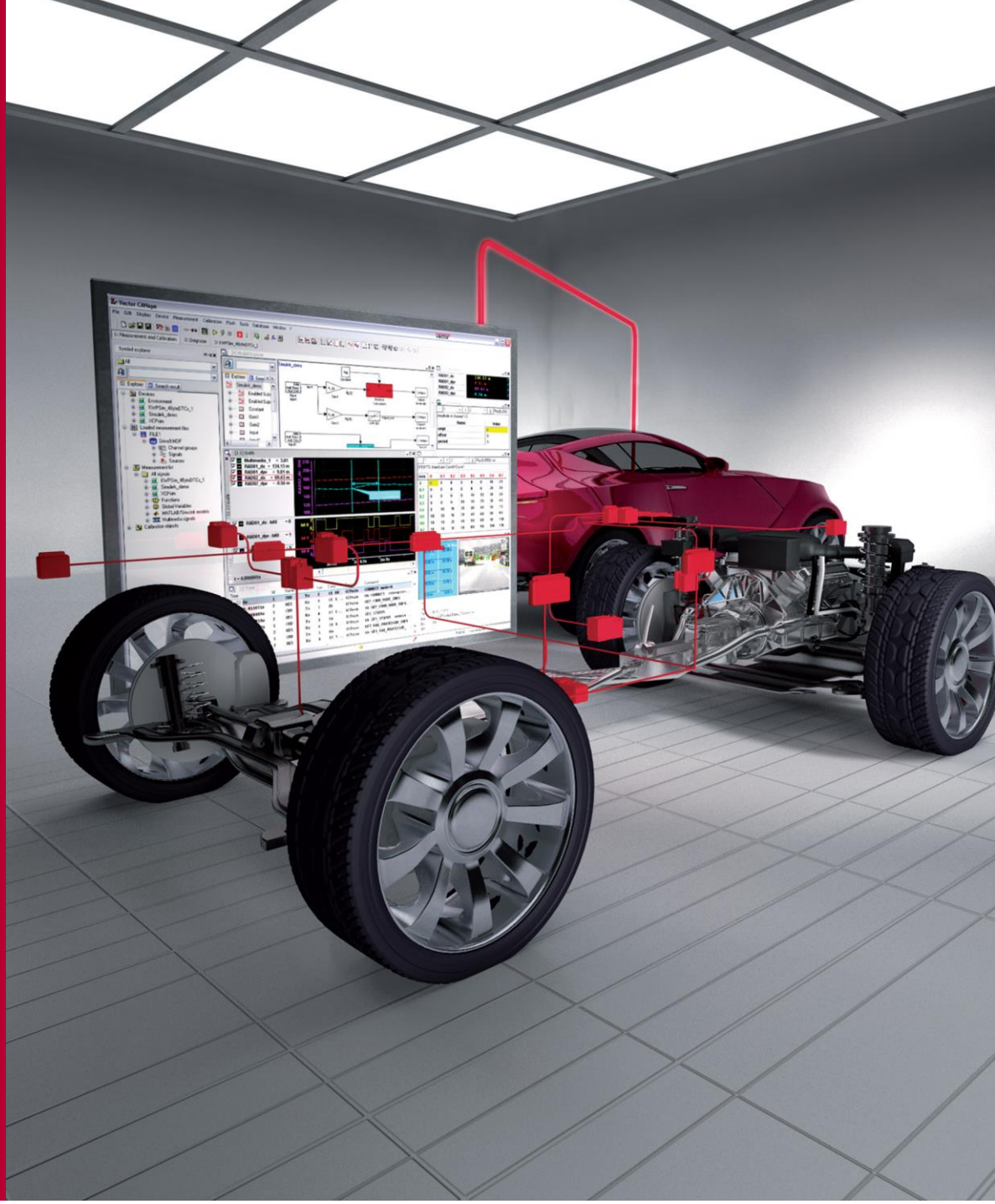


User Manual

CANape CASL

Calculation and Scripting Language

Version 1.2

English

Imprint

Vector Informatik GmbH
Ingersheimer Straße 24
D-70499 Stuttgart

Contents

1 Introduction

In this chapter you will find the following information:

1.1 Objective

Basic information	This manual starts with an introduction to the CANape scripting language where all associated concepts are explained in detail and distinguished from one another.
Syntax	The following chapter covers the syntax of the CANape scripting language. This chapter also serves as a reference guide.
Integrating of functions and scripts	Another chapter describes how to integrate and check functions and scripts in CANape.
General tips	Tips on handling general issues are also provided.

1.2 Vector Product Reference

CANape	The range of available functions differs depending on the respective Vector product (CANape or vSignalyzer) as well as the type (function or script). This manual refers exclusively to CANape. To determine whether the respective functions are also available in vSignalyzer, please refer to the Help.

1.3 CASL Scripting Language in CANape

Proprietary scripting language	CANape uses its own scripting language, hereinafter referred to as CASL (**C**alculation **a**nd **S**cripting **L**anguage).
Syntax	The syntax of CASL is very similar to the C programming language. It permits developers to integrate their own C code or Simulink models.

Note: Do not confuse CASL with the programming language CAPL, which is used in the CANoe and CANalyzer environments.

CAPL is an event-oriented programming language. So-called CAPL program nodes are used to specify when an event will be executed and the nature of the reaction. CASL, on the other hand, is a signal-oriented language.

1.4 Prior Knowledge

Prior programming knowledge	This manual assumes that you have general programming knowledge in the C programming language.

1.5 General Process

Programming process

Five steps are needed to develop a program.

1. Think about which task is to be the primary task of the program.
2. Decide how and when the program is to be executed.
3. Develop suitable code.
4. Compile the program.
5. Test the program in CANape.

1.6 About This User Manual

To Find information quickly

This user manual provides you with the following access help:

> At the beginning of each chapter you will find a summary of the contents.
> The header shows in which chapter of the manual you are.
> The footer shows the version of the manual.
> At the end of the user manual you will find a glossary to look-up used technical terms.
> At the end of the user manual an index will help you to find information quickly.

Conventions

In the two tables below you will find the notation and icon conventions used throughout the manual.

Style	Utilization
bold	Fields/blocks, user/surface interface elements, window- and dialog names of the software, special emphasis of terms. [OK] Push buttons in square brackets File|Save Notation for menus and menu entries
CANape	Legally protected proper names and marginal notes.
`Source Code`	File and directory names, source code, class and object names, object attributes and values
Hyperlink	Hyperlinks and references.
<Ctrl>+<S>	Notation for shortcuts.

Symbol	Utilization
	This icon indicates notes and tips that facilitate your work.
	This icon warns of dangers that could lead to damage.
	This icon indicates more detailed information.

Symbol	Utilization
	This icon indicates examples.
	This icon indicates step-by-step instructions.

1.6.1 Certification

Quality Management System

Vector Informatik GmbH has ISO 9001:2008 certification. The ISO standard is a globally recognized standard.

1.6.2 Warranty

Restriction of warranty

We reserve the right to modify the contents of the documentation or the software without notice. Vector disclaims all liabilities for the completeness or correctness of the contents and for damages which may result from the use of this documentation.

1.6.3 Support

Need help?

You can reach our hotline by telephone at

+49 (0)711 80670-200

or by e-mail at support@vector.com.

1.6.4 Trademarks

Protected trademarks

All brand names in this documentation are either registered or non-registered trademarks of their respective owners.

2 Basic Information

In this chapter you will find the following information:

2.1 Applications for Functions and Scripts

Introduction

CANape contains a function editor for writing cross-device functions and scripts.

The CASL scripting language used for this is similar to the C programming language. For easier use, CANape provides an IntelliSense input, code blocks, and various built-in function groups.

General applications

Functions and scripts can be used to solve a variety of different tasks from simple calculations, e.g., adding signals, to automation of CANape.

Functions are processed synchronously during a measurement. Functions are used mainly for various calculations and applications on an existing measurement file or an active measurement.

Scripts run independently of a measurement and are used for reproducible automation of sequences.

2.2 What Are Functions

Introduction

Functions are parts of a program code that is compiled under its own name. They can be defined across devices in CANape.

Syntax

The code of the function follows the rules of the CASL language (**C**alculation **a**nd **S**cripting **L**anguage).

Tasks

A function can be a mathematical formula or program code, for example, in which variables stand as placeholders for signals or parameters.

They facilitate analysis of measurement signals and enable the setting of calibration objects as well as other interventions at the device level.

Online applications

In the case of online application, functions are executed **during a measurement** when triggered by an event. This occurs according to the measuring mode set in the measurement configuration.

Thus, it is possible to

> calculate virtual measurement signals and
> enable write accesses to device memory or external measuring hardware.

Offline applications

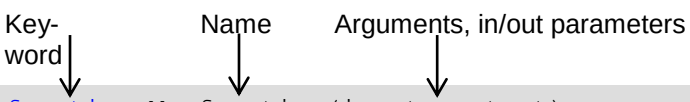
In the case of offline application, functions access existing measurement files.

Thus, it is possible to

> calculate virtual measurement file channels and
> perform Data Mining analyses.

Function structure

The syntactical structure of a function is as follows:



Function header

```
function My_function(input, output)
```

Function body

```
{
  double a = 1;
  long b = 2;
```

```
  output = a + b*input;

  writef("The result is: %d", output);
  return output;
}
```

Parameter passing

When a function is called, parameters can be passed to the function. The parameters are passed to the function generally as reference and can thus be used for the **input** as well as for the **output**.

For more details on the passing of parameters, see section Arguments and In/Out Parameters (of Functions) on page 15.

Control structure `return`

In addition to issuing results from functions by parameter passing, the path via the `return` control structure (also keyword) can also be used. By default a value of type `double` will be returned. The optional return value thus always contains a scalar.

It can be useful to return another type. For this purpose the return type must be declared in the function header. Fields or vectors (arrays) are not allowed as return values. An overview of all available control structures can be found in section Control Structures (Statements) on page 26.

Example: Function header for alternative return value `byte`

```
function byte TestFunction (signal)
{
  return 1;
}
```

Use

The `return` value of functions can be reused flexibly in CANape. For example, these can be displayed as a virtual measurement file channel in a Graphic window. For this reason, it is recommended that an individual scalar return value be returned via `return` and not via the parameter list.

If a function is not to return a value, this can be done in one of the following three ways:

- return
- cancel
- Neither `return` nor `cancel` are in the function code.

Note: Measurement values that are write-protected (e.g., provided with a red pointer in the measurement list) cannot be used as an **output** parameter of a function.

Function call

Functions can be inserted in the measurement signal list or used as a function definition for the virtual measurement file channel (e.g., for Data Mining). They can also be called directly from another user-defined function or a user-defined script. For details, see section Commissioning or Instantiating Functions on page 32.

Process priority

Functions – such as the measurement itself – run under CANape with very high priority. They should therefore be kept as efficient as possible since, otherwise, the function can influence the entire measurement (measured values may be lost, for example). Infinite loops tie up the entire measurement and can only be interrupted by

actuating the <Esc> key for 3 seconds.

Limitations

Because CANape processes are stopped during the measurement for functions, a few CASL functions are not available. They would trigger a runtime error. These CASL functions are therefore not even offered when user-defined functions are written, e.g., this applies to the built-in script function `Sleep()`.

Memory location

Functions are saved as project-related functions in the `canape.ini` configuration file in your working directory.

Function types

A distinction is made in CANape between user-defined functions and CASL functions predefined by CANape.

User-defined functions

User-defined functions include **project functions** written by the user as well as editable **library functions**. Both are displayed in the Symbol Explorer of CANape.

Global function library

You obtain the library functions if you choose to have the global function library created during the setup process (default setting). For details, see section Global Function Library on page 45).

Project function

You see the user-written functions or edited functions under **Project functions** in the tree view of the Functions Editor.

Function groups

By contrast, the user cannot change the CASL functions of various **function groups** predefined by CANape, such as diagnostic or Flash functions. These merely execute certain commands that you, in turn, can use within your user-defined functions and scripts. For details, see section Predefined Function Groups and Code Blocks of CANape on page 16.

Measurement function

A **measurement function** is understood to be the combination of the function and the measurement parameters (see section Using a Function During a Measurement on page 35). Measurement functions are displayed under **Measurement Signal|Functions** in the tree view of the measurement configuration in CANape.

2.3 What Are Scripts

Introduction

Scripts are parts of the program code and can be defined across devices in CANape.

Syntax

The code of a script follows the rules of the CASL language (**C**alculation **a**nd **S**cripting **L**anguage).

It can be written directly in the Editor window of the Functions Editor and is processed sequentially when called.

Tasks

Scripts are used in order to automate or control joint activities in CANape, such as the starting and stopping of measurements and other system-related sequences.

Scripts run **independently of the measurement**.

They can also be used to call external models that are generated in Microsoft Visual Studio or MATLAB/Simulink.

Script call-up

Scripts can be called in different ways. For call-up from the CANape user interface, see section Call-up of Scripts on page 53.

In addition, the script behavior can be controlled using command line options when CANape is running (see section Script Behavior When CANape is Running on page 58).

Process priority of scripts	Scripts are subject to a relatively low process priority. Their execution is guaranteed only every 100 ms.
Debugging	Debugging of scripts using breakpoints is available for diagnosing and locating logic errors in user-defined scripts. For details, see section Debugging of Scripts on page 58.
File format and memory location	Scripts are separate files in the `*.scr` or `*.cns` script format that can be copied to another project directory at any time. Scripts are saved in your working directory by default.
Example scripts	CANape includes examples scripts in its various sample configurations. These are displayed under **Scripts** in the tree view of the Functions Editor after the respective sample configuration is opened. The user-defined scripts are also listed here.

2.4 Differences Between Functions and Scripts

Process priority	Due to their relatively low process priority, scripts run asynchronously relative to the measurement. Functions, on the other hand, are executed with high process priority and synchronously with the measurement. As a result, limitations arise for functions such as the `Sleep()` script function. In comparison to functions, scripts have a more extensive selection of predefined function groups such as additional file and script functions, Flash functions, and diagnostic and Data Mining functions.
Return values	The RETURN keyword can return a value of data type DOUBLE (a scalar, no data field/array, etc.) to the calling routine.

Note: Return values of scripts cannot be evaluated at present.

Debugging	Unlike scripts, functions cannot be debugged using breakpoints due to their high priority. Instead, you can output debug information in the Write window using `Write()`, `Writef()`, `Print()`, or `Printf()` (see section Debugging of Functions on page 47).
Subfunctions	In CANape **subfunctions of scripts** are referred to as subfunctions. Thus, subfunctions are not available for functions. The definition of the subfunction must be written before the main part of the script. Arguments for the subfunction can be specified optionally. If during a measurement a `Function2` is called from a different `Function1`, the term subfunction would also be used for `Function2` in common usage. In CANape, however, the term subfunction is not used in this context.
Program code	Scripts do not require a function header and body. The program code can simply be written directly to the editor and is then processed sequentially.
Format	Functions must be in the Functions Editor export format (`*.cne`) or in the ASCII text

format (`*.txt`). Scripts must be in the script format (`*.scr` or `*.cns`).

Memory location A function is saved to the `canape.ini` configuration file in your working directory. Script files instead are saved as a separate file in the working directory.

Format Scripts are files in the `*.scr` or `*.cns` script format. Functions, however, are embedded in the `canape.ini`.

2.5 Functions Editor

Tasks Global variables, functions, scripts, and Seed & Key algorithms can be created, edited and compiled in the Functions Editor of CANape.

Opening the Functions Editor

1. Click the icon in the toolbar.

   or

   Click **Functions and scripts** in the **Tools** menu.

   The Functions Editor opens.

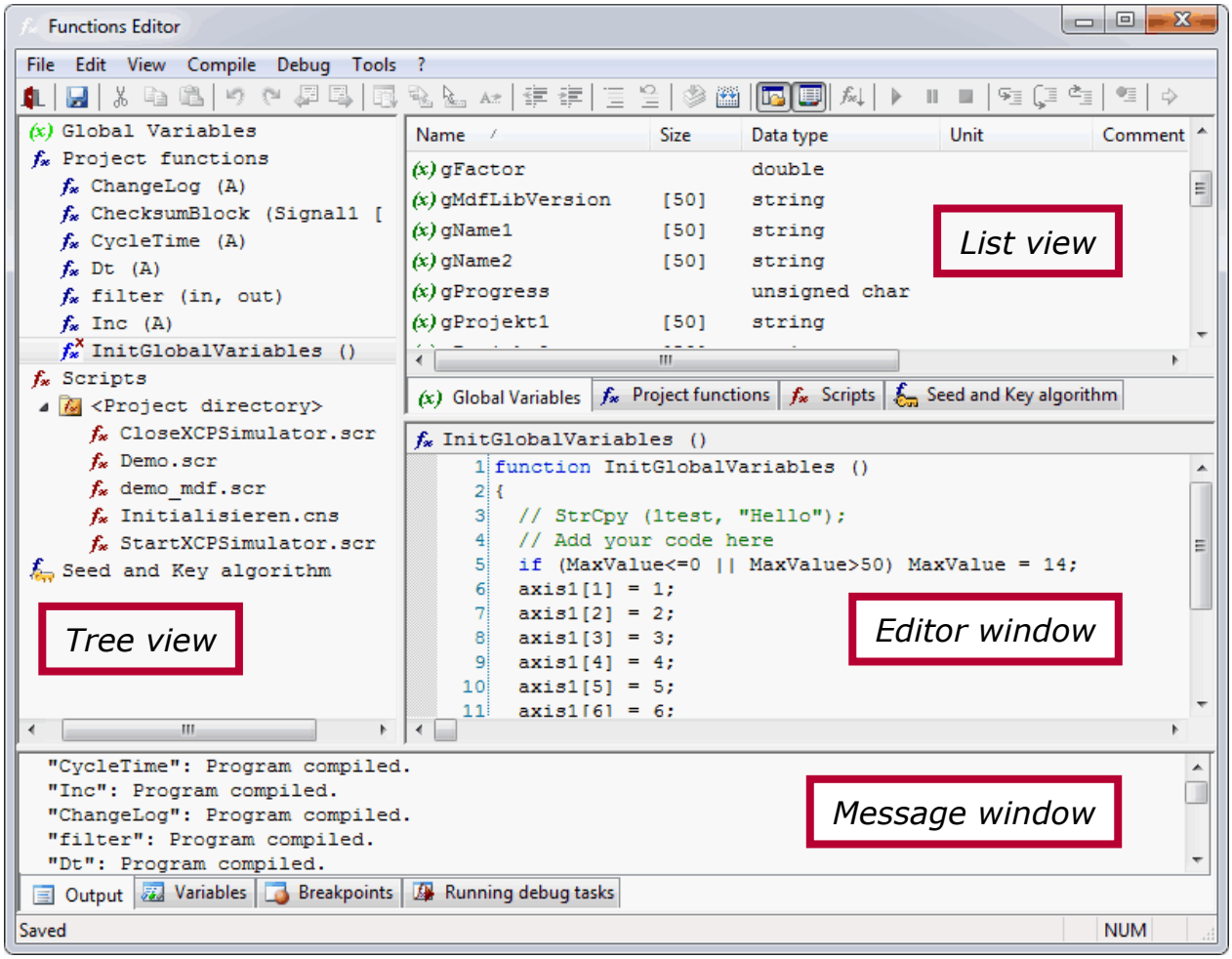


Figure 2-1: The Functions Editor in CANape

Areas of the Functions Editor

The Functions Editor is divided into the following areas:

- Tree view
- List view
- Editor window
- Message window

Tree view

The tree view lists all of the elements present.

- Global variables
- Project functions
- Scripts
- Seed & Key algorithms

List view

Various tabs are located at the lower edge of the list view. Depending on the selected tab, different information about the elements are displayed.

The information columns can be sorted by a click on the respective column header.

Message window

The **Output** tab of the Message window displays error and progress messages. These messages can be processed via the popup menu.

If debugging is activated, the variables, breakpoints, and debug tasks active in each case are also displayed here.

Note: The Functions Editor in CANape offers two types of help that enable functions and scripts to be created with little programming knowledge.

- **Context-sensitive help:** Place the cursor on a function definition in the popup menu of the Editor window and click <F1>. The help page for the corresponding function opens.
- **Automatic syntax completion:** When typing in the Editor window, a suggestion list of all available functions, objects, and global variables whose names begin with the typed-in letters is displayed. The suggestion is applied with **[Enter]** or the <Tab> key.

2.6 Additional Definitions

2.6.1 Variable Types

Agreement

Variables are placeholders for values of a certain type. Variables can be declared and defined in scripts and functions.

Definition (Type + Value)

In the case of **definition**, a value is assigned to the compiler. Undefined local variables always have the start value 0. The **definition** of variables thus serves to create the type of variables and to create data objects in the memory.

Declaration (Name + Type)

In the case of **declaration**, the respective variable is made known to the compiler. The **declaration** always consists of the name of an object and its type. As a result, the compiler knows which type it must connect a name to.

Global variables

Global variables are declared in the list or tree view of the Functions Editor and can be used in any function or script, for example.

Local variables

Local variables can be declared in the function or script header and can also be simultaneously defined. In/out parameters are declared in the parameter list of the function.

Lifespan

While global variables live as long as the program, local variables are only valid

during the block call.

Device variables — A device variable can be used to access a device-internal value.

2.6.1.1 Global Variables

Uses — Global variables are special data objects that can be used by all functions and scripts in the CANape configuration.

You can use **global variables** to pass information to other functions or scripts without the help of in/out parameters.

Definition — Global variables are defined in the Functions Editor. One-dimensional or multidimensional arrays as well as variables can be defined as global variables.

Memory location — Global variables are saved as project-related global variables in the `canape.ini` configuration file in your working directory.

Properties — A global variable consists of a name, comment, data type, and conversion rule, among other things.

Validity — Global variables are valid CANape-wide. They can be inserted into a Calibration window so that the values of the variables can be changed by the user during the measurement.

Overwriting — They are overwritten by local variables and function arguments having the same name.

Static behavior — Global variables are static. They retain their value between two measurements as well as after loading of the project. Their current value is saved in a parameter file when the configuration or project is saved and reloaded when loading the project.

Note: Note that values that were saved in a global variable during a prior measurement are still present at the start of a new measurement.

Integrating in CANape — For information on how to create a global variable, read section Creating a Global Variable on page 61.

In addition, for a description of how you can influence when the global variable is to be set to a defined value, see section Setting a Global Variable to a Defined Value on page 61.

2.6.1.2 Local Variables

Validity — Local variables are only valid within the respective function or script.

Overwriting — They overwrite global variables of the same name and are, in turn, overwritten by function arguments of the same name.

Declaration — In contrast to C up to and including CANape 11.0, local variables can be declared only at the start of a function or script and retain their validity throughout the function.

With CANape 12.0, a declaration within a function and within a block statement (e.g., `If` statement or `for` loop) is also possible.

Definition — Undefined local variables always have the start value 0.

Example: In the following example function, `c` is a local variable and `A` is an in/out parameter.

```
function Function1 (Var A)
{
  double Result;
  Result = A;
  if (A > 0)
  {
    int c = 2; // no syntax error CANape 12.0 or higher
    Result = c;
  }
  return Result;
}
```

Static or non-static behavior

Local variables can exhibit static or non-static behavior in CANape. Static behavior means that the variables retain their value between the respective function calls.

Below is an overview of the behavior of local variables within CANape:

Behavior

Location of variable	Static	Non-static
Script		X
Function in the script	X	
User-defined function	X	
User-defined function when called by a script		X
User-defined function when called by a user-defined function		X
User-defined function when called by a script function		X

Different start values

If a function is used, for example, in the Data Mining environment or as a virtual measurement file channel, CANape provides the option of assigning a different start value of a local variable to each instance of a function.

For information on how to set a local variable to a defined value, see section Setting a Local Variable to a Defined Value on page 63.

2.6.1.3 Device Variables

Access to device-internal value

A device variable can be used to access a device-internal value and to change it, when possible (database-dependent). The communication with the device takes place in the background without the user having to explicitly take care of this.

Changes

Changes to device variables act immediately on the corresponding device.

Inserting

The available device variables can be viewed in the Functions Editor via the popup menu and inserted into functions and scripts. For information on how to insert a device variable, see section Inserting a Device Variable on page 66.

2.6.2 Arguments and In/Out Parameters (of Functions)

Passing as reference

Arguments or in/out parameters are always passed to functions as reference (`by reference`) and can thus also be used for return of values (see also section What Are Functions on page 8).

They overwrite local and global variables having the same name and can only take on

the equivalent `var` and `double` values as the data type.

As a result, the following two functions are identical:

Function A

```
function Add (var A, var B)
{
  return A+B;
}
```

Function B

```
function Add (double A, double B)
{
  return A+B;
}
```

Parameter types Two types of parameters are possible: Scalar and array parameters.

Scalar parameters Any scalar variable, an individual array element, or a constant may be used as an argument for scalar parameters.

Array parameters To define array parameters, **[ ]** must be placed directly after the parameter name. Any array variable, constant strings, or global/database variables of type `String` may be used as an argument for array parameters.

Optionally, the keyword `var` or `double` may be used as the data type prefix. There is no difference in the case of words. Other prefix types are not supported.

2.6.3 Comments

Documentation Comments are used to document a section of code within a program. They are marked by `//` for a single-line comment or are placed between `/* ... */` for multi-line comments.

2.6.4 Taking Upper and Lower Case Into Account

Variable names Variable names in functions and scripts, global variables, and variables referenced from a database in the CANape Functions Editor are case sensitive.

Device and function names Device names and internal function names are not case sensitive.

2.6.5 Predefined Function Groups and Code Blocks of CANape

Support CANape provides you with code blocks and various function groups to support you when drafting your functions.

Code blocks Right click in the Editor window to open the popup menu. Select the desired code block, e.g., a control structure. Become familiar step-by-step with the C-like syntax of CANape's own programming language.

Function groups Select between the various CASL functions predefined by CANape. These are organized into the following function groups:

Precompiler directives Precompiler directives for conditional compilation. On the basis of the precompiler directives, the precompiler can decide based on expressions and symbols which part of the source code can be inserted and compiled and which can be removed,

	accordingly.
Trigonometric functions	Trigonometric functions can be used to calculate relationships between angle and height-width ratio.
Exponential functions	Exponential functions can be used to calculate logarithms and powers and their components.
Miscellaneous functions	**Miscellaneous functions** include all functions not assigned to one of the other groups.
Access attributes	By means of the function group of the access attributes, you have access to features of various data objects, such as measurement values, axes, and maps/characteristic curves. Features of data objects can be color, quantity, status, address, etc.
Program functions	General accesses to CANape are possible using program functions. The following subgroups provide more specific accesses: > User input: Shows various dialogs > Output: Formats outputs of a string, describes the progress display > Control: Controls the various CANape windows > Panel: Sets stop watches and accesses control and display elements. > Configuration: Involves configurations and partial configurations
Device functions	You can use device functions to access devices directly as well as to access their databases, drivers, status, etc.
Measurement	The functions of the **Measurement** function group enable access to a wide range of actions that concern measuring. In the **Recorder** and **Trigger** subgroups, you will find functions that access the recorders and their triggering.
Calibration	The functions of the **Calibration** function group can be used to access parameter sets and to change the calibration mode of devices. In addition, you receive read- and write-access to the device memory and access to the various values (physical, raw, or string values) of objects. The block-wise modification of a device can also be controlled. In the **Datasets** subgroup, you will find the functions for handling datasets.
Evaluation	The functions of the **Evaluation** function group enable access to the global measurement and differential cursor and the time offset as well as to comments and information on trigger events. In addition, portions of an integer signal can be extracted or measurement values of a signal can be determined as a string.
Database	The functions of the **Database** function group enable access to a unit of measurement or names of database variables.
Flash functions	Flash functions can be used to load or unload programs and parameter set files. A parameter set can also be copied to a binary file.
Diagnostic functions	Diagnostic functions ensure access to service parameters and messages (request and response to the ECU). In the Device functions subgroup, you will find functions that can access devices with a diagnostics-capable driver. In addition, two functions are provided that permit implementation of the Seed & Key method.
Script functions	Script functions are used to selectively call, check, and control scripts. Error codes and error text can be output as appropriate or command line arguments can be determined. In addition, there is a script function for temporary loading/unloading of function DLLs.
Data Mining functions	Data Mining functions can start, stop, monitor, and compare Data Mining analyses (and their methods).
String functions	String functions can be used to edit and process strings or the contents of certain

	memory locations, in general. For example, there are functions for copying, comparing, or attaching.
System and time functions	System and time functions can return the different times of the system. In addition, start values of other functions can be specified and their processes can be waited for.
File functions	File functions can be used to handle files. For example, files can be searched for, deleted, renamed, opened, and read. This also applies to the file-specific subgroups HEX, MDF, PAR, and XML.
Obsolete functions	The functions of this group are obsolete and should not be used anymore. However, they are still supported by the Functions Editor.

Note: You will find more detailed information on the various function groups and code blocks in the Help. Select the desired function or code block in the popup menu of the Editor window and press the <F1> key.

2.7 General System Limits

Global variables	The maximum number of global variables is limited by the system memory.
Local variables	The maximum number of local variables is 16383.
Subfunctions	The maximum number of subfunctions that can be called by a script is 10,000.
Project functions	The maximum number of project functions is limited by the system memory.
Data stack	The size of the data stack is 1024 entries. Its data type is `double`.
Call stack	The maximum call depth is 64.
Argument stack	Arguments for subfunctions or user function calls are stored in the argument stack. The maximum size of an argument stack is 10,000 entries.
Line length	The line length of the source code is unlimited.
Statements	Each nesting in the source code (the segment between **{** and **}**) can contain about 9990 statements.

3 Syntax

This chapter contains the following information:

3.1 Differences Between C Programming and CASL

Differences

The CANape scripting language CASL is very similar to the C programming language. However, it differs in the following aspects:

- CANape uses only double values for internal calculations.
- CANape does not use pointers.

Additional feature

The CANape scripting language CASL has a series of additional features compared to the C programming language.

- A missing result type is interpreted as `void`.
- Arrays of any dimension and size may be passed.
- An empty parameter list is allowed like in C++.
- Overloading of functions (i.e., multiple functions with the same name but with different parameter lists) is possible like in C++.
- A parameter check is made like in C++.

3.2 Numbers and Characters

3.2.1 Data Types and Value Ranges

Data types

CANape provides the following data types for use in functions and scripts:

Data type	Value range	Size/bytes
char	-128 to 127	1
unsigned char	0 to 255	1
byte	0 to 255	1
short	-32 768 to 32 767	2
unsigned short	0 to 65535	2
int	-32 768 to 32 767	2
unsigned int	0 to 65 535	2
long	-2 147 483 648 to 2 147 483 647	4
unsigned long	0 to 4 294 967 295	4
float	-3.4·1038 to 3.4·1038 (IEEE 32 Bit)	4
double	-1.7·10308 to 1.7·10308 (IEEE 64 Bit)	8

Creating an array from a data type

Arrays can also be created from any data type. The individual array elements in one- and two-dimensional arrays are accessed using the **[<Index>]** access operator.

Example: Array of data type `int`

```
int Array[2];//Declaration
Array[0]=1;//Definition
Array[1]=3;//Definition
```

3.2.2 Parameter Types for Predefined Functions

In/out parameters

Various types of parameters are available for passing data in the syntax of the functions. These are called parameter types in CANape.

Type	Direction	Description
ARRAYREF	[in/out]	Reference to an array variable (local, global, device, parameter)
BUFFER	[out]	String for the assignment of an output via a reference (local, global, device, parameter)
OPT_INLIST	[in]	Optional list with input arguments
OPT_OUTLIST	[out]	Optional list with output arguments
REFERENCE	[in/out]	The compiler accepts scalar or array variables that are defined via a reference.
SIGNALREF	[in/out]	Signal object defined via a reference, e.g., `<DeviceName>.<SignalName>`.  **Example:** `sizeof(CCPsim.channel1)` For functions requiring a value or status of the signal at a particular time, `<file>.<signal>.mbuffer[index]` can be used, e.g., for `time()` or `phy()`. Without `mbuffer`, for device signals the current state will be used; for file signals the output is undefined.
STRING	[in]	String defined via a value (const, expression, local, global, device, parameter).
VAL	[in]	Scalar value defined via a value (const, expression, local, global, device, parameter).
VALREF	[out]	Reference to a scalar variable (local, global, device, parameter)

Function DLLs

The following are available exclusively for function DLLs:

Type	Direction	Description
REFERENCE	[in/out]	Scalar or array
VALUE	[in]	Scalar or array

3.2.3 Constants

Use in scripts

When constants are used in scripts, the following must be observed for interpretation of hexadecimal constants:

Hexadecimal constants

Hexadecimal constants are interpreted as `signed` by default. In order for a hexadecimal constant to be interpreted as `unsigned`, a `u` or `U` must be added.

Decimal constants

Decimal constants are interpreted correctly.

Note: For more information on the behavior of decimal and hexadecimal constants assigned to a variable, refer to the Help.

Single ASCII characters

In scripts and functions no single ASCII character such as `'A'` can be used. The single quotes are already reserved for accessing signals whose names are not ASAM-compliant.

To assign a single character to a variable, the equivalent hexadecimal notation must be used instead.

Example:

Invalid for constants:

```
char letter = 'A';
```

Use hexadecimal equivalent instaed:

```
char letter = 0x41;
```

3.2.4 Arrays

Collection of data elements

An array (also referred to as a field) is a collection of data elements that all have the same name. The value within the square brackets after the array name shows either the array size or an index value in order to specify the data. The array always begins with index 0. The number of square brackets indicates the dimension of the array.

Passing

Arrays are always passed as reference (`by reference`).

Language usage

A one-dimensional array or a characteristic curve corresponds to a vector.

A two-dimensional or multidimensional array or a map corresponds to a matrix.

Example 1: `int data[3] = {10, 20, 40};`

yields: data[0] = 10, data[1] = 20, data[2] = 40

Example 2: The global array variables `g_count[6][4]` and `g_time[6][4]` are initialized

```
int i=0;
int i=0;
for(k=0;k<=5;k++)
{
  for(i=0;i<=3;i++)
  {
    g_count[k][i]=0;
    g_time[k][i]=0;
  }
}
```

3.2.5 Strings

Character string A string consists of a series of one or more sequential characters that are placed inside double quotation marks.

Structure The last element in a string is a zero character (`\0`), which indicates the end of the string. As a result, the array size is always the number of characters plus one.

Passing Strings are always passed as reference (`by reference`).

Example:
```
char string[12] = "Hello World!";
```

3.2.6 Placeholders

Function `write()` The `write()` function outputs a string with formatting statements same as C command `printf()`. The string is output in lines in the Write window of CANape.

Formatting statement The formatting statements use various placeholders

Form The placeholders have the form:

```
%[Flags][Field Width][.Accuracy]Type
```

Where Flags, Field Width, and Accuracy can be specified optionally.

Placeholders The following types of placeholders are used by the formatting statements.

Type

Type	Output
x, X	Case-sensitive HEX numbers of data type unsigned long (4 bytes)
h, H	HEX numbers of type unsigned short (2 bytes)
b, B	HEX numbers of type unsigned char (1 byte)
o	Octal numbers
e, E	Floating-point numbers in exponential notation
g, G	Floating-point numbers or exponential notation, depending on which is shorter
f	Floating-point numbers
u	Whole positive decimal numbers
d, i	Whole decimal numbers
c	A single character (character)
s	Strings

Examples:

```
Write("%h", 0xffff);    // Output: ffff
Write("%o", 10);        // Output: 12
Write("%e", 12.3456);   // Output: 1.234560e+001
Write("c", 0x41);       // Output: A
Write("%s", "Hello");   // Output: Hello
```

Flags

Any of the placeholders indicated above can be modified optionally with a flag. The following flags can be used:

Flag	Description
+	Numbers are output right-justified with sign.
-	Numbers are output left-justified. Negative numbers are given a sign.
' ' (blank character)	Positive numbers are output with leading blank characters.
0	The field before the number is filled with 0.
#	Type g: Forces a floating point Type x: Receives an 0x before a HEX number Type o: Places a 0 in front of the octal number. Trailing zeros after the decimal point are not shown.

Examples:

```
Write("%+d", 1234);      //Output: +1234
Write("Wert:% d", 20);   //Output: Value: 20
Write("%#g", 03);        //Output: 3.00000
Write("%#x", 03);        //Output: 0x03
Write("%#o", 5.1);       //Output: 05
```

Field width

The field width specifies the length of the output field.

Example: Field width 12

```
long number = 12;
Write("Number:%4d", number);

//Output:
Number: 12
```

Accuracy

The accuracy specifies the number of places after the decimal point for floating-point numbers (e.g., of type `f`) and rounds the value.

Example: Accuracy of 2 places after the decimal point

```
float number = 12.3456;
Write("Number:%.2f", number);

//Output:
Number: 12.35
```

Note: Placeholders can also be used in any other user-defined function that works with strings (e.g., `Sprint()`, `fprint()`).

3.3 Operators

Processing of data objects

The Functions Editor in CANape uses various operators for processing data objects. Data objects that are called by operators are referred to as operands.

Objective

Operators and operands are combined in order to calculate new values. The value of an expression is often also referred to as a return value.

Overview

The following operators are available:

- Arithmetic operators
- Relational operators
- Binary operators
- Logical operators

Arithmetic operators

Operator	Designation	Syntax
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulo, remainder of integer division	a % b

Relational operators

Operator	Designation	Syntax
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b
==	Equal to	a == b
!=	Unequal to	a != b

If the declaration of the syntax is confirmed, 1 (`TRUE`) is returned, otherwise 0 (`FALSE`).

Binary operators

Operator	Designation	Syntax
&	Bitwise AND operation	a & b
|	Bitwise OR operation	a | b
~	Bitwise complement	~a
^	Bitwise Exclusive OR operation	a ^ b
<<	Bitwise Shift Left	a << x
>>	Bitwise Shift Right	a >> x

Logical operators

Operator	Designation	Syntax	Description
!	Logical NOT	!a	If a is true (`TRUE`), the operation returns false (`FALSE`).
&&	Logical AND	a && b	The operation returns only `TRUE` if a

Logical operators

Operator	Designation	Syntax	Description
			and b are TRUE; otherwise FALSE.
||	Logical OR	a || b	The operation returns only TRUE if a or b is TRUE; otherwise FALSE.

Reference: Examples for the various operators can be found in the Help. Select the desired operator in the popup menu of the Editor window and press the <F1> key.

3.4 Control Structures (Statements)

Influencing the control flow

A program consists of a wide range of statements that are generally processed one after the other. The processing sequence of statements is referred to as the control flow. A statement that influences this control flow is therefore also called a control structure.

The Functions Editor of CANape uses various control structures similarly as in the C programming language.

for

The `for` loop is a conditional loop. It executes the statement(s) until the condition is true (unequal to 0).

```
for (Initialization; Condition; Reinitialization)
{..Statement(s)..}
```

while

The `while` loop is a conditional loop. It executes the statement(s) until the condition is not true (equal to 0). If the condition is not true (equal to 0) from the start, the `while` loop, unlike the `for` loop, is not executed at all.

```
while (Condition)
{..Statement(s)..}
```

do-while

The `do-while` loop behaves like the `while` loop except that this loop is run through at least once.

```
do {..Statement(s)..} while (Condition);
```

if

The `if` statement is a conditional branch in the program. If the condition is true (unequal to 0), the statements within the `if` statement block are executed.

```
if (Condition) {..Statement(s)..}
```

if-else

The `if-else` statement is a conditional branch in the program. If the condition is true (unequal to 0), the statements within the `if` statement block are executed. Otherwise the statements in the `else` statement block are executed.

```
if (Condition) {..Statement(s)..}
else {..Statement(s)..}
```

if-else if

The `if-else if` statement is a chain of conditional branches in the program. The statements within the `if` statement block whose condition is true (unequal to 0) are executed. Otherwise the statements in the `else` statement block are executed. If no alternative statement block (`else`) is needed at the end of the branch chain, the last

`else` statement can be omitted.

```
if (Condition) {..Statement(s)..}
else if (Condition) {..Statement(s)..}
...
else {..Statement(s)..}
```

switch

The `switch` statement is used in order to run through multiple branches in a program. In contrast to the `if-else if` statement, the `switch` statement is simpler and more straightforward.

The structure of a `switch` statement consists of a series of `case` labels and an optional `default` label. Two constant expressions in `case` labels must not take on the same value during evaluation.

The constant expression of the `case` labels is checked for equivalence with the expression in `switch`. If the expression in a `case` label matches a constant, the statements starting from this `case` will be executed. The `default` statement is executed if the expression does not match any of the constants.

```
switch (Expression)
{
  case Constant1:
    ..Statement(s)..
    break;
  case Constant2:
    ..Statement(s)..
    break;
  ...
  case ConstantX:
    ..Statement(s)..
    break;
  default:
    ..Statement(s)..
}
```

break

The `break` command aborts a loop (`for` or `while`) or a `switch` statement immediately.

```
break;
```

?:

The `?:` statement is a conditional operator, a short form of `if-else`. If the condition is true (unequal to 0), `Expression1` is returned, otherwise `Expression2` is returned.

```
(Condition) ? Expression1 : Expression2
```

continue

The `continue` command skips the remaining statements of a loop and returns to the start of the loop. In the case of a `for` or `while` loop, the next loop pass-through is executed immediately. In the case of a `do-while` loop, the abort condition is first tested and then the next loop pass-through is started, if necessary.

```
continue;
```

return

The keyword can be used in two ways:

It can be used to end a user-defined function, a subfunction, or a script and to return the control back to the calling routine. If no value is to be passed to the calling routine, `return` is used without a subsequent expression (`return;`).
However, the `return` keyword can also return a value of data type `double` to the calling routine. Return values of scripts cannot be evaluated at present.

```
return [expression];
```

cancel

If this command is used in a function, the function is canceled without a return value. The control is passed back to the calling routine.

If the `cancel` command is used in a subfunction or script, the script is terminated.

```
cancel;
```

Note: A result value is not available for a function call that contains the `cancel` command, i.e., no value is saved in the measurement file or shown in the Display window. As a result, the command can be used, e.g., for data reduction. Or, in the case of an ECU stimulation, the STIM message is not sent and thus the stimulated variable(s) are not downloaded.

function

The keyword `function` defines a user-defined function. In addition, it can be used to define a subfunction within a script.

```
function <function name> (Parameter)
```

Reference: Examples and detailed descriptions for the various control structures can be found in the Help. Select the desired control structure in the popup menu of the Editor window and press the <F1> key.

4 Functions, Scripts, and Variables in CANape

This chapter contains the following information:

4.1 Functions

4.1.1 Writing the Functions

Functions Editor

A function can be written in the Functions Editor of CANape.

Creating functions:

1. Open the Functions Editor using the icon (see also section Functions Editor on page 12).
2. Click with the right mouse button on **Project functions|New** in the tree view of the Functions Editor.

   The **Properties** dialog opens.
3. Name the function as desired and add a comment, if necessary.
4. Confirm your inputs with **[OK]**.
5. Write your new function in the Editor window.

Predefined function groups and code blocks

CANape provides you with code blocks and various predefined CASL functions (organized in function groups) to support you when drafting. Right click in the Editor window to open the popup menu. For additional details, see section Predefined Function Groups and Code Blocks of CANape on page 16.

IntelliSense

Get acquainted with CANape's IntelliSense! For example, type the starting letter(s) of a predefined CASL function. Select the function from the automatically displayed IntelliSense list. Use <Tab> or <Enter> or double-click with the left mouse button to transfer the selected function to the Editor window.

Note the information that CANape additionally displays about the function. For example, the data type of the return value and the number of in/out parameters are indicated here.

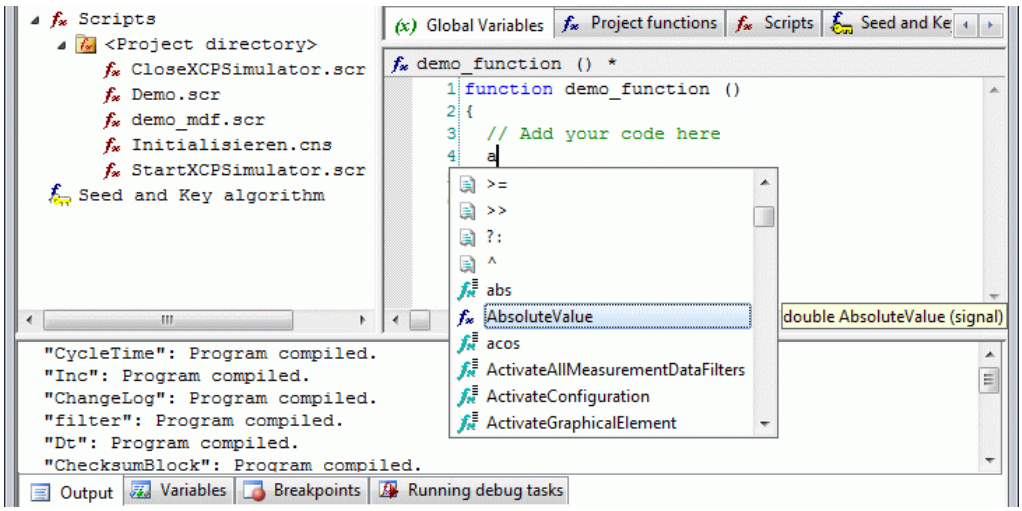


Figure 4-1: IntelliSense list with additional information when a letter is typed

Saving and compiling functions:

6. When you finish writing your function, compile the function using the icon or the **Compile|Compile** menu item.
7. Note the messages output by CANape in the Message window (**Output** page) at the bottom of the screen.

   If the message "The program is compiled" appears, the syntax of your function is correct. The red X that marks your function in the tree view disappears.

   If an error message is displayed, try to eliminate the error with the help of the message. The red X that marks your function in the tree view is retained.
8. Save your function with or **File|Save**.

   The function is saved to the `canape.ini` configuration file in your working directory. The asterisk that marks your function in the tree view disappears. Functions that are not compiled can also be saved in this way.
9. Close the Functions Editor using the icon or the **File|Close** menu.

Note: Functions such as the measurement itself have a very high process priority in CANape. Therefore, you should avoid long-lasting program code because it will negatively affect the performance of your measurements, e.g., measurements might be lost. Infinite loops tie up the entire measurement and can only be interrupted by actuating the <Esc> key for 3 seconds.

4.1.2 Saving and Forwarding Functions (Exporting/Importing)

Saving	Functions are saved in the `canape.ini` configuration file in your working directory.
Exporting/ importing	Functions can be exported from and imported to the Functions Editor.
Formats	Functions must be present in the Functions Editor export format (`*.cne`) or as an ASCII text file (`*.txt`) in order to be imported. Functions present in the Functions Editor export format (`*.cne`), as an ASCII test file (`*.txt`), or in HTML format (`*.html` or `*.htm`) are exported.

Importing functions:

1. Open the Functions Editor using the icon.
2. Select **File|Import|Function**.

   An Explorer window opens.
3. Select the function to be imported and confirm your selection with a double click or with **[Open]**.
4. Functions Editor export format (`*.cne`): In the next prompt, select your function and confirm your selection with **[OK]**.

   or

   ASCII text file (`*.txt`): Confirm the properties of the function.

   The imported function is displayed as an unsaved function (with asterisk) in the tree view.
5. Save your function with or **File|Save**.

Exporting functions:

1. Open the Functions Editor using the icon.
2. Select the function to be exported in the tree view.
3. In the popup menu, select **Export**.

   or

   Activate your mouse in the Editor window, **Project functions** page, and click **Edit|Export**.
4. Select the desired export format in the dialog that appears.
5. In the Explorer window, select the desired memory location and confirm your selection with **[Save]** and, if necessary, with **[OK]**.

   User-defined functions and their links are also saved in the respective configuration (`*.cna file`) of the project. The functions are also stored in the associated CANape INI file.

Forwarding linked functions

In order, for example to make a virtual signal configured offline and its linked function accessible to other users, you can create a new configuration. In this configuration, you can use the desired function as a virtual measurement file channel in a Graphic window (see section Using a Function in a Graphic Window as a Virtual Measurement File Channel on page 33).

Importing a configuration

You can then import this configuration elsewhere.

In CANape Versions before 11.0, you use **File|Load configuration partially** exclusively for this.

In CANape Versions 11.0 and later, you can also add the new configuration to the current configuration. To do so, select **File|Configuration|Configuration manager** and select **[Add] Existing configuration** in the Configuration manager.

You can deactivate the added configuration at any time in the Configuration manager. Partial loading allows only parts of a large configuration to be added to the current configuration.

If you are using device values (e.g., signals) as input values for your function, you must ensure that the utilized device is present in the current configuration.

Note: If there is only one device in the device configuration, its database is opened automatically for selection. If there are multiple devices, you are first prompted via a dialog to select an existing device so that its database will be used for the selection of signals. In this way, cross-device signals can be assigned to the variables of a function.

4.1.3 Commissioning or Instantiating Functions

Function definitions

User-defined functions, i.e., library functions and project functions, are displayed under **Function definitions** in the tree view of the Symbol Explorer.

Project functions are also displayed in the tree view of the Functions Editor. The library functions can also be imported into the Functions Editor (see section Global Function Library on page 45).

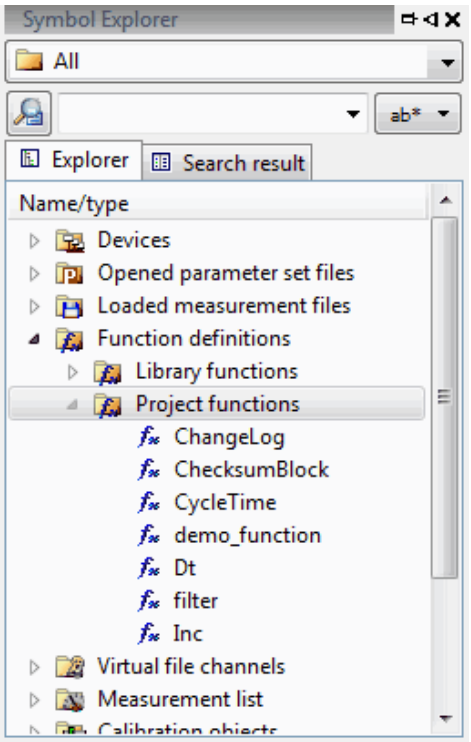


Figure 4-2: Project functions in the Symbol Explorer

Instantiating a function

In order to commission a function, it must be instantiated. This can be done in the following ways:

- Use a function in a Graphic window as a virtual measurement file channel (for details see section Using a Function in a Graphic Window as a Virtual Measurement File Channel on page 33).
- Instantiate a function in the measurement signal list (for details see section Using a Function During a Measurement on page 35).
- Execute a function when a CAN raw message is received (for details see section Executing Functions When a CAN Signal Is Received on page 38).
- Call a function from another function (for details see section Calling a Function From Another Function on page 41).
- Call a function from a script (for details see section Creating a Subfunction in a Script and Calling it on page 42).
- Use a function in Data Mining as a virtual MDF signal (for details see section Using a Function in Data Mining as a Virtual MDF Signal on page 42).

Use of functions

Functions can be inserted in the measurement signal list or used as a function definition for virtual MDF signals (e.g., for Data Mining). They can also be called directly from another user-defined function or a user-defined script. However, this method is not recommended when high performance is required.

4.1.3.1 Using a Function in a Graphic Window as a Virtual Measurement File Channel

Modification of measurement signal via a function can be graphically displayed

In order to graphically display modifications to an original measured measurement signal, you can make these modifications to the measurement signal in a function and have the result of the function displayed in a Graphic window as a virtual measurement file channel.

The function is then applied automatically to every sample value of the measurement signal.

Example: You can add an offset to the original signal, scale the original signal differently, or perform calculations from two measurement signals.

It is possible to calculate the power from input signals `Current` and `Voltage`. In another step, the `Integral()` **library function** can be used to integrate the power for the energy needed over the time period.

Creating a virtual measurement file channel:

1. Using drag & drop, move the desired function into the Graphic window.

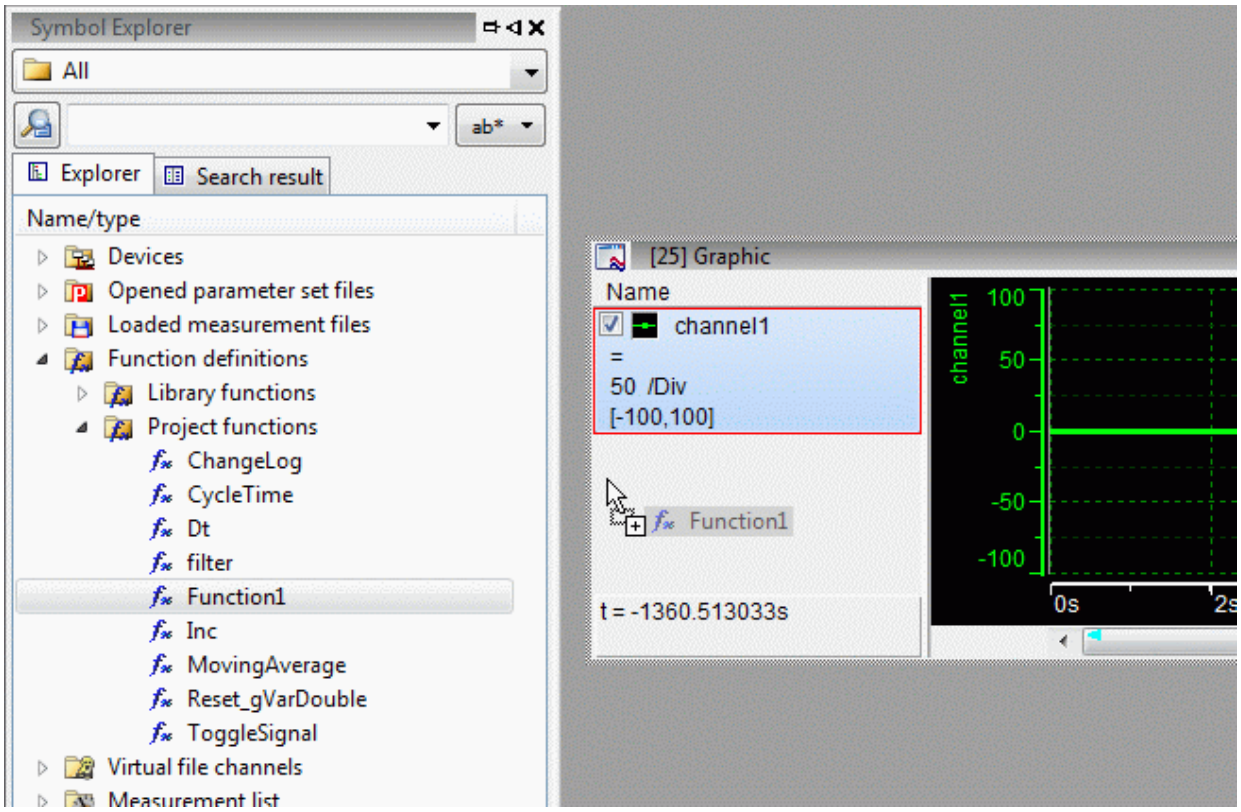


Figure 4-3: Moving the function from the Symbol Explorer to a Graphic window

2. Select **Virtual measurement file channel** in the menu that pops up.
3. Assign your input value(s) to the function. To do so, move the desired input value onto the desired input parameter of the function using drag & drop.

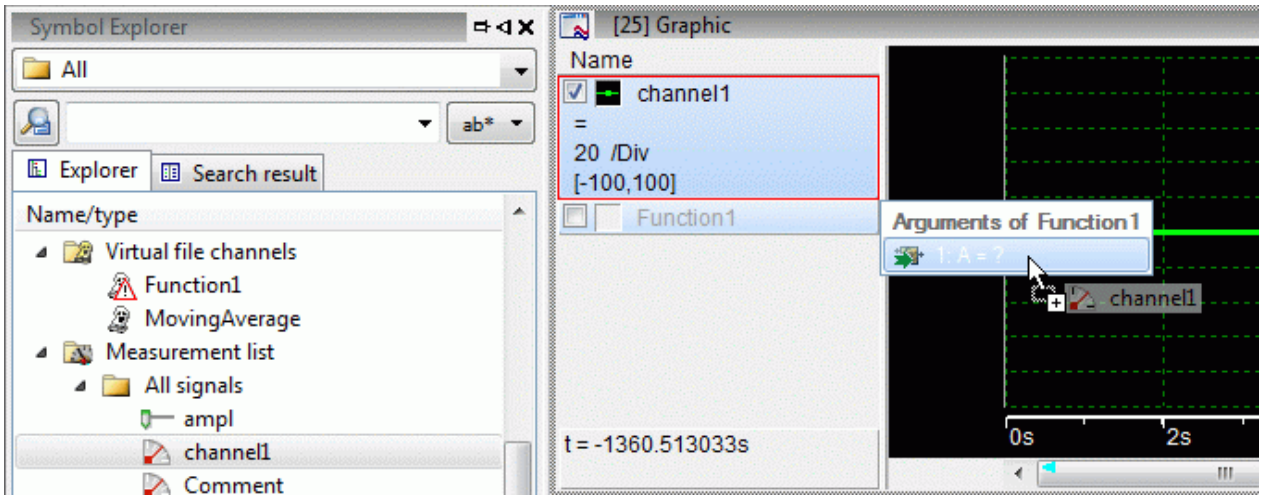


Figure 4-4: Assigning the input value to the function

1. As an alternative to the above-indicated procedure, you can right click in the Graphic window and select **Insert|New virtual measurement file channel from|Function**.
2. Complete the subsequent dialog as described in section Using a Function During a Measurement on page 35.

4.1.3.2 Using a Function During a Measurement

Objective

In order to use a function during a measurement, it must be created in the measurement list of the measurement configuration. It is then also called a measurement function.

Instantiating a function in the measurement configuration/creating a measurement function:

1. Open the measurement configuration using <F4>, the icon, or the **Measurement|Measurement configuration** menu item.
2. Chose the section **Measurement signals**.
3. Click the icon or select the **Edit|Insert function** menu.

   The **Function** dialog opens.

   In the **Function** dialog, you can select an existing function or create a new function.

Example: To change a calibration value cyclically, you could use the following `ToggleSignal` example function with the following function definition:

```
function ToggleSignal (Signal)
{
  if(Signal==199)
  {
    Signal=0;
  }
  else
  {
    Signal=199;
  }
}
```

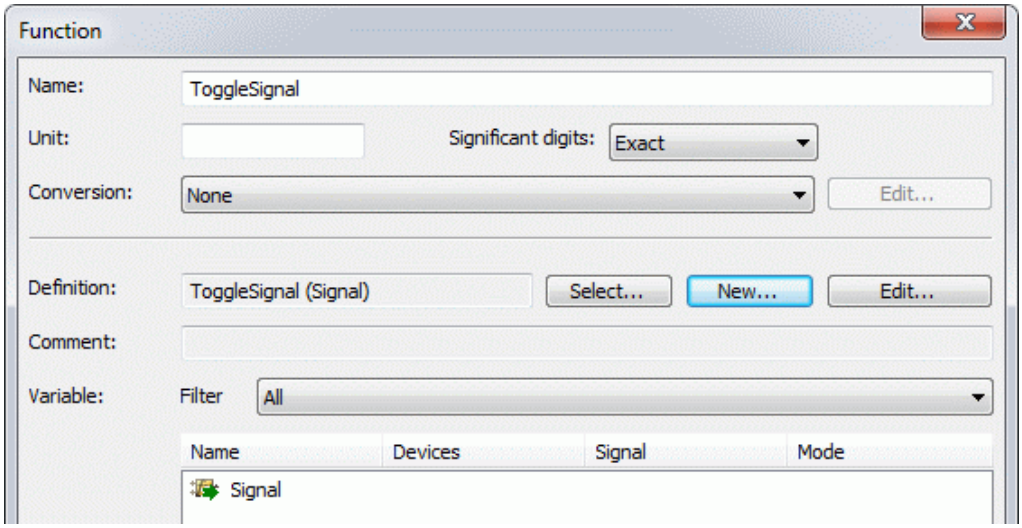
4. In the **Function** dialog, use **[New]** to create a new function

   or

   use **[Select]** to select an existing function.

   The menu guidance for creating a new function corresponds to the description in section Writing the Functions on page 30 starting from step 3.

*Figure 4-5: **Function** dialog after creation of a new function*



Linking the variables

Here, you can also assign any needed input values to the variables of your function. In so doing, you choose between global variables and real or simulated (e.g., `XCPsim`) measurement signals.

You can use the example function, for example, to influence the calibration value `ampl` in the XCP demo.

5. Select the corresponding `Signal` variable.
6. Click [Link manually].

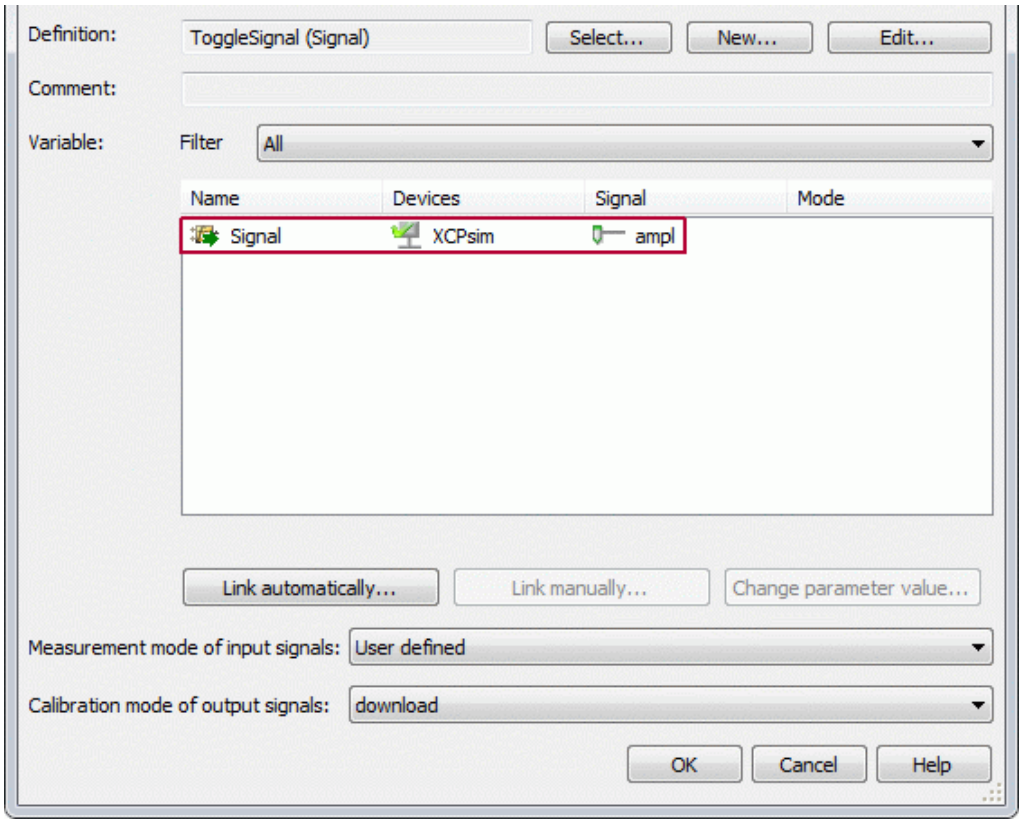
   The **Link Manually** dialog opens.
7. Select the `XCPsim` device and confirm your selection with a double click or with **[OK]**.

   The Database Selection opens.
8. Select the calibration parameter `ampl`.

9. Click **Apply** and close the database selection using the icon.

   The link appears in the **Function** dialog as follows:

Figure 4-6: Variable with assigned signal



10. Close the **Function** dialog with **[OK]**.

    The function will be inserted in the measurement signal list.

    You can reopen the closed dialog at any time with the **Edit function** popup menu command and thus change the signal linking, for example.

11. Select the **measurement mode** of the function, e.g., `cyclic` at a **rate** of 100.

    The calibration value `ampl` of the example function `ToggleSignal` is then calibrated regularly every 100 ms.

Note: If your function is to reset the global variables at the start of the measurement, use measuring mode **on event 'MeasurementStart'**.

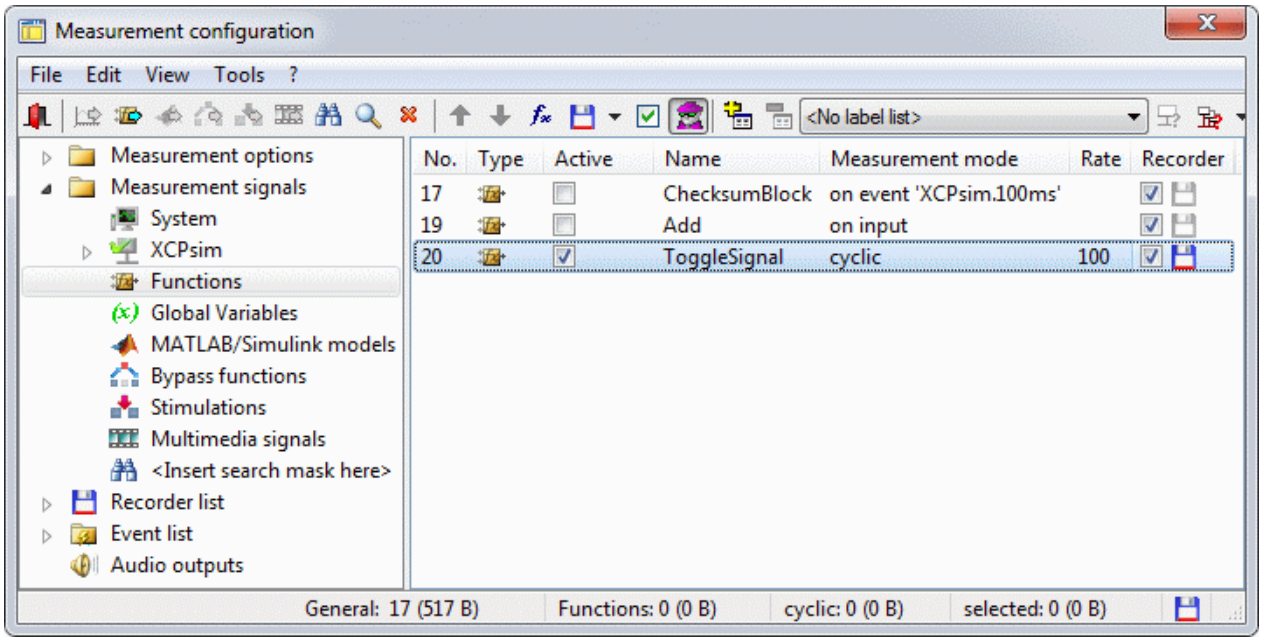


Figure 4-7: Measurement signal list with example function and its measuring mode settings

Note: If there is only one device in the device configuration, its database is opened automatically for selection. If there are multiple devices, you are first prompted via a dialog to select an existing device so that its database will be used for the selection of signals. In this way, cross-device signals can be assigned to the variables of a function.

4.1.3.3 Executing Functions When a CAN Signal Is Received

Objective

If you want to call your function, e.g., when a signal measured via CAN is received, you must ensure that a CAN device is created.

Note:

There are three different options for creating a device:

- Via the menu bar with **Device|New**
- Via the device configuration
- By importing from an existing database and creating (**Device|New from|database**)

More details are available in the CANape Help.

In order to have your function executed when a CAN signal arrives, you can follow practically the same procedure as in section Using a Function During a Measurement on page 35.

1. Open the measurement configuration using the icon or the **Measurement|Measurement Configuration** menu item.
2. Click the icon or select **Edit|Add Function** in the menu.

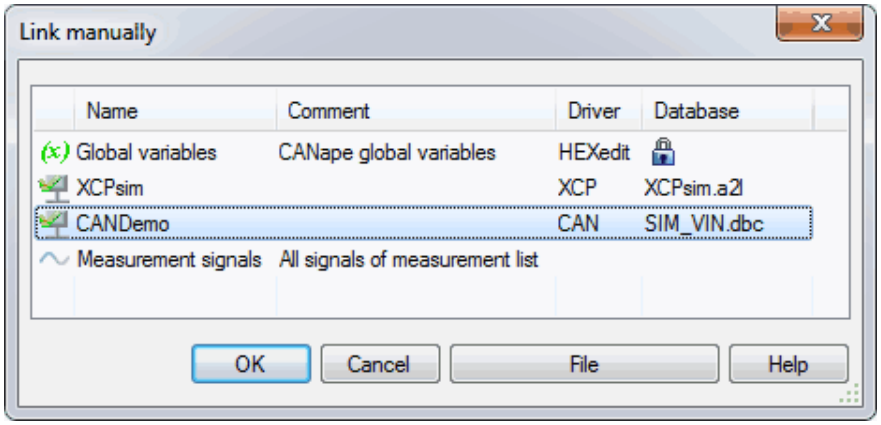
   The **Function** dialog opens.

   In the **Function** dialog, you can select an existing function or create a new function.

3. For example, use the CANape library function `AbsoluteValue()` in order to rectify a sine-wave signal.
4. Name the function, e.g., `Rectify`.
5. Select the `Signal` variable.
6. Click **[Link Manually]**.

   The **Link Manually** dialog opens.
7. To use a CAN signal as an input signal, select the already created device (`CANDemo`).

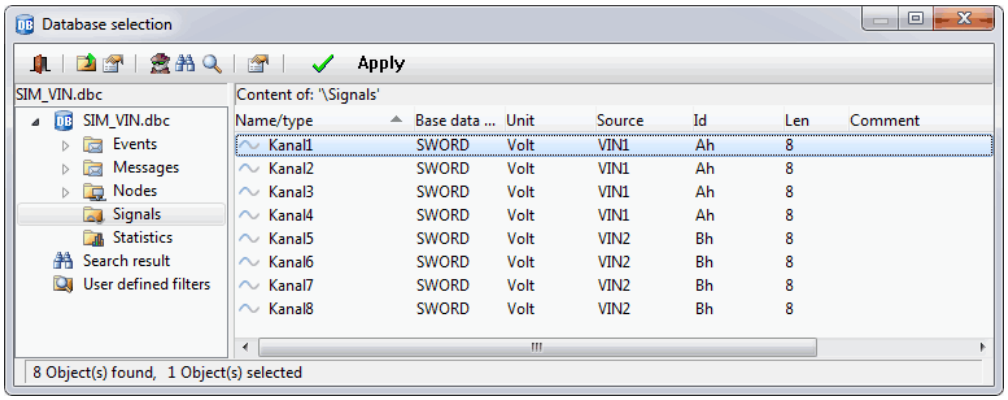
Figure 4-8: Manually linking a signal



8. Confirm your selection with a double click or **[OK]**.

   The Database selection opens.
9. Click the **Signals** folder to display its content.
10. From the folder, select, e.g., the `Channel1` signal.

Figure 4-9: Transferring a signal from the database



11. Click **Apply** and close the database selection using the icon.

    The link is shown in the **Function** dialog.
12. Close the **Function** dialog with **[OK]**.

    The function will be inserted in the measurement signal list.

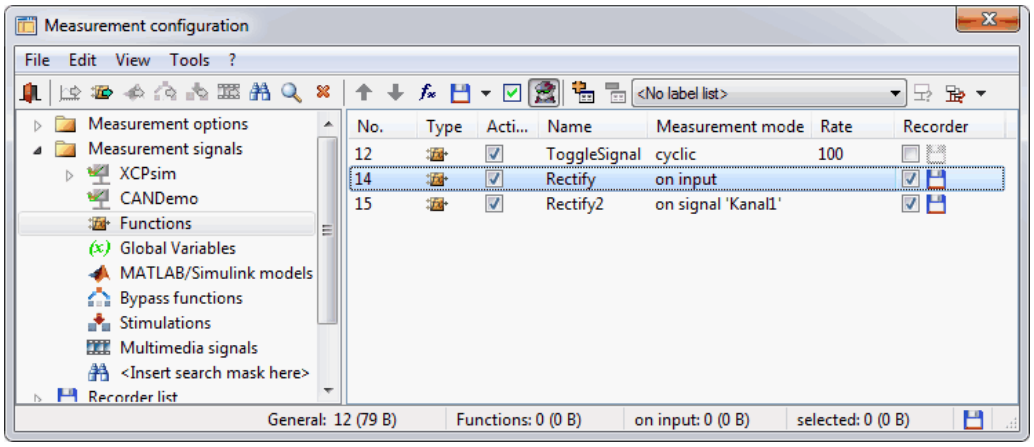


Figure 4-10: Functions in the measurement signal list with different measuring modes

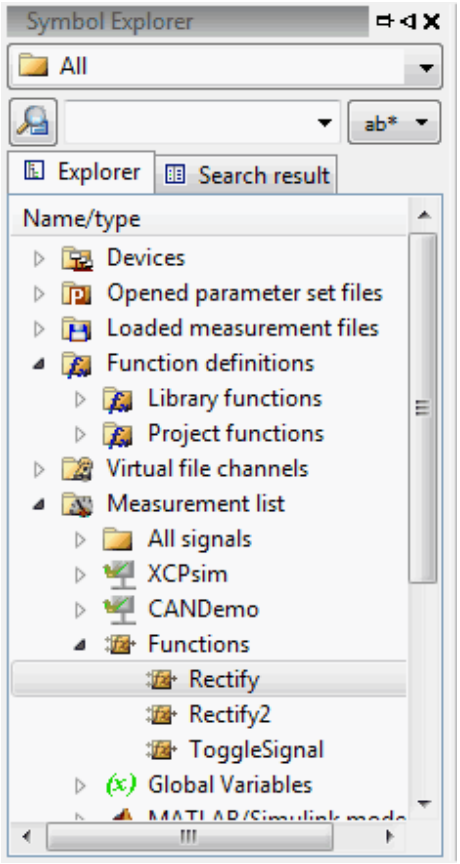
Result

The selection of measuring mode `on input` or, in this case, `on signal 'Channel1'` causes your function to be executed each time its input value receives a new value.

Here, `Channel1` is a CAN signal that is passed to the function as an input signal.

13. Close the measurement configuration using the  icon.

Figure 4-11: Functions in the measurement signal list of the Symbol Explorer



Using a function as a virtual measurement channel

Now you can use your function, for example, as a virtual measurement channel by moving it from the Symbol Explorer to a Graphic window via drag & drop (see also section Using a Function in a Graphic Window as a Virtual Measurement File Channel on page 33).

14. Move the function to a Graphic window using drag & drop.

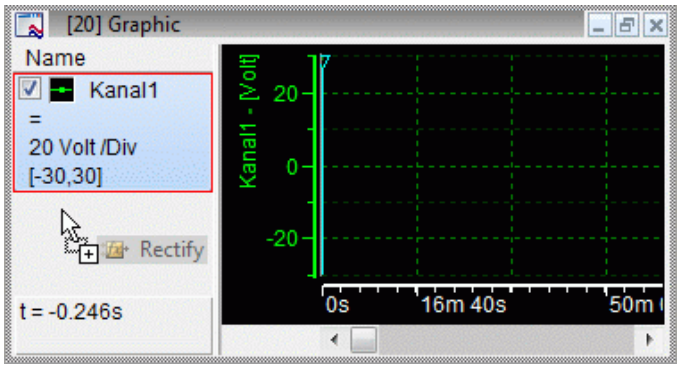


Figure 4-12: Inserting a function in the Graphic window

Result

The result of an example measurement may look like the following:

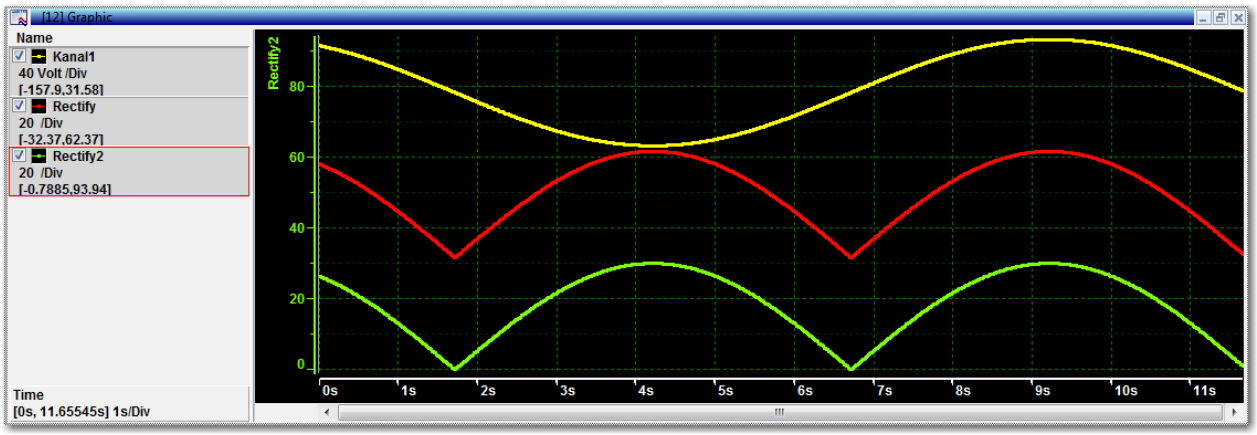


Figure 4-13: Result of a measurement (example)

4.1.3.4 Calling a Function From Another Function

Objective

It is possible to call a function from another function.

Example:

```
function Function1 (Var A)
{
  double Result;
  Result=A*2;
  return SecondaryFunction(Result);
}

function SecondaryFunction (Var B)
{
```

```
  return B+1;
}
```

4.1.3.5 Creating a Subfunction in a Script and Calling it

Subfunctions of scripts

In CANape, subfunctions of scripts are referred to as subfunctions. The advantage of subfunctions is that they can be created together with the calling functions in a single script file and thus can also be exported in their entirety without having to transfer the complete project.

Syntax

However, they do not differ from a conventional function in regard to syntax.

Multiple subfunctions in a script

A script may contain multiple subfunctions. However, the function definitions must precede the main part of the source code of the script (the main part of the script begins after the definition part of all variables and subfunctions).

Validity

Subfunctions are valid only for the script in which they were defined. They cannot be called from another script or other user-defined function. However, it is possible to use `#include` to insert a script containing a set of subfunctions into another script.

4.1.3.6 Calling a Function From a Script

Objective

It is also possible to call a function from a script.

Example:

Function:

```
function Function1 (Var A)
{
  double Result;
  Result=A*2;
  return Result;
}
```

Script:

```
Double Output;
Output=Function1(100);
Write("%d", Output);
```

4.1.3.7 Using a Function in Data Mining as a Virtual MDF Signal

Data Mining

Data Mining is a method for automatic offline evaluation of signals in a series of existing MDF files. A function is applied to these measurement files in order to search through the measurement values according to certain criteria.

Advantage

For example, you can filter function results within a single search run for multiple MDF files.

1. Select **Analysis|Data Mining|Editor** in the menu.

   If you have not yet loaded a measurement file, a dialog appears asking whether you want to load a measurement file.

2. If necessary, select **[Yes]**.

   An Explorer window opens.

3. Select the desired MDF file.
4. Confirm your selection with a double click or with **[Open]**.

   The **Data Mining** window opens.

5. All virtual measurement file channels are displayed in the **Configuration|Methods** area.
6. In addition, you can insert additional ones or create new ones by clicking **Edit|Add Function**.

   Creating a new function: The further menu guidance for creating a new function corresponds to the description in section Writing the Functions on page 30 starting from step 3.

   Selection: The further menu guidance for selecting an existing function corresponds to the description in section Using a Function During a Measurement on page 35 starting from step 3.

4.1.4 Example Functions

Example 1: User-defined function Add()

```
// user function Add()
function Add(Var a, Var b)
{
  return a + b;
}
```

Example 2: Subfunction in a script

```
// simple_sub_functions_demo.scr

// Variable definitions
Var var1;

// Subfunction definitions
Function F1()
{
var1++;
}

Function F2(Var param1, Var increment_constant)
{
  Var var2; // Local variable inside subfunction

  var2 = increment_constant;
```

```
  F1();
  param1 = param1 + var2; //Using parameter for output
}

Function F3(Var param1[], Var param2)
{
  write("%s : (value=%d)", param1, param2);
  return 1;
}

// Main Script Part
var1 = 0;
F1();
F2(var1, 1);

if (var1!=3)
{
  var1 = F3("Error", var1);
}
else
{
  var1 = F3("Ok", var1);
}
Write("var1 = %d", var1);
```

Example 3: Passing of one-dimensional array or characteristic curve

```
function ChecksumBlock(var Signal1[])
{
  unsigned long DpkCtr1_Size, i;
  unsigned long changeVal;

  DpkCtr1_Size = xDimension(Signal1);

  for (i = 0; i < DpkCtr1_Size; i++)
  {
    changeVal += Signal1[i] * (i + 1);
  }

  return changeVal;
}
```

Note: Pay attention to the square brackets of the variables in the function header if you want to pass characteristic curves or one-dimensional arrays to a function.

In the case of multidimensional arrays or maps, the number of square brackets must match the number of the dimension of the respective array/map.

For two-dimensional arrays, use two square brackets one after the other, e.g., `Signal1[][]`.

4.1.5 Global Function Library

Library functions

Use also the global function library included with CANape since CANape 8.0. During the setup process, the global function library containing a few function definitions is created by default.

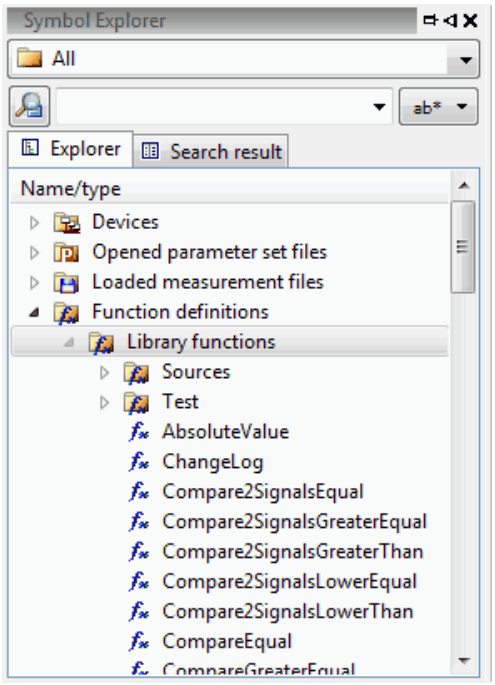
This can be used unchanged or edited according to the user's needs.

The functions of the global function library are referred to as library functions.

You will find the library functions in the Symbol Explorer under **Function Definitions|Library Functions**.

1. If necessary, open the Symbol Explorer with **Display|Symbol Explorer** or using the icon.

Figure 4-14: Display of the library functions in the Symbol Explorer



Note: The global function library contains only functions. It does not contain any scripts, global variables, or Seed & Key algorithms.

Import from library

The library functions must be imported into the Functions Editor for editing.

Export to library

Likewise, modified library functions or the user's own project functions can be exported to the global function library.

The two commands for this can be found in the popup menu of the function in the tree view of the Functions Editor.

4.1.6 Integrating External Function Libraries

Expanded scope of functions

If the scope of functions and options available to you in CANape are not sufficient, you can write the desired functions to your own function library (extension `*.dll`) outside of CANape.

Advantages

Advantages of using an external function library:

> You can integrate already existing C code in CANape.

> You can solve more complex tasks than with the functions and scripts provided by CANape.

> The DLL is executed faster because it is already compiled.

Arguments

Arguments are passed either as value (`by value`) or as reference (`by reference`).

`value`

In the case of `value`, a copy of the parameter value is passed to your function (input parameter). Changes to this parameter value in the function remain invalid for calling programs.

CANape uses only `double` values for internal calculations.

For example, if arguments are passed from a CANape script to your DLL function `by value`, these arguments always appear in your DLL function as `double` type.

`reference`

Arrays (char arrays) or strings cannot be passed `by value`. For this reason, these are always passed `by reference`.

For example, if arguments are passed from a CANape script to your DLL function `by reference` (as reference) and are not `double` type, they appear in your DLL function as a type other than `double` type. A char array is passed as `kDtypByte`, for example.

Even if a parameter value is to be changed by the function (output parameter), you must use the `reference` type.

Return value

The return value of the function is always `double` type.

Testing the behavior

You can determine the detailed behavior of CANape with other data types by debugging your DLL file.
To do so, attach to, e.g., `canape32.exe`, while you call your DLL function from CANape (e.g., in a script).

Reference: For more information on this topic, refer to the Application Note `AN-IMC-1-012_How_to_use_C-Code_Functions_in_CANape.pdf` in your CANape installation directory under `Documentation`. In addition, you will find example code in the C programming language under `Examples\FunctionDemoDLL\Sources`.

Integrating a DDL library:

1. Open the **Tools|Options** menu.

   The Options dialog opens.
2. Select the **Miscellaneous|Functions and Scripts** area.
3. Click the **[Add]** button.

   An Explorer window opens.
4. Select the desired DLL file.
5. Confirm your selection with **[Open]**.

   Your DLL file is shown in the list of libraries.
6. Close the Options dialog with **[OK]**.

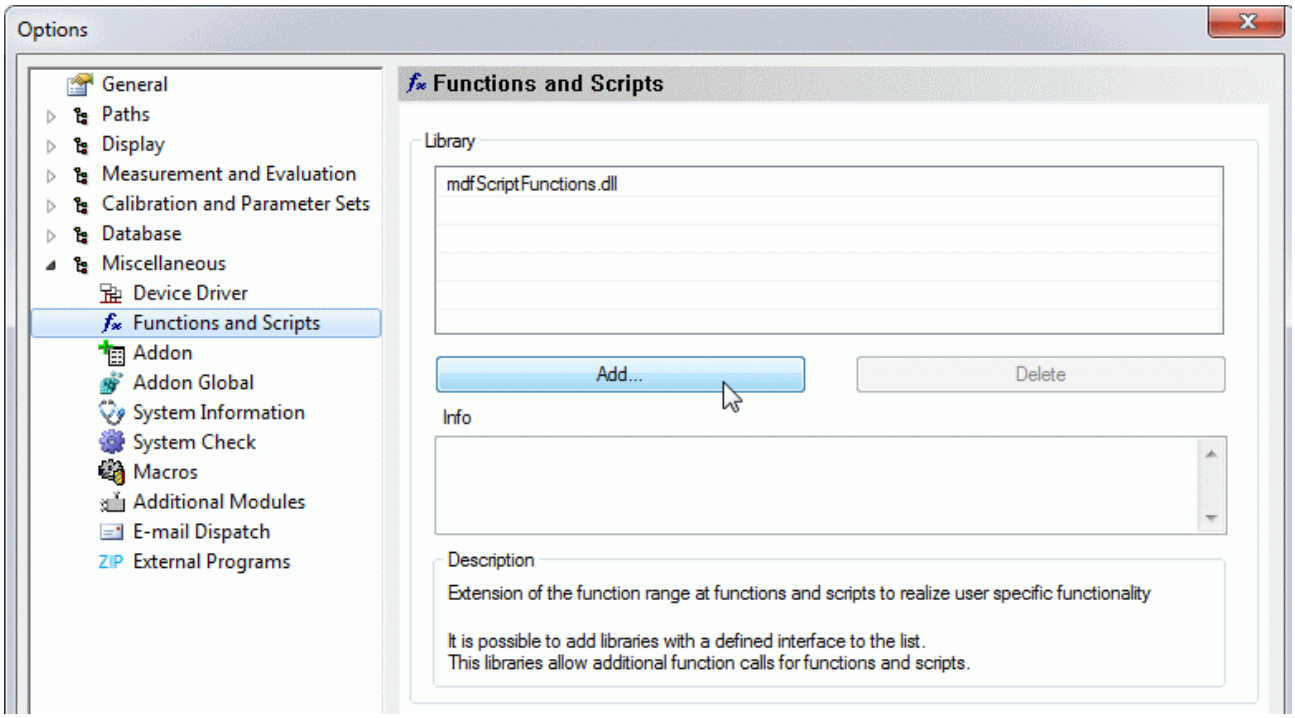


Figure 4-15: Integrating an external function library via the Options dialog

7. Now open the Functions Editor using the f_x icon.
8. Then, click with the right mouse button in the Editor window.

   Analogous to section Predefined Function Groups and Code Blocks of CANape on page 16 you can now use your externally created functions in the Editor window under the popup menu **Insert|DLL functions**.

4.1.7 Debugging of Functions

Objective

We recommend debugging of functions for purposes of diagnosing and locating errors in user-defined functions.

Unlike scripts, functions cannot be debugged in the Functions Editor using breakpoints due to their high priority.

Procedure

For this reason, the output of debug information in the Write window by means of

program functions, such as `Write`, `Writef`, `Print`, or `Printf` must be used instead.

Example: On the basis of the following example, the user wants to determine why the return value of the function (`Result`) assumes only integer values.

```
function Funktion_1 (var Input)
{
  double Result=0;
  double Offset=10;
  Long Result2=0;
  Result=Input+Offset;
  Write("Result Write: %g", Result);// line break, output
                                    before information loss

/*
..
*/
  Result2=Result;// conversion double to long, thus information
                 loss

/*

*/

  Result=Result2;// reconversion to double, information loss
                 remains
  Print("Result Print: %g", Result);// no line break, output
                                    after information loss
  Print("Result Print: %g", Result);// no line break
  Printf("Result Printf: %g", Result);// line break
  return Result;
}
```

Note: The function serves only as an example and would look much more comprehensive in a real application.

Generating debug information in the Write window:

1. Create the example function `Function_1` (see also section Writing the Functions on page 30).
2. Create a Write window using the **Display|Other windows|Write window** menu.
3. Create a Graphic window using the **Display|Measurement windows|Graphic window** menu.
4. Move a signal, e.g., `channel1`, from the Symbol Explorer to the Graphic window using drag & drop.
5. Call the function. To do so, select between one of the following options:
   a) Using a Function in a Graphic Window as a Virtual Measurement File Channel on page 33

b) Calling a Function From a Script on page 42

c) Using a Function During a Measurement on page 35.

If you have only one measurement signal for debugging, you can use drag & drop to move `Function_1` directly to the Graphic window via `channel1` and create the measurement function in the measurement list.

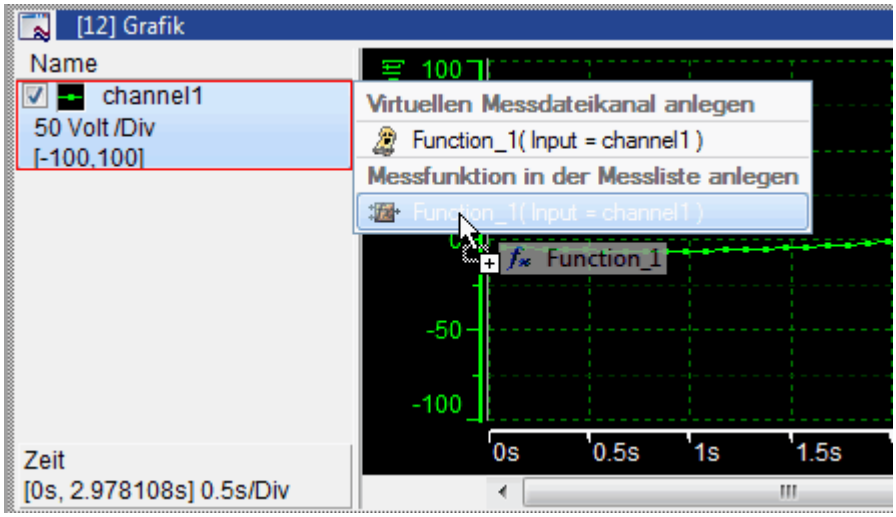


Figure 4-16: Creating a measurement function in the measurement list using drag & drop

6. Start the measurement with .
7. Evaluate the messages in the Write window.

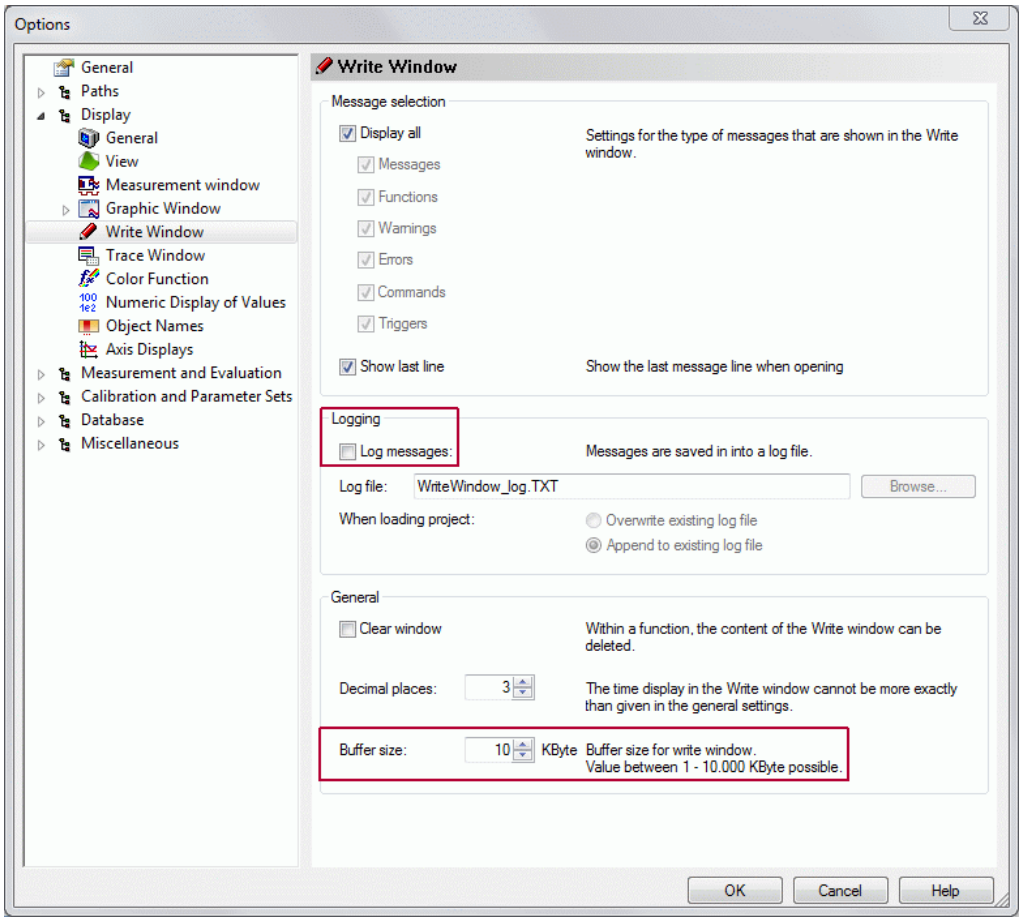
Buffer size

Check the buffer size of your Write window and adapt it if necessary. If the buffer size is not sufficient, have information to that effect written to a logging file.

Configuration of Write window

Click in your Write window with the right mouse button and select the appropriate items under **Configuration** according to Figure 4-17: Configuration of Write window on page 50.

Figure 4-17: Configuration of Write window



4.2 Scripts

4.2.1 Writing the Scripts

Functions Editor

Like functions, scripts can be created with the Functions Editor (see section Functions Editor on page 12).

Creating a new script:

1. Open the Functions Editor using the f_x icon (see also section Functions Editor on page 12).
2. In the tree view of the Functions Editor, click with the right mouse button on **Scripts|New** or select **Edit|New|Script** in the menu**.**

   The **Script Properties** dialog opens.
3. Name the script as desired and select the directory, if necessary. The default directory for scripts is the respective project directory.
4. Select the file type, e.g., `*.cns`.

Note: SCR and CNS are among the available data types. If possible, the CNS extension should be selected in order to avoid mix-ups with Windows screensaver files. For .NET scripts, there are additional data types. The last extension selected is always suggested during a CANape session.

5. Confirm your inputs with **[OK]**.
6. Write your new script in the Editor window.

Predefined function groups and code blocks

Same as for the writing of functions, CANape provides code blocks and a variety of predefined CASL functions (organized in function groups) here as well to aid your (see section Predefined Function Groups and Code Blocks of CANape on page 16).

IntelliSense

Get acquainted with CANape's IntelliSense! For example, type the starting letter(s) of a predefined CASL function. Select the function from the automatically displayed IntelliSense list. Use <Tab> or <Enter> or double-click with the left mouse button to transfer the selected function to the Editor window.

Note the information that CANape additionally displays about the function (see also IntelliSense on page 30).

Inserting device variables in a script:

1. In the popup menu of the Editor window, click **Insert|Variable|Device variables** and the relevant device, e.g., **CCPsim**.

   The Database selection opens.
2. Select the desired variable, e.g., `ampl`.
3. Confirm your selection with a double click or with **Accept**.
4. Your selected device variable, e.g., `CCPsim.ampl`, will be written in your script.

Compiling and saving a script:

1. When you finish writing your script, compile the script using the icon or the **Compile|Compile** menu item.
2. Note the messages output by CANape in the Message window (**Output** page) at the bottom of the screen.

   If the message "The program is compiled" appears, the syntax of your script is correct. The red X that marks your script in the tree view disappears.

   If an error message is displayed, try to eliminate the error with the help of the message. The red X that marks your script in the tree view is retained.
3. Save your script using or **File|Save**.

   Each script is stored as an identically-named file with the previously selected extension in the working directory. The asterisk that marks your script in the tree view disappears. Scripts that are not compiled can also be saved in this way.
4. Close the Functions Editor using the icon or the **File|Close** menu.

Note: When writing, note that scripts in CANape are subject to a relatively low process priority. Their execution is guaranteed only every 100 ms.

4.2.2 Saving and Forwarding Scripts (Exporting/Importing)

Saving

The created scripts are stored automatically in the working directory as a script file. You can add further directories in the Functions Editor. The file type of the script file you have already selected when creating the new script.

Exporting/ importing

Scripts can be exported from and imported to the Functions Editor.

Formats

Scripts are exported in the respective script format (e.g., `*.scr`) or HTML format (`*.html` or `*.htm`).

Scripts must be present in the script format (`*.scr` or `*.cns`) in order to be imported.

4.2.3 Task Manager

Starting the scripts

The Task Manager provides you the option of starting scripts automatically and manually.

1. Open the Task Manager by clicking the icon (Open Task Manager) or using the menu under **Tools|Task Manager**.

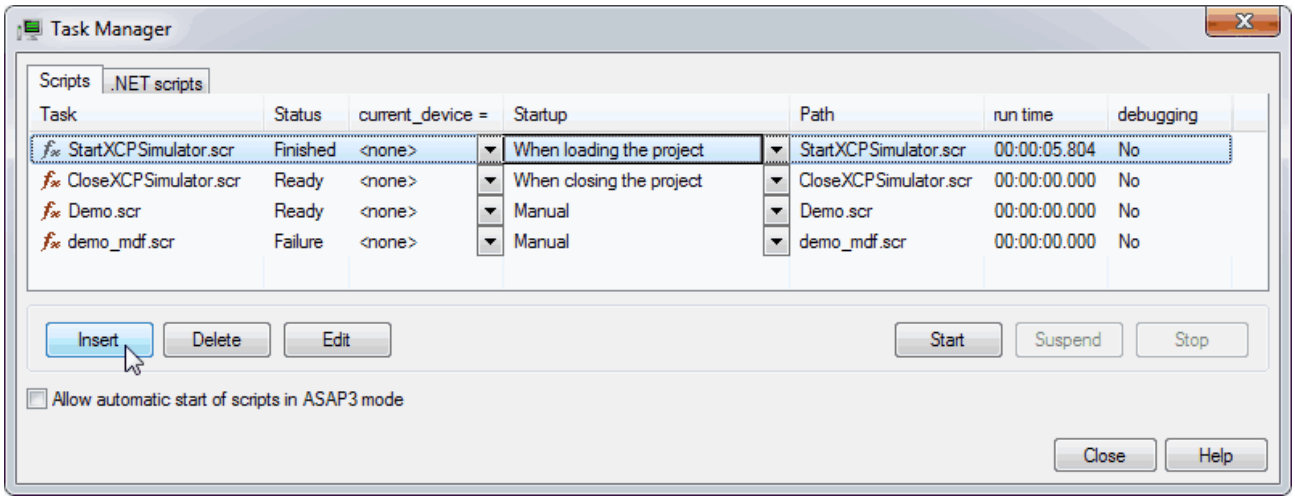


Figure 4-18: Task Manager

2. Insert the desired script using the **[Insert]** button.
3. In the **Startup** column, you can choose between various events and thus specify when your script is to be started.

Start types:

- Manual
- When the project is loaded (and thus automatically when CANape is started)
- When the project is closed (and thus automatically when CANape is ended)
- When the measurement is completed
- After synchronization of the memory image
- When a measurement file is loaded
- When a measurement file is updated
- When the global measurement cursor is activated, moved, and deactivated.
- When a measurement is started (the execution of the script is started at the same time as the measurement)
- Before a measurement is started (at the start of a measurement, the corresponding script is executed; when the script is complete, the measurement starts).

Note: The measurement start is thus delayed by the execution time of the script.

4. If you choose **Manual** for the startup, you can initiate the script execution by pressing the **[Start]** button.
5. If you would like to edit a script, select it and click **[Edit]**.

   The Functions Editor with toolbar, Editor window, and Message window opens.

Note: The modified script must be compiled before exiting the Functions Editor.

6. In the **current_device=** column, it is possible to select the device in the device configuration for which the script is to be executed.

   If the universal device prefix **current_device** is used in the script and the column contains the entry **<none>** (default entry), a dialog, appears in which the device must be selected.

Note: Scripts with the same automatic start type (e.g., "At start of measurement") are started one after the other in the order in which they appear in the Task Manager.

This does not pose a problem for scripts that are processed independently of one another.

However, if a script is not to be started until another script has been completed beforehand (e.g., one script initializes variables and the next script uses these initialized variable), a main script should be called in the Task Manager that executes the individual scripts one after the other using the `CallScript()` function (see also section Call-up of Scripts on page 53).

Important: If values are assigned via scripts (e.g., `ampl=4`), this change takes effect immediately in online mode even if direct calibration is not activated (in the **Calibration|Direct calibration** menu). To go offline, click **Calibration|Go offline** in the menu.

4.2.4 Call-up of Scripts

Multiple options

Scripts can be called in various ways in CANape. You can use one of the following options:

- Executing a Script Via the Menu Bar of CANape, see page 54
- Calling a Script Via the Task Manager, see page 54
- Executing a Script From the Functions Editor, see page 54
- Calling a Script From Another Script, see page 55
- Calling a Script Via a Control on a Panel, see page 56
- Calling a script when executing CANape, see section Script Behavior When CANape is Running on page 58

Write window

In a Write window, you can track the commands executed according to the script, the status messages, and the error messages of the driver as well as values of functions and scripts.

Opening a Write window:

1. Open the Write window via the menu bar **Display|Other windows|Write window**.

4.2.4.1 Executing a Script Via the Menu Bar of CANape

Objective

You can execute scripts directly via the menu bar of CANape.

1. Select **Tools|Execute script** in the menu.

   An Explorer window opens.
2. Select the desired script file, e.g., `Script_1.cns`.
3. Confirm your selection with a double click or with **[Open]**.
4. The script is executed.

Important: If values are assigned via scripts (e.g., `ampl=4`), this change takes effect immediately in online mode even if direct calibration is not activated (in the **Calibration|Direct calibration** menu). To go offline, click **Calibration|Go offline** in the menu.

4.2.4.2 Calling a Script Via the Task Manager

Objective

You can execute scripts directly from the Task Manager using the **manual** startup.

Because you can also open the Functions Editor from the Task Manager, you can likewise call a script there.

1. Open the Task Manager by clicking the  icon (Open Task Manager) or using the menu under **Tools|Task Manager**.
2. Insert the desired script using the **[Insert]** button.
3. If you choose **Manual** for the startup, you can initiate the script execution by pressing the **[Start]** button.

   or

   If you have selected an automatic startup, you can use the **[Edit]** button to open the Functions Editor and call the script from there (see section Executing a Script From the Functions Editor on page 54).

4.2.4.3 Executing a Script From the Functions Editor

Objective

You can execute a created script from the Functions Editor directly after compilation.

1. Open the Functions Editor using the  icon.
2. Select the script to be executed in the tree view.
3. Click the  icon or select **Debug|Execute script** in the menu in order to execute the script.

4.2.4.4 Calling a Script From Another Script

Objective: You can call a subscript from a main script.

Advantage: This is advantageous when you want to ensure a certain chronological sequence of your commands.

Example: In the following example, you see how return values can be returned to the main script using this option:

SubScript.cns:

```
Double Result=0;
Result++;
Write("Result SubScript: %d", Result);
return Result
```

MainScript.cns:

```
Double Result;
int i;
ClearWriteWindow();
for (i=0; i<=10;i++)
{
  Result=CallScript("SubScript.cns",true);
  Write("Result MainScript: %d", Result);
}
```

Example: In the following example, you see how arguments/parameters can be passed to the subscript:

SubScriptWithArguments.cns:

```
long i, nArgs;
char argBuffer[256];

nArgs = GetArgCnt();

Write("Number of specified arguments: %d", nArgs);
for(i = 0; i < nArgs; i++)
{
  GetArg(i, argBuffer);//Get the i. script argument
  Write("Argument %d: %s", i+1, argBuffer);
}
```

MainScript.cns:

```
CallScriptEx("SubScriptWithArguments.cns", "Argument_1
             Argument_2 Argument_3");
```

If the arguments contain blank spaces or paths, these must be passed as follows:

MainScript.cns:

```
CallScriptEx("SubScriptWithArguments.cns",
"MyFirstArgument \"My argument containing spaces\" \"C:\\My
```

```
Path\\My Data File.dat\"");
```

4.2.4.5 Calling a Script Via a Control on a Panel

Objective: Scripts can also be started by controls, such as a button, in CANape.

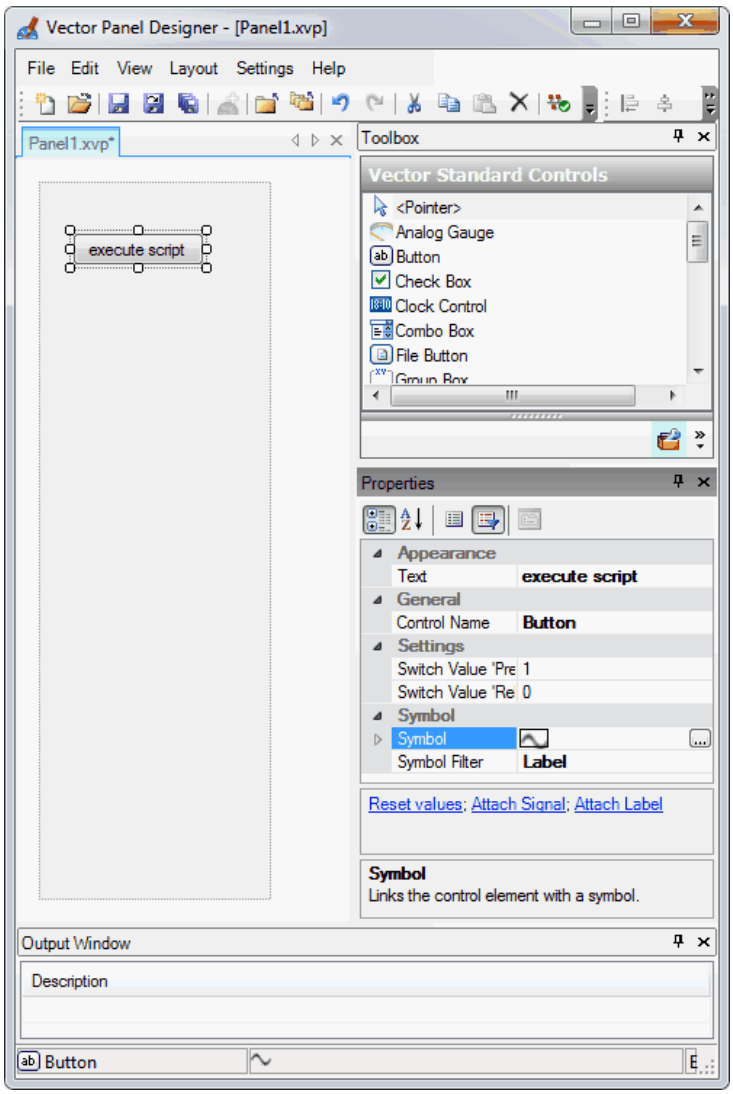
Requirement: You must first create a panel or insert an existing panel in CANape.

Creating a panel:

1. Click the icon or the **Tools|Panel Designer** menu.

   The Vector Panel Designer window opens. It consists of the toolbar, the panel view, the toolbox, the properties, and the Output window.
2. Move the **Button** control from the **Toolbox** to the panel surface using drag & drop.
3. Specify the **Text** to be displayed on the button, e.g., "execute script", under **Properties**.

Figure 4-19: Vector Panel Designer window



4. Click the Assign Label **link.**

   The **Define Label** dialog opens.
5. Under **Label**, enter the name for the control, e.g., `Scriptstart`.
6. Confirm your input with **[OK]**.
7. Save the panel under the desired name in the desired directory.
8. Close the Panel Designer.

   Now you can insert the panel in CANape, see Inserting an existing panel.

Inserting an existing panel:

1. Select **Display|Other windows|Panel window**.

   An Explorer window opens.
2. Select your panel file, e.g., `Panel1.xvp`.
3. Confirm your selection with a double click or with **[Open]**.

   The **Links** dialog opens.

   The individual controls of the panel are linked to the corresponding data objects here.

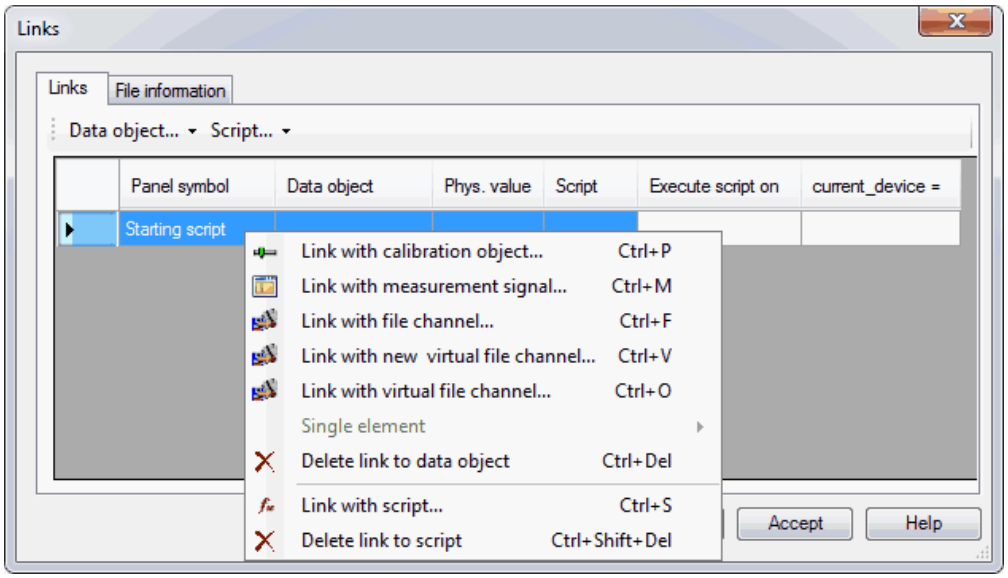


*Figure 4-24: **Links** dialog*

4. Open the popup menu of the `Scriptstart` control and select **Link with script** there.

   or

   On the **Links** page, select the **Script|Link with script** menu.

   An Editor window opens.
5. Select the script that is to be executed when the `Scriptstart` control is actuated, e.g., `Script_1.cns`.
6. Confirm your selection with a double click or with **[Open]**.

   The selected script is now displayed in the appropriate column of the **Links** dialog.

7. Confirm the created link with **[OK]**.
8. Now, when you click your button in the Panel window, your script will be executed.

Changing the script assignment

You change the script assignment at any time by clicking your panel with the right mouse button and selecting Link Data Objects.

9. To do so, click your panel with the right mouse button and select **Link data objects**.

   The **Links** dialog opens. You can change your script assignment using the popup menu of the control or the **Script** menu.

4.2.5 Script Behavior When CANape is Running

Command line options

Three different CANape command line options configure the script behavior when CANape is running. These options can be added to the **destination** in the properties dialog of the link that is used for starting CANape.

Syntax	`canape32.exe [Options] [File]`
-b Script File <*.scr>	Starts the specified script file (batch mode).
-bc Script File <*.scr> Script Argument	Starts the specified script file with the specified arguments (batch mode).
-PATH SCR "path"	Sets the path from which script files are selected for immediate execution.

Note: The path ("path") must be written in quotation marks if it contains blank spaces. It can be specified as an absolute path or relative path to the current working directory (project directory).

4.2.6 Debugging of Scripts

Objective

We recommend debugging of scripts for purposes of diagnosing and locating errors in user-defined scripts.

Breakpoints

In order to execute scripts only up to a certain code line, you can set breakpoints in the Functions Editor.

The **Breakpoints** page of the Message window displays all the set breakpoints. You can use the popup menu to delete and activate/deactivate breakpoints or to navigate to the corresponding source text.

During debugging, you can examine all values of your variables on the **Variables** page of the Message window.

Setting breakpoints

1. You can set a breakpoint in the Functions Editor by clicking the left side (gray column) of the Editor window.

   or

   Place the cursor inside the line in which a breakpoint is to be set and select the icon (Toggle Breakpoint on current line).

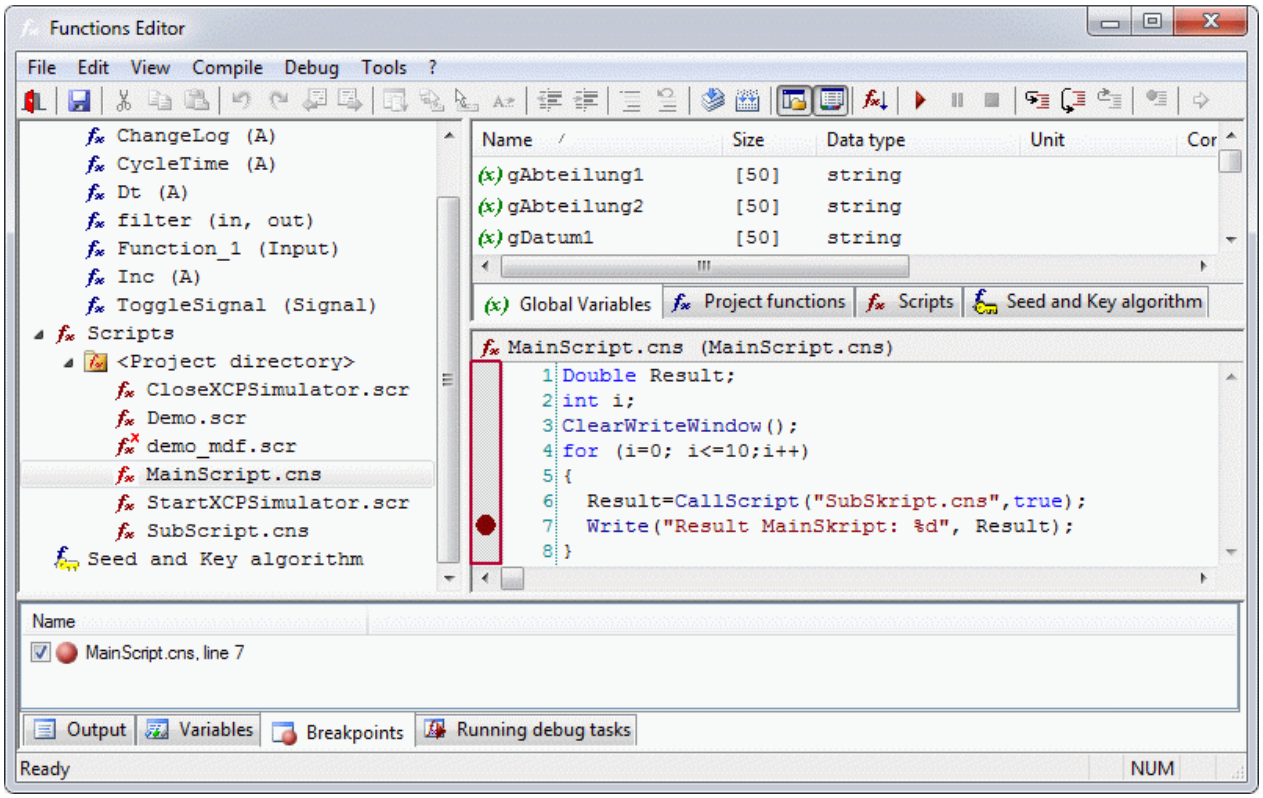


Figure 4-20: Breakpoint in the Editor window

Debugging

2. In order to execute the script up to (and including) the breakpoint, click the icon (Start Debugging current script).
3. To stop the script, simply select (Stop Script).
4. With (Step In), you go through each individual line of your program.

   The debugger then also jumps into called scripts and functions.
5. You can exit these again with the icon (Step Out).
6. On the other hand, use (Step Over) if you are not interested in the details of a script or function that you call from your overlying main script.

4.2.7 Example Scripts

Simple examples

Below are a few simple examples that can be used multiple times:

> Load parameter set

```
CCPsim.LoadParameterset()
```

> Measurement Start/Stop:

```
Start();
Stop();
```

> Measure and Calibrate:

```
x = CCPsim.ampl;
CCPsim.ampl = x;
```

> Send e-mail:

```
SendMail();
```

> Send CAN messages:
`<Device>.SendMessage(0x1a1,0xff,0x7f)`

Example 1: In the following script, sequentially ordered statements are executed automatically after the `CCP Demo` project is loaded:

1. Start measurement.
2. Seconds after the measurement start, the `PWM_Level` parameter is calibrated to 5 volts.
3. After an additional 3 seconds, the measurement stops.

```
Start();
Sleep(5000);
CCPsim.PWM_Level=5;
Sleep(3000);
Stop();
```

Example 2: In the following script, sequentially ordered statements are executed automatically after the `CCP Demo` project is loaded:

1. Open and clear Write window.
2. The value of the `ampl` parameter is cyclically increased until it reaches a predefined limit.

   This limit is defined in global variable `MaxValue`. The value can also be changed in a Calibration window when the measurement is active.
3. Reset parameter to 1 when the limit is reached.

```
//Prepare the Write window
OpenWriteWindow();
ClearWriteWindow();

//Endless loop
while (1==1){

  //Check if the value of the global variable MaxValue is
between 0 and 50.
  if (MaxValue < 0 || MaxValue > 50) MaxValue = 10;

  //Increment the parameter ampl from device ccpsim by step 1
  //if the current value is less than the value of MaxValue
  if (CCPsim.ampl >= MaxValue)
  {
    CCPsim.ampl = 0;
    Write("MaxValue reset on 0, real value:%d", CCPsim.ampl);
  }
  CCPsim.ampl = CCPsim.ampl+1;
```

```
  //Wait 1 s
  Sleep (1000);

}
```

Additional example scripts

CANape includes additional example scripts in its various sample configurations. These are displayed under **Scripts** in the tree view of the Functions Editor after the respective sample configuration is opened.

4.3 Variables

4.3.1 Creating a Global Variable

1. Open the Functions Editor using the icon (see also section Functions Editor on page 12).
2. In the Global Variables tree view, select the New popup menu command.

   The **Variable** dialog opens.
3. Name the variable as desired, e.g., `gVarDouble`, and assign the desired data type and other properties to it.
4. Confirm your inputs with **[OK]**.

   The **Variable** dialog closes and the newly created variable is displayed in the tree view.
5. Once the variable has been saved, it is no longer marked with an *.
6. Close the Functions Editor using the icon or the **File|Close** menu.

   Your global variable is also displayed under **Devices|Global variables** in the tree view of the Symbol Explorer.

4.3.2 Setting a Global Variable to a Defined Value

Defining a value

You can influence when the global variable is to be set to a defined value.

For example, if you want to reset or initialize your global variable at the start of a measurement, you can do this by:

- Writing your own function (e.g., `Reset_gVarDouble()`), or
- Executing a script

Initializing using your own function:

1. In the Editor window, create the example function `Reset_gVarDouble` as follows (see also section Writing the Functions on page 30):

```
function Reset_gVarDouble()
{
  // Enter the code here
  gVarDouble = 0;
  Write("The global variable gVarDouble was changed to the
  value %f.",gVarDouble);
  return;
}
```

2. Compile the function with and save it.
3. Move the global variable `gVarDouble` from the Symbol Explorer (**Devices|Global Variables** area) to a free area of your display page.

   The possible Display and Calibration windows are provided for selection.
4. Select **Default window**.

   A Calibration window with global variable gVarDouble and the value 0 is displayed.
5. Open the measurement configuration using <F4>, the icon, or the **Measurement|Measurement Configuration** menu item.
6. In the popup menu, select **Add Function**.

   The **Function** dialog opens.
7. Click the **[Select]** button.

   A Selection of function definitions opens.
8. Select `gVarDouble` and confirm your selection with a double click or **[OK]**.
9. Close the **Function** dialog with **[OK]**.
10. In the measurement configuration, set the measuring mode of your function to `on event 'MeasurementStart'`.
11. Close the measurement configuration with .
12. Set the value of your global variable `gVarDouble` in the Calibration window to, e.g., 4 (go to the cell with a double click, change the value, and <Enter>.
13. Start a measurement with .

    The value of your global variable `gVarDouble` in the Calibration window is initialized to 0.

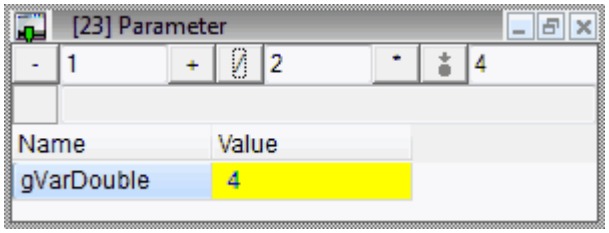


Figure 4-21: Calibration window with modified value = 4

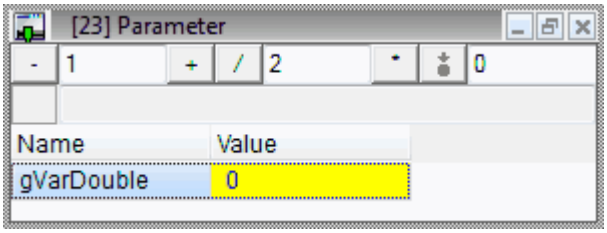


Figure 4-22: Calibration window with initialized value

Initializing using a script:

1. In the Editor window, create the example script `Reset_gVarDouble` as follows (see also section Writing the Scripts on page 50):

```
gVarDouble = 0;
Write("The global variable gVarDouble was changed to the
value %f.",gVarDouble);
```

2. Compile the script with and save it.
3. Create a Calibration window with the global variable `gVarDouble (see steps 3 to 4 above)`.
4. Also create a Write window for better monitoring via **Display|Other windows|Write window**.
5. Open the Task Manager using the icon and use **[Insert]** to insert the newly created `Reset_gVarDouble` script (see also section Task Manager on page 52).
6. Set the **Startup** of the script to **Before the measurement**.
7. Select the `Reset_gVarDouble` script and click **[Start]**.

   The script is executed. The value of your global variable `gVarDouble` in the Calibration window is initialized to 0.

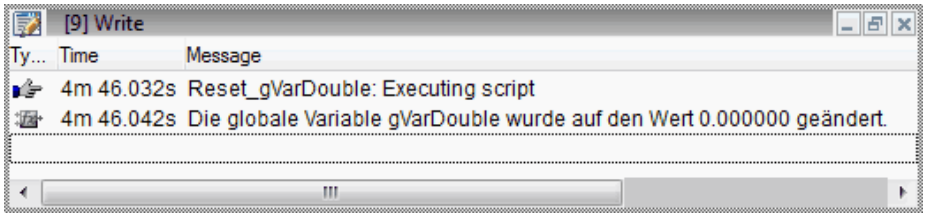


Figure 4-23: Display of the Write window after executing the `Reset_gVarDouble` script

Note: By means of a predefined CASL function of the "Program functions" function group `UpdateCalibrationWindows()`, you can also have the start values displayed in the Calibration windows automatically.

4.3.3 Setting a Local Variable to a Defined Value

Different start values

If a function is used, for example, in the Data Mining environment or as a virtual measurement file channel, CANape provides the option of assigning a different start value of a local variable to each instance of a function. A comment can also be provided for this. It can be seen later under the properties of the function (see section Writing the Functions on page 30).

Example: An example of this is the library function `MovingAverage()` of CANape:

```
function MovingAverage(var signal)
{
  //! number of last measure values to be used for the
  calculation
  int smoothFactor = 4;
....
```

`//!`

The local variable `smoothFactor` is initially assigned the value 4 here. By adding `//!` in front, this variable can assigned a different start value in the properties of a virtual measurement file channel, for example. All characters that appear after `//!`

can also be seen in the properties as information.

Setting another start value (if function is used as a virtual measurement file channel):

1. Move the library function `MovingAverage()` to an empty area of a display page using drag & drop.

   The possible Display and Calibration windows are provided for selection.
2. Select **Default window**.

   A Graphic window is displayed, and **Virtual measurement file channel** and **Measurement function** are available for selection.
3. Click **Virtual measurement file channel**.

   The virtual measurement file channel is displayed in the Graphic window. However, an input signal is not yet assigned to the function.
4. Select **Properties** in the popup menu of the Graphic window.

   The **Properties** dialog opens.
5. Select the line for the input signal and click **[Link manually]**.
6. Select `XCPsim` in the **Link manually** dialog that pops up.
7. Confirm your selection with a double click or **[OK]**.

   The **Database Selection** opens.
8. Select the `channel1` measurement signal.
9. Click **Apply** and close the database selection using the icon.
10. In the **Properties** dialog, now select the line with the function parameter `smoothFactor` and click **[Change parameter value]** .
11. Enter the new value for the initialization, e.g., 10, in the dialog that pops up.
12. Confirm your input with **[OK]**.
13. Confirm your changes in the **Properties** dialog with **[OK]**.

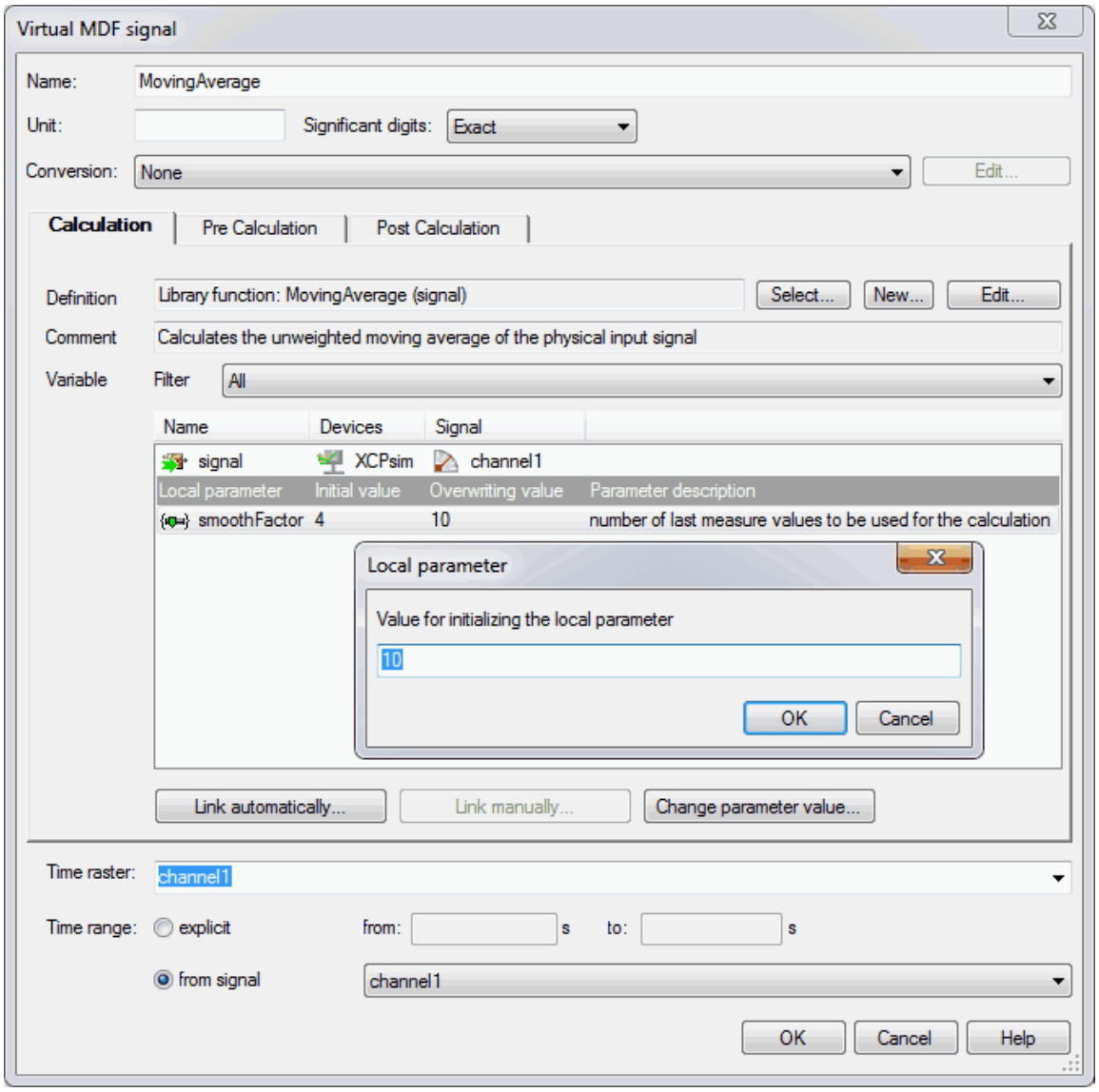


Figure 4-24: Local variable `smoothFactor` is assigned a start value of 10.

4.3.4 Inserting a Device Variable

1. Open the Functions Editor using the icon (see also section Functions Editor on page 12).
2. Select **Device variables** in the popup menu of the Editor window of the Functions Editor.

   The created devices are displayed, e.g., `CCPsim`.
3. Click the desired device `CCPsim`.

   The **Database selection** of the device containing the available device variables opens.
4. Select the desired device variable with a double click.
5. Close the database selection with .

   The inserted device variable is located in the function/script as follows: `<DeviceName>.<VariableName>`.

Note: If the variable name contains one or more dots, the name string must be placed in single quotation marks, e.g., `KWPsim.'ECU_Identification.Ident_Digit_1_0'`

Access in the case of multiple use

If a CAN signal is listed more than once in a DBC file, a script or function can be used to access it.

```
value = <CANdevice>.'$<CAN message name>$<signal>';
```

5 General Tips

This chapter contains the following information:

5.1 Proper Terminating of Functions and Scripts

Terminating with `cancel`

If, for example, a parameter has reached a certain value, the control structure `cancel` can be used to terminate a function or script.

If the `cancel` command is used in a function, the function is aborted without a return value. The control is passed back to the calling routine.

If the `cancel` command is used in a subfunction or script, the script is terminated immediately.

Example: If **[Yes]** is selected in the called dialog (return value is 0), the script is terminated.

```
long result;
result = UserQueryDialog ("Yes | No", /*buttons*/
0, /*question*/ "Do you want to cancel the script?");
if (result == 0)
{
  cancel;
}
print("Continue...");
```

Terminating with `break`

The `break` command aborts a loop (`for` or `while`) or a `switch` statement immediately.

Example: The `break` command aborts a `for` loop.

```
// break

long i;
for (i = 0; i < 15; i++)
{
if (i == 10)
  {
  break;
  }
}
Write("i = %d",  i);
```

5.2 Access to System Information

Device `System`

The option exists in CANape to use system information in scripts. The corresponding option must be activated for this. This option causes a `System` device with a wide range of different system variables to be created.

Uses

This approach allows you to use individual variables of the system in scripts.

Activating system information:

1. In the menu bar, click **Tools|Options**.

   The Options dialog opens.

2. Select **Miscellaneous|System information** in the tree view of the Options dialog.

3. Activate the check box of the system information on the right side of the dialog.

   A device named `System` is created in the device configuration. This device is listed in the list display on the right but not in the tree display on the left.

   The device is also displayed in the Symbol Explorer.

Example 1: Output of system information in the script

```
Write("%s",System.'Windows.User.UserDisplayName' );
Write("%s",System.'Windows.User.UserLoginName' );
```

Example 2: In the measurement file name, you can use a macro that evaluates a script. The script could then appear as follows:

```
char buffer[100];
SPrint(buffer, System.'Windows.User.UserDisplayName');
SetScriptResult ("%s", buffer);
```

6 Addresses

Vector knowledge base	Brief articles about various questions can be found in the Vector Knowledge Base at: http://www.vector.com/vi_knowledgebase_en.html
Addresses on Vector homepage	Please find the contacts of Vector Informatik GmbH and all subsidiaries worldwide via: http://www.vector.com/vi_addresses_en.html

7 Glossary

Argument	When a function is called, (command line) arguments can be passed to the function.
Array	An array is a field of variables of the same type.
ASCII	ASCII is the acronym for the **A**merican **S**tandard **C**ode for Information Interchange. It represents the standard method for coding alphabetic, numeric, and control characters in 7-bit-form.
CASL	CANape uses its own scripting language, also referred to as CASL (**C**alculation **a**nd **S**cripting **L**anguage).
Debugging	Diagnosing and locating of logic errors
Function	A function is started when triggered by events and is processed synchronously during a measurement.
Function parameter	Arguments and in/out parameters are also generally referred to simply as function parameters.
Global variable	Global variables are special data objects that can be used by all functions and scripts in the CANape configuration.
Compiling	A compiler compiles human-readable source text into the machine language understood by the computer. Syntax errors are identified during compilation.
Control structure	The processing sequence of statements is referred to as the control flow. A statement that influences this control flow is therefore also called a control structure.
Local variable	Local variables are only valid within the respective function or script.
MDF file	MDF stands for **M**easurement **D**ata **F**ormat. The MDF format is a binary file format for saving measurement data. CANape saves the measured data in this format. For this reason, the term "measurement file" is often used, which is synonymous with MDF file.
Measurement function	A measurement function is understood to be the combination of the function and the measurement parameters. Measurement functions are displayed under **Measurement signal|Functions** in the tree view of the measurement configuration.
Parameter types	Various types of parameters are available for passing data in the syntax of the functions.
Precompiler	Preprocessor
Script	A script can be started independently of an active measurement. It can be called externally and is executed in parallel with a measurement.
In/out parameter	When a function is called, in/out parameters can be passed to the function.
Virtual measurement file channel	A virtual measurement file channel is used to describe when a seemingly real (virtual) signal from measurement values of previously stored measurements is to be used.

8 Index

Get more Information!

Visit our Website for:

> News

> Products

> Demo Software

> Support

> Training Classes

> Addresses

www.vector.com