Program Structure and Data Types

Parts marked with ***)** are **not** available **for CANalyzer.**

Including other Files textually (CAPL Include)

```
Includes
{
      #include "MySource.cin"
}
```

Global Variables, <type> = data type

```
variables
{
      <type>       myVar[=val];  // elementary
      <type>       myArray[10];  // array
      char         myString[20] [="val"];
}
```

<type>

▶ signed	int(16), long(32), int64(64)
▶ unsigned	byte(8), word(16), dword(32), qword(64)
▶ Floating point	float(64), double(64)
▶ character	char(8)
▶ Messages	message <CAN-id> name message <DBC-Name> name
▶ Other types	Associative Fields → Help Domain specific types → Help

Values in round brackets: Number of bits (not in the source code!).
All types are usable as local variables.
`elcount(myString)` returns the array size.

Please note:

- ▶ Local variables are implicitly static.
- ▶ No program statements prior to variable declarations

Comments

```
… // rest of line is comment
/* all inside is comment */
```

Ctrl-k-c: comment the selected area
Ctrl-k-u: uncomment

Own Types – Enums

Type definition in the variables section:

```
enum eMyEnumType {eMyEnum1 [=1],eMyEnum2 [=2]};
```

Declaration of an enum variable & initialization:

```
enum eMyEnumType gMyEnum = eMyEnum1;
```

Usage:

```
if (gMyEnum == eMyEnum2)
```

Own Types – Structs

Type definition in the variables section:

```
struct MyStructType
{
      int    anInt;
      long   aLong;
};
```

Declaration of a struct variable and initialization:

```
struct MyStructType gMyStruct [= {20,-1}];
```

Program Structure and Data Types - continued

Access to struct members:

```
gMyStruct.aLong = 22;
```

Constants

```
const <type> = <val>;
```

Events

CAPL is 100% event driven, event handler syntax:

```
on <event type> [additional parameter]
{
      // variable declaration
      // program code
}
```

<event type>

a) System Events

▶ key 'character' or *	keyboard events
▶ timer <tmr>	Timer expired
▶ preStart	pre measurement start
▶ start	after measurement start
▶ stopMeasurement	before measurement stop

b) Value Objects

▶ signal <sig>	Signal value changed *)
▶ signal_update <sig>	Signal write access *)
▶ sysvar <sys>	System variable value changed
▶ sysvar_update <sys>	System variable write access
▶ envvar	Value change environment variable

c) Messages

▶ message <msg>	CAN1.frmStatus
▶ message <id>-<id>	0x100-999
▶ message *	any message
message [*]	All messages, also those that are processed by another event handler in this CAPL source code.

This

Event data can be accessed via `this`.

on key *	this = character
on signal	this[.raw] = physical [raw] value *)
on message	this.<selector>
▶ TIME	measurement time
▶ ID	CAN-ID
▶ DLC	message length (data)
▶ DIR	RX, TX
▶ CAN	channel number
▶ BYTE(0…7), WORD(0..6), DWORD(0…4), QWORD(0) raw data	

▶ Signals	`on message <msg>`
	`{`
	`  this.sigTemp`

Raw data of the temperature signal is returned (compare to “on signal this“).
Physical values are accessed with .phys.

C-Syntax

Operators

a = 23 **+** b;	add and assign
* / -	multiplication, division, subtraction
r = 37 **%** 7;	modulo division for integer, here: 2
a**++**; b--;	increment or decrement an integer type
c**+**=22.0;	increment and assignment
-= *= /=	decrement, multiplication, division and assignment
a==b	comparison operator „equal“
< <= > >= ! =	other comparison operators
! (a<7) **&&** (b>20)	logical not(!), and (&&)
||	logical or
& | ~	bitwise and, or, complement
^	bitwise exclusive or (XOR)
a = 2**<<**3;	bit-shift left (afterwards a is 16)
>>	bit-shift right

Please note: Don’t mix up comparison (==) and assignment (=) operator!

If Decision

```
if (c <= 50)
{      // if c less or equal to 50
}
```

Optional branches:

```
else if ((c > 50) && (c <= 70))
{      // if c in ]50,70]
}
…
else
{ … }  // in all other cases
```

Switch Selection

```
switch (c)           // integer
{
      case 50:       // if c==50: …
      …
      break;         // leave this branch
      case 100:
      case 101:      // cntr==100 or cntr==101
      break;
      …
      default:       // in all other cases
      break;
}
```

For Loop

for (initialization; condition; modification):

```
int i;
for (i=0; i<10; i=i+1)
{      // statements
}
```

While Loop

```
while(c<50)
{      // statements
}
```

C-Syntax - continued

do while – loop
Loop is executed at least once

```
do
{       // statements
} while (c<50);
```

Please note:
Do not implement infinite loops. CANoe must be terminated via task manager.

Functions

```
[<return type>] MyFunction (<type> arg1, …)
{
       [return [constant | Variable];]
}
```

return type: int, long, int64, byte, word, dword, float, double
void: no return value

Calling a function

```
[var=]MyFunction([val][,val]…);
```

Signals and system variables as function parameters

```
void MyFunction(signal *sig, sysvar * sys, message * msg)
{
      $sig = 20.0;
      @sys = 20.0;
      $sig = DBLookup(sig).maximum; // see attributes
      output(msg);
}
```

DBlookup([msg|sig]) allows to access database attributes.

CAPL & CANoe

Signals and system variables should be accessed via symbol explorer or per auto-completion.

Signals

$<sig> = 1;	phys. signal value is set and with next message it will be sent by the interaction layer.
$<sig>.**raw** = 1;	assign signal raw data
v = **$**<sig>[**.raw**];	access last received Signal[raw] value. Physical values are of type double.
$<sig> =<msg>.<sig>::<text>	Access to text table definitions

System Variables
Assigning values:

```
@<sys> = 1;
```

`@sysvar::<sys>` is also possible
Reading values:

```
if (@<sys>==2.0)
```

Special system variables, arrays:

```
SysSetVariable<type>(<sys>,buffer[]);
SysGetVariable<type>(<sys>,buffer[],size);
```

<type>: String, IntArray, LongArray, FloatArray, Data
SysVar with conversion formula also as `.raw` for raw values.

CAPL & CANoe - continued

Timer

Declaration in the variables section:

```
msTimer myTmr;
```

Simple Timer:

```
setTimer(myTmr, <time>);
```

Start periodic timer, times in milliseconds:

```
setTimerCyclic(myTmr,<period>[,first period]);
```

Stopping

```
cancelTimer(myTmr);
```

Check if active:

```
isTimerActive(myTmr)
```

Event Handler

```
on timer myTmr
{ … }
```

Messages

Declaration of message variables:

```
message <*|id|msg> myMsg;
```

Modification:

```
myMsg.<selector> = <val>;
```

<selector> see section `this`.

Copy messages:

```
myMsg2 = myMsg1;
```

or in an event handler (e.g. a gateway)

```
on message CAN1.*
{
       message * myMsg2;
       myMsg2 = this;
       myMsg2.sigA = this.sigA * 2.0;
       myMsg2.CAN = 2;
       output(myMsg); // send
}
```

Attributes

Attribute names with (*) are OEM specific. Vector IL:

```
<msg>.<attribute>
```

- GenMsgSendType*
- GenMsgCycleTime*
- GenMsgILSupport*

```
<sig>.<attribute>
```

- GenSigStartValue*
- GenSigInactiveValue*
- GenSigSendType*
- factor // for conversion
- offset // for conversion
- unit // string
- maximum // physical
- minimum // physical

Useful CAPL Functions

Write Window

- Write window output void write(char format[], ...);

```
write("Sig phys=%f, Raw=%d", $sig, $sig.raw);
```

%... = Format expression

%d	decimal	**%4d**	4 digits
%f	float	**%6.2f**	6 digits in total, including 2 dec. places
%x	hex	**%08X**	8 digits, leading zeroes, capital letters
%s	string		for character arrays

- writeCreate create a new write window tab → Help
- writeClear clear a write tab → Help
- writeEx extended write → Help
- snprintf output to a string buffer → Help

Interaction Layer (control from within the node)

- Stop message sending
  `ILDisableMsg(<msg>)`
- Activate message sending
  `ILEnableMsg(<msg>)`
- Other:
  `IL…()` → Help

CAPL Tips

- Watch for compiler warnings, enable them all in CANoe options
- Use comments
- Use name prefixes, e.g.

c<Name>	constants
g<Name>	global variables
ut<Name>	utility functions
p<Name>	Function parameters

- Avoid complex calculations in conditions (readability, debugging)
  Instead: assign intermediate results to local variables
- Even if not needed for single statements after if: use curly brackets
- Event handler should not be a long runner
- Allow reuse with CAPL include files
- Return values should be assigned to local variables.
  Their values are visible when debugging

Notation on this Sheet

< >	mandatory
[ ]	optional or array index
val	value or string
sig	a signal name
sys	a system variable name
var	a CAPL variable
my…	user identifier
msg	message name
→ Help	see online help (F1)

Test Modules *)

Structure
MainTest is called upon test module start and determines the sequence of test cases. Test groups organize the test module structure and reporting.

```
void MainTest()
{
       … variables declaration
       MyTestcase1(…);
       testGroupBegin (<title>, <description>);
              MyTestcase1(…);
              …
              MyTestcaseN(…);
       testGroupEnd();
}
```

Functions marked with `testcase` are test cases. They do not have return values.

```
testcase MyTestcase1()
{ … }
```

Accordance with Simulation Nodes:
- All event handler
- C-syntax
- Data types
- System variables und signal access

Test verdict
Every test step, test case, test group and the test module has a test result Fail or Pass.

Test cases
Most test cases can be structured logically into three phases:
1. Stimulation: signals, IOs, messages …
2. Wait for a reaction
3. Evaluation

The phases are made of test steps that set the test verdict:

```
testcase myTestcase()
{       …
       testStep<Pass|Fail>("id", "description");
       …
       testStep ("id", "description"); // w/o verdict
}
```

id is a user definable text for identification.

Functions / Test Functions
Frequently used tests should be implemented as test functions or functions (for re-use). Both produce different reporting data.

Stimulation

```
@<sys> = <val>; // system variable
$<sig> = <val>; // with interaction layer
output(<msg>);  // local message object
```

Test Modules - continued

Waiting for Events → Help:

```
long testWaitFor<what>(…, <max time in millisec.>);
```

<what>
- Timeout — fixed time span
- SignalIn/OutSideRange — sys / sig value within / outside range
- SignalMatch — sys / sig value
- ValueInput — user input
- Message — message reception
- TesterConfirmation — user confirmation → Help
- MeasurementEnd — wait for measurement end

Return value: 0 = timeout; 1 = event

Evaluation
Using C statements:

```
if … else …
```

Better for signals and system variables:

```
testValidateSignalMatch("id",<Sys/Sig>,<val>);
testValidateSignalIn/OutRange -> Help
```

Reporting

```
testCaseTitle
testModuleTitle
teststep[Pass|Fail]("id", "text …%d text", <var>);
testreportAdd<what>(…)
```

<what>
- EngineerInfo
- ExternalRef
- Image
- WindowCapture
- … → Help

Fault Insertion
From the test module for all simulation nodes
- TestDisableMessage<(msg>)
- TestEnableMessage(<msg>)
- TestEnableCRCCalculation(<msg>
- TestResetAllFaultInjections()
- … → Help

Background Checks
Configuration: (identification is done with a dword handle)

```
<dword> = ChkCreate_<checktype> (<parameter>)
```

<Check type>
- MsgAbsCycleTimeViolation
- ErrorFramesOccured
- … → Help

Control:

```
chkControl_<Start|Stop|Reset|Destroy(<dword>)
```

Evaluation:

```
chkQuery_<what>(<dword>)
```

<what>
- NumEvents
- EventStatus
- … → Help

Test Modules - continued

Check as constraint (=observation of the test environment)
or as condition (=observation of the ECU behavior). Both write events to the report and impact the verdict:

```
testAdd<Condition|Constraint>(dword[,1])
```

With the optional second parameter being 1 the verdict is not influenced.

Stimulations
Create signal value sequences. Creation with

```
<dword> = stmCreate_<what>(<signal>,<parameter>)
```

<what>
- CSV — comma separated values
- Ramp
- Toggle
- … → Help

Control:

```
stmControl_Start|Stop(<dword>)
```

Diagnostics
Diagnostic objects must be declared. The service qualifier should be inserted via Drag & Drop from the Symbol Explorer:

```
DiagRequest <service qualifier> myReq;
DiagResponse <service qualifier> myResp;
```

Selecting the diagnostic target:

```
DiagSetTarget("<ECU qualifier>");
```

Set a request parameter, the parameter qualifier taken from the Symbol Explorer:

```
diagSetParameter(myReq, "<Parameter>", val);
```

Send:

```
DiagSendRequest(myReq);
```

Wait for a response:

```
int = testWaitForDiagResponse(myName,10000);
```

Evaluation, the response can be addressed with the request (request and response belong together):

```
double = DiagGetRespParameter(myReq, "Para");
```

Alternative:

```
DiagGetLastResponse(myReq, myResp);
```

Other functions:
- DiagGetLastResponseCode(myReq) -1 upon positive response, else negative response code
- DiagGetComplexParameter(…) dynamic data → Help
- Diag… → Help

Test Module Tips

- Implement functions for re-use in test cases (readability, maintenance, effort).
- Check return values and document problems with TestStepFail().