

Scripting language for PP2CAN

User manual

Compiler and interpreter version 3.00

CANLAB s.r.o. · www.canlab.cz

Contents

Contents	2
Introduction	4
Purpose of scripts	4
First script	6
Script structure	7
Script types	7
Variables and objects.....	7
Comments.....	8
Data types and constants	9
Writing constants	9
Hexadecimal and binary notation.....	9
Implicit conversions.....	10
Type cast.....	10
Expressions and operators	11
Assignment	11
Statements.....	13
if, else if, else	13
while	13
for	13
switch.....	13
Stopping the script.....	13
Functions	14
User functions.....	14
Sections func_0 to func_9	14
Built-in functions	15
Conversion functions.....	15
String functions.....	17
Output.....	18
Arithmetic and mathematical operations	19
CAN interface.....	22
Date and time	22
Display and control of variables	23
Global variables	25
Other functions.....	26
Built-in constants	27
Objects.....	34
Object assignment	34
obj_can_msg.....	35
obj_lin	38

obj_rs232	39
obj_vector_int	40
obj_vector_double	42
obj_deque_int	44
obj_deque_double	45
obj_timepoint	46
obj_csvfile	46
obj_cfgfile	47
obj_logfile	48
obj_datagrid	48
obj_dataset	49
obj_datagraph	50
obj_datagauge	50
obj_socket_server	51
obj_socket_client	52
obj_pid	52
obj_memory	53
obj_dllfnc	54
obj_uds	56
obj_j1939	58
Transfer of long data (UDS, J1939)	60
Transmission	60
Reception	60
obj_j1939 protocols	60
Example: UDS read DID (0x22)	60
Example: J1939 transmission by RTS/CTS and BAM	61
LIN	63
Interpreter environment	64
Compiler	65
Example and test scripts	66
Self-checking tests	66
Examples and script pairs	66
List of scripts	66

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.

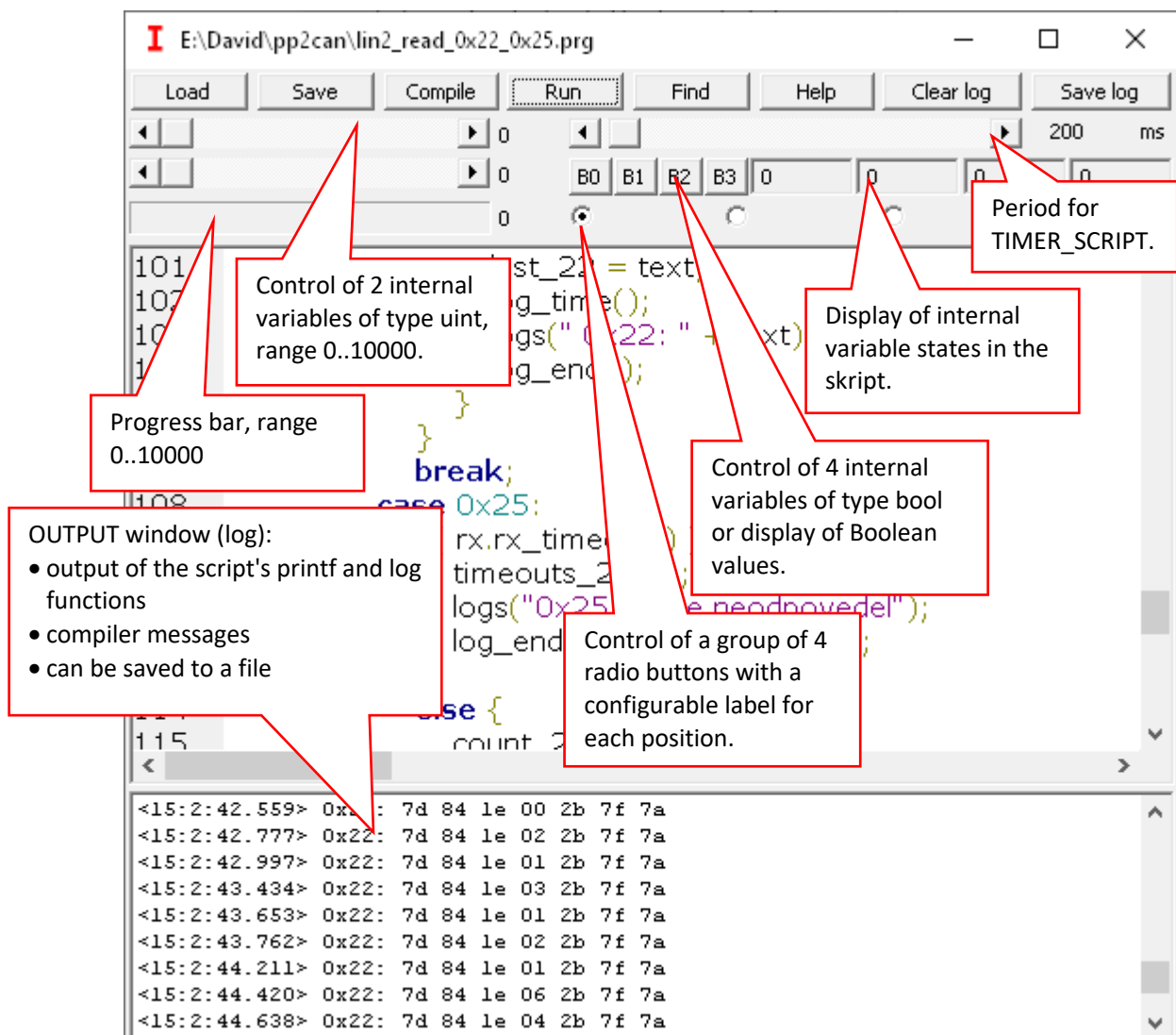


Fig. 1: CAN script window

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.

Window element	Meaning
Help	Help window describing the language, functions and objects.
Clear log, Save log	Clearing and saving the content of the OUTPUT window.
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.

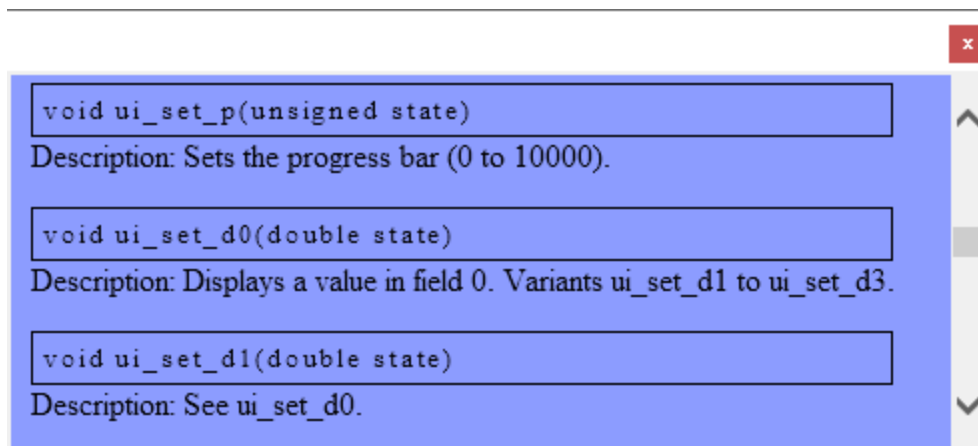


Fig. 2: CAN script help window

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.

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 sections does not matter, but declarations of variables and objects should come first.

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

Script types

The script type determines when PP2CAN calls the body section.

Type	Calls of the body section
CAN_FUNCTION	For every received CAN message; the message is read by the load() method of obj_can_msg.
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 a 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, objects in the objects section, always in the form type name1, name2;. They are global: valid in all sections and functions, and they 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
```

Identifiers start with a letter and may contain letters, digits and underscores. They are case sensitive; keywords are written in lower case.

Comments

```
// comment to the end of the line
/* comment over
   several lines */
```

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
int	32 bits	-2147483648 to 2147483647
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

Writing constants

Notation	Type	Example
integer	unsigned, above 32 bits unsigned64	100, 4294967296
floating-point (with a dot)	float	1.5, 2.5e-3, 1.0e10
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', '\t'
string in quotes	string	"text", "a\tb", "\"", "\\"
true, false	bool	

- A negative number is written with unary minus: -1, -2.5. The constant 1 is unsigned, the expression -1 is int.
- Floating-point constants are float. For a calculation with full double precision it is enough that one operand is double, e.g. (double)1 / 3.
- 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 need a cast to char.
- Built-in constants are available in scripts, e.g. M_PI, M_INT32_MAX, COLOR_RED (chapter Built-in constants).

Example: T01_data_types_constants.prg

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;  // bitwise operations with constants
u1 = 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 bigger type to a smaller one is a compile error: neither `c = 0x1FF`; nor `u = 0x100000000`; is allowed. A value can be narrowed by a cast: `c = (char)0x1FF`;
- Assigning 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 function `from_hex("1A2B")` converts a string with a hexadecimal number to a value; 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 bigger one; float with a 64-bit integer gives double and int with unsigned64 gives int64. char can be converted to string automatically.

An assignment and passing a parameter to a function allow only a conversion to a bigger type. A conversion to a smaller type (e.g. double to int) is a compile error and requires a cast.

Note: when comparing int 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`. Example: `T03_implicit_conversions.prg`

Type cast

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

- Integer to a smaller integer: the upper bits are cut off, e.g. `(char)300 = 44`, `(char)-1 = 255`.
- Floating-point to integer: the fractional part is truncated, `(int)-3.9 = -3`; values out of range are limited to the minimum or maximum of the type, NaN gives 0.
- To bool: the result is true if the value is not zero.
- Only char can be cast to string: `(string)c`. Numbers are converted to text by `i2str`, `d2str` and other 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
```

Example: `T04_explicit_casts.prg`

Expressions and operators

An expression consists of variables, constants, calls of functions with a return value, operators and parentheses. The operators in the table are sorted 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, type cast	minus: numbers; !: bool; ~: integers
* / %	multiplication, division, remainder	% integers only
+ -	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
&	bitwise AND	integers
^	bitwise exclusive OR (XOR)	integers
	bitwise OR	integers
&&	logical AND and OR (same priority)	bool

- The && and || operators have the same priority and are evaluated from left: `a || b && c` means `(a || b) && c`. Always use parentheses when combining them.
- Integer division truncates toward zero (`-7 / 2 = -3`), the remainder has the sign of the dividend (`-7 % 2 = -1`). An 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`.
- The calculation is done in the type of the operands: `char + char` is computed in 8 bits, unsigned * unsigned in 32 bits. For a bigger range cast one operand, e.g. `(int64)a * b`.
- For the ~, << and >> operators char is promoted to int, so `c << 8` does not lose the upper bits.
- Strings are compared alphabetically and case sensitively.

Examples: [T02_arithmetic.prg](#), [T06_bitwise_operators.prg](#), [T07_comparison_logic.prg](#), [T20_expressions_priority.prg](#)

Assignment

Notation	Meaning
<code>x = expression;</code>	assignment
<code>x += expression; -= *= /= %=</code>	compound assignment: <code>x = x + expression</code> etc.
<code>x &= expression; = ^= <<= >>=</code>	compound bitwise assignment
<code>x++; x--; ++x; --x;</code>	increment and decrement by 1

- An assignment is a statement, not an expression (`a = b = 0;` is not allowed).
- For 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 object variables (`msg.size += 2;`) and strings (`s += "abc"; s += c;`).

- `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.

Example: T05_compound_assignment.prg

Statements

Statements end with a semicolon. The bodies of if, else, while and for are always enclosed in braces, even when they contain a single statement.

if, else if, else

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

while

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

for

The first and third parts of the for header may contain only assignments separated by commas.

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

switch

The switch expression can be a number, a character or a string. Several values separated by commas can follow case. Every case and default branch must end with break; (or return in a function); default must be the last one. Strings are compared as a whole and case sensitively. break ends only the switch, not the surrounding loop.

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

The language has no continue statement and no break for loops; a loop is ended by its condition. Examples: T08_control_flow.prg, T17_switch_details.prg

Stopping the script

The function stop() ends the script (the shutdown section is executed). 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;
    counter++;
    return counter;
end
```

- Definition: function [type | void] name([ref] type parameter, ...), the body ends with 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, 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: its value is copied back after the return, so the argument must be a variable of the same type.
- Local variables are declared at the beginning of the body and are 0 on every call. A static variable keeps its value between calls. Local names hide global variables of the same name.
- return expression; returns a value, return; ends a function without a value.
- Functions can call other functions up to a depth of 10; recursion (also indirect) is not allowed and the interpreter stops the script.
- Parameters are of basic types. Objects are global and a function uses them directly by name.
- The result of a function or method with a return value must be used (assigned or used in an expression).

Examples: T09_user_functions.prg, T19_edge_cases.prg

Sections func_0 to func_9

Simple functions without parameters and a return value working with global variables. They are called by 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, 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 (e.g. 2, 16).

string u2str(unsigned value)

Converts an unsigned integer to a string.

string f2str(float value)

Converts a floating-point number to a string.

string d2str(double value)

Converts a floating-point number to a string.

int str2i(string value)

Converts an integer written in a string to its value.

unsigned str2u(string value)

Converts an integer written in a string to its value.

char str2c(string value)

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

bool str2b(string value)

Converts a logical value written in a string. The first character decides: 1, y, Y, a, A means true, anything else false.

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 to its value.

unsigned from_hex(string value)

Converts a hexadecimal number written in a string (e.g. "1A2B") to its value.

int truncf(float value)

Returns the integer part of a floating-point number.

int truncd(double value)

Returns the integer part of a floating-point number.

int roundf(float value)

Rounds a floating-point number.

int roundd(double value)

Rounds a floating-point number.

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) of an unsigned word.

char get_bfi(int value, int byte_index)

Returns the given byte (index 0 to 3) of an int word.

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

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).

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 by characters without them.

Examples: T11_strings.prg, T12_string_conversions.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 time as <h:m:s.ms>. 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.

Examples: T13_math_functions.prg, T14_bit_functions_bcd.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 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.

Examples: T16_time_timepoint.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.

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.

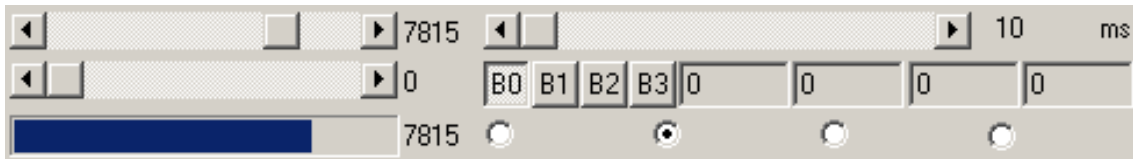


Fig. 3: Controls of the script window

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.

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. A constant takes precedence over a variable of the same name.

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)0.785398163397448309616
M_E	(double)2.71828182845904523536
M_SQRT2	(double)1.41421356237309504880
M_SQRT1_2	(double)0.707106781186547524401

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

Name	Value
ABI_FASTCALL	(unsigned int)2
ABI_MS_CDECL	(unsigned int)3
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

Name	Value
VK_SPACE	(int)0x20
VK_PRIOR	(int)0x21
VK_NEXT	(int)0x22
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

Name	Value
VK_P	(int)0x50
VK_Q	(int)0x51
VK_R	(int)0x52
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

Name	Value
VK_F11	(int)0x7A
VK_F12	(int)0x7B
VK_F13	(int)0x7C
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	windows for displaying and entering values, graph, gauge
obj_pid	PID controller
obj_dllfnc	calling a function of a DLL

Object assignment

The assignment a = b; between objects copies or converts data. Supported combinations:

Target = source	Result
obj_can_msg = obj_can_msg	copy of the whole message (ID, flags, port, data)
obj_vector_int = obj_can_msg	data bytes of the message (the vector content is replaced)
obj_deque_int = obj_can_msg	data bytes are appended to the queue
obj_can_msg = obj_vector_int	bytes from the vector (max. 64), size by the count; above 8 bytes fd = true
vector = vector / deque	copy of elements; a vector content is replaced
deque = vector / deque	elements are appended to the queue
int <-> double containers	conversion of values; double to int by truncation
obj_memory = obj_vector_int / obj_can_msg	allocates a block and copies the bytes
obj_vector_int / obj_deque_int = obj_rs232, socket	received data
obj_rs232, socket = obj_vector_int / obj_deque_int	sending data

Target = source	Result
obj_datagraph = container	values into the graph (line 0)
obj_csvfile = container	values into the active table row
obj_can_msg = obj_uds / obj_j1939	next frame to send
obj_uds / obj_j1939 = obj_can_msg	passing a received frame
obj_uds / obj_j1939 <-> obj_vector_int	transfer data

Assigning an object to itself changes nothing. After an assignment the internal iterator of the target container points to the first element.

Examples: C06_object_assignment.prg, C07_object_assignment_matrix.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(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 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.

int next()

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

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()

Next element.

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). 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.

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 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()

Next element.

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 (comma separated table data) that can be imported to and exported from Excel. Values are read and written as strings; use the conversion functions to convert them to numbers.

Methods**void read(string file)**

Reads a file.

void write(string file)

Writes the data to a file.

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

Writes a value to the given row and column.

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

Reads the value of the given row and column into value.

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

Creates or enlarges the table to the given size, new cells are filled with value.

unsigned size_r()

Returns the number of rows.

unsigned size_c()

Returns the number of columns.

*Example: 003_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: 003_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.

Zobrazeni dat	
Scrollbar	0
Radio	2
BO	Off
INT	-547
UNSIGNED	453
DOUBLE	-54.699999
	11111111111111

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.

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).

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. 5: 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.

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: 004_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: 002_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: 006_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.
- One execution of the body section is limited to 5 s; a longer run (e.g. an endless loop) is stopped by PP2CAN. Use the wait function only exceptionally.
- Open object windows (obj_datagrid, obj_dataset, obj_datagraph, obj_datagauge) 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.
- Nesting of function calls (user functions and func_N) is limited to 10 levels, recursion is not allowed.
- 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 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

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.

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)
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

Script	Content
006_j1939_loopback.prg	obj_j1939 - transmitter and receiver in one script
007_can_fd_messages.prg	CAN FD messages (obj_can_msg)
008_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