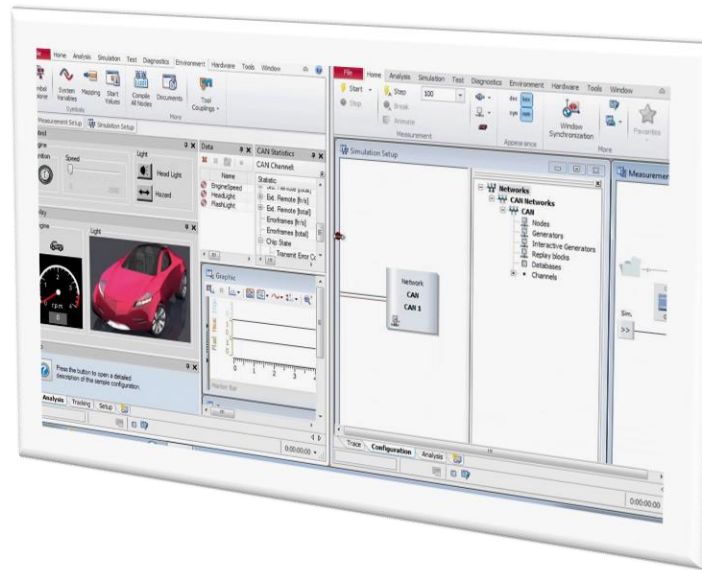


CANoe, CAPL Training

- by S.Thaminmun Ansari, Proprietor,
- Yahasiya Software

Agenda

- Working with CANoe tool and brief discussion of features



Vector CANoe

CANoe is a universal development,
test and analysis environment for
CAN bus systems,

Role of CANoe In Development Process

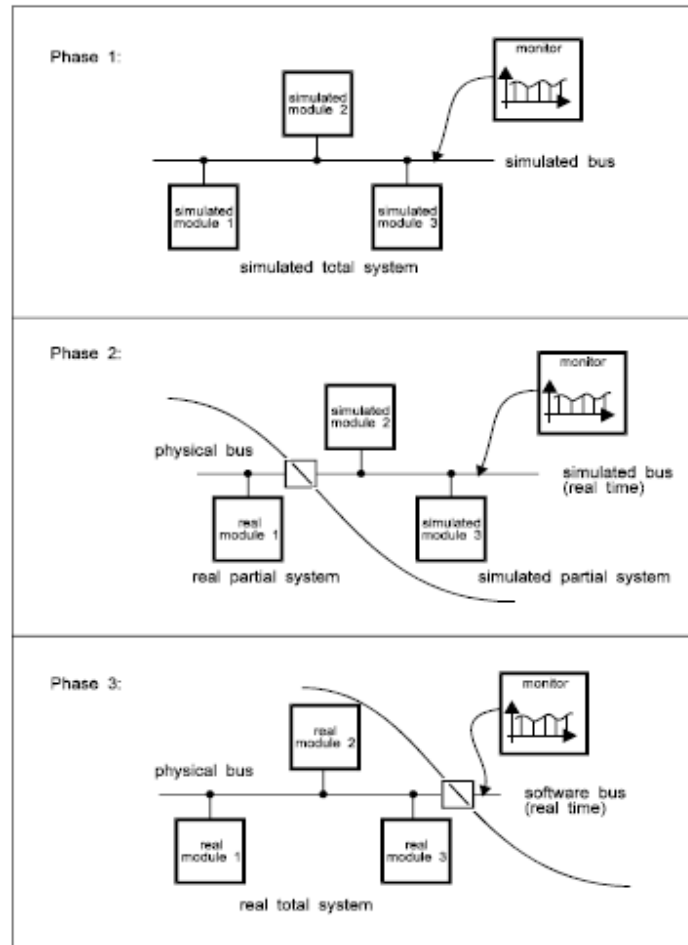


Figure 1 : Phase model of the development process

Tips for using CANoe

- In order to work with CANoe, we need a hardware from “Vector” which include CAN transceiver called as CAN Card, CANcase XL or VN1610 CAN interface
- To use CANoe knowledge of CAN is needed
- Demo version does not need CAN hardware and can not interact with external CAN network

CANoe Vs CANalyzer

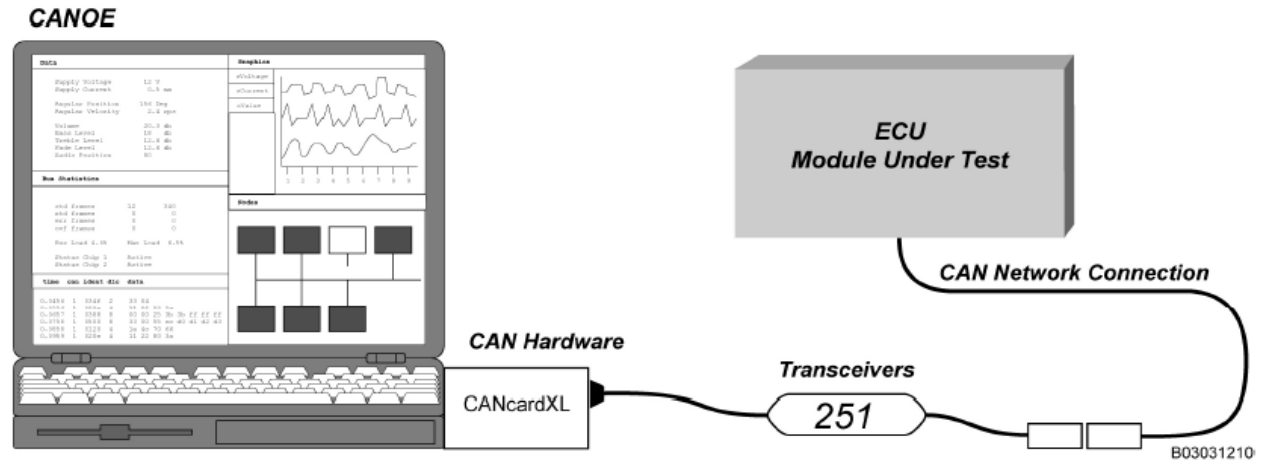
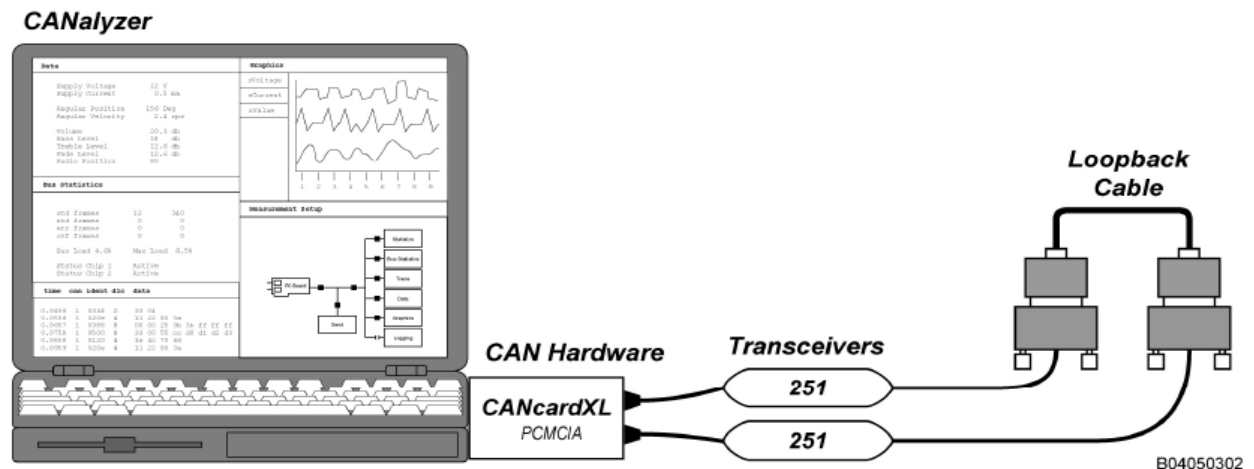


Figure 5 – Using CANoe to Simulate the Rest of the System

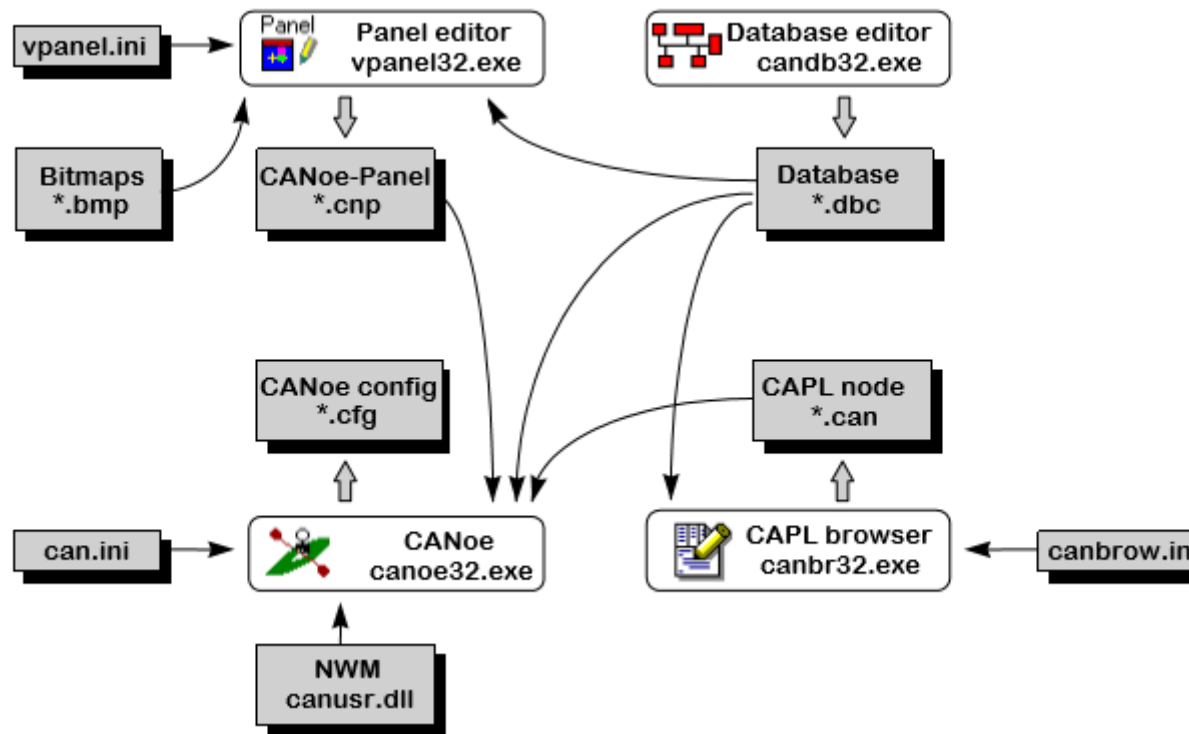


Why does the CANoe Need a
hardware to work in real
environment?

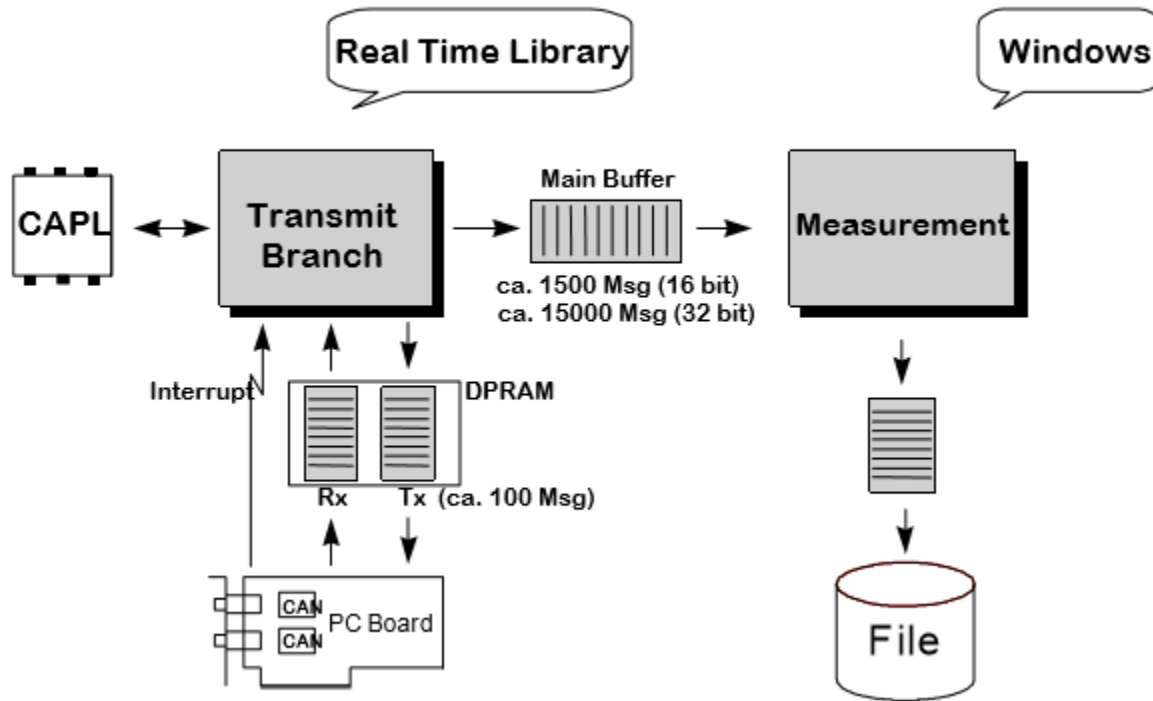
Any alternative?

A blue geometric graphic consisting of several parallel diagonal lines, forming a triangular shape in the bottom-left corner of the slide.

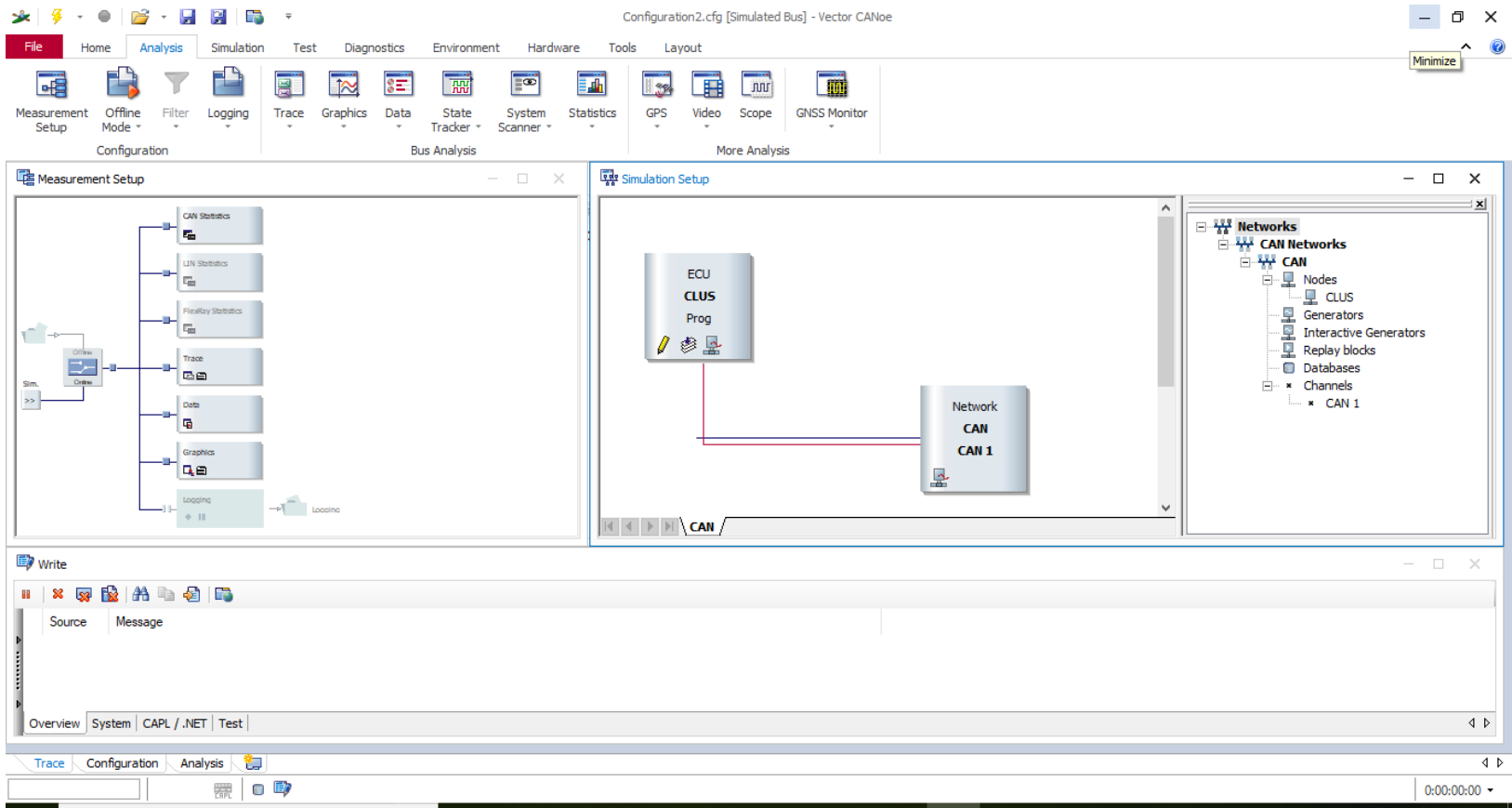
CANoe System Overview



CANoe internal structure



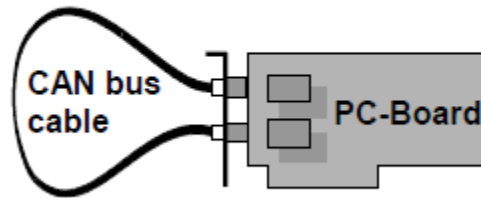
CANoe Overview-Simulation & Measurement Setup



Simulation setup

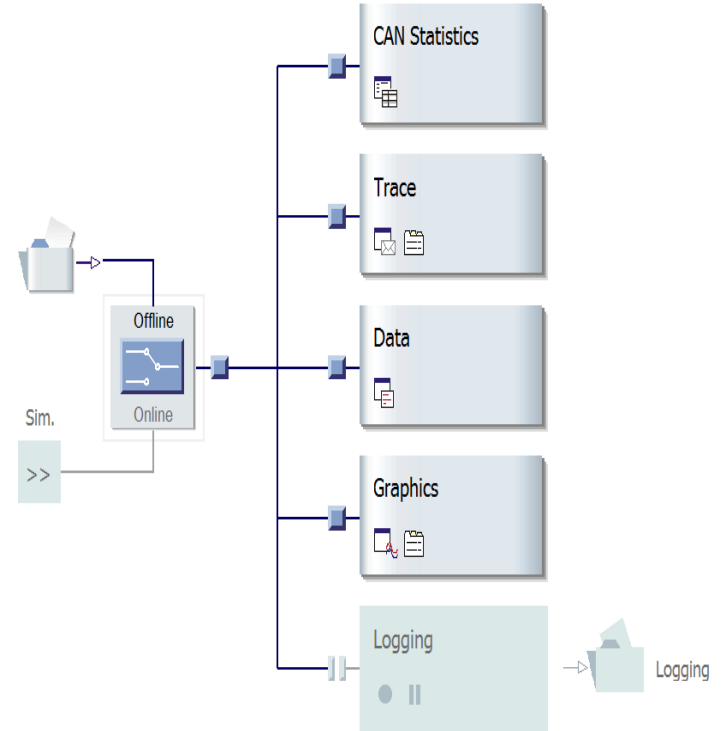
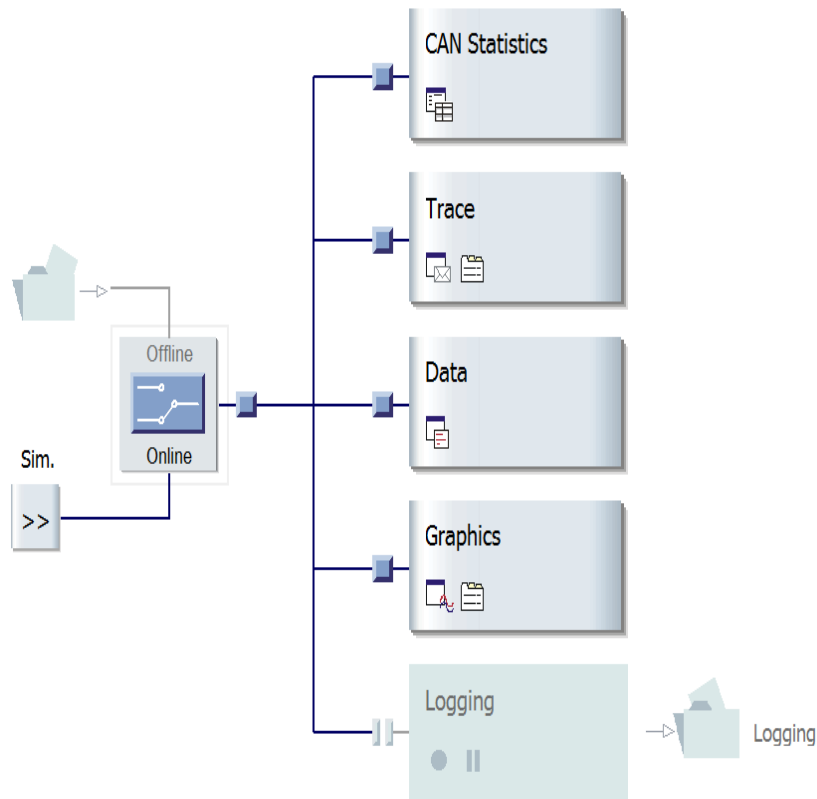
- Configuration of Simulation Setup
 - Configuring Networks
 - Configuring Network Nodes
 - Associating Databases
 - Associating Network hardware
- Layout of the Simulation Setup
 - Network Explorer
 - Listing of Nodes, IG, Replay blocks, Databases and Channels

Testing CAN hardware with loop back cable



Measurement Setup

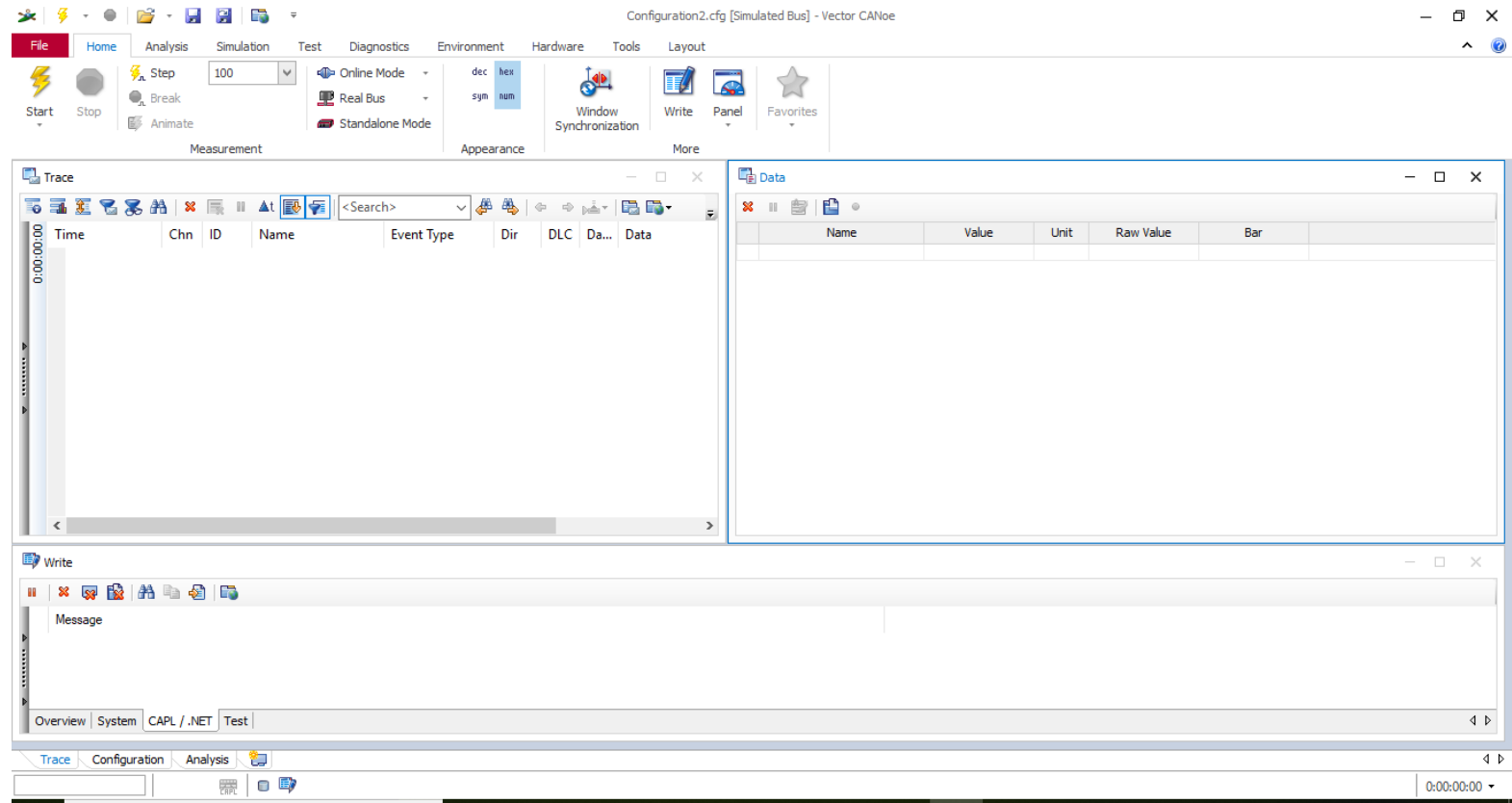
- Data flow in measurement setup



Measurement Setup

- Configuration of Measurement setup
- Working with Evaluation block in measurement setup
- Message attributes
- Simulation mode

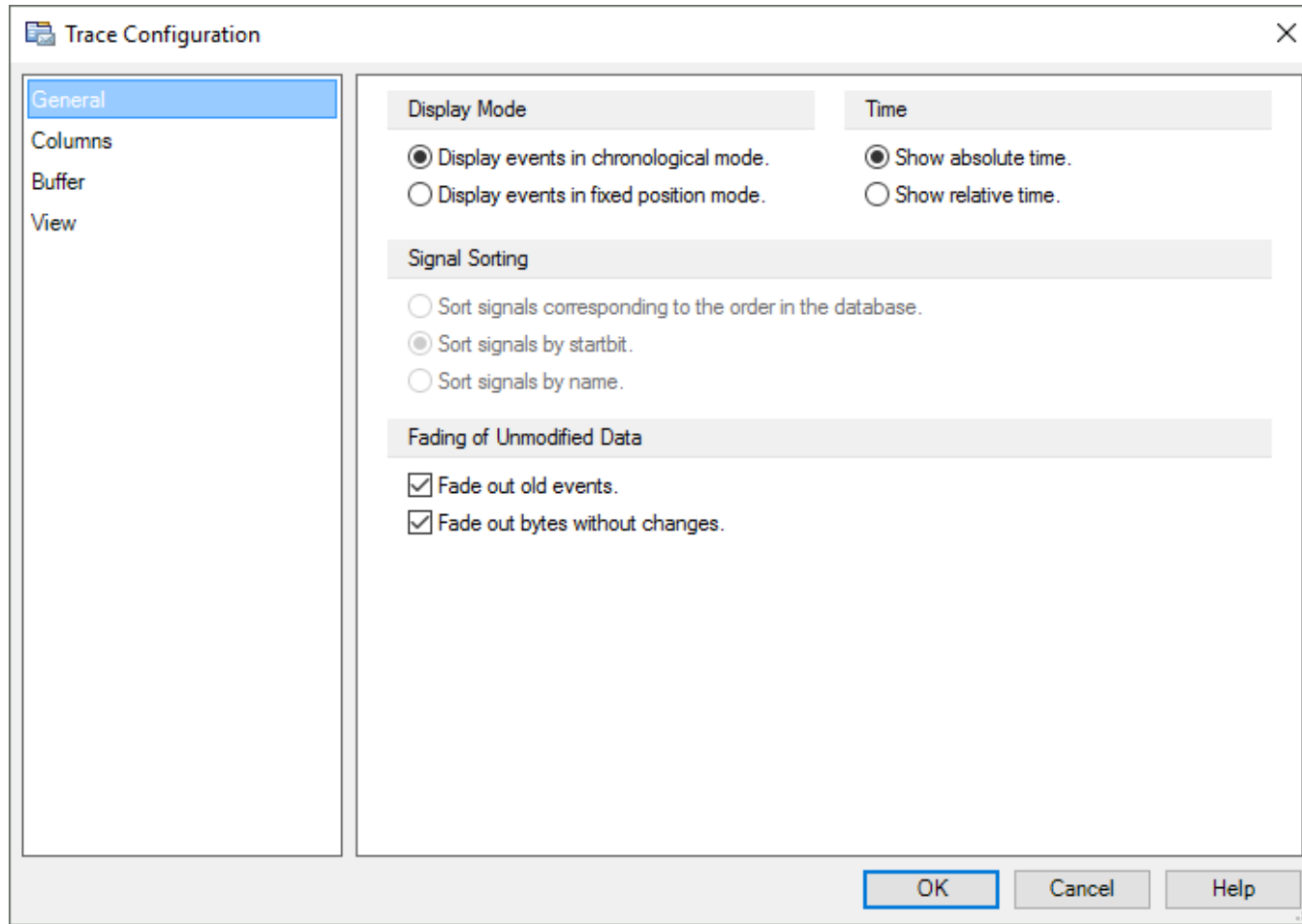
CANoe Overview – Trace, Write, Data & Graphics window



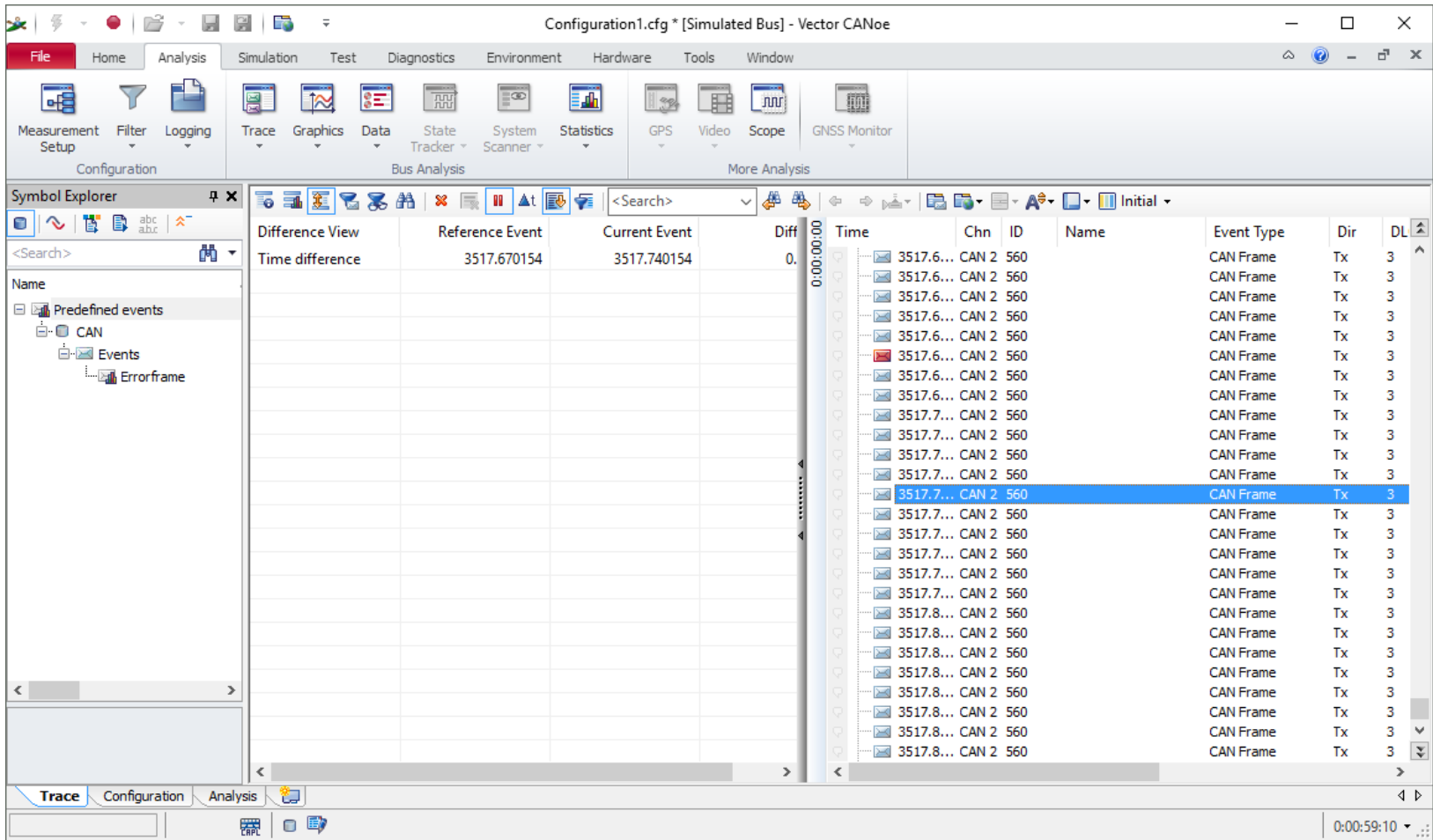
Trace window

- Standard configuration of trace window
- Configuration of columns in trace window
- Trace window options from tool bar
- Detail view

Evaluating CAN messages in Trace window



Difference and Detailed views



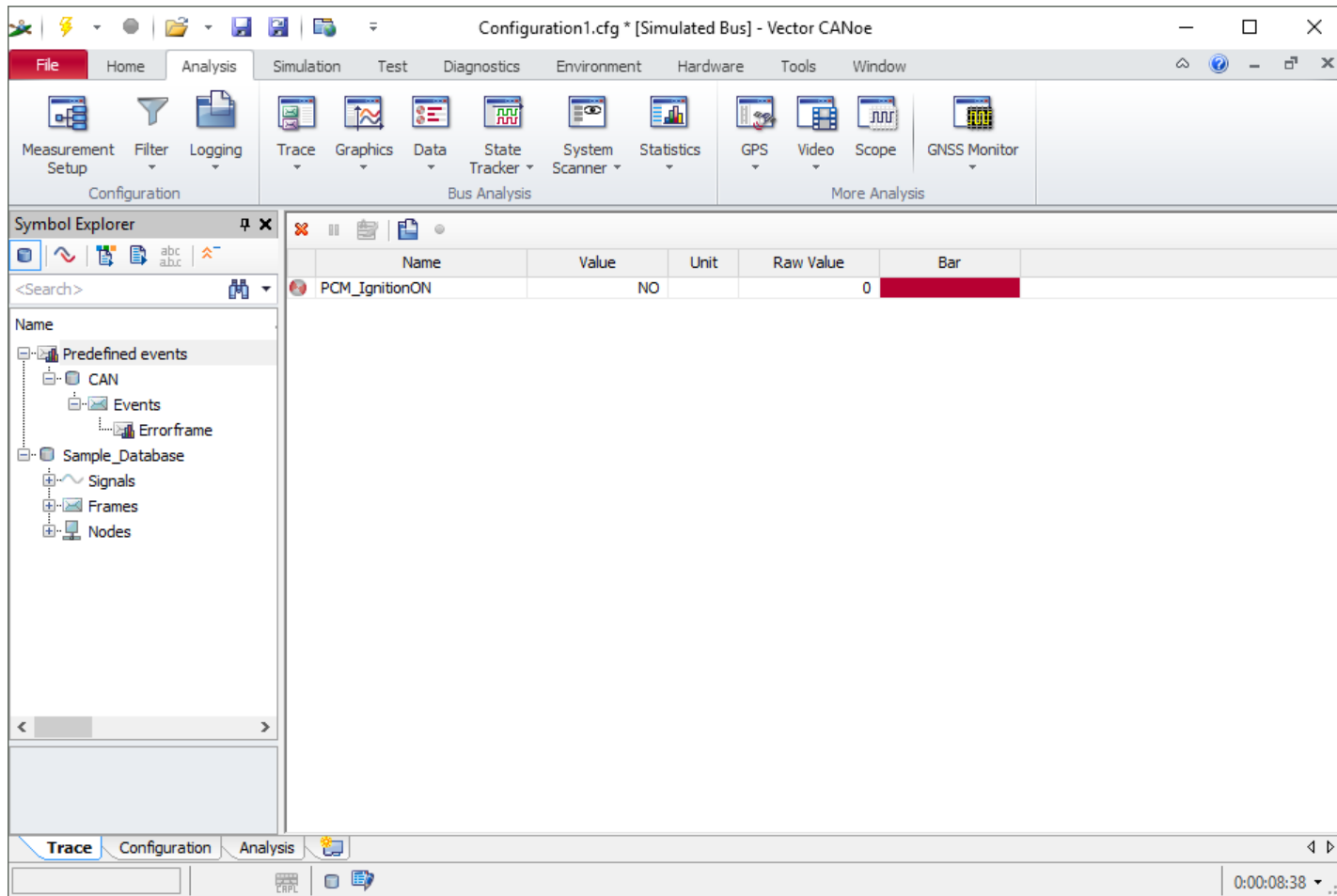
Data Window

- Configuration of signal
- Display types
- Activity Indicator

Analysis of Signal Values in the Data Window

- The purpose of the Data window is to assist in the study of momentary signal values.
- The signals values to be displayed are exclusively dependent upon information from the database

Example of data analysis using Data window



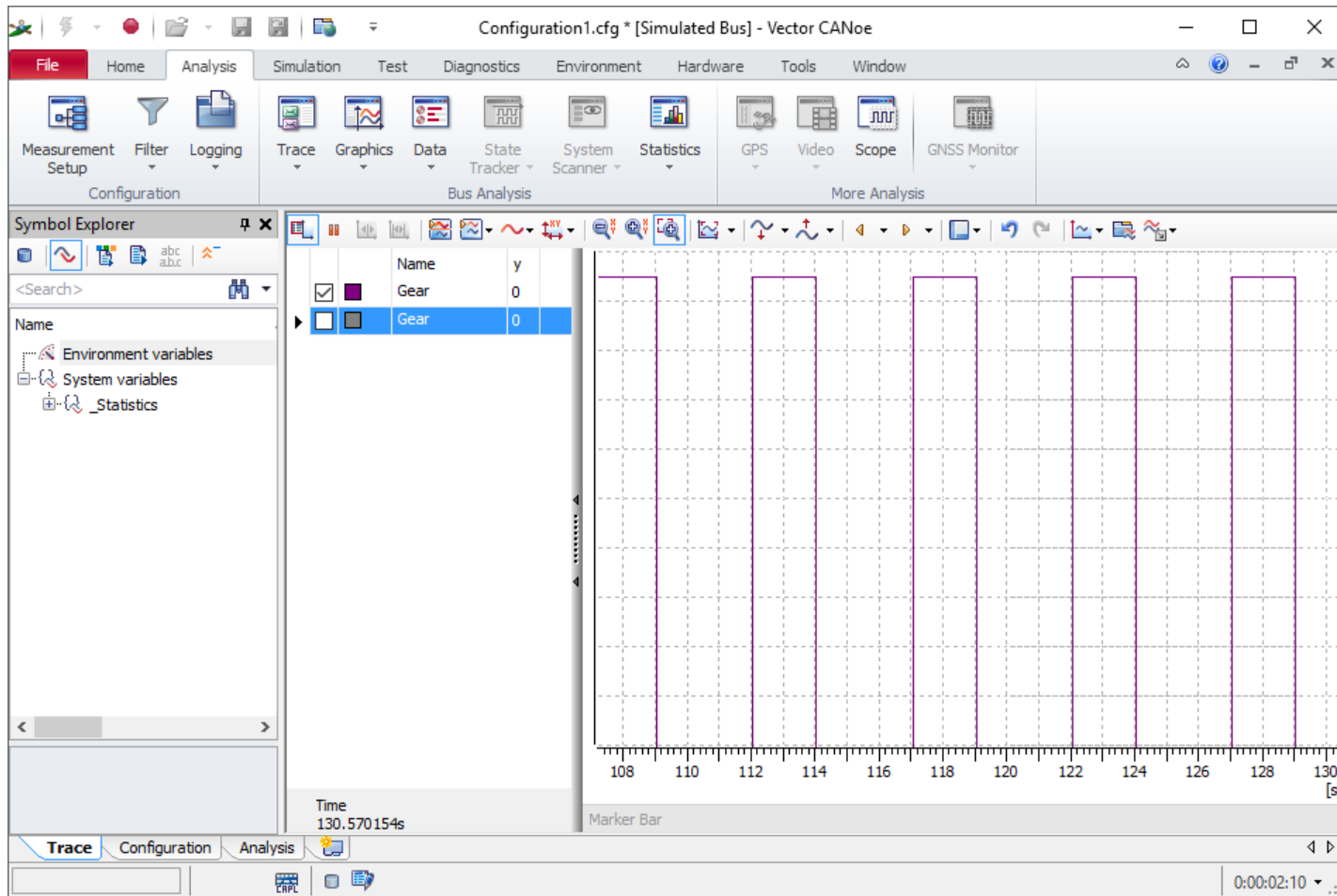
Graphics window

- Selecting signal
- Arrangement of signal
- Signal layout
- Measurement mode point
- Measurement mode difference
- Display modes
- Layout functions
- Copy screen shot

Analyzing Data using graphics window

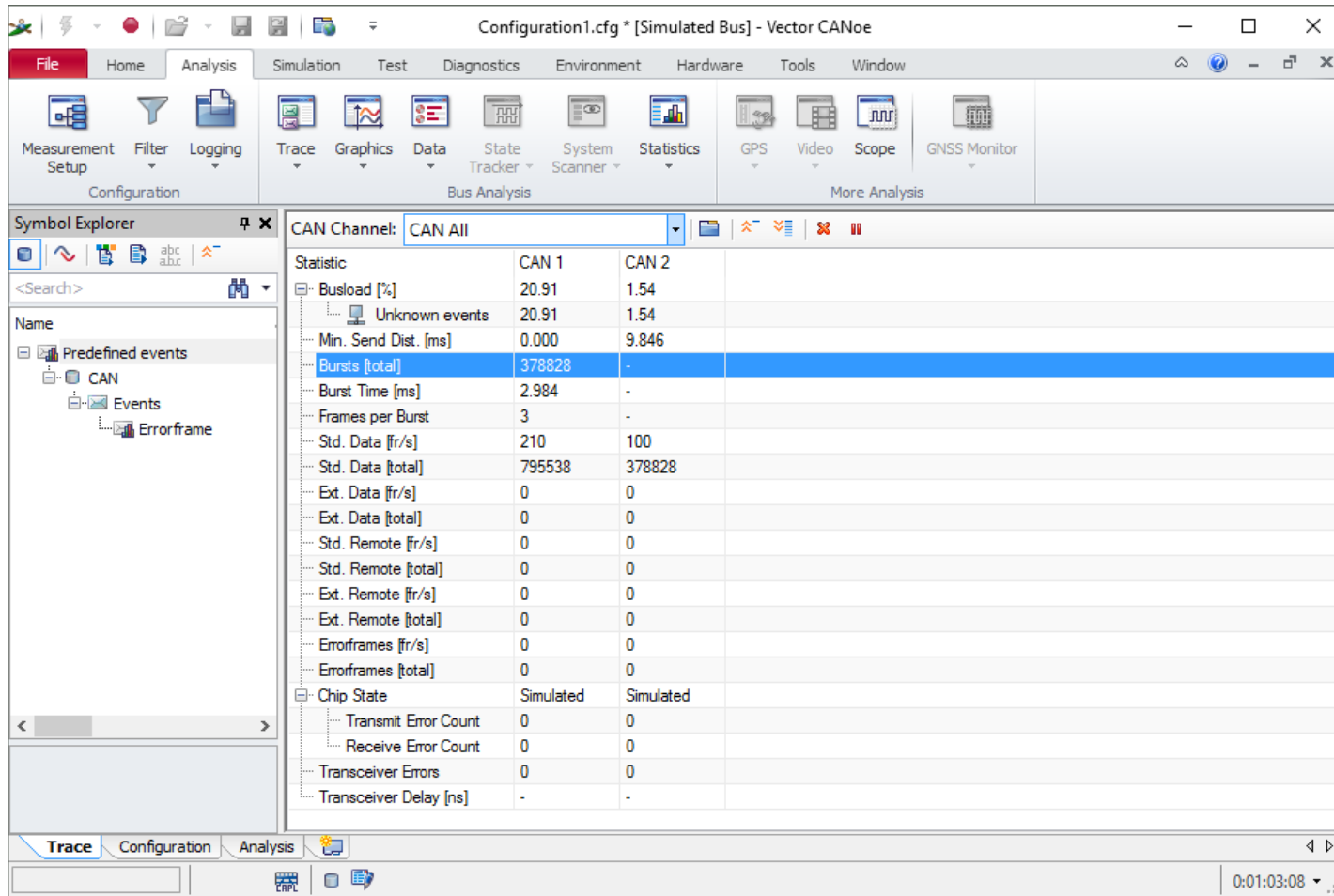
- While the Data window displays momentary signal values, you can have the time responses of signal values displayed in the Graphics window. After the end of measurement the signal responses are available for study by user-friendly analysis functions

Example of analysis in Graphics window



Write window

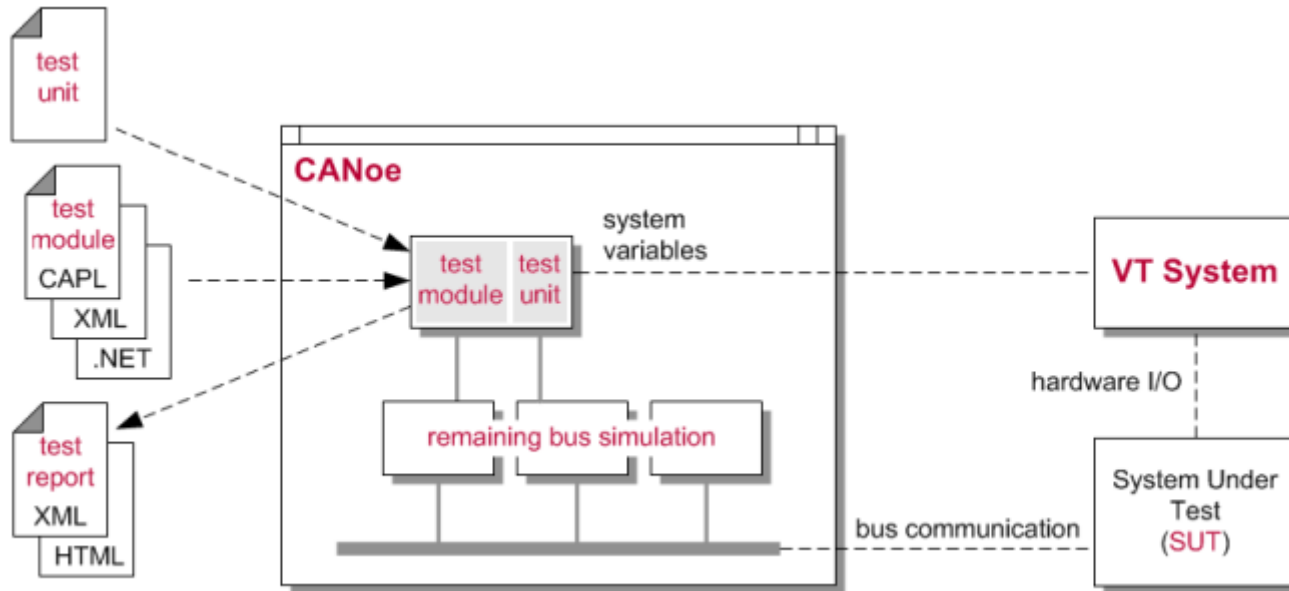
Statistic Window



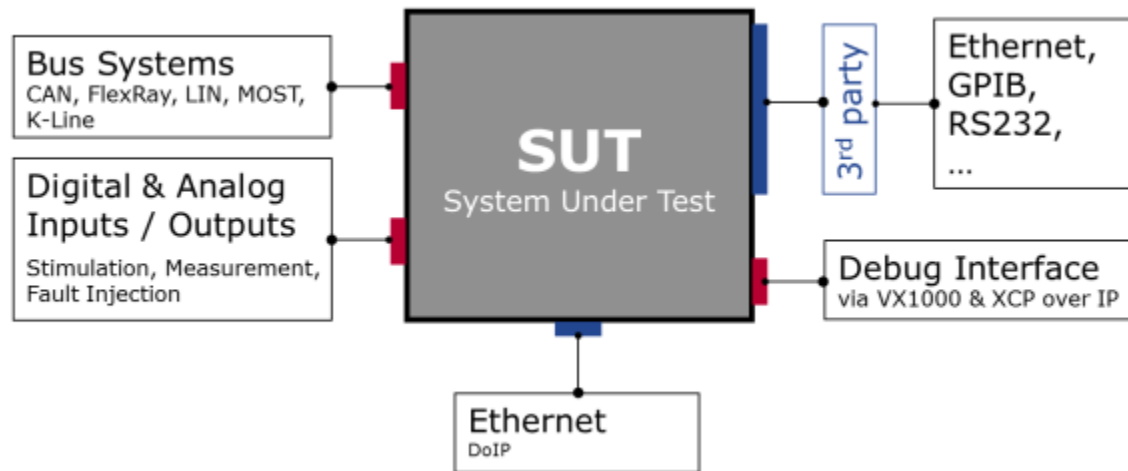
The screenshot shows the Vector CANoe interface with the 'Statistics' window open. The window title is 'Configuration1.cfg * [Simulated Bus] - Vector CANoe'. The 'CAN Channel' is set to 'CAN All'. The 'Statistic' table displays various metrics for CAN 1 and CAN 2. The 'Bursts [total]' row is highlighted in blue.

Statistic	CAN 1	CAN 2
Busload [%]	20.91	1.54
Unknown events	20.91	1.54
Min. Send Dist. [ms]	0.000	9.846
Bursts [total]	378828	-
Burst Time [ms]	2.984	-
Frames per Burst	3	-
Std. Data [fr/s]	210	100
Std. Data [total]	795538	378828
Ext. Data [fr/s]	0	0
Ext. Data [total]	0	0
Std. Remote [fr/s]	0	0
Std. Remote [total]	0	0
Ext. Remote [fr/s]	0	0
Ext. Remote [total]	0	0
Errorframes [fr/s]	0	0
Errorframes [total]	0	0
Chip State	Simulated	Simulated
Transmit Error Count	0	0
Receive Error Count	0	0
Transceiver Errors	0	0
Transceiver Delay [ns]	-	-

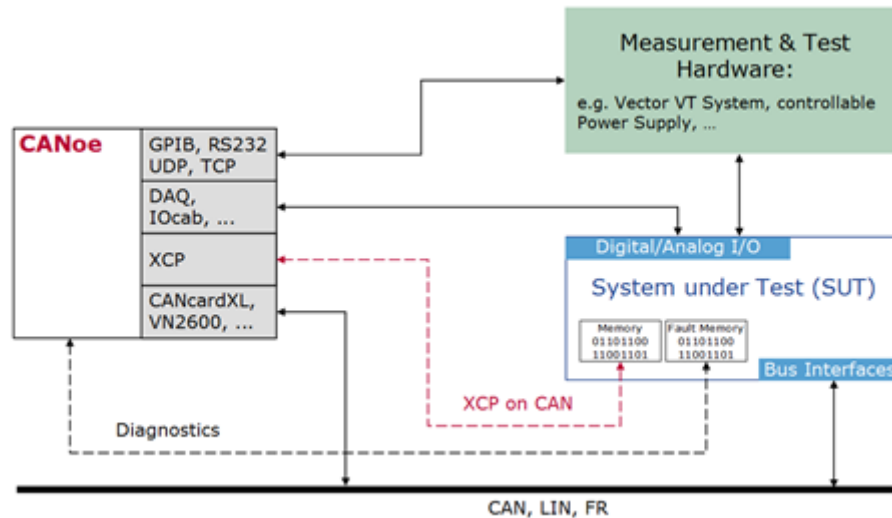
CANoe as testing tool



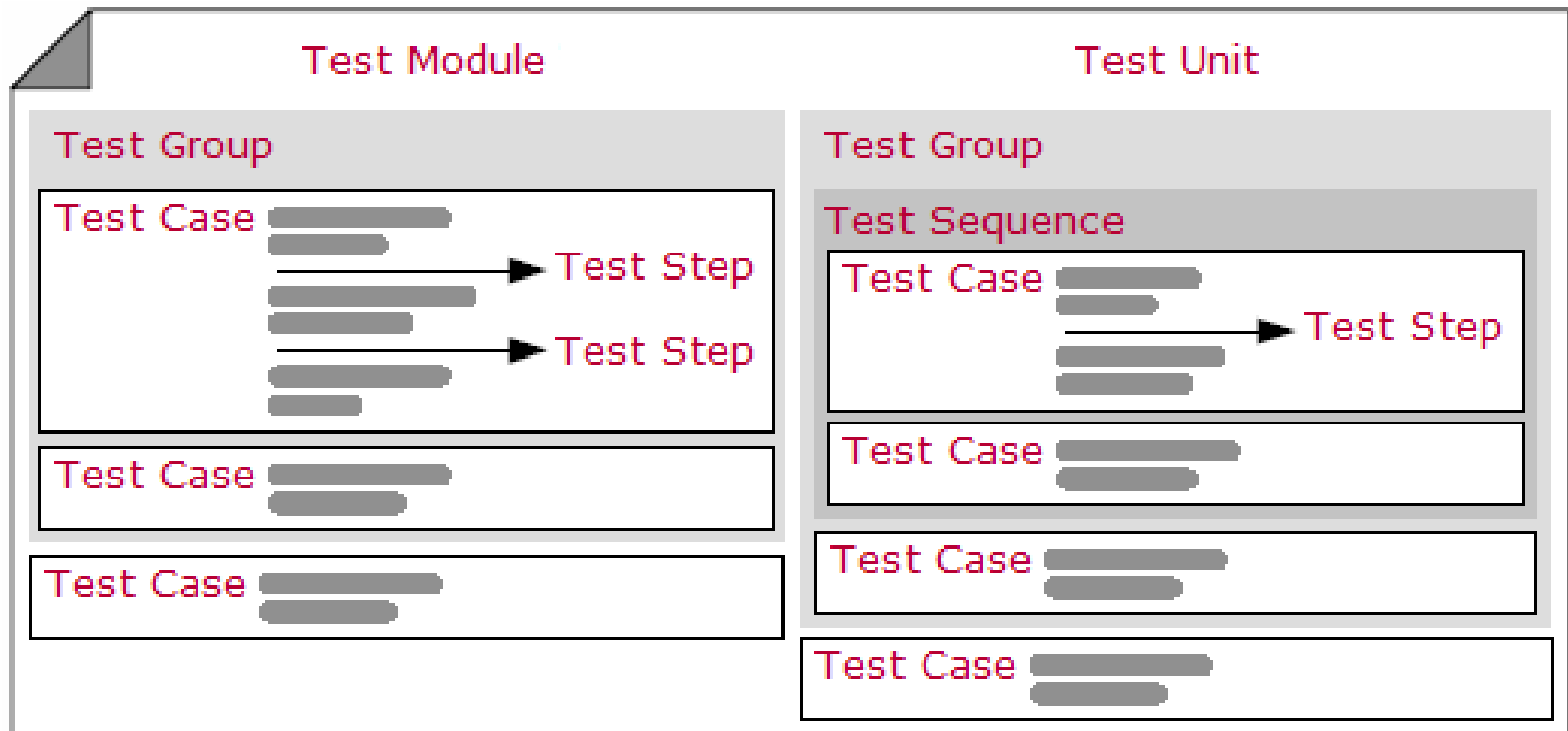
Schematic description of test integration



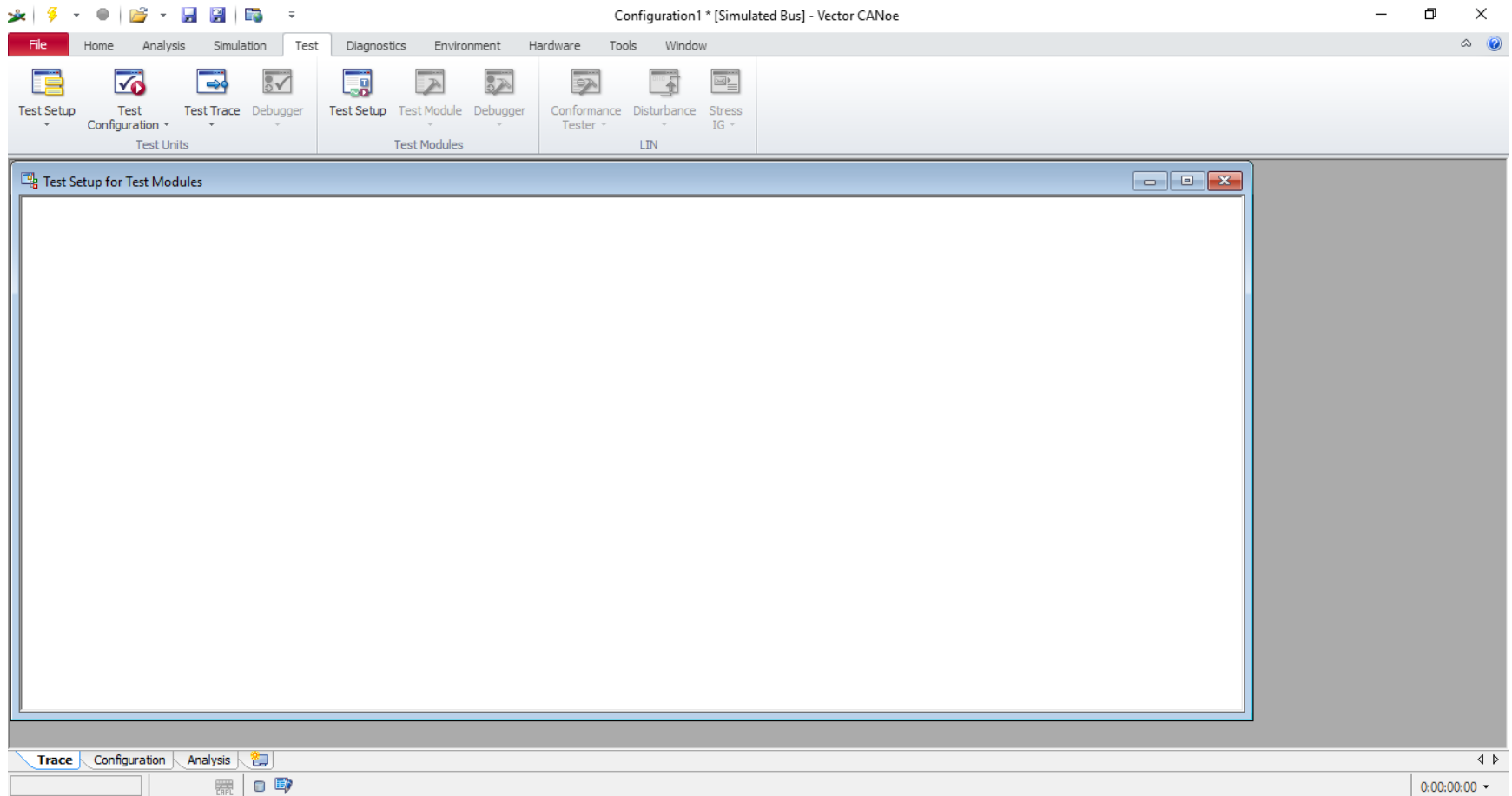
the connection between CANoe and the SUT



Test sequences in CANOe



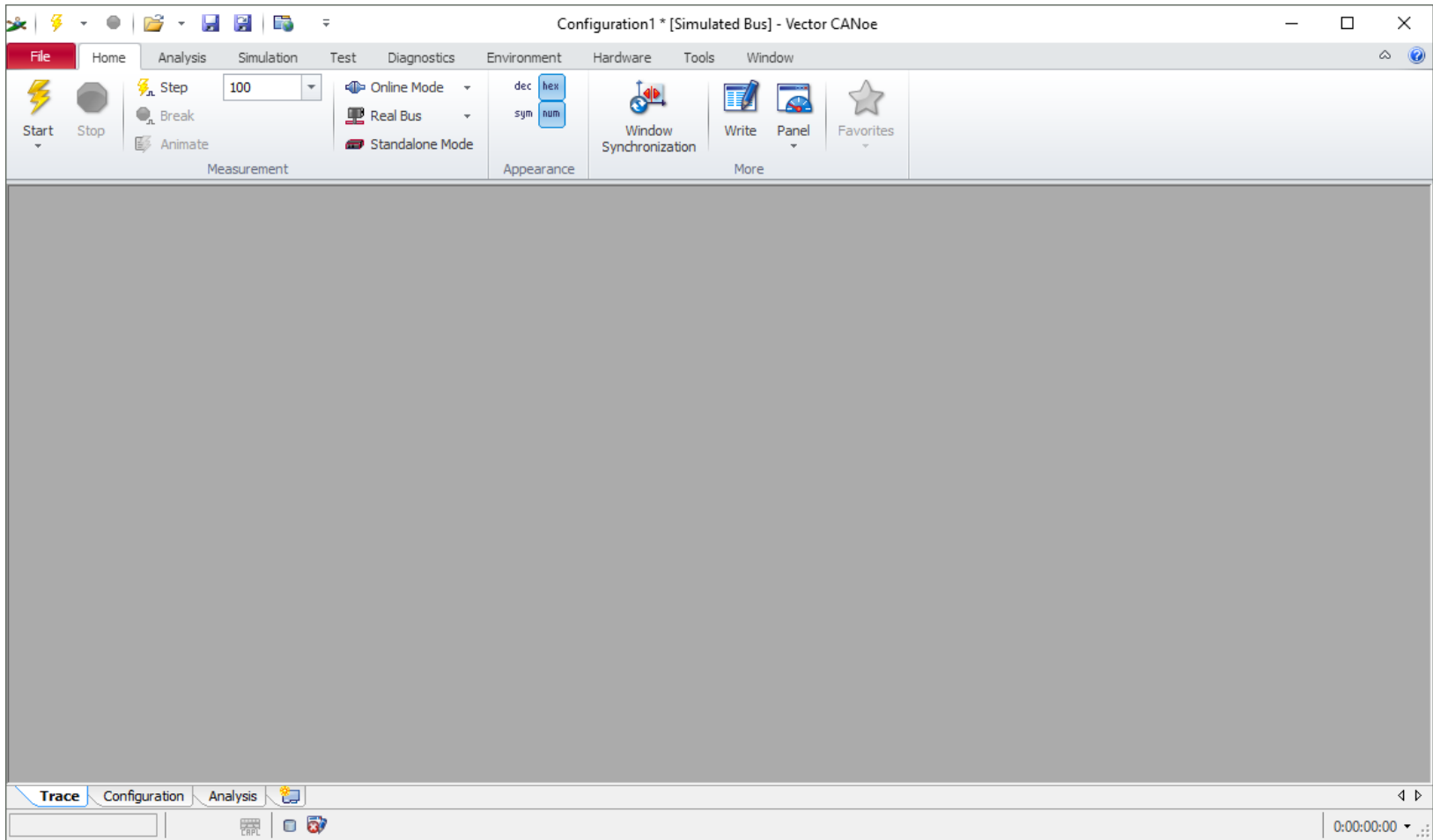
CANoe Overview - Test setup window



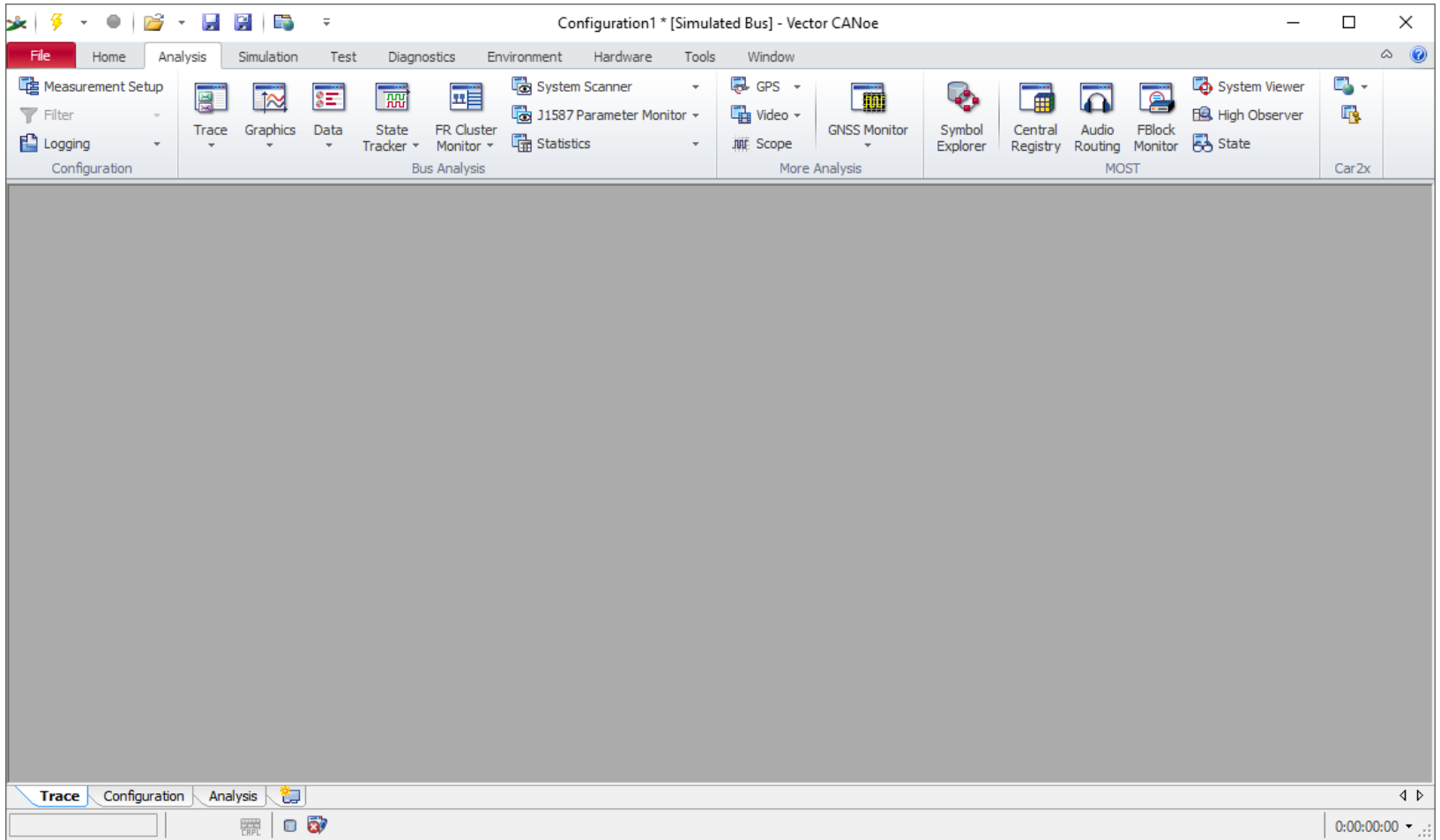
Test script Template for CAPL

```
void MainTest ()
{
    TestModuleTitle (“<Test Module Title>”);
    TestModuleDescription (“<Test Module Description>”);
    TestGroupBegin(“<Test Group Name>”, “<Test group Description>”);
    Testcase1();
    Testcase2();
    TestGroupEnd();
}
testcase Testcase1()
{
    // Test Steps...
}
```

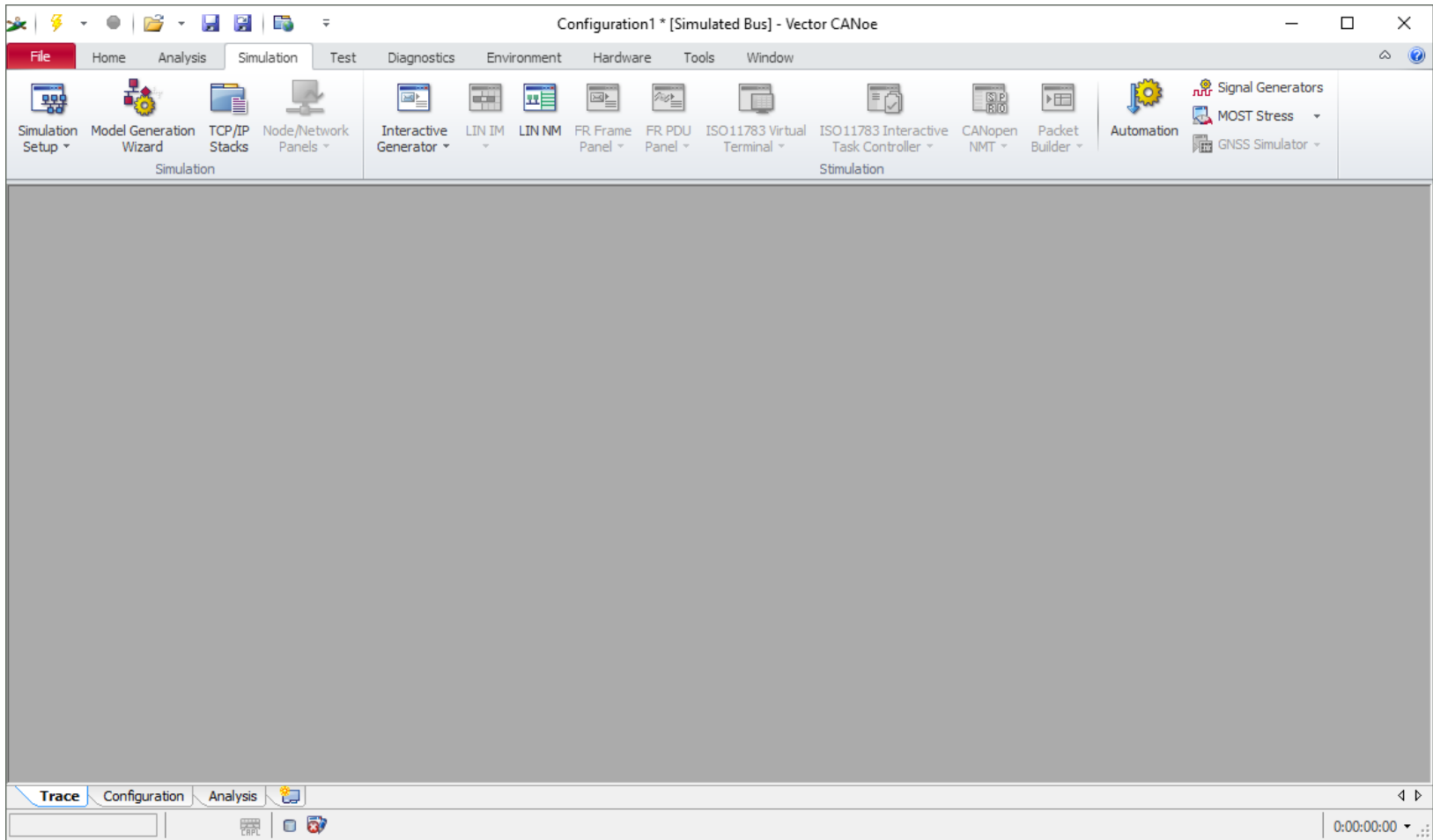

CANoe menu/Tab – home tab



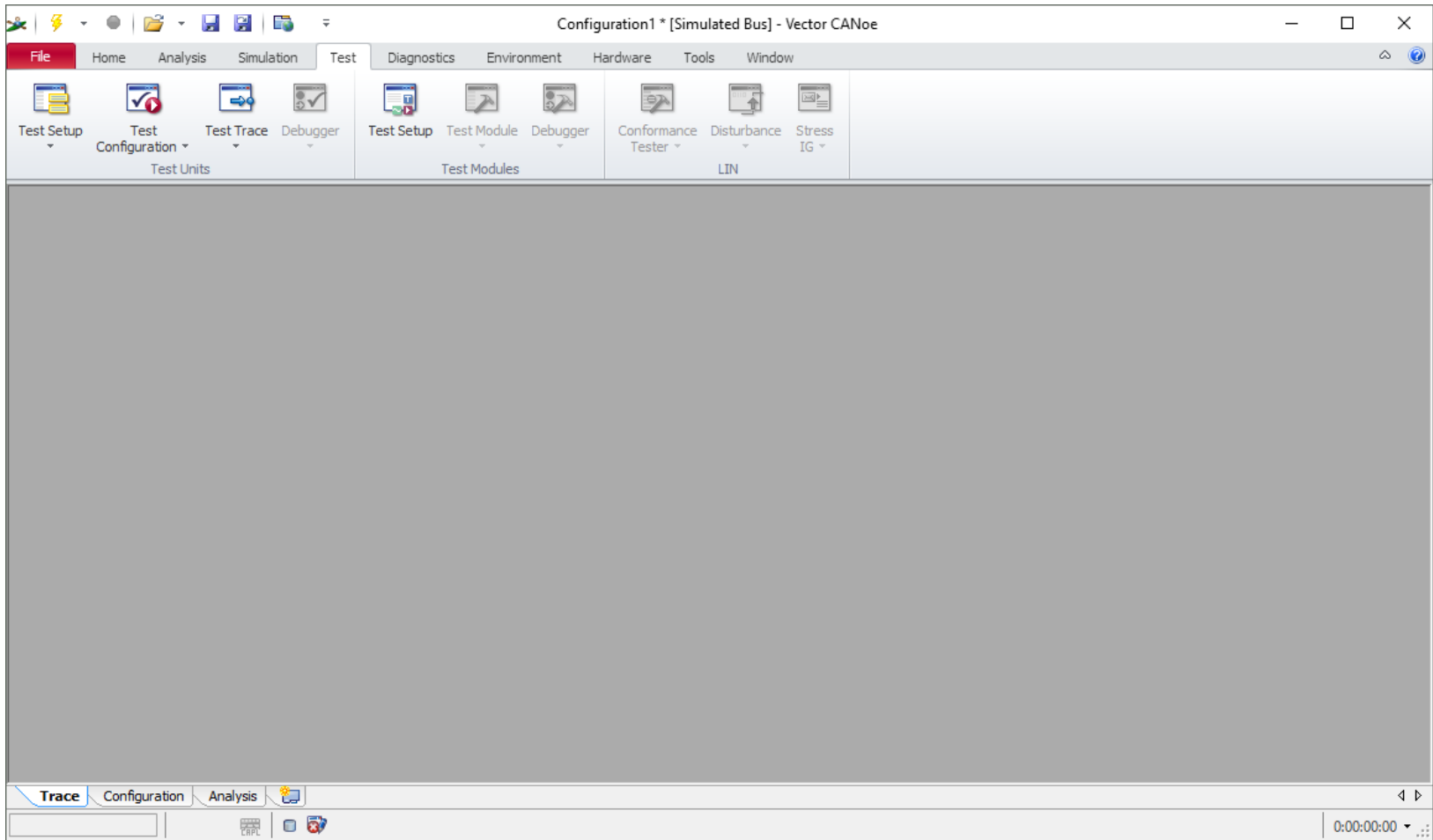
CANoe menu/Tab – Analysis tab



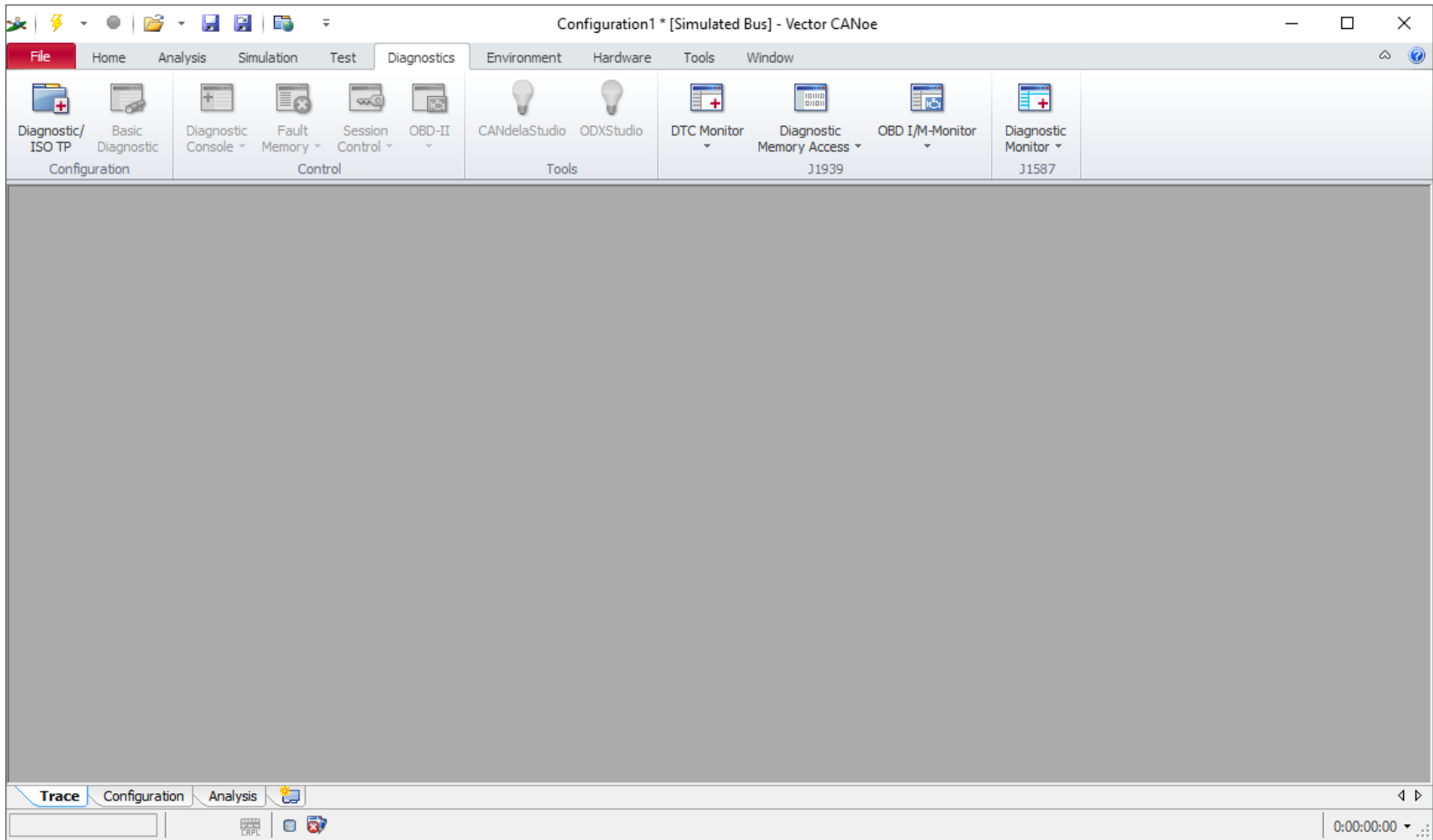
CANoe Menu/Tab – Simulation



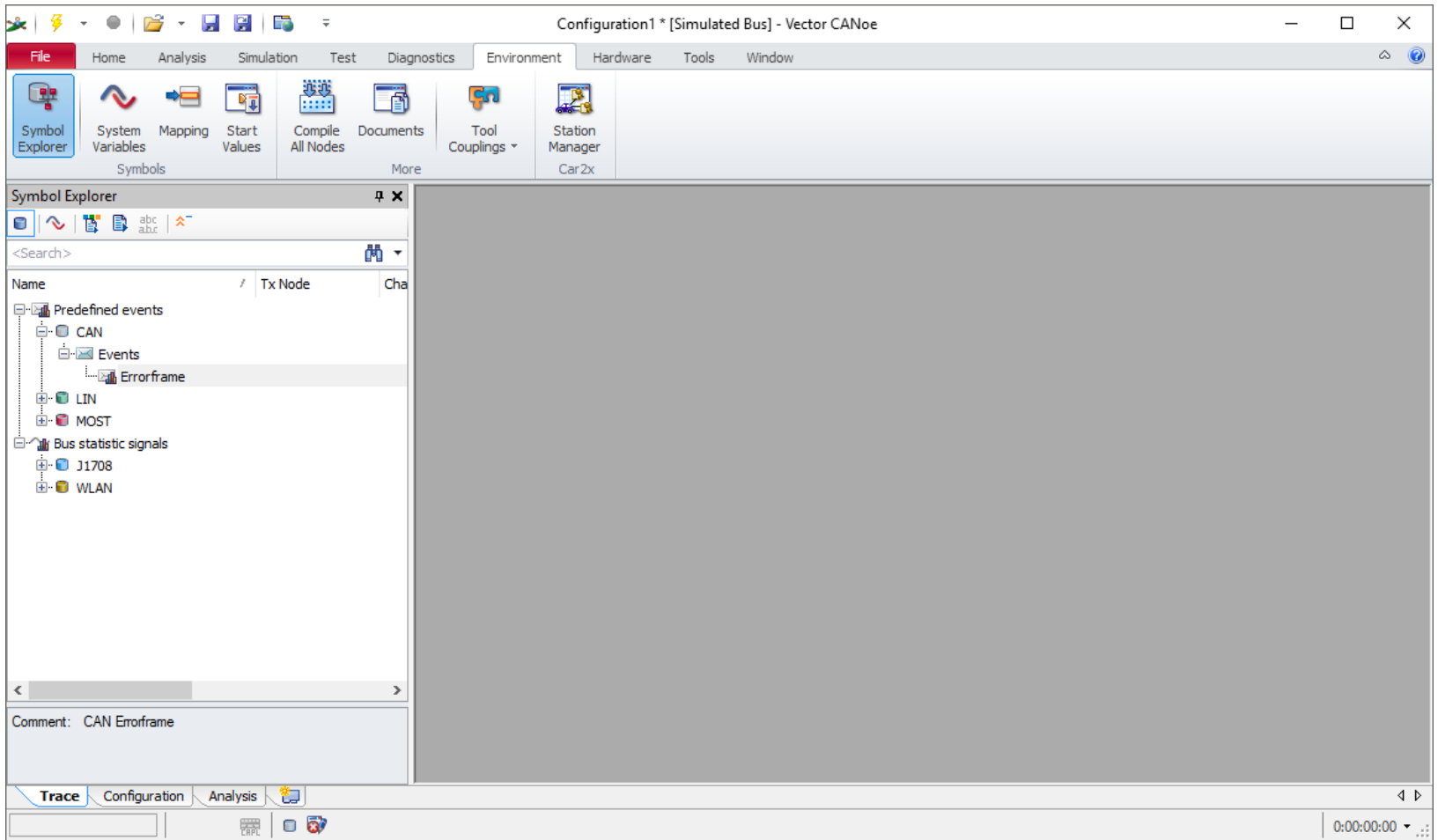
CANoe Menu/Tab – Test



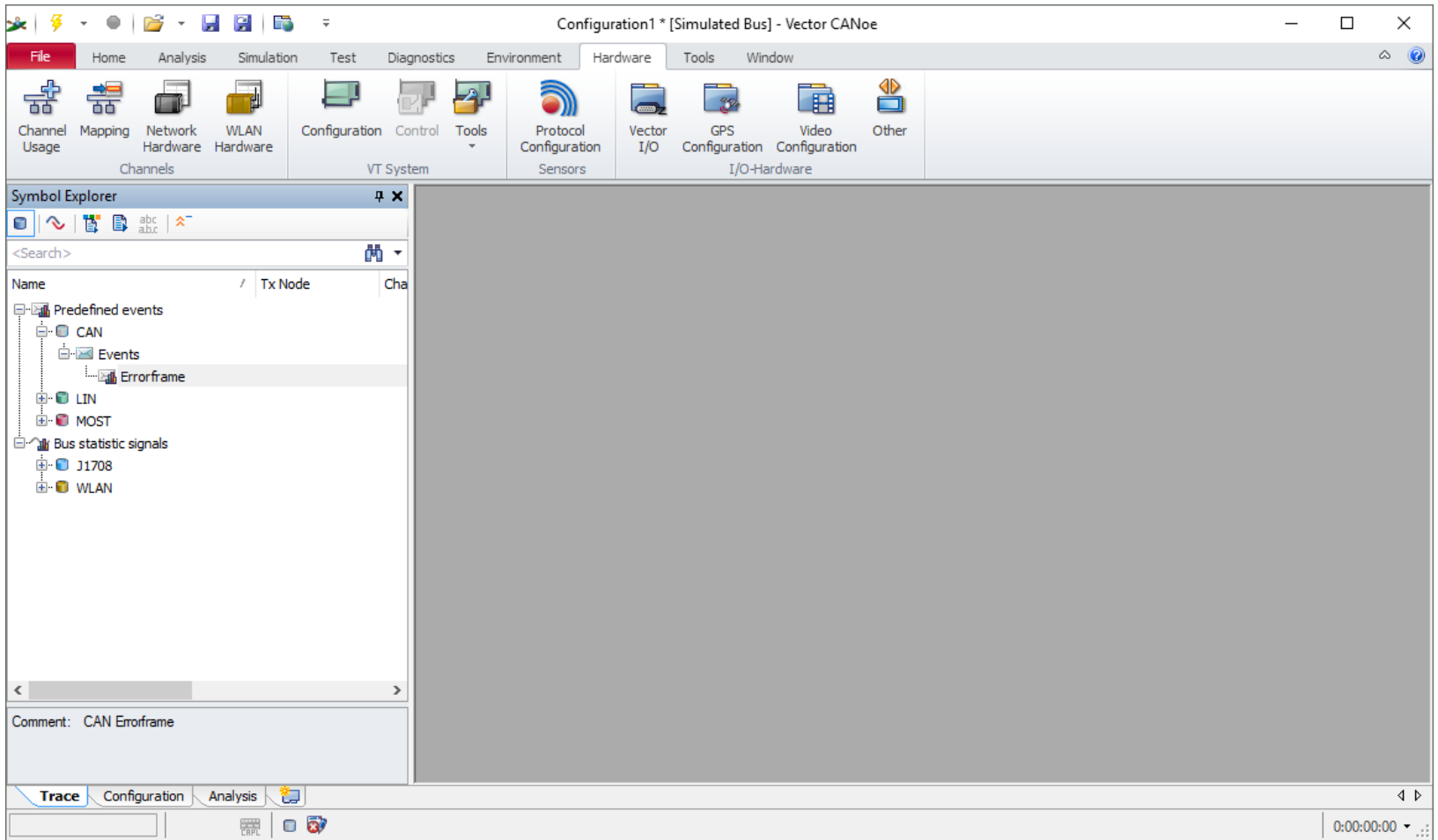
CANoe Menus/Tab Diagnostic



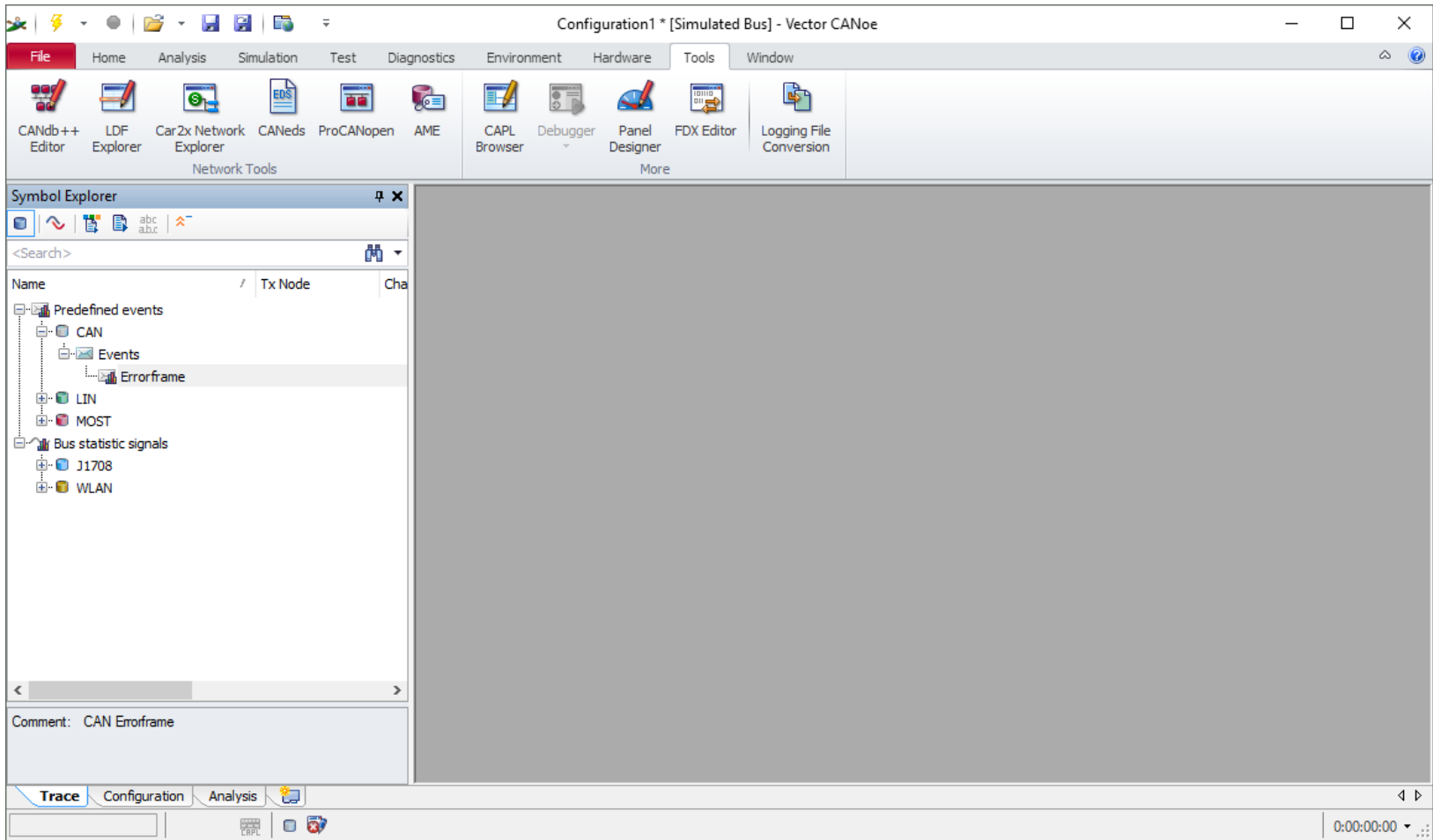
Menu/Tab Environment



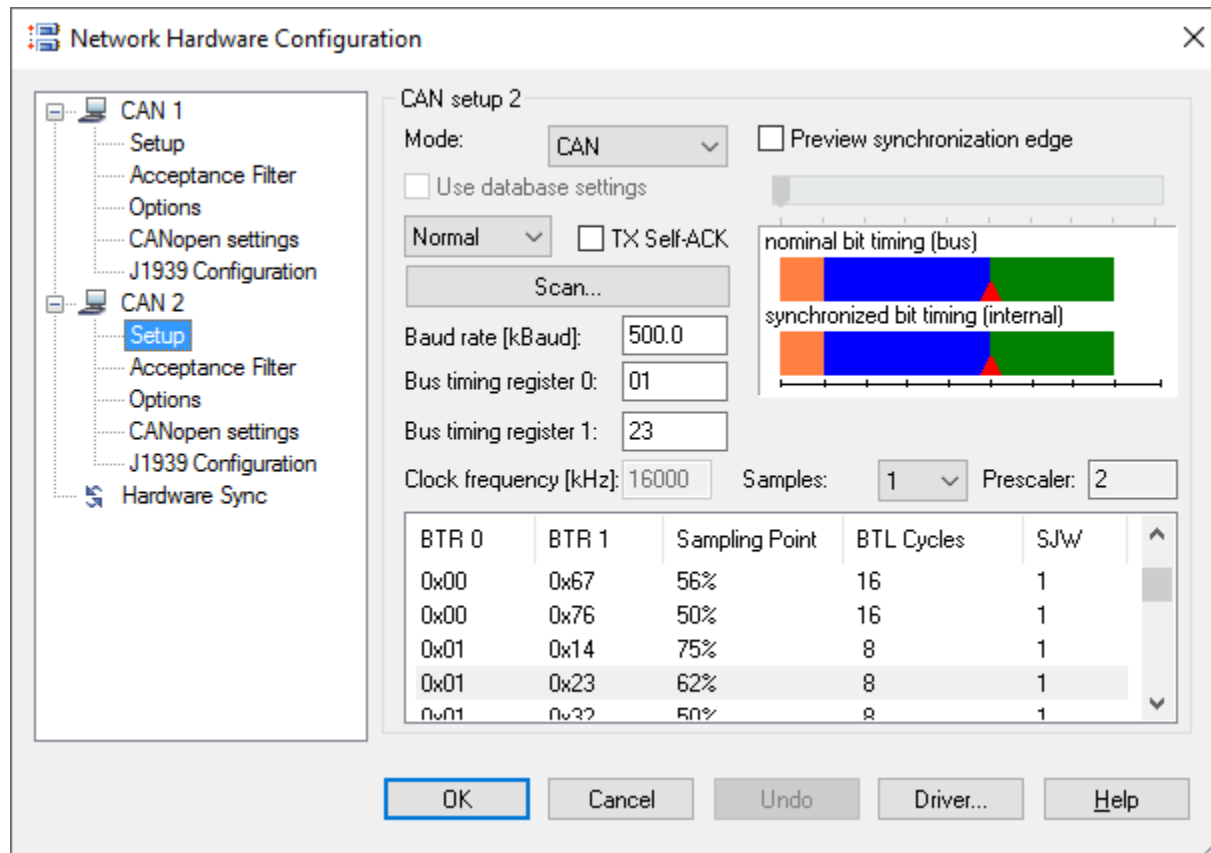
Menu/Tab hardware



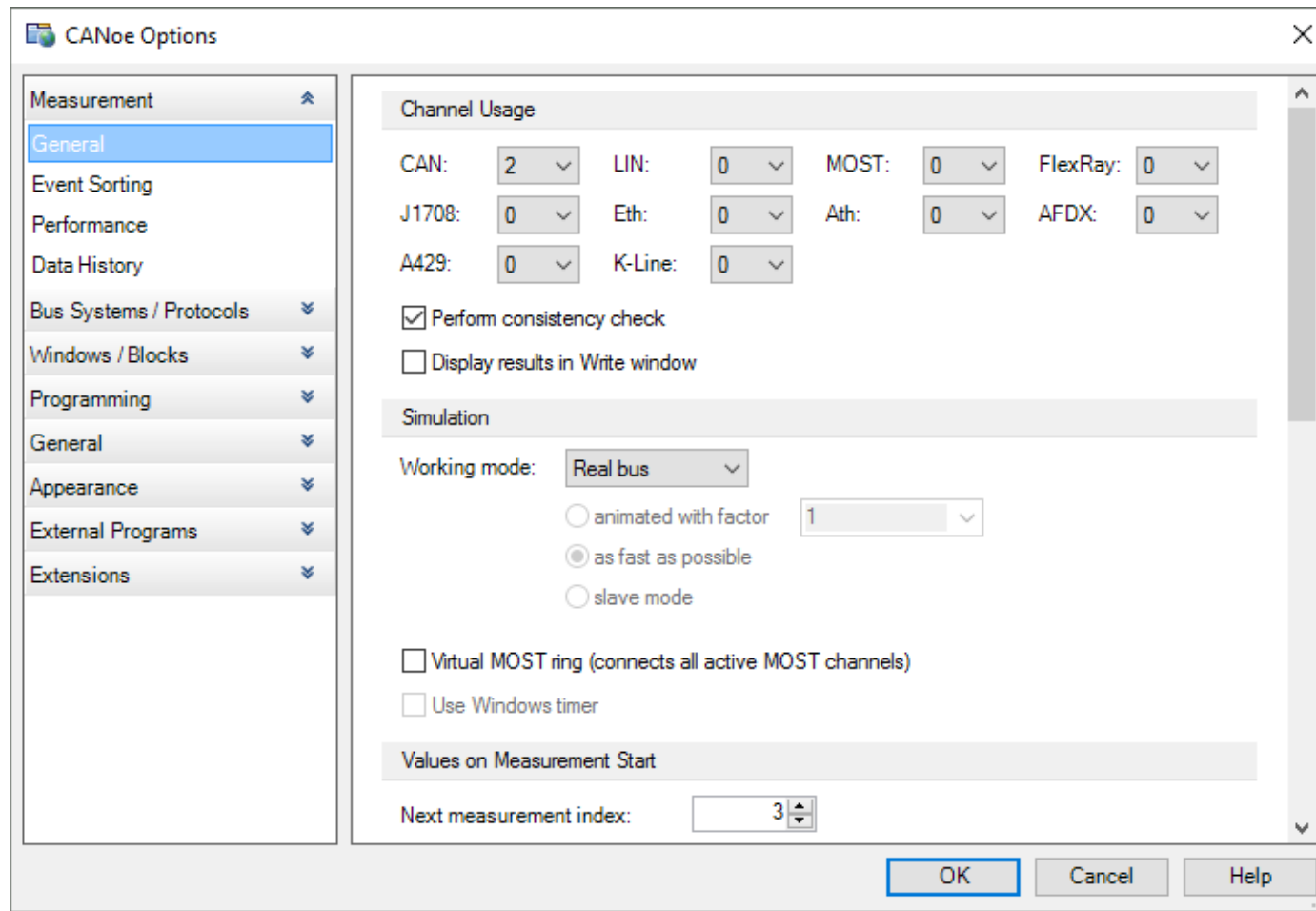
CANoe Menu /Tab Tools



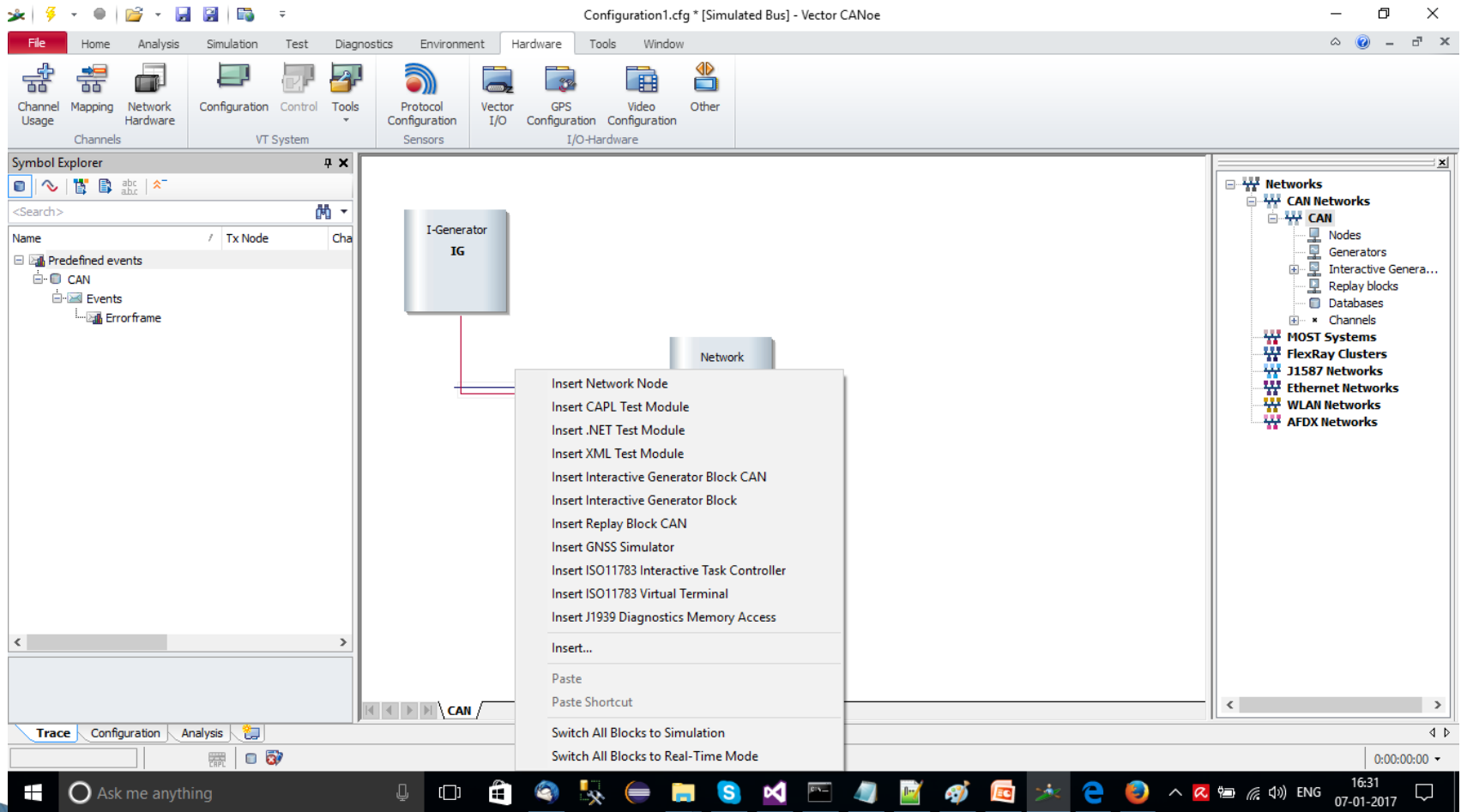
Setting up CAN bus



Channel usage Dialog



Simulation Setup & Context menus



IG Block

- IG block creates a Data source for CAN bus
- User without any programming experience can encode and transmit a CAN message in a Channel
- IG block shall transmit message periodically and event basis
- is enabled and disabled by space bar

IG Block

- Configuration of triggering
- Configuration of transmit list
- Entry of signal values
- Function generator for transmit list
- Keyboard control

IG block configuration

The screenshot shows the Vector CANoe software interface. The main window is titled "Configuration2.cfg * [Simulated Bus] - Vector CANoe". The "CAN IG" window is open, showing a table of CAN messages. The table has columns: Row, Send, Trigger, Name, ID, Channel, Type, and DLC. Two rows are visible: Row 1 with a periodic trigger of 200 ms for EngineData (ID 64, CAN 1, CAN Data, DLC 8) and Row 2 with a manual trigger on key 1 for ABSData (ID 201, CAN 1, CAN Data, DLC 1). Below the table is a "Signals" section with a "Raw Data" table. This table has columns: Name, Generator Control, Generator Type, Raw Value, Raw Step, Phys Value, Phys Step, Unit, Start Bit, and Length. The signals listed are EngSpeed, EngineTemp, IdleRunning, PetrolLevel, EngForce, and EnginePower, each with specific generator settings and physical values.

Row	Send	Trigger	Name	ID	Channel	Type	DLC
1		Periodic: 200 ms	EngineData	64	CAN 1	CAN Data	8
2		Manual, on key: 1	ABSData	201	CAN 1	CAN Data	1

Name	Generator Control	Generator Type	Raw Value	Raw Step	Phys Value	Phys Step	Unit	Start Bit	Length
EngSpeed	[Icon]	Sine	CCD	CCD	3277	400	rpm	0	16
EngineTemp	[Icon]	Ramps and ...	0	6	0	12		16	7
IdleRunning	[Icon]	None	0	1	0	1		23	1
PetrolLevel	[Icon]	None	0	D	0	13		24	8
EngForce	[Icon]	Random	0	CCD	0	3277		32	16
EnginePower	[Icon]	None	0	CCD	0	3277		48	16

Demo of Transmitting CAN messages using IG Block

Working with Symbolic Data

- signals such as RPM, temperature and engine load, which are provided by individual controllers, and are sent on the bus with the help of CAN messages.
- To describe this information symbolically, CANoe provides you with the database format DBC and a database editor with which you can read, create and modify CAN databases

Example of symbolic data

The screenshot shows the Vector CANdb++ Editor interface. The left sidebar displays a tree view of the database structure, including Networks, ECUs, Environment variables, Network nodes, ABS, BCM, PCM, Messages, and Signals. The 'Network nodes' node is selected and highlighted. The main pane displays a table of network nodes with the following data:

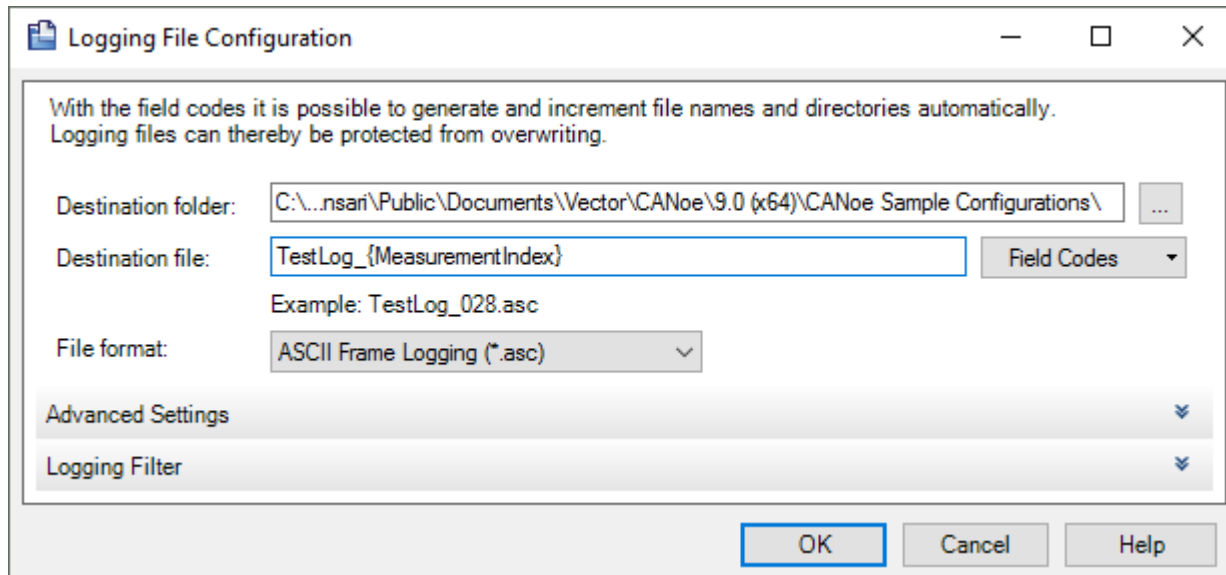
Name	Address	Comment	NosPackage	PackageID	DiagNo
ABS	0x2		I3_v1*	08.05.00.*	GGDS*
BCM	0x1		I3_v1*	08.05.00.*	GGDS*
PCM	0x0		I3_v1*	08.05.00.*	GGDS*

At the bottom of the window, it indicates '3 Node(s)' and 'Ready'.

Logging Measurement

- CANoe has extensive logging functions for data logging. In the standard measurement setup the logging branch is shown at the very bottom of the screen.
- You can easily recognize it by the file icon that symbolizes the log file. The log file is filled with CAN data during the measurement.

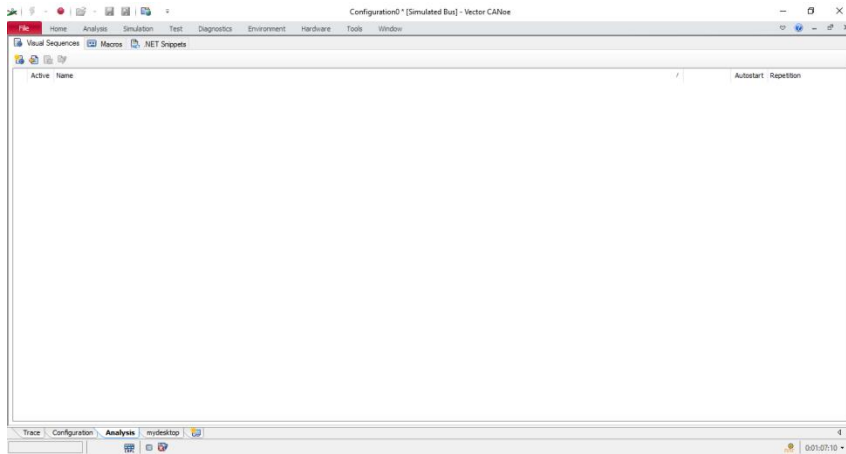
Logging configuration



Evaluating Logs

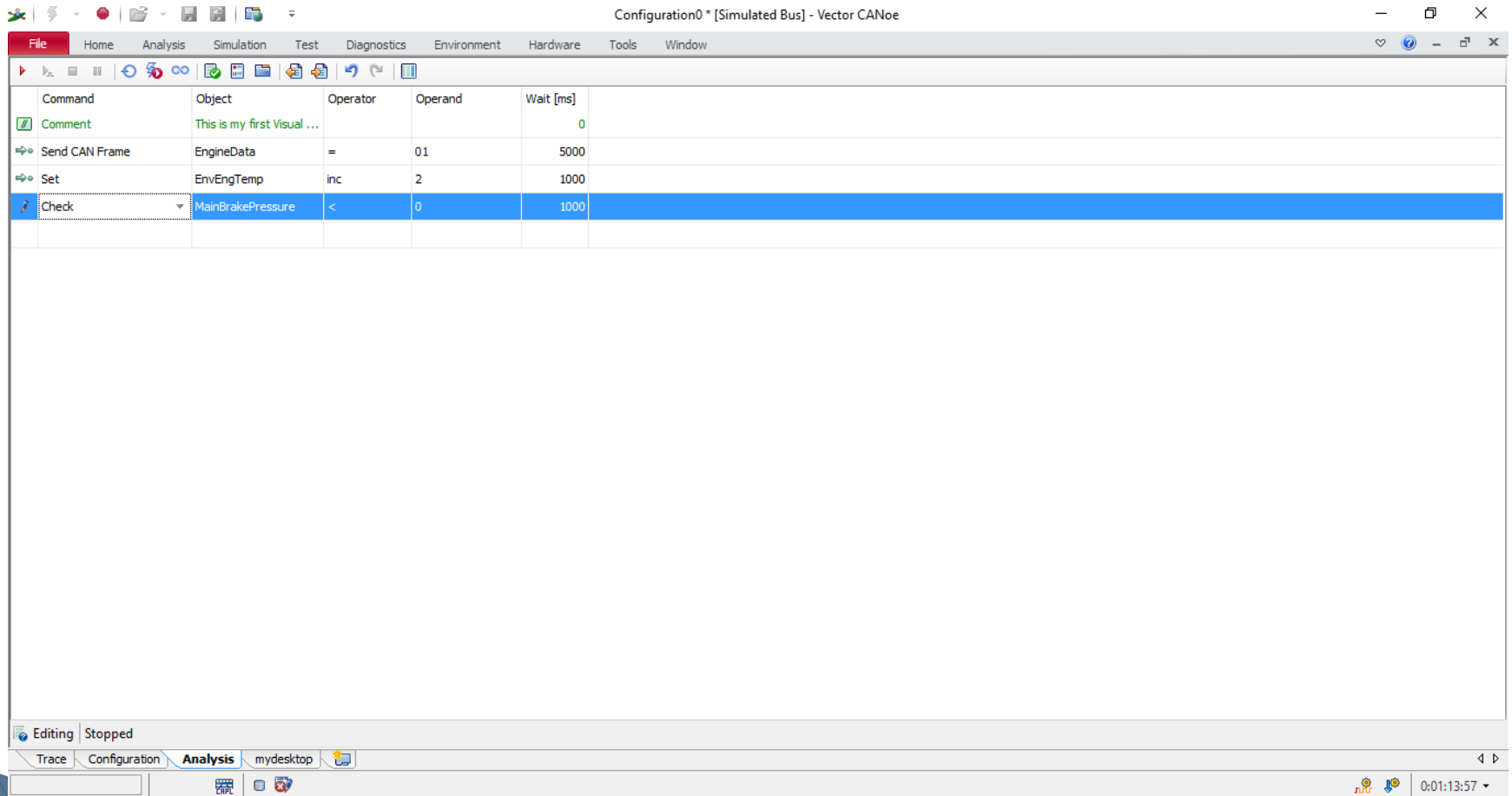
- Replay blocks
- Offline analysis

Automation – Visual Sequence

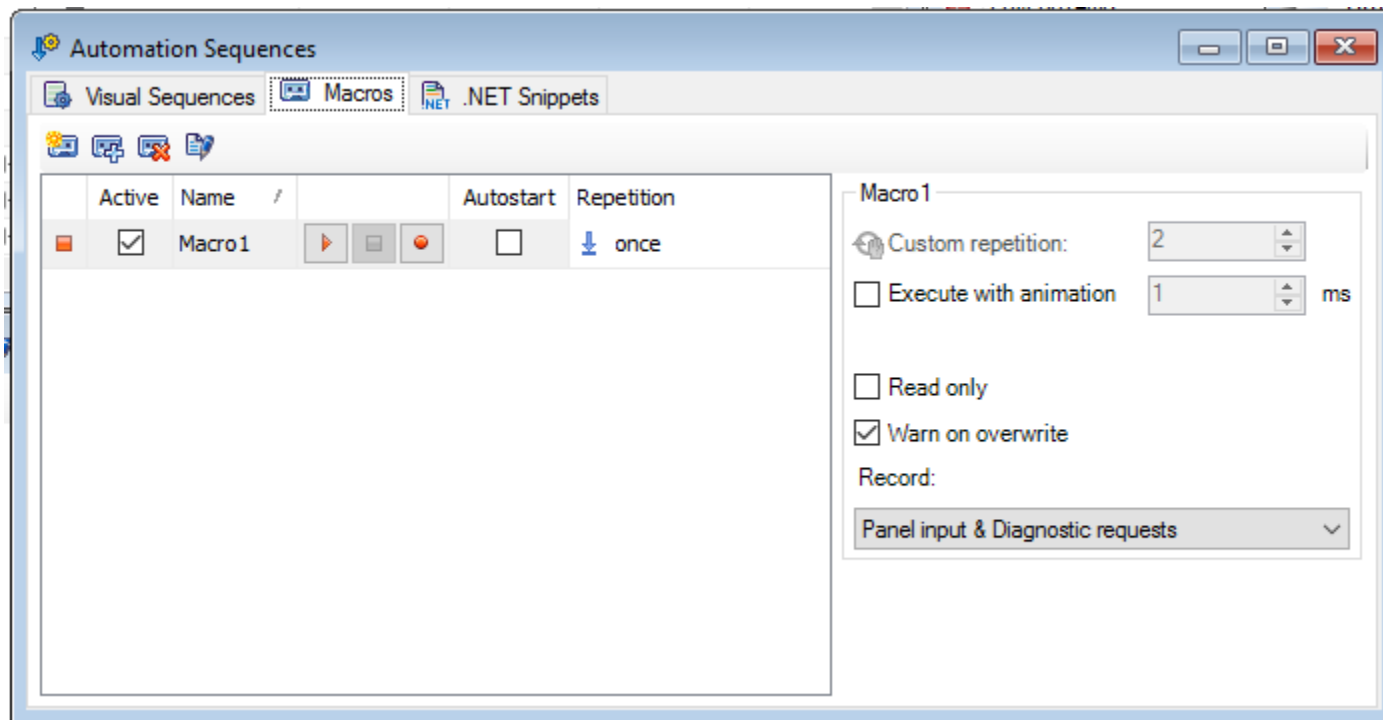


- Visual sequence helps to Create the visual or pick and design test cases
- Sequence is framed using ready made keywords and saved in the file
- Output of test can be logged and shall be executed cyclically

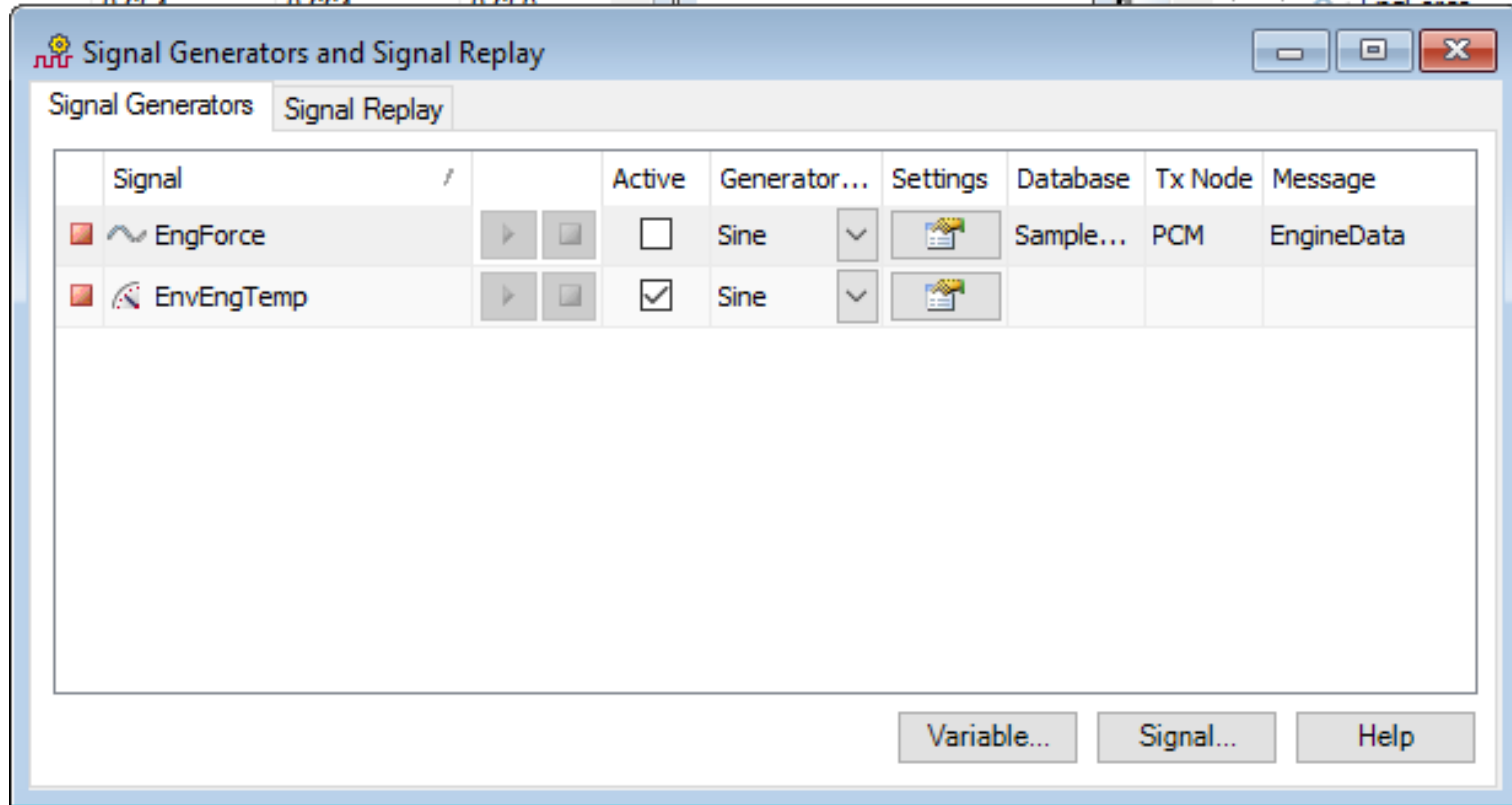
Example of Visual Sequence



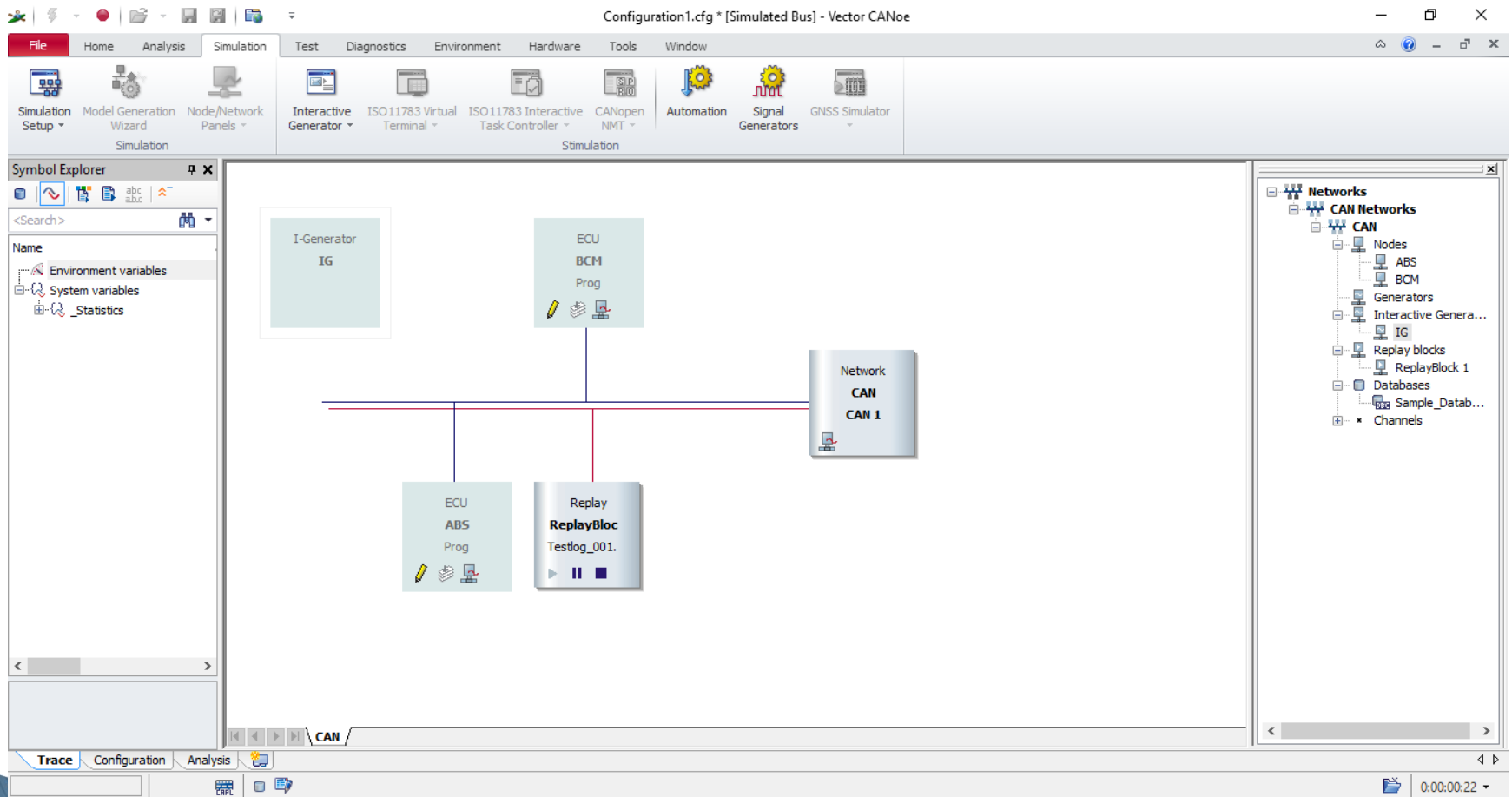
CANoe Macro



Signal Generation & Signal Replay



Replay blocks



Other Blocks

- Trigger Block
- Filter Block
- Channel filter
- Pass and Stop Filters for Environment Variables

CANDb++

- Working with Databases
- Creating and Modifying Databases
- Access to Database Information
- Associating the Database
- Resolving Ambiguities

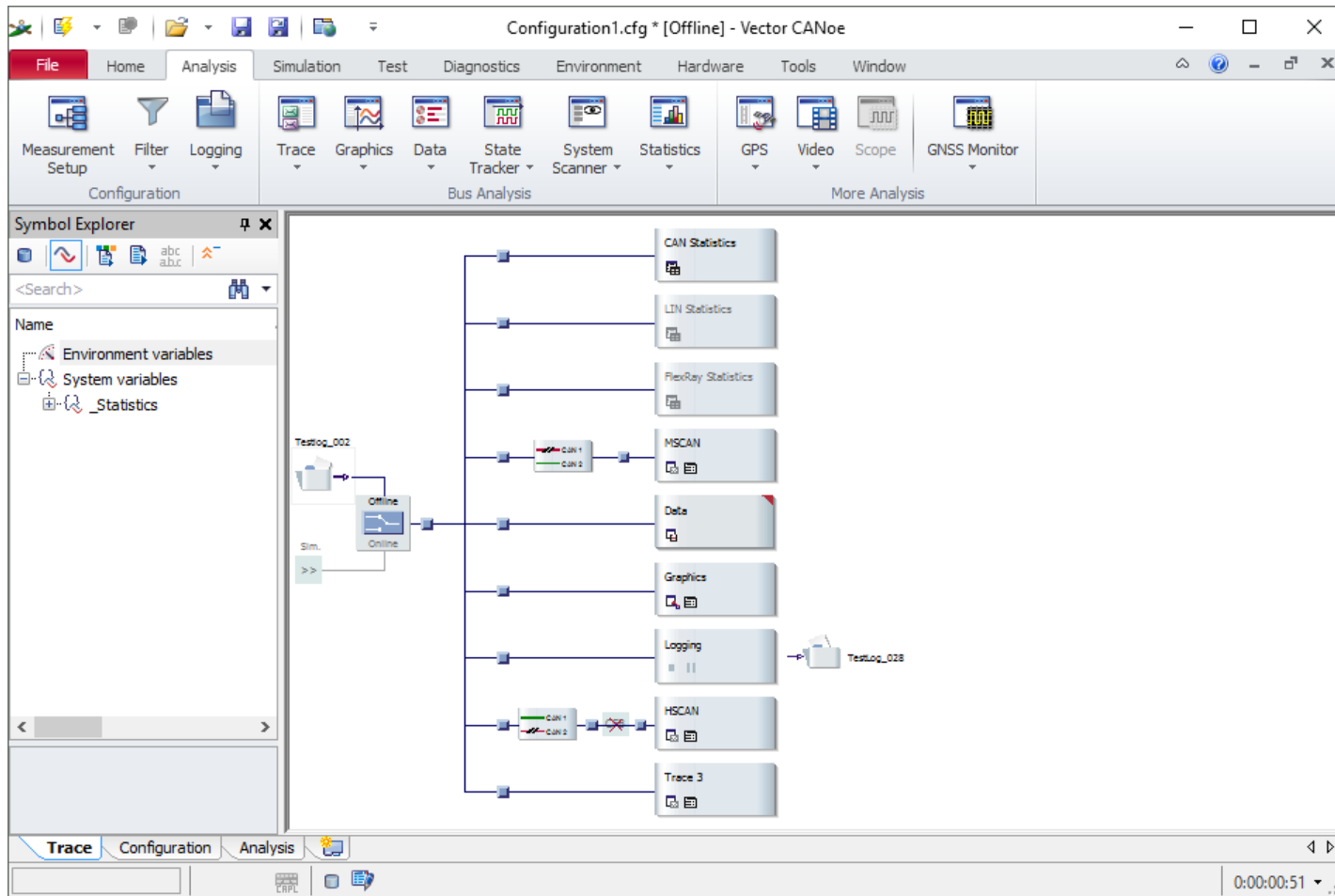
CAPL Browser

- Opening Browser
- Browser window
- Compiling CAPL programs
- Searching for Run time errors
- Accessing Database from CAPL

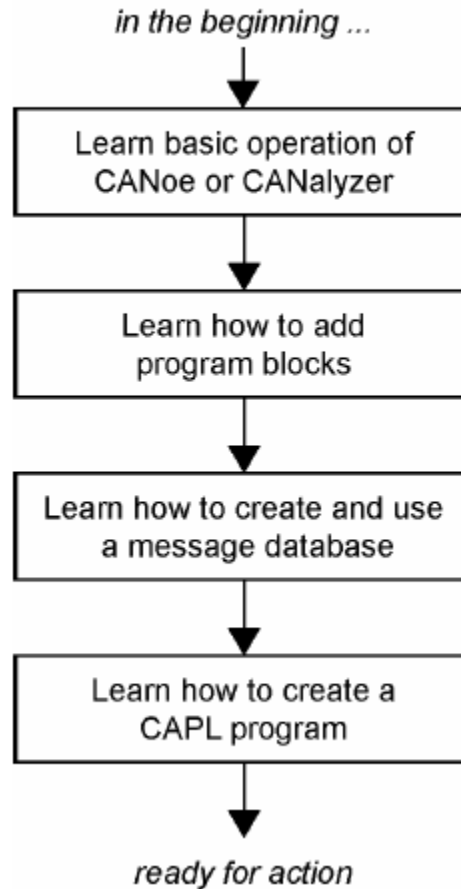
The Panel Designer

- Introduction
- Editing a Panel
- Display and Control Elements
- The hexadecimal Editor
- Working with Bitmap Controls

Offline analysis



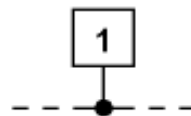
Step to learn CAPL



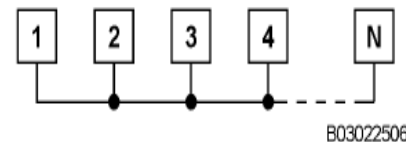
Overview About CAPL

- Based on the C programming language, CAPL, or Communication Access Programming Language, is the programming language used exclusively within the PC-based tool environments of CANalyzer and CANoe
- In simple term CAPL used in CANalyzer helps Monitor and Control CAN traffic in a single module
- CAPL used in the CANoe helps to monitor CAN traffic of real module and synthesis of other modules present in the network.
- CAPL is a C like language. It is structured based on C
- It has additional features related to CAN

CANalyzer
Single Node - Control



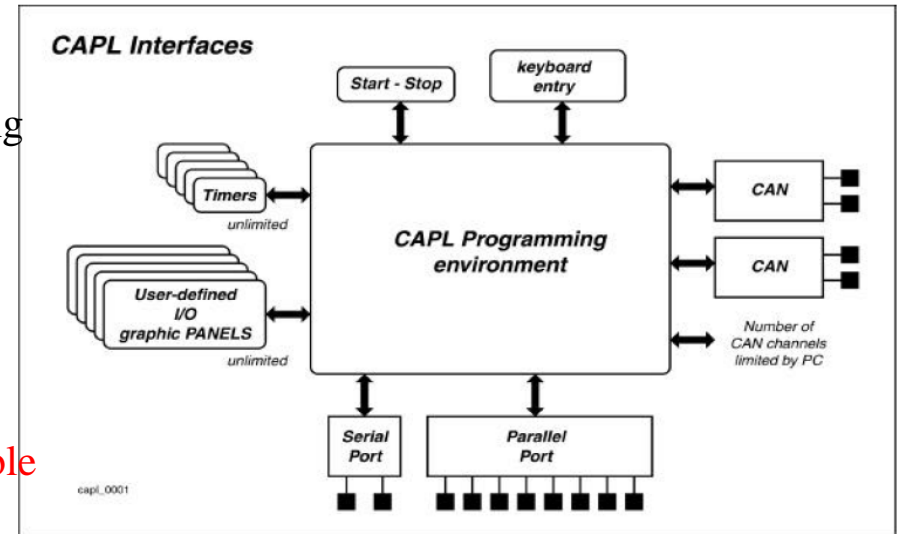
CANoe
Multiple Node - Control



B03022506

Highlights In CAPL

- Each CAPL program is a task
- All tasks are independent
- Each task has its own On start and on stop event procedures
- Each task can have n no of timers
- Available hardware resource are sharable among the tasks. (i.e any task CAN access available resource)
- Key board event can be used to raise event
- CAPLS can have GUI using Panels using EV variable or Signal (CANoe)
- **Can't debug or step run and watch is not possible**
- Write window acts like output screen in C programs
- Communication protocol function for RS232, TCP/IP are readily available. Hence we can interface external devices using std network protocol.



CAPL – Event-Driven Software

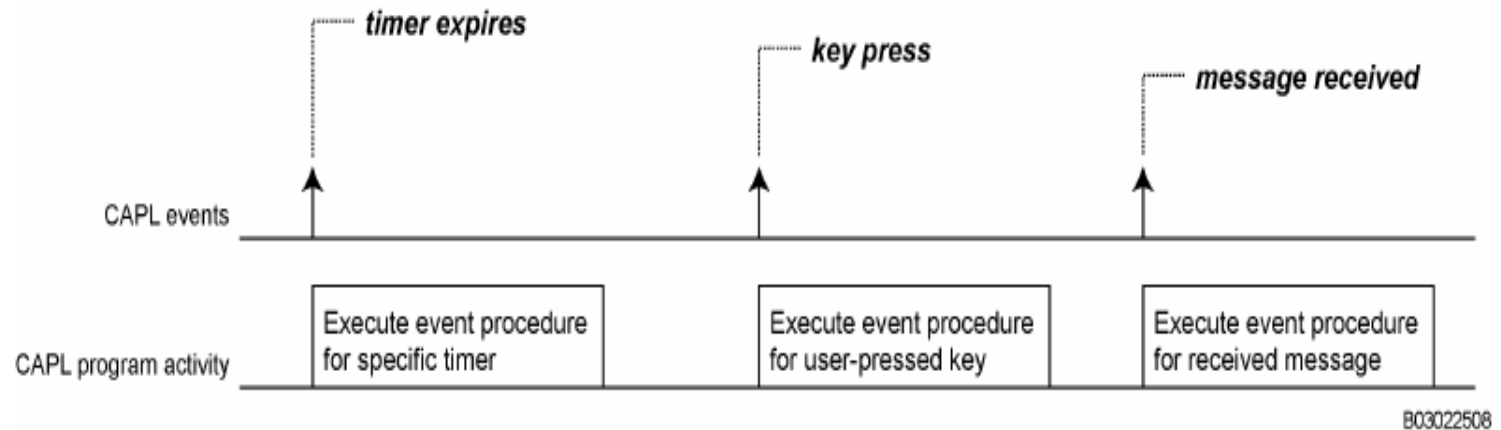


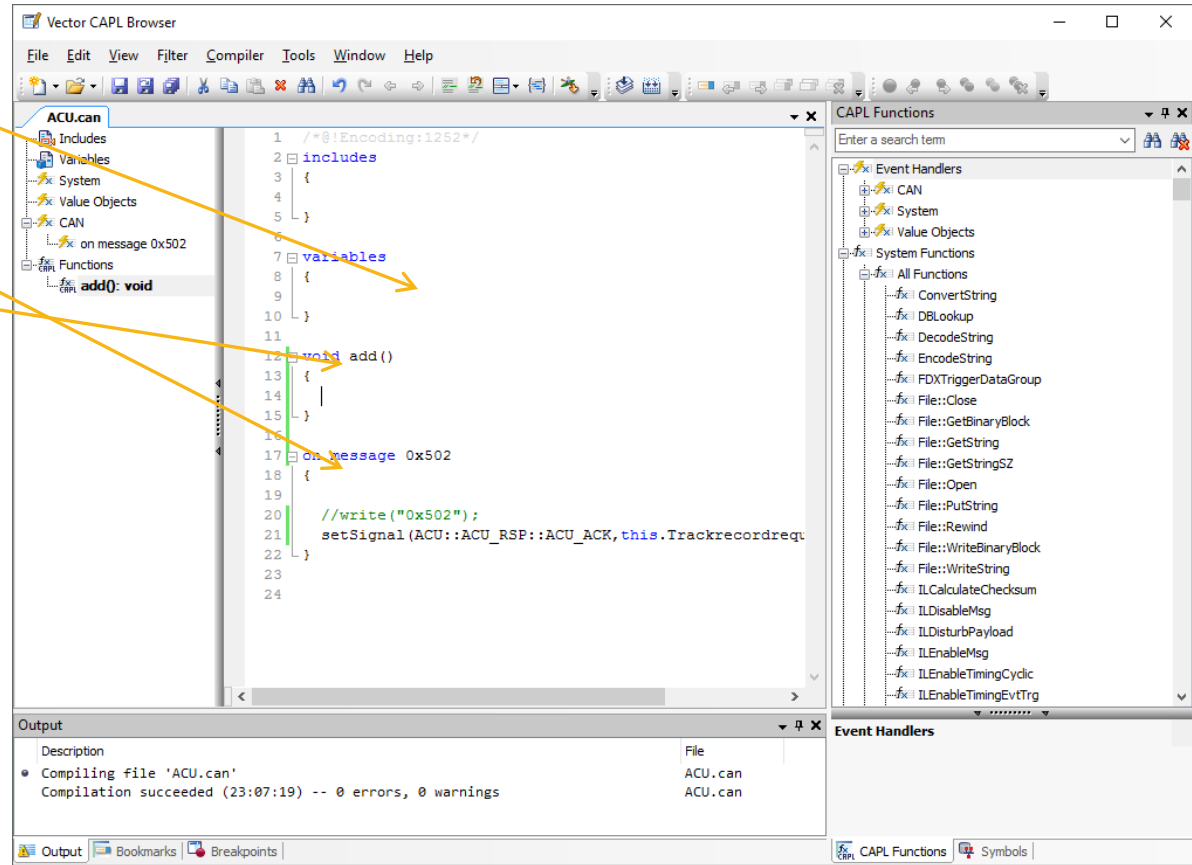
Figure 2 – CAPL Software Execution is Event-Driven

CAPL Fundamentals

- You should be familiar with the CAN protocol.
- You should be comfortable using either CANalyzer or CANoe (whichever you intend to use with CAPL) and CANdb, the CAN Database Editor
- You should also be familiar with the C programming language. It is possible to learn to program in CAPL without knowing C, but the basic program structure is the same
- CAPL utilizes two types of files. Source code files (*.CAN) contain CAPL programs
- When you compile a .CAN file in the CAPL Browser, CANalyzer, or CANoe a binary file with the extension .CBF (CAPL Binary File) is created. These binary files can only be interpreted and executed by a CANalyzer or CANoe simulation

CAPL Program Organization

- Global Variable Declarations
- Event Procedures
- User-Defined Functions



Standard C syntax supported

- Instruction blocks: { ... }
- Type casting, i.e. (int) x
- if { ... } and if {...} else { ... }
- switch, case, default
- for.., while.., do..while loops
- continue and break
- return
- // Comments
- /* Comments */

C keywords supported

Supported:

break
case
char
continue
default
do
double
else
float
for
if
int
long
return
switch
while

Not supported:

auto
const
enum
extern
goto
register
short
signed
sizeof
static
struct
typedef
union
unsigned
volatile

CAPL data types

Data Type	Description	Size	Unsigned/Signed
char	character	8 bit	unsigned
byte	byte	8 bit	unsigned
int	integer	16 bit	signed
word	word	16 bit	unsigned
long	long integer	32 bit	signed
dword	double word	32 bit	unsigned
float	single precision floating point	64 bit ¹	signed
double	single precision floating point	64 bit	signed
message	a communication message	--	--
timer	a timer with second resolution	--	--
msTimer	a timer with millisecond resolution	--	--

Table 6 – CAPL Data Types

Standard C Operators supported

Mathematical and Relational Operators	
+, -, *, /, %	add, subtract, multiply, divide, modulo
==, !=	Equal to, not equal to
>, >=, <, <=	Greater than, greater than or equal to, less than, less than or equal to
++, --	Increment, decrement (can be used as a prefix or postfix)
=	Assignment
+=, -=, *=, /=, %=	Compound assignment (addition, subtraction, multiplication, division, modulo)

Standard C Operators supported

Logical Operators

&&,	Logical AND, logical OR
?:	Conditional operator
!	Logical negation

Bitwise Operators

~	Bitwise negation
&, ^,	Bitwise AND, bitwise XOR, bitwise OR
<<, >>	Left shift, right shift
<<=, >>=	Compound assignment (left shift, right shift)
&=, ^=, =	Compound assignment (bitwise AND, bitwise XOR, bitwise OR)

Important Differences from C

- Function overloading is allowed
- The “this” keyword
- Local variables are static
- Dynamic memory and the elCount() function
- Variables are assigned a default value implicitly
- Run-time error handling

CAPL equivalents to C functions

C Function	CAPL Function	Notes
sizeof	elCount	sizeof has no exact equivalent
sprintf	snprintf	Similar string formatting parameters used
printf	write	Similar string formatting parameters used

Declaration

Below is a list of some examples of declarations:

```
char letter_a = "a";  
int number_days_in_year = 365;  
message wake-up xxx;           // see Chapter 11 on using messages for more detail  
timer one_second;              // see Chapter 14 on using timers for more detail  
int j, k = 2;                   // j = 0 implicitly  
double x = 33.7;  
char p;
```

Initialization

In contrast to standard C, the compiler initializes all numeric variables to **0** and all string-type variables to **null**. All message-type variables in CAPL are usually initialized to the transmit request state with its data field defaulted to **0**. The state represents the direction of transmission before the message is transmitted. Timer-type variables, on the other hand, are not automatically initialized and do not need to be initialized.

Local variables are static

One important difference in CAPL when compared to C is that local variables are always declared statically. This means that they are initialized only once – the first time the event procedure or user-defined function is executed. When variables enter the procedure, they assume the value they had at the end of the previous execution of the procedure.

Let's look at the following example:

```
void myFunc()
{
    byte value = 10;           // static; called once
    write("value = %d", value);
    ...
    value = 35;
}
```

Type casting

CAPL supports type casting. There are times when you will want to convert the value of a variable of one data type to another data type for purposes of evaluating an expression. There are two ways to do this in CAPL, both of which are the same as in C. As an example, let us declare the following variable:

```
int sum;
```

```
sum = 1.6 + 1.7;
```

```
sum = (int) 1.6 + (int) 1.7;
```

Arrays

Another type of variable used in CAPL is an **array**, a collection of data items that all have the same name. Individual data items within an array must all be of the same type. The array name is always followed by square brackets that contain either the array size for a declaration statement or an index value to specify the data location within the array.

Each of the individual elements in an array is accessed by an integer index. When numbering the elements of an array, the first element is always the number 0 (zero), not 1.

CAPL supports the following types of arrays:

- Integer arrays
- Character arrays
- One-dimensional arrays
- Multi-dimensional arrays

Branching statements

```
float MPH = 70.0;  
float speed;  
float cruising_speed;  
...  
if (speed >= MPH) cruising_speed = speed;  
...
```

Switch statement

```
switch (expression) {  
    case value1: statement1; statement2; ... break;  
    case value2: statement1; statement2; ... break;  
    ...  
    default: statement1; statement2; ... break;  
}
```

Switch statement

```
float value1, value2, result;  
char operator;  
  
...  
switch ( operator )  
{  
    case '+':    result = value1 + value2;  
                break;  
  
    case '-':    result = value1 - value2;  
                break;  
  
    case '*':    result = value1 * value2;  
                break;  
  
    case '/':    if ( value2 == 0) write ("Division by zero!");  
                else result = value1 / value2;  
                break;  
  
    default:    write ("Unknown operator.");  
                break;  
}
```

While Statement

while (expression) {statement};

```
byte binaryArray[16];    // global variable

binary ( int number )    // user-defined CAPL function
{
    int index;
    index = 0;

    while ( number != 0 )
    {
        binaryArray[index++] = number % 10;
        number = number / 10;
    }
}
```

Do while Statement

```
do {statement}  
while(expression);
```

```
byte binaryArray[16];      // global variable
```

```
binary ( int number )      // user-defined CAPL function  
{  
    int index;  
    index = 0;  
  
    do  
    {  
        binaryArray[index++] = number % 10;  
        number = number / 10;  
    }  
    while ( number != 0 );  
}
```

For statement

```
for (expression_1; expression_2; expression_3)
    {statement}
```

```
byte binaryArray[16];    // global variable
```

```
binary ( int number )    // user-defined CAPL function
```

```
{
    int index;
    index = 0;

    for ( ; number != 0; )
    {
        binaryArray[index++] = number % 10;
        number = number / 10;
    }
}
```

Unconditional branching

Break Statement

```
int x, y, z;

while ( x != 0 )
{
    doThis (x);
    if ( x == 0 )
        break;
    doThat (x);
}
doSomethingElse (y, z);
```

Continue Statement

```
int x, y, z;

while ( x != 0 )
{
    doThis (x);
    if ( x == 4 )
        continue;
    doThat (x);
}
doSomethingElse (y, z);
```

Return Statement

```
long Power(int x, int y)
{
    int i;
    long result;
    result = 1;

    for (i = 1; i <= y; i++)
        result *= x;

    return result;
}
```


The “this” keyword

```
on message 555
{
    byte val;
    val = 0;

    /* checks to see if the message was received on CAN channel 1*/
    if (this.CAN == 1)
    {
        // sets “val” to the value of the first data byte in the CAN message
        val = this.byte(0);
    }
}
```

CAPL functions - User defined

In CAPL, a user-defined function is similar to a function defined in the C programming language. A function call is possible in any CAPL event procedure. These functions can contain any legal CAPL code, and are globally accessible. Putting frequently used code in a function makes programs more efficient and organized.

Function overloading is allowed

```
void printme (double num)
{
    write("Floating point %f", num);
}
```

```
void printme (int num, char units[])
{
    write("%d %s", num, units);
}
```

Logging function

Logging Functions	
Function	Purpose
setLogFileName	Sets the name of the log file
setPostTrigger	Sets the posttrigger time for logging
setPreTrigger	Sets the pretrigger time for logging
startLogging	Triggers a Logging block to start logging
stopLogging	Triggers a Logging block to stop logging
trigger	Starts all Logging blocks to a log
writeToLog	Writes a formatted string to the log file with time stamp
writeToLogEx	Writes a formatted string to the log file without time stamp

Table 20 – Logging Functions

String handling function

String Handling Functions	
Function	Purpose
atol	Converts a string to a long integer
ltoa	Converts a number to a string
snprintf	Creates a formatted string
strlen	Gets the length of a string
strncat	Concatenates two strings
strncmp	Compares two strings
strncpy	Copies a string

Table 21 – String Handling Functions

Measurement and control functions

Functions	Purpose
ILControlStart	Connect Node to the bus
ILControlStop	Remove Node from the bus

Measurement Control Functions	
Function	Purpose
canOffline	Disconnects node from bus (CANoe's Simulation Setup window only)
canOnline	Reconnects node to the bus (CANoe's Simulation Setup window only)
getBusContext	Returns current bus context (CANoe's Simulation Setup window only)
getBusNameContext	Returns a specific bus context (CANoe's Simulation Setup window only)
getStartdelay	Gets delay time value of current node (CANoe's Simulation Setup window only)
halt	Halts the simulation (only possible in the Simulated Mode of CANoe)
inspect	Updates the variables in the <i>Inspect</i> pane of the Write window (CANoe only)
setBusContext	Sets new context of current bus (CANoe's Simulation Setup window only)
setStartdelay	Sets delay time value of present node (CANoe's Simulation Setup window only)
stop	Stops the measurement

Table 22 – Measurement Control Functions

WriteEx() Function

- ▶ void writeEx(dword sink, dword severity, char format[], ...)
- ▶ Writes the text into the last line of the specified window or into a tab of the Write window without previously creating a new line.

Window or tab of the Write window

-3	Trace window
-2	Trace window for logging output
-1	Reserved
0	Output to the System tab of the Write window
1	Output to the CAPL tab of the Write window
4	Output to the Test tab of the Write window

Type of message for output in the Write window

0	Success
1	Information
2	Warning
3	Error

Statistic functions

Statistics Functions	
Function	Purpose
isStatisticAcquisitionRunning	Checks to see if acquisition range is running
startStatisticAcquisition	Sets new acquisition range
stopStatisticAcquisition	Stops acquisition range

Table 23 – Statistics Functions

CAN Protocol functions

CAN Protocol Functions	
Function	Purpose
canSetChannelAcc	Sets channel acceptance
canSetChannelMode	Activates/deactivates TX and TXRQ display of a channel
canSetChannelOutput	Activates/deactivates acknowledgement bit
getCardType	Determines the type of CAN platform being used
getChipType	Determines the type of CAN controller being used
resetCAN	Resets all CAN Controllers
resetCANEx	Resets a specific CAN Controller
setBtr	Sets the baud rate for a particular channel in the Bit Timing Register (BTR)
setCanCabsMode	Sets a mode for a CANcab
setOcr	Sets the Output Control Register

Table 24 – CAN Protocol Functions

Replay block functions

Replay Block Functions	
Function	Purpose
ReplayResume	Resumes replaying log file after it was suspended
ReplayStart	Starts replaying log file from the beginning
ReplayState	Returns the state the Replay block is in
ReplayStop	Stops replaying log file
ReplaySuspend	Suspends replaying log file

Table 26 – Replay Block Functions

Environment variable and Panel functions

CANoe Environment Variables and Panels Functions	
Function	Purpose
callAllOnEnvVar	Calls all environment variable procedures
enableControl	Enables or disable an element on a panel
getValue	Reads the value of an environment variable
getValueSize	Gets the size of an environment variable in bytes
makeRGB	Sets primary color values
putValue	Assigns a value to an environment variable
putValueToControl	Assigns values to multi display control in panels
setControlBackColor	Sets the background color of a panel element
setControlForeColor	Sets the foreground color of a panel element
setControlProperty	Sets a property of an ActiveX control

Table 27 – Environment Variable Functions

Variable and Object Types and Declarations

CAPL Data Types	
8-Bit	byte, char
16-Bit	int.word
32-Bit	dword, long
64-Bit (Floating Point)	float. double
object	msTimer, Timer, message, pg

Examples of variable declarations:

```
int j, k = 2;          /* j = 0 implicitly */  
double x = 33.7;  
char p;
```

Examples of array declarations:

```
long data[10];  
int vector[5] = { 0, 1, 2, 3, 4 };  
int matrix[2][4] = { {2, 4, 6, 8}, {1, 3, 5, 7} };  
char userName[10] = "Jack Beam";
```

The Timer and msTimer Objects

- Since CAPL is designed to provide an event-driven environment, timers provide an easy way to cause periodic events.
- CAPL allows an infinite number of user-defined timers that can be set using units of seconds (Timer) or milliseconds (msTimer).
- Timers are set using the setTimer function and canceled using cancelTimer function. The most common way to set up a periodic timer is to set the timer in the initialization (or “on Start”) function, set up an event based on this timer to perform a given task, and then reset the timer at the end of this task.
- Example:

```
Timer t1;  
setTimer(t1, 60);  
msTimer t2;  
setTimer(t2, 100);
```

```
/* Units of seconds */  
/* Set to one minute */  
/* Units of milliseconds */  
/* Set to one tenth of a second */
```

Declaring messages in CAPL

- ▶ they must be declared (like variables) before they can be used.
- ▶ messages can be specified in two ways: either in an associated database or in a CAPL program itself. Declaring a message specified in an associated database is easy — just use the symbolic name of the message.

Examples

```
message EngineTemp eTemp;  
message EngineStatus eStat1, eStat2;  
message 411 info;           // Message ID 411 (decimal)  
message 0xF2 windowStatus; // Message ID F2 (hex)  
message 142x doorStatus;    // Extended CAN message ID 142  
message * userdefined;      // User defined message generic declaration  
// Declare message ID FF (hex) with 4 data bytes  
message 0xFF rearAxle = { DLC = 4 };
```

Accessing message signals

- ▶ If you have defined a message in an associated database, it is easy to access individual signals within the message.
- ▶ Signals are specified as a part of a message by putting a dot after the message name and then the signal name.

Example

```
message EngData EDMsg;    // Declare EDMsg as type EngData  
EDMsg.EngSpeed = 100;      // Access signal EngSpeed
```

Physical values and the phys attribute

$$\text{Physical Value} = (\text{Raw Value} * \text{Factor}) + \text{Offset}$$

The screenshot shows a configuration window titled "Signal 'ACFRDEFSW'". It has several tabs: "Definition", "Messages", "Receivers", "Attributes", "Value Descriptions", and "Comment". The "Definition" tab is active. The fields are as follows:

- Name: ACFRDEFSW
- Length [Bit]: 1
- Byte Order: Intel (dropdown)
- Unit: (empty field)
- Value Type: Unsigned (dropdown)
- Init. Value: 0
- Factor: 1
- Offset: 0
- Minimum: 0
- Maximum: 0
- Value Table: <none> (dropdown)
- ☒ Automatic min-max calculation

At the bottom, there are buttons for "OK", "Cancel", "Apply", and "Help".

Message selectors

CAPL Message Selectors		
Selector	Description	Valid values
ID	Message identifier	Any valid CAN message ID
CAN	Chip number	1 or 2
DLC	Data Length Code	0 to 8
DIR	Direction of transmission	RX (Receive) TX (Transmit) TXREQUEST (Transmit Request)
RTR	Remote TransmissionRequest	0 (not an RTR) 1 (RTR)
TYPE	Combination of DIR and RTR	See below
TIME	Time stamp of the message in units of 10ms (read only)	Long integer
SIMULATED	Sent by a simulated node (read only)	0 (real node) 1 (simulated node)

Message selectors

Valid Values for the TYPE Message Selector		
Constant	Meaning	Logical Derivation
RXREMOTE	Remote message was received	((DIR == RX) && RTR)
TXREMOTE	Remote message was transmitted	((DIR == TX) && RTR)
TXREQUESTREMOTE	Transmission request was set for remote message	((DIR == TXREQUEST) && RTR)
RXDATA	Data message was received	((DIR == RX) && !RTR)
TXDATA	Data message was transmitted	((DIR == TX) && !RTR)
TXREQUESTDATA	Transmission request was set for data message	((DIR == TXREQUEST) && !RTR)

Accessing raw message data

Example

```
message 400 msg;  
msg.DLC = 8; // 8 bytes of data  
msg.long(0) = 333; // long integer starting at first byte  
msg.byte(4) = 25; // Byte starting at fifth byte  
msg.word(5) = 747; // Word starting at sixth byte  
msg.byte(7) = 100; // Byte starting at eighth byte
```

Example of Message Data Layout							
Byte (7)	word (5)		Byte (4)	long(0)			
100	747		25	333			
Eight Byte	Seventh Byte	Sixth Byte	Fifth Byte	Fourth Byte	Third Byte	Second Byte	First Byte

Sending and receiving messages

```
message EngineData msg;  
msg.Rpm.phys = 225;
```

```
msg.CAN = 2;  
output(msg);
```

```
// Declare msg as EngineData  
// Set physical value,actual value  
// stored in message bits will be  
// different.  
// Output on CAN channel 2  
// Send EngineData message out
```

Environmental Variables (CANOe)

- Environmental variables are simple to use in CAPL. Once they are defined in an associated database, you can insert the variable name into your code with the right mouse button menu.
- CAPL provides two functions, `getValue()` and `putValue()`, that read and set the value of an environmental variable, respectively. Since an environmental variable can be an integer, floating point number, or a string, CAPL provides three ways to call each function

getValue() and putValue() Declarations		
Data Type	getValue() functions	putValue() functions
Integer	<code>int getValue(EnvVarName);</code>	<code>putValue(EnvVarName, int val);</code>
Floating Point	<code>float getValue(EnvVarName);</code>	<code>putValue(EnvVarName, float val);</code>
String	<code>getValue(EnvVarName, char buffer[]);</code>	<code>putValue(EnvVarName, char val[]);</code>

Environmental Variables (CANOe)

Example

```
char[100] lightColor;          // A 100-character buffer will suffice for the string
int lightState;                // Light state is an integer
                                // First, read the value of the EV_LightState
                                // environmental variable into the lightState integer
lightState = getValue(EV_LightState);
                                // Next, read the value of the EV_LightColor
                                // environmental variable into the lightColor string
getValue(EV_LightColor, lightColor);
```

Built-in Constants

- CAPL provides some built-in constant values for use in your programs. Some of them only are used in specific cases.
- However, you might find the PI constant useful. It contains the value of pi to the maximum precision useful to CAPL. Also, the constants LPT1, LPT2, and LPT3 can be used in the place of port addresses for the inport() and outport() functions.

Events in CAPL

- ▶ CAPL is an event-driven language. This means that sections of a program are executed in response to specific occurrences (events) rather than the whole program being executed all at once.
- ▶ In CAPL, an event procedure is associated with every event that is important to the application. For example, you can define event procedures for a user's keystroke, the beginning of a measurement, or the reception of a certain message.

```
// Message event block with required parameter.  
// Event occurs when message ID 101 is received.  
on message 101  
{  
... // Your event code is executed here  
}  
// Error frame event block without a parameter.  
// Event occurs when an error frame is received.  
on errorFrame  
{  
... // Your event code is executed here  
}
```


Events in CAPL

CAPL EVENT TYPES		
Event description	Event statement syntax	Value of "this" inside event procedure
Receipt of a CAN message	on message message	the message received
Press of a key	on key key	the key pressed
Initialization of measurement (before start)	on preStart	N/A
Program start	on start	N/A
End of measurement	on stopMeasurement	N/A
CAN controller goes into ErrorActive state	on errorActive	contains error counts
CAN controller goes into ErrorPassive state	on errorPassive	contains error counts
CAN controller goes into Warning Limit state	on warningLimit	contains error counts
CAN controller goes into Bus Off state	on busOff	contains error counts
Elapse of a user-defined timer	on timer timer	N/A
Occurrence of an error frame	on errorFrame	N/A
Environment variable change (CANoe only)	on envVar env. variable	the env. variable

The “on message” event

on message 123	Event occurs when message 123 (decimal) received
on message 0x123	Event occurs when message 123 (hex) received
on message EngineTemp	Event occurs when message EngineTemp received (EngineTemp must be defined in database)
on message CAN1.123	Event occurs when message 123 received on CAN1
on message *	Event occurs when any message is received
on message CAN2.*	Event occurs when any message is received on CAN2
on message 0, 5, 10-20	Event occurs when messages 0, 5, and 10 through 20 are received

The “on message” event

on message CAN1.123
on message CAN2.*
on message 123

/* Message 123 on CAN1 calls this */
/* Any message on CAN2 calls this */
/* This is never called. Message 123
would call one of the above two
events */

on message *

/* Called by any message received on
CAN1 except message 123 */

The “on key” event

on key 'a'	Event occurs when the (lower case) "a" key is pressed
on key 'A'	Event occurs when the (capital) "A" key is pressed
on key End	Event occurs when the End key is pressed (notice absence of quotes)
on key shiftF1	Event occurs when Shift-F1 is pressed
on key ctrlPageDown	Event occurs when Control-Page Down is pressed
on key *	Event occurs when any key is pressed (see below)

The “on key” event

```
on key * {  
  switch(this) {           // “this” holds the value of the key  
    ‘a’ :  
      write (“a or b was pressed.”);  
      break;  
    default:  
      write (“A key besides a or b was pressed.”);  
      break;  
  }  
}
```

The preStart, start, stopMeasurement events, and CallAllOnEnvVar()

- ▶ The preStart, start, and stopMeasurement events are used to perform actions before, at the start of, and after CANalyzer or CANoe measurements

The errorActive, errorPassive and busOff events

CAN Controller States	
Error Active	Transmit and receive error counts are less than or equal to 127
Error Passive	Transmit or receive error count equals or exceeds 128
Bus Off	Transmit error count is greater than or equal to 256

this.errorCountRX	Receive error count
this.errorCountTX	Transmit error count

bus off event

```
on busOff
{
write (“CAN controller is in the Bus Off
state.”);
write (“Stopping measurement.”);
stop();
}
```

The “on timer” event and the `setTimer()` and `cancelTimer()` functions

```
variables {  
  msTimer t;  
  ...  
}  
on start {  
  setTimer(t, 20);  
  ...  
}  
on timer t {  
  ... // Perform some action here  
  setTimer(t, 20);  
}
```

The “on errorFrame” event

The “on errorFrame” event is similar to the “on message” event, except it occurs whenever an error frame is received on the bus

RS232 CAPL functions

Function Name	Short Description
RS232Open	Opens serial port.
RS232Close	Closes a serial port.
RS232Configure	Configures serial port
RS232Send	Sends block of data asynchronously on serial port.
RS232Receive	Sets receiver buffer for serial port.
RS232OnSend	Handler will be called after completion of request issued with RS232Send.
RS232OnReceive	Handler will be called regularly if data has been received at port. Starts with RS232Receive.
RS232OnError	Handler will be called if an error has occurred.

P block placement in measurement setup

CAPL Function Group

File IO Functions

<code>openFileRead</code>	Opens the file for the read access.
<code>fileClose()</code>	Closes the specified file.
<code>openFileWrite()</code>	Opens the file for the write access.
<code>setFilePath()</code>	Sets the read and write path to the directory.
<code>setWritePath()</code>	Sets the write path for the function <code>openFileWrite</code> .
<code>fileGetBinaryBlock</code>	Reads characters from the specified file in binary format.
<code>fileGetString</code>	Reads a string from the specified file.
<code>fileGetStringSZ</code>	Reads a string from the specified file.
<code>filePutString</code>	Writes a string in the specified file.
<code>fileRewind</code>	Resets the position pointer to the beginning of the file.
<code>fileWriteBinaryBlock</code>	Writes bytes in the specified file.

CAPL Function Group

File IO Functions	
Function Name	Short Description
getProfileArray	Reads the value of the given variable from the specified section in the specified file.
getProfileFloat	
getProfileInt	
getProfileString	
writeProfileFloat	Opens the file, searches the section and writes the variable.
writeProfileInt	
writeProfileString	

CAPL Function Group

Byte Conversion Functions	
swapDword	Re-order bytes of double (64 bit data type)
swapInt	Re-order bytes of integer (8 bit data type)
swapLong	Re-order bytes of long integer (32 bit data type)
swapWord	Re-order bytes of word (16 bit data type)

String Handling Functions	
atol	Convert a string to a long integer
ltoa	Convert a number to a string
snprintf	Create a formatted string
strlen	Get the length of a string
strncat	Concatenate two strings
strncmp	Compare two strings
strncpy	Copy a string

CAPL Function Group

Math Functions	
abs	Calculate an absolute value
cos	Calculate a cosine
exp	Calculate an exponent
random	Calculate a random value
sin	Calculate a sine
sqrt	Calculate a square root

Message Handling Functions	
isExtId	Check for extended identifier
isStdId	Check for standard identifier
mkExtId	Create extended identifier
output	Output message, pg, or error frame onto the bus
setMsgTime	Assign time to message (obsolete)
valOfId	Get the value of a message identifier

CAPL Function Group

Timer Functions	
cancelTimer	Stop an active timer
setTimer	Set a timer

User Interface Functions	
beep	Make the system beep
fileName	Output the program name in the Write window
keypressed	Query currently depressed key
write	Output text to the Write window
sysExit	Exit CANalyzer or CANoe
sysMinimize	Minimize CANalyzer or CANoe window

CAPL Function Group

Logging Functions	
setLogFileName	Set the file name of the log file
setPostTrigger	Set the posttrigger time for logging
setPreTrigger	Set the pretrigger time for logging
trigger	Activate logging triggering
writeToLog	Write a formatted string to the log file

Measurement Setup and Control Functions	
getCardType	Determine the type of CAN platform being used
getChipType	Determine the type of CAN controller being used
resetCan	Reset the CAN controller
runError	Trigger a run-time error
setBtr	Set baud rate
setOcr	Set the Output Control Register
stop	Stop the measurement

CAPL Function Group

Time Functions	
timeDiff	Calculate time difference between two messages
timeNow	Get current system time
getLocalTime	Get array time information
getLocalTimeString	Get human-readable time string

CAPL Function Group

Node Control Functions (CANoe Only)

canOnline	Reconnects node to the bus
putValue	Disconnects node from bus

Port I/O Functions

inport	Read a byte from a port
outport	Send a byte to a port

Miscellaneous Functions

elCount	Count the number of elements in an array
---------	--

J1939-Specific Functions

getThisMessage	Read data from a parameter group
----------------	----------------------------------

CAPL Test feature set

CAPL Test feature set

Functions	Short Description
checkSignalInRange	Checks the value of the signal, the system variable or the environment variable against a condition
testValidateSignalInRange	
testValidateSignalOutsideRange	
CheckSignalMatch	Checks a given value against the value of the signal, the system variable or the environment variable.
TestValidateSignalMatch	Checks a given value against the value of the signal, the system variable or the environment variable.
GetRawSignal	Retrieves the current raw value of a signal.
getSignal	Gets the value of a signal.
getSignalTime	Gets the time point when the signal value has been changed to the current value.
RegisterSignalDriver	Registers the given callback as a 'signal driver' for the signal.
SetRawSignal	Sets the raw value of a signal.
SetSignal	Sets the transmitted signal to the accompanying value.

CAPL Test feature set

Functions	Short Description
TestResetEnvVarValue	Resets an environment variable to initial value.
TestResetNamespaceSysVarValues	Resets all system variables of the given namespace (and all sub-namespaces) to their initial value.
TestResetNodeSignalValues	Resets all tx-signals of a node to their initial value.
TestResetSignalValue	Resets a signal to initial value.
TestResetSysVarValue	Resets a system variable to initial value.

Diagnostic Basics

- DID – Diagnostic Identifier
- VIN – Vehicle Identification Number
- DTC - Diagnostic trouble code
- UDS – Unified Diagnostic service

Diagnostic service

- Information exchange initiated by a client in order to require diagnostic information from a server and/or to modify its behaviour for diagnostic purposes

Tester

- system that controls functions such as test, inspection, monitoring or diagnosis of an on-vehicle electronic control unit and which may be dedicated to a specific type of operator (e.g. a scan tool dedicated to garage mechanics or a test tool dedicated to assembly plant agents)

Diagnostic data

- Data that is located in the memory of an electronic control unit which may be inspected and/or possibly modified by the tester (diagnostic data includes analogue inputs and outputs, digital inputs and outputs, intermediate values and various status information)
- Example : Examples of diagnostic data include vehicle speed, throttle angle, mirror position, system status, etc. Three types of values are defined for diagnostic data

Diagnostic session

- current mode of the server, which affects the level of diagnostic functionality NOTE
Defining a repair shop or development testing session selects different server functionality (e.g. access to all memory locations may only be allowed in the development testing session).

Diagnostic routine

- routine that is embedded in an electronic control unit and that may be started by a server upon a request from the client

Permanent DTC

- stored in NVRAM and not erasable by any test equipment command or by disconnecting power to the on-board computer

DiagnosticSessionControl - 10 hex

Table 25 — Request message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	DiagnosticSessionControl Request Service Id	M	10	DSC
#2	sub-function = [diagnosticSessionType]	M	00-FF	LEV_ DS_

Default Session - 01

- This diagnostic session enables the default diagnostic session in the server(s) and does not support any diagnostic application timeout handling provisions (e.g. No TesterPresent service is necessary to keep the session active).

Programming Session-02

- This diagnosticSession enables all diagnostic services required to support the memory programming of a server.
- If the server runs the programmingSession in the boot software, the programmingSession shall only be left via an ECUReset (11 hex) service initiated by the client, a DiagnosticSessionControl (10 hex) service with sessionType equal to default Session, or a session layer timeout in the server.

Extended Diagnostic Session-03

- This diagnostic Session can e.g. be used to enable all diagnostic services required
- to support the adjustment of functions such as “Idle Speed”, “CO Value”, etc. in the server's memory.
- It can also be used to enable diagnostic services, which are not specifically tied to the adjustment of functions

Session Control

Table 27 — Positive response message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	DiagnosticSessionControl Response Service Id	S	50	DSCPR
#2	diagnosticSessionType	M	00-7F	DS_
#3 : #n	sessionParameterRecord[] #1 = [data#1 : data#m]	C ^a : C	00-FF : 00-FF	SPREC_ DATA_1 : DATA_m
^a C is the presence, structure and content of the sessionParameterRecord and is data-link-layer-dependant and therefore defined in the implementation specification(s) of ISO 14229.				

Session control

Table 29 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	subFunctionNotSupported Send if the sub-function parameter in the request message is not supported.	M	SFNS
13	incorrectMessageLengthOrInvalidFormat The length of the message is wrong.	M	IMLOIF
22	conditionsNotCorrect This code shall be returned if the criteria for the request DiagnosticSessionControl are not met.	M	CNC

ECUReset (11 hex) service

Table 32 — Request message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	ECUReset Request Service Id	M	11	ER
#2	sub-function = [resetType]	M	00-FF	LEV_ RT_

HardReset - 01

- This value identifies a “hard reset” condition which simulates the power-on/start-up sequence typically performed after a server has been previously disconnected from its power supply (i.e. battery).
- The performed action is implementation specific and not defined by ISO 14229. It might result in the re-initialization of both volatile memory and non-volatile memory locations to predetermined values.

Hard reset

Table 34 — Positive response message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	ECUReset Response Service Id	S	51	ERPR
#2	resetType	M	00-7F	RT_
#3	powerDownTime	C ^a	00-FF	PDT
^a C: This parameter is present if the sub-function parameter is set to the enableRapidPowerShutDown value (04hex).				

Hard reset

Table 36 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
12	subFunctionNotSupported Send if the sub-function parameter in the request message is not supported.	M	SFNS
13	incorrectMessageLengthOrInvalidFormat The length of the message is wrong.	M	IMLOIF
22	conditionsNotCorrect This code shall be returned if the criteria for the ECUReset request is not met.	M	CNC
33	securityAccessDenied This code shall be sent if the requested reset is secured and the server is not in an unlocked state.	M	SAD

SecurityAccess (27 hex) service

- The purpose of this service is to provide a means to access data and/or diagnostic services which have restricted access for security, emissions or safety reasons. Diagnostic services for downloading/uploading
- routines or data into a server and reading specific memory locations from a server are situations where security access may be required.
- Improper routines or data downloaded into a server could potentially damage the electronics or other vehicle components or risk the vehicle's compliance to emissions, safety or security standards. The security concept uses a seed and key relationship.

SecurityAccess (27 hex) service

- A typical example of the use of this service is as follows:
 - client requests the “seed”;
 - server sends the “seed”;
 - client sends the “key” (appropriate for the Seed received);
 - server responds that the “key” was valid and that it will unlock itself.

Request seed

Table 39 — Request message definition — sub-function = requestSeed

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	SecurityAccess Request Service Id	M	27	SA
#2	sub-function = [securityAccessType = requestSeed]	M	01, 03, 05, 07-7D	LEV_ SAT_RSD
#3 : #n	securityAccessDataRecord[] = [parameter#1 : parameter#m]	U : U	00-FF : 00-FF	SECACCDR_ PARA1 : PARAM

Send key

Table 40 — Request message definition — sub-function = sendKey

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	SecurityAcces Request Service Id	M	27	SA
#2	sub-function = [securityAccessType = sendKey]	M	02, 04, 06, 08-7E	LEV_ SAT_SK
#3 : #n	securityKey[] = [key#1 (high byte) : key#m (low byte)]	M : U	00-FF : 00-FF	SECKEY_ KEY1HB : KEYmLB

Security unlock

Table 43 — Positive response message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	SecurityAccess Response Service Id	S	67	SAPR
#2	securityAccessType	M	00-7F	SAT_
#3	securitySeed[] = [seed#1 (high byte) : seed#m (low byte)]	C ^a	00-FF	SECSEED_ SEED1HB
:		:	:	:
#n		C	00-FF	SEEDmLB

^a C: The presence of this parameter depends on the securityAccessType parameter. It is mandatory that it be present if the securityAccessType parameter indicates that the client wants to retrieve the seed from the server.

Example flow

Table 46 — SecurityAccess request message flow example #1

Message direction:	client → server		
Message type:	Request		
A_Data byte	Description (all values are in hexadecimal)	Byte value (hex)	Mnemonic
#1	SecurityAccess request SID	27	SA
#2	SecurityAccessType = requestSeed, suppressPosRspMsgIndicationBit = FALSE	01	SAT_RSD

Table 47 — SecurityAccess positive response message flow example #1

Message direction:	server → client		
Message type:	Response		
A_Data byte	Description (all values are in hexadecimal)	Byte value (hex)	Mnemonic
#1	SecurityAccess response SID	67	SAPR
#2	securityAccessType = requestSeed	01	SAT_RSD
#3	securitySeed [byte#1] = seed #1 (high byte)	36	SECHB
#4	securitySeed [byte#2] = seed #2 (low byte)	57	SECLB

Example flow

Table 48 — SecurityAccess request message flow example #1

Message direction:	client → server		
Message type:	Request		
A_Data byte	Description (all values are in hexadecimal)	Byte value (hex)	Mnemonic
#1	SecurityAccess request SID	27	SA
#2	securityAccessType = sendKey, suppressPosRspMsgIndicationBit = FALSE	02	SAT_SK
#3	securityKey [byte#1] = key #1 (high byte)	C9	SECKEY_HB
#4	securityKey [byte#2] = key #2 (low byte)	A9	SECKEY_LB

Table 49 — SecurityAccess positive response message flow example #1

Message direction:	server → client		
Message type:	Response		
A_Data byte	Description (all values are in hexadecimal)	Byte value (hex)	Mnemonic
#1	SecurityAccess response SID	67	SAPR
#2	securityAccessType = sendKey	02	SAT_SK

TesterPresent (3E hex) service

- This service is used to indicate to a server (or servers) that a client is still connected to the vehicle and that certain diagnostic services and/or communications that have been previously activated are to remain active.
- This service is used to keep one or multiple servers in a diagnostic session other than the default Session.
- This can either be done by transmitting the Tester Present request message periodically or, in case of the absence of other diagnostic services, preventing the server(s) from automatically returning to the default Session.
- The detailed session requirements that apply to the use of this service when keeping a single server or multiple servers in a diagnostic session other than the default Session can be found in the implementation specifications of ISO 14229.

Request & response message definition

Table 60 — Request message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	TesterPresent Request Service Id	M	3E	TP
#2	sub-function = [zeroSubFunction]	M	00/80	LEV_ ZSUBF

Table 62 — Positive response message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	TesterPresent Response Service Id	S	7E	TPPR
#2	zeroSubFunction	M	00	ZSUBF

Suppressed positive response

Table 67 — TesterPresent request message flow example #1

Message direction:		client → server		
Message type:		Request		
A_Data byte	Description (all values are in hexadecimal)		Byte value (hex)	Mnemonic
#1	TesterPresent request SID		3E	TP
#2	zeroSubFunction, suppressPosRspMsgIndicationBit = TRUE		80	ZSUBF

ReadDataByIdentifier (22 hex) service

- The ReadDataByIdentifier service allows the client to request data record values from the server identified by
- one or more dataIdentifiers. The client request message contains one or more two-byte dataIdentifier values that identify data record(s)
- maintained by the server (refer to C.1 for allowed dataIdentifier values). The format and definition of the dataRecord shall be vehicle-manufacturer- or system-supplier-specific, and may include analogue input and
- output signals, digital input and output signals, internal data and system status information if supported by the server.

Request message definition

Table 129 — Request message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	ReadDataByIdentifier Request Service Id	M	22	RDBI
#2	dataIdentifier[] #1 = [byte#1 (MSB) byte#2]	M	00-FF	DID_ HB
#3		M	00-FF	LB
:	:	:	:	:
#n-1	dataIdentifier[] #m = [byte#1 (MSB) byte#2]	U	00-FF	DID_ HB
#n		U	00-FF	LB

Positive response message definition

Table 131 — Positive response message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	ReadDataByIdentifier Response Service Id	M	62	RDBIPR
#2	dataIdentifier[] #1 = [byte#1 (MSB) byte#2]	M	00-FF	DID_ HB
#3		M	00-FF	LB
#4	dataRecord[] #1 = [data#1 : data#k]	M	00-FF	DREC_ DATA_1
:		:	:	:
:(k-1)+4		U	00-FF	DATA_m
:	:	:	:	:
#n-(o-1)-2	dataIdentifier[] #m = [byte#1 (MSB) byte#2]	U	00-FF	DID_ HB
#n-(o-1)-1		U	00-FF	LB
#n-(o-1)	dataRecord[] #m = [data#1 : data#o]	U	00-FF	DREC_ DATA_1
:		:	:	:
#n		U	00-FF	DATA_k

Supported negative response codes

Table 133 — Supported negative response codes

Hex	Description	Cvt	Mnemonic
13	incorrectMessageLengthOrInvalidFormat This response code shall be sent if the length of the request message is invalid.	M	IMLOIF
22	conditionsNotCorrect This response code shall be sent if the operating conditions of the server for performing the required action are not met.	U	CNC
31	requestOutOfRange This code shall be sent if: 1) none of the requested dataIdentifier values are supported by the device; 2) the client exceeded the maximum number of dataIdentifiers allowed to be requested at a time.	M	ROOR
33	securityAccessDenied This code shall be sent if at least one of the dataIdentifiers is secured and the server is not in an unlocked state.	M	SAD

ReadMemoryByAddress (23 hex) service

Table 138 — Request message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	ReadMemoryByAddress Request Service Id	M	23	RMBA
#2	addressAndLengthFormatIdentifier	M	00-FF	ALFID
#3 : #(m-1)+3	memoryAddress[] = [byte#1 (MSB) : byte#m]	M : C ₁ ^a	00-FF : 00-FF	MA_ B1 : Bm
#n-(k-1) : #n	memorySize[] = [byte#1 (MSB) : byte#k]	M : C ₂ ^b	00-FF : 00-FF	MS_ B1 : Bk
^a The presence of the C ₁ parameter depends on address length information parameter of the addressAndLengthFormatIdentifier.				
^b The presence of the C ₂ parameter depends on the memory size length information of the addressAndLengthFormatIdentifier.				

ReadMemoryByAddress (23 hex) service

Table 140 — Positive response message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	ReadMemoryByAddress Response Service Id	M	63	RMBAPR
#2 : #n	dataRecord[] = [data#1 : data#m]	M : U	00-FF : 00-FF	DREC_ DATA_1 : DATA_m

WriteDataByIdentifier (2E hex) service

Table 217 — Request message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	WriteDataByIdentifier Request Service Id	M	2E	WDBI
#2	dataIdentifier[] = [byte#1 (MSB) byte#2]	M	00-FF	DID_ HB
#3		M	00-FF	LB
#4	dataRecord[] = [data#1 : data#m]	M	00-FF	DREC_ DATA_1
:		:	:	:
#m+3		U	00-FF	DATA_m

WriteDataByIdentifier (2E hex) service

Table 219 — Positive response message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	WriteDataByIdentifier Response Service Id	M	6E	WDBIPR
#2	dataIdentifier[] = [byte#1 (MSB) byte#2]	M	00-FF	DID_ HB
#3		M	00-FF	LB

Write Memory by Address 0x3D

Table 224 — Request message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	WriteMemoryByAddress Request Service Id	M	3D	WMBA
#2	addressAndLengthFormatIdentifier	M	00-FF	ALFID
#3 : #m+2	memoryAddress[] = [byte#1 (MSB) : byte#m]	M : C ₁ ^a	00-FF : 00-FF	MA_ B1 : Bm
#n-r-2-(k-1) : #n-r-2	memorySize[] = [byte#1 (MSB) : byte#k]	M : C ₂ ^b	00-FF : 00-FF	MS_ B1 : Bk
#n-(r-1) : #n	dataRecord[] = [data#1 : data#r]	M : U	00-FF : 00-FF	DREC_ DATA_1 : DATA_r

Positive response message definition

Table 226 — Positive response message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	WriteMemoryByAddress Response Service Id	M	7D	WMBAPR
#2	addressAndLengthFormatIdentifier	M	00-FF	ALFID
#3 : #(m-1)+3	memoryAddress[] = [byte#1 (MSB) : byte#m]	M : C ₁ ^a	00-FF : 00-FF	MA_ B1 : Bm
#n-(k-1) : #n	memorySize[] = [byte#1 (MSB) : byte#k]	M : C ₂ ^b	00-FF : 00-FF	MS_ B1 : Bk
^a The presence of the C ₁ parameter depends on the address length information parameter of the addressAndLengthFormatIdentifier.				
^b The presence of the C ₂ parameter depends on the memory size length information of the addressAndLengthFormatIdentifier.				

ReadDTCInformation (19 hex) service

Table 242 — Request message definition — sub-function = reportNumberOfDTCByStatusMask, reportByStatusMask, reportMirrorMemoryDTCByStatusMask, reportNumberOfMirrorMemoryDTCByStatusMask, reportNumberOfEmissionsRelatedOBDDTCByStatusMask, reportEmissionsRelatedOBDDTCByStatusMask

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	ReadDTCInformation request Service Id	M	19	RDTCl
#2	sub-function = [reportNumberOfDTCByStatusMask reportDTCByStatusMask reportMirrorMemoryDTCByStatusMask reportNumberOfMirrorMemoryDTCByStatusMask reportNumberOfEmissionsRelatedOBDDTCByStatusMask reportEmissionsRelatedOBDDTCByStatusMask]	M	01 02 0F 11 12 13	LEV_ RNOTCBSM RDTCSM RMMDTCSM RNOMMDTCBSM RNOOBDDTCBSM ROBDDTCBSM
#3	DTCStatusMask	M	00-FF	DTCSM

Response message definition

A_Data byte	Parameter name	Cvt	Hex value	Mnemonic
#1	ReadDTCInformation response Service Id	M	59	RDTICIPR
#2	reportType = [reportNumberOfDTCByStatusMask reportNumberOfDTCBySeverityMaskRecord reportNumberOfMirrorMemoryDTCByStatusMask reportNumberOfEmissionsRelatedOBDDTCByStatusMask]	M	01 07 11 12	LEV_ RNODTCBSM RNODTCBSMR RNOMMDTCBSM RNOOBDDTCBSM
#3	DTCStatusAvailabilityMask	M	00-FF	DTCSAM
#4	DTCFormatIdentifier = [ISO15031-6DTCFormat ISO14229-1DTCFormat SAEJ1939-73DTCFormat ISO11992-4DTCFormat]	M	00 01 02 03	DTCFID_ 15031-6DTCF 14229-1DTCF J1939-73DTCF 11992-4DTCF
#5	DTCCount[] = [DTCCCountHighByte DTCCCountLowByte]	M	00-FF	DTCC_ DTCCHB
#6		M	00-FF	DTCCLB

CANdela Studio

- The specification tool CANdelaStudio is a central component of the Vector CANdela solution and supports users in creating and editing a formal vehicle ECU diagnostic specification.

CANdela Studio

- Once a diagnostic specification has been created, it is used for the following process steps and increases thereby the consistency in the entire diagnostic development process:
 - Implementing the diagnostic software
 - Automated conformity tests of the diagnostic software
 - Data supply for the various diagnostic testers in development, manufacturing and the service garage
 - Starting point for test sequences in diagnostic testers in production and the service garage

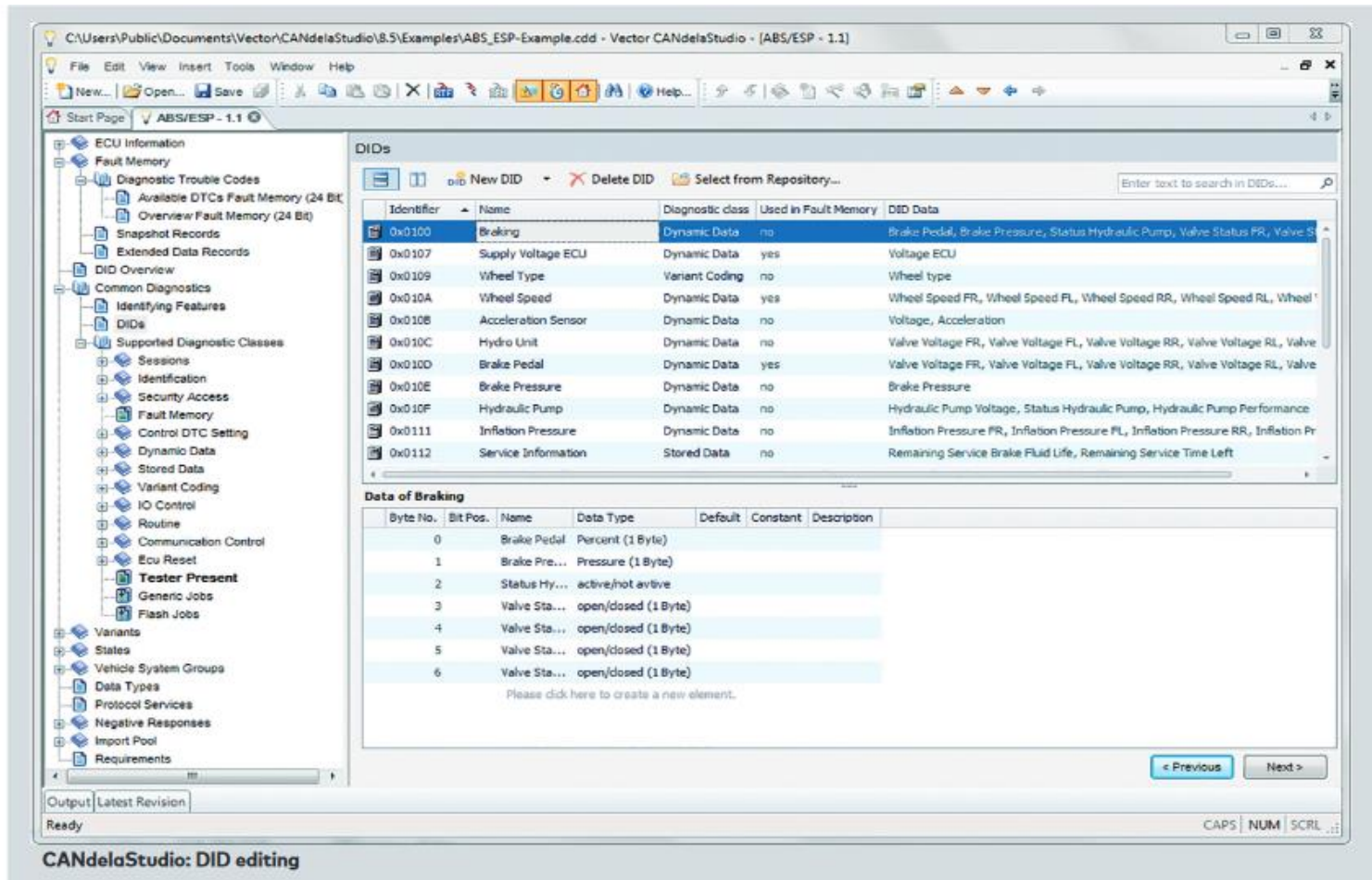
Overview of Advantages

- The user is guided by a templates concept. Despite differences in requirements, the diagnostic data is always described in a similar structure, even for different automotive OEMs. This enables reusability across projects
- Consistency checks during input assure high data quality.

Overview of Advantages

- Import and export from/to different exchange formats (ODX, CSV, RTF, HTML, A2L, XML, CDI) simplify reuse and further use of data
- Support of different protocols on different networks (UDS, KWP2000, OBD, WWH-OBD, CAN, LIN, MOST, FlexRay, DoIP)

Candela studio tool look



Gateway

- The increased complexity of automotive and industrial networks and the need for data transparency and information exchange within the overall system led to the introduction of gateways.
- Theoretically, the term gateway is not quite correctly used in automotive applications. In the literature, the term „gateway” is used for a network node of a communication network equipped for interfacing with another network that uses different protocols.
- It may contain devices such as protocol translation, rate converters and signal translators to provide system interoperability.

Gateway

- In automotive and industrial control applications, the term gateway is preferred even though the data is transferred between networks using the same protocol, because these gateways perform more functions than the forwarding of messages.

Gateway Functions

- Message filtering (to prevent the overload of a low speed network when transferring messages from a high-speed network)
- message transfers with identifier translation, message integration (combining parts of the data of several messages into a new message)
- the synchronization of time-triggered networks (when implemented) to guarantee that the information is updated on time.

Gateway Implementation

- the gateway functionality could be implemented in software, as long as several CAN modules are available in the ECU. But a large amount of messages would cause a high load on the CPU, leaving less performance for the ECU control applications until real-time operation can no longer be guaranteed

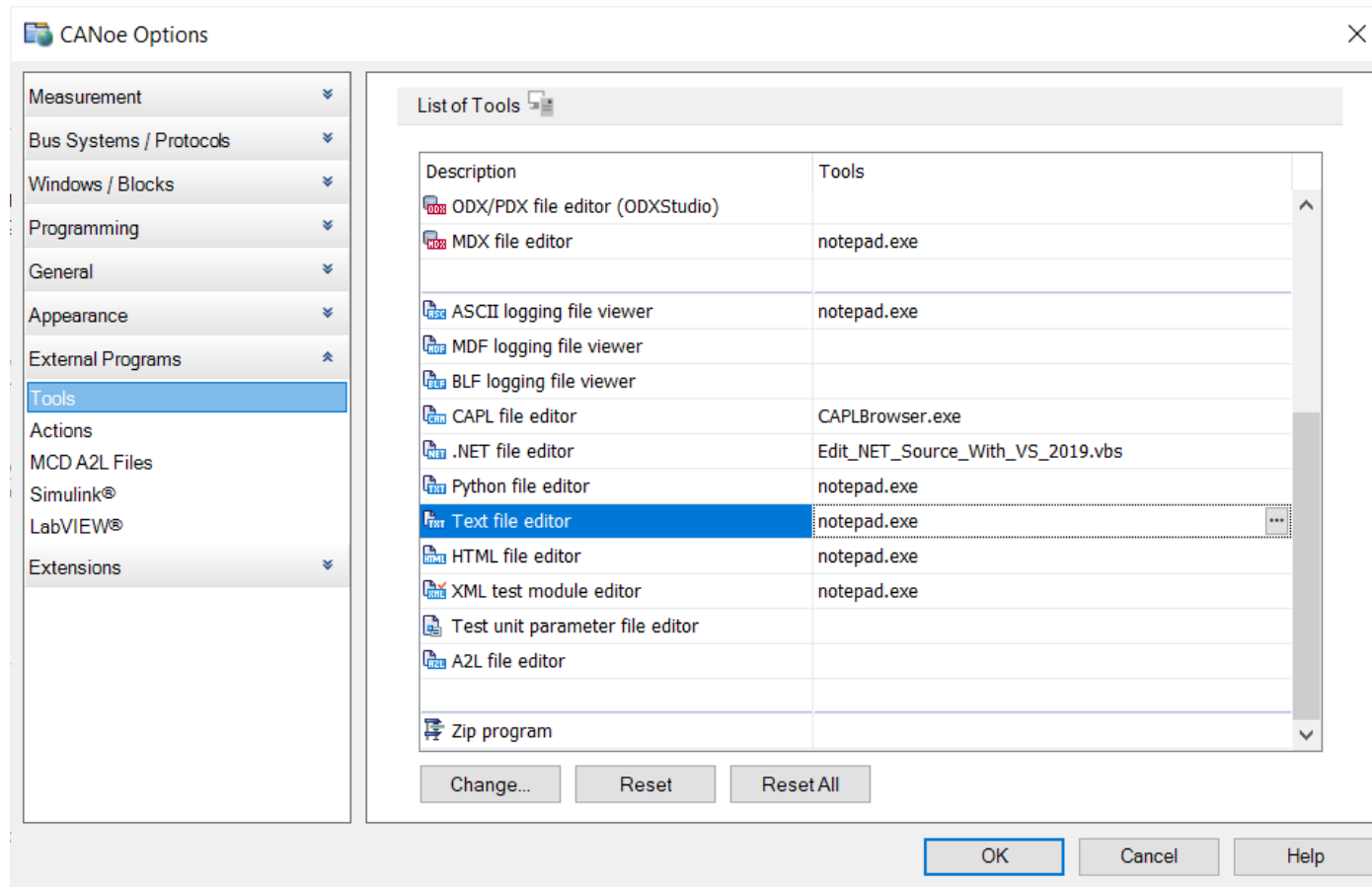
Gateway CAPL

- on message CAN1.*
{
 message CAN2.* m;
 if(this.DIR==RX) // if it is a received frame
 {
 if(this.CAN==1)
 {
 m=this;
 output(m); // send it to the other channel
 }
 }
}

Using CANoe .net API

- The CANoe environment provides a .NET API to be used for simulation, test, and snippet programming.
- The CANoe .NET API is an Embedded Domain Specific Language extension that offers the possibility to use object-oriented programming languages, e.g. C# in the CANoe environment.
- .NET languages provide extended capabilities to structure, to reuse and to debug programs.

Configuring .Net file editor



CANoe .NET API Components

- CANoe programming interface
- Type libraries

Type libraries

- Vector.Tools
- Vector.Tools.Internal
- Vector.CANoe.Runtime
- Vector.CANoe.Runtime.Internal
- Vector.CANoe.TFS
- Vector.CANoe.TFS.ITE
- Vector.CANoe.TFS.Internal
- Vector.CANoe.Threading
- Vector.Diagnostics
- Vector.Scripting.UI
- Vector.CANoe.VTS
- Vector.CANoe.Sockets
- <demo>.dll
- <configuration name>.cfg_sysvars.dll
- <system variable file>.dll
- <program name>_CapLibraries.dll

.NET Editor Selection

- Visual Studio .NET 2005 - 2013 can be used as IDE for .NET programs in CANoe.
- All IDEs are supported in CANoe by Visual Basic scripts that facilitate the automatic creation of solutions, project references, access to type libraries etc.
- To configure the .NET editor to be used in CANoe use the Options dialog (menu: Configuration | Options | External Programs | Tools | .NET file editor) and select the script (vbs file) matching your Visual Studio version. The available scripts for Visual Studio are delivered with the CANoe installation and located in the folder
- **C:\Program Files\Vector CANoe 15\Exec32\Scripts**

.NET API Components

- .NET Programs in CANoe
- .NET Test Modules
- .NET Simulation Nodes
- .NET Snippets
- .NET Test Libraries

Programming with the CANoe .NET API

- Signals (bus signals, environment and system variables)
- Messages
- Timer
- Event Procedures

Test Features

- Test Module Types
 - Unstructured test modules
 - Structured test modules

.Net Structured test modules

```
public class MyTestModule : StructuredTestModule
{
    public override void StructuredMain()
    {
        DynamicTestGroup();
        TestGroupBegin("Id", "Title", "Description");
        TestCase1();
        TestCase3();
        TestGroupEnd();
    }
}
```


.Net Structured test modules

```
[TestCase("Test case title")]

public void TestCase1()
{

    Report.TestStep("Description of the test step");

}
[TestGroup("Dynamic Test case sequence", "depends on preceding test case results")]

public void DynamicTestGroup()
{

    TestCase1();
    if (TestModule.VerdictLastTestCase == Verdict.Passed)
    { TestCase2(); // ... }

}
```

Reporting Commands

- `Report.TestCaseComment(...)`
- `Report.TestStep(...)`
- `TestStepFail(...)`
- `TestStepPass(...)`
- `TestStepWarning(...)`

Wait Points

- Simple timeout: `Execution.Wait(50);`
- Bus signal: `Execution.Wait(1);`
- System variable: `Execution.Wait(2);`
- Environment variable: `Execution.Wait(EnvDoorLocked.Locked);`
- Message: `Execution.WaitForCANFrame(ref frame, 500);`

Checks

- `Vector.CANoe.TFS.ValueCheck`
 - is used to monitor the value criteria of bus signals, environment and system variables
- `Vector.CANoe.TFS.AbsoluteCycleTimeCheck`, `Vector.CANoe.TFS.RelativeCycleTimeCheck` and `Vector.CANoe.TFS.OccurenceTimeCheck`
 - check the timing of messages
- `Vector.CANoe.TFS.DlcCheck`
 - monitors the message length indication
- `Vector.CANoe.TFS.UserCheck`
 - class is used to monitor customized system conditions.

User Dialogs

- **Tester Confirmation Dialog:**

```
int result = Execution.WaitForConfirmation("Please confirm!");
```

or

```
int result = ConfirmationDialog.Show(("Please confirm!", "Confirmation Dialog");
```

References

- CAN Specification v2.0 sep 1991 , 1991, Robert Bosch GmbH, Postfach 30 02 40, D-70442 Stuttgart
- CANoe_Manual_EN.pdf version 4.1, Vector Informatik GmbH
- ProgrammingWithCAPL.pdf , December 14 2004, Vector CANtech, Inc
- https://en.wikipedia.org/wiki/CAN_bus
- Road vehicles — Unified diagnostic services (UDS) — Specification and requirements, ISO14229, Second edition 2006-12-01

Thank you