

# MICROSAR DET

## Technical Reference

Version 10.0.0

Authors	Hartmut Hörner
Status	Released

## Document Information

### History

Author	Date	Version	Remarks
Hartmut Hörner	2007-11-29	1.0	Initial version
Hartmut Hörner	2008-01-03	1.1	Update to AUTOSAR 3
Hartmut Hörner	2008-04-14	1.2	Naming changed to AUTOSAR short name, screen shots updated. (ESCAN00025687)
Hartmut Hörner	2008-09-16	1.3	Added DET extension mechanism based on callout. Added chapter 4.4.
Hartmut Hörner	2010-01-13	2.0	Update to AUTOSAR 4
Hartmut Hörner	2012-04-20	2.1	Added usage hints related to silent BSW concept in 4.7. (ESCAN00058419)
Hartmut Hörner	2013-04-09	2.2	Added Configurator 5 and service port interface (ESCAN00066511)
Hartmut Hörner	2013-09-13	2.3	Added DLT forwarding support for Configurator 5 (ESCAN00068394, ESCAN00069807)
Hartmut Hörner	2014-12-10	2.3.1	Added description of BCD-coded return value of Det_GetVersionInfo() (ESCAN00079310)
Hartmut Hörner	2015-06-12	2.4.0	File name changed (ESCAN00081049) Added chapter 4.2.
Hartmut Hörner	2016-12-24	10.0.0	Update to AUTOSAR 4.3 (FEAT-1939)

### Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	AUTOSAR_SWS_DET.pdf	4.3.0
[2]	AUTOSAR	AUTOSAR_BasicSoftwareModules.pdf	V1.0.0

## Scope of the Document

This technical reference describes the general use of the MICROSAR Default Error Tracer (DET).

Note that this release of the DET supports only AUTOSAR 4 and the configuration tool Configurator 5. If you need a DET module for previous AUTOSAR versions or tools an older version is required.



### Caution

We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector's release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

## Contents

<b>1</b>	<b>Component History .....</b>	<b>7</b>
<b>2</b>	<b>Introduction.....</b>	<b>8</b>
2.1	Architecture Overview .....	8
<b>3</b>	<b>Functional Description .....</b>	<b>10</b>
3.1	Features .....	10
3.1.1	Deviations .....	10
3.1.2	Additions/ Extensions.....	10
3.1.3	Limitations.....	11
3.2	Initialization .....	11
3.3	States .....	11
3.4	Main Functions .....	11
3.5	Error Handling.....	11
3.5.1	Development Error Reporting.....	11
3.5.2	Production Code Error Reporting .....	12
3.6	Handling of development errors - Debugging with the DET .....	12
3.6.1	Extended Debug Features .....	12
3.6.1.1	Filters.....	13
3.6.1.2	Logging.....	14
3.6.1.3	Break handler .....	15
3.6.1.4	Filtering of DLT forwarding .....	16
3.7	Handling of runtime errors and transient faults.....	16
3.8	Extension of the DET .....	17
3.9	Usage of ErrorId .....	17
<b>4</b>	<b>Integration.....</b>	<b>18</b>
4.1	Scope of Delivery.....	18
4.1.1	Static Files .....	18
4.1.2	Dynamic Files .....	18
4.2	Critical Sections .....	18
4.3	Include Structure.....	18
4.4	Handling of Recursions .....	19
4.5	Multi-core system.....	19
4.6	Partitioning and memory protection.....	19
4.7	Usage Hints for Operation in Safety Related ECUs.....	20
<b>5</b>	<b>API Description.....</b>	<b>21</b>
5.1	Type Definitions .....	21

5.2	Services provided by DET.....	21
5.2.1	Det_Init .....	21
5.2.2	Det_InitMemory.....	22
5.2.3	Det_Start.....	22
5.2.4	Det_GetVersionInfo.....	23
5.2.5	Det_ReportError.....	23
5.2.6	Det_ReportRuntimeError .....	24
5.2.7	Det_ReportTDet_R	

## Illustrations

Figure 2-1	AUTOSAR 4.3 Architecture Overview .....	8
Figure 2-2	Interfaces to adjacent modules of the DET .....	9

## Tables

Table 1-1	Component history.....	7
Table 3-1	Supported AUTOSAR standard conform features .....	10
Table 3-2	Not supported AUTOSAR standard conform features .....	10
Table 3-3	Features provided beyond the AUTOSAR standard.....	10
Table 3-4	Service IDs .....	11
Table 3-5	Errors reported to DET .....	12
Table 4-1	Static files .....	18
Table 4-2	Generated files .....	18
Table 5-1	Type definitions.....	21
Table 5-2	Det_Init.....	22
Table 5-3	Det_InitMemory .....	22
Table 5-4	Det_Start .....	23
Table 5-5	Det_GetVersionInfo .....	23
Table 5-6	Det_ReportError .....	24
Table 5-7	Det_ReportRuntimeError .....	25
Table 5-8	Det_ReportTransientFault.....	25
Table 5-9	Services used by the DET .....	26
Table 5-10	<DetErrorHook> .....	27
Table 5-11	<DetReportRuntimeErrorCallout> .....	27
Table 5-12	<DetReportTransientFaultCallout>.....	28
Table 5-13	DETSERVICE .....	28
Table 7-1	Glossary .....	31
Table 7-2	Abbreviations.....	31

## 1 Component History

The component history gives an overview over the important milestones that are supported in the different versions of the component.

Component Version	New Features
0.01.00	Creation
2.00.00	Update for AUTOSAR Release 2.0
3.00.00	Update for AUTOSAR Release 2.1
3.01.00	GetVersionInfo API added
3.02.00	Extended debug features added
4.00.00	Update for AUTOSAR Release 3 compiler abstraction and memmap added
4.01.00	DET entry callout
5.00.00	Update for AUTOSAR Release 4
6.00.00	Support of Configurator 5 (MSR3)
7.00.00	Support of Configurator 5 (MSR4)
8.00.00	DLT and service port interface
9.00.00	safeBSW
10.00.00	safeBSW ASIL release
20.00.00	Update for AUTOSAR Release 4.3

Table 1-1 Component history

## 2 Introduction

This document describes the functionality, API and configuration of the AUTOSAR BSW module DET as specified in [1].

<b>Supported AUTOSAR Release*:</b>	4.3	
<b>Supported Configuration Variants:</b>	pre-compile	
<b>Vendor ID:</b>	DET_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
<b>Module ID:</b>	DET_MODULE_ID	15 decimal (according to ref. [2])

\* For the detailed functional specification please also refer to the corresponding AUTOSAR SWS.

The DET is the central error handler in the AUTOSAR architecture. All other basic software modules can report development errors, runtime errors and transient faults to the DET.

### 2.1 Architecture Overview

The following figure shows where the DET is located in the AUTOSAR architecture.

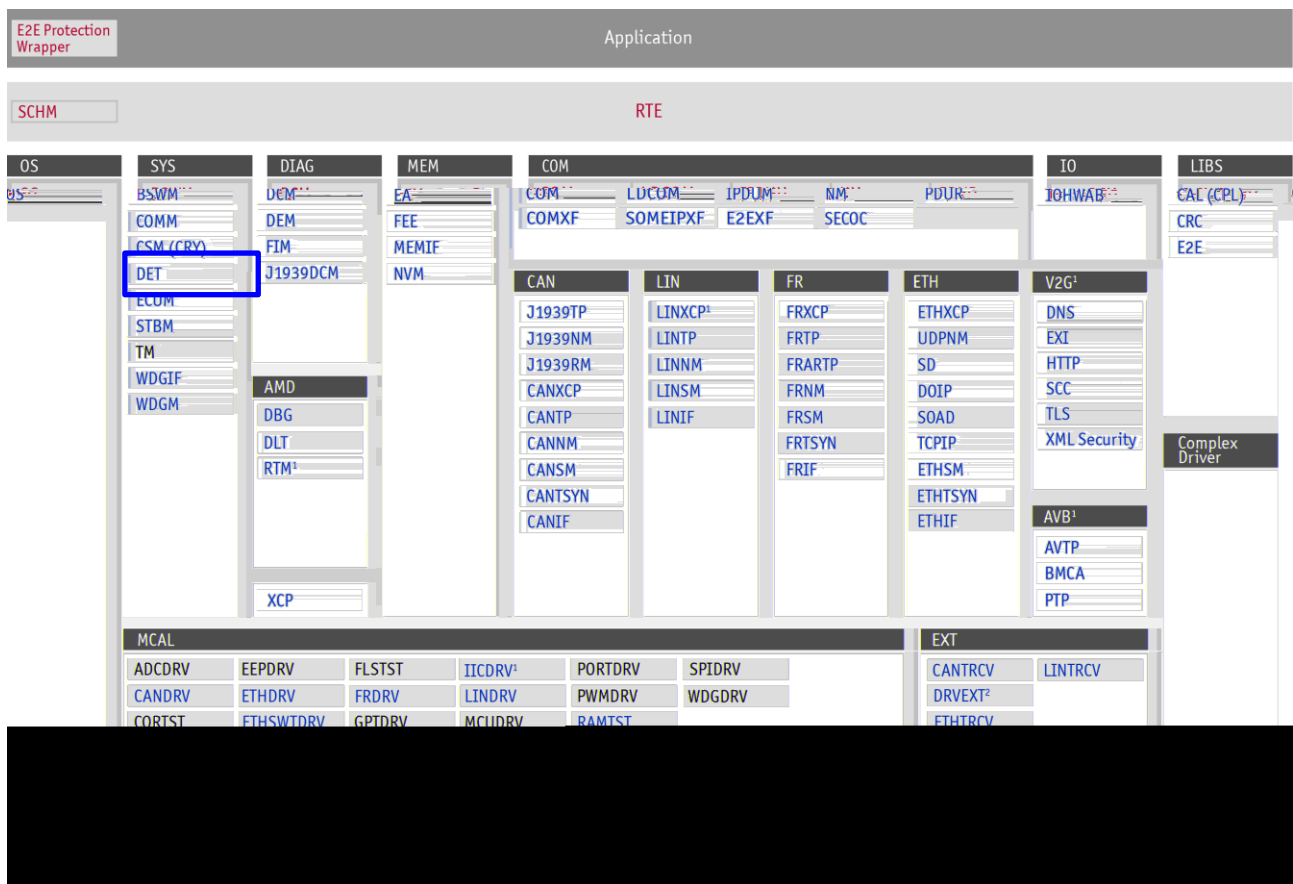


Figure 2-1 AUTOSAR 4.3 Architecture Overview



The next figure shows the interfaces to adjacent modules of the DET. These interfaces are described in chapter 5.

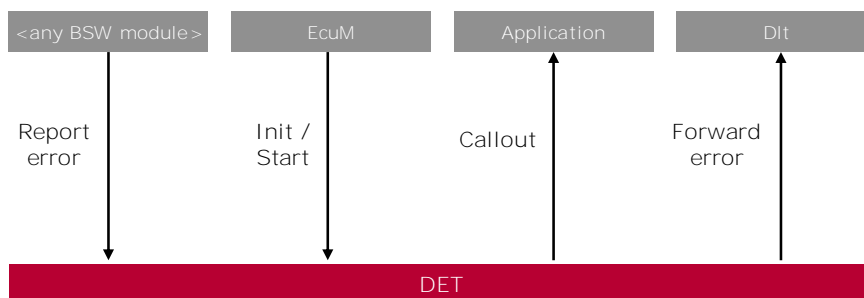


Figure 2-2 Interfaces to adjacent modules of the DET

Applications do not access the services of the BSW modules directly. They use the service ports provided by the BSW modules via the RTE. The service ports provided by the DET are listed in chapter 5.5.1.2 and are defined in [1].

## 3 Functional Description

### 3.1 Features

The features listed in the following tables cover the complete functionality specified for the DET.

The AUTOSAR standard functionality is specified in [1], the corresponding features are listed in the tables

> Table 3-1 Supported AUTOSAR standard conform features

> Table 3-2 Not supported AUTOSAR standard conform features

Vector Informatik provides further DET functionality beyond the AUTOSAR standard. The corresponding features are listed in the table

> Table 3-3 Features provided beyond the AUTOSAR standard

The following features specified in [1] are supported:

Supported AUTOSAR Standard Conform Features
Initialization and start services
Error reporting services
Service port interface
Forwarding of DET errors to the DLT module
Configurable lists of error hooks and callouts

Table 3-1 Supported AUTOSAR standard conform features

#### 3.1.1 Deviations

The following features specified in [1] are not supported:

Not Supported AUTOSAR Standard Conform Features
none

Table 3-2 Not supported AUTOSAR standard conform features

#### 3.1.2 Additions/ Extensions

The following features are provided beyond the AUTOSAR standard:

Features Provided Beyond The AUTOSAR Standard
Extended Debug Features: Since AUTOSAR specifies only the interface and not the functionality of the DET all provided debugging features are AUTOSAR extensions.

Table 3-3 Features provided beyond the AUTOSAR standard

### 3.1.3 Limitations

There are no known limitations.

## 3.2 Initialization

The DET is initialized and operational after the API `Det_Init` has been called. In [1] an additional `Det_Start` service is specified to handle cases where it is necessary to split the initialization in two phases. Since this is not applicable the `Det_Start` function is empty.

The API `Det_InitMemory` may have to be used in addition, please refer to the API description 5.2.2 for details.

## 3.3 States

The DET has no internal state machine, it is operational after initialization.

The module uses its initialization state to perform a check if the module has been initialized.

## 3.4 Main Functions

The DET has no main function since it does not perform cyclic tasks.

## 3.5 Error Handling

### 3.5.1 Development Error Reporting

Development errors detected in DET API functions are reported to the DET using the service `Det_ReportError` as specified in [1].

The reported DET ID is 15.

The reported service IDs identify the services which are described in 5.2. The following table presents the service IDs and the related services:

Service ID	Service
0x00	<code>Det_Init</code>
0x01	<code>Det_ReportError</code>
0x02	<code>Det_Start</code>
0x03	<code>Det_GetVersionInfo</code>
0x04	<code>Det_ReportRuntimeError</code>
0x05	<code>Det_ReportTransientFault</code>

Table 3-4 Service IDs

The errors reported to DET are described in the following table:

Error Code	Description
0x01	DET_E_PARAM_POINTER Det_GetVersionInfo called with NULL parameter pointer

Table 3-5 Errors reported to DET


### 3.5.2 Production Code Error Reporting

The DET does not report production errors.

## 3.6 Handling of development errors - Debugging with the DET

The DET is called for each development error which is reported by other BSW modules. Since it is potentially not safe to continue the program when such an error occurs, the default handling of development errors is an endless loop.

A breakpoint should always be set in this loop which can found in the local function Det\_EndlessLoop. When the breakpoint is hit, the parameters of the function Det\_ReportError 5.2.5 can be inspected in the debugger. By means of these parameters it is possible to find out which error occurred; it is however sometimes more convenient to use a stack trace if the debugger provides this.



**A breakpoint should always be set in the endless loop**

```
#else /* DET_DEBUG_ENABLED */
#if ! defined( C_COMP_ANSI_CANOE )
/* Endless loop for breakpoint in case of development error */
while(1)
{
; /* ##### typical place for a breakpoint if extended debugging support is disabled*/
}
#endif /* C_COMP_ANSI_CANOE */
#endif /* DET_DEBUG_ENABLED */
```

In a simulated target based on CANoe an error message is logged in the CANoe write window.

### 3.6.1 Extended Debug Features

Sometimes setting a breakpoint in the endless loop is not sufficient for debugging, therefore some extended debug features are provided. These features are thought as a debugging aid, thus they are accessible via the debugger and do not have special APIs.

To use these features the attribute “Enable Extended Debug Support” must be enabled.

### 3.6.1.1 Filters

Sometimes it happens that a BSW module reports DET errors which are known to be uncritical. Such errors can be ignored by discarding the related calls to `Det_ReportError`.

To implement this functionality the DET provides a set of filters where the errors to be discarded can be configured. It is possible to use the patterns `0xff` or `0xffff` as wild cards (don't care patterns).



#### Configuration of filters

- configure the required number of filters in configuration tool with the attribute "Number of Global Filters"
- enable filtering globally in the debugger by setting `detStatus.globalFilterActive` to 1

detStatus	{globalFilterActive='□' logActive=0x00 logIndex=0x00 ...}
globalFilterActive	0x01 '□'
logActive	0x00
logIndex	0x00
breakOnLogOverrun	0x00
breakFilterActive	0x00
unlockBreak	0x00

- configure the required filters in the debugger by setting `detGlobalFilter` elements

detGlobalFilter	0x0040b278 detGlobalFilter {moduleId=0x0020 instanceId=0x00 apiId=0x07 ...}
[0x0]	{moduleId=0x0020 instanceId=0x00 apiId='□' ...}
moduleId	0x0020
instanceId	0x00
apiId	0x07 '□'
errorId	0x03 '□'
[0x1]	{moduleId=0x0000 instanceId=0x00 apiId=0x00 ...}
[0x2]	{moduleId=0x0000 instanceId=0x00 apiId=0x00 ...}



#### Filter examples

a) ignore error 3 of API7 of module 20 in instance 0

```
moduleId=20
instanceId=0
apiId=7
errorId=3
```

b) ignore all errors of module 20 in instance 0

```
moduleId=20
instanceId=0
apiId=0xff
errorId=0xff
```

### 3.6.1.2 Logging

The DET provides a log buffer for incoming error messages. Error messages which have been filtered are not logged.

The contents of the log buffer can be viewed with the debugger.



#### Configuration of logging

- configure the required size of the log buffer in the configuration tool with the attribute “Size of Log Buffer”
- enable logging globally in the debugger by setting `detStatus.logActive` to 1

detStatus	{globalFilterActive=0x00 logActive='□' logIndex=0x00 ...}
globalFilterActive	0x00
logActive	0x01 '□'
logIndex	0x00
breakOnLogOverrun	0x00
breakFilterActive	0x00
unlockBreak	0x00



#### Logging example

The variable `detStatus.logIndex` shows the index in the log buffer with the last logged development error. Use the elements of `detLogBuffer` to view the logged errors.

detLogBuffer	0x0040b23c detLogBuffer {moduleId=0x0000 instanceId=0x00 apiId=0x00 ...}
[0x0]	{moduleId=0x0000 instanceId=0x00 apiId=0x00 ...}
[0x1]	{moduleId=0x0001 instanceId='□' apiId='□' ...}
[0x2]	{moduleId=0x0001 instanceId='□' apiId='□' ...}
[0x3]	{moduleId=0x0001 instanceId='□' apiId='□' ...}
moduleId	0x0001
instanceId	0x02 '□'
apiId	0x03 '□'
errorId	0x04 '□'
[0x4]	{moduleId=0x0000 instanceId=0x00 apiId=0x00 ...}
[0x5]	{moduleId=0x0000 instanceId=0x00 apiId=0x00 ...}
[0x6]	{moduleId=0x0000 instanceId=0x00 apiId=0x00 ...}
[0x7]	{moduleId=0x0000 instanceId=0x00 apiId=0x00 ...}
[0x8]	{moduleId=0x0000 instanceId=0x00 apiId=0x00 ...}
[0x9]	{moduleId=0x0000 instanceId=0x00 apiId=0x00 ...}
detStatus	{globalFilterActive=0x00 logActive='□' logIndex='□' ...}
globalFilterActive	0x00
logActive	0x01 '□'
logIndex	0x03 '□'
breakOnLogOverrun	0x00
breakFilterActive	0x01 '□'
unlockBreak	0x00

By default all elements of the variable (s. above) `detLogBuffer` are initialized with zero.

By setting `detStatus.breakOnLogOverrun` in the debugger it is possible to enter the endless loop if the log buffer is full.









### 3.6.1.3 Break handler

For some errors it is possible to continue operation. Therefore it is possible to unlock the endless loop with the debugger to continue the program. Since the same error could occur multiple times and to avoid ending up in the endless loop again it is possible to configure a special filter set for the break handler. Such errors are logged (if logging is active) but do not lead to a break.















#### Configuration of break handler filters

- configure the required number of break handler filters in configuration tool with the attribute “Number of Break Handler Filters”
- enable break handler filtering globally in the debugger by setting `detStatus.breakFilterActive` to 1

	 <code>detStatus</code>	<code>{globalFilterActive=0x00 logActive=0x00 logIndex=0x00 ...}</code>
	 <code>globalFilterActive</code>	<code>0x00</code>
	 <code>logActive</code>	<code>0x00</code>
	 <code>logIndex</code>	<code>0x00</code>
	 <code>breakOnLogOverrun</code>	<code>0x00</code>
	 <code>breakFilterActive</code>	<code>0x01 '□'</code>
	 <code>unlockBreak</code>	<code>0x00</code>

- configure the required break handler filters in the debugger by setting `detBreakFilter` elements

	 <code>detBreakFilter</code>	<code>0x0040b220 detBreakFilter {moduleId=0x0020 instanceId=0x00 apiId=0x00 ...}</code>
		 <code>[0x0]</code>
		<code>{moduleId=0x0020 instanceId=0x00 apiId='y' ...}</code>
	 <code>moduleId</code>	<code>0x0020</code>
	 <code>instanceId</code>	<code>0x00</code>
	 <code>apiId</code>	<code>0xff 'y'</code>
	 <code>errorId</code>	<code>0xff 'y'</code>
	  <code>[0x1]</code>	<code>{moduleId=0x0000 instanceId=0x00 apiId=0x00 ...}</code>
	  <code>[0x2]</code>	<code>{moduleId=0x0000 instanceId=0x00 apiId=0x00 ...}</code>

For some filter examples please refer to 3.6.1.1.

In the following example it is described how the endless loop can be unlocked in the debugger.



### How to unlock the endless loop

Set detStatus.unlockBreak to 1 to leave endless loop:

```
#endif
while(detStatus.unlockBreak==0) /* set this variable to 0 to un
{
; /* ##### typical place for a breakpoint if extended debuggi
}
detStatus.unlockBreak=0; /* PRQA S 3201 */
#else /* DET_DEBUG_ENABLED */
```

#### Watch 1

Name	Value
detLogBuffer	0x0040b23c detLogBuffer {moduleId=0x0000 instanceId=0x00 apiId=...
detStatus	{globalFilterActive=0x00 logActive=0x00 logIndex=0x00 ...}
globalFilterActive	0x00
logActive	0x00
logIndex	0x00
breakOnLogOverrun	0x00
breakFilterActive	0x00
unlockBreak	0x01 '1'

#### 3.6.1.4 Filtering of DLT forwarding

It is sometimes necessary to suppress the forwarding of specific errors to the DLT module to avoid overload. To implement this functionality the DET provides DLT filters where the errors which shall not be forwarded can be configured. It is possible to use the patterns 0xff or 0xffff as wild cards (don't care patterns). These filters work in the same way as the global filters described in 3.6.1.1, please refer to the description there. This functionality is supported for all error reporting APIs.

### 3.7 Handling of runtime errors and transient faults

If errors are reported via the API services `Det_ReportRuntimeError` and `Det_ReportTransientFault` execution continues (i.e. no endless loop is entered).

The following extended debug features are available:

- Logging (3.6.1.2)
- Filtering of DLT forwarding (3.6.1.4)



### 3.8 Extension of the DET

Sometimes the built-in debug features of the DET may not be sufficient or some special handling of errors is required. Examples for such use cases include:

- Logging of DET errors via debug interface
- Transmission of DET errors on a serial bus system
- Error handling which requires direct access to the hardware (e.g. disabling of specific interrupts)
- Complex application specific error handling

To support such extensions the DET provides callout functions which are called first when the DET is entered. The callouts have to be provided by the application. They receive all parameters of the DET's error reporting functions. For details please refer to API description (5.5.1).

### 3.9 Usage of ErrorId

All three error reporting functions have an ErrorId parameter which is forwarded to the DLT module and used by some extended debug features, e.g. filtering and logging. Note, that the DLT module and the extended debug features do not distinguish the actual error kind (i.e. development error, runtime error or transient fault). It is therefore recommended to use a different number range of ErrorIds for each error kind. In the AUTOSAR 4.3 specifications this is already considered for modules reporting runtime errors.

## 4 Integration

This chapter gives necessary information for the integration of the MICROSAR DET into an application environment of an ECU.

### 4.1 Scope of Delivery

The delivery of the DET contains the files which are described in the chapters 4.1.1 and 4.1.2:

#### 4.1.1 Static Files

File Name	Description
Det.c	This is the source file of the DET
Det.h	This is the header file of the DET

Table 4-1 Static files

#### 4.1.2 Dynamic Files

The dynamic files are generated by the configuration tool [config tool].

File Name	Description
Det_cfg.h	This is configuration header file containing pre-compile parameters.
Det_cfg.c	This is configuration source file containing pre-compile parameters.

Table 4-2 Generated files

## 4.2 Critical Sections

The DET has code sections which need protection against preemption. Therefore the DET uses one exclusive area which typically requires an interrupt lock up to the highest interrupt level where DET error reports can be produced:

## 4.4 Handling of Recursions

If DET errors occur within the call context of the DET recursions could be caused. This can happen in a callout function or a subroutine of it if BSW API functions are used there.

These cases are handled by an internal recursion detection mechanism in the DET so the application needs not to take care of them. If the configurable number of recursions is exceeded the DET enters an endless loop. The recursion detection mechanism can be deactivated by setting this number to zero.

Assure that enter and exit functions of the DET's exclusive area do not produce DET errors.

## 4.5 Multi-core system

Usage in a multi-core system is possible under one of the following conditions:

- The DET's exclusive area is mapped to a spinlock
- The logging feature (3.6.1.2) and the recursion detection mechanism (4.4) is not used

## 4.6 Partitioning and memory protection

It is assumed that the BSW is trusted. There may however be exceptional cases which require operation of parts of the BSW in a separate partition, e.g. due to different ASIL level. In such a case the DET is still usable if one of the following concepts is used:

- Memory protection configuration allows write access to the DET data by all partitions containing DET reporting software. Note, that this could lead to corruption of the DET's internal data if it is overwritten by a QM application. As a result the debug features of the DET may not work as expected.
- The logging feature (3.6.1.2) and the recursion detection mechanism (4.4) is not used

## 4.7 Usage Hints for Operation in Safety Related ECUs

The silent BSW concept assures that a BSW module does not corrupt memory of the application and other BSW modules. In this context the following aspects have to be considered for the DET:

- In the callout functions (5.5.1) the DET passes four parameters to the application which could be used as indices by the application. Please note, that the DET does not perform plausibility checks of the value ranges of those parameters because the errors reported to the DET are not known by the DET in advance. The producer and consumer (could both be application code) has to perform plausibility checks of the index parameters if necessary.
- If the extended debug feature “logging” is used depending on the scheduling concept of the ECU DET errors could be logged from different contexts and it has therefore to be secured that the critical section `DET_EXCLUSIVE_AREA_0` reaches up the highest processing level of the application which can produce DET errors.
- The debug features of the DET are intended for the development phase of an ECU. If the DET is used in production code the extended debug features should be switched off because they are only relevant if a debugger is attached.

## 5 API Description

For an interfaces overview please see Figure 2-2.

### 5.1 Type Definitions

The types defined by the DET are described in this chapter.

Type Name	C-Type	Description	Usage
DetInfoType	struct	structure used to configure filters and store log data, using 0xFF for a filter item means don't care	Refer to chapter 3.6.1 for details.
DetStatusType	struct	structure to control the operation of DET debug extension	Refer to chapter 3.6.1 for details.
Det_ConfigType	uint8	Parameter type of the init function – currently not used	The dummy symbol Det_ConfigPtr is provided by the generator.

Table 5-1 Type definitions

### 5.2 Services provided by DET

#### 5.2.1 Det\_Init

Prototype	
void <b>Det_Init</b> ( const Det_ConfigType* ConfigPtr )	
Parameter	
ConfigPtr	Pointer to configuration structure is currently not used by the DET.
Return code	
-	-
Functional Description	
Initializes the DET.	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; Service ID: see table 'Service IDs'</li> <li>&gt; This function is synchronous.</li> <li>&gt; This function is non-reentrant.</li> </ul>	

Expected Caller Context
> Should be called from a safe context on task level

Table 5-2 Det\_Init

## 5.2.2 Det\_InitMemory

Prototype	
void <b>Det_InitMemory</b> ( void )	
Parameter	
-	-
Return code	
-	-
Functional Description	
Initializes the state variable for the un-init check of the DET. If this function is used it must be called before Det_Init.	
Particularities and Limitations	
<ul style="list-style-type: none"> <li>&gt; Service ID: see table 'Service IDs'</li> <li>&gt; This function is synchronous.</li> <li>&gt; This function is non-reentrant.</li> <li>&gt; Should only be called once by the EcuM when the system is started</li> <li>&gt; Only needed if the startup code does not support initialized RAM</li> </ul>	
Expected Caller Context	
> Should be called from a safe context on task level	

Table 5-3 Det\_InitMemory

## 5.2.3 Det\_Start

Prototype	
void <b>Det_Start</b> ( void )	
Parameter	
-	-
Return code	
-	-
Functional Description	
Starts the DET. This service currently has no functionality, i.e. the API function is empty.	

Particularities and Limitations
<ul style="list-style-type: none"><li>&gt; Service ID: see table 'Service IDs'</li><li>&gt; This function is synchronous.</li><li>&gt; This function is reentrant.</li><li>&gt; Call could be omitted</li></ul>
Expected Caller Context
<ul style="list-style-type: none"><li>&gt; No restriction</li></ul>

Table 5-4 Det\_Start

## 5.2.4 Det\_GetVersionInfo

Prototype	
<pre>void Det_GetVersionInfo ( Std_VersionInfoType *versioninfo )</pre>	
Parameter	
versioninfo	Version information of the DET
Return code	
-	-
Functional Description	
This API returns version information, vendor ID and AUTOSAR module ID of the component. The versions are decimal coded.	
Particularities and Limitations	
<ul style="list-style-type: none"><li>&gt; Service ID: see table 'Service IDs'</li><li>&gt; This function is synchronous.</li><li>&gt; This function is reentrant.</li><li>&gt; This API is only available if enabled in configuration</li></ul>	
Expected Caller Context	
<ul style="list-style-type: none"><li>&gt; No restriction</li></ul>	

Table 5-5 Det\_GetVersionInfo

## 5.2.5 Det\_ReportError

Prototype	
<pre>Std_ReturnType Det_ReportError ( uint16 ModuleId, uint8 InstanceId,                                 uint8 ApiId, uint8 ErrorId )</pre>	
Parameter	
ModuleId	Module ID of calling module
InstanceId	The identifier of the index based instance of a module, starting from 0, If the module is a single instance module it shall pass 0 as the InstanceId.
ApiId	ID of API service in which error is detected (defined in SWS of calling module)

ErrorId	ID of detected development error (defined in SWS of calling module)
<b>Return code</b>	
Std_ReturnType	Always E_OK
<b>Functional Description</b>	
<p>Used to report development errors from other BSW modules to the DET. If extended debug features are disabled the DET enters an endless loop in case of an embedded target or issues an error message in the CANoe write window in case of a simulated target.</p> <p>For details please refer to chapter 3.6.</p>	
<b>Particularities and Limitations</b>	
<ul style="list-style-type: none"> <li>&gt; Service ID: see table 'Service IDs'</li> <li>&gt; This function is synchronous.</li> <li>&gt; This function is reentrant.</li> <li>&gt; If this function is called the DET may enter an endless loop, therefore it is strongly recommended to put a breakpoint in the local function <code>Det_EndlessLoop</code>.</li> </ul>	
<b>Expected Caller Context</b>	
<ul style="list-style-type: none"> <li>&gt; No restriction</li> </ul>	

Table 5-6 Det\_ReportError

## 5.2.6 Det\_ReportRuntimeError

<b>Prototype</b>	
<pre>Std_ReturnType Det_ReportRuntimeError ( uint16 ModuleId, uint8 InstanceId,  uint8 ApiId, uint8 ErrorId )</pre>	
<b>Parameter</b>	
ModuleId	Module ID of calling module
InstanceId	The identifier of the index based instance of a module, starting from 0, If the module is a single instance module it shall pass 0 as the InstanceId.
ApiId	ID of API service in which error is detected (defined in SWS of calling module)
ErrorId	ID of detected runtime error (defined in SWS of calling module)
<b>Return code</b>	
Std_ReturnType	Always E_OK
<b>Functional Description</b>	
<p>Used to report runtime errors from other BSW modules to the DET. This function returns and issues an error message in the CANoe write window in case of a simulated target.</p> <p>For details please refer to chapter 3.6.1.4.</p>	





### 5.3 Services used by DET

In the following table services provided by other components, which are used by the DET are listed. For details about prototype and functionality refer to the documentation of the providing component.

Component	API
DLT	Dlt_DetForwardErrorTrace

Table 5-9 Services used by the DET

### 5.4 Callback Functions

The DET does not provide callback functions.

### 5.5 Configurable Interfaces

#### 5.5.1 Callout Functions

At its configurable interfaces the DET defines callout functions. The declarations of the callout functions are provided by the BSW module, i.e. the DET. It is the integrator's task to provide the corresponding function definitions. The definitions of the callouts can be adjusted to the system's needs. The DET callout function declarations are described in the following tables:

##### 5.5.1.1 <DetErrorHook>

Prototype	
<code>Std_ReturnType &lt;DetErrorHook&gt; ( uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId )</code>	
Parameter	
ModuleId	Module ID of calling module
InstanceId	The identifier of the index based instance of a module, starting from 0, If the module is a single instance module it shall pass 0 as the InstanceId.
ApiId	ID of API service in which error is detected (defined in SWS of calling module)
ErrorId	ID of detected development error (defined in SWS of calling module)
Return code	
Std_ReturnType	E_OK or E_NOT_OK If the last error hook which is called returns E_NOT_OK the extended debug features filtering and logging are skipped and Det_ReportError returns immediately.
Functional Description	
This hook routine can be used to forward development error information received by Det_ReportError to the application for further processing.	
Particularities and Limitations	
> none	

Call context
> Called in the context of the corresponding error reporting API which can be task or interrupt.

Table 5-10 &lt;DetErrorHook&gt;

### 5.5.1.2 <DetReportRuntimeErrorCallout>

Prototype	
Std_ReturnType <DetReportRuntimeErrorCallout> ( uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId )	
Parameter	
ModuleId	Module ID of calling module
InstanceId	The identifier of the index based instance of a module, starting from 0, If the module is a single instance module it shall pass 0 as the InstanceId.
ApiId	ID of API service in which error is detected (defined in SWS of calling module)
ErrorId	ID of detected runtime error (defined in SWS of calling module)
Return code	
Std_ReturnType	E_OK or E_NOT_OK If the last error hook which is called returns E_NOT_OK the extended debug feature logging is skipped.
Functional Description	
This hook routine can be used to forward runtime error information received by Det_ReportRuntimeError to the application for further processing.	
Particularities and Limitations	
> none	
Call context	
> Called in the context of the corresponding error reporting API which can be task or interrupt.	

Table 5-11 &lt;DetReportRuntimeErrorCallout&gt;

### 5.5.1.3 <DetReportTransientFaultCallout>

Prototype	
Std_ReturnType <DetReportTransientFaultCallout> ( uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId )	
Parameter	
ModuleId	Module ID of calling module
InstanceId	The identifier of the index based instance of a module, starting from 0, If the module is a single instance module it shall pass 0 as the InstanceId.
ApiId	ID of API service in which error is detected (defined in SWS of calling module)
ErrorId	ID of detected transient fault (defined in SWS of calling module)

Return code	
Std_ReturnType	E_OK or E_NOT_OK The return value of the last error hook which is called will be returned by Det_ReportTransientFault.
Functional Description	
This hook routine can be used to forward transient fault information received by Det_ReportTransientFault to the application for further processing.	
Particularities and Limitations	
> none	
Call context	
> Called in the context of the corresponding error reporting API which can be task or interrupt.	

Table 5-12 &lt;DetReportTransientFaultCallout&gt;

## 5.6 Service Ports

### 5.6.1 Client Server Interface

A client server interface is related to a Provide Port at the server side and a Require Port at client side.

#### 5.6.1.1 Provide Ports on DET Side

At the Provide Ports of the DET the API functions described in 5.2 are available as Runnable Entities. The Runnable Entities are invoked via Operations. The mapping from a SWC client call to an Operation is performed by the RTE. In this mapping the RTE adds Port Defined Argument Values to the client call of the SWC, if configured.

The following sub-chapters present the Provide Ports defined for the DET and the Operations defined for the Provide Ports, the API functions related to the Operations and the Port Defined Argument Values to be added by the RTE.

##### 5.6.1.1.1 DETService

Operation	API Function	Port Defined Argument Values
ReportError ( IN uint8 Apild, IN uint8 ErrorId )	Det_ReportError	uint16 ModuleId uint8 InstanceId
ReportRuntimeError ( IN uint8 Apild, IN uint8 ErrorId )	Det_ReportRuntimeError	uint16 ModuleId uint8 InstanceId

Table 5-13 DETService

A separate DETService Port is needed for each instance of an AUTOSAR SW-C which wants to report errors to the DET module which corresponds to the service port of the SW-C. Each DETService Port needs a ModuleId and an InstanceId as port defined argument values. These values are set automatically and symbolic name value defines for the

ModuleIds and InstanceIds are generated. The required service ports and their ModuleIds and InstanceIds are configured in Configurator 5.



### Migration hints

Note that the DETService Port was specified differently in previous AUTOSAR releases. The InstanceId was part of the operation.

In order to update to the new specification it is recommended to

- ▶ replace the old service port configuration by new instance specific service ports in the configuration tool and to
- ▶ adapt the SWC source code by removing the InstanceId parameter from the service port calls.

If it is not possible to change the SWC source code some compatibility support is provided in the configuration tool which allows configuring a service port for ModuleIds only, i.e. without configuring InstanceIds. In that case a second port called DETServiceLegacy is generated which provides the old operation including an InstanceId. If you want to use DETServiceLegacy it has to be configured instead of DETService in the SWC configuration.

## 6 Configuration

In the DET the attributes can be configured with the following tools:

- > Configuration in Configurator 5, for a detailed description refer to the online help

### 6.1 Configuration Variants

The DET supports the configuration variants

- > `VARIANT-PRE-COMPILE`

The configuration classes of the DET parameters depend on the supported configuration variants. For their definitions please see the `Det_bswmd.arxml` file.

## 7 Glossary and Abbreviations

### 7.1 Glossary

Term	Description
Stack trace	A stack trace (also called stack backtrace or stack traceback) is a report of the active stack frames instantiated by the execution of a program. Although stack traces may be generated anywhere within a program, they are mostly used to aid debugging by showing where exactly an error occurs. The last few stack frames often indicate the origin of the bug.

Table 7-1 Glossary

### 7.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
BSW	Basis SoftWare
DEM	Diagnostic Event Manager
DET	Default Error Tracer
DLT	Diagnostic Log and Trace
pPort	Provide Port
rPort	Require Port
RTE	RunTime Environment
SWC	SoftWare Component

Table 7-2 Abbreviations

