Watchdog 3.0 driver user guide

TRAVEO™ T2G family

About this document

Scope and purpose

This guide describes the architecture, configuration, and usage of the watchdog (WDG) driver. This document explains the functionality of the driver and provides a reference of the driver's API.

The installation, build process, and general information on the use of EB tresos Studio are not within the scope of this document. See the *EB tresos Studio for ACG8 user's guide* [8] for detailed information about this topic.

Intended audience

This document is intended for anyone who uses the WDG driver of the TRAVEO™ T2G family.

Document structure

Chapter 1 General overview gives a brief introduction to the WDG driver, explains the embedding in the AUTOSAR environment, and describes the supported hardware and development environment.

Chapter 2 Using the WDG driver provides detailed steps on how to use the WDG driver in an application.

Chapter 3 Structure and dependencies describes the file structure and the dependencies of the WDG driver.

Chapter 4 EB tresos Studio configuration interface describes the driver's configuration with the EB tresos Studio.

Chapter 5 Functional description gives a functional description of all services offered by the WDG driver.

Chapter 6 Hardware resources gives a description of all hardware resources used by the driver.

The Appendix A and Appendix B provides a complete API reference and access register table.

Abbreviations and definitions

Table 1 **Abbreviation**

Abbreviation	Description
API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
AUTOSAR	Automotive Open System Architecture
BSW	Basic Software. Standardized part of software which does not fulfill a vehicle functional job.
DEM	Diagnostic Event Manager
DET	Default Error Tracer
EB tresos ECU AUTOSAR Suite	A collection of AUTOSAR Basic Software modules and a Runtime Environment integrated in a common configuration and build environment.
EB tresos Studio	Elektrobit Automotive configuration framework

Abbreviation	Description
ILO	Internal Low-speed Oscillator
LF	Source clock of MCWDT which is selectable from several clock sources.
MCAL	Microcontroller Abstraction Layer
MCU	Micro Controller Unit
ms	Millisecond
OS	Operating System
RAM	Random Access Memory
ROM	Read Only Memory
WDG	Watchdog
WDT	Basic Watchdog timer
MCWDT	Multi-Counter Watchdog Timer
SRSS	System Resources Sub-System

Related documents

AUTOSAR requirements and specifications

Bibliography

[1] General specification of basic software modules, AUTOSAR release 4.2.2.

[2] Specification of watchdog driver, AUTOSAR release 4.2.2.

[3] Specification of standard types, AUTOSAR release 4.2.2.

[4] Specification of ECU configuration parameters, AUTOSAR release 4.2.2.

[5] Specification of default error tracer, AUTOSAR release 4.2.2.

[6] Specification of diagnostics event manager, AUTOSAR release 4.2.2.

[7] Specification of memory mapping, AUTOSAR release 4.2.2.

Elektrobit automotive documentation

Bibliography

[8] EB tresos Studio for ACG8 user’s guide.

Hardware documentation

The hardware documents are listed in the delivery notes.

Related standards and norms

Bibliography

[9] Layered software architecture, AUTOSAR release 4.2.2.

Table of contents

About this document ... 1
Table of contents ... 3
1 General overview ... 5
1.1 Introduction to the WDG driver ... 5
1.2 User profile ... 5
1.3 Embedding in the AUTOSAR environment ... 5
1.4 Supported hardware ... 6
1.5 Development environment ... 6
1.6 Character set and encoding ... 6
1.7 Multicore support ... 6
1.7.1 Multicore type ... 7
1.7.1.1 Single core only (multicore type I) ... 7
1.7.1.2 Core-dependent instances (multicore type II) ... 7
1.7.1.3 Core-independent instances (multicore type III) ... 8
1.7.2 Virtual core support ... 8
2 Using the WDG driver ... 9
2.1 Installation and prerequisites ... 9
2.2 Configuring the WDG driver ... 9
2.3 Adapting your application ... 9
2.4 Starting the build process ... 10
2.5 Measuring stack consumption ... 11
2.6 Memory mapping ... 11
2.6.1 Memory allocation keyword ... 11
2.6.2 Memory allocation and constraints ... 12
2.6.3 Assembler code ... 12
3 Structure and dependencies ... 13
3.1 Static files ... 13
3.2 Configuration files ... 13
3.3 Generated files ... 13
3.4 Dependencies ... 14
3.4.1 AUTOSAR OS ... 14
3.4.2 MCU driver ... 14
3.4.3 DET ... 14
3.4.4 Watchdog interface ... 14
3.4.5 DEM ... 14
3.4.6 BSW scheduler ... 14
3.4.7 Error callout handler ... 14
4 EB tresos Studio configuration interface ... 15
4.1 General configuration ... 15
4.2 WDG settings configuration ... 16
4.3 WDG timer configuration list ... 17
4.4 WDG settings fast configuration list ... 19
4.5 WDG settings slow configuration list ... 20
4.6 WDG settings off configuration list ... 21
4.7 WDG DemEventParameter reference ... 22
4.8 WdgMulticore ... 23
4.9 WdgCoreConfiguration ... 23

Table of contents

4.10 WDG external configuration ... 23
4.11 WdgPublishedInformation ... 23
5 Functional description ... 24
5.1 Inclusion ... 24
5.2 Initialization ... 24
5.3 Reconfiguration during runtime ... 24
5.4 API parameter checking ... 24
5.4.1 Wdg_66_IA_Init() ... 24
5.4.2 Wdg_66_IA_SetMode() ... 26
5.4.3 Wdg_66_IA_SetTriggerCondition() ... 26
5.4.4 Wdg_66_IA_GetVersionInfo() ... 27
5.5 Runtime checks ... 28
5.6 Reentrancy ... 28
5.7 Debugging support ... 28
5.8 Functions available without core dependency ... 28
6 Hardware resources ... 29
6.1 Interrupts ... 29
7 Appendix A – API reference ... 30
7.1 Data types ... 30
7.1.1 Wdg_66_IA_ConfigType ... 30
7.1.2 WdgIf_ModeType ... 30
7.2 Constants ... 30
7.2.1 Error codes ... 30
7.2.2 Version information ... 31
7.2.3 Module information ... 31
7.2.4 API service IDs ... 32
7.2.5 Invalid core ID value ... 32
7.3 Functions ... 32
7.3.1 Wdg_66_IA_Init ... 32
7.3.2 Wdg_66_IA_SetMode ... 33
7.3.3 Wdg_66_IA_SetTriggerCondition ... 34
7.3.4 Wdg_66_IA_GetVersionInfo ... 34
7.4 Required callback functions ... 35
7.4.1 DET ... 35
7.4.2 DEM ... 36
7.4.3 Callout functions ... 36
7.4.3.1 Error callout API ... 36
7.4.3.2 Get core ID API ... 37
8 Appendix B – Access register table ... 38
8.1 SRSS (MCWDT) ... 38
8.2 SRSS (WDT) ... 40
Revision history ... 42
Disclaimer ... 43

1 General overview

1.1 Introduction to the WDG driver

The WDG driver is a set of software routines for handling the WDG module. The driver provides services for initializing, changing the operation mode, and setting the trigger condition (timeout). The driver is compliant with the AUTOSAR standard and is implemented according to the *Specification of watchdog driver* [2].

The WDG driver is delivered with a plugin for the EB tresos Studio, which allows you to statically configure the driver options. The driver provides an interface to define symbolic names and the functionality of all configuration options. The WDG driver is designed and implemented for use with additional WDG drivers. All API functions, DEM errors, DET errors, and data types are prefixed with vendor specific string “_66_IA_”. IA is the short form for InternalA.

1.2 User profile

This guide is intended for users with a basic knowledge of the following domains:

- Embedded systems
- C programming language
- AUTOSAR standard
- Target hardware architecture

1.3 Embedding in the AUTOSAR environment

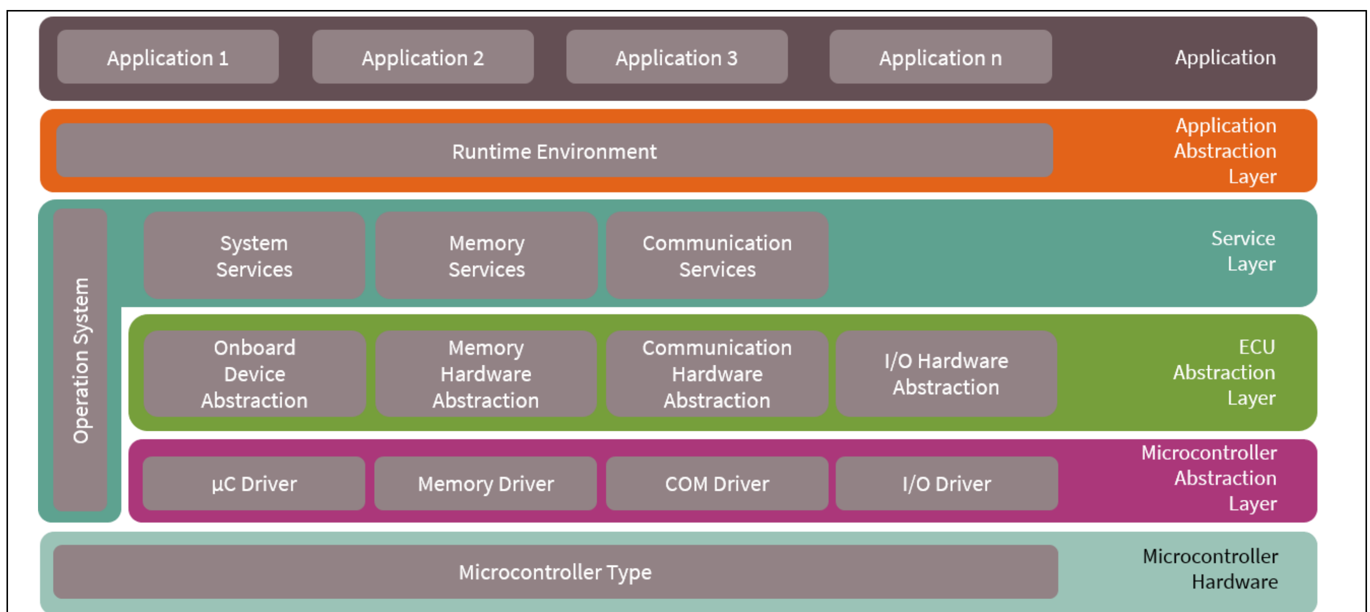

Figure 1 Overview of AUTOSAR software layers

Figure 1 depicts the layered AUTOSAR software architecture. The WDG driver (Figure 2) is part of the MCAL, the lowest layer of basic software in the AUTOSAR environment.

As an internal microcontroller driver, WDG driver provides a standardized and microcontroller-independent interface to higher software layers for accessing WDG timer of the ECU hardware.

For an overview of the AUTOSAR layered software architecture, see the *Layered software architecture* [9].

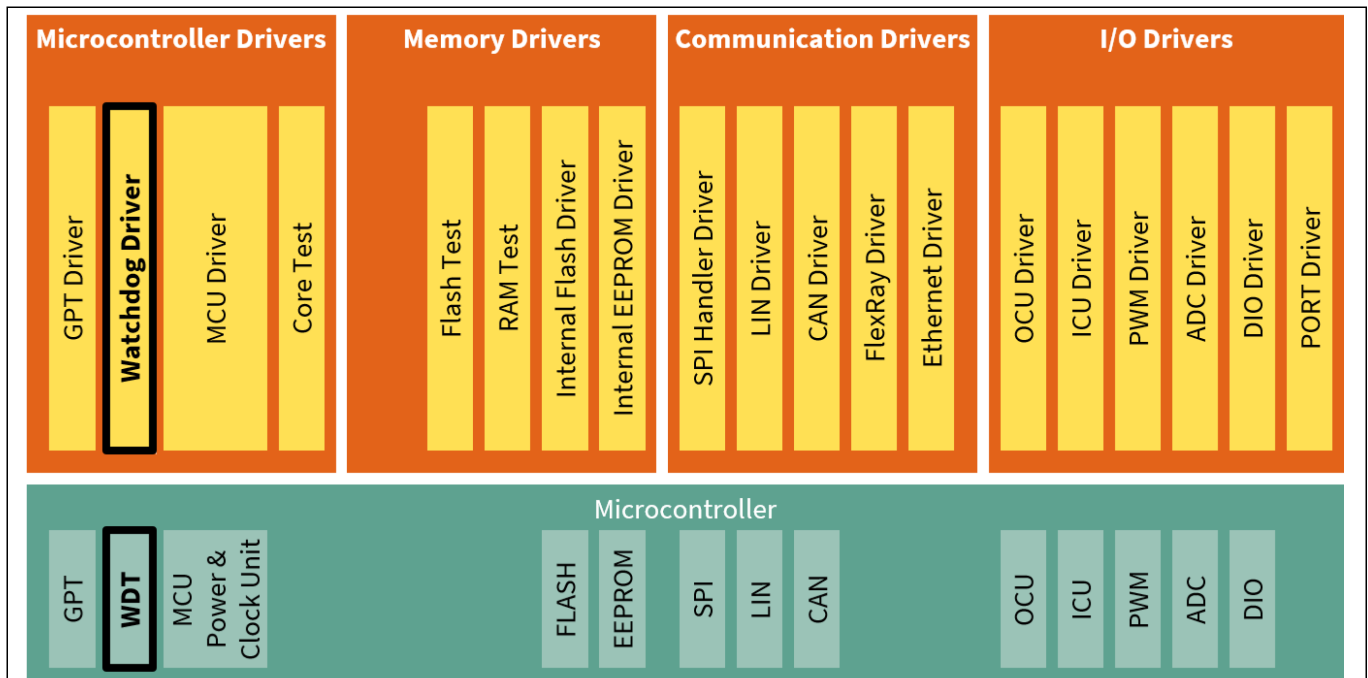

Figure 2 WDG driver in MCAL layer

1.4 Supported hardware

This version of the WDG driver supports the TRAVEO™ T2G microcontroller. The supported derivatives are listed in the release notes.

Additional derivatives which contain only a subset of the capabilities of one derivative mentioned above can be supported by providing a resource file with its properties.

1.5 Development environment

The development environment corresponds to AUTOSAR release 4.2.2. The modules BASE, DEM, MAKE, MCU, and RESOURCE are needed for proper functionality of the WDG driver.

1.6 Character set and encoding

All source code files of the WDG driver are restricted to the ASCII character set. The files are encoded in UTF-8 format, with only the 7-bit subset (values 0x00 … 0x7F) being used.

1.7 Multicore support

The WDG driver supports the multicore type II. `Wdg_66_IA_GetVersionInfo()` also supports multicore type III. For each multicore type, see the following sections.

Note: *If multicore type III is required, the section including the data related to the read-only API or atomic write API must be allocated to the memory, and can be read from any cores.*

1.7.1 Multicore type

In the following section, type I, type II, and type III are defined as multicore characteristics.

1.7.1.1 Single core only (multicore type I)

For this multicore type, the driver is available only on a single core. This type is referred as "Multicore Type I".

Multicore type I has the following characteristic:

- The peripheral channels are accessed by only one core.

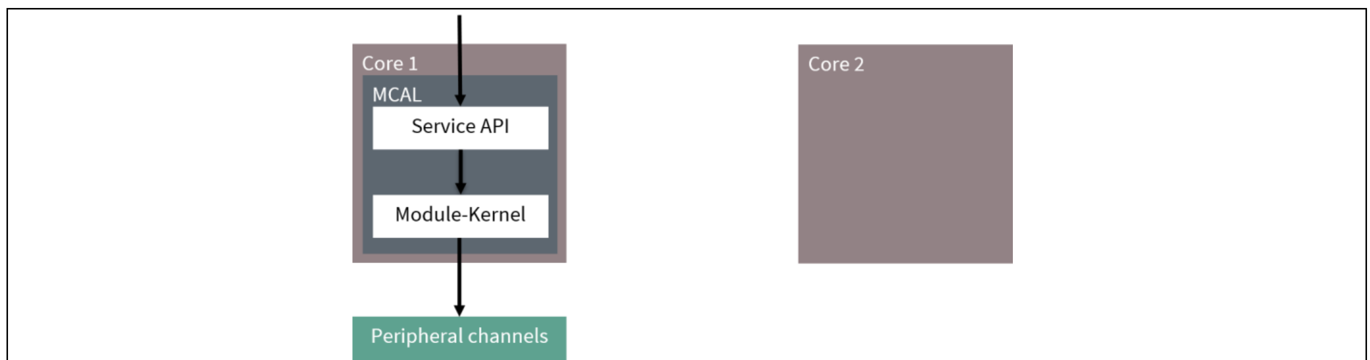

Figure 3 Overview of the multicore type I

1.7.1.2 Core-dependent instances (multicore type II)

For this multicore type, the driver has core-dependent instances with individually allocable hardware. This type is referred as "Multicore Type II".

Multicore type II has the following characteristics:

- The driver code is shared among all cores
  - A common binary is used for all cores
  - A configuration is common for all cores
- Each core runs an instance of the driver
- Peripheral channels and their data can be individually allocated to cores, but cannot be shared among cores
- One core will be the master; the master core must be initialized first
  - Cores other than the master core are called satellite cores.

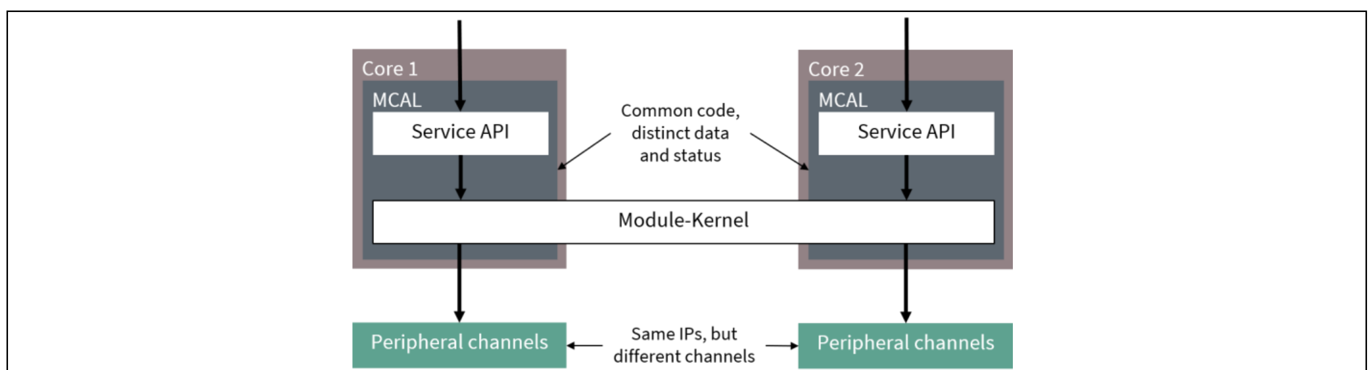

Figure 4 Overview of the multicore type II

1.7.1.3 Core-independent instances (multicore type III)

For this multicore type, the driver has core-independent instances with globally available hardware. This type is referred as “Multicore Type III”.

Multicore type III has the following characteristics:

- The code of the driver is shared among all cores
  - A common binary is used for all cores
  - A configuration is common for all cores
- Each core runs an instance of the driver
- Peripheral channels are globally available for all cores

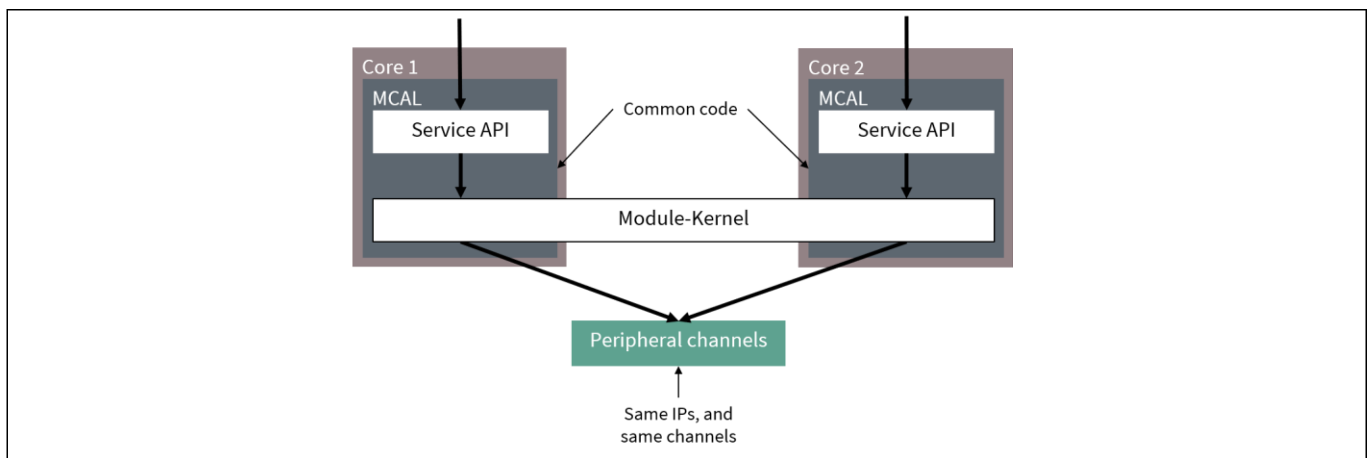

Figure 5 Overview of the multicore type III

1.7.2 Virtual core support

The WDG driver supports a number of cores. The configured cores need not be equal to the physical cores.

The WDG driver calls a configurable callout function (`WdgGetCoreIdFunction`) to identify the core that is currently executing the code. This function can be implemented in the integration scope. The function can be written such that it does not return the physical core, but instead returns the SW partition ID, OS application ID, or any attribute/parameter. By interpreting these as the core, the WDG driver can support multiple SW partitions on a single physical core.

2 Using the WDG driver

This chapter describes all necessary steps to incorporate the WDG driver into your application.

2.1 Installation and prerequisites

Note: Before continuing with this chapter, see the EB tresos Studio for ACG8 user's guide [8]. You can find the required basic information about the installation procedure of EB tresos ECU AUTOSAR components and the usage of the EB tresos Studio and the EB tresos ECU AUTOSAR build environment. You will also find information on how to setup and integrate your own application within the EB tresos ECU AUTOSAR build environment.

The installation of the WDG driver complies with the general installation procedure for EB tresos ECU AUTOSAR components given in the *EB tresos Studio for ACG8 user's guide* [8] . If the driver has been successfully installed, the driver will appear in the module list of the EB tresos Studio (see *EB tresos Studio for ACG8 user's guide* [8]).

This guide assumes that the project is properly set up and is using the application template as described in the *EB tresos Studio for ACG8 user's guide* [8]. This template provides the necessary folder structure, project, and makefiles needed to configure and compile an application within the build environment. You must be familiar with the usage of the command line shell.

2.2 Configuring the WDG driver

This section provides an overview of the configuration structure, defined by AUTOSAR, on how to use the WDG driver.

The following basic containers are used to specify the behavior of WDG driver:

- `WdgGeneral`: This container is mainly used to restrict or extend the API of the WDG module and enable or disable DET.
- `WdgDemEventParameterRefs`: This container holds references to the `DemEventParameter` elements, which will be invoked using the Dem_`ReportErrorStatus` API in case the corresponding error occurs.
- `WdgSettingsConfig`: This container holds the watchdog settings for each mode, all post-build parameters are handled via this container.
- `WdgMulticore`: This container contains the multicore configuration of the WDG driver.

The configuration data stored by containers `WdgExternalConfiguration` and `WdgPublishedInformation` are not processed.

For detailed information and description, see 4 EB tresos Studio configuration interface.

2.3 Adapting your application

To use the WDG driver in your application, include the MCU and WDG driver header files by adding the following lines of code in your source file:

```
#include "Mcu.h" /* MCU Driver */
#include "Wdg_66_IA.h" /* WDG Driver */
```

This publishes all needed functions, prototypes, and symbolic names of the configuration to the application. Also, you need to implement the error callout function for ASIL safety extension.

Declare the error callout function in file specified by the `WdgIncludeFile` parameter and implement the error callout function in your application (see 7.4 Required callback functions, Error callout API).

The error callout function name can be configured by the `WdgErrorCalloutFunction` parameter.

In the next step, the MCU and WDG need to be initialized and configured. The steps to configure the WDG driver in the EB tresos Studio are explained 4 EB tresos Studio configuration interface. The WDG module will be automatically enabled if an appropriate parameter configuration of the WDG module is available in the application.

The MCU and WDG initialization should be done for both master core and satellite cores:

```
Mcu_Init(&Mcu_Config[0]);
Wdg_66_IA_Init(&Wdg_66_IA_Config[1]);
```

The master core must be initialized prior to the satellite core. All cores must be initialized with the same configuration.

To trigger watchdog timer (WDT/MCWDT) with the timeout parameter or trigger an immediate watchdog reset (WDR), the `Wdg_66_IA_SetTriggerCondition()` function must be called. In case of RAM mode, the trigger routine should be called by the application directly instead of the `Wdg_66_IA_SetTriggerCondition()` function after flash area is erased.

```
Wdg_66_IA_SetTriggerCondition(1000);
```

Your application must provide the notification functions and its declarations that you configured. The file containing the declarations must be included using the `WdgGeneral/WdgIncludeFile` parameter. The notification functions take no parameters and have void return type:

```
void MyNotificationFunction(void)
{
/* Insert your code here */
}
```

Note: *Notification function is controlled by `WdgEnableWarningIrq` which uses a warning interrupt to notify the application before WDR happens. If this interrupt is enabled, an interruption is triggered when the watchdog counter reaches the warning limit value. Notification function does not work correctly if this interrupt is disabled. Set up the interrupt levels appropriately according to system environment.*

2.4 Starting the build process

Do the following to build your application.

Note: *For a clean build, use the build command with target `clean_all` before. (make `clean_all`)*

1. On the command shell, type the following command to generate the necessary configuration dependent files. See 3.3 Generated files.

```
> make generate
```

2. Type the following command to resolve the required file dependencies:

```
> make depend
```

3. Type the following command to compile and link the application:

```
> make (optional target: all)
```

The application is now built. All files are compiled and linked to a binary file, which can be downloaded to the target hardware.

2.5 Measuring stack consumption

Do the following to measure stack consumption. It requires the Base module for proper measurement.

Note: All files (including library files) should be rebuilt with the dedicated compiler option. The executable file built in this step must be used only to measure stack consumption.

1. Add the following compiler option to the Makefile to enable stack consumption measurement:

   ```
   -DSTACK_ANALYSIS_ENABLE
   ```

2. Type the following command to clean library files:

   ```
   > make clean_lib
   ```

3. Follow the build process described in 2.4 Starting the build process.
4. Measure the stack consumption by following the instructions given in the release notes.

2.6 Memory mapping

The *Wdg_66_IA_MemMap.h* file in the *$(TRESOS_BASE)/plugins/MemMap_TS_T40D13M0I0R0/include* directory is a sample. This file is replaced by the file generated by MEMMAP module. Input to MEMMAP module is generated as *Wdg_Bswmd.arxml* in the *$(PROJECT_ROOT)/ output/generated/swcd* directory of your project folder.

2.6.1 Memory allocation keyword

- `WDG_66_IA_START_SEC_CODE_ASIL_B`/ `WDG_66_IA_STOP_SEC_CODE_ASIL_B`

  The memory section type is CODE. All executable code is allocated in this section.

- `WDG_66_IA_START_SEC_CONST_ASIL_B_UNSPECIFIED`/ `WDG_66_IA_STOP_SEC_CONST_ASIL_B_UNSPECIFIED`

  The memory section type is CONST. The following constants are allocated in this section:

  - All configuration data except reset
  - Hardware register base address data
  - Pointer to the current driver status
  - Pointer to the current driver mode
  - Pointer to the current timeout value

- `WDG_66_IA_START_SEC_CONST_ASIL_B_32`/ `WDG_66_IA_STOP_CONST_INIT_ASIL_B_32`

  The memory section type is CONST. The following constant is allocated in this section:

  - Trigger function size

- `WDG_CORE[MasterCoreId]_66_IA_START_SEC_VAR_INIT_ASIL_B_GLOBAL_8`/ `WDG_CORE[MasterCoreId]_66_IA_STOP_SEC_VAR_INIT_ASIL_B_GLOBAL_8`

  `MasterCoreId` means the `WdgCoreConfigurationId` command specified in the `WdgMasterCoreReference` reference command.

The memory section type is VAR. The following variables are allocated in this section:

- SRSS version. See Hardware documentation for details.

- WDG_CORE[CoreId]_66_IA_START_SEC_VAR_INIT_ASIL_B_GLOBAL_UNSPECIFIE/WDG_CORE[CoreId]_66_IA_STOP_SEC_VAR_INIT_ASIL_B_GLOBAL_UNSPECIFIED

  The memory section type is VAR. The following variables are allocated in this section:

  - Current driver status
  - Pointer to the configuration data

- WDG_CORE[CoreId]_66_IA_START_SEC_VAR_INIT_ASIL_B_LOCAL_UNSPECIFIED/WDG_CORE[CoreId]_66_IA_STOP_SEC_VAR_INIT_ASIL_B_LOCAL_UNSPECIFIED

  The memory section type is VAR. The following variables are allocated in this section:

  - Current mode
  - Current timeout value

2.6.2 Memory allocation and constraints

All memory sections that store init or uninit status must be zero-initialized before any driver function is executed on any core. If core consistency checks are disabled, inconsistent parameters are detected and reported by PPU and SMPU.

- WDG_CORE[WdgCoreConfigurationId]_START_VAR_[INIT_POLICY]_ASIL_B_LOCAL_[ALIGNMENT] / WDG_CORE[WdgCoreConfigurationId]_STOP_VAR_[INIT_POLICY]_ASIL_B_LOCAL_[ALIGNMENT]

  This section is read/write accessed only from the core represented by WdgCoreConfigurationId. Therefore, this section can be allocated to any RAM region. It is recommended to allocate the section to cache-able SRAM, not TCRAM.

- WDG_ CORE[WdgCoreConfigurationId]_START_VAR_[INIT_POLICY]_ASIL_B_GLOBAL_[ALIGNMENT] / WDG_ CORE[WdgCoreConfigurationId]_STOP_VAR_[INIT_POLICY]_ASIL_B_GLOBAL_[ALIGNMENT]

  This section is read/write accessed from the core represented by WdgCoreConfigurationId and read accessed from the other cores. Therefore, this section must not be allocated to TCRAM. For the core represented by WdgCoreConfigurationId, this section must be allocated to either non-cache-able or write-through cache-able SRAM area. For performance, it is recommended to allocate the section to write-through cache-able SRAM. For other cores, this section must be allocated to non-cache-able SRAM area.

- STACK section

  TCRAM has dedicated memory for each core at the same address, and because of its performance it is recommended to allocate STACK to TCRAM.

For the details of INIT_POLICY and ALIGNMENT, see the *Specification of memory mapping* [7].

2.6.3 Assembler code

Assembler code for the WDG driver uses the fixed memory section names in Table 2.

Table 2 Fixed section names

Section name	Allocate area
WDG_66_IA_TRIGGER	ROM area

3 Structure and dependencies

The WDG driver consists of static, configuration, and generated files.

3.1 Static files

- $(PLUGIN_PATH)=$(TRESOS_BASE)/plugins/WDG_TS_* is the path to the WDG module plugin.
- $(PLUGIN_PATH)/lib_src contains all static source files of the WDG driver. These files represent the functionality of the driver. These files are independent of any configuration sets. The files are packed together into a static library.
- $(PLUGIN_PATH)/src contains configuration dependent source files or device specific files. Each file will be rebuilt when the configuration set is changed.

All necessary source files will be automatically compiled and linked during the build process and all include paths will be set if the WDG driver is enabled.

- $(PLUGIN_PATH)/include is the basic public include directory needed by the user to include Wdg_66_IA.h.
- $(PLUGIN_PATH)/autosar directory contains the AUTOSAR ECU parameter definition with vendor, architecture, and device specific adaptations to create a correct matching parameter configuration for the WDG module.

3.2 Configuration files

The configuration of the WDG driver is done with the EB tresos Studio. When saving a project, the configuration description is written to the *Wdg.xdm* file, which is in *$(PROJECT_ROOT)/config* of your project folder. This file serves as input for the generation of the configuration dependent source and header files during the build process.

3.3 Generated files

During the build process the following files are generated based on the current configuration description. These files are in the folder *output/generated* of your project folder.

include/Wdg_66_IA_Cfg.h, include/Wdg_66_IA_IncludeFiles.h, include/Wdg_66_IA_Cfg_Arch.h and include/Wdg_66_IA_PBcfg.h define all symbolic names, macros, and configuration settings needed by the module.

- *src/Wdg_66_IA_PBcfg.c* contains the constant structure for the WDG configuration.
- *src/Wdg_66_IA_Irq.c* contains the interrupt service routine for the warning interrupt.
- src/Wdg_66_IA_Trigger_Asm_GHS.s defines the trigger routine.
- *src/ Wdg_66_IA_CalloutWrapper.c* defines the internal function to get the core ID.

Note: *Generated source files need not to be added to your application make file. These files will be compiled and linked automatically during the build process.*

- *swcd/Wdg_Bswmd.arxml* contains Bsw module description.

Note: *Additional steps are required for the generation of BSW module description. In EB tresos Studio, follow the menu path **Project** > **Build Project** and click **generate_swcd**.*

3.4 Dependencies

3.4.1 AUTOSAR OS

The AUTOSAR operating system handles the interrupts used by the WDG driver. See 6.1 Interrupts for more information.

`GetCoreID` can optionally be set to the configuration parameter `WdgGetCoreIdFunction`.

3.4.2 MCU driver

`Mcu_GetCoreID` can optionally be set to the configuration parameter `WdgGetCoreIdFunction`. See the MCU driver's user guide for details.

3.4.3 DET

If the default error detection is enabled in the WDG module configuration, the DET needs to be installed, configured and integrated into the application as well.

3.4.4 Watchdog interface

The WDG driver uses types of the WDG interface. Therefore, the WDG interface (respectively the *WdgIf_Types.h*) needs to be available to build the WDG driver.

3.4.5 DEM

The DEM needs to be always installed, configured, and integrated with the application as well.

You should use this driver via the *Wdg_66_IA.h* interface and be responsible to assign the standard `WDG_E_DISABLE_REJECTED`, `WDG_E_MODE_FAILED`, `WDG_E_HW_TIMEOUT`, and `WDG_E_WDG_STOPPED` via macros.

3.4.6 BSW scheduler

The WDG driver uses the following services of the BSW scheduler to enter and leave critical sections.

- `SchM_Enter_Wdg_66_IA_WDG_EXCLUSIVE_AREA_0(void)`
- `SchM_Exit_Wdg_66_IA_WDG_EXCLUSIVE_AREA_0(void)`

You must ensure that the BSW scheduler is properly configured and initialized before using the WDG driver.

Note: *These services are valid if only WDT is configured as watchdog timer for the core.*
In other words, if MCWDT is configured, these services would not be effective.

3.4.7 Error callout handler

The error callout handler is called on every error that is detected, regardless of whether default error detection is enabled or disabled. The error callout handler is an ASIL safety extension that is not specified by AUTOSAR. It is configured via configuration `WdgErrorCalloutFunction` parameter.

4 EB tresos Studio configuration interface

The GUI is not part of this delivery. For further information, see the *EB tresos Studio for ACG8 user's guide* [8].

4.1 General configuration

The module comes with the following preconfigured with default settings, which must be adapted when necessary:

- `WdgDevErrorDetect` enables or disables the development error notification for the WDG module.
  - Setting this parameter to FALSE will disable the notification of development errors via DET. However, in contrast to
  - AUTOSAR specification, detection of development errors is still enabled as safety mechanisms (fault detection).
- `WdgDisableAllowed` enables or disables the option to disable the WDG driver during runtime.
- `WdgIndex` represents the WDG driver's ID that can be referenced by the WDG interface. This value will be assigned to the following symbolic name:
  - The symbolic name derived of the `WdgGeneral` container short name prefixed with "`WdgConf_`" (`WdgConf_WdgGeneral_WdgGeneral`).
- `WdgInitialTimeout` represents the trigger condition to be initialized during `Init` function. This condition should not be higher than `WdgMaxTimeout`. The range is 0-65.535 seconds.

Note: *More than one mode is supported as default mode (fast or slow), so `WdgInitialTimeout` is not used any more. Instead, `WdgFastTimeoutValue / WdgSlowTimeoutValue` are used for initial timeout value of each mode.*

- `WdgMaxTimeout` represents maximum timeout to which the WDG trigger condition can be initialized. The input parameter of `Wdg_66_IA_SetTriggerCondition()` should not be higher than `WdgMaxTimeout`. The range is 0-65.535 seconds. The parameter of `Wdg_66_IA_SetTriggerCondition()` is a millisecond unit value; therefore, the WDG module converts `WdgMaxTimeout` to a millisecond value and stores this value as an inside parameter.

Note: *When MCWDT is configured, the maximum timeout would be limited to a value lower than 65.535 according to `WdgTimerClockRef` (see 4.3 WDG timer configuration list).*

This is because the watchdog timer counter of MCWDT is 16 bits, although WDT has 32-bit watchdog timer counter.

For example, when the `WdgTimerClockRef` is 32768Hz, duration of 1 count of the timer counter is 1 / 32768 seconds.

The maximum value of 16-bit counter is 0xFFFF (65535).

Then the maximum timeout of MCWDT is 1.999 (65535 / 32768) seconds.

- `WdgRunArea` indicates whether the WDG trigger execution area is either from ROM (Flash) or RAM.
- `WdgTriggerLocation` is the location (memory address) of the WDG trigger routine.

Note: *`WdgTriggerLocation` shows the trigger function name. The function name is specific (i.e. `Wdg_66_IA_ActivateTrigger`) and cannot be edited.*

- `WdgTriggerAddress` is location (memory address) of the WDG trigger routine (Actual address). The range is between the base address to the end address of SRAM (SRAM0, SRAM1 or SRAM2, it depends on hardware specification) area.

Note: *`WdgTriggerAddress` should be multiples of four and within an allowed range; otherwise errors would occur in configuration phase. This value is editable only when `WdgRunArea` is set to RAM. Bit0 of the address should be set to ON (1) when calling the WDG trigger function by jumping directly from Arm® instructions, because the code is assembled by thumb instructions.*

For example, if the address in RAM is configured to 0x8000000, then the calling of WDG trigger function should use (0x8000000 | 0x0000001).

- `WdgVersionInfoApi` enables or disables the version information API.
- `WdgDemEventModeFailed` enables or disables the DEM `ModeFailed` Event checks and report.
- `WdgDemEventDisableRejected` enables or disables the DEM `DisableRejected` Event checks and report.
- `WdgDemEventHwTimeout` enables or disables the DEM `HardwareTimeout` Event checks and report.
- `WdgDemEventWdgStopped` enables or disables the DEM `WdgStopped` Event checks and report.
- `WdgErrorCalloutFunction` is used to specify the error callout function name. The function is called on every error. The ASIL level of this function limits the ASIL level of the WDG driver.

Note: *`WdgErrorCalloutFunction` must be valid a C function name, otherwise an error would occur in configuration phase.*

- `WdgIncludeFile` is a list of the filenames that should be included within the driver. Any application-specific symbol that is used by the WDG configuration (e.g. error callout function) should be included by configuring this parameter.

Note: *`WdgIncludeFile` must be a unique filename with extension .h; otherwise some errors would occur in configuration phase.*

4.2 WDG settings configuration

- `WdgDefaultMode` is the default mode for WDG driver initialization.
  - `WDGIF_FAST_MODE`
  - `WDGIF_SLOW_MODE`
  - `WDGIF_OFF_MODE`

Note: *WDGIF_OFF_MODE is only supported when WdgDisableAllowed is TRUE.*

4.3 WDG timer configuration list

- `WdgTimerConfigList` is the array of the watchdog timer configuration which is used by WDG driver:

Note: *WDG driver can configure one or two watchdog timers for one core.*
Supported combinations of the watchdog timers for one core follow these three patterns:
- Only MCWDT
- Only WDT
- MCWDT and WDT
In case MCWDT and WDT are configured for one core, MCWDT must be set before WDT.
If MCWDT and WDT are configured and an MCWDT reset occurs, the WDT keeps running and causes an undesired reset according to the WDT settings when the WDT counter expires.
The WDT reset cannot be avoided.

- `WdgCoreAssignment` specifies the reference to `WdgCoreConfiguration` for the core assignment.

Note: *`WdgCoreAssignment` must have the target's `WdgCoreConfiguration` setting.*

The same resource cannot be allocated to multiple cores.

- `WdgCPUSelect` is the core number where the MCWDT assigns the DeepSleep action. The range is 0-3.

Note: *The core number is defined by the hardware specification.*

- `WdgTimerSelection` is the watchdog timer which is configured to be used:
  - `WDG_TIMER_WDT`: Basic watchdog timer
  - `WDG_TIMER_MCWDT[n]`: Multi-Counter watchdog timer.
    [n]: the number of specific MCWDT channel, the maximum number of [n] is defined by the hardware specification.
- `WdgStopWDT` specifies whether WDG driver stops WDT during initialization to avoid WDT would be running by default setting and cause WDR.

Note: *This parameter is enabled if only MCWDT is configured for the same core.*
Make sure that the core for which this parameter is TRUE is initialized first, and the core to which WDT is assigned is initialized next. If you reverse the order, WDT will be stopped unexpectedly.

- `WdgEnableWarningIrq` enables or disables a warning notification for the specific watchdog timer. This function is used for notifying the application before the watchdog timer expires. The notification function's name can be configured with `WdgWarningNotification`. If `WdgEnableWarningIrq` is enabled, then the notification function must be provided by the application. Also, the warning interrupt must be configured properly; see 6.1 Interrupts.

Note: *If this interrupt is enabled, the following sequence takes place when the watchdog counter reaches to warn limit value:*
1. Watchdog counter reaches to warn limit value.
2. Warning interrupt is triggered
3. Trigger the action which is configured by WDG driver when watchdog counter reaches trigger timeout value
Step 2 will not occur, if the warning interrupt is disabled.

- `WdgWarningNotification` specifies a function name to be called in case of a warning interrupt. This parameter is ignored if `WdgEnableWarningIrq` is disabled.

Note: *`WdgWarningNotification` should be a C function name. Notifications must be declared and defined outside WDG module. The file containing the declarations must be included using the parameter `WdgGeneral/WdgIncludeFile`.*

- `WdgDebugModeConfig` is used to freeze or run the watchdog during the debugging mode:
  - `WDG_DEBUGMODE_FREEZE`: The watchdog is configured to freeze during debugging mode.
  - `WDG_DEBUGMODE_RUN`: The watchdog is configured to run during debugging mode.

Note: *This parameter must be same for all configured timers.*

- `WdgDeepsleepConfig` is used to freeze or run the watchdog mode services in Deep Sleep mode:
  - `WDG_DEEPSLEEP_FREEZE`: The watchdog is configured to freeze during Deep Sleep mode.
  - `WDG_DEEPSLEEP_RUN`: The watchdog is configured to run during Deep Sleep mode.

Note: *This parameter would be invalid for WDT if MCWDT and WDT are configured for the core.*

- `WdgHibernateConfig` is used to freeze or run the watchdog mode services in Hibernate mode:
  - `WDG_HIBERNATE_FREEZE`: The watchdog is configured to freeze during Hibernate mode.
  - `WDG_HIBERNATE_RUN`: The watchdog is configured to run during Hibernate mode.

Note: *This parameter is invalid for MCWDT.*

- `WdgLowerActionConfig` is the action when the watchdog timer is serviced before lower limit is reached:
  - `WDG_ACTION_RESET`: The lower action is configured to cause a reset. This can be set only for WDT.
  - `WDG_ACTION_FAULT`: The lower action is configured to generate a fault. This can be set only for MCWDT.
  - `WDG_ACTION_FAULT_RESET`: The lower action is configured to generate a fault and then cause a reset. This can be set only for MCWDT.
- `WdgUpperActionConfig` is the action when the watchdog timer is reached upper limit:
  - `WDG_ACTION_RESET`: The upper action is configured to cause a reset. This can be set only for WDT.
  - `WDG_ACTION_FAULT`: The upper action is configured to generate a fault. This can be set only for MCWDT.
  - `WDG_ACTION_FAULT_RESET`: The upper action is configured to generate a fault and then cause a reset. This can be set only for MCWDT.
- `WdgTimerClockRef` is the reference to the MCU clock configuration.

  This parameter is used to calculate maximum timeout that can be set to the hardware.

Note: *MCU clock reference will only support `McuClock` that includes `MCU_CLOCK_LF*` and `MCU_CLOCK_ ILO0*` in `McuClockReferencePoint`.*

  - `McuClock`: Clock reference point.
  - `McuClockReferencePointFrequency`: The frequency for the specific `McuClockReferencePoint`.

If watchdog timer is configured as MCWDT and there is no `McuClock` that includes `MCU_CLOCK_LF*` in `McuClockReferencePoint`, an error will occur in configuration phase.

If watchdog timer is configured as WDT and there is no `McuClock` that includes `MCU_CLOCK_ ILO0*` in `McuClockReferencePoint`, an error will occur in configuration phase.

See the *MCU user guide* for more information.

4.4 WDG settings fast configuration list

`WdgSettingsFastList` is the array of the following hardware depending settings of WDG driver's "fast" mode:

Note: Number of configured timers should be consistent with the number of used cores.

- `WdgFastTimerConfigRef` is the reference to the timer configuration for the watchdog driver's "fast" mode.

Note: Only one timer could be selected for one core. If both WDT and MCWDT are configured for the core, only MCWDT can be selected.

- `WdgFastTimeoutValue` represents trigger timeout value in "fast" mode. The range is 1-65535 ms.

Note: This value must be less than or equal to `WdgMaxTimeout`. Otherwise, an error would occur in configuration phase.

In case MCWDT is configured, `WdgMaxTimeout` is usually limited to a value lower than 65535 (see 4.1 General configuration).

- `WdgFastWarnLimitPercent` specifies the percentage of warning limit compared to trigger timeout value in "fast" mode. The range is 1-99%.
- `WdgFastLowerLimitPercent` specifies the percentage of the lower limit compared to trigger timeout value in "fast" mode. The range is 0-98%.
- `WdgFastMaxWaitTime` represents the waiting watchdog timer status change in "fast" mode.

In case WDT is configured, watchdog timer must be disabled before setting of hardware register. It takes up to three cycles of ILO (about 91.5 µs). After setting of hardware register, watchdog timer must be enabled. It also takes up to three cycles of ILO.

In case MCWDT is configured, watchdog timer must be disabled before setting of hardware register in initialization phase. It takes up to two cycles of LF (the duration is decided by the source clock of LF). After setting of hardware register, watchdog timer must be enabled. It also takes up to two cycles of LF.

WDG driver must wait those durations. This parameter is used for error detection when hardware changing does not become possible or does not take effect within designated time. So, it is recommended to set a higher value for this parameter, around 250 µs. Range is 1-65535 µs.

- `WdgFastMcuClockRef` is the reference to the MCU clock configuration, which is used to calculate the loop count of 1 µs.

Note: MCU clock reference will only support `McuClock` that includes `MCU_CLOCK_ROOT*` in `McuClockReferencePoint`.

- `McuClock`: Clock reference point.
- `McuClockReferencePointFrequency`: The frequency for the specific `McuClockReferencePoint`.

If there is no `McuClock` that includes `MCU_CLOCK_ROOT*` in `McuClockReferencePoint`, an error will occur in configuration phase.

See the *MCU user guide* for more information.

4.5 WDG settings slow configuration list

`WdgSettingsSlowList` is the array of the following hardware depending settings of WDG driver's "slow" mode:

Note: Number of configured timers should be consistent with the number of used cores.

- `WdgSlowTimerConfigRef` is reference to the timer configuration for the watchdog driver's "slow" mode.

Note: Only one timer could be selected for one core. If both WDT and MCWDT are configured for the core, only MCWDT can be selected.

- `WdgSlowTimeoutValue` represents trigger timeout value in "slow" mode. The range is 1-65535 ms.

Note: This value must be less than or equal to `WdgMaxTimeout`. Otherwise, an error would occur in configuration phase.
In case MCWDT is configured, `WdgMaxTimeout` is usually limited to a value lower than 65535 (see 4.1 General configuration).

- `WdgSlowWarnLimitPercent` is used to specify the percentage of warning limit compared to trigger timeout value in "slow" mode. The range is 1-99%.
- `WdgSlowLowerLimitPercent` is used to specify the percentage of lower limit compared to the trigger timeout value in "slow" mode. The range is 0-98%.
- `WdgSlowMaxWaitTime` represents the waiting watchdog timer status change in "slow" mode.

  In case WDT is configured, watchdog timer must be disabled before setting of hardware register. It takes up to three cycles of ILO (about 91.5 µs). After setting of hardware register, watchdog timer must be enabled. It also takes up to three cycles of ILO.

  In case MCWDT is configured, watchdog timer must be disabled before setting of hardware register in initialization phase. It takes up to two cycles of LF (the duration is decided by the source clock of LF). After setting of hardware register, watchdog timer must be enabled. It also takes up to two cycles of LF.

  WDG driver must wait those durations. This parameter is used for error detection when hardware changing does not become possible or does not take effect within designated time. So, it is recommended to set a higher value for this parameter, around 250 µs. Range is 1-65535 µs.

- `WdgSlowMcuClockRef` is reference to the MCU clock configuration, which is used to calculate loop count of 1 µs.

Note: MCU clock reference will only support `McuClock` that includes `MCU_CLOCK_ROOT*` in `McuClockReferencePoint`.

- `McuClock`: Clock reference point.
- `McuClockReferencePointFrequency`: The frequency for the specific `McuClockReferencePoint`.

If there is no `McuClock` that includes `MCU_CLOCK_ROOT*` in `McuClockReferencePoint`, an error will occur in configuration phase.

See the *MCU user guide* for further information.

4.6 WDG settings off configuration list

`WdgSettingsOffList` is the array of the following hardware depending settings of WDG driver's "off" mode:

Note: *Number of configured timers should be consistent with the number of used cores.*

- `WdgOffTimerConfigRef` is reference to the timer configuration for the watchdog driver's "off" mode.

Note: *Only one timer could be selected for one core. If both WDT and MCWDT are configured for the core, only MCWDT can be selected.*

- `WdgOffTimeoutValue` is not used.
- `WdgOffWarnLimitPercent` is not used.
- `WdgOffLowerLimitPercent` is not used.
- `WdgOffMaxWaitTime` represents the waiting watchdog timer status change in OFF mode. Watchdog timer is disabled.

  In case WDT is configured, it takes up to three cycles of ILO (about 91.5 µs).

  In case MCWDT is configured, it takes up to two cycles of LF (the duration is decided by the source clock of LF).

  So, it is recommended to set a higher value for this parameter, around 125 µs. Range is 1-65535 µs.
- `WdgOffMcuClockRef` is reference to the MCU clock configuration, which is used to calculate loop count of 1 µs.

Note: *MCU clock reference will only support `McuClock` that includes `MCU_CLOCK_ROOT*` in `McuClockReferencePoint`.*

- `McuClock`: Clock reference point.
- `McuClockReferencePointFrequency`: The frequency for the specific `McuClockReferencePoint`.

If there is no `McuClock` that includes `MCU_CLOCK_ROOT*` in `McuClockReferencePoint`, an error will occur in configuration phase.

See the *MCU user guide* for further information.

4.7 WDG DemEventParameter reference

- `WDG_E_DISABLE_REJECTED` refers to the configured DEM event that reports "Initialization or mode switch failed because it would disable the WDG while disabling is not allowed".

Note: *This parameter is effective when `WdgDemEventDisableRejected` is TRUE. `WDG_E_DISABLE_REJECTED` should have valid reference; otherwise an error would occur in configuration phase.*

- `WDG_E_MODE_FAILED` refers to the configured DEM event that reports "Setting a WDG mode failed (during initialization or mode switch)".

Note: *This parameter is effective when `WdgDemEventModeFailed` is TRUE. `WDG_E_MODE_FAILED` should have valid reference; otherwise an error would occur in configuration phase.*

- `WDG_E_HW_TIMEOUT` refers to the configured DEM event that reports "Hardware timeout (during initialization or mode switch or setting trigger condition)".
  - "Hardware timeout" means that the hardware status was not changed in the period specified by `WdgFastMaxWaitTime`, `WdgSlowMaxWaitTime`, or `WdgOffMaxWaitTime`.

Note: *This parameter is effective only when `WdgDemEventHwTimeout` is TRUE. `WDG_E_HW_TIMEOUT` should have valid reference; otherwise an error would occur in configuration phase.*

- `WDG_E_WDG_STOPPED` refers to the configured DEM event that reports "WDG stopped (during setting trigger condition in off mode)".

Note: *This parameter is effective when `WdgDemEventWdgStopped` is TRUE. `WDG_E_WDG_STOPPED` should have valid reference; otherwise an error would occur in configuration phase.*

4.8 WdgMulticore

- `WdgCoreConsistencyCheckEnable` enables core consistency check during runtime. If enabled, WDG ISR handler checks if the watchdog timer related to the interrupt reason is allowed on the current core.

Note: Development error detect will be enabled in the WDG driver to enable this parameter.

- `WdgGetCoreIdFunction` specifies the API to be called to get the core ID.

Note: `WdgGetCoreIdFunction` must be a valid C function name. `Mcu_GetCoreID` and `GetCoreID` can optionally be set to the configuration parameter `WdgGetCoreIdFunction`.

- `WdgMasterCoreReference` references to the master core configuration.

Note: WdgMasterCoreReference must have the target's WdgCoreConfiguration setting.

- `WdgCoreConfigurationId` is the core assignment. Range is 0 to 254.

Note: This value is assigned to a symbolic name. Use only the symbolic core ID names defined in Wdg_66_IA_Cfg.h.

4.9 WdgCoreConfiguration

- `WdgCoreConfigurationId` is a zero-based, consecutive integer value. This is used as a logical core ID.

Note: `WdgCoreConfigurationId` must be unique across `WdgCoreConfiguration`.

- `WdgCoreId` is WDG core ID. This ID is assigned to WDG timers. This ID is returned from the configured `WdgGetCoreIdFunction` execution to identify the executing core.

Note: `WdgCoreId` must be unique across `WdgCoreConfiguration`.

4.10 WDG external configuration

This container is intended for using external watchdog timer. But this property is not used.

4.11 WdgPublishedInformation

`WdgTriggerMode` represents watchdog trigger mode (`WDG_TOGGLE`, `WDG_WINDOW`, or `WDG_BOTH`). The value is fixed to `WDG_BOTH`.

5 Functional description

5.1 Inclusion

The *Wdg_66_IA.h* file includes all necessary external identifiers. Therefore, the application only needs to include *Wdg_66_IA.h* to make all API functions and data types available.

5.2 Initialization

`Wdg_66_IA_Init` function initializes the WDG driver and sets the default WDG mode. Since it is possible to set more than one configuration, this function can be called with different configuration sets.

```
Wdg_66_IA_Init(&Wdg_66_IA_Config[1]);
```

Note: *Make sure that initialization has been performed before any other WDG API function is called on each core.*
`Wdg_66_IA_Init()` must be called on the master core before any cores are initialized. If `Wdg_66_IA_Init()` is called on the satellite core, the master core must be already initialized.
The same configuration set must be specified on all cores during initialization.

A repeated call of the `Wdg_66_IA_SetTriggerCondition(1000)` API function is required to prevent the WDG from triggering a reset.

Note: *The value of timeout (milliseconds) should not be higher than the value of `WdgMaxTimeout`.*

5.3 Reconfiguration during runtime

Reconfiguration of the WDG driver is not possible at runtime. You can reinitialize with a different configuration set, but you should ensure all timers are stopped before you switch the configuration set.

5.4 API parameter checking

The driver's services perform regular error checks.

When an error occurs, the error hook routine (configured via `WdgErrorCalloutFunction`) is called and the error code, service ID, module ID, and instance ID are passed as parameters.

If default error detection is enabled, all development errors are also reported to the DET, a central error hook function within the AUTOSAR environment. The checking itself cannot be deactivated for safety reasons.

The following development error checks are performed by the services of the WDG driver:

5.4.1 Wdg_66_IA_Init()

- `Wdg_66_IA_Init()` checks if the configuration is within valid range on master core; otherwise calls DET with `WDG_66_IA_E_INIT_FAILED`.
- `Wdg_66_IA_Init()` checks if the ConfigPtr parameter is different from the configuration pointer which is already initialized by master core when called on satellite cores; otherwise calls DET with `WDG_66_IA_E_DIFFERENT_CONFIG`.
- `Wdg_66_IA_Init()` checks if the default mode is within valid range; otherwise calls DET with `WDG_66_IA_E_PARAM_CONFIG`.

- `Wdg_66_IA_Init()` verifies that the supported modes are `WDG_SLOW_MODE`, `WDG_OFF_MODE`, and `WDG_FAST_MODE`. If the mode is not allowed, the DEM message `WDG_E_MODE_FAILED` will be reported.

If the default mode is `WDGIF_OFF_MODE` and disabling is not allowed, the DEM message `WDG_E_DISABLE_REJECTED` will be reported.

Note: *WDG disables and enables watchdog timer to initialize registers according to configuration parameters:*

- Disabling wait time and applied modes: Before change register settings, it is necessary to write ENABLE bit of CTL register to disable watchdog timer and check the status until ENABLED bit of CTL register is disabled.

  Applied to off, slow, and fast modes.

- Enabling wait time and applied modes: It is also necessary to write ENABLE bit of CTL register to enable watchdog timer and check the status until ENABLED bit of CTL register is enabled.

  Applied to slow and fast modes

- Time to take effect

  Each of the above wait time is different between WDT and MCWDT.

  - WDT
    Takes up to three cycles of ILO (about 91.5 µs).
    When the default mode is off, total wait time will be up to about 91.5 µs.
    When the default mode is slow or fast, total wait time will be up to about 183.0 µs.
  - MCWDT
    Takes up to two cycles of LF (source clock of LF is configurable).
    When the default mode is off, total wait time will be up to two cycles of LF.
    When the default mode is slow or fast, total wait time will be up to four cycles of LF.

Note: *When WDT is configured and watchdog timer is disabled, watchdog timer continues counting until `ENABLED` bit of `CTL` register to be disabled.*
When MCWDT is configured and watchdog timer is serviced, watchdog timer continues counting until CTR0_SERVICE bit of SERVICE register to be effective.
For example, even though an application calls `Wdg_66_IA_SetTriggerCondition()` before the watchdog timer expires, watchdog reset might occur because of the time lag of watchdog hardware.

  - WDT
    The time lag is three cycles of ILO, which is the source clock of the watchdog timer. Duration of exclusive area: The registers are set within the exclusive area which is possibly up to about 183.0 µs. Exclusive area is valid when only WDT is configured.
    Calculation of timeout value: The timeout value is exchanged to a watchdog count (round down to the nearest decimal). For example, when timeout value is 1 ms (0.001 s), the count will be 32 which means 0.9766 ms).
  - MCWDT
    The time lag is three cycles of LF, which is the source clock of the watchdog timer. Exclusive area is not used.
    Calculation of timeout value: The timeout value is exchanged to a watchdog count (round down to the

nearest decimal). For example, when timeout value is 1 ms (0.001 s), the count will be 32 which means 0.9766 ms).

5.4.2 Wdg_66_IA_SetMode()

If the new mode is `WDGIF_OFF_MODE` and disabling is not allowed, the DET error `WDG_66_IA_E_PARAM_MODE` will be reported and the DEM messages `WDG_E_DISABLE_REJECTED` and `WDG_E_MODE_FAILED` will be reported.

If the new mode is not within the valid range, the DET error `WDG_66_IA_E_PARAM_MODE` will be reported.

If the new mode is not listed in the supported modes defined in the WDG driver, the DET error `WDG_66_IA_E_PARAM_MODE` will be reported and the DEM message `WDG_E_MODE_FAILED` will be reported.

Note: *`WDG_66_IA_FAST_MODE`, `WDG_66_IA_SLOW_MODE`, and `WDG_66_IA_OFF_MODE` are in the list.*

If the new mode is same as current mode, `Wdg_66_IA_SetMode()` returns `E_OK` without any operations.

Note: *If the parameter "mode" is not changed from the current value, this API returns quickly without any operations.*

The behavior when the parameter "mode" is changed is different between WDT and MCWDT.

- WDT

  WDG must disable watchdog timer to set registers and enable it to restart according to the parameter.

- MCWDT

  WDG writes the SERVICE register and sets other registers without disabling and enabling MCWDT.

  When the SRSS version is two and the lower limit after the change is smaller than the current watchdog timer counter, the WDG must wait for SERVICE register's status before changing other registers to avoid a reset. After that, the watchdog timer counter will restart from zero.

  It takes up to three cycles of LF (the duration is decided by the source clock of LF). For details of the SRSS version, see Hardware documentation.

Same timing restrictions are applied as described for `Wdg_66_IA_Init()`. See 5.4.1 Wdg_66_IA_Init().

5.4.3 Wdg_66_IA_SetTriggerCondition()

The `Wdg_66_IA_SetTriggerCondition()` function checks whether the timeout that passed is equal to or less than the maximum allowed timeout; if not, the function calls DET with `WDG_66_IA_E_PARAM_TIMEOUT`.

Note: *If the parameter "timeout" is not changed from the current value, this API will retrigger the watchdog timer through the SERVICE register.*

The SERVICE register of the WDT takes up to three cycles of the ILO (about 91.5 µs) to take effect.

(For example, if this API is called and the SERVICE bit of the SERVICE register is written when the remaining count before expiry is less than three ILO cycles at that time, the watchdog timer will continue to count three more cycles of the ILO; this will cause a reset in this case).

When the SERVICE register is written again before it takes effect, writing will be ignored.

For example, when the mode is WDG_66_IA_FAST_MODE and `WdgFastLowerLimitPercent` is configured, or when the mode is WDG_66_IA_SLOW_MODE and `WdgSlowLowerLimitPercent` is configured, if this API is called consecutively and the SERVICE bit of the SERVICE register is written, the second and later writings will be ignored. After the SERVICE register takes effect, if this API is called again before the lower limit is reached, the lower limit violation will be triggered.

About the SERVICE register of the WDT, an HW erratum is reported.

If this API writes the SERVICE bit of the SERVICE register and the system goes to DeepSleep or Hibernate mode within four cycles of the ILO, the next writing of the SERVICE bit of the SERVICE register after waking up within two cycles of ILO may be ignored. As a result of this behavior, the WDT will continue to count and cause an undesired interrupt or reset.

This erratum has effect only on CYT2Bx series. To determine if your device is affected, see Hardware documentation.

SERVICE register of MCWDT takes up to three cycles of LF (the duration is decided by the source clock of LF) to take effect.

(For example, if this API is called and write CTR0_SERVICE bit of SERVICE register when the remaining count before expiry is less than three at that time, watchdog timer will continue to count three cycles of LF more, so that it will cause a reset in this case).

When the SERVICE register is written again before it takes effect, writing will be ignored.

For example, when the mode is WDG_66_IA_FAST_MODE and `WdgFastLowerLimitPercent` is configured, or when the mode is WDG_66_IA_SLOW_MODE and `WdgSlowLowerLimitPercent` is configured, if this API is called consecutively and the CTR0_SERVICE bit of the SERVICE register is written, the second and later writings will be ignored. After the SERVICE register takes effect, if this API is called again before the lower limit is reached, the lower limit violation will be triggered.

If the “timeout” parameter is changed, the behavior is different between WDT and MCWDT.

- WDT

  WDG must disable the watchdog timer to set the registers and enable it to restart according to the parameter.

- MCWDT

  WDG writes to the SERVICE register and sets other registers without disabling and enabling MCWDT.

  Restrictions as same as that of the SERVICE register are applied as described in `Wdg_66_IA_SetMode()`. See 5.4.2 Wdg_66_IA_SetMode().

Same timing restrictions are applied as described for `Wdg_66_IA_Init()`. See 5.4.1 Wdg_66_IA_Init().

5.4.4 Wdg_66_IA_GetVersionInfo()

`Wdg_66_IA_GetVersionInfo()` reports the DET `WDG_66_IA_E_PARAM_POINTER`, if the parameter `versioninfo` parameter is a NULL pointer.

5.5 Runtime checks

If Wdg_66_IA_Init() is called on the master core, the API checks whether the satellite cores are already initialized. If the satellite cores are initialized, Wdg_66_IA_Init() will report the WDG_66_IA_E_ALREADY_INITIALIZED error.

If Wdg_66_IA_Init() called on the satellite cores, the API checks that whether the master core is already initialized. If the master core is not initialized, Wdg_66_IA_Init() will report the r WDG_66_IA_E_INIT_FAILED error.

Wdg_66_IA_Init(), Wdg_66_IA_SetMode(), and Wdg_66_IA_SetTriggerCondition() APIs check whether the running core ID is valid, otherwise will report the WDG_66_IA_INVALID_CORE error.

The Wdg_66_IA_SetMode() and Wdg_66_IA_SetTriggerCondition() APIs check whether the WDG's state is WDG_IDLE and whether the driver is already initialized properly. Otherwise the error callout handler and DET will be called with the WDG_66_IA_E_DRIVER_STATE parameter.

Wdg_66_IA_SetTriggerCondition() checks if current mode is WDG_OFF_MODE, then the DEM message WDG_E_WDG_STOPPED will be reported.

In case WdgCoreConsistencyCheckEnable is enabled, ISR handler checks if the watchdog timer related to the interrupt reason is allowed on the current core. If not allowed, error WDG_66_IA_E_INVALID_CORE will be reported.

5.6 Reentrancy

All functions except Wdg_66_IA_GetVersionInfo are not reentrant.

5.7 Debugging support

The WDG driver does not support debugging.

5.8 Functions available without core dependency

The following function is available on any core without any restriction:

- Wdg_66_IA_GetVersionInfo()

6 Hardware resources

6.1 Interrupts

If the warning interrupt is enabled (see parameter `WdgEnableWarningIrq`), one of the following interrupt handlers must be configured in OS to be called on the corresponding interrupt. The ISR should be allocated to the same core as the allocated resource. The ISR must be declared in the AUTOSAR OS as Category 1 Interrupt or Category 2 Interrupt.

```
ISR(Wdg_66_IA_WarnIntWDT_Cat2)
ISR_NATIVE(Wdg_66_IA_WarnIntWDT_Cat1)
```

```
ISR(Wdg_66_IA_WarnIntMCWDT[n]_Cat2)
ISR_NATIVE(Wdg_66_IA_WarnIntMCWDT[n]_Cat1)
```

Note: *The interrupt service routines' name suffixes must match the configured ISR category.*
[n]: the number of specific MCWDT channel.

Note: *On the Arm® Cortex®-M4 CPU, priority inversion of interrupts may occur under specific timing conditions in the integrated system with TRAVEO™ T2G MCAL. For more details, see the following errata notice.*

Arm® Cortex®-M4 Software Developers Errata Notice - 838869:
"Store immediate overlapping exception return operation might vector to incorrect interrupt"

If the user application cannot tolerate the priority inversion, a DSB instruction should be added at the end of the interrupt function to avoid the priority inversion.

TRAVEO™ T2G MCAL interrupts are handled by an ISR wrapper (handler) in the integrated system. Thus, if necessary, the DSB instruction should be added just before the end of the handler by the integrator.

7 Appendix A – API reference

7.1 Data types

7.1.1 Wdg_66_IA_ConfigType

Type

```
typedef struct
{
  const Wdg_66_IA_SettingCommonType * SettingCommonPtr;
  const Wdg_66_IA_SettingType *       SettingWdgPtr;
  const uint8                         CoreCount;
} Wdg_66_IA_ConfigType;
```

Description

`Wdg_66_IA_ConfigType` defines a structure which holds the WDG driver's configuration set.

7.1.2 WdgIf_ModeType

Type

```
typedef enum
```

Description

This type is derived from the WDG interface. It represents the mode types used for switching the WDG's mode.

7.2 Constants

7.2.1 Error codes

The service might return the f error codes, listed in Table 3, if default error detection is enabled:

Table 3 Error codes

Name	Value	Description
WDG_66_IA_E_DRIVER_STATE	0x10	WDG is already busy when triggering or mode switching will be performed.
WDG_66_IA_E_PARAM_MODE	0x11	Mode is not a valid parameter.
WDG_66_IA_E_PARAM_CONFIG	0x12	Configuration set is not OK.
WDG_66_IA_E_PARAM_TIMEOUT	0x13	Function `SetTriggerCondition` is called with an invalid timeout (bigger than maximum allowed).
WDG_66_IA_E_PARAM_POINTER	0x14	Function `GetVersionInfo` is called with NULL pointer.
WDG_66_IA_E_INIT_FAILED	0x15	`ConfigPtr` is not a valid pointer to configuration set when WDG initializing.
WDG_66_IA_E_EXTENDED_MODE_FAILED	0x20	Mode switching failed due to some reasons (e.g. hardware limitation). This error id is used to call error callout handler.

Name	Value	Description
WDG_66_IA_E_EXTENDED_DISABLE_REJECTED	0x21	The WDG is trying to disable the watchdog although it is not allowed. This error id is used to call error callout handler.
WDG_66_IA_E_EXTENDED_HW_TIMEOUT	0x22	The WDG hardware status change wait timeout. This error id is used to call error callout handler.
WDG_66_IA_E_EXTENDED_WDG_STOPPED	0x23	The WDG is trying to set trigger condition during the watchdog is stopped. This error id is used to call error callout handler.
WDG_66_IA_E_INVALID_CORE	0x24	API is called from the invalid core or ISR occurs on the invalid core.
WDG_66_IA_E_DIFFERENT_CONFIG	0x25	Intended config initialization of this core does not match with the initialized config of other cores.
WDG_66_IA_E_ALREADY_INITIALIZED	0x26	Wdg_Init is called by the master core while other cores are already initialized.

The following DEM messages can be enabled individually:

WDG_E_MODE_FAILED	defined by DEM	Mode switching failed due to lack of hardware support for this mode (DEM error).
WDG_E_DISABLE_REJECTED	defined by DEM	WDG is not allowed to be disabled (DEM error).
WDG_E_HW_TIMEOUT	defined by DEM	Timeout period expired while changing hardware status (DEM error).
WDG_E_WDG_STOPPED	defined by DEM	Trigger condition is not allowed to be set during the watchdog is stopped (DEM error).

7.2.2 Version information

The version information, listed in Table 4, is published in the driver's header file.

Table 4 Version information

Name	Value	Description
WDG_SW_MAJOR_VERSION	See release notes	Major version number
WDG_SW_MINOR_VERSION	See release notes	Minor version number
WDG_SW_PATCH_VERSION	See release notes	Patch version number

7.2.3 Module information

Table 5 Module information

Name	Value	Description
WDG_MODULE_ID	102	Module ID
WDG_VENDOR_ID	66	Vendor ID

7.2.4 API service IDs

The API service IDs, listed in Table 6, are published in the driver's header file:

Table 6 API service IDs

Name	Value	Description
`WDG_66_IA_API_INIT`	0x00	Service ID of `Wdg_66_IA_Init`
`WDG_66_IA_API_SETMODE`	0x01	Service ID of `Wdg_66_IA_SetMode`
`WDG_66_IA_API_SET_TRIGGER_CONDITION`	0x03	Service ID of `Wdg_66_IA_SetTriggerCondition`
`WDG_66_IA_API_GET_VERSION_INFO`	0x04	Service ID of `Wdg_66_IA_GetVersionInfo`
`WDG_66_IA_API_WARNINT`	0xFF	Service ID of `Wdg_66_IA_WarningInterrupt_Arch`

7.2.5 Invalid core ID value

Table 7 Invalid core ID

Name	Value	Description
`WDG_66_IA_INVALID_CORE`	0xFF	Invalid core ID

7.3 Functions

7.3.1 Wdg_66_IA_Init

Syntax

```
void Wdg_66_IA_Init(
    const Wdg_66_IA_ConfigType* ConfigPtr
)
```

Service ID

0x00

Parameters (in)

- `ConfigPtr` - Pointer to WDG driver configuration set.

Parameters (out)

None

Return value

None

DET errors

- `WDG_66_IA_E_INVALID_CORE` - API is called from the invalid core.
- `WDG_66_IA_E_INIT_FAILED` - The pointer to the configuration set for initialization is invalid.
- `WDG_66_IA_E_PARAM_CONFIG` - The default mode is invalid or the WDG failed to initialize.
- `WDG_66_IA_E_ALREADY_INITIALIZED` - API is called by the master core while other cores are already initialized.

- `WDG_66_IA_E_DIFFERENT_CONFIG` - Intended config initialization of this core does not match with the initialized config of other cores.

DEM errors

- `WDG_E_DISABLE_REJECTED` - WDG cannot be turned OFF when `DisableAllowed` is `FALSE`.
- `WDG_E_MODE_FAILED` - The DefaultMode is not supported by hardware.
- `WDG_E_HW_TIMEOUT` - WDG initialization failed due to timeout of changing hardware status.

Description

This function initializes the WDG driver.

7.3.2 Wdg_66_IA_SetMode

Syntax

```
Std_ReturnType Wdg_66_IA_SetMode(
    WdgIf_ModeType Mode
)
```

Service ID

0x01

Parameters (in)

- `Mode` - Mode the WDG should be switched to.

Parameters (out)

None

Return value

`E_OK` or `E_NOT_OK`

DET errors

- `WDG_66_IA_E_INVALID_CORE` - API is called from the invalid core.
- `WDG_66_IA_E_DRIVER_STATE` - WDG is not initialized yet or currently not in idle state.
- `WDG_66_IA_E_PARAM_MODE` - The parameter mode is not supported.

DEM errors

- `WDG_E_MODE_FAILED` - Switching of mode failed due to lack of hardware support for this mode.
- `WDG_E_DISABLE_REJECTED` - Switching to off mode is not allowed or WDG is currently not in idle state.
- `WDG_E_HW_TIMEOUT` - Switching of mode failed due to timeout of changing hardware status.

Description

This function switches the mode of the WDG between the following three modes:

- `WDGIF_OFF_MODE`
- `WDGIF_SLOW_MODE`
- `WDGIF_FAST_MODE`

7.3.3 Wdg_66_IA_SetTriggerCondition

Syntax

```
void Wdg_66_IA_SetTriggerCondition(
    uint16 timeout
)
```

Service ID

0x03

Parameters (in)

- `timeout` - The new trigger condition validity period in milliseconds.

Parameters (out)

None

Return value

None

DET errors

- `WDG_66_IA_E_INVALID_CORE` - API is called from the invalid core.
- `WDG_66_IA_E_DRIVER_STATE` - WDG is not initialized yet or currently not in idle state.
- `WDG_66_IA_E_PARAM_TIMEOUT` - Input timeout is greater than the maximum allowed timeout.

DEM errors

- `WDG_E_HW_TIMEOUT` - Switching of mode failed due to timeout of changing hardware status.
- `WDG_E_WDG_STOPPED` - Setting of trigger condition during the watchdog is stopped.

Description

This function triggers watchdog timer with parameter timeout. If the value is 0, it triggers a watchdog reset, immediately.

7.3.4 Wdg_66_IA_GetVersionInfo

Syntax

```
void Wdg_66_IA_GetVersionInfo(
    Std_VersionInfoType* versioninfo
)
```

Service ID

0x04

Parameters (in)

None

Parameters (out)

- `versioninfo` - Version information of the WDG driver is stored at the previously given address.

Return value

None

DET errors

- `WDG_66_IA_E_PARAM_POINTER` - Input version information pointer is NULL.

DEM errors

None

Description

Returns the version information of the module.

7.4 Required callback functions

7.4.1 DET

If default error detection is enabled, the WDG driver uses the following callback function provided by DET. If you do not use DET, you must implement this function within your application.

Det_ReportError

Syntax

```
Std_ReturnType Det_ReportError
(
    uint16 ModuleId,
    uint8 InstanceId,
    uint8 ApiId,
    uint8 ErrorId
)
```

Reentrancy

Reentrant

Parameters (in)

- `ModuleId` - Module ID of calling module.
- `InstanceId` - `WdgCoreConfigurationId` of the core that calls this function or WDG_66_IA_INVALID_CORE.
- `ApiId` - ID of the API service that calls this function.
- `ErrorId` - ID of the detected development error.

Return value

Returns always `E_OK` (is required for services).

Description

Service for reporting development errors.

7.4.2 DEM

If DEM notifications are enabled, the WDG driver uses the following callback function that is provided by DEM. If you do not use DEM, you must implement this function within your application.

Dem_ReportErrorStatus

Syntax

```
void Dem_ReportErrorStatus
(
    Dem_EventIdType EventId,
    Dem_EventStatusType EventStatus
)
```

Reentrancy

Reentrant

Parameters (in)

- `EventId` - Identification of an event by assigned event ID.
- `EventStatus` - Monitor test result of given event.

Return value

None

Description

Service for reporting diagnostic events.

7.4.3 Callout functions

7.4.3.1 Error callout API

The AUTOSAR WDG module requires an error callout handler. Each error is reported to this handler; error checking cannot be switched OFF. The name of the function to be called can be configured by parameter *WdgErrorCalloutFunction*.

Syntax

```
void Error_Handler_Name
(
    uint16 ModuleId,
    uint8 InstanceId,
    uint8 ApiId,
    uint8 ErrorId
)
```

Reentrancy

Reentrant

Parameters (in)

- `ModuleId` - Module ID of calling module.
- `InstanceId` - `WdgCoreConfigurationId` of the core that calls this function or WDG_66_IA_INVALID_CORE.

- `ApiId` - ID of the API service that calls this function.
- `ErrorId` - ID of the detected error.

Return value

None

Description

Service for reporting errors.

7.4.3.2 Get core ID API

The AUTOSAR WDG module requires a function to get the valid core ID. This function is being used to determine the core from which the code is being executed. The name of the function to be called can be configured by the `WdgGetCoreIdFunction` parameter.

Syntax

```
uint8 GetCoreID_Function_Name (void)
```

Reentrancy

Reentrant

Parameters (in)

None

Return value

- `CoreId` - ID of the current core.

Description

Service for getting the valid core ID.

Note: *This function returns the core ID configured in* `WdgMulticore/WdgCoreConfiguration/WdgCoreId`. *For example: Two cores are configured in the* `WdgCoreConfiguration`.

Executing core	**WdgCoreConfigurationId**	**WdgCoreId**
CM7_0	0	15
CM7_1	1	16

When the function is called from the CM7_0 core, it returns 15, and when called from the CM7_1 core, it returns 16.

8 Appendix B – Access register table

8.1 SRSS (MCWDT)

Table 8 SRSS access register table of MCWDT

Register	Bit No.	Access size	Value	Description	Timing	Monitoring mask	Monitoring value

`CTL`	31:0	Word (32 bits)	Depends on configuration value or API	MCWDT control register of subcounter 0	`Wdg_66_IA_Init`
					`Wdg_66_IA_SetMode`
					`Wdg_66_IA_SetTriggerCondition`
						0x80000001	0x80000001
							(After MCWDT is set to slow/fast mode by calling `Wdg_66_IA_Init` / `Wdg_66_IA_SetMode`)
							
							0x00000000
							(After MCWDT is set to off mode by calling `Wdg_66_IA_Init` / `Wdg_66_IA_SetMode`)

`LOWER_LIMIT`	15:0	Word (32 bits)	Depends on configuration value or API	MCWDT lower limit register of subcounter 0	`Wdg_66_IA_Init`
					`Wdg_66_IA_SetMode`
					`Wdg_66_IA_SetTriggerCondition`
						0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)

`UPPER_LIMIT`	15:0	Word (32 bits)	Depends on configuration value or API	MCWDT upper limit register of subcounter 0	`Wdg_66_IA_Init`
					`Wdg_66_IA_SetMode`
					`Wdg_66_IA_SetTriggerCondition`
						0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)

`WARN_LIMIT`	15:0	Word (32 bits)	Depends on configuration value or API.	MCWDT Warn limit register of subcounter 0	`Wdg_66_IA_Init`
					`Wdg_66_IA_SetMode`
					`Wdg_66_IA_SetTriggerCondition`
						0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)

`CONFIG`	31:0	Word (32 bits)	Depends on configuration value or API.	MCWDT configuration register of subcounter 0	`Wdg_66_IA_Init`	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)

Register	Bit No.	Access size	Value	Description	Timing	Monitoring mask	Monitoring value
CNT	15:0	Word (32 bits)	-	MCWDT count register of subcounter 0	Do not use.	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
CPU_SELECT	31:0	Word (32 bits)	Depends on configuration value or API.	MCWDT CPU selection register	Wdg_66_IA_Init	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
LOCK	31:0	Word (32 bits)	0x00000003	MCWDT lock register	Wdg_66_IA_Init Wdg_66_IA_SetMode Wdg_66_IA_SetTriggerCondition	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
SERVICE	31:0	Word (32 bits)	0x00000000 0x00000001	MCWDT service register	Wdg_66_IA_SetMode Wdg_66_IA_SetTriggerCondition	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
INTR	31:0	Word (32 bits)	0x00000000 0x00000001	MCWDT interrupt register	Wdg_66_IA_WarnIntMCWDT[n]_Cat1 Wdg_66_IA_WarnIntMCWDT[n]_Cat2 ([n]: the number of specific MCWDT channel)	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
INTR_SET	31:0	Word (32 bits)	-	MCWDT interrupt set register	Do not use.	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
INTR_MASK	31:0	Word (32 bits)	Depends on configuration value or API.	MCWDT interrupt mask register	Wdg_66_IA_Init Wdg_66_IA_SetMode Wdg_66_IA_SetTriggerCondition	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
INTR_MASKED	31:0	Word (32 bits)	-	MCWDT interrupt masked register	Do not use.	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)

8.2 SRSS (WDT)

Table 9 SRSS access register table of WDT

Register	Bit No.	Access size	Value	Description	Timing	Monitoring mask	Monitoring value

`CTL`	31:0	Word (32 bits)	Depends on configuration value or API	WDT control register	`Wdg_66_IA_Init`
					`Wdg_66_IA_SetMode`
					`Wdg_66_IA_SetTriggerCondition`
						0x80000001	0x80000001
							(After WDT is set to slow/fast mode by calling `Wdg_66_IA_Init` / `Wdg_66_IA_SetMode` or after `Wdg_66_IA_SetTriggerCondition` is called in slow/fast mode)
							
							0x00000000
							(After WDT is set to off mode by calling `Wdg_66_IA_Init` / `Wdg_66_IA_SetMode`)

`LOWER_LIMIT`	31:0	Word (32 bits)	Depends on configuration value or API	WDT lower limit register	`Wdg_66_IA_Init`
					`Wdg_66_IA_SetMode`
					`Wdg_66_IA_SetTriggerCondition`
						0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)

`UPPER_LIMIT`	31:0	Word (32 bits)	Depends on configuration value or API	WDT upper limit register	`Wdg_66_IA_Init`
					`Wdg_66_IA_SetMode`
					`Wdg_66_IA_SetTriggerCondition`
						0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)

`WARN_LIMIT`	31:0	Word (32 bits)	Depends on configuration value or API.	WDT Warn limit register	`Wdg_66_IA_Init`
					`Wdg_66_IA_SetMode`
					`Wdg_66_IA_SetTriggerCondition`
						0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)

`CONFIG`	31:0	Word (32 bits)	Depends on configuration value or API.	WDT configuration register	`Wdg_66_IA_Init`	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)

Register	Bit No.	Access size	Value	Description	Timing	Monitoring mask	Monitoring value
CNT	31:0	Word (32 bits)	-	WDT count register	Do not use.	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
LOCK	31:0	Word (32 bits)	0x00000003	WDT lock register	Wdg_66_IA_Init Wdg_66_IA_SetMode Wdg_66_IA_SetTriggerCondition	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
SERVICE	31:0	Word (32 bits)	0x00000000 0x00000001	WDT service register	Wdg_66_IA_SetTriggerCondition	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
INTR	31:0	Word (32 bits)	0x00000000 0x00000001	WDT interrupt register	Wdg_66_IA_WarnIntWDT_Cat1 Wdg_66_IA_WarnIntWDT_Cat2	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
INTR_SET	31:0	Word (32 bits)	-	WDT interrupt set register	Do not use.	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
INTR_MASK	31:0	Word (32 bits)	Depends on configuration value or API.	WDT interrupt mask register	Wdg_66_IA_Init Wdg_66_IA_SetMode Wdg_66_IA_SetTriggerCondition	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)
INTR_MASKED	31:0	Word (32 bits)	-	WDT interrupt masked register	Do not use.	0x00000000 (monitoring is not needed.)	0x00000000 (monitoring is not needed.)

Revision history

Document revision	Date	Description of changes
**	2020-08-11	Initial release
*A	2020-11-19	Deleted unused memory section from section "Memory Allocation Keyword". Changed description in section "Memory Allocation Keyword". MOVED TO INFINEON TEMPLATE.
*B	2021-05-18	Modified description regarding WDG_66_IA_E_PARAM_MODE in chapter 5.4.2.
*C	2021-08-19	Added a note in 6.1 Interrupts
*D	2021-12-21	Updated to the latest branding guidelines.
*E	2022-07-12	Added caution regarding WDT in chapter 5.4.3.
*F	2023-03-23	Added caution regarding SERVICE register in chapter 5.4.3. Added chapter 2.6.3. Updated the description in chapter 4.1.
*G	2023-06-06	Updated the description in chapter 2.6.1.
*H	2023-10-06	Corrected core identification keyword in section 2.6.
*I	2023-12-08	Web release. No content updates.
*J	2024-03-18	Corrected ASIL keyword in section 2.6.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2024-03-18
Published by

Infineon Technologies AG
81726 Munich, Germany

© 2024 Infineon Technologies AG.
All Rights Reserved.

Do you have a question about this document?
Email:
erratum@infineon.com

Document reference
002-30200 Rev. *J

Warnings
Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.