MICROSAR OS
CANoe VTT Library

Manual

Version 5.0

Document ID: OS03.00140.4

Vector Informatik GmbH, Ingersheimer Str. 24, D-70499 Stuttgart
Tel. +49 711 80670-0, Fax +49 711 80670 555,
Email: info@vector.com, Internet: http://www.vector.com

80087

Subsidiaries

France

Vector France SAS

168, Boulevard Camélinat
F-92240 Malakoff

Tel.: +33 1 4231 4000
Fax: +33 1 4231 4009

information@fr.vector.com
http://www.vector-france.com

Japan

Vector Japan Co., Ltd.

Seafort Square Center Bld. 18F
2-3-12, Higashi-shinagawa, Shinagawa-ku
J-140-0002 Tokyo

Tel.: +81 3 5769 6970
Fax: +81 3 5769 6975

info@jp.vector.com
http://www.vector-japan.co.jp

Great Britain

Vector GB Ltd.

Rhodium
Central Boulevard
Blythe Valley Park
Solihull, Birminham
West Midlands B90 8AS
United Kindom

info@uk.vector.com
http://www.vector.com/vu_index_en.html

Korea

Vector Korea IT Inc.

#1406 Mario Tower
222-12 Guro-dong, Guro-gu
Seoul 152-848
Republic of Korea

info@kr.vector.com
http://www.vector.com/vk_index_ko.html

India

Vector Informatik India Pvt. Ltd

Lokesh Madan
4/1/1/1, Sutar Icon, Sus Road
Pashan, Pune – 411 021
India

info@in.vector.com
http://www.vector.com/vh_index_hi.html

Sweden

VecScan AB

Theres Svenssons Gata 9
SE-41755 Göteborg

Tel.: +46 31 76476 00
Fax: +46 31 76476 19

info@se.vector.com
http://www.vecscan.com/

USA

Vector CANtech, Inc.

Suite 550
39500 Orchard Hill Place
USA-Novi, Mi 48375

Tel.: +1 248 449 9290
Fax: +1 248 449 9704

info@us.vector.com
http://www.vector-cantech.com

For distributor addresses please have a look on our website:

http://www.vector.com

Certified Quality Management System

The Quality Management of Vector Informatik GmbH has been certified throughout since 1998-08-19:

- 2010-07-19 (latest certification until the elaboration of this document)
  according to DIN EN ISO 9001:2008
  Certificate number: 12 100 23612 TMS
- 1998-08-19 (first certification)
  according to DIN EN ISO 9001:1994-08
  Certificate number: 70 100 F 1498 TMS

Typographical conventions

Note:	Identifies very important notes
•	Identifies enumerations
[OK]	Notation for buttons in dialog boxes
<TAB>	Notation for keys on the computer keyboard
<Strg>+<Z>	Notation for keys on the computer keyboard that are to be pressed simultaneously
Add… **File│File open…**	Notation for menu, command and dialog names
`on message 0x100`	Notation for CAPL syntax and MS-Dos syntax

Contents

1 Overview

This documentation describes the MICROSAR OS CANoe VTT Library and the implementation-specific part of the OSEK[1]/Autosar operating system emulation in the Vector CANoe simulation environment.

The common part of all MICROSAR OS implementations is described in the separate document /MICROSAR OS/.

The implementation is based on the OSEK OS specification 2.2.1 described in the document /OSEK OS/ and the Autosar OS specification 5.0.0 described in /Autosar OS/. This documentation assumes that the reader is familiar with the OSEK and Autosar specifications.

“OSEK/VDX” is a registered trademark of Continental Automotive GmbH (until 2007: Siemens AG).

[1] (Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug, Open Systems and their interfaces for Electronics in motor vehicles)

2 Related Documents[2]

Abbreviation	File Name and Description
/Autosar OS/	Autosar OS Specification 5.0.0[3]
/OSEK OS/	OSEK/VDX Operating System Specification 2.2.1[4]
/MICROSAR OS_HW/	File: MicrosarOS_CANoeVTT.pdf User manual of MICROSAR OS CANoe Library and the CANoe-specific part of Vector *MICROSAR OS* (This document)
/MICROSAR OS/	File: TechnicalReference_Microsar_Os.pdf User manual of Vector *MICROSAR OS*, general part
	File: Tutorial_osCAN.pdf Tutorial of Vector *MICROSAR OS*

Table 1: Documents

[2] Depending on the delivered package some documents may not be included

[3] This document is available in PDF-format on the internet a the Autosar homepage (http://www.autosar.org)

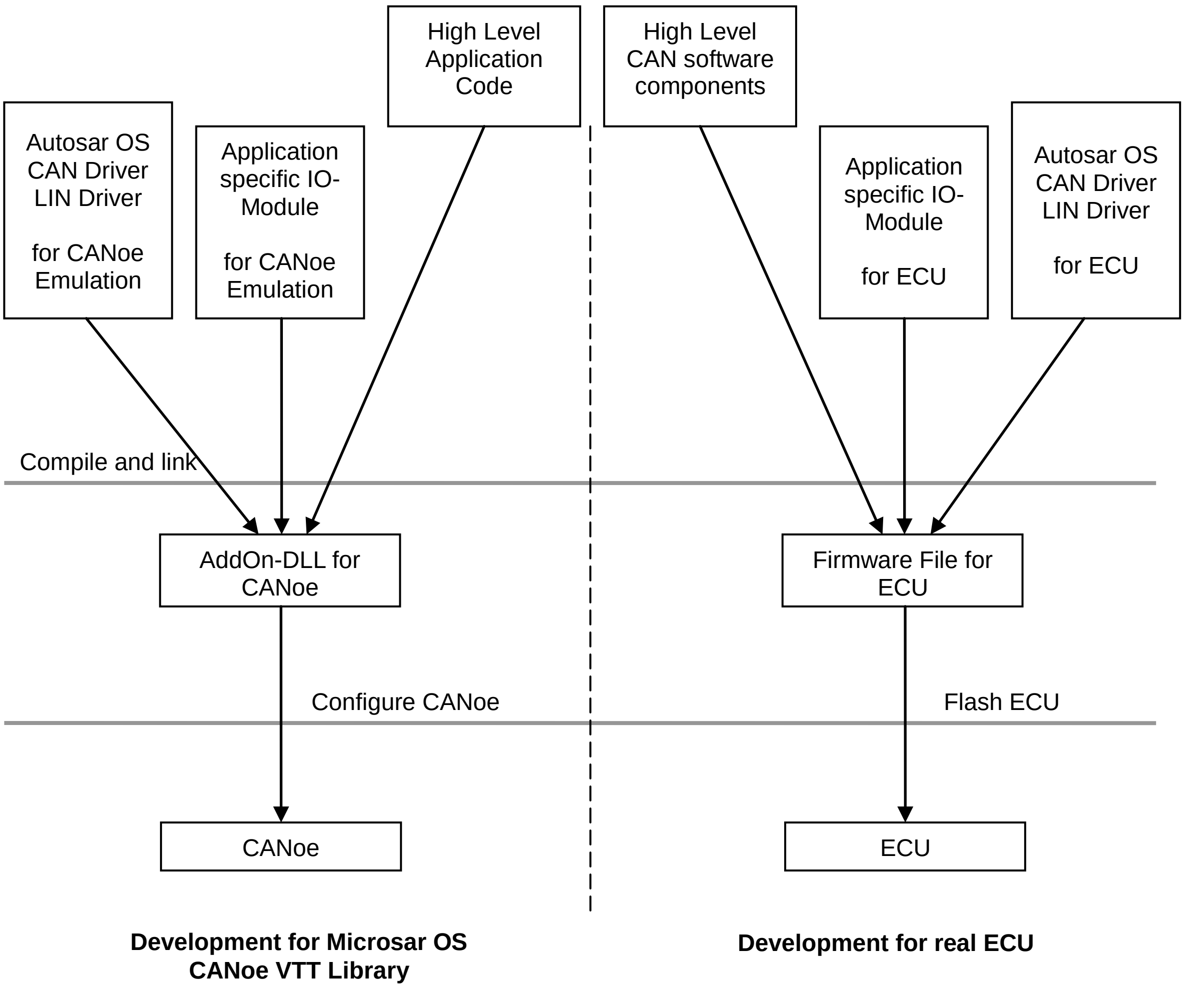
[4] This document is available in PDF-format on the internet a the Autosar homepage (http://www.osek-vdx.org)

3 Introduction

CANoe and the MICROSAR OS Library help to realize the software development for different ECUs to an extended development level before the final hardware is available.

The library allows to run OSEK/Autosar applications simultaneously from all CAN or LIN networked ECUs in real time on a PC. The OSEK/Autosar OS application which will be developed by the user is translated for the PC in a commercial C compiler and bound to the CANoe VTT MICROSAR OS Library. Therefore the user will also be able to use the extensive possibilities of CANoe's CAN and LIN bus simulation for virtual OSEK/Autosar nodes.

The figure below compares the two processes, the one for developing a real ECU and the other for developing an emulation within CANoe.



Developing an application for a real ECU, various C source code modules are taken, compiled and linked together to a firmware file. The firmware file is then flashed to the ECU or loaded into an emulator. The modules needed are: The operating system, drivers for a CAN or LIN bus controller and other application-specific devices, perhaps one or more high-level software components like NM (Network Management) and TP (Transport Protocol) and the high-level routines of the application.

Using the CANoe emulation, the hardware-specific modules have to be replaced. These are the operating system and device drivers. The operating system is part of the MICROSAR OS CANoe VTT Library. The CAN and LIN drivers for CANoe emulation modules have the same API as all Vector CAN and LIN drivers for microcontrollers. That is why they can be simply replaced by each other.

For the I/O simulation apart from the CAN and LIN bus, the user must change the target specific I/O access. Normally they are replaced by access to environment variables of CANoe. Environment variables allow an easy usage of CANoe Panels.

After replacing all hardware-specific modules, the hardware-independent modules are compiled and linked together with the operating system library. The result will be an AddOn-DLL for CANoe. At last the DLL has to be added to the CANoe configuration. Now the simulation within CANoe can be executed.

4 CANoe MICROSAR OS Library-specific part of Vector *MICROSAR OS*

4.1 Overview of OSEK properties

Autosar OS specification:	5.0.0
OSEK OS specification:	2.2.1
Scalability Classes supported:	SC1 only
Conformance class:	ECC2, ECC1, BCC2, BCC1
Scheduling policy (source-code version):	mixed-preemptive
Scheduling points:	depending on scheduling policy
Maximum number of tasks:	65535
Maximum number of events per task:	32
Maximum number of activations per task:	255
Maximum number of priorities:	8192
Maximum number of counters:	256
Maximum number of alarms:	32767
Maximum number of resources:	8192
Maximum number of schedule tables:	65535
Maximum number of expiry points:	65535 (cyclical expiry points counted by their multiplicity)
Status levels:	STANDARD and EXTENDED
Nested Interrupts:	possible
Interrupt level resource handling:	available
ORTI:	not available
Library Version:	only Library Version supported
Stack optimization option:	not available
Supported C compiler/linker:	Microsoft Visual C++ 2010

4.2 Installation

The installation is described in the common document /MICROSAR OS/. The CANoe-specific files are described below.

4.2.1 OIL configurator

The OIL configurator is a general tool for different OSEK\MICROSAR OS implementations. The implementation-specific parts are the code generator and the OIL implementation files for the code generator (installation paths are described in /MICROSAR OS/.

4.2.2 OIL implementation files

The implementation-specific files will be copied onto the local hard disk. The OIL tool has knowledge about these files through the INI-file OILGEN.INI (the correct path is set by the installation program).

CPU	Implementation file	Standard object file	Description
CANoe	CANoeOsekEmu.i40	CANoeOsekEmu.s40	Library Version

4.3 OIL Attributes

This chapter describes all platform specific configuration attributes for CANoe VTT implementation of MicrosarOS. All general attributes which apply to all Microsar OS are described in the general documentation.

4.3.1 System Timer Attributes

The platform specific attributes of the SystemTimer object (if configured as *Standard*) are:

OIL Attribute name	XML Attribute name	Description
TickTime	OsOSTickTime	Specifies the tick time of the System Timer in microseconds.
EnableNesting	OsOSEnableNesting	If set to FALSE, MicrosarOS disables interrupts at the start of the SystemTimer ISR, so no nesting will occur (interrupts are not disabled ISR entry).
InterruptPriority	OsOSInterruptPriority	Specifies the System Timer interrupt level

4.4 ISR attributes

The platform specific attributes of an ISR object are:

OIL Attribute name	XML Attribute name	Description
InterruptSource	OsIsrInterruptSource	Specifies the interrupt source for this ISR.
InterruptLevel	OsIsrInterruptLevel	Specifies the interrupt priority. Higher

OIL Attribute name	XML Attribute name	Description
		values means higher priorities.

Platform specific attributes of an ISR object

4.5 Timer and Alarms

4.5.1 Range of Alarms

Depending on the OS attribute *TickTime* the range of the alarms is different:

Max_Range = TickTime · 32767

4.5.2 Selection of the Tick Time

When the TickTime is shorter, the interrupt load is higher. Therefore, the TickTime should be chosen as big as possible (greatest unique divider of all alarm times). In addition, alarm management can be delayed if the TickTime is too short.

Caution
It is not allowed to disable interrupts longer than the TickTime duration in the user application. If interrupts are disabled longer than the TickTime the alarm management could be handled wrong. This error is not detected by the operating system.

4.5.3 Timer Hardware Usage

Schedule tables and alarms are driven by counters. Counters themselves are driven by a hardware timer. The CANoe VTT OS only supports a single software timer which can be used by the OS for tick time generation.

The selection of the used timer is made with the OIL/XML attribute “SystemTimer”/ “OsOSSystemTimer” (see 4.3.1). The following table shows the possible choices for this attribute:

SystemTimer	Description
Standard	The CANoe VTT system timer (interrupt source number 0) is used for OS tick time generation.

Timer hardware usage

4.6 Stack handling

The MICROSAR OS CANoe VTT Library uses the stack handling of Windows threads. The user does not need to specify the stack size a correct stack is assigned and maintained by Windows.

4.7 Interrupt handling

The MICROSAR OS CANoe VTT Library does not pass real interrupts of the PC to the OSEK application. Instead, it is possible to trigger an ISR from within the application by using the following function:
`CANoeEmuProcessor_RequestInterrupt(sint32 irqNumber);`
The parameter `irqNumber` specifies the ID of the interrupt source. All valid source IDs can be found in the header file tcb.h. For instance, `CANoeEmuProcessor_RequestInterrupt(IRQ_CanIsrTx_0)` triggers the interrupt source CanIsrTx_0.

Caution
When entering/exiting an ISR, the interrupt flag is not modified.

Attention

Interrupt processing in CANoe VTT differs from interrupt processing on a microcontroller. Interrupts in CANoe VTT do not occur while any task is busy. The following code may not work:

```
volatile osuint8 trigger;

TASK(Task0)
{
      while(1)
      {
            trigger = 0;
            while(!trigger)      // wait for ISR0
            {
            }

            // do something ...
      }

      TerminateTask();
}

void ISR0()
{
      trigger = 1;
}
```

The reason for this is that CANoe cannot interrupt a running task. In the above case, CANoe freezes and must be killed. This problem can be avoided by manually giving the control back to CANoe. This can be done, for instance, by calling function `CANoeAPI_ConsumeTicks(1)` inside of the inner while loop. This function simply switches to the CANoe environment in order to consume a single time unit. Before switching back to the task, CANoe may check if there is some interrupt pending and, if required, starts execution of the corresponding interrupt service routine.

4.7.1 Category 1 ISRs

A category 1 ISR is not allowed to use OSEK API calls. This type of ISR is completely transparent to the OS. A category 1 ISR is implemented as a void-void function.

Implementation of a category 1 ISR

```
void MyIsr(void)        // category 1 ISR
{
   /* ISR code */
}
```

4.7.2 Category 2 ISRs

A category 2 ISR is allowed to use OSEK API calls (which may lead to a task switch). Category 2 ISRs are handled by the OS. It is implemented by using the OSEK ISR macro.

Implementation of a category 2 ISR

```
ISR(MyIsr)              // category 2 ISR
{
   /* ISR code */
}
```

4.7.3 Nested ISRs

The OS supports interrupt nesting. Each higher priority interrupt may interrupt any lower priority interrupt at any point of execution. For category 2 interrupts, nesting can be disabled by setting the OIL/XML attribute `EnableNesing /OsIsrEnableNesting` to FALSE.

Note
Even with `EnableNesing /OsIsrEnableNesting` set to FALSE, it is possible that nesting occurs during ISR entry/exit code. Only user code cannot be interrupted.

For category 1 interrupts, this has to be done explicitly by disabling the global interrupt flag.

```
void MyIsr(void)     // category 1 ISR
{
  CANoeEmuProcessor_DisableInterrupts();
    …
  CANoeEmuProcessor_EnableInterrupts();
}
```

Caution
If interrupts are disabled during a user ISR they have to be enabled again at the end of the user ISR function.

Caution
All category 1 ISRs must always have a higher priority than the highest priority of any category 2 ISR. It is the user's responsibility to ensure this.

4.7.4 Unhandled Exceptions

All ISRs which are implemented in the application have to be defined in the OIL/XML file. An interrupt source has to be assigned to the ISR. If interrupt / exception sources are left unassigned (no ISR defined for this exception number) then the OS generates a branch to the OS internal function "osUnhandledException".

When any unassigned interrupt source without associated ISR signals an interrupt the OS issues an unhandled exception error and goes into shutdown.

4.7.5 Enumerated Unhandled ISRs

During development of an application the source which has triggered an unhandled exception must be identified. This is sometimes complex or not possible.

Therefore the OS offers the feature "EnumeratedUnhandledISRs". If this attribute in the OIL is set to TRUE, additional code is generated which is capable to detect the triggering interrupt source in case of an unhandled exception.

The exception number of the ISR source which has triggered the interrupt can be read from the variable "osISRUnhandledException_Number".

Caution
The variable "osISRUnhandledException_Number" shall only be read inside the ErrorHook if the current error number is 0x2801 (osdErrUEUnhandledException).

4.7.6 Timer and Bus interrupts

Timer and bus (CAN, Flexray or LIN) interrupt service routines will be called by the OSEK emulation any time this is necessary. The emulation does not call these functions while processing tasks (see example in 4.7).

5 Implementation Specific Behavior

5.1 API Functions

5.1.1 DisableAllInterrupts

The function DisableAllInterrups disables all interrupts by calling CANoeEmuProcessor_DisableInterrupts(). The previous value of the interrupt-disable flag is saved.

Caution
Nested calls of DisableAllInterrupts are not allowed.

5.1.2 EnableAllInterrupts

The function EnableAllInterrupts restores the interrupt-disable flag which has previously been saved by DisableAllInterrupts.

Caution
Nested calls of EnableAllInterrupts are not allowed.

5.1.3 SuspendAllInterrupts

The function SuspendAllInterrups disables all interrupts by calling CANoeEmuProcessor_DisableInterrupts(). The previous value of the interrupt-disable flag is saved.

Info
Nested calls of SuspendAllInterrupts are allowed.

5.1.4 ResumeAllInterrupts

The function ResumeAllInterrupts restores the interrupt-disable flag which has previously been saved by SuspendAllInterrupts.

Info
Nested calls of ResumeAllInterrupts are allowed.

5.1.5 SuspendOsInterrupts

The function SuspendOsInterrups disables all interrupts by calling CANoeEmuProcessor_DisableInterrupts(). The previous value of the interrupt-disable flag is saved.

Info
Nested calls of SuspendOsInterrupts are allowed.

5.1.6 ResumeOsInterrupts

The function ResumeOsInterrupts restores the interrupt-disable flag which has previously been saved by SuspendOsInterrupts.

Info
Nested calls of ResumeOsInterrupts are allowed.

5.1.1 GetResource

If GetResource is called for an interrupt resource the OS disables the interrupts. The old interrupt state is stored.

5.1.2 ReleaseResource

ReleaseResource is the counterpart of the GetResource. It restores the interrupt state which was previously saved by GetResource (in case of an interrupt resource).

5.2 Hook Routines

The OSEK OS specification allows additional implementation specific parameters in hook routines. The context for called hook routines is implementation specific and described below. All Hook routines are called with disabled interrupts

5.2.1 ErrorHook

Is called	If an error occurs (API error, Syscheck or Assertion)
Call context	Can be called from task context, interrupt context or OS context

5.2.1 StartupHook

Is called	Inside the function StartOS()
Call context	Startup context

5.2.1 ShutdownHook

Is called	Before the system goes into an endless loop if a shutdown has occurred.
Call context	Can be called from Task context, interrupt context or OS context

5.2.2 PretaskHook

Is called	Before a new task is started or if an old task is resumed.
Call context	OS context

5.2.3 PostTaskHook

Is called	Each time a task leaves the running state.
Call context	Task / OS context

6 Setup of the Tools

6.1 Setting up the OIL configurator (OilCfg)

The OIL configurator has to be configured accordingly to be used together with the MICROSAR OS CANoe VTT Library, which means that an implementation file and a generator for the MICROSAR OS CANoe Library is required.

For the implementation file the following entry in the menu "Edit\Implementations..." of the OIL tool is needed:

- **Name**: `VectorCANoeOSEKEmulationLibrary`
- **File name**: `$(OIL_implementation_files_path\ CANoeOsekEmu.i40`

The generator is configured in the menu "Generators\Generator…" of the Oil tool. The MICROSAR OS CANoe Library requires the following settings:

- **Name**: `genCANoe`
- **File name**: `$(OIL_configurator_path)\genCANoe.exe`
- **Target path**: `tcb`
- **Command line**: `-r %e% -x -d %g% -g %f%`

Both of these settings (Implementation VectorCANoeOSEKEmulationLibrary and Generator genCANoe) are needed for the configuration and generation of an application for the MICROSAR OS CANoe Library.

The installation will set up all needed information automatically.

7 How to create a project

This chapter describes how to create a new application for the MICROSAR OS CANoe VTT Library.

7.1 Creation of a directory structure

First of all define an expressive identifier for the ECU to be simulated (Don't use more than 32 characters as well as no blanks or other special signs. Example: ECU for an anti-blocking system, identifier 'ABS'). This identifier has to be entered into the database (CANdb) later on.

The project directory has to be created next. If only one single ECU shall be developed with the MICROSAR OS CANoe VTT Library, it can be put as a new subdirectory to the example application. The name of the subdirectory will give a better overview of the identifiers of the ECU (Example: `\CANoeEmu\appl\ABS )`.

If several ECU's should be emulated which are connected in a compound system (e.g. all ECU's of one vehicle), then create a directory for the compound system parallel to the directory of the example application and then create a subdirectory for each ECU there. In addition you should create a subdirectory `exec32` in the compound directory in which the DLLs are saved later on.

Examples:

- `\CANoeEmu\CarXY` (compound directory)
- `\CANoeEmu\CarXY\exec32` (emulation-DLLs)
- `\CANoeEmu\CarXY\ABS` (ECU)
- `\CANoeEmu\CarXY\ASR` (ECU)
- `\CANoeEmu\CarXY\ESP` (ECU)

Note:	The project directory should be placed in the directory of the example application or in a parallel directory (see above mentioned example), so that each source and include file can be addressed the same way as in the makefiles. Using this mechanism of relative addressing the whole project structure can be copied from one computer to another without adjusting the path in the makefiles or without determining a certain installation directory.

The following subdirectories shall be created in the project directory:

- **tcb**
  The source and header files which are generated by the oil configurator will be stored here.
- **CANdb**
  The database files (CANdb)
- **CANoe**
  The CANoe configurations are stored here

The file NodeConfig.h has to be copied from the template file `$(Applications_path)\template` into the project directory and its entries have to be adjusted to the application requirements. Furthermore the application-specific source files have to be added to the project directory.

7.2 Setting up a Visual Studio Project

If all required directories and files are available, the project file or makefile for the compiler can be created.

Using Microsoft Visual C++ 2010 create a new *Win32 Project*. In the application Wizard select *DLL* as Application type and as Additional options select *Empty Project.* After clicking on *finish* you project will be created.

- Create a *main.c* source file (see template in Listing 1).
- Add all C source files (*.c) of the generated *tcb* folder to your project.
- Add the *NodeConfig.cpp* source file (*src* directory) to your project.
- Add the NodeConfig.def resource file (src directory) to your project
- Open the *Project Property Page* dialog box (Alt+F7) and expand the *Configuration Properties* tree on the left.
  - In *VC++ Directories/Include Directories* add:
    1. the path to the *include* directory of the OS
    2. the path to the *tcb* directory containing all generated files
  - In *VC++ Directories/Include Directories* add the path to the *lib* directory of the OS.
  - In *C/C++/Code Generation/Runtime Library* select *Multi-threaded Debug (/MTd)*
  - In *Linker/Input/Module Definition File* insert the path to the file *src\NodeConfig.def*

Now, you should be able to build your project (F7).

Listing 1

```
#include <CANoeEmuProcessor.h>
#include <CANoeApi.h>
#include <os.h>
#include <osekext.h>
#include <stdio.h>

void main(void)
{
       StartOS(OSDEFAULTAPPMODE);
}

void CANoeAPI_InitHook(void)
{

       CANoeAPI_SetNameOfEcuInstance("CANModule");
       CANoeAPI_SetVersionOfEcuInstance("0.1");
       CANoeAPI_SetNameOfWriteTab("CANoeEmu CANModule");
       CANoeAPI_SetRootNameSpaceOfSystemVariables("VTT::CANModule");
       CANoeAPI_SetMainFunction(&main);
       CANoeAPI_ConfigureInterruptController(oskNumberOfInterruptSources,32);
}
```

7.3 Configuration of CANoe

Consider the Simulation Setup in Figure 1 which consists of a network node and an ECU node. In order to run your application just right-click on the ECU node and select *Configuration.* Go to the tab *Components* and click on the *Add* button. Set the path to your application .dll file and press OK. Now, you can launch the simulation.

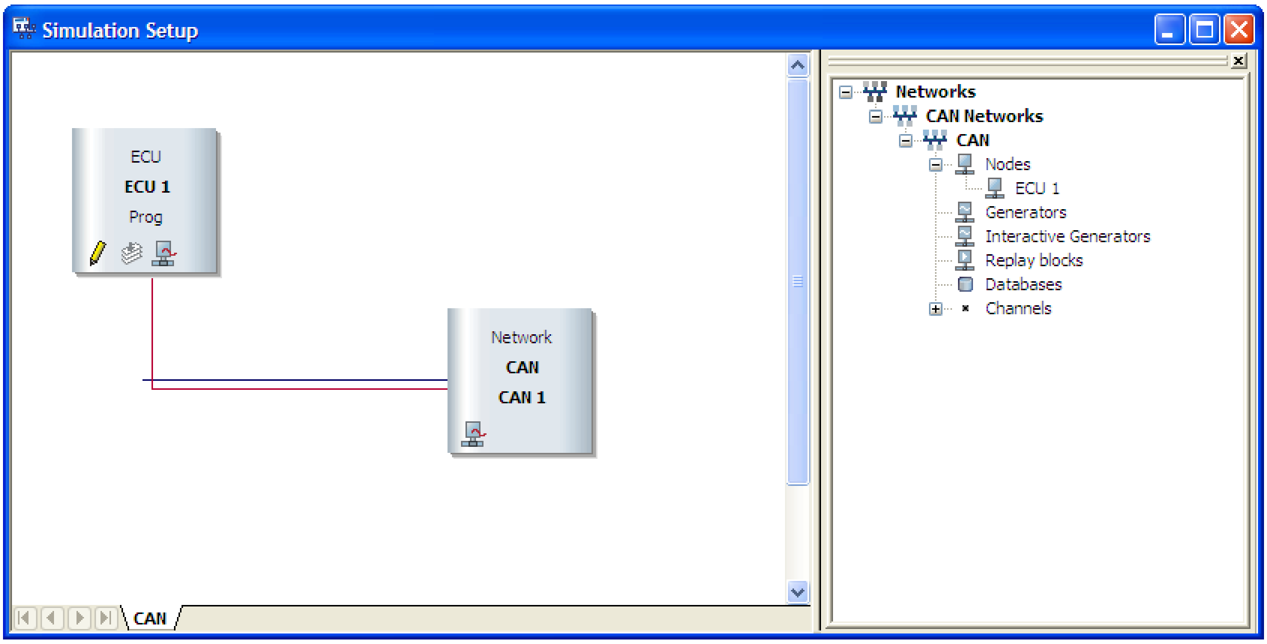


Figure 1

8 Specific Feature of the OSEK/Autosar OS Emulation

8.1 Technical Background

The main entry point of your applicatoin is the function `CANoeAPI_InitHook()`(see Listing 1) that is called once after the .dll file was loaded into CANoe. At this point you have to specify your *main* function that is called when starting the simulation. This is done with the function `CANoeAPI_SetMainFunction()`.

The header file *CANoeAPI.h* declares some functions, which provide access to some features of CANoe. For example, it is possible to access CANoe environment variables, put text to the “Write Window” of CANoe and get the actual time of the simulation. For more information read the comments in this header file.

Interrupts don’t occur at any time, they occur only during the execution of the IDLE loop or between task switches. The emulation is not a non-preemptive system, and also not a completely preemptive one. Normally, this behavior is sufficient for the simulations.

Do avoid endless loops! Due to the fact that the system is not preemptive, endless loops normally lead to a standstill of CANoe (See also chapter 4.7).

The emulation-DLL is a CANoe NodeLayer-DLL and therefore it has to export the following functions. CANoe cannot load the DLL if one of these functions is missing. The linker definition file ‘NodeConfig.def’ provides the suitable exportation of the functions.

- VIARequiredVersion
- VIASetService
- VIAGetModuleApi
- VIAReleaseModuleApi

8.2 CAPL Functions

Some CAPL functions are available for the control of the emulation. They are described as follows:

8.2.1 OSEK_Reset

```
void OSEK_Reset();
```

The function ‘OSEK_Reset‘ causes a hard reset of the emulation. A running emulation is stopped without calling of ‘ShutdownHook’. After that, the function ‘CANoeAPI_Main‘ is called.

Example:

```
// restart of emulation on button ‘r’ (‘r’ like re-set)
on key ‘r’
{
   write(“Reset”);
```

```
    OSEK_Reset(0)
}
```

8.3 Access to CANoe System Variables

The header file *CANoeAPI.h* from the MICROSAR OS CANoe VTT Library contains function declarations for the access to CANoe system variables. For a description of CANoe system variables see the online help of CANoe.

You can declare system variables with the functions

- `CANoeAPI_SysVar_DeclareInt`,
- `CANoeAPI_SysVar_DeclareFloat`

You can put values to system variables with the functions

- `CANoeAPI_SysVar_SetInt`,
- `CANoeAPI_SysVar_SetInt`

You retrieve the value from a system variable by using one of the functions

- `CANoeAPI_SysVar_GetInt`,
- `CANoeAPI_SysVar_GetFloat`

You can also register callback functions for system variables, which then are called every time the content of the environment variable has changed.

- `CANoeAPI_SysVar_SetHandlerInt`,
- `CANoeAPI_SysVar_SetHandlerFloat`

8.4 State Model of the Emulation

The emulation process can be distringuished into several states as depicted in Figure 2. The application may register a handler that is called when the global state of the emulation changes.

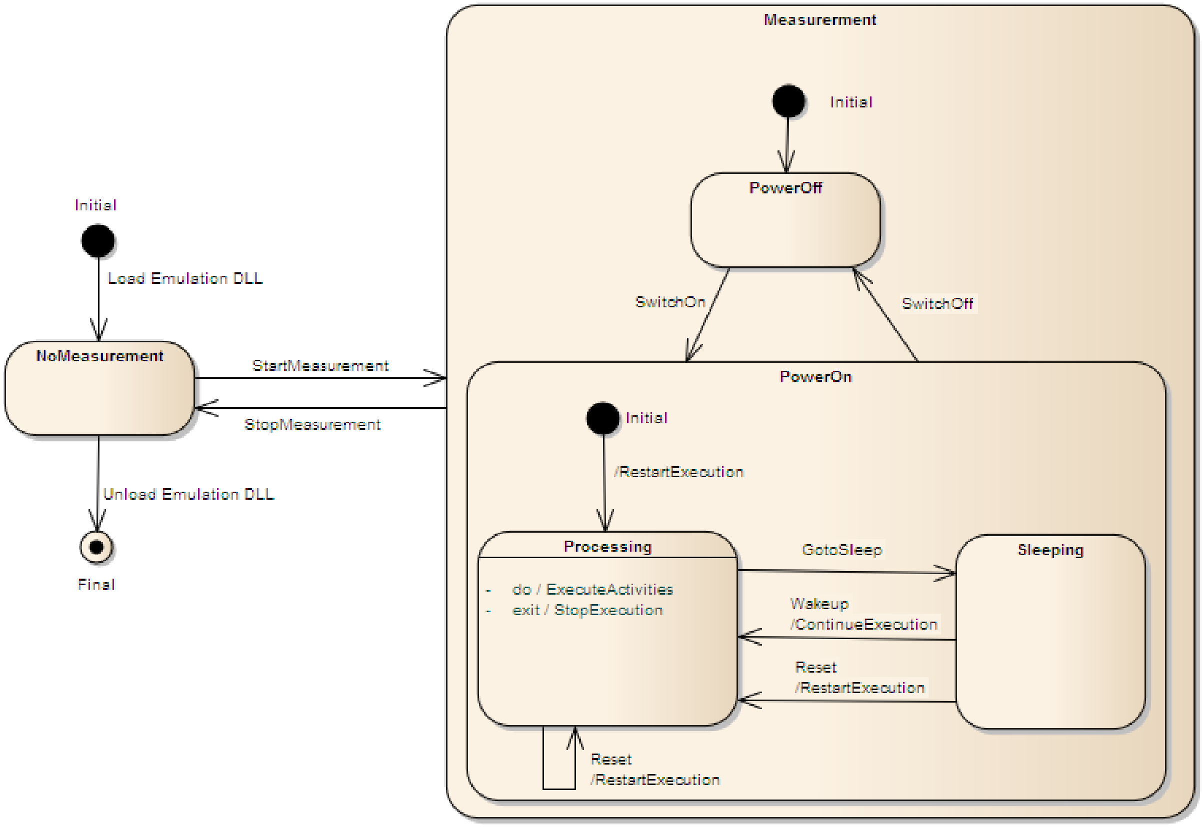


Figure 2

8.4.1 Using the State Change Handler

The following function has to be called inside of the function CANoeAPI_InitHook() in order to register an emulation state change handler:

```
void CANoeAPI_SetEcuStateHandler( VEcuStateHandler handler );
```

Type definition of `VecuStateHandler`:

```
typedef void (*VEcuStateHandler)( uint8 action, uint8 oldState,
uint8 newState);
```

Possible actions according to the state machine in Figure 2 are:

- `CANOEAPI_ECUSTATE_INITIAL`
- `CANOEAPI_ECUSTATE_NOMEASUREMENT`
- `CANOEAPI_ECUSTATE_POWEROFF`
- `CANOEAPI_ECUSTATE_PROCESSING`
- `CANOEAPI_ECUSTATE_SLEEPING`
- `CANOEAPI_ECUSTATE_FINAL`

Possible states according to the state machine in Figure 2 are:

- `CANOEAPI_ECUACTION_NOACTION`

- CANOEAPI_ECUACTION_LOAD
- CANOEAPI_ECUACTION_UNLOAD
- CANOEAPI_ECUACTION_INITMEASUREMENT
- CANOEAPI_ECUACTION_STARTMEASUREMENT
- CANOEAPI_ECUACTION_STOPMEASUREMENT
- CANOEAPI_ECUACTION_SWITCHON
- CANOEAPI_ECUACTION_SWITCHOFF
- CANOEAPI_ECUACTION_GOTOSLEEP
- CANOEAPI_ECUACTION_WAKEUP
- CANOEAPI_ECUACTION_RESET

9 FAQ (Frequently Asked Questions)

9.1 The CAPL Function OSEK_MapChannel is missing

The CAPL function OSEK_MapChannel was previously used to create a mapping between a CAN channel of CANoe and a driver index of the simulated CAN driver, which is used in the emulation. With MICROSAR OS CANoe Library 3.01, this mapping is now defined by the configuration of the generation tool for the driver. There you can specify the name of the bus used in CANoe for each channel of the emulated driver.

9.2 CANoe does not find or load my DLL

First check that every dependent dynamically linked library is available. If you have linked your DLL to another one which is not available in the DLL search path, then CANoe cannot load your DLL.

The tool ‘Dependency Walker’ can be used for this checks. You can download it from ‘www.dependencywalker.com’. It is mostly installed together with the Microsoft Visual C++ Compiler.

Please check the exported symbol table of your DLL with the same tool. The following four symbols are required by CANoe in order to recognize it as a CANoe NodeLayer-DLL:

- VIARequiredVersion
- VIASetService
- VIAGetModuleApi
- VIAReleaseModuleApi

If these symbols are missing or are only available in a decorated form (for example: _VIAGetModuleApi@8 or _VIAReleaseModuleApi@4) then most likely you forgot to add a linker definition file (.def) to your Visual Studio project. Please add the file NodeConfig.def or a similar file to your project and rebuild your library.

9.3 Measurement stops after spending some time at a break point

After spending some time at a break point inside the debugger, CANoe stops the measurement with the following message on the write window:

```
Real-time processing has been interrupted (data may be lost!!))
Time jump inside simulation! Measurement stopped
```

This happens, because the debugger stops all execution threads of CANoe, but it cannot stop the hardware interfaces for the bus systems (CANcardX, CANcardXL, LINda, …). After a longer stop inside the debugger, CANoe and the bus interfaces run out of synchronization and CANoe stops the measurement.

When debugging, please switch the 'Working Mode' of CANoe to 'Simulated Bus' and change the entry 'WindowsTimer' in the 'SYSTEM' section of the file 'can.ini' to one. Doing this CANoe runs a simulation without using the hardware interfaces (When you run a simulation without standard Windows timer, CANoe uses the hardware interfaces for the creation of timer events).

```
[SYSTEM]
// WindowsTimer=1: CANoe uses a standard Windows timer in simulation mode
WindowsTimer=1
```

9.4 Debugger always stops inside module VDONGLE

This is a problem with the anti-debug feature of the USB dongle device. If your CANoe installation runs without a hardware dongle at the USB port, you can deactivate it by switching the entry 'UseUSBDongle' in the ini file 'can.ini' to zero.

```
[License]
// Usage of an USB Dongle Device for Licensing
// 1: Search for an USB Dongle Device and use it when the dongle is available
// 0: Do not use USB Dongle Device
UseUSBDongle=0
```

10 Requirements

Supported Compilers and Linkers:

- Microsoft Visual C++ 2010

Supported CANoe / DENoe

- As runtime environment for the emulation, an installation of CANoe / DENoe 8.1.32 or newer is required.
- When you want to emulate a CAN driver, you need a CANoe with CAN option of course.
- For emulation of a LIN driver, CANoe / DENoe with LIN option is required.
- For emulation of a FlexRay driver, CANoe / DENoe 8.1.60 or newer together with FlexRay option is required.
- For emulation of an Ethernet driver, CANoe / DENoe with IP option is required.

Supported Operating Systems

- Microsoft Windows XP or later