TASKING

Freedom From Memory Interference

An ASIL Aware Static Analysis and Associated Tools

INTRODUCTION

This whitepaper explains a novel technique to detect and repair memory interferences "ASIL Aware Static Analysis of Memory Interferences" and shows how it is implemented in the TASKING Safety Checker tool. This tool facilitates memory interference detection at software build time, whereas today's hardware and operating systems interference detection mechanisms operate at runtime and possibly detect failures not earlier than after a product has been shipped to the end user.

This whitepaper is published in two parts. Part one provides an introduction into the subject and addresses the current state of the art regarding memory interference detection and recovery from the perspective of safety standards and safety mechanisms implemented in automotive microcontrollers and operating systems.

At the application software level, partitioning can be regarded as a flow-based property. To demonstrate that two software elements belong to different partitions that do not interfere, it is necessary to show that there is no mechanism allowing control or data to flow between them. Static analysis tools support control and data flow analysis that can be applied on the source code or object file. To enable a static analyzer to detect memory interferences it must have knowledge about:

- The SILs taken into consideration
- The allowed inter partition accesses
- The locations of device configuration data (e.g., special function registers)
- The SIL of each code object (functions) and each data object (variables)

This white paper explains the technique - ASIL Aware Static Analysis of Memory Interferences, and shows how it is implemented in the TASKING Safety Checker. Benefits and drawbacks of alternative solutions are described, as well as the capabilities and limitations of the tool. There will also be examples given that show what kind of interferences are detected and what the error diagnostics look like.

STATIC ANALYSIS ON SOURCE CODE VERSUS OBJECT CODE

Static analysis can be applied on source code as well as on binary files. An analysis of the binary file seems to have advantages. The analysis does not depend on the code generation tools used (e.g. the compiler), and can potentially reveal errors that are introduced by code generation tools without depending on the availability of the source code. However, due to the lack high level type information and knowledge about the optimizations applied by the compiler it is often impossible to reconstruct the call and data flow graphs. It is also impossible to trace faults back to the source statement(s) that introduced the fault.

The quality of the analysis improves if it is done on the source code, and accurate elaborate diagnostics can be provided since a fault can be traced back to its origin in the source code. The main drawback is the dependence on the availability of (all) source code. However, if the tool is capable of analyzing individual software elements and storing the "verification data" in an encrypted format, it can then be reused when these software elements are integrated with other software elements, thus mitigating the issue.

SILS TAKEN INTO CONSIDERATION

The term "safety class" refers to SILs and pseudo SILs. This term is introduced for multiple reasons:

To Make the Tool Safety Standard Independent

Different safety standards use different terms to refer to levels of safety measures. To reduce the risk of misinterpretation, the tool's configuration files and diagnostic output contain the same terms as used by the applied safety standard. For example: ISO 26262 uses the terms QM, for quality managed software with no safety

requirements associated and ASILs A, B, C and D, where D represents the most stringent level. EN 50128 uses the levels SIL 0, 1, 2, 3, 4, where 4 represents the most stringent level. DO-178 uses development assurance levels DAL A, B, C, D, E, where A is the most stringent level and E means no safety effect.

To Differentiate Between Different Software Elements With the Same (A)SIL Assigned

Consider AUTOSAR MCAL drivers that are assigned ASIL D. Typically MCAL drivers are allowed to read and write device configuration data, whereas other ASIL D software elements are not allowed to modify the processor's configuration data. Generally, to detect interference between software elements with the same ASIL assigned, additional (pseudo) SILs are used to identify the software elements and specify their access rights.

The example below shows the configuration data for ISO 26262 ASIL levels. ASIL_B is divided into two groups which facilitates fine-grained interference checking between software elements with the same ASIL.

```
__SAFETY_CLASS_ATTRIBUTES__
{
        // Class Name
        { 0, "QM" },
        { 1, "ASIL A" },
        { 2, "ASIL B_SE1" },
        { 3, "ASIL B_SE2" },
        { 4, "ASIL C" },
        { 5, "ASIL D" }
};
```

INTER PARTITION ACCESS RIGHTS

Once the safety classes are known the inter partition access rights can then be specified. The access rights are described using an inter partition access right table, shown in the illustration below. For each safety class it is specified whether it is allowed to read data from, write data to, or execute code from objects assigned to other safety classes. The access table is user configurable and typically created at the start of a project.

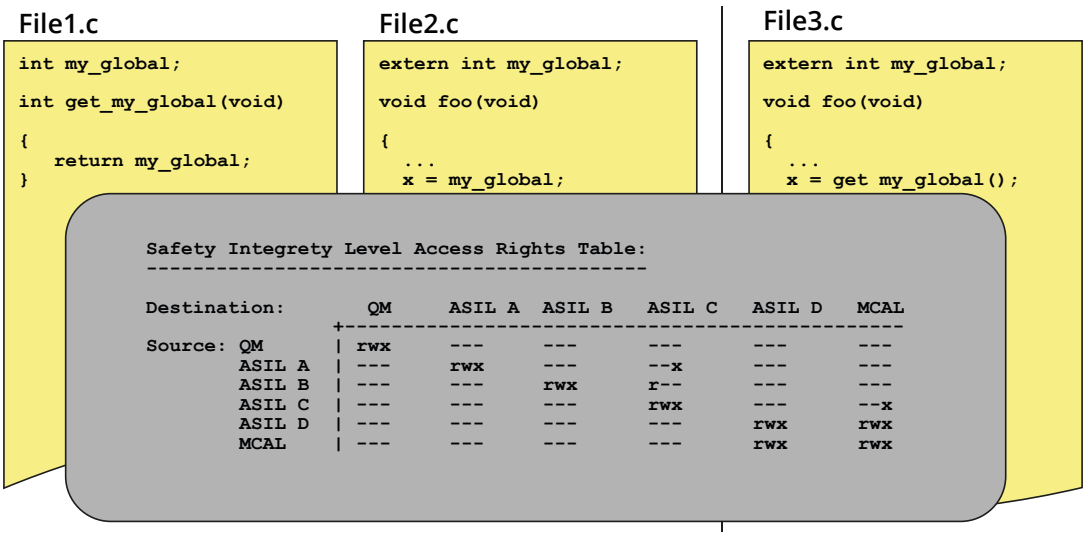

Illustration 10: Inter Partition Access Rights Table

The safety class access rights table pictured above illustrates that:

- All safety classes have full access to objects of the same safety class.
- Software elements assigned ASIL A are allowed to call functions assigned ASIL C and are not allowed to read or write variables assigned ASIL C.

- MCAL is a pseudo-SIL and is treated as ASIL D with the exception that ASIL C functions are allowed to call MCAL functions, whereas ASIL C functions are not allowed to call ASIL D functions.

SIL ASSIGNMENT TO CODE AND DATA OBJECTS

The next step is to assign safety classes to functions and variables. The SIL assignments to code and data objects are continuously updated over the lifecycle of a project when additions are required and new software elements are integrated in the system. It's important to be able to easily manage a large set of data in order to efficiently use the tool. The applied mechanism is shown in the illustration below.

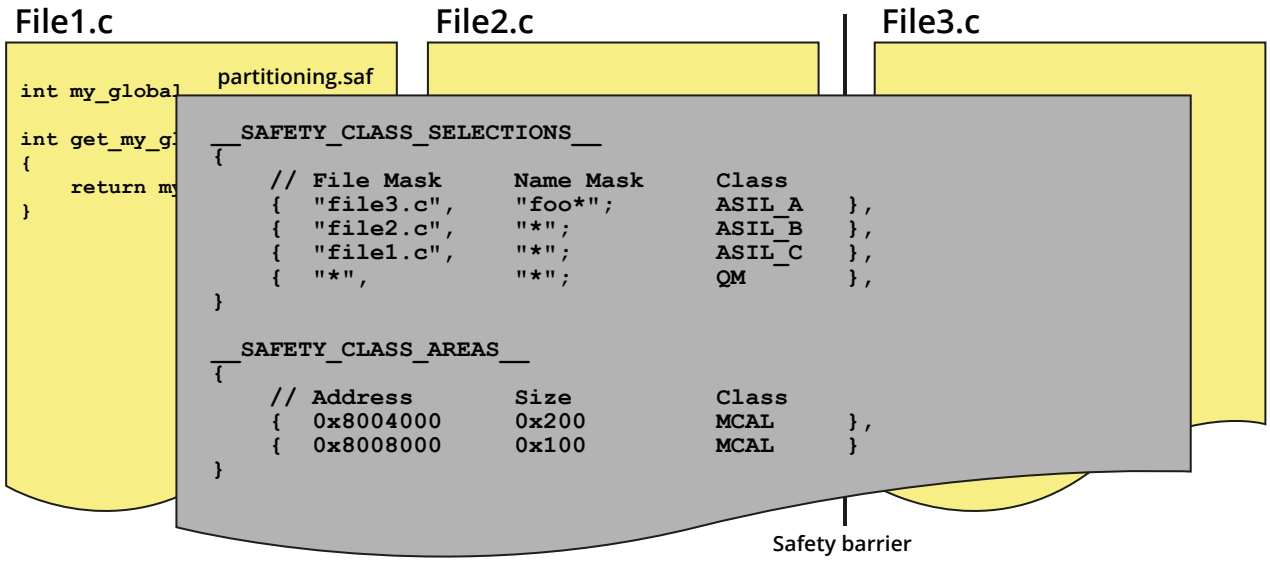

Illustration 11: SIL Assignment to Functions and Variables

Regular expressions are used to select code and data objects based on module name, symbol name, and type (code or data). Notice that in the above example all functions and variables that are not explicitly assigned to a specific ASIL level will be handled as QM code, caused by the entry `{ "*", "*", QM }`.

Locations of Device Configuration Data

In order to facilitate the detection of illegal references to device configuration data, the addresses of these data areas must be specified, as shown in the lower part of the example above. Here it is specified that the address 0x8004000, ..., 0x80041FF, and 0x8008000, ..., 0x80080FF contain device configuration data and these addresses may only be accessed/dereferenced by software elements that have inter partition access rights to safety class MCAL.

TASKING SAFETY CHECKER VARIANTS

The TASKING Safety Checker is available in two variants, both performing the analysis at the source code level. One variant is integrated in the TASKING toolset for Tricore®. The downside of this variant is that it is only available in the most recent version of this toolset and cannot be used in existing projects where an earlier version of the toolset is employed. It also cannot be used in projects where other silicon and/or compiler tools are applied.

The other variant is a stand-alone tool which can be used in any project, independent of the type of microcontroller used. This variant of the Safety Checker can be configured to support 8, 16, 32, or 64-bit microcontrollers, and can handle ISO-C extensions and intrinsic functions that may occur in the source code.

CHECKS IMPLEMENTED IN THE SAFETY CHECKER TOOL

The data and control flow algorithms implemented in the Safety Checker tools take the following into consideration:

- Access (read/write/call) of a variable/function
- Access (read/write/call) of an argument
- Access (read/write/call) of a function return value
- Access (read/write/call) of a fixed address
- Pass (the address of) a variable/function as parameter to a function
- Pass an argument as parameter to a function
- Pass a function return value as parameter to a function
- Pass (the address of) a variable with local lifetime as parameter to a function
- Pass a fixed address as parameter to a function
- Return (the address of) a variable/function
- Return an argument
- Return a function return value
- Return a fixed address

The following restrictions apply: Safety Checker cannot (always) keep track of addresses stored in global variables and the current version does not merge values assigned in different functions.

TOOL USAGE

The Safety Checker tool is integrated into the build flow as shown in the illustration below. Here it is assumed that File1.c and File2.c are developed by a third party. This third party will run the Safety Checker tool on both files and supply the output file File.vd, together with the object files, to the system integrator.

File VD-archive contains Safety Checker outputs received from other third parties, whereas File3.c is a development done by the system integrator.

The system integrator specifies the “SILs taken into account”, the “inter partition access rights”, the “SIL assignment to code and data objects”, and the “Locations of device configuration data”. This information is stored in file “Partitioning.saf.”

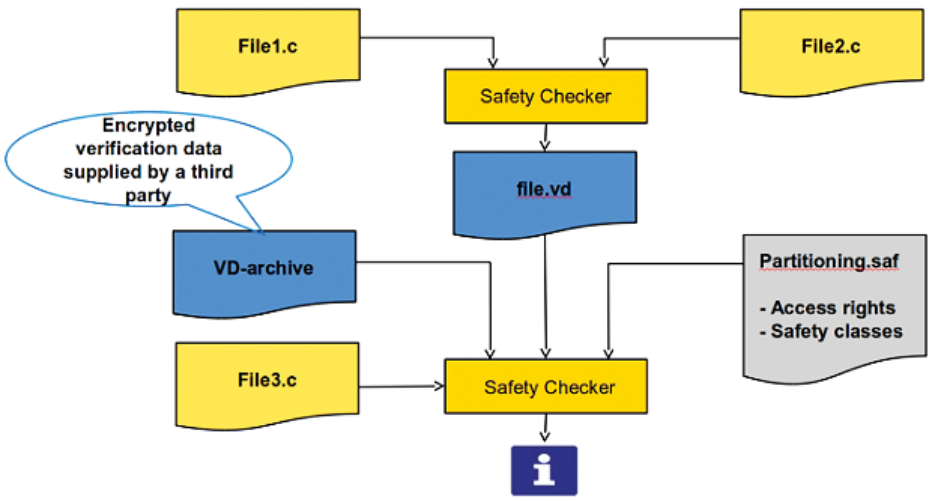

Illustration 12: Safety Checker Usage

SAFETY CHECKER DIAGNOSTIC OUTPUT

The Safety Checker produces an error file and a report file. The error file contains all errors and potential errors in the software. Potential errors represent (r/w/x) accesses where the Safety Checker cannot determine whether or not the access causes an interference. Errors messages occur with elaborate diagnostics that show the root cause of the problem, which is shown in the example below.

Consider that the inter partition data access does not allow read access to ASIL A data objects from ASIL C code. In such case the Safety Checker will produce the given diagnostic message. The whole control flow and data flow that caused the interference to occur is provided. This enables the system integrator to either rapidly fix the issue, or to approach the supplier whose software caused the issue.

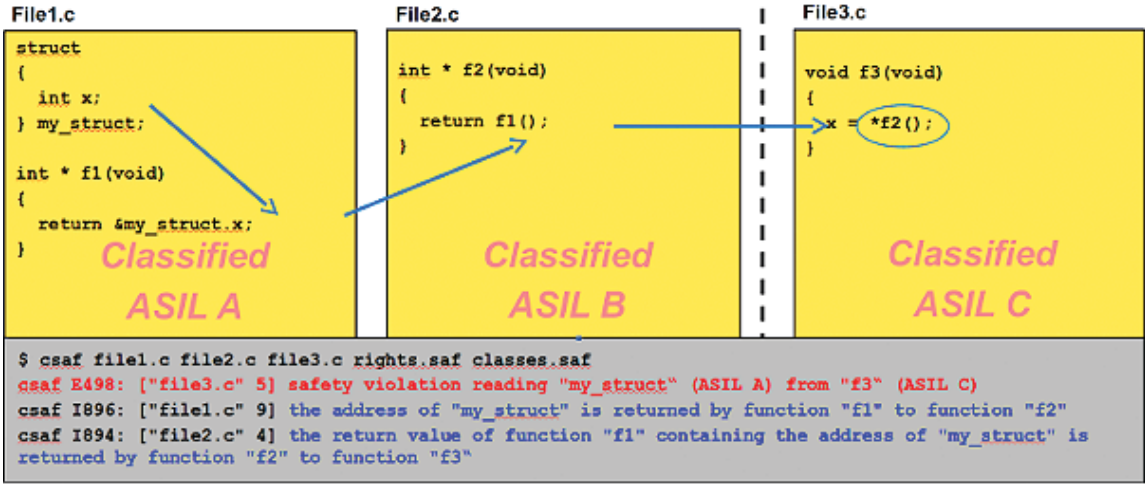

Illustration 13: Safety Checker Diagnostic Output

The report file contains the evidence of proof that the listed accesses (r/w/x) do not violate the freedom from interference requirement for the given software elements and associated access rights specifications.

CONCLUSION

The TASKING Safety Checker raises the standards for preventing interferences in applications. By using this safety analysis tool, you can easily find common engineering errors in your source code which violate ISO 26262 requirements that trigger MPU exceptions in the field. The TASKING Safety Checker also offers a fine-grained fault detection mechanism that reveals inter-partition and intra-partition interferences whereas the latter remain undetected by other MPU or AUTOSAR RTOS based solutions.

The SIL Aware Static Analysis within the TASKING Safety Checker covers the full code base. A partial analysis can start once the first software elements have been implemented and can continue until all software elements are integrated in the system.

The tool has been designed for use within the automotive software supply chain where IP protection is of primary importance. An OEM or tier one has the ability to do an analysis on an integrated system, without requiring access to the source code of the software that is supplied by their third parties.

Combined together, the TASKING Safety Checker l facilitates a significant reduction of costs and efforts to gain freedom from memory interference in your applications.

For more information about Safety Checker contact TASKING