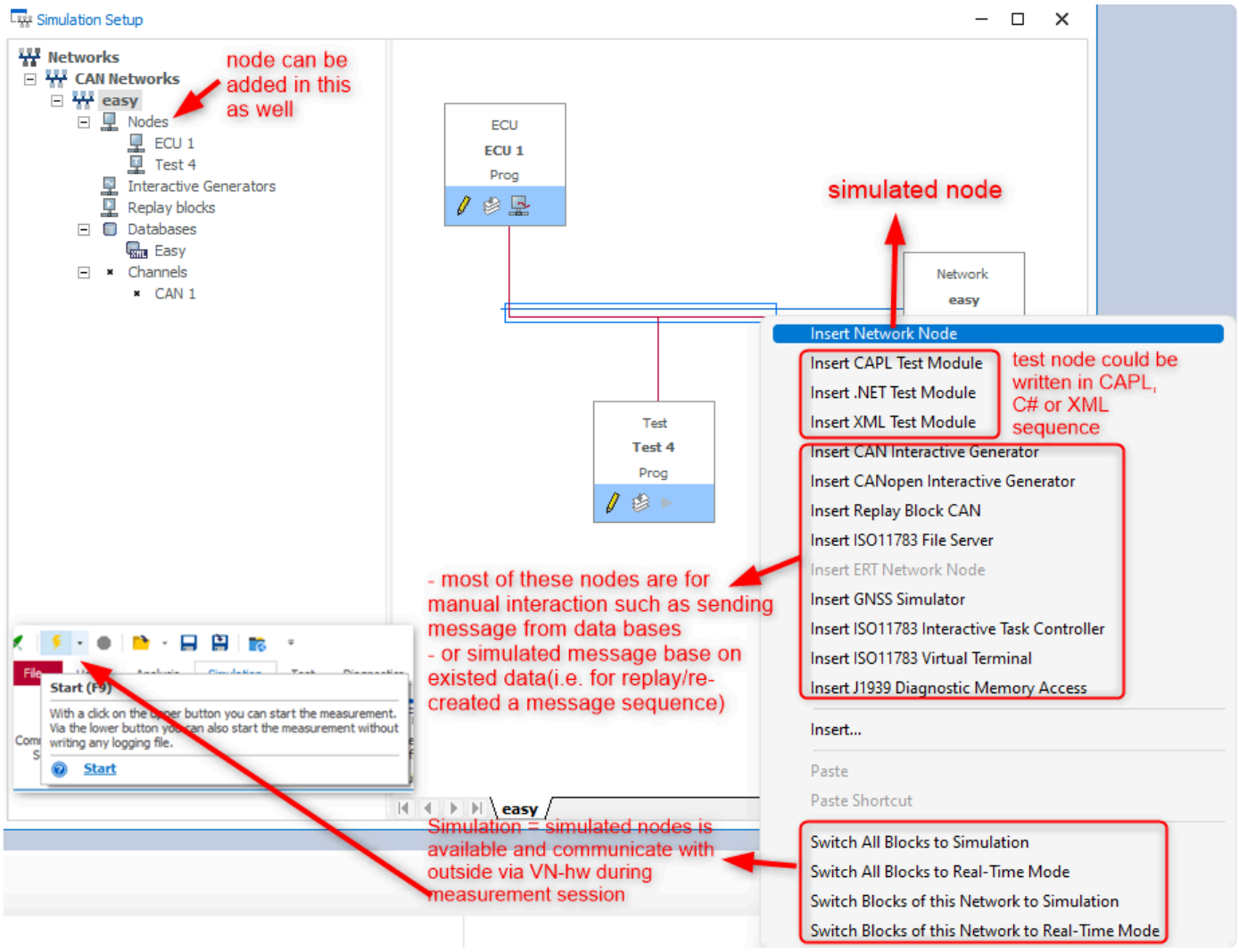
CAPL scripting

October 12th, 2022 in programming
Last updated on: December 28th, 2022

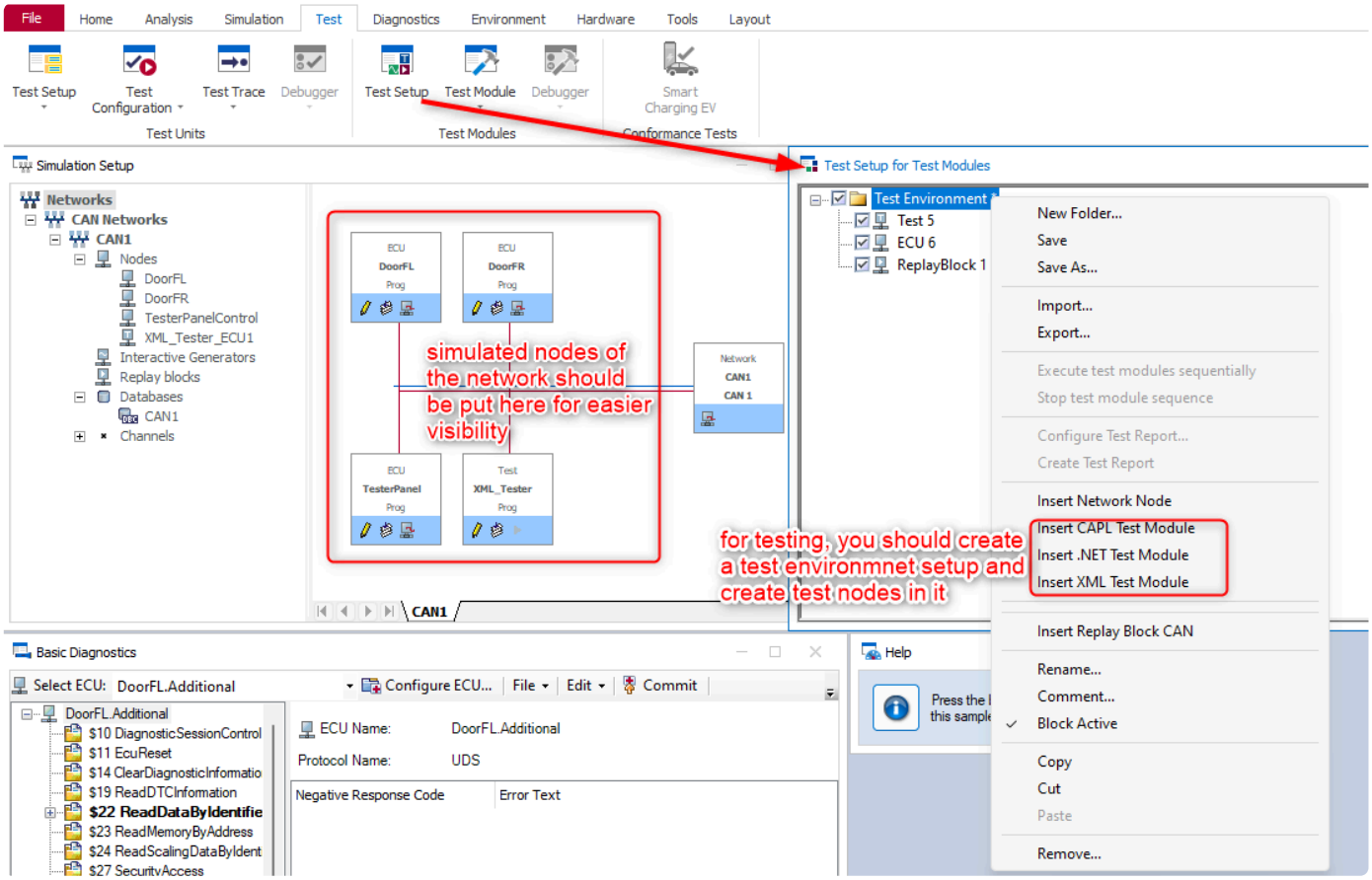
CAPL is an acronym for Communication Access Programming Language, which is a programming language used in Vector testing tools chain. It is used to create:

- Simulation network nodes
- Measurement, analysis components
- Testing modules

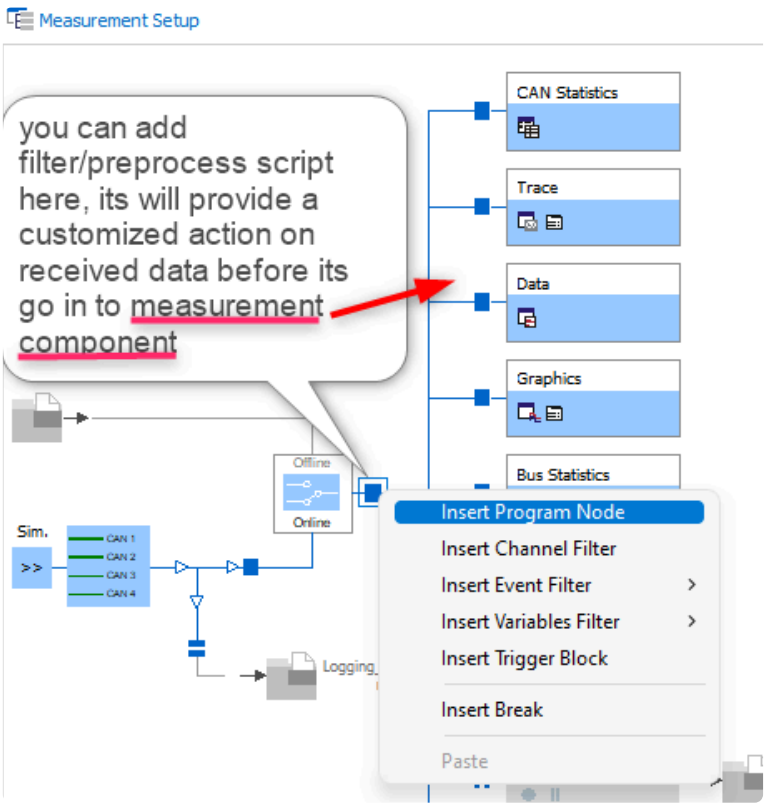
> **NOTE: All of the images in this article are captured from Vector CANoe tool and their example projects.*



simulation node in simulation window



test nodes in test environment window



measurement node in measurement window

Some characteristics of CAPL script are:

- C like syntax.
- Event-driven (individual functions(evaluate, set signal values, send messages) that react to received messages, expired timers event, change of a signals or system variables in the environment).
- CAPL program can be developed(.can, .cin) and compiled(.cbf) using Vector tool CAPL Browser(IDE).
- Direct access to messages, signals, system variables and diagnostic parameters(via database file such as DBC, CDD, ODX-d) because CAPL Browser is integrated with CANoe,CANalyzer tool chain.
- Can link to user dll(e.g. diagnostic operation: encryption API, security key/seed calculation).

> *.can: is execution code and compiled*
> *.cin: is for common/library function, and shall be included by .can*

I. Program Structure

Sample CAPL script - simulation node that monitor CAN message and update the sysvar value to be displayed on CANoe Panel

C

```
/*@@var:*/
variables
{
   const long kOFF = 0;
   const long kON = 1;

   int gDebugCounter = 0;
}
/*@@end*/

/*@@startStart:Start:*/
on start
{
   setwriteDbgLevel(0); // set DbgLevel = 1 to get more information in Write-Window
   write("Press key 1 to set DbgLevel = 1 for more information in the Write-Window");
   write("Press key 0 to set DbgLevel = 0 for less information in the Write-Window");
}
/*@@end*/

/*@@key:'0':*/
on key '0'
{
   setwriteDbgLevel(0);
}
/*@@end*/
```

```
/*@@key:'1':*/
on key '1'
{
   setwriteDbgLevel(1);
}
/*@@end*/

/*@@msg:CAN1.easy::EngineState (0x123):*/
on message EngineState
{
   // engine state received
   // "this" keyword is used to access to received message EngineState
   if (this.dir == RX)
   {
      // get raw value "EngineSpeed" from "this" message, which is EngineState (0x123)
      // convert to physical value and store to system variable EngineSpeedDspMeter, which ca
      @sysvar::Engine::EngineSpeedDspMeter = this.EngineSpeed / 1000.0;
   }
}
/*@@end*/

/*@@msg:CAN1.easy::LightState (0x321):*/
on message LightState
{
   gDebugCounter++;

   if (this.dir == RX)
   {
      SetLightDsp(this.HeadLight, this.FlashLight);

      if(gDebugCounter == 10)
      {
         // write debug text to Write-Window if the DebugLevel is set and smaller/equal to 1
         writeDbgLevel(1,"LightState RX received by node %NODE_NAME%");
         gDebugCounter = 0;
      }
   }
   else
   {
      if(gDebugCounter == 10)
      {
         writeDbgLevel(1,"Error: LightState TX received by node %NODE_NAME%");
         gDebugCounter = 0;
      }
   }
}
/*@@end*/

/*@@caplFunc:SetLightDsp(long,long):*///function
void SetLightDsp (long headLight, long hazardFlasher)
{
   if(headLight == kON)
   {
      if(hazardFlasher == kON)
      {
         // @sysvar is global sysvar namespace, then Lights namespace and LightDisplay variabl
```

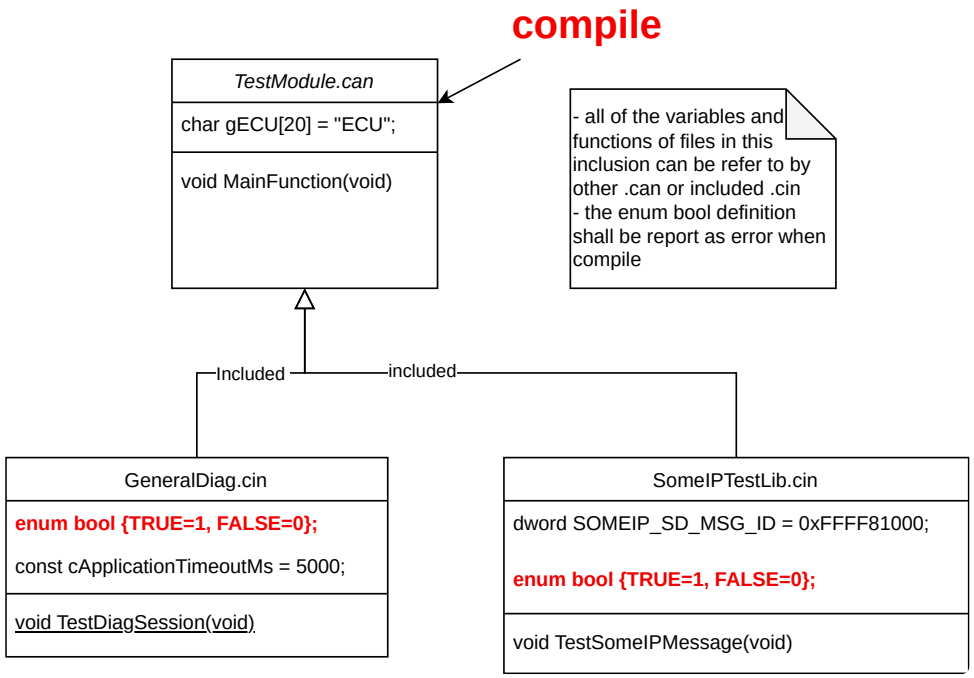
```
        // syntax similar to C++
        @sysvar::Lights::LightDisplay = 7;
      }
      else if(hazardFlasher == kOFF)
      {
        @sysvar::Lights::LightDisplay = 4;
      }
   }
   else if(headLight == kOFF)
   {
      if(hazardFlasher == kON)
      {
        @sysvar::Lights::LightDisplay = 3;
      }
      else if(hazardFlasher == kOFF)
      {
        @sysvar::Lights::LightDisplay = 0;
      }
   }
}
/*@@end*/
```

Above sample script shall be stored in a *.can file, which then shall be complied with CAPL Compiler into executable file in CBF(CAPL Binary Format *.cbf). For a test node, main section of CAPL test script are:

I.1 include

- Include files contain arbitrary but complete sections of a CAPL program: includes, variables and functions. All variables and functions that are included via the Include files will be available as global variables and functions.
- The sequence of inclusion is irrelevant. The compiler reports any duplicate symbols as an error between included files and between included files and the main program.
  Code and data from included and parent files may use each other mutually.
- One exception to the prohibition of duplicate symbols is that on start, on preStart, on preStop and on stopMeasurement may coexist in both the included file and the parent file. In these functions, the code is executed sequentially: first the code from the included file and then the code from the

declare data types, define variables and provide an (inline) function library.



I.2 variable

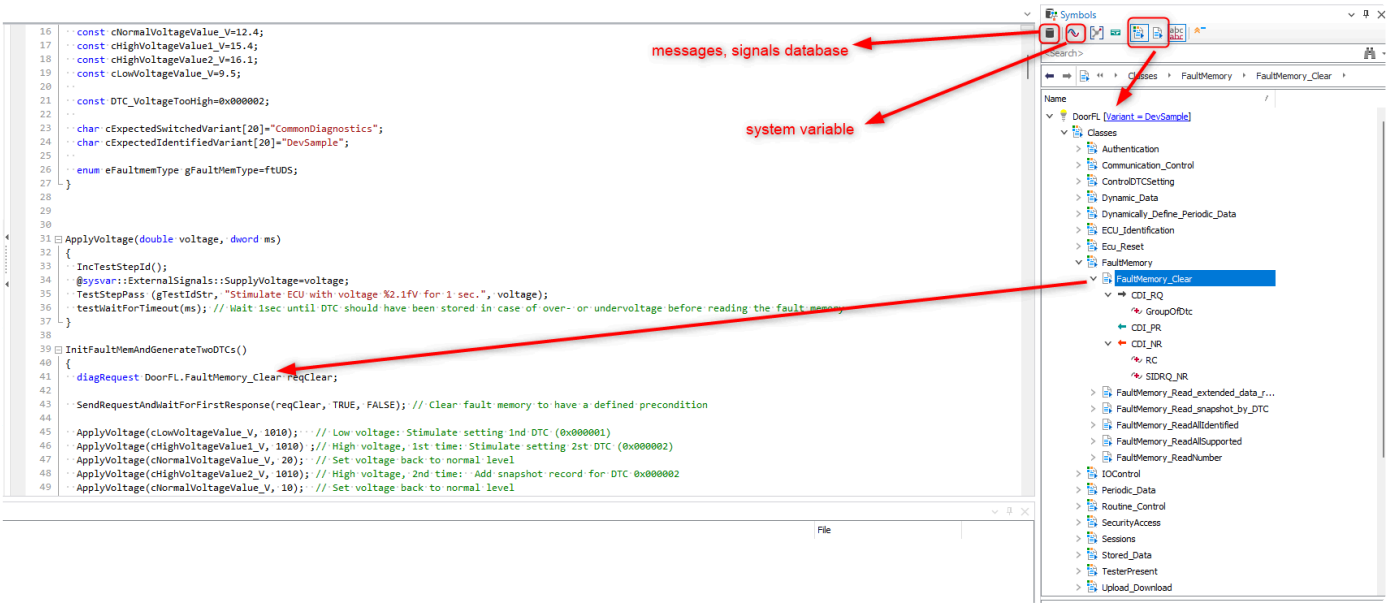
- CAPL program variables can be declared in each function as local variable or under variables section as global variables.

> *NOTE: CAPL only allow declare local variable at the beginning section of the function.*

C

```
variables
{
// similar to C language, CAPL provide some primitive type such as:
byte var1; // unsigned, 1 Byte
word var2; // unsigned, 2 Byte
dword var3; // unsigned, 4 Byte
qword var4; // unsigned, 8 Byte
int i, j = 2; // i = 0, signed 2 Byte
long var5; // signed, 4 Byte
int64 var6; // signed, 8 Byte

float f2 = 11.0; // 8 Byte
double f = 12.2; // 8 Byte

char array[12] = "char"; // array of 12 ascii char, 1 Byte each

// structure, enum and Associative Fields(dictionary, key value data type)
_align(1) struct streamListener
{
   struct
   {
      byte MAC[6];
      byte SSID[2];
```

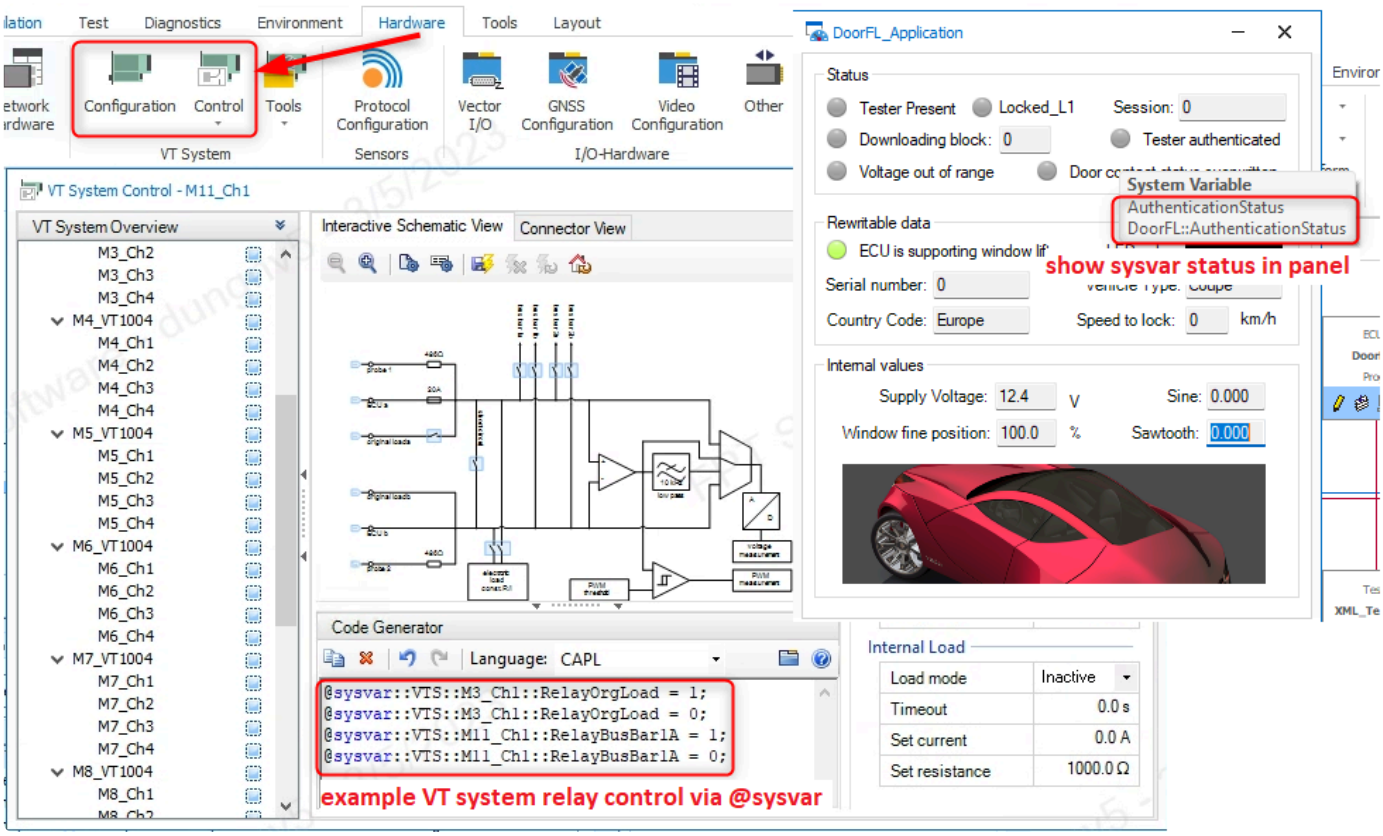
```
   } stream_ID;
   byte sink_ID;
   byte multicast_address[6];
   byte channel_ID;
   byte stream_state;
};
struct streamListener stream01; // declare variable of type `struct streamListener`

struct UDP_Data
{
     ip_Endpoint ep_source;
     ip_Endpoint ep_destination;
     dword mac_source;
     dword mac_destination;
     dword vlan;
};
struct UDP_Data gSomeIP_UDPList[long]; // data types for the keys can be `long`, `int64`, `f
// loop through elements in dicts
for (long msg_id: gSomeIP_UDPList)
{
   // process each one gSomeIP_UDPList[msg_id]
}

enum bool {TRUE = 1, FALSE = 0}; // define enum type
enum bool result; // declare variable of type `enum bool`

// Frequently used CANoe classes and objects are predefined, declaration of these objects ty
diagRespone * req;
diagRequest DoorFL.FaultMemory_Clear reqClear; // ECU_name.DiagService_name << this reference

linFrame * linMsg;
message 0x701 msg;

}
```



- Struct, array, associative field, object, when used as function parameter will be passed as reference by default.
- Other primitive data type has to used & to mark that it shall be passed as reference. E.g. void Function(byte& ref); Function(var);, and implicit conversion, e.g., from byte to long, is not possible.

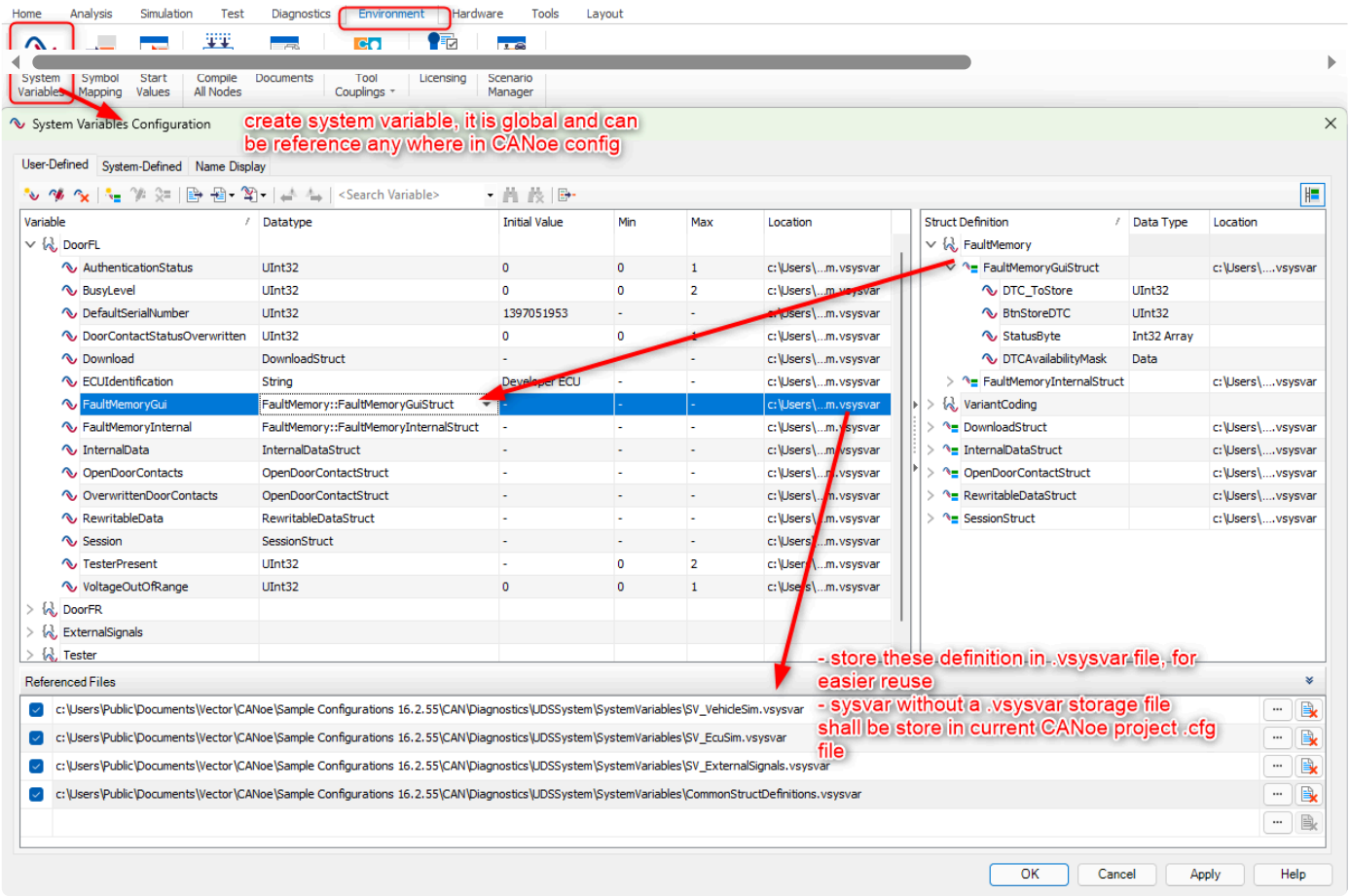
> Some C concept, function such as align, memcpy, memcmp are also available and greatly help the testing operation(e.g. parse datastream to/from data structure) for user defined data type.
>
> › memcpy, memcpy_h2n, memcpy_n2h, memcpy_off, memcmp
>
> › _align(1) struct streamListener{...}, __size_of(struct streamListener), __alignment_of(struct streamListener), __offset_of(struct streamListener, sink_ID)

- sysvar or System Variables are defined under tab Environment/System Variables, it can be accessed everywhere in CANoe configuration. We define **user sysvar to shared, route data around testing environment between test nodes, simulated nodes, measurement nodes, CANoe panel**.
- CANoe also provide sysvar so test nodes can interact with other component in a testing environment such as VT System.

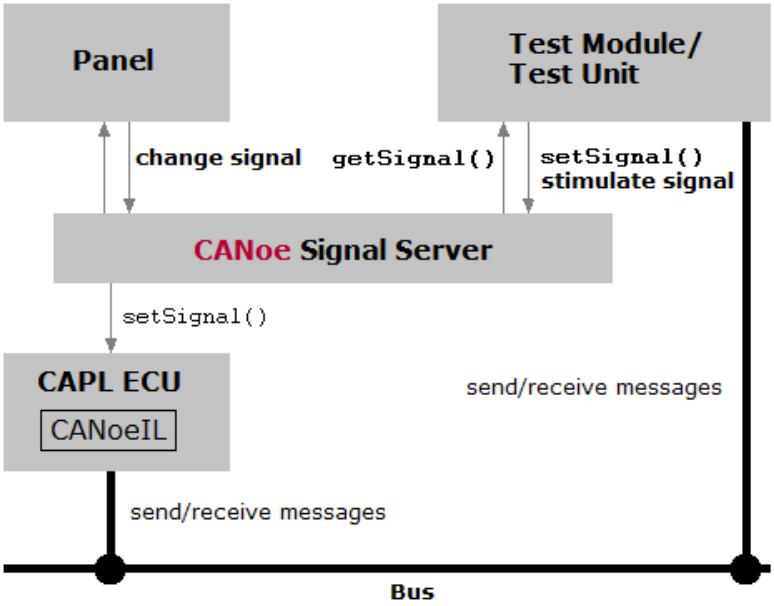



```c
on sysvar DoorFL::RewritableData.MirrorSettings.Availability
{
    // on sysVar is called only when the value of the variable changes. It can also be wri
```

```
}
on sysvar_update DoorFR::AuthenticationStatus
{
      // use on sysVar_update, if you want to be notified of value updates to the variable w
}
// some of the ways to manipulate sysvar value
@sysvar::DoorFL::AuthenticationStatus = 1;
authenStatus = @sysvar::DoorFL::AuthenticationStatus;

testResetSysVarValue(sysvar::DoorFL::AuthenticationStatus); // signal as function parameter sha
testWaitForSysVar(sysvar::DoorFL::AuthenticationStatus, 1000);
dwordValue = sysGetVariableDWord(sysvar::DoorFL::InternalData);
sysSetVariableDWord(sysvar::DoorFL::InternalData, 0x022);
```



- signal representation of the bus signals. Access to the signals is carried out with the syntax $signal, in addition you can specify the read/write operation either be raw or phys $signal.raw, $signal.phys
- keep in mind, signal value does not change immediately, only after the signal is being transmitted again on the network then it can be said to be updated, and read out value is the last value transmitted on the network.



signal access concept with CANoe restbus simulation

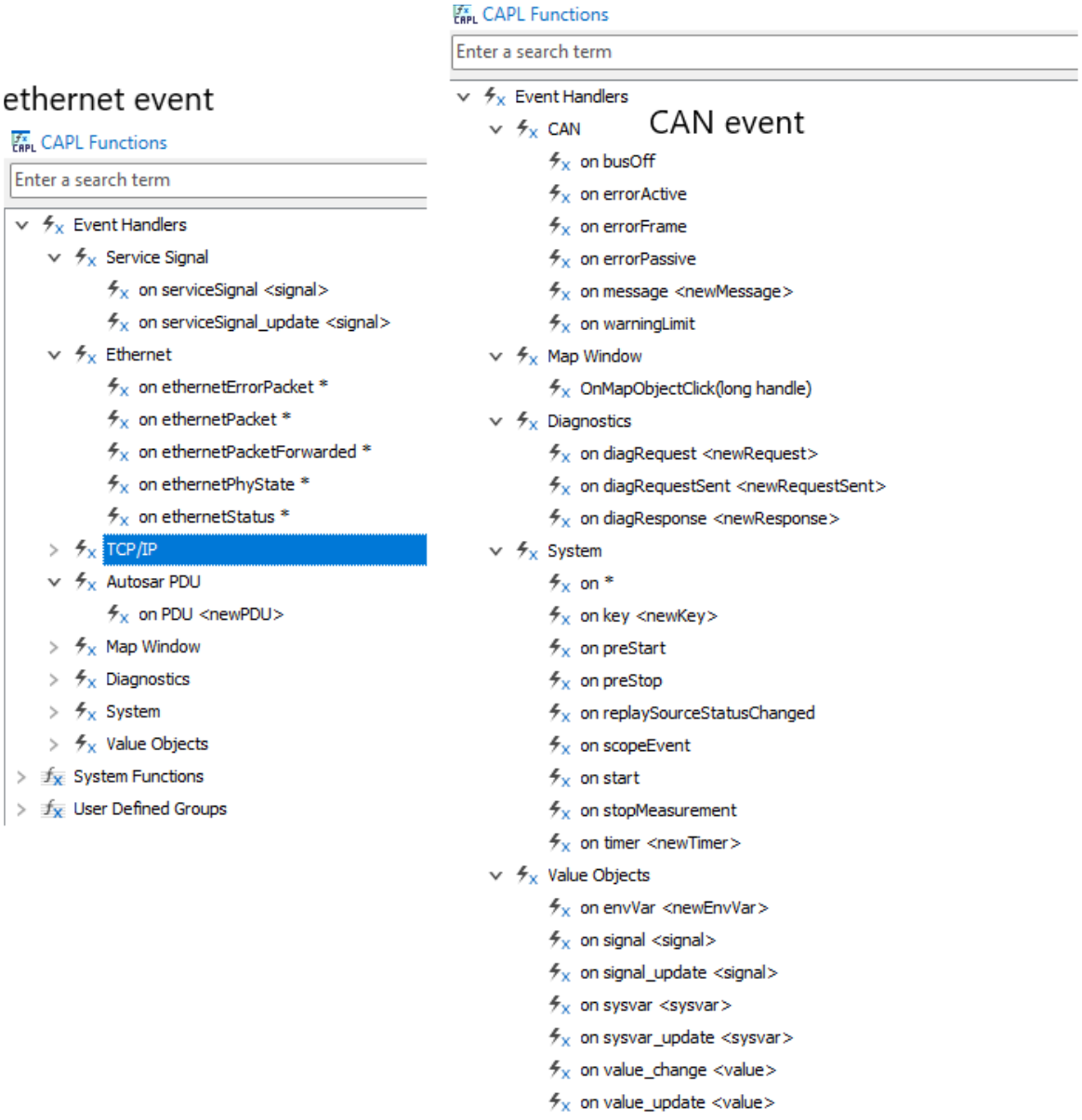
> *CAPL ECU(CANoeIL) is a restbus simulation node(generated from DBC file, in which message/signals information of each nodes in the network are defined). It shall stimulate the message transmission on the bus, which we can update message signal value in test node and create stimulated input for the DUT.*
>
> *In case of your configuration is created manually, no CANoeIL simulation node and either with or without DBC, you have to implement the callback functions as a signal transmission/receive driver by your own if you want to manipulate signal as below.*

C

```
// Database objects are identified with the following syntax

// Network access:
// [dbNetwork::]DBName
// Node access:
// [[dbNetwork::]DBName::][dbNode::]NodeName
// Message access:
// [[dbNetwork::]DBName::][[dbNode::]NodeName::][dbMsg::]MessageName
// Signal access:
// [[dbNetwork::]DBName::][[dbNode::]NodeName::][[dbMsg::]MessageName::][dbSig::]SignalName

setSignal($EngineSpeed, 200);
// after set signal value, wait for it to be updated
testWaitForSignalUpdate($EngineSpeed, 1000);

var1 = readSignal($EngineSpeed);
var = $EngineSpeed.phys;
```

```
// wait for signal change to certain condition
testWaitForSignalChange($EngineSpeed, 1000);
testWaitForSignalInRange($EngineSpeed, 300, 400, 1000)
testWaitForSignalMatch($EngineSpeed, 500, 1000);

// signal as function parameter
foo ( signal * s )
{
    write("Signal value: %g", $s);
}

// similar to sysvar, you can also define special event procedures that are called as soon
on signal LightSwitch::OnOff // or on signal_change LightSwitch::OnOff
{
   v1 = this.raw;
   v2 = $LightSwitch::OnOff.raw;
   // v1 and v2 are the same, value of LightSwitch::OnOff signal in this function can be get
}
on signal_update signalname // called with every signal reception
on signal ( signalname1 | signalname2 | ...) // handling several signals
// /Help01/CANoeCANalyzerHTML5/CANoeCANalyzer.htm#Topics/CANoeCANalyzer/Test/TestFeatureSet/TFSSi
// /Help01/CANoeCANalyzerHTML5/CANoeCANalyzer.htm#Topics/CANoeCANalyzer/LibrariesPackages/VectorI
```

1.3 on <event>



CAPL event available in CAPL editor

Except for the testcases, which can be called on test execution, all of the CANoe nodes are operated base on event processing.
In a test nodes most of the time, you don't need to define the event handler, because CANoe provide a very well defined API libraries, that we can use to create a sequential testing operation (e.g. TestWaitFor* APIs to wait for certain event to occur).

TestWaitForRawSignalMatch	Checks the given raw value against the value of the signal. The resolution of the signal is considered.
TestWaitForSignalChange	Waits for an event from a signal which value is changed.
TestWaitForSignalInRange	Checks if the signal, the system or the environment variable value is within or outside a defined value range.
TestWaitForSignalOutsideRange	
TestWaitForSignalMatch	Checks if a given value matches the value of the signal, the system variable or the environment variable.
TestWaitForSignalUpdate	Waits for an event from a signal.
TestWaitForSysVar	Waits for the next system variable.
TestWaitForStringInput	Creates a dialog in which the tester can enter a text.
TestWaitForSyscall	Starts an external application and check its exit code.
TestWaitForTesterConfirmation	Creates a popup window and waits for tester confirmation.
TestWaitForTextEvent	Waits for the signaling of the specified textual event from the individual test module.
TestWaitForTimeout	Waits until the expiration of the specified timeout time.
TestWaitForTimeoutSilent	Waits until the expiration of the specified timeout time. The function does not write in the Test Feature Set report.
TestWaitForUserFileSync	Starts synchronization of user files between client and server system in a distributed environment.
TestWaitForValueInput	Creates a dialog in which the tester can enter a number.
CAN Functions	**Short Description**
TestWaitForMessage	Waits for the occurrence of a specified message.
TestWaitForSignalAvailable	Tests the availability of a specific signal and waits if necessary until its availability.
TestWaitForSignalsAvailable	Tests the availability of all the send signals of a node and waits if necessary until all the send signals of the node are available.
Ethernet Functions	**Short Description**
TestWaitForEthernetLinkStatus	Waits for an Ethernet link status.
TestWaitForEthernetPacket	Waits for the occurrence of the first Ethernet packet matching the conditions passed as arguments.
TestWaitForEthernetPhyState	Waits for the occurrence of the specified Ethernet PHY state.

CAPL TestWaitFor* APIs - Help01/CANoeCANalyzerHTML5/CANoeCANalyzer.htm#Topics/CAPLFunctions/Test/CAPLfunctionsTFSOverview.htm

But some time, you might need access to a lower layer level of data, which can not be provided by CANoe APIs or you just want to create a custom procedure to filter event, in that cases define these `on <event>` handle and a notification mechanism(e.g. using sysvar, text event notification) will help you a lot during testing process.

C

```c
// handling event on SimulatedHeadUnit node port in EthernetNetwork bus
on ethernetPacket ethernetPort::EthernetNetwork::SimulatedHeadUnit.*
{
      ip_Endpoint endpoint_src;
      ip_Endpoint endpoint_des;
      // filter vlanid 1 for someip, filter udp packet
      if ((this.GetVlanId() == 1) && this.udp.IsAvailable())
      {
            // process further message, store required data into global variables or system va
            testSupplyTextEvent("Event_ServiceDiscovery");
      }
}

void checkSomeIPSD(void) // << test function, which is executed sequentially.
{
      testWaitForTextEvent("Event_ServiceDiscovery", 5000); // wait for SOMEIP Service discover
      // process data received from `on <event>`
}
// Help01/CANoeCANalyzerHTML5/CANoeCANalyzer.htm#Topics/Shared/CAPL/General/EventProceduresOvervi
```

Above is a sample on how to create/handle your event and integrate it into your testing sequence.

I.4 functions

Syntactically, function definition and usage in CAPL is the same as C.
Beside that CAPL provides some advanced programming mechanism such as:

- Overloading of functions (i.e. multiple functions with the same name but different parameter lists).

```c
int diagSendRequestWaitResult(diagRequest *req)
{
   ...
}
int diagSendRequestWaitResult(diagRequest *req, dword reqTimeout, dword respTimeout)
{
   ...
}
```

- Function delegate: Instead of passing the name of a CAPL function explicitly defined elsewhere in the program as a parameter to a predefined function (e.g. consumedMethodRef::CallAsync), you can also define a small function directly at the position of the parameter. This function has no name and is called a delegate (in other programming languages you will also find the terms lambda expression or anonymous function).

```c
MirrorAdjustment.consumerSide[CANoe,LeftMirror].Adjust.CallAsync(-50, 0,
     delegate (callContext MirrorAdjustment.Adjust cco1)
     {
        write("Call returned with result %d", cco1.Result);
     }
);
// Help01/CANoeCANalyzerHTML5/CANoeCANalyzer.htm#Topics/Shared/CAPL/General/Delegates.htm
// Help01/CANoeCANalyzerHTML5/CANoeCANalyzer.htm#Topics/CAPLFunctions/CommunicationObjects/Methoc
// Help01/CANoeCANalyzerHTML5/CANoeCANalyzer.htm#Topics/CAPLFunctions/CommunicationObjects/CAPLfu
// Help01/CANoeCANalyzerHTML5/CANoeCANalyzer.htm#Topics/CANoeCANalyzer/CommunicationConcept/Progr
```

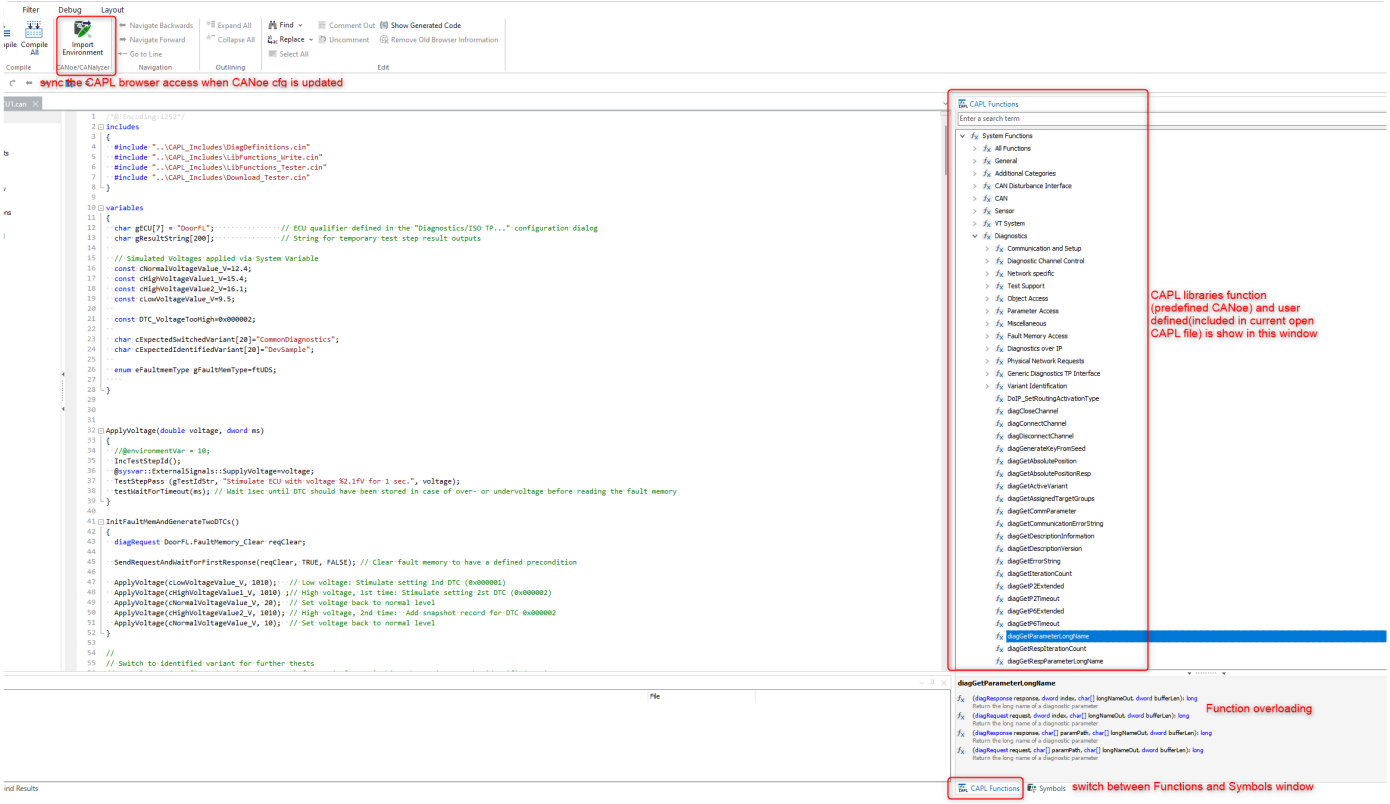
A test function/control function is a predefined test procedure that is parameterized with concrete parameter values for execution. These are the most frequently used test procedures that are occupied with values in an XML test module and brought to

execution. The execution of the function is noted in the test report and eventually leads to a verdict change of the surrounding test case.

C

```
testfunction testEnvPreparation() // prefix testfunction keyword
{
   // << this "testEnvPreparation" can be reference in other test cases,
   // test sequences of XML test node, VTest studio test sequence,..
}

// sample of xml test node call CAPL testfunction and testcase
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<testmodule title="Diag Test" version="">
     <variants>
          <variant name="ECU_Premium">ECU_Prem</variant>
          <variant name="ECU_Basic">ECU_Basic</variant>
     </variants>
     <preparation>
          <vardef name="variant_no" type="int" default="0">0</vardef>
          <varset name="variant_no" variants="ECU_Premium">0</varset>
          <varset name="variant_no" variants="ECU_Basic">1</varset>
          <capltestfunction name="testEnvPreparation" title="Test environment preparation">
             <caplparam name="variant" type="int"><var name="variant_no"></caplparam>
          </capltestfunction>
     </preparation>

     <testgroup title="Diagnostic">
          <capltestcase name="diagnostic_001_opr_normal_load" title="test output diagnostic op
     </testgroup>

     <completion>
          <capltestfunction name="testEnvCleanup" title="Test environment cleanup"/>
     </completion>
</testmodule>
```

CAPL also come with a library of predefined(intrinsic) functions. For the selection of a predefined function, the CAPL browser makes available the CAPL Function Explorer.



CAPL library functions

I.5 testcases

C

```
testcase diagnostic_001_opr_normal_load() // prefix testcase keyword
{
   /*    TC variables    */
   int result = PASS;
   byte diagnosticTestResult = enDiagActType_NA;
   /*    TC setup    */
   tcSetup("diagnostic_001_opr_normal_load", "Output actuator diagnostic - normal load conditio
   tcDescription("Trigger diagnostic test of actuator incase of normal load ");

   traceability(""); // CAPL built-in function: testReportAddMiscInfo
   environment("");   // CAPL built-in function: testReportAddSUTInfo
   precondition(""); // CAPL built-in function: testReportAddMiscInfo

   /* TC pre-condition setup    */
   if (newTestStep("Preparation", "Power up board, Init diag", "Diag session available"))
   {
      /* specific precondition setup for test environment hardware and DUT */
      tcPreRun(enLabType_SETUP_A);
      /* init diag, change to extended session */
      InitDiagSession();
      ChangeDiagSession(enDiagSessionType_EXTENDED);
   }

   /* Test step 1 */
   if (nextTestStep("Send request service 0x31 - RID 0xFFFE with TargetActuator = 0x01", "Rec
   {
```

```
      if (PASS == RequestDiagnosticActuator(enActuatorType_ACTUATOR_01, &diagnosticTestResult))
      {
         if (diagnosticTestResult == enDiagActType_NORMAL)
         {
            testStepPass("Output diag test as expected");
         }
         else
         {
            testStepFail("Output diag test is different compare to expected");
         }
      }
   }

   /* TC post-run clean up */
   tcPostRun(enLabType_SETUP_A);
}
```

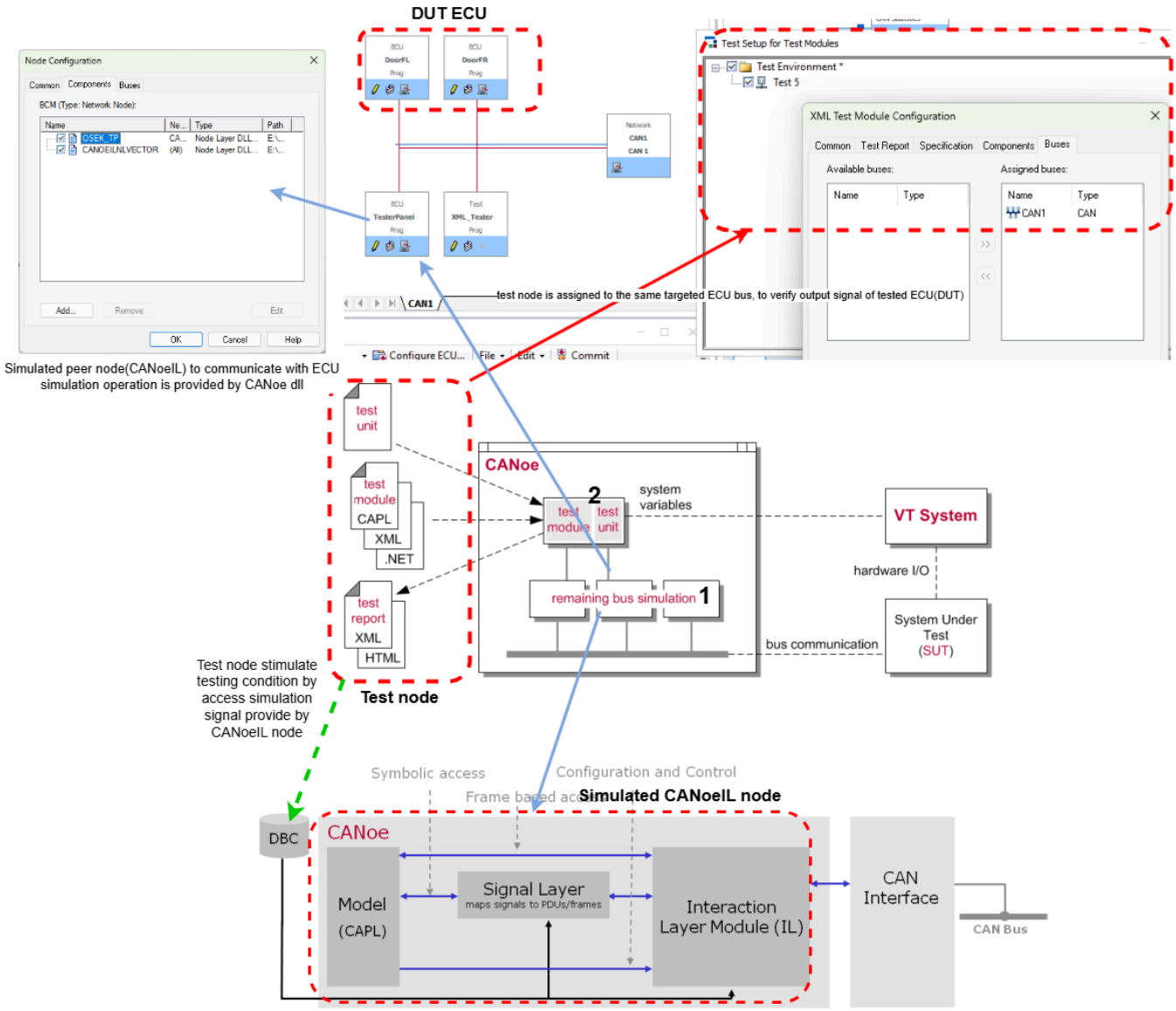
Above is a sample testcase structure

- testcase description and traceability information
- testcase precondition setup
- test steps
- testcase post-run clean up

These testcase can be lated called by CAPL test node MainTest or XML test node.

II. Execution Concept

A key difference between CAPL and C or C++ relates to when and how program elements are called. In C, for example, all processing sequences begin with the central start function main().
In CAPL, on the other hand, a program contains an entire assortment of procedures of equal standing, each of which reacts to external events.



CANoe test environment

Above is a typical test environment CANoe configuration.

1. Simulated peer node : generated directly from input database with add-on “Model Generation Wizard” tool or can be created manually. It provide signal simulation base on database via provided CANoeIL DLL.
2. User define test node : this is where we implement test script, trigger test signal via simulated peer node signal driver(CANoeIL) and CANoe VT system. Then verify response of ECU via its feedback signal on the bus or measurement provided by VT system.

For stimulation and examination purposes the test modules/test units can access

- the complete remaining bus simulation
- the connected buses (e.g. CAN, LIN, FlexRay, IP)

- the digital and analog input and output lines of the Device Under Test using general purpose I/O cards or VT System via system variables
- other external real time systems (e.g. HIL systems and LabVIEW models) using the FDX interface
- other external measurement systems (e.g. GPIB and Ethernet) using appropriate interfaces

An ECU may be tested by itself, or as part of a network consisting of various ECUs where the object of testing is called a System Under Test or SUT. CANoe's options are available to the test modules/test units , e.g. panels for user interaction or writing of outputs to the Write Window.

III. Actual Project Example

Refer to following example github repos, it contain a CANoe simulation configuration (base on CANoe sample UDS config - to reuse the diagnostic database because without license you can not create new .cdd file and on the other hand I don't have access to ODX format standard specification either):

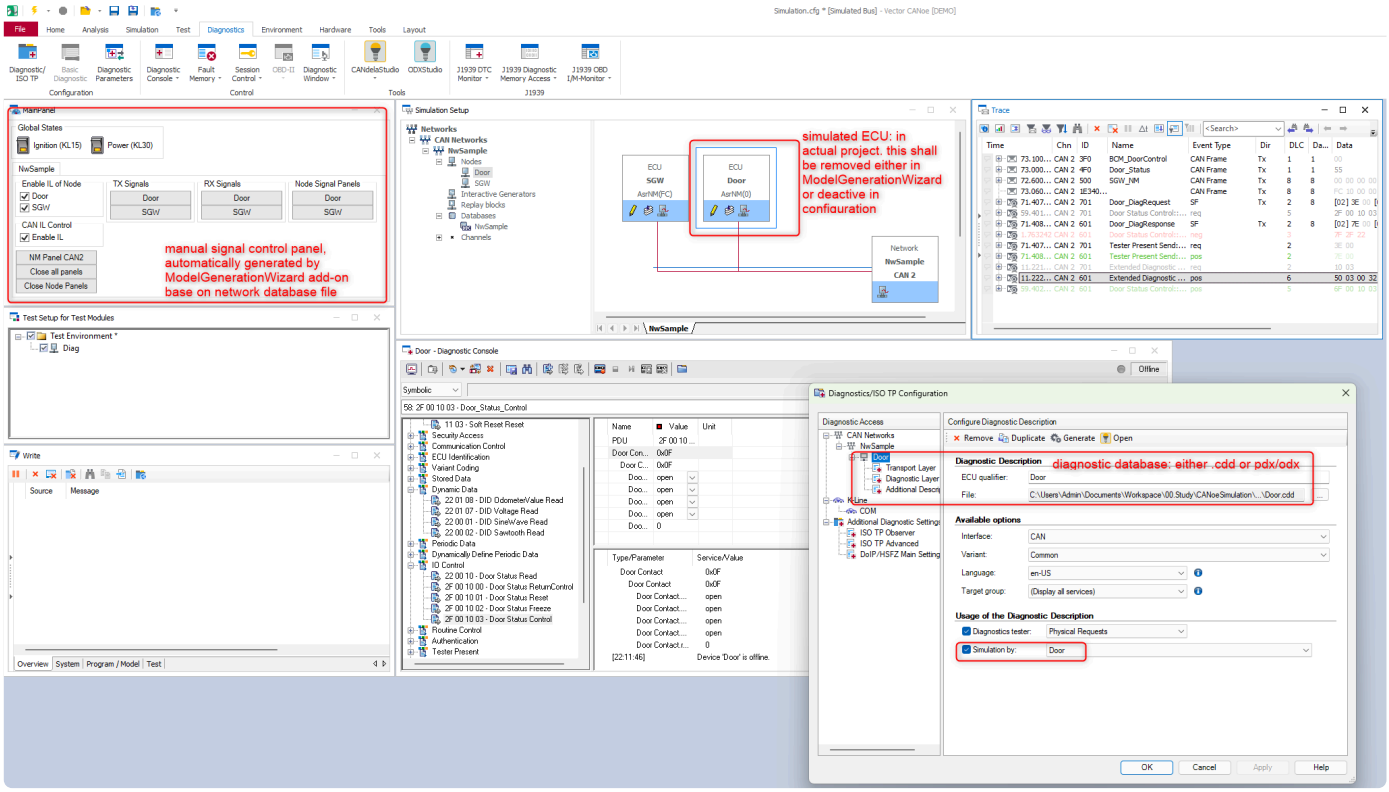
- DUT ECU - Door: simulated real ECU in a project
- SGateway: simulated peer node ECU, which is defined in network description dbc.
- Door.cdd: diagnostic description database file
- NwSample.dbc: CAN database for CAN network simulation.

Normally in a project, we should have a network dbc file, where relation between target developed ECU and other neighbor ECUs is defined.

- Target developed ECU shall be our DUT ECU.
- Other neighbor ECUs shall be simulated by CANoe(peer node) to provide stimulated inputs.
- Hardware configuration VN hardware device and bus characteristic.

Another input for testing environment is diagnostic description database file either in Vector defined .cdd or ODX standard format(.odx-d, .pdx).
⟹ These database file shall be imported to CANoe tool and help with test script implementation.

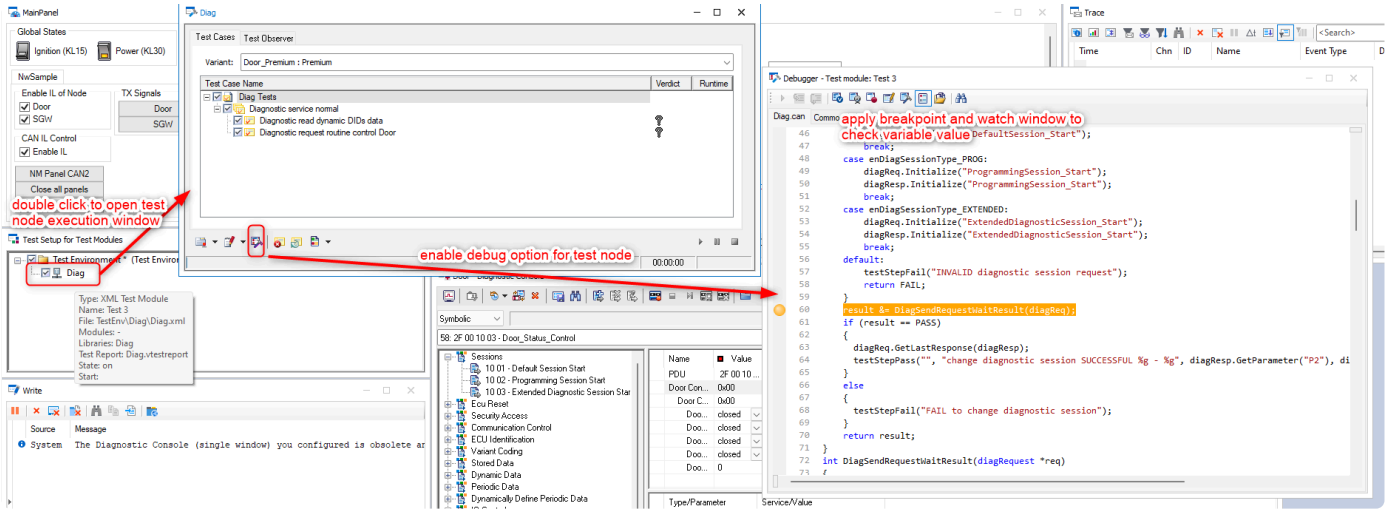
If your test setup have VT system, some additional configuration for it is needed as well.



Example test environment setup(CANoe18)

CAPL testcase implementation

During CAPL test script development, you can also debug CAPL as below.



CAPL debug

> In general, CAPL provide a lot of CANoe environment specific supported APIs(create stimulation, validation, and report) but for complex automation action, data processing such as: directory, file manipulate, XML/json parsing, cryptography operation,... Higher level programming language with rich libraries support such as C# .NET should be preferred.
> My suggestion for optimal setup of CANoe automation test environment is:
> - XML test node: easier for management/display/control during testing (regression test, manual/automation,...)
> - CAPL script: core implementation of testcases/testfunction to be called in XML test node, due to well supported/native Vector CANoe test APIs
> - C# .NET user library .DLL: can be easy implemented, built into .DLL and included to/called by main CAPL script, help expand automation functionalities. Ease the verification step in case of dealing with complex data and environment manipulation, which is very much limited with CAPL.

#testing #beginner

← Previous Next →

Related Posts

Diagnostic over IP(DoIP) Overview January 29th, 2026
Adaptive AUTOSAR Overview February 26th, 2024

ISO26262 - Fault Tolerance Time Interval(FTTI) October 4th, 2023
