TASKING®

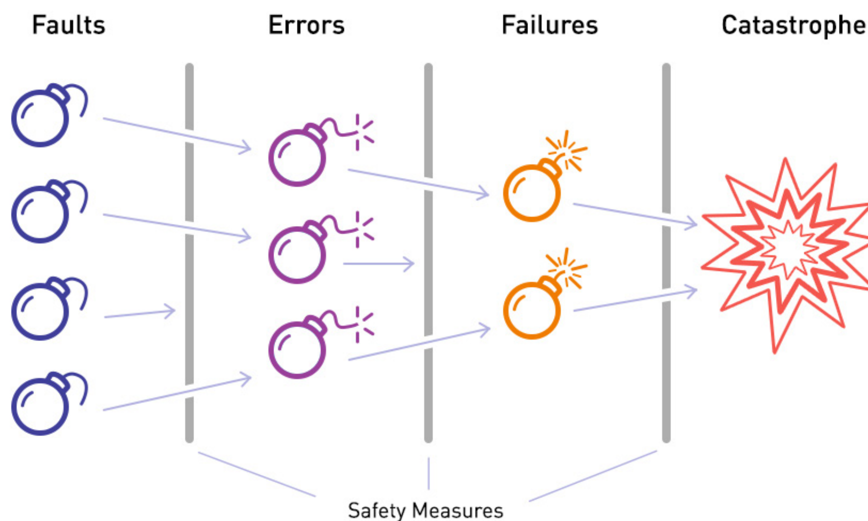
SAFETY CHECKER

OVERVIEW GUIDE

ADVANCING AUTOMOTIVE APPLICATION DEVELOPMENT

Safety-critical software functions required in a car are traditionally placed in separate, single-core Electronic Control Unit (ECU). With this practice, it's easy to ensure that different functions with potentially different functional safety requirements and Automotive Safety Integrity Level (ASIL) are physically insulated and protected against interference from each other.

Today, it is common to combine many of these single core ECUs into a few multi-core ECUs to save costs on wiring and energy consumption. With this new process, functions with different safety requirements and ASIL levels must coexist on the same ECU where no physical insulation is provided. ISO 26262 [5] requires that all software components be developed to the highest ASIL level unless they are partitioned and freedom from interference between the software partitions is established [6].



To avoid the high costs of moving all software components to the highest ASIL level, many software suppliers have started to use software partitioning and Memory Protection Units (MPUs). The MPU is one of the methods supported by the ISO 26262 [6] to establish freedom from interference for memory access and is the most commonly used method today. However, incorrect usage of the MPU can lead to massive financial and legal risks due to critical safety failures in the field. Specifically, the following pitfalls are usually encountered when using the MPU:

- MPUs are notoriously hard to configure correctly. Turning on the MPU often triggers lots of unacceptable MPU access violations (traps) due to small coding and configuration mistakes.
- Debugging and correcting MPU configuration and coding mistakes "trap-by-trap" is time-consuming and costly.
- Achieving high/full test coverage, especially for exceptional corner cases, is prohibitively expensive [3].
- Incorrect MPU configurations and coding mistakes triggering MPU traps are hard to eliminate completely by testing and often create substantial costs when not detected until after delivery of your software "in the field" [1,2].
- Late consideration of MPUs and time-intensive MPU testing delay the discovery of bugs, thus driving the increase in development costs [1,4].

In the remainder of this overview guide, we will discuss the TASKING Safety Checker tool which was developed specifically to mitigate the risks and problems outlined above. First, we will look at the key features of the TASKING Safety Checker before we evaluate how you use the tool in a step-by-step walkthrough. Lastly, we will discuss how the TASKING Safety Checker helps reduce the risk of releasing code which triggers MPU traps by up to **95%** while also saving **69%** of your MPU related testing and bug fixing costs.

OVERVIEW OF THE TASKING SAFETY CHECKER

The TASKING Safety Checker is the only automated analysis tool that enables developers to reduce dramatically the risk of MPU access violations generating in the field. The TASKING Safety Checker also significantly reduces the cost of testing for MPU traps and fixing the offending code.

The tool statically analyzes the component's source code for access violations that would trigger MPU traps as if the code was executed with all possible inputs using an appropriately configured MPU. This differentiates the TASKING Safety Checker from other static analysis tools, which do not analyze MPU access violations and other information including reads from uninitialized memory and data overflows.

Notably, the TASKING Safety Checker does not require hardware, and without any running code. This allows for fast and early detection of common coding mistakes with a quick run of the TASKING Safety Checker after a code change. It also takes into account all possible executions of the provided source code, so that very high coverage, including odd corner cases, is achieved without the cost of creating test cases. Furthermore, a detailed root cause analysis for each identified memory access violation is displayed so that the cost of debugging MPU traps to find and fix the offending code is removed.

Since the tool identifies memory access violations statically (e.g., without running the code), memory access rights can be specified with a much higher granularity without adding any impact to run-time performance. Higher granularity memory access rights allow for detection of more safety violations, (e.g., safety violations between software components that are within the same safety class but should be shielded from each other regardless).

The TASKING Safety Checker can be easily integrated into your preferred development environment or continuous integration framework. This will allow you to minimize the cost of coding mistakes in the day-to-day work of your developers by providing instant feedback before such errors propagate through your code base and cause expensive knock on effects.

AN EXAMPLE WORKFLOW IN THE TASKING SAFETY CHECKER

In this section, an example application of the TASKING Safety Checker is presented.

1. Defining Safety Classes

First, the MPU memory access rights must be defined using the TASKING Safety Checker input format (e.g. within the file "partitioning.saf"), which is given as a command line option when the tool is executed. By providing these settings in an extra file, no source code changes are required to use the TASKING Safety Checker.

TIP

If the memory access rights information is already available in a structured format (e.g., from the architectural documentation), the input file required by the TASKING Safety Checker can be easily generated from the existing information using common, free tools like grep, sed and friends. Otherwise, the input format is a great way to document safety requirements in a formal manner that can be verified.

The TASKING Safety Checker input format supports the C-preprocessor to increase readability. Hence, easy-to-read macros are used to define the different safety classes as shown below:

```
//short cuts for access rights read, write, execute and none
#define R (__SAFETY_CLASS_ACCESS_RIGHTS_READ__) //read
#define W (__SAFETY_CLASS_ACCESS_RIGHTS_WRITE__) //write
#define X (__SAFETY_CLASS_ACCESS_RIGHTS_CALL__) //execute
#define N (0U) //none

//define different safety classes, e.g. ASIL levels
#define QM (0U)
#define ASIL_A (1U)
#define ASIL_B (2U)
#define ASIL_C (3U)
#define ASIL_D (4U)
```

EXPERT NOTE

Next to the common ASIL levels shown above (A-D), any additional safety classes can be defined to detect and prohibit interference between different software components. For example, we could define safety classes `LEFT_DOOR (5U)` and `RIGHT_DOOR (6U)` to restrict access between the two software components that handle the left and right door respectively, although they may have the same ASIL level.

Typically, just the lower granularity access rights that protect only the different ASIL levels (A-D) from each other are configured into the MPU since this is sufficient to fulfill the requirements from ISO 26262 and limits the performance impact of using the MPU. Nevertheless, using such additional, high granularity memory access rights with the TASKING Safety Checker allows more bugs to be caught early during development with no performance penalty.

In the next step, we define the access rights between the different safety classes.

2. Defining Access Rights Between Different Safety Classes

Access rights are defined in an array of structs, where each entry has the format {Src, Dest, Rights}. By default, all safety classes may only access themselves and no other classes. This ensures that no access rights are enabled by accident.

```
__SAFETY_CLASS_ACCESS_RIGHTS__
{
/* Src Dst Rights */
{ QM, QM, N }, /* Disallow access between QM and itself */
{ ASIL_D, ASIL_B, W }, /* An ASIL D function is allowed to write ASIL B data */
{ ASIL_A, ASIL_B, X }, /* An ASIL A function is allowed to execute an ASIL B function */
{ ASIL_C, ASIL_D, R }, /* An ASIL C function is allowed to read ASIL D data */
{ LEFT_DOOR, RIGHT_DOOR, R }, /* only read access allowed */
{ RIGHT_DOOR, LEFT_DOOR, R } /* only read access allowed */
};
```

For the last step, we need to define which functions and variables belong to the different safety classes.

3. Mapping Functions, Variables and Addresses into Safety Classes

Variables and functions that are either global or static can be mapped to different safety classes using an array of structs, where each entry has the format `{"filename.c", "Regular Expression to match functions or global variables", Safety Class}`, as shown in the example below.

> **PLEASE NOTE:**
>
> Local variables inherit the safety class of the function they are declared in. Furthermore, note that all variables and functions that are not explicitly mapped into a safety class are automatically mapped into safety class `(0U)` which corresponds to `QM` in our example.

```
__SAFETY_CLASS_SELECTIONS__
{
/* FileMask, NameMask, SafetyClass */
{ "file1.c", "*", ASIL_A }, //"*" is a regular expression
{ "file2.c", "*", ASIL_B }, //everything in "file2.c" is assigned to safety class ASIL_B
{ "file3.c", "f3", ASIL_C }, //"f3" in file3.c is assigned to safety class ASIL_C
{ "file3.c", "y", ASIL_C } //global variable "y" in module "file3.c" is assigned to safety
class ASIL_C
};

//safety areas allow to assign memory areas to safety class, e.g. to protect SFRs from
unintended access
__SAFETY_CLASS_AREAS__
{
// StartAddress, Size, Class
{ 0x8004000, 0x200, ASIL_D }, //some SFRs
{ 0x8008000, 0x100, ASIL_D }
};
```

This step concludes the configuration of the TASKING Safety Checker. We can now move onto applying safety checker analysis on our sources.

4. Applying Safety Checker Analysis on Sources

After specifying our memory access rights, the sources files can then be analyzed for MPU access right violations. As an example, we examine the source files in the figure below:

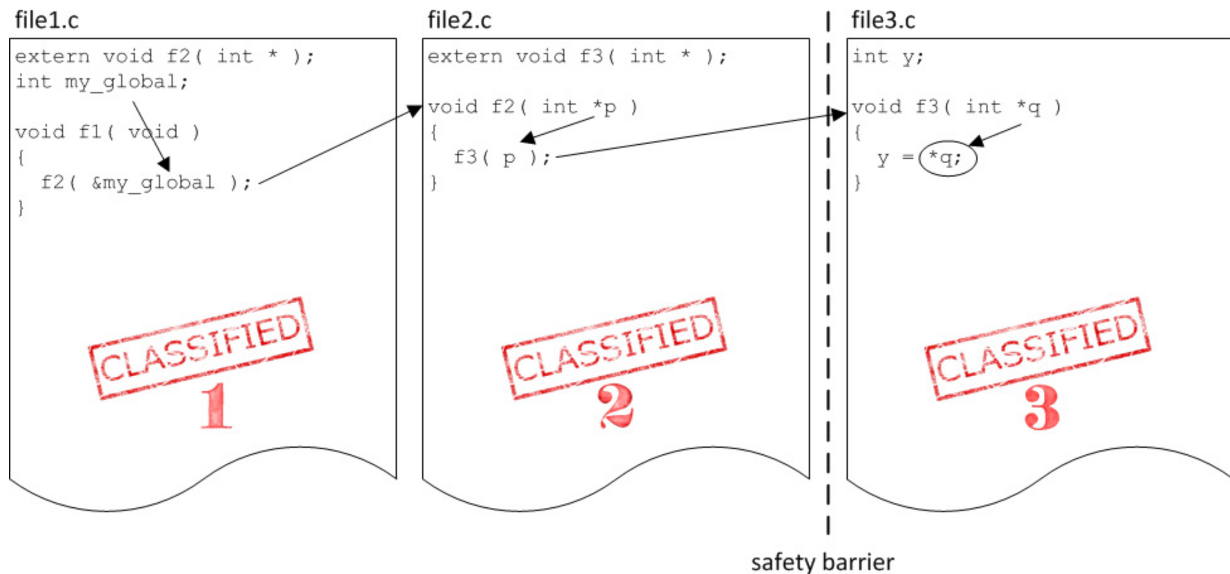


Figure 1 - Examining Source Files for MPU Access Violations

Within these sources, the address of variable `my_global` is passed to function `f2()`. Within function `f2()`, the address of `my_global` is stored in p and passed to function `f3()`. Within function `f3()` the address of `my_global` is stored in `q` and dereferenced followed by a read action.

According to `__SAFETY_CLASS_SELECTIONS__` (from 3.), `f3()` is in `ASIL_C` and my_global is in `ASIL-A`. `__SAFETY_CLASS_ACCESS_RIGHTS__` does not allow access from `ASIL_C` to `ASIL-A`, so the read from `my_global` via `q` in `f3()` is illegal and would trigger an MPU trap, given a correctly configured MPU.

Instead of specifying a test case and tracing the MPU traps generated by the test through several potentially large and complex source files, the TASKING Safety Checker outputs a root cause analysis for each identified memory access violation, as shown in the next section.

5. Fixing MPU Memory Access Violations

Running the TASKING Safety Checker on the input data from the previous sections produces the results shown below. The source of each safety violation can be easily identified using the root cause analysis output from the tool.

```
$ csaf file1.c file2.c file3.c partitioning.saf
csaf E498: [“file3.c” 5] safety violation reading “my_global” (class ASIL_A)
from “f3” (class 3)
csaf I899: [“file1.c” 6] the address of “my_global” is passed from
function “f1” as parameter #1 to function “f2”
csaf I898: [“file2.c” 5] parameter #1 containing the address of “my_global”
is passed from function “f2” as parameter #1 to function “f3”
csaf E499: [“file2.c” 5] safety violation calling “f3” (class ASIL_C)
from “f2” (class 2)
csaf E499: [“file1.c” 6] safety violation calling “f2” (class ASIL_B)
from “f1” (class 1)
3 errors, 0 warnings
```

With the access violation descriptions provided by the TASKING Safety Checker, the problems can be understood quickly and a fix can be planned and implemented within minutes. Once fixed, a quick, second run of the tool can be used to verify the solution. When running the TASKING Safety Checker from within the IDE, the developer can simply click on each output message to directly jump to the offending source location.

COMPARING DEVELOPMENT COSTS AND ROI

In Table 1, we compare the different steps required when handling MPU traps with and without the TASKING Safety Checker. The third column shows the relative costs of each activity when performed with and without the TASKING Safety Checker.

Also, an approximation of the cost reduction of MPU related test and bug fixing costs achieved for each step by using the TASKING Safety Checker is given. We assume conservatively that **40%** of overall MPU related development costs goes into test specification, **40%** into test execution, **15%** into code reviews and **5%** into bug fixing. Furthermore, we assume that the cost ratio of fixing bugs during development, during testing, and in the field is **1:10:100** [3]:

HANDLING OF MPU TRAPS WITHOUT TASKING SAFETY CHECKER	HANDLING OF MPU TRAPS WITH TASKING SAFETY CHECKER	RELATIVE COST ANALYSIS
Define Memory Access Rights (Conceptual+MPU Settings)	Define Memory Access Rights (Conceptual+MPU Settings)	1:1
Manual Code Review to Catch Common Coding Errors	Automatic, Full Coverage of Common Coding Errors	100:1 Reduction of Engineering Time/ Cost for this Task = Cost Reduction of ca. 15%
Create Test Cases (including Special Conditions for High Coverage)	Reduced Testing Effort Due to Automatic High Code Coverage (including special conditions) of the Analysis	3:1 = Cost Reduction of ca. 13% [3] Due to Reduced Test Spec Effort
Run Many Tests, too Expensive to Run After Every Code Change, Slow Turn Around Time and High Fix Costs	Run Analysis After Every Code Change for Quick Turn Around Time and Early/ Cost Effective Fixes	10:1 [1] = Cost Reduction of ca. 36% [3] Due to Early Bug Detection
Trace MPU Access Violations to Origin to Understand Cause	Read Cause from Analysis Result	20:1 Reduction of Engineering Time/ Cost for this Task (Assuming it Takes 20 Mins to Debug/Trace the Cause of an MPU Access Violation) = Cost Reduction of ca. 5%
Fix Code	Fix Code	1:1
Fix Access Violations from Common Coding Errors Missed by Manual Review and Tests	Automatic, Full Coverage of Common Coding Errors Which were Identified by a Major Automotive Software Supplier and TASKING Safety Checker Early Adaptor to Cause ca. 95% of MPU Traps	100:1 [1] = Cost Reduction of 14% * (Ratio of MPU vs Non-MPU recall bugs) of Recall Costs [2] (Reduce Risk of Releasing MPU Traps into the Field by ca. 95%)

Table 1 - Comparison of Different Steps and Costs Associated with Handling MPU traps with and without the TASKING Safety Checker.

USAGE SCENARIOS WITH THE TASKING SAFETY CHECKER

The TASKING Safety Checker can be used as a stand-alone tool which accepts any ANSI C / ISO C / C90 / C99 compliant source files, irrespective of whether the source code is compiled with or without a TASKING compiler. Also, the TASKING Safety Checker can be configured to handle many non-standard C extensions. It is easy to set up and performs a complete memory access violation analysis on the given sources using an easy-to-create, text-based access rights table as the input (see the workflow section above for an example).

Alternatively, the TASKING Safety Checker can be obtained as an optional upgrade for supported TASKING Toolsets (e.g., TASKING 6.0r1 TriCore Toolset). When integrated with a TASKING toolset, the TASKING Safety Checker requires no additional configuration for non-standard C extensions. The description of the memory access rights is given as part of the TASKING linker layout language (lsl script), thus providing a single, integrated definition language to handle all memory and layout related issues.

The integrated version of the tool is the preferred option for customers that are already using TASKING toolsets and are familiar with the TASKING linker layout language. Customers that use TASKING Safety Checker with non-TASKING Toolsets and compilers are bound to the standalone version.

SOFTWARE INTEGRATORS

The TASKING Safety Checker can be optionally used to create partial analysis results at the site of software suppliers using a system integrator defined memory access rights configuration. This allows the supplier to verify that his code does not violate the safety constraints given by the system integrator. Furthermore, these partial results can be combined by the system integrator to obtain a full memory access violation result for the entire integrated system. This allows you to easily and quickly pinpoint memory access violations created by combining the different software components from the different suppliers.

SUMMARY

We have demonstrated in this overview guide how the application of TASKING Safety Checker by your developers early on during the development of your application code will reduce your MPU related test and bug fixing costs by 69% (sum of all savings in Table 1 up to "Fix Code"). Also, reducing the risk of delivering embarrassing, safety critical errors to your customers by up to 95% (last row in Table 1).

The TASKING Safety Checker is easy to integrate into any existing development environment, and allows your developers to quickly analyze and resolve errors at the time of development. This tool is indispensable for any developer working on code that may trigger an MPU, and it provides an automated and cost-effective solution to identify memory access right violations for a variety of common safety standards including ISO 26262-6 6.4.14, IEC 61508-3 7.4.2.8 and SIRF 400 Chapter 5.

Get started with the TASKING Safety Checker today by registering for a free evaluation.

BIBLIOGRAPHY

[1] Boehm, B. W. Understanding and controlling software costs. Journal of Parametrics 8, 32–68 (1988).

[2] http://www.popsci.com/software-rising-cause-car-recalls

[3] Software debugging, testing and verification by Hailpern and Santhanam, 2002 (http://www.cs.uleth.ca/~benkoczi/3720/pres/debug-test-verify_hailpern02.pdf)

[4] Tassey, G. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI Project 7007, (2002).

[5] ISO26262-6 Section 6.4.14, http://www.iso.org/iso/catalogue_detail?csnumber=51362

[6] ISO26262-6 Appendix D, http://www.iso.org/iso/catalogue_detail?csnumber=51362