Document Title	Explanation of Interrupt Handling within AUTOSAR
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	307

Document Status	Final
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	4.3.1

Document Change History

Date	Release	Changed by	Change Description
2017-12-08	4.3.1	AUTOSAR Release Management	• Editorial changes
2016-11-30	4.3.0	AUTOSAR Release Management	• Editorial changes
2014-10-31	4.2.1	AUTOSAR Release Management	• Editorial changes
2013-03-15	4.1.1	AUTOSAR Administration	• Finalized for Release 4.1 •
2010-02-02	3.1.4	AUTOSAR Administration	• Legal disclaimer revised •
2008-08-13	3.1.1	AUTOSAR Administration	• Legal disclaimer revised •
2007-12-21	3.0.1	AUTOSAR Administration	• Initial Release

Disclaimer

Table of Contents

1 Introduction and purpose of document 4

2 Acronyms and abbreviations 5

3 Related documentation 6

3.1 Input documents 6
3.2 Related standards and norms 6

4 Summary of Interrupt Configuration 7

5 Overview of Interrupt Operation 9

5.1 Distinction between cat1 and cat2 interrupts 9

6 Steps in the operation of interrupts 11

6.1 Handling cat1 interrupts 11
6.1.1 Initial state 11
6.1.2 When the hardware requests an interrupt 11
6.2 Handling cat2 interrupts 13
6.2.1 Initial state 13
6.2.2 When the hardware requests an interrupt 13

7 Configuration of Interrupts 15

7.1 Device Driver configuration and code 15
7.1.1 Placement of Interrupt Handlers 16
7.2 OS configuration 16
7.3 BSW Scheduler configuration 17
7.3.1 TASK for main functions 17
7.3.2 Other TASKs in the stack 18
7.3.3 Critical sections 18
7.3.4 Summary 20

8 Recommendations for the use of cat1 interrupts 21

8.1.1 Communication between adjacent modules using cat1 interrupts 21
8.1.2 Trust 22

1 Introduction and purpose of document

This document captures the way that interrupts work and are configured in Autosar. The purpose of the document is to guide the specification work of the WPs that are specifying modules that, in some way interact with interrupts.

2 Acronyms and abbreviations

Acronym:	*Description:*
ISR	Interrupt Service Routine. Also used as a macro to declare in C a cat2 interrupt service routine.
RETI	Return from Interrupt
GCE	Generic Configuration Editor

Abbreviation:	*Description:*
Cat2	Category 2. Cat2 ISRs are supported by the OS and can make OS calls.
Cat1	Category 1. Cat1 interrupts are not supported by the OS and are only allowed to make a very small selection of OS calls to enable and disable all interrupts.

Terminology:	*Description:*
Interrupt Handler	In the case of a Cat2 interrupts, the ISR is synonymous with Interrupt Handler. In the case of Cat1 interrupt the Interrupt handler is the function called by the hardware interrupt vector. In both cases the Interrupt handler is the user code that is normally a part of the BSW module. So we consider the Interrupt Handler to be a user level piece of code. However, in the case of Cat2 interrupts are initially handled in the OS's interrupt handler before the user's interrupt handler is called.
Interrupt Logic	This is the MCU logic that controls all interrupts for all devices. This is normally controlled by the OS.
Device	A hardware I/O device that, for the purposes of this document, can also cause interrupts.
Device Interrupt Enable Bit	This is the bit/bits within one hardware device, that is controlled by the device driver, to enable/disable the interrupt source for that device only.
Interrupt Frame	An interrupt frame is the code which is generated by the compiler, or the assembler code, for prefix and postfix of interrupt routines. This code is microcontroller specific
Definition Ref	A reference from one part of the XML to another. In particular, the XML for a BSW module may refer to another BSW module's XML for certain information. This prevents the same information appearing in multiple places in the XML.
Code Generator	A BSW module is delivered in two parts: code and a Code Generator. The Code Generator consumes complete and correctly formed XML for the BSW module and generates code and data that configures the module.

3 Related documentation

3.1 Input documents

None

3.2 Related standards and norms

None

4 Summary of Interrupt Configuration

This chapter summarises the configuration parameters required for interrupts and where (i.e. in which module) each parameter resides. This summary is from the point-of-view of the ECU Configuration. It is assumed that some system configuration editor with a high level view of the ECU is responsible for the parameter values being placed in the ECU Configuration.

The following table represents a summary of the information to be found in the rest of this document. For longer explanations for what the parameters mean, please read the rest of this document.

BSW Module	***Code contained***	***Parameters Contained in XML***	***Rationale***
OS	All relevant code is automatically generated by the OS code generator.	Interrupt Priority, Category, Vector and name.	All of these are necessary in order to configure the OS. The high level tools need to ensure that only legal combinations of priority, vector and category are used.
BSW Scheduler	Code to enter and exit critical regions that need to be guarded in the interrupt handler. This code is automatically generated by the BSW Scheduler's code generator.	Definition ref to OS object that defines this interrupt. Definition refs to other OS objects (TASKs or Interrupts) that access the critical region. The BSW Scheduler's code generation may also generate RESOURCEs that need to be pushed into the OS's configuration XML.	The references to the OS are required to find out the priority and type of objects (TASK or interrupt) that access critical region. Knowing these enables the BSW Scheduler's code generator to generate the appropriate code.
Device driver module	Declaration of the interrupt handler. This is in C and written by the module's author, i.e. it is not automatically generated. Note that the declarations are different in the category 1 and 2 cases.	Definition ref to OS object that defines this interrupt.	The C definition of the interrupt handler needs to agree with the name and the category in the OS object. The name of the handler needs to be consistent between the C source and OS XML. The C source may need some information in the OS XML for correct generator (for example, some compilers need a

BSW Module	*Code contained*	*Parameters Contained in XML*	*Rationale*
			vector address in order to declare a cat 1 handler).

No timing information is to be found in the above table. This is because no timing model exists currently. Therefore it is not clear how the BSW scheduler can be configured to use anything other than interrupt enable and disable for critical sections.

Note that this document makes references to the BSW Scheduler. Although the BSW Scheduler is now being incorporated into the RTE the arguments presented are still valid.

5 Overview of Interrupt Operation

This overview first explains the steps involved in the handling of an interrupt and then maps those steps onto the different BSW modules involved.

5.1 Distinction between cat1 and cat2 interrupts

There are significant differences between cat1 and cat2 interrupts. These are summarized in the next table.

Attribute	***Cat1***	***Cat2***
Interaction with the OS	Cat1 interrupts are not allowed to interact with the OS data structures. In practice this means that the only OS calls they can make are to enable/disable all interrupts.	Cat2 interrupts are allowed to make most OS calls, other are illegal.
Latency. This is the time from the interrupt being requested by the hardware to the first instruction of the Interrupt Handler.	Cat1 interrupts have typically lower latency than cat2. This is their main advantage.	Cat2 ISRs have typically higher latency than cat1.

Support by the OS. This means that the OS's code generator and libraries abstract the interrupt from the hardware in some portable way.	Unsupported. This means that the code to get safely into and out of the interrupt handler is not generated by the OS and, therefore, has to be generated in some other way.
	Typically the way that this code is generated depends upon the compiler and processor.
		Supported. This is by the ISR macro in C files that declares a function, in a portable manner, to be an interrupt handler. For example,
		ISR(Can_tx) {
		/* some code */
		}
		This is their main advantage.

Configuration. This means capturing enough information in the XML for the OS so that interrupt can be described.	The XML contains all the relevant information about cat1 interrupts. However, it may or may not be used, depending upon the target.	The XML contains all the relevant information about a cat2 interrupts. This is then used to generate the vector table and interrupt hardware manipulation code to get into and out of cat2 ISRs.
Control of interrupt logic. (Target dependency)	Where interrupt logic needs to be manipulated to get in or out of the handler; whether or not	The OS performs the appropriate manipulation.

	this happens depends upon the target. For example, some compilers have an “interrupt” key word to help with this. However, the level of support is very variable.	
Communication with other threads: either tasks or other interrupts handlers. This is really about how mutual exclusion on buffers is handled.	In a TASK or lower priority interrupt, exclusion would be achieved by locking out all interrupts. This is because there is no API to set the priority to specific level and no API to disable/enable a specific interrupt source. In the cat1 Interrupt Handler there would be no need to lock out interrupts because the critical region is shared with a lower priority thread (a TASK or lower priority interrupt). We assume that there is no use case for two Cat1 Interrupt Handlers to directly communicate.	The OS RESOURCE abstraction is the best way to handle interaction between cat2 ISR and TASKs, or other cat2 ISRs. This mechanism knows about cat2 interrupt priorities as well as TASK priorities and, therefore, locks to the lowest priority that guarantees exclusion. The correct priority can be calculated off-line by a tool.

6 Steps in the operation of interrupts

Due to the significant differences between cat1 and cat2 interrupts they are handled in very different ways in the OS and application code. So this section gives an overview of how they are handled.

In each case we discuss what state needs to be set up before the interrupt can occur, and who is responsible for setting that state.

6.1 Handling cat1 interrupts

6.1.1 Initial state

The vector table entry for the interrupt needs to be set so that it points to the interrupt handler.

For cat1 interrupts setting this entry is target-specific. Some implementations of the Autosar OS may support setting the vector table whereas others may not. In the case where the OS does not support this some other method needs to be used such as compiler directives or modifying the vector table.

The interrupt handler needs to be declared correctly. Often there is compiler support for this. For example:

```
__interrupt Can_tx() {
        /* some user application code */
}
```

However, sometimes there is no compiler support and, typically, the syntax and semantics of the support change between compilers. So it may be necessary to provide processor and compiler-specific support for declaring a cat1 handler.

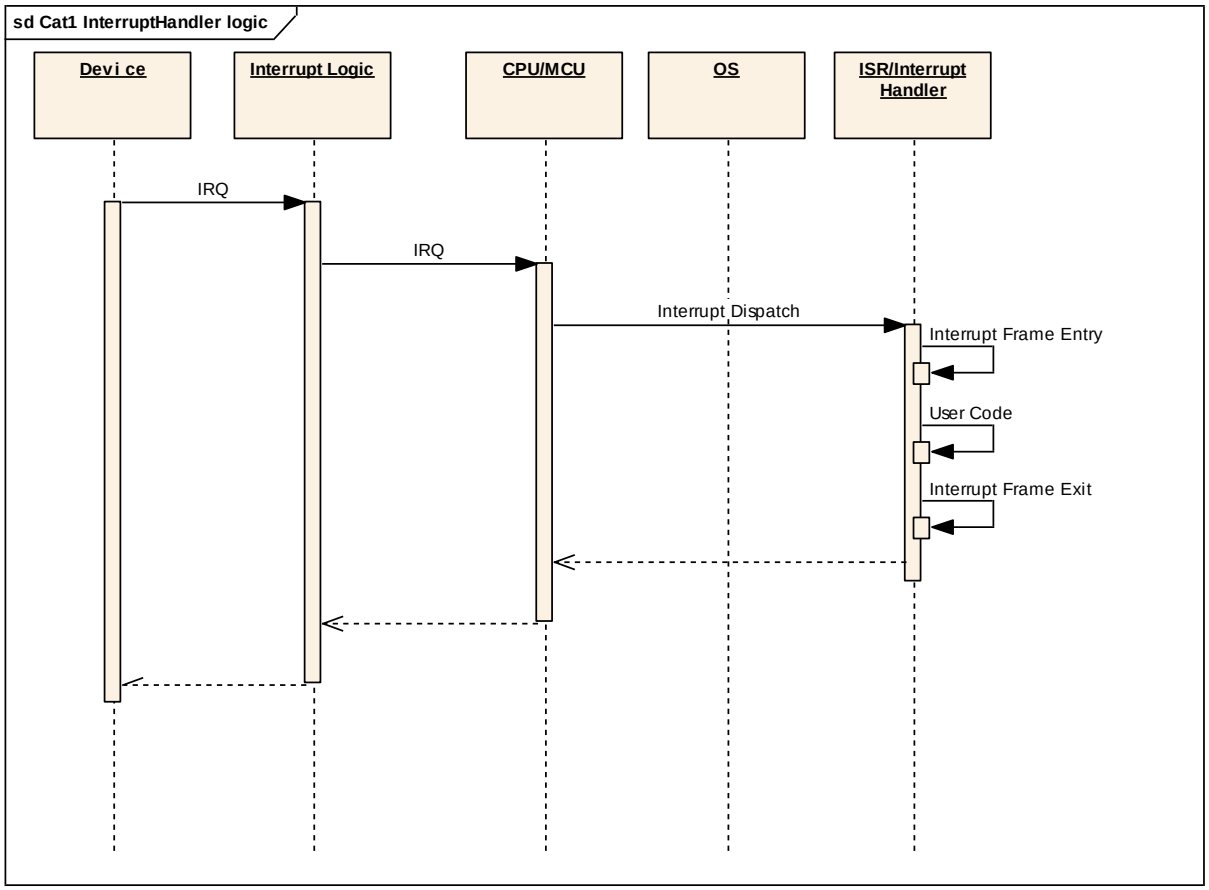
The processor's interrupt logic needs to be set up so that it can request an interrupt. Typically there is no OS support for this in the cat1 case. So this would need to be performed on a per-target basis.

The interrupting device needs to be set up so that it generates an interrupt under the required conditions. This setup is required in the cat1 and cat2 cases and is typically part of the device driver (i.e. in the user's domain).

6.1.2 When the hardware requests an interrupt

These are the steps that occur from when a device requests an interrupt up to the return to the interrupted thread.

Action	***Responsibility***
Requesting the interrupt	Device
Prioritization. Waiting until the processor's priority is low enough that the interrupt can be recognized.	Interrupt logic. Sometimes part of the CPU, and sometimes not.
Recognizing the interrupt. This is allowing the request to interrupt the CPU.	CPU
Saving interrupt state.	CPU
Following the vector table's entry into the interrupt handler.	CPU
Interrupt handler preamble, such as saving compiler-specific registers, etc.	Interrupt handler code generated in response to the __interrupt keyword (or whatever method was used).
Performing the action associated with the interrupt.	User code in the interrupt handler. Note that this code cannot make most OS calls.
Dismissing the interrupt in the device so that it does not immediately re-occur.	User code in the interrupt handler. Note that this code cannot make most OS calls.
Setting the interrupt controller's state so that the interrupt can occur again.	Post-amble code generated by the __interrupt keyword (or whatever method was used). However, may also be in the user's domain. Depends upon the compiler.
Restore compiler registers.	Interrupt handler code generated in response to the __interrupt keyword.
RETI	Interrupt handler code generated in response to the __interrupt keyword.
Restore state of interrupted thread.	CPU



6.2 Handling cat2 interrupts

These provide a much higher level of abstraction than cat1 interrupts but are more expensive at run-time and consume more of the RAM and ROM allocated to the OS.

6.2.1 Initial state

The processor's vector table entry for the interrupt needs to be set so that it points to the OS. For cat2 interrupts setting this entry is handled by the OS's code generator.

The interrupt handler needs to be declared correctly. This is defined as follows for AUTOSAR:

```
ISR(Can_tx) {
        /* some user application code */
}
```

The ISR macro may or may not cause the OS to be entered when an interrupt occurs. However, the main point of this macro is that it encapsulates a cat2 interrupt handler. Therefore the code that ISR expands to is an implementation decision.

The processor's interrupt logic needs to be set up so that it can request an interrupt. In the cat2 case the OS handles this.

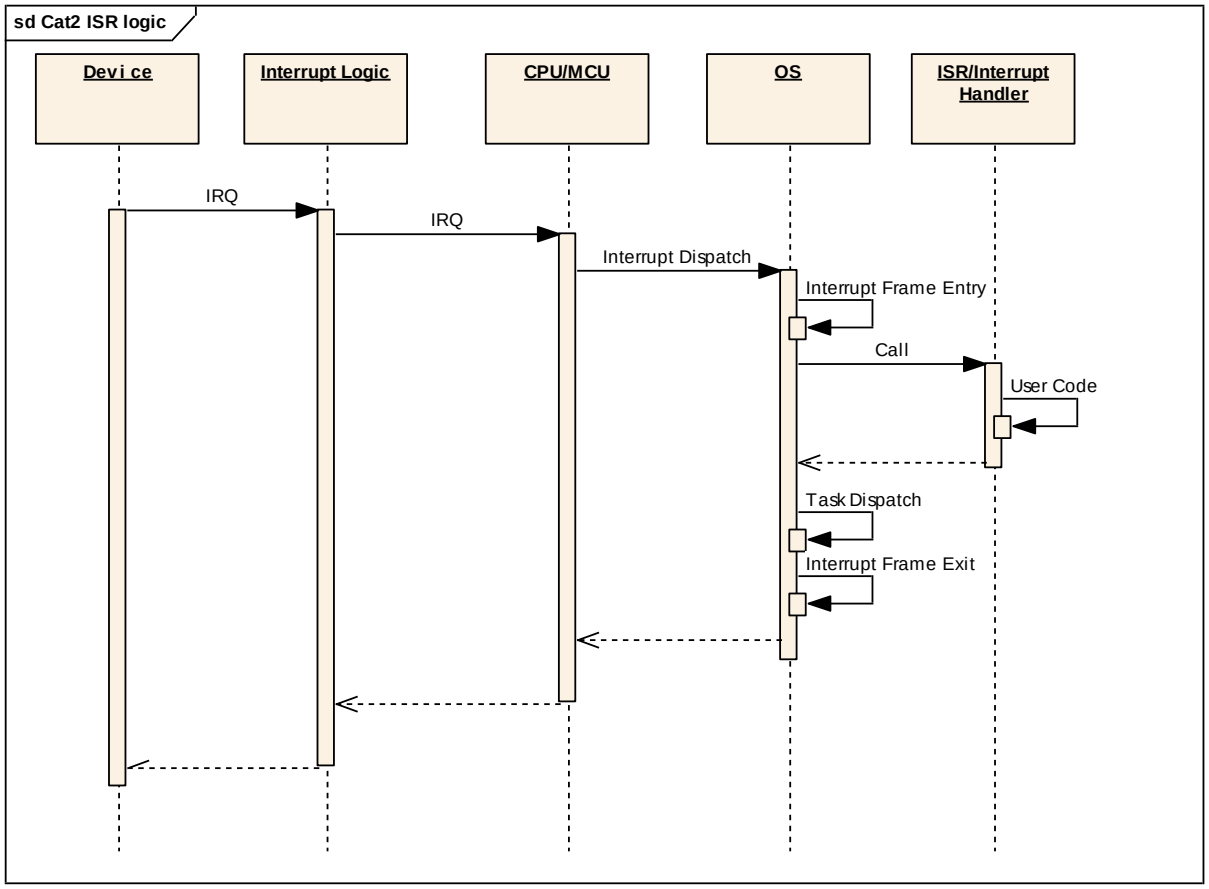
The interrupting device needs to be set up so that it generates an interrupt under the required conditions. This setup is required in the cat1 and cat2 cases and is typically part of the device driver (i.e. in the user's domain).

6.2.2 When the hardware requests an interrupt

The CPU behavior required for a cat2 interrupt is the same as a cat1. Most other aspects are different.

Action	***Responsibility***
Requesting the interrupt	Device
Prioritization. Waiting until the processor's priority is low enough that the interrupt can be recognized.	Interrupt logic. Sometimes part of the CPU, and sometimes not.
Recognizing the interrupt. This is allowing the request to interrupt the CPU.	CPU
Saving interrupt state.	CPU
Following the vector table's entry into the OS.	CPU
OS preamble, such as saving compiler-specific registers, etc. Establishing an OS wrapper around the ISR.	Code generated by the OS's code generator and code which is part of the OS library.
Performing the action associated with the interrupt.	User code in the ISR. Note that this code can make any OS calls.
Dismissing the interrupt in the device so that it does no immediately re-occur.	User code in the ISR. Note that this code can make any OS calls.
Leaving the handler and reentering the OS.	When the OS is reentered it checks for TASK

	activations and then runs appropriate tasks at user priority.
Setting the interrupt logic's state so that the interrupt can occur again.	Part of the OS's generated code and library.
Restore compiler registers.	Part of the OS's generated code and library.
RETI	Part of the OS's generated code and library.
Restore state of interrupted thread.	CPU



7 Configuration of Interrupts

The discussion in chapter 5 identified the following actors in the handling of interrupts:

- The device driver
- The OS
- The BSW scheduler
- Not-configured items

We will now consider the configuration issues for each of these actors, in both the cat1 and cat2 cases.

7.1 Device Driver configuration and code

Each device driver needs to contain the code for the interrupt handler. i.e. the author of the device driver must also write the interrupt handler code as part of the device driver's implementation. However, the code is different in the cat1 and cat2 cases.

Cat2 ISRS are the simplest because they have OS support. The code is based upon the following template:

```
ISR(<name>) {
        <user code to handle ISR>
        <user code to dismiss interrupt>
}
```

The <name> must agree with that chosen in the OS configuration.

Cat1 interrupt handlers are a problem because they do not have OS support. Typically the template would be:

```
<some target specific preamble to mark this function as an interrupt handler>
<name>() {
        <user code to handle ISR>
        <user code to set up interrupt controller>
        <user code to dismiss interrupt>
}
```

However, the exact code required is very processor and compiler specific and not portable. This may not be a problem as device drivers tend not to be very portable anyway.

It is the device driver author's responsibility to write the handlers and, particularly in the cat1 case, to ensure that the correct interaction with the interrupt controller takes place.[1]

[1] In the author's experience this interaction is hard to get right. We expect this to be a source of bugs.

7.1.1 Placement of Interrupt Handlers

Category 1 interrupt handlers are used because they have the fastest response time. Therefore doing anything that slows down category 1 interrupts is counter productive. Therefore category 1 interrupt handlers shall reside in the driver for that interrupting device.

Category 2 handlers are slower and, therefore, more scope in their placement could be allowed. However, to put them anywhere other than the same place as the category 1 handlers multiplies complexity for no useful benefit.

Therefore category 2 interrupt handlers shall also reside in the driver for that interrupting device – there is no thunk in the BSW Scheduler or elsewhere. The term “thunk” in this context means a short piece of code in the BSW Scheduler that simply calls the real handler in the device driver.

7.2 OS configuration

The OS needs to know some fairly complex information in order to configure interrupts correctly. In both cat1 and cat2 cases the following must be known:

- The interrupt vector
- The interrupt priority
- The Interrupt Handler’s <name>
- The category

On some targets one parameter implies a limited range of values for another. For example, on the TriCore the vector implies the priority. So not all combinations of vector, priority and category are legal.

The Interrupt Handler’s <name> can be set either by the person configuring the OS or the person configuring the device driver. It doesn’t really matter who thinks of the name provided that the same name is used in the OS configuration and in the device driver.

Setting the vector, priority and category is much more interesting.[2]

The category (cat1 or cat2) selected in the OS configuration must agree with the implementation strategy chosen in the driver and interface. For validator 2 style configuration[3], the person writing the configuration must ensure that the implementation strategies in the driver, interface and the OS category agree with each other.

In the longer term, however, it would be preferable to have some automatic support. For example, the XML for each module describes which categories are allowed and the configuration captures which category is used. This will enable automatic code generation.

[2] Especially as the Autosar XML has no parameters for the vector or priority.
[3] This is where all modules are configured automatically by hand. i.e. there is no automatic cross module checking or consistency.

So for validator 2 it is probably adequate that the GCE and operator manually select the category based upon information about the implementation of the BSW module.

Setting the vector and priority will also be manual (i.e. via the GCE) in validator 2. However, once again, in the long term, some automated help would be required.

There is typically a degree of dependency between the category, vector and priority.[4] The GCE does not patrol these dependencies and so they must be patrolled by the user of the GCE in conjunction with the manuals for the OS.

In the medium term one would expect knowledge of these dependencies to be built into higher level authoring tools so that the correct relationships are guaranteed by the authoring tools.

So the priority and vector are also configuration items for the OS.

The RESOURCEs to be supplied by the OS must also be specified. In order for these to work correctly the OS must know all the objects (TASKs and ISRs) that refer to each RESOURCE.[5] The BSW Scheduler is responsible for handling critical regions and, therefore, is logically responsible for configuring the OS. However, the BSW scheduler needs knowledge of what critical sections each BSW module needs.

There is no extra OS configuration required for the suspend and resume all interrupts calls.

7.3 BSW Scheduler configuration

The BSW scheduler fulfils two purposes:

1. to provide a TASK that calls the BSW main functions, and
2. to provide code that is responsible for locking critical sections. Therefore critical sections are implemented via the BSW Scheduler only.

The configuration of these two aspects will now be discussed. This discussion relies very heavily upon the author's experience of the communications stack.

The authors are also making the assumption that the BSW scheduler will be written so that it attempts to use the most appropriate method for protecting critical regions.[6]

7.3.1 TASK for main functions

The configuration for the BSW scheduler needs to know which main functions to call and in what order. Typically the code generator for the BSW scheduler will generate code that calls the main functions in order. For example (I've made up plausible names):

[4] Many implementations mandate that at cat1 interrupts must be of a higher priority than the highest cat2 interrupt.
[5] If the OS does not get all of this information correctly critical sections will have obscure bugs.
[6] For more information on this see [1].

```
void Run_com_stack() {
        canif_main_rx();
        linif_main_rx();
        frif_main_rx();
        pdur_main_rx();
        pdumux_main_rx();
        com_main_rx();
        com_main_gw();
        com_main_tx();
        etc…
        TerminateTask();
}
```

This means that the BSW scheduler needs to know the TASKs that contains the main functions in order to tell the OS configuration to configure the RESOURCEs correctly. The BSW scheduler also needs to know the threads of control through the BSW modules and the critical regions referenced. The BSW Scheduler needs to be able to find this information from its configuration data.

7.3.2 Other TASKs in the stack

Typically, BSW modules are entered from flows of control other than the main functions. Therefore, also of interest is, for example, the context that the BSW is entered in from the RTE. This is because the RTE's TASK (or one of its many TASKs) will also, eventually access a critical section and, therefore, call the BSW scheduler. Therefore the RTE's TASK (or TASKs) need to be added to the list that references the RESOURCE used.

All such flows of control into the BSW must be identified and used to configure the BSW scheduler and hence the OS.

7.3.3 Critical sections

This chapter is in 3 sub-sections. The first two sub-sections discuss the issues around critical sections in the two categories of interrupt handler. The final sub-section discusses critical section implementation in the BSW scheduler. So the first two sub-sections describe the problems that the interrupt handlers have and the third sub-section describes how the BSW Scheduler helps to solve those problems.

7.3.3.1 Mutual exclusion in category 1 handlers

1) Assume that we have a cat1 interrupt called C1 and some other thread (either a TASK or an interrupt) called T1.

2) Assume that the priority(C1) > priority(T1).

Assumption 2 implies that, when C1 is running T1 cannot pre-empt, and if T1 is running then C1 can pre-empt. Therefore it is necessary to place protection in T1. Typically this protection is to contain the critical section in matching `SuspendAllInterrupts and ResumeAllInterrupts` calls.

The complimentary case is when:
3) assume that priority(C1) < priority(T1).

In this case T1 must be another cat1 interrupt. (i.e. it cannot be anything else) and, therefore, it is C1's responsibility to contain the critical section protection. We assume that there is no use case for this in Autosar. However, for completeness we now describe the issues in this case.

The dependency between the code and the configuration must be guaranteed to be consistent. This can be done two ways:

- Manually: put the suspend and resume calls into the driver, low overheads and high probability of getting it wrong, or

- Automatically: using the BSW Scheduler module, put the suspend and resume calls into the BS W scheduler with calls to the BSW scheduler in the driver. This has higher overheads and lower probability of getting it wrong.

We would suggest that the BSW scheduler handles this problem, i.e. the automatic option. Therefore it is necessary for the BSW scheduler's configuration to know the category of an interrupt handler.

7.3.3.2 Mutual exclusion in category 2 handlers

In order to achieve mutual exclusion between TASKs and CAT2 ISRs OS RESOURCEs may be used. Interrupt locking can also be used, and is also discussed. RESOURCEs work in all 2 unique cases in the cross product (TASK/ISR, ISR/ISR).

Disabling interrupts can also be used for mutual exclusion. If the two ISRs are of the same priority there is no need for mutual exclusion because they can not run at the same time.

In the user code a critical section is enclosed in calls to the BSW scheduler to enter and leave the critical section. In the BSW scheduler these enter/leave calls are resolved either into resource lock and unlock calls or into interrupt suspend and resume calls. The choice is in the domain of the BSW scheduler's configuration algorithms. However, whichever decision is made, the BSW scheduler needs to have the right information to make that decision and must also inform the OS of any additional configuration objects.

7.3.3.3 Critical sections in the BSW Scheduler

In the BSW scheduler critical sections will typically be implemented in one of two ways:

- Suspend/resume or enable/disable interrupts
- RESOURCEs

This is an important point for the configuration point-of-view. In order to protect critical sections the BSW scheduler could simply suspend/resume all interrupts to enter/leave critical sections. This is very simple to configure requiring almost no knowledge of the application's behavior (i.e. you don't need extra RESOURCEs and, therefore, don't need to know which TASKs/ISRs reference them). However, it also can lead to very long high priority blocking times if the time spent in the critical sections is long.

A better BSW scheduler will use RESOURCEs so that less time is spent at high priority. However, much more information is required for this. In the discussion above we attempted to identify where that information comes from.

A further solution is to make everything a Cat2 ISR or TASK. Then Suspend/Resume OS Interrupts can be used. This requires no knowledge of which BSW requires which RESOURCE. However, it does block out all TASKS and cat2 ISRs in every critical section.

The decision about which scheme is the most appropriate depends upon having timing figures for many parts of the BSW software and a timing model. Currently neither of these exist.

In the case where the BSW scheduler is used to decouple cat1 interrupts (see section 8) then it is only permitted to use suspend/resume all interrupts.

7.3.4 Summary

The configuration of the BSW scheduler is a problem. To configure it trivially (the suspend/resume interrupts case) is simple but has significant disadvantages such as long high priority blocking times. A better configuration (RESOURCEs) needs a lot of information in the BSW scheduler that must then be transferred to the OS. It is not clear how this information is obtained.

8 Recommendations for the use of cat1 interrupts

Most device drivers will be able to use cat2 ISR and shall use cat2 ISRs as the preferred method for handling interrupts.

Cat1 interrupts shall only to be used in the following limited set of circumstances:

- When the interrupt arrival rate causes unacceptable overheads in the OS due cat2 wrappers, or
- when the interrupt latency must be so low that a cat2 ISR is not fast enough.
- when the defined Interrupt Handler would require low jitter.

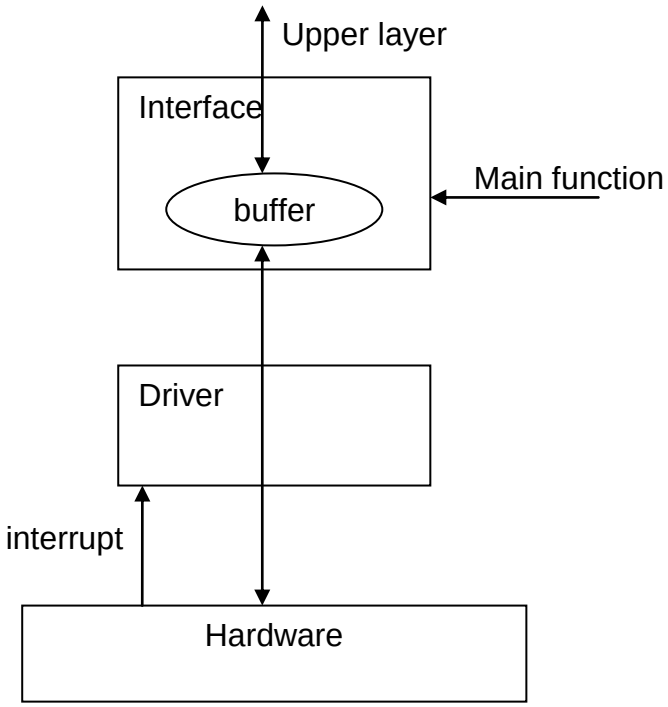
When a Cat1 interrupt is used in a driver the interrupt should be decoupled as soon as possible. At the latest this should be the layer immediately above the driver.

8.1.1 Communication between adjacent modules using cat1 interrupts

Propagating cat1 interrupts too far from the driver is a problem because it means that, for critical sections, large amounts of the stack (memory, communications, whatever) need to know about cat1 interrupts and the blocking times are long.

Long blocking times are a special worry with cat1 interrupts because **all** interrupts are blocked out. Not just a subset.

Therefore cat1 interrupts shall be decoupled as soon as possible. In practice this will work as follows. The figure below show two adjacent modules: a driver that handles cat1 interrupts and an interface.



When the Upper layer wants to send data downwards it asks the interface. The interface locks the buffer by suspending all interrupts via the BSW Scheduler, copies

the data into the buffer or directly to the driver, and then resumes interrupts again via the BSW Scheduler.

When data is received by the driver via an interrupt the driver asks the interface to buffer the data and then exits. This minimizes the amount of time spent at the cat1 priority. At some later point in time the main function is run. This locks out the cat1 interrupts and then copies the data upwards from the buffer by making calls to the upper layer(s).

If the data to be copied to the upper layers then it may be necessary to code the main function as a series of small critical sections. For example, the next piece of code shows one large critical section.

```
Suspend interrupts();
While data in buffers {
        Copy single buffer
}
Resume interrupts();
```

This is the smallest and fastest implementation but has the longest blocking time. A similar implementation is:

```
While data in buffers {
        Suspend interrupts();
        Copy single buffer
        Resume interrupts();
}
```

This is less efficient but has a shorter blocking time and, therefore, is less likely to delay rapidly occurring cat1 interrupts.

8.1.2 Trust

Temporal and spatial protection is specified for the OS to support untrusted code in order to detect and prevent time or space overruns. These checks cannot be implemented for cat1 interrupts. Therefore all cat1 interrupt handlers must be trusted.

However, the situation is rather worse than would be indicated by the previous paragraph. Any code that locks out all interrupts (by suspend all interrupts) also prevents the timer that monitors execution time from interrupting. Therefore temporal protection of any module that, **even without its knowledge as this is a BSW scheduler decision,** locks out all interrupts must be trusted.

The ramification of this is that the simple BSW scheduler implementation, using only disable interrupts for mutual exclusion, implies that all BSW modules must be trusted in such an ECU.