign_rules.prg*

Assignment of vectors and queues

A vector (obj_vector_...) and a queue (obj_deque_...) behave differently in assignments: a vector is storage that an assignment only reads, a queue is a buffer that the destination takes the data out of. The rules apply to all assignments (= and +=):

Situation	obj_vector_int / obj_vector_double	obj_deque_int / obj_deque_double
destination of =	the content is replaced	the elements are appended to the end
destination of +=	the elements are appended to the end	the elements are appended to the end
source of an assignment (= and +=)	does not change	the elements taken by the destination are removed
destination with a fixed capacity (obj_can_msg and obj_lin by size, obj_datagrid 10 values, string 1023 characters)	the destination takes what fits	the destination takes what fits; the rest stays in the queue
destination that cannot take the data now (closed port, server without TCP, client without connection, closed graph or table window, log file not open)	nothing happens	the queue stays unchanged
source without data (nothing received, closed port, no connection, closed dataset, CSV after the last row, empty string)	the vector has length 0	the queue stays unchanged
assignment to itself (v = v; q = q;)	nothing changes	nothing changes
comparison ==, !=	does not change	does not change

```
obj_vector_int v;
obj_deque_int q, q2;

q += 1; q += 2;
v = q;           // v = {1, 2}, q is empty
q2 += 9;
q2 = v;          // q2 = {9, 1, 2}, v stays {1, 2}
q = q2;          // q = {9, 1, 2}, q2 is empty
msg.size = 2;
msg = q;         // msg.data(0) = 9, data(1) = 1, {2} stays in the queue
socket = v;      // sends 1, 2; v stays (can be sent again)
socket = q;      // sends 2, q is empty (without a connection it would stay)
```

- A queue is meant for data that are processed only once (receiving and sending in parts, splitting a stream of bytes into messages). Data that are used repeatedly (e.g. the same request sent to a socket again and again) belong to a vector.
- A vector to a CAN or LIN message transfers the whole content and sets size by the number of elements; a queue takes only the size bytes set in the message.
- To keep the content of a queue, copy it to a vector first (v = q;) and work with the vector; q = v; puts it back into the queue.
- Change against version 3.01: previously the source queue was emptied only when sent to obj_rs232 and sockets, even if the port was not open; other assignments (v = q;, q2 = q;, graph, CSV ...) left it unchanged. Sending a vector to a socket emptied it. A vector from a closed port, a socket without connection or a closed dataset stayed unchanged. Since version 3.02 the rules in the table apply everywhere.

Adding to a container

The operator += appends one value or the content of another object to the end of a container:

```
v += 5;          // the same as v.push_back(5);  obj_vector_..., obj_deque_...
vd += 0.25;
v += w;          // appends all elements of w to the end of v
v += v;          // doubles the content
q += msg;        // appends the data bytes of a CAN message
q += q2;         // moves the elements of queue q2 to the end of q, q2 is emptied
v += "abc";      // appends the characters of a string (3.02)
mem += v;        // obj_memory grows and the data are appended
```

Destination	Source

Destination	Source
obj_vector_int	number (as push_back), obj_vector_int, obj_vector_double, obj_deque_int, obj_deque_double, obj_can_msg, obj_lin, obj_memory, string (3.02)
obj_vector_double	number (as push_back), obj_vector_int, obj_vector_double, obj_deque_int, obj_deque_double, obj_can_msg, obj_lin, obj_memory, string (3.02)
obj_deque_int	number (as push_back), obj_vector_int, obj_vector_double, obj_deque_int, obj_deque_double, obj_can_msg, obj_lin, obj_memory, string (3.02)
obj_deque_double	number (as push_back), obj_vector_int, obj_vector_double, obj_deque_int, obj_deque_double, obj_can_msg, obj_lin, obj_memory, string (3.02)
obj_memory	obj_memory, obj_vector_int, obj_deque_int, obj_vector_double (3.02), obj_deque_double (3.02), obj_can_msg, obj_lin

The source queue is emptied by += (except q += q;, which doubles the queue), the source vector does not change. A string appends its characters as bytes 0 to 255.

Other combinations are compile errors, e.g. Unsupported operator += : "obj_can_msg" cannot append "unsigned". For data members of objects (msg.size += 2;) += is an ordinary compound assignment.

Object comparison

The operators == and != compare two objects of the same type and give a bool value:

Type	Compared
obj_can_msg	id1, stext (and id2 for an extended identifier), rtr, fd, size and the data bytes up to size; the data of an RTR message are not compared
obj_lin	identifier, length, data
obj_vector_*, obj_deque_*	number and values of the elements
obj_memory	size and content

```
if( msg != last_msg ) {          // send only on a change
    msg.send();
    last_msg = msg;
}
```

Objects of different types cannot be compared: Objects "obj_can_msg" and "obj_vector_int" cannot be compared. Message bytes after size are not compared.

> *Examples: C09_object_assignment_new.prg, C10_object_assign_edge.prg*

obj_can_msg

Working with a CAN message: building and sending a message, reading a received message (CAN_FUNCTION script). The data part has 64 bytes (CAN FD); classic CAN uses the first 8. The get_/set_ methods for multi-byte values use the index of the first byte; the index must suit the data length (e.g. set_uint32 at most 60).

Data members

unsigned id1

Message identifier (11-bit part).

unsigned id2

Second part of the identifier (18-bit part) for an extended identifier.

int size

Number of data bytes (DLC), 0 to 8, for CAN FD up to 64.

bool stext

false = standard (11-bit) identifier, true = extended (29-bit).

bool rtr

RTR flag.

int port

Port (CAN channel) number for transmission; for a received message the receiving port. Default 1.

bool fd

CAN FD message.

bool brs

CAN FD: bit rate switch.

Methods

void send()

Sends the message to the CAN bus.

void load()

Loads the message that started a CAN_FUNCTION (CAN_FUNCTION_AND_TIMER) script. It has no meaning for timer calls.

unsigned get_ID29()

Returns the identifier in the 29-bit form.

void set_ID29(unsigned id)

Sets id1 and id2 from an identifier in the 29-bit form.

void set_ID29_hex(string id)

Like set_ID29, the identifier is given as a hexadecimal string.

char data(int i)

Returns the i-th data byte.

void set_data(int i, char byte)

Sets the i-th data byte.

void data_from_hex(string hex)

Sets the data bytes from a hexadecimal string (e.g. "0102A0FF") and sets size according to the number of bytes.

void set_all(char b0, char b1, char b2, char b3, char b4, char b5, char b6, char b7)

Sets the first 8 data bytes.

void set_alli(int b0, int b1, int b2, int b3, int b4, int b5, int b6, int b7)

Like set_all, the values are given as int (the lowest byte is used).

void fill(char value)

Sets all 64 data bytes to the given value.

void fill_data(char value)

Same as fill.

void set_bool(int i, bool value)

Writes a bool value to the i-th byte (0 or 1).

bool get_bool(int i)

Reads a bool value from the i-th byte.

void set_bit(int i, bool value)

Sets a bit of the data part (bit index 0 to 63).

bool get_bit(int i)

Returns a bit of the data part (bit index 0 to 63).

void set_int16(int i, int value)

Writes a signed 16-bit value from the i-th byte. The _be variant uses the opposite byte order (big endian).

int get_int16(int i)

Reads a signed 16-bit value. The _be variant uses the opposite byte order (big endian).

void set_int16_be(int i, int value)

See set_int16.

int get_int16_be(int i)

See get_int16.

void set_uint16(int i, unsigned value)

Writes an unsigned 16-bit value. The _be variant uses the opposite byte order (big endian).

unsigned get_uint16(int i)

Reads an unsigned 16-bit value. The _be variant uses the opposite byte order (big endian).

void set_uint16_be(int i, unsigned value)

See set_uint16.

unsigned get_uint16_be(int i)

See get_uint16.

void set_int32(int i, int value)

Writes a signed 32-bit value. The _be variant uses the opposite byte order (big endian).

int get_int32(int i)

Reads a signed 32-bit value. The _be variant uses the opposite byte order (big endian).

void set_int32_be(int i, int value)

See set_int32.

int get_int32_be(int i)

See get_int32.

void set_uint32(int i, unsigned value)

Writes an unsigned 32-bit value. The _be variant uses the opposite byte order (big endian).

unsigned get_uint32(int i)

Reads an unsigned 32-bit value. The _be variant uses the opposite byte order (big endian).

void set_uint32_be(int i, unsigned value)

See set_uint32.

unsigned get_uint32_be(int i)

See get_uint32.

void set_float(int i, float value)

Writes a 32-bit float value.

float get_float(int i)

Reads a 32-bit float value.

`void set_double(int i, double value)`

Writes a 64-bit double value.

`double get_double(int i)`

Reads a 64-bit double value.

`void set_value(unsigned64 value, int start_bit, int length, bool big_endian, bool is_signed)`

Writes a signal of any length (1 to 64 bits) into the first 8 data bytes. The start bit is the lowest bit of the signal; for Motorola order (big_endian = true) higher bits continue into bytes with a lower index (DBC gives the highest bit for Motorola).

`unsigned64 get_value(int start_bit, int length, bool big_endian, bool is_signed)`

Reads a signal of any length from the given bit. A signed signal is sign-extended (the result can be cast to int64).

`void set_string(int first, int length, string str)`

Writes length characters of the string to the data from byte first.

`void get_string(int first, int length, string str)`

Stores length data bytes from byte first into str.

`unsigned get_timestamp()`

Time stamp of the received message.

`unsigned get_time_h()`

Reception time of the message - hour.

`unsigned get_time_m()`

Reception time - minute.

`unsigned get_time_s()`

Reception time - second.

`unsigned get_time_ms()`

Reception time - milliseconds.

`void print()`

Prints the message to the OUTPUT window in decimal, e.g. ST ID 111-8: 0,0,0,0,0,0,0,0 or EXT ID 111-123456(29221440)-8: RTR (11-bit part, 18-bit part, the 29-bit form in parentheses, DLC).

`void print_hex()`

Like print, values in hexadecimal.

Signals (set_value, get_value)

The set_value and get_value methods write and read a signal of any length 1 to 64 bits in the first 8 data bytes. The start bit is always the lowest bit of the signal (bit 0 = bit 0 of byte 0, bit 8 = bit 0 of byte 1). Intel (big_endian = false): further bits ascend towards higher bytes. Motorola (big_endian = true): higher bits continue into bytes with a lower index. A 12-bit Motorola signal from bit 8 therefore occupies the whole byte 1 (lower 8 bits) and bits 0 to 3 of byte 0. Note: DBC files give the most significant bit as the start bit of Motorola signals.

CAN FD

A CAN FD message has fd = true and up to 64 data bytes; valid lengths above 8 bytes are 12, 16, 20, 24, 32, 48 and 64. brs enables the higher data phase bit rate. The get_/set_ methods for bytes, numbers, text and bits work with all 64 bytes. data_from_hex and assigning a vector with more than 8 bytes set fd = true automatically.

> *Examples: O01_can_message_build.prg, O07_can_fd_messages.prg, O08_can_memory_timepoint_more.prg, P04a/P04b (CAN FD transmitter and receiver)*

obj_lin

LIN message for the USB2LIN converter: writing a LIN table item (master/slave) and loading a received LIN message in a LIN_FUNCTION script. See the LIN chapter.

Data members

`unsigned index`

Table item index (0 to 31).

`unsigned id`

LIN identifier (0 to 63).

`int size`

Number of data bytes (0 to 8).

`int port`

LIN port (1 or 2).

`unsigned timeout`

Response timeout.

`unsigned period`

Transmission period of the item in master mode: 0 off (LIN_PERIOD_OFF), 0xFFFF once (LIN_ONE_SHOT).

`bool rx`

true = data are received (response of another node), false = data are sent by the converter.

`unsigned checksum`

Checksum of the received message.

`unsigned flags`

Flags of the received message (bit 0x80 = response timeout).

`unsigned error_code`

Error code of the received message.

Methods

`void set()`

Writes the table item (index, id, size, period, timeout, rx and data) to the converter. Prints a message for a wrong index, length or port.

`char data(int i)`

Returns the i-th data byte (0 to 7).

`void set_data(int i, char byte)`

Sets the i-th data byte (0 to 7).

`void load()`

Loads the LIN message that started a LIN_FUNCTION (LIN_FUNCTION_AND_TIMER) script: id, size, port, data, checksum, flags, error_code and reception time.

`bool rx_timeout()`

Returns true if the response timeout expired for the received message.

`unsigned get_timestamp()`

Time stamp of the received message.

unsigned get_time_h()

Reception time - hour.

unsigned get_time_m()

Reception time - minute.

unsigned get_time_s()

Reception time - second.

unsigned get_time_ms()

Reception time - milliseconds.

> *Example: D05_lin_master.prg*

obj_rs232

Working with a serial port.

Methods

bool open(unsigned port_number, unsigned baud_rate)

Opens the given serial port at the given speed. Returns true on success.

bool open_ex(unsigned port_number, unsigned baud_rate, unsigned byte_size, unsigned parity, unsigned stop_bits)

Like open, plus data bits (4 to 8), parity (0 to 4 = none, odd, even, mark, space) and stop bits (0, 1, 2 = 1, 1.5, 2).

void close()

Closes the port.

bool is_ok()

Returns true if the port was opened successfully.

bool data_received()

Returns true if received data are available.

char get()

Returns a received byte.

void send(char data)

Sends a byte.

void send_string(string str)

Sends a string.

void send_log()

Sends the content of the log text buffer (the buffer is kept).

void set_rts(bool state)

Sets the RTS signal.

void set_dtr(bool state)

Sets the DTR signal.

bool get_cts()

Returns the state of the CTS signal.

bool get_dsr()

Returns the state of the DSR signal.

obj_vector_int

Array of integers (simplified vector container). The container keeps an internal iterator (current element) used by first, next, get, insert, erase, find and other methods.

Methods

`int size()`

Returns the number of elements.

`void clear()`

Removes all elements.

`void push_back(int value)`

Appends an element.

`int pop_back()`

Returns the last element and removes it.

`int at(int index)`

Returns the element at index.

`void modify(int index, int value)`

Changes the element at index.

`int first()`

Moves the iterator to the first element and returns its value.

`bool next_exist()`

Returns true if there is another element after the current one; false at the end and in an empty vector.

`int next()`

Moves the iterator to the next element and returns its value. After the last element it returns 0 and the iterator stays at the end.

`int get()`

Returns the current element (of the iterator).

`void iterator(int index)`

Moves the iterator to element index.

`void insert(int value)`

Inserts a value before the current element.

`void erase()`

Removes the current element.

`void remove(int first, int last)`

Removes the elements first to last-1.

`void sort()`

Sorts the elements in ascending order.

`void reverse()`

Reverses the order of the elements.

`void fill(int value)`

Sets all elements to the given value.

```
void replace_value(int old_value, int new_value)
```

Replaces all occurrences of a value.

```
void add(int first, int last, int value)
```

Adds the value to the elements first to last-1. Similarly sub, mul, div.

```
void sub(int first, int last, int value)
```

See add (subtraction).

```
void mul(int first, int last, int value)
```

See add (multiplication).

```
void div(int first, int last, int value)
```

See add (division).

```
int sum()
```

Returns the sum of the elements.

```
double average()
```

Returns the average of the elements.

```
int median()
```

Returns the median of the elements.

```
void min()
```

Moves the iterator to the smallest element.

```
void max()
```

Moves the iterator to the biggest element.

```
int min_val()
```

Returns the smallest value.

```
int max_val()
```

Returns the biggest value.

```
int count(int value)
```

Returns the number of elements with the given value.

```
int find(int value)
```

Searches for a value from the current element; returns its index (the iterator moves after it) or -1. Repeated calls find all occurrences.

```
int find_string(string str)
```

Searches for the character sequence of the string from the current element; returns the index or -1.

```
int find_first_lower(int value)
```

Returns the index of the first element lower than value, or -1.

```
int find_first_bigger(int value)
```

Returns the index of the first element bigger than value, or -1.

```
int adjacent_find()
```

Searches from the beginning for two equal adjacent elements; returns the index of the first one, or -1.

```
int adjacent_next()
```

Like adjacent_find, searches from the current element (next occurrence).

```
int xor()
```

Returns the XOR of all elements (e.g. a checksum).

```
unsigned64 crc(char bits, unsigned64 poly, unsigned64 init, bool ref_in, bool ref_out,
unsigned64 xor_out)
```

Computes a CRC over the elements (bytes) with any parameters, e.g. CRC-16/CCITT: crc(16, 0x1021, 0xFFFF, false, false, 0).

```
unsigned fletcher16_mod255()
```

Fletcher-16 checksum (modulo 255) of the elements (bytes).

```
unsigned fletcher16_mod256()
```

Fletcher-16 checksum (modulo 256).

```
void to_string(string str)
```

Converts the elements (lower bytes) to the characters of a string.

```
void tostring(string str)
```

Same as to_string (name used by older scripts).

```
void from_string(string str)
```

Fills the vector with the character codes of a string (the old content is removed).

```
void from_string_append(string str)
```

Like from_string, the characters are appended.

```
void from_hex_string(string hex)
```

Fills the vector with bytes from a hexadecimal string, e.g. "30313233".

```
void encode_base64(string str)
```

Encodes the elements (bytes) into a Base64 string.

```
void decode_base64(string str)
```

Decodes a Base64 string into the vector.

```
void print_hex()
```

Prints the elements in hexadecimal to the OUTPUT window (8 per line).

> *Examples: C01_vector_int_basics.prg, C02_vector_int_algorithms.prg, C03_vector_text_checksums.prg, C08_containers_more.prg*

obj_vector_double

Array of floating-point numbers. The methods have the same meaning as for obj_vector_int, the values are double.

Methods

```
int size()
```

Number of elements.

```
void clear()
```

Removes all elements.

```
void push_back(double value)
```

Appends an element.

```
double pop_back()
```

Returns and removes the last element.

```
double at(int index)
```

Element at index.

```
void modify(int index, double value)
```

Changes the element at index.

```
double first()
```

Iterator to the first element.

```
bool next_exist()
```

Another element exists.

```
double next()
```

Moves the iterator to the next element and returns its value; after the end it returns 0.

```
double get()
```

Current element.

```
void iterator(int index)
```

Sets the iterator.

```
void insert(double value)
```

Inserts a value before the current element.

```
void erase()
```

Removes the current element.

```
void remove(int first, int last)
```

Removes the elements first to last-1.

```
void sort()
```

Sorts ascending.

```
void reverse()
```

Reverses the order.

```
void fill(double value)
```

Sets all elements.

```
void replace_value(double old_value, double new_value)
```

Replaces a value.

```
void add(int first, int last, double value)
```

Adds the value to the elements first to last-1.

```
void sub(int first, int last, double value)
```

Subtracts the value.

```
void mul(int first, int last, double value)
```

Multiplies by the value.

```
void div(int first, int last, double value)
```

Divides by the value.

```
double sum()
```

Sum of the elements.

```
double average()
```

Average of the elements.

```
double median()
```

Median of the elements.

`void min()`

Iterator to the smallest element.

`void max()`

Iterator to the biggest element.

`double min_val()`

Smallest value.

`double max_val()`

Biggest value.

`int find(double value)`

Searches for a value from the current element, returns the index or -1.

`int count(double min, double max)`

Number of elements in the range min to max (inclusive).

`int find_first_lower(double value)`

Index of the first element lower than value, or -1.

`int find_first_bigger(double value)`

Index of the first element bigger than value, or -1.

`int adjacent_find()`

Index of two equal adjacent elements, or -1.

`int adjacent_next()`

Next occurrence of equal adjacent elements.

> *Examples: C04_vector_double.prg, C08_containers_more.prg*

obj_deque_int

Double-ended queue of integers (deque). In an assignment (= and +=) the destination takes the elements out of the queue; assigning to a queue appends the elements to its end (see Assignment of vectors and queues). The container keeps an internal iterator (current element) used by first, next, get, insert, erase, find and other methods.

Methods

`int size()`

Returns the number of elements.

`void clear()`

Removes all elements.

`void push_back(int value)`

Appends an element at the end.

`void push_front(int value)`

Inserts an element at the front.

`int pop_back()`

Returns and removes the last element.

`int pop_front()`

Returns and removes the first element.

`int at(int index)`

Returns the element at index.

`int first()`

Moves the iterator to the first element and returns it.

`bool next_exist()`

Returns true if another element exists.

`int next()`

Moves the iterator and returns the next element; after the end it returns 0.

`int get()`

Returns the current element.

`void insert(int value)`

Inserts a value before the current element.

`void erase()`

Removes the current element.

`void tostring(string str)`

Converts the elements (lower bytes) to the characters of a string.

`void to_string(string str)`

Same as tostring.

> *Examples: C05_deque_queue_stack.prg, C08_containers_more.prg*

obj_deque_double

Double-ended queue of floating-point numbers. The methods and the behaviour in assignments have the same meaning as for obj_deque_int.

Methods

`int size()`

Number of elements.

`void clear()`

Removes all elements.

`void push_back(double value)`

Appends at the end.

`void push_front(double value)`

Inserts at the front.

`double pop_back()`

Returns and removes the last element.

`double pop_front()`

Returns and removes the first element.

`double at(int index)`

Element at index.

`double first()`

First element (iterator).

`bool next_exist()`

Another element exists.

`double next()`

Moves the iterator to the next element and returns its value; after the end it returns 0.

`double get()`

Current element.

`void insert(double value)`

Inserts before the current element.

`void erase()`

Removes the current element.

obj_timepoint

Measuring time intervals. The object keeps the moment of its creation (script start) and two time points set by set_begin and set_end.

Methods

`void set_begin()`

Stores the current time as the begin point.

`void set_end()`

Stores the current time as the end point.

`unsigned64 duration_s()`

Interval between set_begin and set_end in seconds. The _ms and _us variants in milliseconds and microseconds.

`unsigned64 duration_ms()`

See duration_s.

`unsigned64 duration_us()`

See duration_s.

`unsigned64 duration_s_swap()`

Like duration_s and the end point becomes the new begin point (measuring consecutive intervals).

`unsigned64 duration_ms_swap()`

See duration_s_swap.

`unsigned64 duration_us_swap()`

See duration_s_swap.

`unsigned64 duration_from_start_s()`

Time since the object was created, in seconds. Variants _ms and _us.

`unsigned64 duration_from_start_ms()`

See duration_from_start_s.

`unsigned64 duration_from_start_us()`

See duration_from_start_s.

> *Example: T16_time_timepoint.prg*

obj_csvfile

Working with CSV files (table data) that can be imported to and exported from Excel. The default cell separator is the semicolon (Excel with Czech regional settings saves CSV this way); set_separator sets another one, set_separator(0) detects it from the file contents. Values are read and written as strings; use the conversion functions or the assignment to a vector to convert them to numbers.

- Empty lines at the end of the file (an end of line after the last data) are not table rows; empty lines in the middle stay as rows with one empty cell.
- Rows may have different numbers of cells. size_c() returns the number of cells of the longest row, size_row() the number of cells of a given row. A cell outside the table or after the end of a shorter row has the default value (set_default, default "0").
- A UTF-8 BOM at the beginning of the file (Excel "CSV UTF-8") is skipped.
- A nonexistent file gives an empty table (size_r() = 0) and is_ok() returns false.
- The assignment vector = csv reads the active row and moves to the next one (chapter Object assignment). Cells are converted to numbers decimally, hexadecimally (0x1A) and with a decimal comma (1,5) unless the comma is the separator. The vector always has size_c() items.
- The assignment obj_logfile = vector writes values separated by commas; read such a file after set_separator(',') or set_separator(0).
- Quotes around cells are not processed, so a cell must not contain the separator.

```
csv.set_separator(0);                       // separator from the file
csv.read(script_dir() + "/data.csv");       // file next to the script
if( !csv.is_ok() ) { logs("File missing"); log_endl(); stop(); }
row = csv;                                  // 1st row
while( row.size() > 0 ) {
    msg = row;
    msg.send();
    row = csv;                              // next row
}
```

Methods

`void read(string file)`

Reads a file. Sets the active row for the assignment to a vector to the first row.

`void write(string file)`

Writes the data to a file with the set separator (default semicolon).

`bool set(unsigned row, unsigned col, string value)`

Writes a value to the given row and column. A cell outside the table enlarges the table.

`void get(unsigned row, unsigned col, string value)`

Reads the value of the given row and column into value. A cell outside the table or after the end of a shorter row returns the default value (set_default, default "0").

`void reserve(unsigned rows, unsigned cols, string value)`

Sets the size of the table; new cells are filled with value.

`unsigned size_r()`

Returns the number of rows.

`unsigned size_c()`

Returns the number of columns (cells of the longest row).

`void set_active_row(unsigned row)`

Sets the row read by the next assignment vector = csv (0 = first row).

`unsigned get_active_row()`

Returns the row read by the next assignment vector = csv. Requires the interpreter 3.02.

`void set_separator(char separator)`

Sets the cell separator for read and write (default ';'). 0 = read detects the separator from the first non-empty line: semicolon, otherwise tab, otherwise comma. Requires the interpreter 3.02.

`char get_separator()`

Returns the separator; after read with automatic detection the detected character. Requires the interpreter 3.02.

`void set_default(string value)`

Sets the value of a missing cell for get and the assignment to a vector (default "0"). Requires the interpreter 3.02.

`unsigned size_row(unsigned row)`

Returns the number of cells of a row in the file (0 outside the table). Requires the interpreter 3.02.

`bool is_ok()`

Returns true if the last read could open the file. Requires the interpreter 3.02.

> *Example: O03_files_cfg_csv_log.prg*

obj_cfgfile

Working with cfg/ini configuration files with sections [section] and items name=value. Items can be int, unsigned, float, bool (1, 0, y, n, yes, no) and string. The default value is given as text.

Methods

`void set_name(string file)`

Sets the file to read from and write to.

`int read_int(string section, string item, string default_value)`

Reads an item as int; returns default_value if not found.

`unsigned read_unsigned(string section, string item, string default_value)`

Reads an item as unsigned.

`float read_float(string section, string item, string default_value)`

Reads an item as float.

`bool read_bool(string section, string item, string default_value)`

Reads an item as bool.

`void read_string(string section, string item, string value, string default_value)`

Reads an item as a string into value.

`void write_int(string section, string item, int value)`

Writes an item.

`void write_unsigned(string section, string item, unsigned value)`

Writes an item.

`void write_float(string section, string item, float value)`

Writes an item.

`void write_bool(string section, string item, bool value)`

Writes an item.

`void write_string(string section, string item, string value)`

Writes an item.

A file name without a path refers to the PP2CAN working directory.

> *Example: O03_files_cfg_csv_log.prg*

obj_logfile

Writing to a text file, for example an event log.

Methods

`void set_name(string file)`

Sets (opens) the file for writing.

`void write_log()`

Writes the content of the log text buffer and an end of line.

`void write_string(string str)`

Writes a string.

`void write_string_endl(string str)`

Writes a string and an end of line.

`void close()`

Flushes and closes the file.

obj_datagrid

Window displaying up to 10 quantities (index 0 to 9) with a name and a value. Similar to the ui_... functions, but several windows can be used.

Methods

`void open(string name)`

Opens the window with the given title.

`void set_name(int index, string name)`

Sets the name of a quantity.

`void set_value_string(int index, string value)`

Sets the value of a quantity as text.

`void set_value_int(int index, int value)`

Sets an int value.

`void set_value_unsigned(int index, unsigned value)`

Sets an unsigned value.

`void set_value_double(int index, double value)`

Sets a double value.

`void update()`

Displays the names and values that were set.

`void save_pos()`

Stores the window position (by its title) in the application settings.

void load_pos()

Restores the window position stored by save_pos.

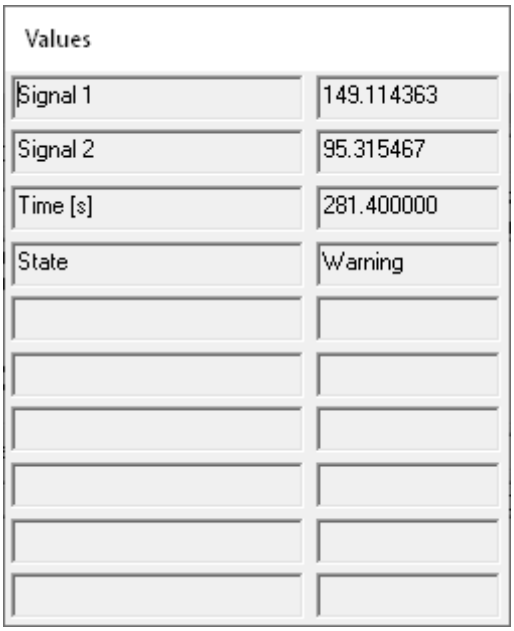


Fig. 4: Window of obj_datagrid

obj_dataset

Window for entering up to 10 values by the user (index 0 to 9). The user writes a value and confirms it with a button; the script detects it by updated and reads it by get_value_... Integers can also be entered in hexadecimal (0x...).

Methods

void open(string name)

Opens the window with the given title.

void set_name(int index, string name)

Sets the name of an item.

bool updated()

Returns true if the user confirmed a new value since the last call.

void get_value_string(int index, string value)

Reads the value of an item as text.

int get_value_int(int index)

Reads the value as int.

unsigned get_value_unsigned(int index)

Reads the value as unsigned.

double get_value_double(int index)

Reads the value as double.

void set_value_string(int index, string value)

Presets the value of an item (as if entered by the user).

void set_value_int(int index, int value)

Presets an int value.

```
void set_value_unsigned(int index, unsigned value)
```

Presets an unsigned value.

```
void set_value_double(int index, double value)
```

Presets a double value.

```
void save_pos()
```

Stores the window position (by its title) in the application settings.

```
void load_pos()
```

Restores the window position stored by save_pos.

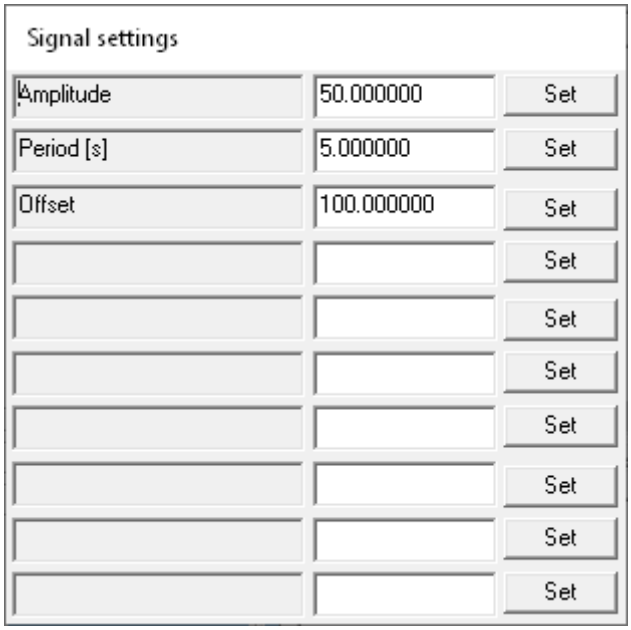


Fig. 5: Window of obj_dataset

obj_datagraph

Simple scrolling graph of up to 4 quantities (index 0 to 3) in the range 0 to 60000. Open the window by the open method in the init section. Settings made by the script are applied by the window timer (within 50 ms).

Since version 3.02 the window has the same size as the obj_datagauge window and a time axis is shown below the graph: time since the window was opened (12.5 s, 1:05, 1:02:03) in round steps that depend on the window width. Each point has the time of its set_value call, not of update(). When the mouse is over the graph, the values of the lines and the time of the point are shown.

Methods

```
void open(string name)
```

Opens the window with the given title.

```
void set_value(int index, int value)
```

Adds a value of a quantity.

```
void set_min(int value)
```

Sets the graph minimum (0 to 59000).

```
void set_max(int value)
```

Sets the graph maximum (1 to 60000).

```
void show_as_bar(int index, bool bar)
```

Shows the quantity as a bar graph.

```
void update()
```

Redraws the graph.

```
void save_pos()
```

Stores the window position (by its title) in the application settings.

```
void load_pos()
```

Restores the window position stored by save_pos.

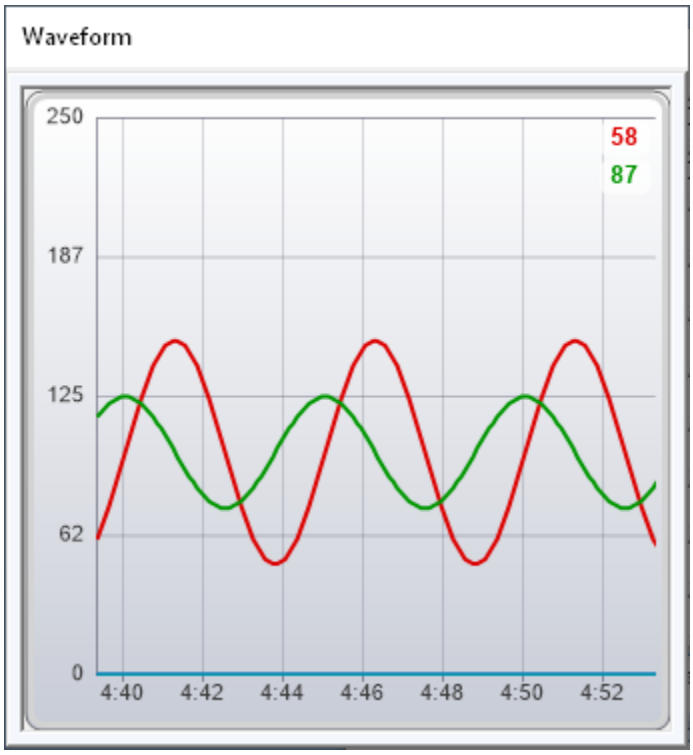


Fig. 6: Window of obj_datagraph

> *Examples: D02_display_windows.prg, D04_can_receiver_monitor.prg*

obj_datagauge

Window with a needle gauge with three coloured scale ranges. Open the window by the open method in the init section. Settings made by the script are applied by the window timer (within 50 ms).

Methods

```
void open(string name)
```

Opens the window with the given title.

```
void set_value(double value)
```

Sets the displayed value.

```
void set_range(double min, double max)
```

Sets the scale range.

```
void set_range1(double value)
```

Sets the end of the first (green) range.

```
void set_range2(double value)
```

Sets the end of the second (yellow) range; the rest is red.

```
void set_angle_range(double start, double end)
```

Sets the angular range of the scale.

```
void set_steps(int major, int minor)
```

Sets the number of major and minor scale steps.

```
void set_background(int r1, int g1, int b1, int r2, int g2, int b2)
```

Sets the background colours (gradient of two RGB colours).

```
void set_needle_color(int r, int g, int b)
```

Sets the needle colour.

```
void set_digit_pos(int x, int y)
```

Sets the position of the digital value.

```
void save_pos()
```

Stores the window position (by its title) in the application settings.

```
void load_pos()
```

Restores the window position stored by save_pos.

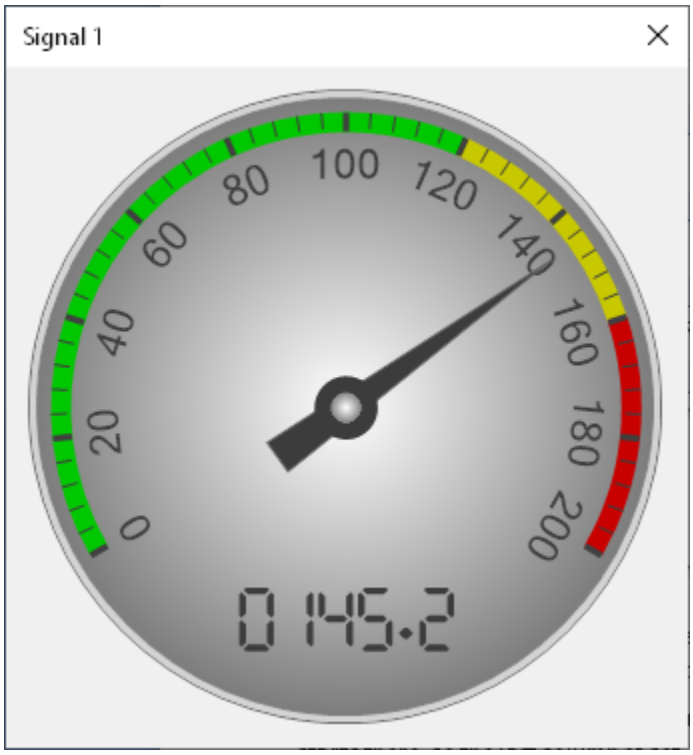


Fig. 7: Window of obj_datagquge

obj_datahistogram

Window with a histogram: columns for ranges of values and the number of values in each of them (since version 3.02, requires the interpreter 3.02). The window has the same size and look as the windows of obj_datagraph

and obj_datagauge. Columns and counts belong to the object, so the methods work also without an open window; after a change the window redraws itself in its timer (within 50 ms). The y axis is set automatically from the greatest count (round steps); with many columns only every n-th column is labelled.

A value belongs to a column when from ≤ value < to. The upper limit belongs to a column only if no other column starts there, so the last column of set_ranges contains max as well. If columns overlap, the first one applies. Values outside all columns are counted separately (get_out_of_range). A histogram has at most 1000 columns.

```
objects:
    obj_datahistogram hist;
end

init:
    hist.open("Engine speed");
    hist.set_ranges(0.0, 3000.0, 12);      // 12 columns of 250 rpm
    hist.set_axis_labels("Speed [rpm]", "Count");
end

body:
    hist.add_value(rpm);                   // with every message
end
```

Methods

`void open(string name)`

Opens the window with the given name.

`void set_ranges(double min, double max, int count)`

Removes the existing columns and creates count columns of equal width from min to max (count 1 to 1000, otherwise no column).

`int add_range(double from, double to)`

Adds a column for values from from to to (reversed limits are swapped). The label is the lower limit. Returns the index of the column, or -1 if there are already 1000 columns.

`void clear_ranges()`

Removes all columns.

`void set_label(int index, string label)`

Sets the label of a column below the x axis.

`int add_value(double value)`

Adds a value to its column. Returns the index of the column, or -1 for a value outside the columns.

`void set_count(int index, int count)`

Sets the count of a column (negative = 0), e.g. a histogram computed by the script.

`int get_count(int index)`

Returns the count of a column (0 for an invalid index).

`void clear()`

Sets all counts to zero; the columns remain.

`int get_total()`

Returns the sum of the counts of all columns.

`int get_out_of_range()`

Returns the number of values that belonged to no column.

`int get_ranges()`

Returns the number of columns.

`void set_title(string title)`

Sets the title of the graph.

`void set_axis_labels(string x, string y)`

Sets the labels of the x and y axes.

`void set_color(int r, int g, int b)`

Sets the color of the columns (default blue 0, 70, 220).

`void set_max(int max)`

Sets a fixed maximum of the y axis; 0 = from the greatest count (default).

`void save_pos()`

Saves the window position (by its name) to the application settings.

`void load_pos()`

Loads the window position saved by save_pos.

> *Example: T37_histogram.prg*

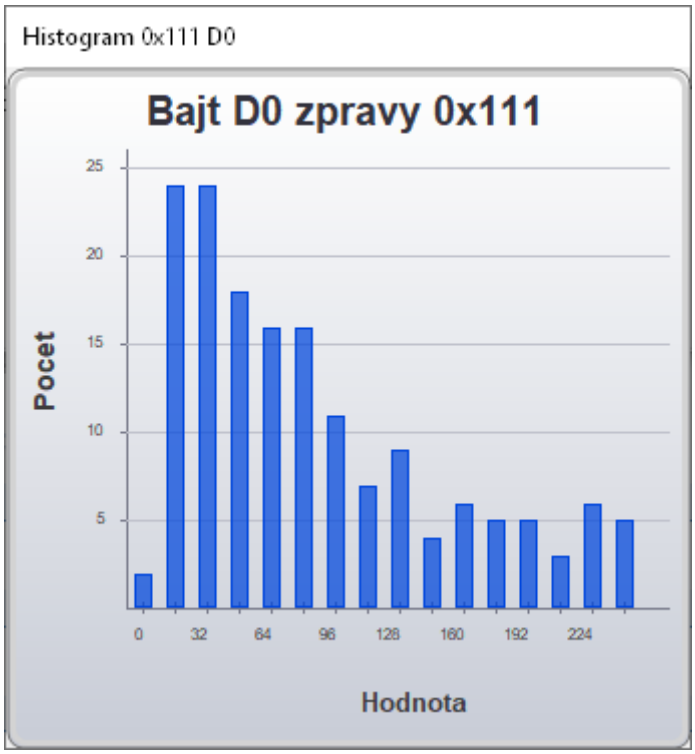


Fig. 8: Window of obj_dataghistogram

obj_socket_server

Server for TCP or UDP communication (always only one of them and one port). Several clients can connect to the server.

Methods

`void OpenTCP(unsigned port)`

Opens a TCP port.

`void OpenUDP(unsigned port)`

Opens a UDP port.

`void close()`

Closes the port.

`bool is_open()`

Returns true if the server is running.

`bool data_received()`

Returns true if data were received.

`char get()`

Returns a received byte.

`void send(char data)`

Sends a byte to all clients (TCP).

`void send_string(string str)`

Sends a string to all clients (TCP).

`void send_udp(char data, char ip1, char ip2, char ip3, char ip4)`

Sends a byte (UDP) to the given IP address.

`void send_string_udp(string str, char ip1, char ip2, char ip3, char ip4)`

Sends a string (UDP) to the given IP address.

`void buffer_push(char data)`

Adds a byte to the transmit buffer (max. 1024 bytes). It is better to send a packet at once.

`void buffer_send()`

Sends the buffer (TCP) and empties it.

> *Example: P03a_tcp_echo_server.prg*

obj_socket_client

Client for TCP or UDP communication.

Methods

`void OpenTCP(unsigned ip1, unsigned ip2, unsigned ip3, unsigned ip4, unsigned port)`

Connects to a server at the given IP address and TCP port.

`void OpenUDP(unsigned ip1, unsigned ip2, unsigned ip3, unsigned ip4, unsigned port)`

Connects to a server at the given IP address and UDP port.

`void close()`

Closes the connection.

`bool is_open()`

Returns true if the connection is open.

`bool data_received()`

Returns true if data were received.

`char get()`

Returns a received byte.

`void send(char data)`

Sends a byte.

`void send_string(string str)`

Sends a string.

`void buffer_push(char data)`

Adds a byte to the transmit buffer (max. 1024 bytes).

`void buffer_send()`

Sends the buffer (TCP) and empties it.

> *Example: P03b_tcp_client.prg*

obj_pid

PID controller. Default settings: dt 1, max 255, min 0, Kp 0.1, Kd 0.01, Ki 0.5.

Methods

`void set_dt(double dt)`

Calculation period.

`void set_max(double max)`

Output maximum.

`void set_min(double min)`

Output minimum.

`void set_kp(double kp)`

Proportional gain.

`void set_ki(double ki)`

Integral gain.

`void set_kd(double kd)`

Derivative gain.

`double calculate(double setpoint, double process_value)`

Calculates the manipulated variable for the setpoint and the process value (call it with the period dt).

> *Example: O04_pid_simulation.prg*

obj_memory

Memory block, e.g. for passing data to a DLL function (obj_dllfnc). The get_/set_ methods have the same meaning as for obj_can_msg and check the index range.

Methods

`void alloc(unsigned size)`

Allocates a block of the given size in bytes (frees the previous one).

`void free()`

Frees the block.

`unsigned size()`

Returns the block size.

unsigned pointer()

Returns the block address (for obj_dllfnc.set_param_memory).

void set(unsigned address, char value)

Writes a byte at the given position; nothing is written outside the block.

char get(unsigned address)

Reads a byte from the given position.

void fill(char value)

Fills the block with a value.

void print()

Prints the block to the OUTPUT window.

void set_int16(int i, int value)

See obj_can_msg.

int get_int16(int i)

See obj_can_msg.

void set_int16_be(int i, int value)

See obj_can_msg.

int get_int16_be(int i)

See obj_can_msg.

void set_uint16(int i, unsigned value)

See obj_can_msg.

unsigned get_uint16(int i)

See obj_can_msg.

void set_uint16_be(int i, unsigned value)

See obj_can_msg.

unsigned get_uint16_be(int i)

See obj_can_msg.

void set_int32(int i, int value)

See obj_can_msg.

int get_int32(int i)

See obj_can_msg.

void set_int32_be(int i, int value)

See obj_can_msg.

int get_int32_be(int i)

See obj_can_msg.

void set_uint32(int i, unsigned value)

See obj_can_msg.

unsigned get_uint32(int i)

See obj_can_msg.

void set_uint32_be(int i, unsigned value)

See obj_can_msg.

unsigned get_uint32_be(int i)

See obj_can_msg.

void set_float(int i, float value)

See obj_can_msg.

float get_float(int i)

See obj_can_msg.

void set_double(int i, double value)

See obj_can_msg.

double get_double(int i)

See obj_can_msg.

void set_string(int first, int length, string str)

See obj_can_msg.

void get_string(int first, int length, string str)

See obj_can_msg.

> *Example: O02_memory_object.prg*

obj_dllfnc

Calling a function of an external DLL. Procedure: load_dll, set_abi, set_param_num, parameter types (set_param_type) and return type (set_return_type), parameter values (set_param_...), call and reading the result (get_return_...). Types are given by the PARAM_... constants, the calling convention by the ABI_... constants (ABI_STDCALL is supported). At most 32 parameters. The object is available only in the 32-bit version of PP2CAN.

Methods

bool load_dll(string dll_name, string function_name)

Loads the library and looks up the function.

bool set_abi(int abi)

Sets the calling convention (ABI_STDCALL).

bool set_param_num(unsigned count)

Sets the number of parameters.

bool set_param_type(unsigned index, unsigned type)

Sets the parameter type (PARAM_UINT8 ... PARAM_POINTER).

bool set_return_type(unsigned type)

Sets the return type (PARAM_VOID ... PARAM_POINTER).

bool set_param_sint8(unsigned index, int value)

Sets a parameter value. Similarly for the other types.

bool set_param_uint8(unsigned index, unsigned value)

See set_param_sint8.

bool set_param_sint16(unsigned index, int value)

See set_param_sint8.

bool set_param_uint16(unsigned index, unsigned value)

See set_param_sint8.

bool set_param_sint32(unsigned index, int value)

See set_param_sint8.

bool set_param_uint32(unsigned index, unsigned value)

See set_param_sint8.

bool set_param_sint64(unsigned index, int64 value)

See set_param_sint8.

bool set_param_uint64(unsigned index, unsigned64 value)

See set_param_sint8.

bool set_param_float(unsigned index, float value)

See set_param_sint8.

bool set_param_double(unsigned index, double value)

See set_param_sint8.

bool set_param_string(unsigned index, string value)

Passes a pointer to a string (PARAM_POINTER).

bool set_param_memory(unsigned index, unsigned pointer)

Passes a pointer to an obj_memory block (value of obj_memory.pointer()).

bool call()

Calls the function. On an error (function not loaded, types or values not set) prints a message and returns false.

int get_return_sint8()

Returns the return value. Similarly for the other types.

unsigned get_return_uint8()

See get_return_sint8.

int get_return_sint16()

See get_return_sint8.

unsigned get_return_uint16()

See get_return_sint8.

int get_return_sint32()

See get_return_sint8.

unsigned get_return_uint32()

See get_return_sint8.

int64 get_return_sint64()

See get_return_sint8.

unsigned64 get_return_uint64()

See get_return_sint8.

float get_return_float()

See get_return_sint8.

double get_return_double()

See get_return_sint8.

obj_uds

Transfer of data longer than one CAN message according to ISO-TP (ISO 15765-2), e.g. for UDS diagnostics. Classic CAN and CAN FD, normal, extended and mixed addressing, data up to 16 MB. See the chapter Transfer of long data.

Data members

`unsigned tx_id`

ID of transmitted frames (default 0x7E0).

`unsigned rx_id`

ID of received frames (default 0x7E8).

`bool ext_id`

29-bit IDs.

`int port`

Port (CAN channel) of the issued frames (default 1).

`bool fd`

CAN FD.

`bool brs`

CAN FD: bit rate switch.

`int frame_size`

Frame length for CAN FD (8 to 64, default 8).

`int addressing`

Addressing: ISOTP_NORMAL (default), ISOTP_EXTENDED, ISOTP_MIXED. With extended and mixed addressing the first byte of a frame is the address.

`unsigned tx_address`

Address byte of transmitted frames.

`unsigned rx_address`

Address byte of received frames.

`int padding`

Padding of unused bytes (default 0xCC); -1 = no padding (shortened DLC).

`int rx_block_size`

Block size (BS) in the Flow Control prepared by the object on reception (0 = unlimited).

`int rx_stmin`

STmin in the Flow Control prepared by the object on reception.

Methods

`void clear()`

Clears the data and ends a running transfer in both directions.

`void set_string(string str)`

Sets the data to send from a string.

`void set_data(int i, char byte)`

Sets the i-th data byte (the data are extended if necessary).

char data(int i)

Returns the i-th data byte (sent or received).

int size()

Returns the number of data bytes.

void get_string(string str)

Stores the data in a string (max. 1023 characters).

void start()

Splits the data into frames (Single Frame, or First Frame and Consecutive Frames) and starts the transmission. The first frame is available immediately. With an invalid setting or without data, error() is set to TP_ERR_CONFIG.

bool has_frame()

Returns true if the object has a frame to send (a data frame allowed by flow control or a prepared response).

bool next_frame(obj_can_msg msg)

Puts the next frame into msg and returns true; if there is nothing to send, returns false and leaves msg unchanged. The assignment msg = object; works the same way.

void flow(int n)

Allows n more data frames (0 = all remaining). The script calls it after receiving Flow Control or CTS from the other side.

void select(int index)

Sets the index of the next frame to send (0 = the first frame). For J1939 the index is the packet number from CTS.

void abort()

Ends the transfer. For J1939 with a connection (RTS/CTS, ETP) an Abort message is prepared for sending.

int frame_count()

Returns the number of frames of the transmission.

bool tx_done()

Returns true if all frames of the transmission were issued.

bool receive(obj_can_msg msg)

Passes a received frame. Returns true if the frame belongs to the configured connection and the object processed it; flow control frames for the transmission (Flow Control, CTS, EOMA) are not processed and false is returned. The assignment object = msg; works the same way.

bool complete()

Returns true if all data were received. The data are then available through data, size, get_string or the assignment vec = object;.

int error()

Returns the error code: TP_OK, TP_ERR_SEQUENCE (wrong frame order), TP_ERR_UNEXPECTED (frame outside a transfer), TP_ERR_SIZE (data too long), TP_ERR_ABORT (aborted), TP_ERR_CONFIG (invalid setting), TP_ERR_FRAME (invalid frame).

> *Examples: O05_uds_loopback.prg, P02a_uds_ecu_simulator.prg, P02b_uds_tester.prg*

obj_j1939

Data transfer according to SAE J1939: single message, J1939-21 TP (BAM, RTS/CTS) and ETP, J1939-22 for CAN FD (Multi-PG, FD.TP). The protocol is selected by the data length and the destination. See the chapter Transfer of long data.

Data members

`unsigned pgn`

PGN of the transferred parameter group.

`int priority`

Priority of a single message and Multi-PG (default 6). Transport protocol frames have priority 7.

`unsigned source`

Own address (default 0xF9).

`unsigned dest`

Address of the other side, 0xFF = global (default). On reception with a value other than 0xFF only messages from this address are accepted.

`int port`

Port (CAN channel) of the issued frames (default 1).

`bool fd`

J1939-22 (CAN FD).

`bool brs`

J1939-22: bit rate switch (default true).

`int rx_packets_per_cts`

Number of packets in the CTS prepared by the object on reception (0 = maximum 255).

Methods

`void clear()`

Clears the data and ends a running transfer in both directions.

`void set_string(string str)`

Sets the data to send from a string.

`void set_data(int i, char byte)`

Sets the i-th data byte (the data are extended if necessary).

`char data(int i)`

Returns the i-th data byte (sent or received).

`int size()`

Returns the number of data bytes.

`void get_string(string str)`

Stores the data in a string (max. 1023 characters).

`void start()`

Splits the data into frames (single message, BAM, RTS/CTS, ETP or J1939-22) and starts the transmission. The first frame is available immediately. With an invalid setting or without data, error() is set to TP_ERR_CONFIG.

`bool has_frame()`

Returns true if the object has a frame to send (a data frame allowed by flow control or a prepared response).

bool next_frame(obj_can_msg msg)

Puts the next frame into msg and returns true; if there is nothing to send, returns false and leaves msg unchanged. The assignment msg = object; works the same way.

void flow(int n)

Allows n more data frames (0 = all remaining). The script calls it after receiving Flow Control or CTS from the other side.

void select(int index)

Sets the index of the next frame to send (0 = the first frame). For J1939 the index is the packet number from CTS.

void abort()

Ends the transfer. For J1939 with a connection (RTS/CTS, ETP) an Abort message is prepared for sending.

int frame_count()

Returns the number of frames of the transmission.

bool tx_done()

Returns true if all frames of the transmission were issued.

bool receive(obj_can_msg msg)

Passes a received frame. Returns true if the frame belongs to the configured connection and the object processed it; flow control frames for the transmission (Flow Control, CTS, EOMA) are not processed and false is returned. The assignment object = msg; works the same way.

bool complete()

Returns true if all data were received. The data are then available through data, size, get_string or the assignment vec = object;.

int error()

Returns the error code: TP_OK, TP_ERR_SEQUENCE (wrong frame order), TP_ERR_UNEXPECTED (frame outside a transfer), TP_ERR_SIZE (data too long), TP_ERR_ABORT (aborted), TP_ERR_CONFIG (invalid setting), TP_ERR_FRAME (invalid frame).

> *Example: O06_j1939_loopback.prg*

Transfer of long data (UDS, J1939)

The obj_uds (ISO-TP) and obj_j1939 (SAE J1939) objects split data longer than one CAN message into frames and assemble data from received frames. Timing (STmin, delays between BAM frames) and the evaluation of flow control from the other side are controlled by the script: the object issues further data frames only after the script allows them by flow(n). Until then has_frame() returns false. One object represents one connection and handles both directions; use two objects for simultaneous transmission and reception.

Transmission

- The data are filled by set_string, set_data or by the assignment object = vec; from obj_vector_int.
- start() prepares the frames. The first frame (Single Frame, First Frame, BAM, RTS) is available immediately.
- Frames are issued by next_frame(msg) or by the assignment msg = object; and sent by msg.send().
- After receiving Flow Control (UDS) or CTS (J1939) from the other side the script calls flow(n), for J1939 select(packet number from CTS) before it. Flow Control with the overflow status or an Abort message is handled by abort().
- For J1939 a CTS with 0 packets means holding the transfer: do not call flow in that case (flow(0) allows all frames).

Reception

- Received messages are passed by receive(msg) or by the assignment object = msg;. The object accepts only frames of its connection (ID and address for UDS, PGN and addresses for J1939).
- For a First Frame (UDS) or RTS (J1939) the object prepares the response (Flow Control, CTS; EOMA at the end for J1939). The script issues it like other frames (next_frame) and sends it.
- When all data are received complete() returns true; errors (e.g. a missing frame) are reported by error().
- Receiving a new transfer overwrites the data of the object.

obj_j1939 protocols

Data	dest = 0xFF (global)	dest = address
up to 8 B	single message	single message
9 to 1785 B	TP BAM	TP RTS/CTS
1786 B to 16 MB	not possible	ETP (DPO is inserted automatically)
`fd = true, up to 60 B`	Multi-PG	Multi-PG
`fd = true, above 60 B`	FD.TP BAM (up to 15300 B)	FD.TP RTS/CTS

A PGN of the PDU2 format (PF 240 and more) is always sent globally. Abort uses reason 3 (timeout). CTS and EOMA messages for the own transmission are evaluated by the script: TP.CM (PGN 0xEC00) with control byte 0x11 (CTS: byte 1 number of packets, byte 2 next packet number) and 0x13 (EOMA), ETP.CM (0xC800) with byte 0x15 (CTS: packet number in bytes 2 to 4) and 0x17, J1939-22 FD.TP.CM (0x4D00) with the control code in the lower 4 bits of byte 0 (1 = CTS: segment number in bytes 4 to 6, count in byte 7; 3 = EOMA).

Example: UDS read DID (0x22)

```
script:
    type = CAN_FUNCTION_AND_TIMER;
end
variables:
    bool sent;
end
objects:
    obj_uds uds;
```

```
    obj_can_msg msg;
    obj_vector_int vec;
end
init:
    uds.tx_id = 0x7E0;
    uds.rx_id = 0x7E8;
    uds.padding = 0x55;
    sent = false;
    ui_set_timer_script_period(10);
end
body:
    if( is_timer_call() ) {
        if( !sent ) {
            uds.clear();
            uds.set_data(0, 0x22);
            uds.set_data(1, 0xF1);
            uds.set_data(2, 0x90);
            uds.start();
            sent = true;
        }
        // request and Flow Control for the response
        while( uds.next_frame(msg) ) { msg.send(); }
    }
    else {
        msg.load();
        if( uds.receive(msg) ) {
            if( uds.complete() ) {
                vec = uds;
                vec.print_hex();
            }
        }
        else if( msg.id1 == 0x7E8 && (msg.data(0) & 0xF0) == 0x30 ) {
            // Flow Control for our long request
            if( (msg.data(0) & 0x0F) == 0 ) { uds.flow(msg.data(1)); }
            if( (msg.data(0) & 0x0F) == 2 ) { uds.abort(); }
        }
    }
end
```

Example: J1939 transmission by RTS/CTS and BAM

```
script:
    type = CAN_FUNCTION_AND_TIMER;
end
variables:
    unsigned id;
end
objects:
    obj_j1939 j;
    obj_can_msg msg;
end
init:
    j.pgn = 0xEF00;
    j.source = 0x10;
    j.dest = 0x30;           // 0xFF = BAM
    j.set_string("Data longer than 8 bytes");
    j.start();
    ui_set_timer_script_period(50);
end
body:
    if( is_can_function_call() ) {
        msg.load();
        if( !j.receive(msg) && msg.stext ) {
            id = msg.get_ID29();
            // TP.CM_CTS for us: select and flow
            if( ((id >> 16) & 0xFF) == 0xEC && (id & 0xFF) == 0x30 && msg.data(0) == 0x11 ) {
                if( msg.data(1) > 0 ) {
                    j.select((int)msg.data(2));
                    j.flow((int)msg.data(1));
                }
            }
        }
    }
    else {
        // one frame per timer period (50 to 200 ms for BAM)
        if( j.next_frame(msg) ) { msg.send(); }
```

```
    }
end
```

The next_frame and receive methods take an obj_can_msg object as a parameter.

LIN

With the USB2LIN converter a script can configure a LIN port (speed, master/slave/scan mode, LIN version), write LIN table items and process received LIN messages. Received messages are handled by the script types LIN_FUNCTION and LIN_FUNCTION_AND_TIMER; PP2CAN calls them for every LIN message received while the LIN table dialog is open. To process LIN messages in a CAN_FUNCTION script, select the PP2CAN mode in which a LIN message is passed as a CAN message.

- Master mode: the converter periodically transmits headers of table items with a non-zero period (LIN_ONE_SHOT = once). If rx = false it also sends the data, with rx = true it waits for the slave response.
- Slave mode: the converter responds with data to a header with the same ID if rx = false; with rx = true it only receives the data.
- Scan mode: the converter only listens to the bus traffic.

```
script:
    type = LIN_FUNCTION_AND_TIMER;
end
variables:
    bool ok;
end
objects:
    obj_lin tab;
    obj_lin rx;
end
init:
    ok = lin_config(1, LIN_MASTER, 19200, LIN_VER_20);
    tab.port = 1;
    tab.index = 0;
    tab.id = hex(10);
    tab.size = 8;
    tab.period = 100;
    tab.timeout = 50;
    tab.rx = true;
    tab.set();
end
body:
    if( is_lin_function_call() ) {
        rx.load();
        if( rx.rx_timeout() ) { prints("timeout"); }
        else { logc_hex(rx.data(0)); log_endl(); }
    }
end
```

Interpreter environment

- 4 scripts can run at the same time. Each has its own variables; data between scripts are passed through global variables (global_..._get/set).
- The init section is executed at start, the body section on every call according to the script type, the shutdown section at stop. Variables keep their values between calls.
- The execution time of a section is limited: body 5 s, init 5 min, shutdown 3 min. The limit also applies to user functions and func_N sections called from the section. When it is exceeded (e.g. an endless loop), the interpreter stops the script and prints Time limit 5000 ms exceeded, forced STOP instruction. Use the wait function only exceptionally.
- Open object windows (obj_datagrid, obj_dataset, obj_datagraph, obj_datagauge, obj_datahistogram) in the init section. Settings made by the script are applied by the window timer (within 50 ms).
- A string has at most 1023 characters, the buffer of the log functions at most 511 characters; a longer text is truncated.
- An integer division by zero prints the message Division by zero and the result is 0.
- Indexes and ranges outside valid values (at, get_bfu, set_string of a message, methods of obj_memory, first/last ranges of vectors ...) cause no error and no write outside memory: a read returns 0, a write is not done.
- Conversions of text to numbers give the same result on all computers; values out of range are limited to the minimum or maximum of the type.
- Nesting of function calls (user functions and func_N) is limited to 10 levels, recursion is not allowed.
- Files given by a relative path are looked up in the working directory of PP2CAN (function work_dir), not in the directory of the script. Open a file next to the script with the path script_dir() + "/name".
- A crash of a script (read or write outside memory, division by zero, stack overflow, out of memory) stops only the script, PP2CAN keeps running. The place of the crash is printed, e.g. Script crashed: access violation (read of address 0x0) in section body, instruction 57 (obj_csvfile.get), forced STOP instruction. Such a message means an error of the interpreter or of an object, not of the script – please send the script that caused it. After a write outside memory the memory of PP2CAN may be damaged, so restart PP2CAN after such a message. A damaged .scr file is not loaded.
- Before running, the interpreter checks all instructions of the script (numbers of variables, objects, string constants, jump targets). A damaged or incompatible .scr file is refused with a description of the place of the error and is not run.
- Object limits (3.02): obj_memory.alloc at most 256 MB; obj_csvfile at most 1,000,000 rows, 10,000 columns and 20,000,000 cells (set and reserve over the limit leave the table unchanged and print a message); size of obj_can_msg is limited to 0 to 64, of obj_lin to 0 to 8; get_string reads at most 1023 characters; the CRC width of obj_vector_int.crc must be 1 to 64 bits (otherwise the result is 0).
- In V2CAN mode or with connected ports a sent message can be received more than once (echo, second port). Scripts that receive their own or repeated messages should recognize duplicates (e.g. by a counter, see D04 and P04b).

Compiler

The Compile button in the script window starts the compiler compiler.exe. It checks the syntax and semantics (declarations, types, parameter counts) and translates the script into the binary form (.scr) executed by the interpreter. Errors are printed as <file: name> <line: number> error: text; after a successful compilation Script ready is printed. The compiler has version 3.02; a script using functions, methods or object operations that require a newer interpreter (3.01 or 3.02) is saved in a format that an older interpreter refuses to load, and the compiler points this out with a note.

The compiler can also be run from the command line:

```
compiler infile -t|-b outfile [-cz|-en]
```

Parameter	Meaning
-t	text output (instruction listing for checking)
-b	binary output .scr for PP2CAN
-cz	error messages in Czech (default)
-en	error messages in English

Warnings

Besides errors the compiler prints warnings in the form <file: name> <line: number> warning: text. Warnings do not stop the compilation: the script is saved and the number of warnings is printed at the end (Total warnings: n).

Warning	Cause
Function "f" may end without a return statement (the result is undefined)	A function with a return value in which some path (e.g. if without else) does not end with return. A loop while(true) without break that is left only by return does not cause the warning.
Statement is never executed (it follows return, break, continue or an endless loop)	Unreachable code. An endless loop is while(true) without break.
Variable "x" is never read	A variable (global or local) is only written or not used at all.
Operators && and || have the same priority in this language and are evaluated from left to right; use parentheses	a || b && c without parentheses (chapter Expressions and operators).
Unknown escape sequence "\U" in a string, the backslash is kept; write \\ or / in file paths	A backslash with an unknown character in a string (chapter Escape sequences).

Common error messages

The messages are shown in English as printed with the -en option; the default Czech texts have the same meaning.

Message	Cause and solution
Expected symbol "{"	The body of if, else, while or for is not in braces { }.
Expected symbol "}"	A block not closed before end or before the next section.
Expected symbol "end"	A section or function without end.
Expected symbol "break"	A case or default branch does not end with break, return or continue.
Symbol "x" caused a syntax error	A symbol where the grammar does not allow it (e.g. an extra }, else without if).

Message	Cause and solution
Operator "=" has operands of wrong type	Assignment of a larger type to a smaller one (double to int, int to unsigned, string to int); use a cast or a conversion function.
Identifier "x" is unknown	An undeclared variable or function, a typing error in the name.
Unsupported assignment: "A" cannot be assigned from "B"	An unsupported combination of objects (chapter Object assignment).
Unsupported operator += : "A" cannot append "B"	Chapter Adding to a container.
Objects "A" and "B" cannot be compared	Comparison of objects of different types.
Statement break can be used only in a loop or switch	break outside a loop and switch.
Statement continue can be used only in a loop	continue outside a loop (also in a switch outside a loop).
Statement return can be used only in a function	return in the init, body, shutdown or func_N section.
Only one default branch is allowed in a switch	Two default branches.
Section "x" is defined more than once	A section (except function) appears twice in the script.
Constant "X" must be initialized by a literal …	The value of const is not a literal or is out of the range of the type.
Constant "X" cannot be used here (a constant cannot be assigned)	Assignment to a constant.

Binary format

A compiled script (.scr) contains the script type, constants, object declarations and the instructions of all sections. The compiler writes the oldest format the script needs, so that it can also run in older PP2CAN versions:

Format	Use	Interpreter
V1	scripts that need no features of the formats V2 and V3	all versions
V2	user functions, LIN scripts, objects obj_lin, obj_uds and obj_j1939, general type cast and other instructions of version 3.00	3.00 and newer
V3	functions str_sub, str_trim, u2hex, clampi, tick_ms and others from this manual, operators += and == for objects, assignments of obj_memory, obj_lin, obj_csvfile, obj_logfile, obj_dataset, obj_timepoint, obj_pid	3.01 and newer
V4	function work_dir, new methods of obj_csvfile (set_separator, get_separator, set_default, size_row, is_ok, get_active_row), object obj_datahistogram, assignment between a string and an object, new assignment combinations (double containers, queue to obj_can_msg and obj_datagrid, queue from obj_csvfile and obj_dataset)	3.02 and newer

Language features that are compiled to existing instructions (break, continue, const, initialization, case ranges, blocks, escape sequences, script_dir, % for int64 ...) do not change the format. An older interpreter refuses to load a script in a format it does not understand.

Example and test scripts

The manual comes with a set of scripts in two variants: the cz folder contains scripts with Czech comments (without diacritics), the en folder with English comments. Names of variables and objects are in English. Comments "Napoveda:" / "Help:" at the first use of every function, method, keyword and operator explain their meaning, so the scripts also serve as a learning aid.

Self-checking tests

Scripts T, C and O are self-checking: they compare every result of a calculation or conversion with the expected value, print PASS or FAIL and a summary at the end, e.g. ==== T01: 29 PASS, 0 FAIL ====. The tests run in the init section, the body section stops the script. There should be no FAIL in the output. The comparison is done by the user functions check_int, check_double, check_str and others at the beginning of every test.

Examples and script pairs

- Scripts D are interactive examples (script window, data windows, CAN transmitter and receiver, LIN).
- Pairs (D03 + D04, P01a + P01b, P02a + P02b, P03a + P03b, P04a + P04b) are run at the same time in two CAN script windows. For pairs communicating over CAN use the V2CAN mode, two connected ports or two converters.
- D04 and P04b recognize duplicate messages (the same counter) and do not count them as losses; P02a and P02b ignore an immediately repeated frame.
- O03 creates the files script_test.cfg, script_test.csv and script_test.log itself in the PP2CAN working directory; no prepared files are needed.
- P03a and P03b communicate over TCP at 127.0.0.1, port 5000; start the server P03a first.
- D05 requires the USB2LIN converter and an open LIN table dialog.
- D06 is a summary example of the language: constants, initialization, break and continue, case ranges, functions and operations with objects.
- The scripts in the folders neg, warn and runtime check the compiler and the interpreter: neg scripts must not compile and state the expected error on the first line (// EXPECT: ...), warn scripts state the expected warnings (// EXPECT_WARNING: ...), runtime scripts the expected output of the interpreter (// EXPECT_OUTPUT: ...). These scripts and T22 to T35, C09, C10 and D06 have comments in Czech only.

List of scripts

Script	Content
C01_vector_int_basics.prg	obj_vector_int - basics
C02_vector_int_algorithms.prg	obj_vector_int - algorithms
C03_vector_text_checksums.prg	obj_vector_int - texts, hexadecimal data and checksums
C04_vector_double.prg	obj_vector_double - array of floating-point numbers
C05_deque_queue_stack.prg	obj_deque_int and obj_deque_double - queue and stack
C06_object_assignment.prg	Object assignment
C07_object_assignment_matrix.prg	Object assignment - all supported combinations
C08_containers_more.prg	More container methods
D01_ui_controls.prg	Script window controls (interactive example)
D02_display_windows.prg	Windows for entering and displaying values
D03_can_sender.prg	Periodic CAN message transmitter (pair with D04)
D04_can_receiver_monitor.prg	CAN message receiver and monitor (pair with D03)

Script	Content
D05_lin_master.prg	LIN master with the USB2LIN converter
O01_can_message_build.prg	obj_can_msg - building and reading a CAN message (without sending)
O02_memory_object.prg	obj_memory - memory block
O03_files_cfg_csv_log.prg	Files: configuration (cfg/ini), CSV table and log
O04_pid_simulation.prg	obj_pid - control simulation
O05_uds_loopback.prg	obj_uds (ISO-TP) - tester and ECU in one script
O06_j1939_loopback.prg	obj_j1939 - transmitter and receiver in one script
O07_can_fd_messages.prg	CAN FD messages (obj_can_msg)
O08_can_memory_timepoint_more.prg	More methods of obj_can_msg, obj_memory and obj_timepoint
P01a_global_producer.prg	Data producer through global variables (run together with P01b)
P01b_global_consumer.prg	Data consumer through global variables (run together with P01a)
P02a_uds_ecu_simulator.prg	UDS ECU simulator (run with P02b or with a real tester)
P02b_uds_tester.prg	UDS tester (run with P02a or against a real ECU)
P03a_tcp_echo_server.prg	TCP echo server (run with P03b)
P03b_tcp_client.prg	TCP client (run after P03a)
P04a_can_fd_sender.prg	CAN FD message transmitter (run with P04b)
P04b_can_fd_receiver.prg	CAN FD message receiver (run with P04a)
T01_data_types_constants.prg	Data types and constants
T02_arithmetic.prg	Arithmetic operators
T03_implicit_conversions.prg	Implicit (automatic) type conversions
T04_explicit_casts.prg	Explicit type cast (type)expression
T05_compound_assignment.prg	Compound assignment and ++ --
T06_bitwise_operators.prg	Bitwise operators & | ^ ~ << >>
T07_comparison_logic.prg	Comparison and logical operators
T08_control_flow.prg	Control structures: if, else if, while, for, switch
T09_user_functions.prg	User functions
T10_func_sections.prg	Sections func_0 to func_9 and the statement func(N)
T11_strings.prg	Working with strings
T12_string_conversions.prg	Conversions between numbers and strings
T13_math_functions.prg	Mathematical functions
T14_bit_functions_bcd.prg	Functions for bits, bytes and BCD code
T15_global_variables.prg	Global variables (shared by up to 4 PP2CAN scripts)
T16_time_timepoint.prg	Date, time and interval measurement (obj_timepoint)
T17_switch_details.prg	Details of the switch statement
T18_more_functions.prg	More built-in functions
T19_edge_cases.prg	Edge cases and safety of the language
T20_expressions_priority.prg	Expressions and operator priority
T21_body_calls.prg	Repeated calls of the body section
C09_object_assignment_new.prg	Object assignment of memory, lin, uds, csv, logfile, timepoint, pid; operators += and ==
C10_object_assign_edge.prg	Object assignment - edge cases, comparison, iterator

Script	Content
D06_new_features_301.prg	Summary example of the language (constants, break/continue, case ranges, objects)
T22_switch_return_if_break.prg	switch: return in a branch, if before break
T23_break_continue.prg	break and continue in loops
T24_escapes_literals.prg	Escape sequences and numeric literals
T25_initialization_const.prg	Initialization in declarations and const constants
T26_new_functions.prg	Functions str_sub, u2hex, clampi, tick_ms and others
T27_case_ranges.prg	Ranges in case
T28_runtime_safety.prg	Range checks in the interpreter
T29_discard_results.prg	Calls of functions and methods without using the result
T30_braces_layout.prg	Braces and code layout
T31_switch_flow.prg	switch: return, break, continue, blocks, order of branches
T32_assign_data.prg	Assignment of data between types, limit values
T33_constants.prg	Constants and literals
T34_number_text.prg	Conversions of numbers to text and back
T35_numeric_conditions.prg	Numeric conditions in if, while, for and the operators ! &&
T36_v302_features.prg	New in 3.02: obj_csvfile (separator, default value, short rows, hex), % for int64, work_dir, script_dir
T37_histogram.prg	obj_datahistogram: columns, sorting of values, limits, counts
T38_socket_vector.prg	Vector, queue and string through a socket: sending, receiving, no connection
T39_assign_rules.prg	Assignment rules 3.02: queue to CAN/LIN by size, datagrid, closed windows and files, double containers, string
neg/N01 … N28	Compile errors (every script states the expected error)
warn/W01 … W10	Compiler warnings
runtime/R01 … R03	Time limit and interpreter messages