TASKING®

Safety Checker by TASKING User Guide

MA000-740 (v1.0) May 13, 2016

Table of Contents

Manual Purpose and Structure

Manual Purpose

You should read this manual if you want to know:

- how to use the Safety Checker by TASKING
- the features of the Safety Checker by TASKING

Manual Structure

Chapter 1, *Installing the Software*

Explains how to install and license the Safety Checker by TASKING.

Chapter 2, *Introduction*

Contains an introduction to the Safety Checker by TASKING and contains an overview of the features.

Chapter 4, *Using the Safety Checker*

Explains how to use the Safety Checker by TASKING.

Chapter 5, *Tutorial*

Contains a step-by-step tutorial how to prepare your safety critical project for analysis by the Safety Checker.

Chapter 6, *Using the Utilities*

Contains a description of the make utility and archiver.

Chapter 7, *Tool Options*

Contains a detailed list of all command line options of the Safety Checker, make utility and archiver.

Chapter 8, *CERT C Secure Coding Standard*

Contains an overview of the CERT C Secure Coding Standard recommendations and rules that are supported by the Safety Checker by TASKING.

Chapter 9, *MISRA C Rules*

Contains an overview of the supported and unsupported MISRA C rules.

Related Publications: ISO 26262 Standard

- ISO 26262-9:2011(E) Road vehicles - Functional safety - [2011, ISO]

Related Publications: C Standard

- C A Reference Manual (fifth edition) by Samual P. Harbison and Guy L. Steele Jr. [2002, Prentice Hall]
- ISO/IEC 9899:1990, Programming languages - C [ISO/IEC]
- ISO/IEC 9899:1999(E), Programming languages - C [ISO/IEC]

  More information on the standards can be found at http://www.ansi.org/

Related Publications: CERT C Secure Coding Standard

- The CERT C Secure Coding Standard by Robert C. Seacord [October 2008, Addison Wesley]
- The CERT C Secure Coding Standard web site http://www.securecoding.cert.org/

  For general information about CERT secure coding, see http://www.cert.org/secure-coding

Related Publications: MISRA C

- MISRA C:2012, Guidelines for the use of the C language in critical systems [MIRA Ltd, 2013]

  See also http://www.misra-c.com/
- MISRA C:2004, Guidelines for the Use of the C Language in Critical Systems [MIRA Ltd, 2004]

  See also http://www.misra-c.com/
- Guidelines for the Use of the C Language in Vehicle Based Software [MIRA Ltd, 1998]

  See also http://www.misra-c.com/

Chapter 1. Installing the Software

This chapter guides you through the installation process of the Safety Checker by TASKING®. It also describes how to license the software.

1.1. Installation for Windows

System Requirements

Before installing, make sure the following minimum system requirements are met:

- Windows 7 or higher
- 2 GHz Pentium class processor
- 1 GB memory
- 3 GB free hard disk space
- Screen resolution: 1024 x 768 or higher

Installation

1. If you received a download link, download the software and extract its contents.

   - or -

   If you received a CD-ROM, insert it into the CD-ROM drive.

2. Run the installation program (**setup.exe**).

   The TASKING Setup dialog box appears.

3. Select a product and click on the **Install** button. If there is only one product, you can directly click on the **Install** button.

4. Follow the instructions that appear on your screen.

1.2. Licensing

TASKING products are protected with TASKING license management software (TLM). To use a TASKING product, you must install that product and install a license.

The following license types can be ordered from Altium.

Node-locked license

A node-locked license locks the software to one specific computer so you can use the product on that particular computer only.

For information about installing a node-locked license see Section 1.2.3.2, *Installing Server Based Licenses (Floating or Node-Locked)* and Section 1.2.3.3, *Installing Client Based Licenses (Node-Locked)*.

Floating license

A floating license is a license located on a license server and can be used by multiple users on the network. Floating licenses allow you to share licenses among a group of users up to the number of users (seats) specified in the license.

For example, suppose 50 developers may use a client but only ten clients are running at any given time. In this scenario, you only require a ten seats floating license. When all ten licenses are in use, no other client instance can be used.

For information about installing a floating license see Section 1.2.3.2, *Installing Server Based Licenses (Floating or Node-Locked)*.

License service types

The license service type specifies the process used to validate the license. The following types are possible:

- **Client based** (also known as 'standalone'). The license is serviced by the client. All information necessary to service the license is available on the computer that executes the TASKING product. This license service type is available for node-locked licenses only.

- **Server based** (also known as 'network based'). The license is serviced by a separate license server program that runs either on your companies' network or runs in the cloud. This license service type is available for both node-locked licenses and floating licenses.

  Licenses can be serviced by a cloud based license server called "**Remote TASKING License Server**". This is a license server that is operated by TASKING. Alternatively, you can install a license server program on your local network. Such a server is called a "**Local TASKING License Server**". You have to configure such a license server yourself. The installation of a local TASKING license server is not part of this manual. You can order it as a separate product (SW000089).

  The benefit of using the Remote TASKING License Server is that product installation and configuration is simplified.

Unless you have an IT department that is proficient with the setup and configuration of licensing systems we recommend to use the facilities offered by the Remote TASKING License Server.

1.2.1. Obtaining a License

You need a license key when you install a TASKING product on a computer. If you have not received such a license key follow the steps below to obtain one. Otherwise, you cannot install the software.

Obtaining a server based license (floating or node-locked)

- Order a TASKING product from Altium or one of its distributors.

  A license key will be sent to you by email or on paper.

If your node-locked server based license is not yet bound to a specific computer ID, the license server binds the license to the computer that first uses the license.

Obtaining a client based license (node-locked)

To use a TASKING product on one particular computer with a license file, Altium needs to know the computer ID that uniquely identifies your computer. You can do this with the **getcid** program that is available on the TASKING website. The detailed steps are explained below.

1. Download the **getcid** program from http://www.tasking.com/support/tlm/download.shtml.

2. Execute the **getcid** program on the computer on which you want to use a TASKING product. The tool has no options. For example,

   ```
   C:\Tasking\getcid
   Computer ID: 5Dzm-L9+Z-WFbO-aMkU-5Dzm-L9+Z-WFbO-aMkU-MDAy-Y2Zm
   ```

   The computer ID is displayed on your screen.

3. Order a TASKING product from Altium or one of its distributors and supply the computer ID.

   A license key and a license file will be sent to you by email or on paper.

When you have received your TASKING product, you are now ready to install it.

1.2.2. Frequently Asked Questions (FAQ)

If you have questions or encounter problems you can check the support page on the TASKING website.

http://www.tasking.com/support/tlm/faq.shtml

This page contains answers to questions for the TASKING license management system TLM.

If your question is not there, please contact your nearest Altium Sales & Support Center or Value Added Reseller.

1.2.3. Installing a License

The license setup procedure is done by the installation program.

If the installation program can access the internet then you only need the licence key. Given the license key the installation program retrieves all required information from the remote TASKING license server. The install program sends the license key and the computer ID of the computer on which the installation program is running to the remote TASKING license server, no other data is transmitted.

If the installation program cannot access the internet the installation program asks you to enter the required information by hand. If you install a node-locked client based license you should have the license file at hand (see Section 1.2.1, *Obtaining a License*).

Floating licenses are always server based and node-locked licenses can be server based. All server based licenses are installed using the same procedure.

1.2.3.1. Configure the Firewall in your Network

For using the TASKING license servers the TASKING license manager tries to connect to the Remote TASKING servers `lic1.tasking.com` .. `lic4.tasking.com` at the TCP ports 8080, 8936 or 80. Make sure that the firewall in your network is transparently enabled for one of these ports.

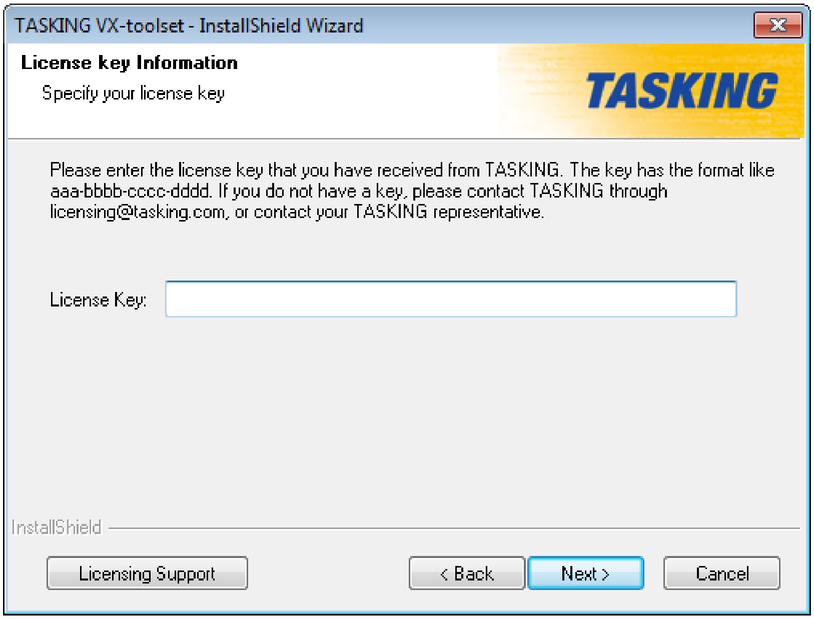
1.2.3.2. Installing Server Based Licenses (Floating or Node-Locked)

If you do not have received your license key, read Section 1.2.1, *Obtaining a License* before you continue.

1. If you want to use a local license server, first install and run the local license server before you continue with step 2. You can order a local license server as a separate product (SW000089).

2. Install the TASKING product and follow the instruction that appear on your screen.

   The installation program asks you to enter the license information.



TASKING VX-toolset - InstallShield Wizard

License key Information

Specify your license key

Please enter the license key that you have received from TASKING. The key has the format like aaa-bbbb-cccc-dddd. If you do not have a key, please contact TASKING through licensing@tasking.com, or contact your TASKING representative.

License Key:

3. In the **License key** field enter the license key you have received from Altium and click **Next** to continue.

   The installation program tries to retrieve the license information from a remote TASKING license server. Wait until the license information is retrieved. If the license information is retrieved successfully subsequent dialogs are already filled-in and you only have to confirm the contents of the dialogs by clicking the ***Next*** *button. If the license information is not retrieved successfully you have to enter the information by hand.*

4. Select your **License type** and click **Next** to continue.

   You can find the license type in the email or paper that contains the license key.

5. Select **Remote TASKING license server** to use one of the remote TASKING license servers, or select **Local TASKING license server** for a local license server. The latter requires optional software.

6. (For local license server only) specify the **Server name** and **Port number** of the local license server.

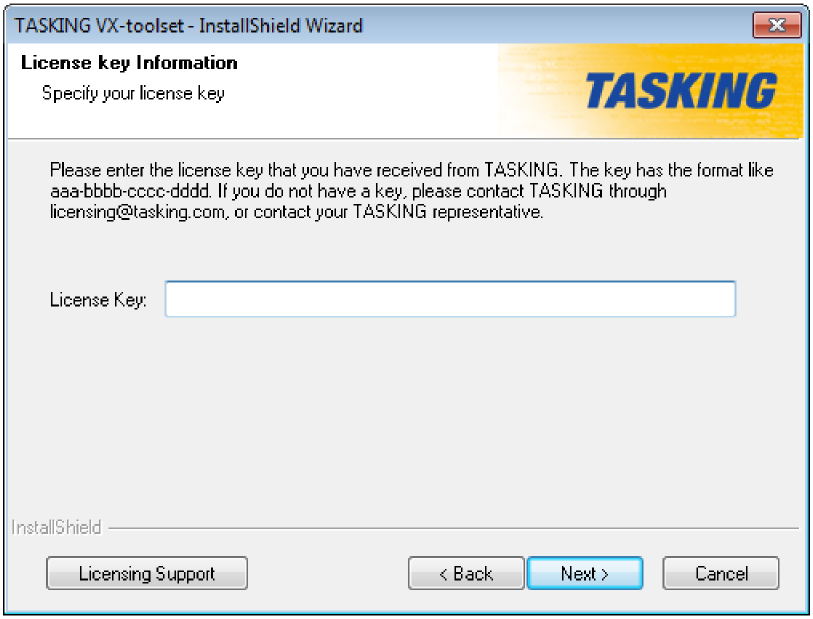
7. Click **Finish** to complete the installation.

1.2.3.3. Installing Client Based Licenses (Node-Locked)

If you do not have received your license key and license file, read Section 1.2.1, *Obtaining a License* before continuing.

1. Install the TASKING product and follow the instruction that appear on your screen.

   The installation program asks you to enter the license information.



TASKING VX-toolset - InstallShield Wizard

License key Information

Specify your license key

Please enter the license key that you have received from TASKING. The key has the format like aaa-bbbb-cccc-dddd. If you do not have a key, please contact TASKING through licensing@tasking.com, or contact your TASKING representative.

License Key:

2. In the **License key** field enter the license key you have received from Altium and click **Next** to continue.

   *The installation program tries to retrieve the license information from a remote TASKING license server. Wait until the license information is retrieved. If the license information is retrieved successfully subsequent dialogs are already filled-in and you only have to confirm the contents of the dialogs by clicking the **Next** button. If the license information is not retrieved successfully you have to enter the information by hand.*

3. Select **Node-locked client based license** and click **Next** to continue.

4. In the **License file content** field enter the contents of the license file you have received from Altium.

   The license data is stored in the file licfile.txt in the etc directory of the product.

5. Click **Finish** to complete the installation.

Chapter 2. Introduction

The Safety Checker by TASKING is a stand alone solution for safety-related code analysis, for example for ISO 26262 certification.

Automotive safety integrity levels are an absolute must to guarantee safe and reliable execution of applications in the real-time environment of an automobile. Yet developers constantly struggle with the need to partition their applications and reduce the interference between software components while still maintaining the highest safety levels possible.

One way to achieve this goal in mixed criticality systems is to prove that low-level safety functions, like audio related applications, do not interfere with high-level safety functions, like braking systems. This so called ‘Freedom from Interference’ is the goal, but not an easy one to achieve in new multi-core architectures where more functions are being combined into one control unit.

The Safety Checker by TASKING provides you with the needed tools to perform this task, including:

- The ability to work at the compiler level so that you do not have to change the source code in an application.
- Information about memory allocated is already included, with unique ways to identify memory write/read access.
- Information about the safety classes of functions are built into the application, allowing you to easily address the implementation of safety requirements with SIL (Safety Integrity Level) aware static analysis.

Features of the Safety Checker

- C syntax checking and semantic analysis. The Safety Checker supports ISO/IEC 9899:1990 (C90) and ISO/IEC 9899:1999 (C99)
- Deep analysis and warn for potential user errors.
- Verification of MISRA C guidelines: MISRA C:1998, MISRA C:2004, MISRA C:2012
- Verification of CERT C secure coding standard.
- SIL aware static analysis of interferences in the memory space and special function register space (SFRs)

2.1. Mixed Criticality Systems and ISO 26262

Mixed criticality is the concept of allowing software at different levels of criticality to interact and coexist in the same Electronic Control Unit (ECU). For example, software for controlling the electronic steering wheel unit and software controlling the door-open light have different levels of criticality. You have to assure that the 'simple' door-open software does not interfere with the 'high classified' steering software. You have to assure this so-called 'freedom from interference'.

2.1.1. ISO 26262

ISO 26262 is a Functional Safety standard, titled "Road vehicles – Functional safety". You can use it to certify mixed criticality systems. This standard gives two options:

1. Treat all software according to the highest safety level.

2. Prove freedom from interference.

The first option is to treat all software according to the highest safety level. See it as a “Free upgrade”. This can be a good choice for relatively simple software, like controlling a light to indicate an open door. But if sophisticated audio software has to be upgraded to a higher safety level, it is a complete different story. Then you could probably better obtain for option 2.

The second option is to prove freedom from interference. Evidence is made available that, for each safety requirement allocated to the element, it cannot interfere with any sub-element with a higher safety level. The Safety Checker focusses on this option.

2.1.2. Freedom From Interference

Interference can occur in the time domain and in the memory (space) domain.

Interference in the time domain

This occurs when a safety-relevant function is blocked by faulty software. This kind of problem can be detected by what is called a watchdog. Mostly a combination of hardware and software which is able to recognize the problem and then tries to recover or put the system in a safe state. This is mainly the domain of OS and hardware suppliers.

Interference in the memory (space) domain

You have to prevent that one software partition can change the code or data of another software partition.

Interference in the memory domain is also the concern of OS and hardware suppliers. One technical solution is to use a Memory Protection Unit (MPU) or a Memory Management Unit (MMU).

MMUs are rarely found in embedded processors for many reasons. For example, an MMU can increase the area it occupies in the processor significantly and its power consumption can be doubled. The comprehensive in-memory data structures that hold the memory access permissions result in vulnerability with respect to transient hardware errors that can break address mappings and permissions.

From all hardware protection units, MPUs are the most common in processors for the deeply embedded and real-time systems for their simple design and predictable run-time behavior. MPUs provide low-cost and low-overhead memory protection that is suited for cost-sensitive markets. The simplistic design and the robustness with respect to transient hardware faults make MPU-based protection easily verifiable and therefore a good choice for safety-critical applications. The absence of caches and in-memory data structures results in deterministic behavior that does not increase the complexity of worst-case execution time analysis, thus MPUs meet the requirements of real-time applications.

So, the Memory Protection Unit (MPU) can assure freedom from interference in the memory space domain. The MPU checks at run-time for illegal memory access. If such an access occurs the system can try to recover or put itself in a safe state.

But that is quite a rough remedy and not very helpful if your application is distributed to all your customers already. It would be valuable if you knew about such interferences at an earlier stage. But when and how?

This is were the Safety Checker can help you. It can help you to check and analyze your code and warn when there is no freedom of interference at an early stage. This will be made clear in the following sections.

2.2. C Syntax Checking

Basic syntax checking

The Safety Checker performs basic C syntax checking.

For example:

```
int ;  /* empty declaration */
```

Invocation of the Safety Checker results in a simple error, like every compiler should report.

```
$ csaf a.c
csaf W514: ["a.c" 1/1] empty declaration
```

Old C style

The Safety Checker has the ability to check for C90 code and C99 code.

For example:

```
foo(bar)    /* no type specifiers */
{
  return bar;
}
```

Invocation:

```
$ csaf --iso=90 a.c
0 errors, 0 warnings

$ csaf --iso=99 a.c
csaf W535: ["a.c" 1/1] no type specifiers - assumed 'int'
csaf W538: ["a.c" 1/5] undeclared parameter "bar" defaults to "int"
```

The Safety Checker complains because no (explicit) types are specified. Although no one would dare to write these lines of code anymore, it is still valid C according to the C90 standard. The C99 standard is much more strict and therefore it is also the default

C99 extensions

Some language extensions are only available in C99 code and not in C90 code.

For example:

```
/* invalid type for ISO C90 */
long long int my_long_long;
```

Invocation:

```
$ csaf --iso=90 a.c
csaf E243: ["a.c" 2/1] invalid type "long long int"

$ csaf --iso=99 a.c
0 errors, 0 warnings
```

Type `long long` is not supported in C90 code. However, if your code has to meet the ISO C90 standard, use the option **--iso=90**.

2.3. Semantic Analysis

Superfluous code

The Safety Checker can warn for superfluous code.

For example:

```
if (b1 || (!b1 && b2))
{
  ...
}
```

Invocation:

```
$ csaf a.c
csaf W549: ["a.c" 1/13] condition is always false
```

The Safety Checker signals the condition in column 13 is always false. Is this true? Yes, it is. The right side of the expression (`!b1 && b2`) will only be evaluated if the left side (`b1`) is false. Therefore, the value of `b1` in the right side of the expression will always be evaluated as false. We can see all combinations by creating a truth table:

`b1`	`b2`	`b1 || (!b1 && b2)`
True	True	True
True	False	True
False	True	True
False	False	False

The truth table indicates this is function could be simply rewritten as:

```
if (b1 || b2)
{
  ...
}
```

In this case the extra check is harmless, but superfluous code could sometimes point you to a potential hazard. The Safety Checker also checks for unused variables, assignment to a variable which is not used further on, parts of code that is never reached, etcetera.

Boundary checking

The Safety Checker signals out of bounds access.

For example:

```
int my_array[10];

for (int j = 20; j < 30; j++)
{
  my_array[j] = 0;
}
```

Invocation:

```
$ csaf a.c
csaf W561: ["a.c" 5/13] access out of bounds
```

Type checking

The Safety Checker can perform type checking. For example with incompatible conversion specifications in a `printf`/`scanf` format string.

```
#include <stdio.h>

void f(long value)
{
    printf( "%d\n", value ); /* incorrect argument type */
    printf( "%ld\n", value ); /* OK */
}
```

Invocation:

```
$ csaf a.c
csaf W570: ["a.c" 5/21] type of argument #2 is incompatible with format
```

2.4. MISRA C Checks

The Safety Checker has support for checking against MISRA C guidelines.

For example:

```
#include <stdint.h>

int x;    /* int32_t is safer */
```

Invocation:

```
$ csaf --misrac=all a.c
csaf E354: ["a.c" 3/5] MISRA C 2004 rule 6.3 violation: [A] specific-length
           typedefs should be used instead of the basic types
```

Because the size of an integer is target-specific, MISRA C 2004 rule 6.3 says: specific-length typedefs should be used instead of the basic types.

See Section 4.8.2, *C Code Checking: MISRA C* and Chapter 9, *MISRA C Rules* for more information.

2.5. CERT C Checks

The Safety Checker has support for CERT C secure code checking.

For example:

```
#define SQUARE(n) ((n) * (n))

y = SQUARE(++x);
```

Invocation:

```
$ csaf --cert a.c
csaf W700: ["a.c" 4/14] CERT check EXP30: do not depend on order
           of evaluation between sequence points
```

The macro in this example will expand as "`++x * ++x`". So, x will be increased twice. CERT will warn for this.

See Section 4.8.1, *C Code Checking: CERT C* and Chapter 8, *CERT C Secure Coding Standard* for more information.

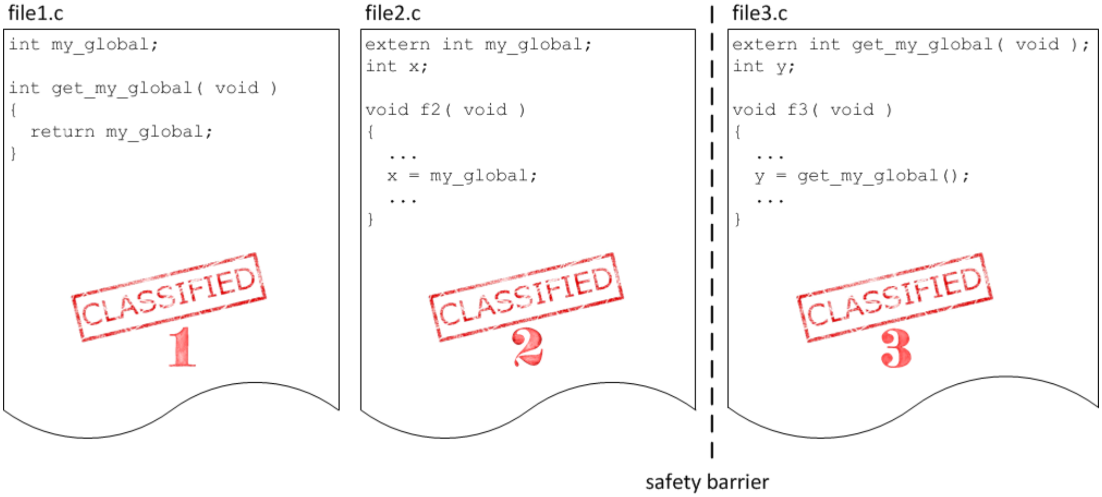
2.6. SIL Aware Safety Checks

How the Safety Checker can perform SIL aware static analysis on your application is demonstrated by the following four examples.

Note that the safety classes used by the Safety Checker are just arbitrary numbers. Which number you assign to which SIL level is up to you.

Example 1

Imagine we have three modules, all belonging to a different safety class. And we want an almost physical barrier between module 3 and the two other modules.



file1.c

```
int my_global;

int get_my_global( void )
{
  return my_global;
}
```

file2.c

```
extern int my_global;
int x;

void f2( void )
{
  ...
  x = my_global;
  ...
}
```

file3.c

```
extern int get_my_global( void );
int y;

void f3( void )
{
  ...
  y = get_my_global();
  ...
}
```

Safety class access rights

Now access rights can, for example, be specified in a way module 2 has direct read-access to global data declared in module 1. But module 3 is not allowed to read data from module 1 directly. It should call the Get-function to get the value of the variable `my_global`. How will the safety class access rights table look like?

```
* Safety Class Access Rights Table
==================================

    Target:    0    1    2    3
            +--------------------
Source: 0   |  rwx  ---  ---  ---
        1   |  ---  rwx  ---  ---
        2   |  ---  r--  rwx  ---
        3   |  ---  --x  --x  rwx
```

The safety class access rights table shows that:

- All safety classes have (by default) full access to objects of the same safety class.
- Safety class 2 has read access to safety class 1 objects.
- Safety class 3 has execute access to safety class 1 and 2 objects.

How to pass this information to the Safety Checker?

You can specify the access rights table as an initialized predefined C structure and you can insert this structure anywhere in the source code.

```
__SAFETY_CLASS_ACCESS_RIGHTS__
{
    // Src Dst Rights

    {  2,  1,  __SAFETY_CLASS_ACCESS_RIGHTS_READ__ },
    {  3,  1,  __SAFETY_CLASS_ACCESS_RIGHTS_CALL__ },
    {  3,  2,  __SAFETY_CLASS_ACCESS_RIGHTS_CALL__ }
};
```

Most obvious you will store it in a separate file with a specific name for example `partitioning.saf`. This file is simply passed to the Safety Checker just as if it was an ordinary source file.

Safety class selection and safety class areas

How does the Safety Checker determine which part of the code belongs to which safety class?

The safety class selection table and the safety class areas table are used for this. For example,

```
__SAFETY_CLASS_SELECTIONS__
{
    // File Mask    Name Mask   Class
    {  "file1.c",   "*",        1     },
    {  "file2.c",   "*",        2     },
    {  "file3.c",   "f3",       3     },
    {  "file3.c",   "y",        3     }
    // everything else gets class 0 by default
};

__SAFETY_CLASS_AREAS__
{
    // Address    Size      Class
    {  0x8004000, 0x200,    4      },
    {  0x8008000, 0x100,    4      }
};
```

Both tables are also specified as an initialized predefined C structure and you can insert these structures anywhere in the source code. Most obvious you will store it in the same file as the access rights, for example `partitioning.saf`.

In the safety class selection table you can make a selection on:

- Module name or parts of a module name
- Symbol name or parts of a symbol name

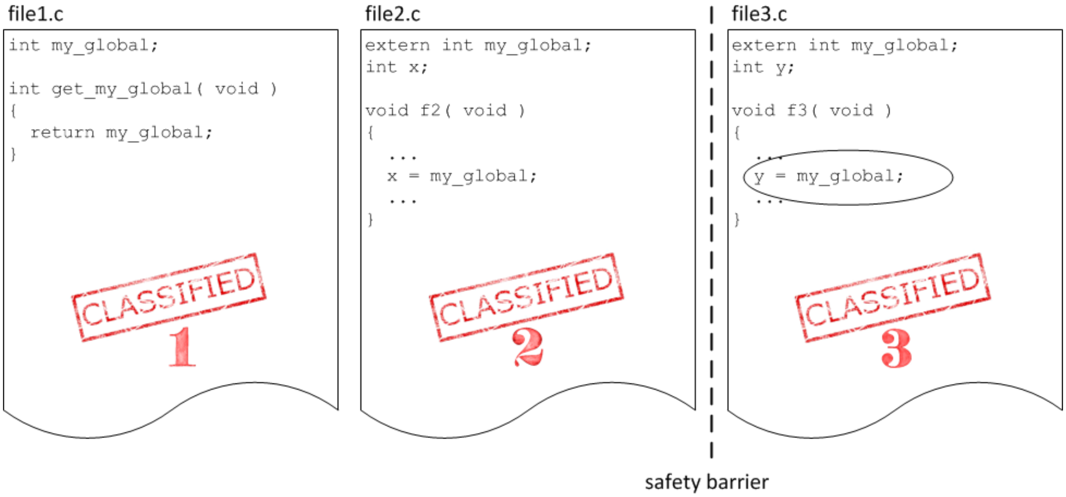
In the safety class selection table above everything from `file1` is marked class 1, everything from `file2` is marked class 2 and only function `f3` and variable `y` from `file3` are marked class 3. All other objects are marked class 0 by default.

For fixed address areas like special function registers (SFR), you specify the start address and the size of the area in the safety class areas table.

Now we have set all rights correctly. Module 3 has no direct access to (static) data of module 1 and therefore module 3 has to use the function `get_my_global()` to get the value of the static variable `my_global`.

Example 2

But what would happen if module 3 would access the variable `my_global` directly?

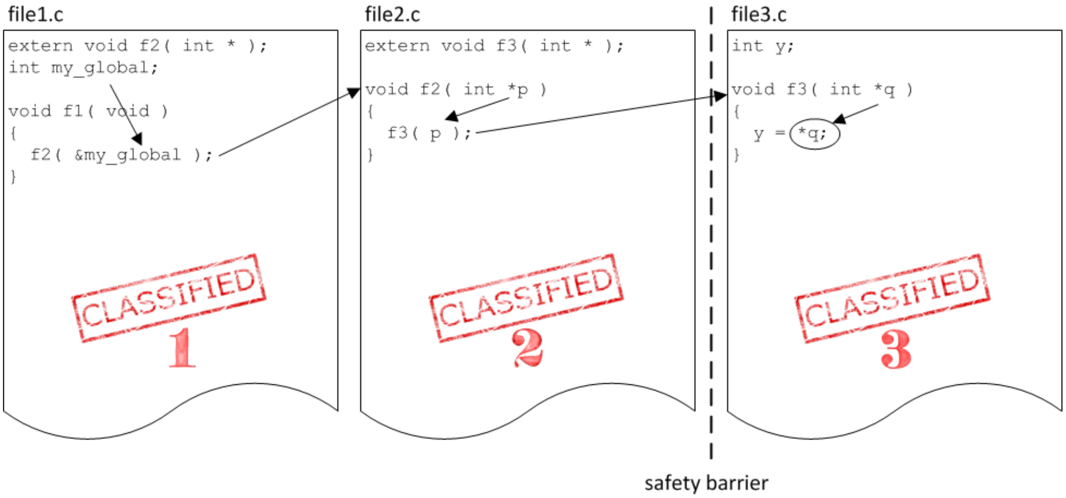


As you may expect, the Safety Checker reports the safety violation:

```
$ csaf file1.c file2.c file3.c partitioning.saf
csaf E498: ["file3.c" 7] Safety violation reading "my_global" (class 1)
           from "f3" (class 3)
```

Example 3

Example 2 showed a situation where global data was accesses directly. But what will happen if we pass the address of variable `my_global` from module 1 to module 2 to module 3 and let module 3 read the content of the address? This is a completely accepted way to work in C. Follow the arrows in the following example.



```
extern void f2( int * );
int my_global;

void f1( void )
{
  f2( &my_global );
}
```

```
extern void f3( int * );

void f2( int *p )
{
  f3( p );
}
```

```
int y;

void f3( int *q )
{
  y = *q;
}
```

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 the specified access rights table, this is not permitted. But how could the Safety Checker know?

Call tree

First, the Safety Checker constructs a call tree. The tree starts at the *, from left to right.

```
        +--f3()
        |
    +--f2()
    |
*--f1()
```

Data flow

Then, the Safety Checker then extends the call tree to a data flow. In this case the address of a variable is passed to another function, but it is also possible to return the address of a variable.

```
        +--f3(&my_global)
        |
    +--f2(&my_global)
    |
*--f1()
```

Access tree

After the data flow is constructed, the Safety Checker checks the data accesses.

```
        +--f3(&my_global)
        |   :
        |   :.. [chk] Check access for reading "my_global" from "f3"  <-- 4
        |   :.. [chk] Check access for writing "y" from "f3"          <-- 3
        |
    +--f2(&my_global)
    |   :
    |   :.. [chk] Check access for calling "f3" from "f2"             <-- 2
    |
*--f1()
    :
    :.. [chk] Check access for calling "f2" from "f1"                 <-- 1
```

Point 4 is about reading (indirectly) the variable `my_global` by module 3.

You can tell the Safety Checker to generate a report with this information.

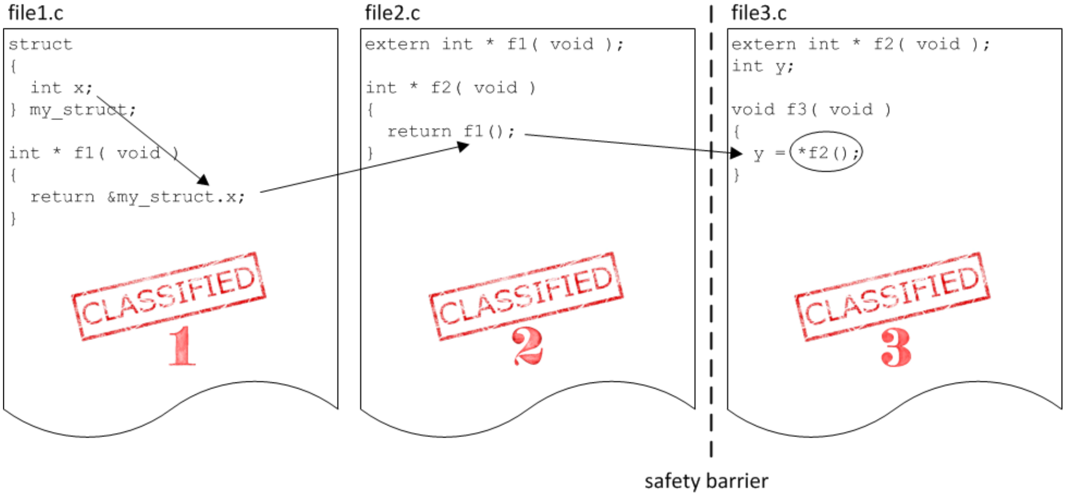
The Safety Checker reports this access violation as follows:

```
$ csaf file1.c file2.c file3.c partitioning.saf
csaf E498: ["file3.c" 5] safety violation reading "my_global" (class 1)
           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 3)
           from "f2" (class 2)
csaf E499: ["file1.c" 6] safety violation calling "f2" (class 2)
           from "f1" (class 1)
3 errors, 0 warnings
```

Not only the violation itself is reported, but also information is provided how the address of the variable `my_global` is passed from function `f1()` in module 1 to function `f3()` in module 3. This information can help you to find the origin of the safety violation.

Example 4

Follow the arrows in the following example.



```
struct
{
  int x;
} my_struct;

int * f1( void )
{
  return &my_struct.x;
}
```

```
extern int * f1( void );

int * f2( void )
{
  return f1();
}
```

```
extern int * f2( void );
int y;

void f3( void )
{
  y = *f2();
}
```

The address of a structure member is returned by function `f1()` in module 1. The return value of `f1()` is returned by function `f2()` in module 2. And function `f3()` in module 3 reads the content of the address returned by function `f2()`.

How does the Safety Checker handle this?

Call tree

Again, the Safety Checker first constructs call tree. This time `f3()` calls `f2()` which calls `f1()`.

```
        +--f1()
        |
    +--f2()
    |
*--f3()
```

Data flow

The Safety Checker then extends the call tree to a data flow.

```
        +--f1() -> &my_struct
        |
    +--f2() -> &my_struct
    |
*--f3()
```

As you can see, the address of `my_struct` (not `my_struct.x`) is returned by `f1()` and therefore also indirectly by `f2()`.

Access tree

After the data flow is constructed, the Safety Checker checks the data accesses.

```
        +--f1() -> &my_struct
        |
    +--f2() -> &my_struct
    |   :
    |   :.. [chk] Check access for calling "f1" from "f2"     <-- 4
    |
*--f3()
    :
    :.. [chk] Check access for reading "my_struct" from "f3"  <-- 3
    :.. [chk] Check access for writing "y" from "f3"          <-- 2
    :.. [chk] Check access for calling "f2" from "f3"         <-- 1
```

Point 3 is about reading (indirectly) the `my_struct` member by module 3.

The Safety Checker reports this access violation as follows:

```
$ csaf file1.c file2.c file3.c partitioning.saf
csaf E499: ["file2.c" 5] safety violation calling "f1" (class 1)
           from "f2" (class 2)
csaf E498: ["file3.c" 6] safety violation reading "my_struct" (class 1)
           from "f3" (class 3)
csaf I896: ["file1.c" 8] the address of "my_struct" is returned by
           function "f1" to function "f2"
csaf I894: ["file2.c" 5] the return value of function "f1" containing the
           address of "my_struct" is returned by function "f2" to function "f3"
2 errors, 0 warnings
```

Not only the violation itself is reported, but also information is provided how the address of the variable `my_struct` is passed from function `f1()` in module 1 to function `f3()` in module 3. This information can help you to find the origin of the safety violation.

Accesses not checked

Due to the complexity of the application, it is not always possible to reconstruct the correct data flow. But at least you want to know all the accesses not checked by the Safety Checker. So, this is also reported. For example.

```
$ csaf file1.c file2.c file3.c partitioning.saf
csaf E799: ["file3.c" 4] unable to check safety violation
```

Chapter 3. C Language

This chapter describes the features of the C language supported by the Safety Checker by TASKING and describes how to configure the Safety Checker for non-standard C code.

The Safety Checker by TASKING fully supports the ISO-C standard, and additional generic language extensions, but it does not add extra possibilities to program the special functions of a target.

It is required to configure the Safety Checker for non-standard C code. The Safety Checker generates a syntax error when a target specific feature is used. See Section 3.6, *Configuring the Safety Checker for Non-Standard C Code*.

In addition to the ISO-C standard, the Safety Checker supports the following:

- attributes
- pragmas to control the Safety Checker from within the C source
- predefined macros

All TASKING non-standard keywords have two leading underscores (__).

3.1. Data Types

The Safety Checker supports the ISO C99 defined data types. The default sizes of these types are shown in the following table.

C Type	Size	Limits
_Bool	1	0 or 1
signed char	8	$[-2^{7}, 2^{7}-1]$
unsigned char	8	$[0, 2^{8}-1]$
short	16	$[-2^{15}, 2^{15}-1]$
unsigned short	16	$[0, 2^{16}-1]$
int	32	$[-2^{31}, 2^{31}-1]$
unsigned int	32	$[0, 2^{32}-1]$
enum [1]	8 16 32	$[-2^{7}, 2^{7}-1]$ or $[0, 2^{8}-1]$ $[-2^{15}, 2^{15}-1]$ or $[0, 2^{16}-1]$ $[-2^{31}, 2^{31}-1]$
long	32	$[-2^{31}, 2^{31}-1]$
unsigned long	32	$[0, 2^{32}-1]$
long long	64	$[-2^{63}, 2^{63}-1]$
unsigned long long	64	$[0, 2^{64}-1]$
float (23-bit mantissa)	32	[–3.402E+38, –1.175E-38] [+1.175E-38, +3.402E+38]

C Type	Size	Limits
double long double (52-bit mantissa)	64	[-1.797E+308, -2.225E-308] [+2.225E-308, +1.797E+308]
_Imaginary float	32	[–3.402E+38i, –1.175E-38i] [+1.175E-38i, +3.402E+38i]
_Imaginary double _Imaginary long double	64	[-1.797E+308i, -2.225E-308i] [+2.225E-308i, +1.797E+308i]
_Complex float	64	real part + imaginary part
_Complex double _Complex long double	128	real part + imaginary part
pointer to data or function	32	[0, 2^{32}-1]

You can change most of the default integer sizes by using one of the options **--bit-size-<type>**

[1] When you use the `enum` type, the Safety Checker will use the smallest suitable integer type (`char`, `unsigned char`, `short`, `unsigned short` or `int`).

3.2. Shift JIS Kanji Support

In order to allow for Japanese character support on non-Japanese systems (like PCs), you can use the Shift JIS Kanji Code standard. This standard combines two successive ASCII characters to represent one Kanji character. A valid Kanji combination is only possible within the following ranges:

- First (high) byte is in the range 0x81-0x9f or 0xe0-0xef.

- Second (low) byte is in the range 0x40-0x7e or 0x80-0xfc

Safety Checker option **-Ak** enables support for Shift JIS encoded Kanji multi-byte characters in strings and (wide) character constants. Without this option, encodings with 0x5c as the second byte conflict with the use of the backslash ('\') as an escape character. Shift JIS in comments is supported regardless of this option.

Note that Shift JIS also includes Katakana and Hiragana.

Example:

```
// Example usage of Shift JIS Kanji
// Do not switch off option -Ak
// At the position of the italic text you can
// put your Shift JIS Kanji code
int  i;  // put Shift JIS Kanji here
char c1;
char c2;
unsigned int ui;
const char mes[]="put Shift JIS Kanji here";
const unsigned int ar[5]={'K','a','n',
                     'j','i'};
                     // 5 Japanese array
void main(void)
{
    i=(int)c1;
    i++; /* put Shift JIS Kanji here\
            continuous comment */
    c2=mes[9];
    ui=ar[0];
}
```

3.3. Attributes

You can use the keyword `__attribute__` to specify special attributes on declarations of variables, functions, types, and fields.

Syntax:

`__attribute__((`*name*`,...))`

or:

`__`*name*`__`

The second syntax allows you to use attributes in header files without being concerned about a possible macro of the same name. This second syntax is only possible on attributes that do not already start with an underscore. For example, you may use `__noreturn__` instead of `__attribute__((noreturn))`.

const

You can use `__attribute__((const))` to specify that a function has no side effects and will not access global data. See also attribute `pure`.

The following kinds of functions should not be declared `__const__`:

- A function with pointer arguments which examines the data pointed to.
- A function that calls a non-const function.

export

You can use `__attribute__((export))` to specify that a variable/function has external linkage and should not be removed.

```
int i __attribute__((export)); /* 'i' has external linkage */
```

flatten

You can use `__attribute__((flatten))` to force inlining of all function calls in a function, including nested function calls.

Unless inlining is impossible or disabled by `__attribute__((noinline))` for one of the calls, the generated code for the function will not contain any function calls.

format(*type*,*arg_string_index*,*arg_check_start*)

You can use `__attribute__((format(type,arg_string_index,arg_check_start)))` to specify that functions take `printf`, `scanf`, `strftime` or `strfmon` style arguments and that calls to these functions must be type-checked against the corresponding format string specification.

type determines how the format string is interpreted, and should be `printf`, `scanf`, `strftime` or `strfmon`.

arg_string_index is a constant integral expression that specifies which argument in the declaration of the user function is the format string argument.

arg_check_start is a constant integral expression that specifies the first argument to check against the format string. If there are no arguments to check against the format string (that is, diagnostics should only be performed on the format string syntax and semantics), *arg_check_start* should have a value of 0. For `strftime`-style formats, *arg_check_start* must be 0.

Example:

```
int foo(int i, const char * my_format, ...) __attribute__((format(printf, 2, 3)));
```

The format string is the second argument of the function `foo` and the arguments to check start with the third argument.

leaf

You can use `__attribute__((leaf))` to specify that a function is a leaf function. A leaf function is an external function that does not call a function in the current compilation unit, directly or indirectly. The attribute is intended for library functions to improve dataflow analysis. The attribute has no effect on functions defined within the current compilation unit.

malloc

You can use `__attribute__((malloc))` to improve optimization and error checking by telling the Safety Checker that:

- The return value of a call to such a function points to a memory location or can be a null pointer.
- On return of such a call (before the return value is assigned to another variable in the caller), the memory location mentioned above can be referenced only through the function return value; e.g., if the pointer value is saved into another global variable in the call, the function is not qualified for the malloc attribute.
- The lifetime of the memory location returned by such a function is defined as the period of program execution between a) the point at which the call returns and b) the point at which the memory pointer is passed to the corresponding deallocation function. Within the lifetime of the memory object, no other calls to malloc routines should return the address of the same object or any address pointing into that object.

noreturn

Some standard C function, such as abort and exit cannot return. The Safety Checker knows this automatically. You can use `__attribute__((noreturn))` to tell the Safety Checker that a function never returns. For example:

```
void fatal() __attribute__((noreturn));

void fatal( /* ... */ )
{
  /* Print error message */
  exit(1);
}
```

The function `fatal` cannot return. The Safety Checker can optimize without regard to what would happen if `fatal` ever did return. This can produce slightly better code and it helps to avoid warnings of uninitialized variables.

pure

You can use `__attribute__((pure))` to specify that a function has no side effects, although it may read global data. See also attribute `const`.

section("*section_name*")

You can use `__attribute__((section("name")))` to specify that a function must appear in the object file in a particular section. For example:

```
extern void foobar(void) __attribute__((section("bar")));
```

puts the function `foobar` in the section named `bar`.

used

You can use `__attribute__((used))` to prevent an unused symbol from being removed by the Safety Checker. Example:

```
static const char copyright[] __attribute__((used)) = "Copyright 2016 Altium BV";
```

When there is no C code referring to the `copyright` variable, the Safety Checker will normally remove it. The `__attribute__((used))` symbol attribute prevents this.

unused

You can use `__attribute__((unused))` to specify that a variable or function is possibly unused. The Safety Checker will not issue warning messages about unused variables or functions.

3.4. Pragmas to Control the Safety Checker

Pragmas are keywords in the C source that control the behavior of the Safety Checker. Pragmas overrule Safety Checker options. Put pragmas in your C source where you want them to take effect. Unless stated otherwise, a pragma is in effect from the point where it is included to the end of the compilation unit or until another pragma changes its status.

The syntax is:

```
#pragma [label:]pragma-spec pragma-arguments [on | off | default | restore]
```

or:

```
_Pragma( "[label:]pragma-spec pragma-arguments [on | off | default | restore]" )
```

Some pragmas can accept the following special arguments:

`on`	switch the flag on (same as without argument)
`off`	switch the flag off
`default`	set the pragma to the initial value
`restore`	restore the previous value of the pragma

Label pragmas

Some pragmas support a label prefix of the form "*label***:**" between `#pragma` and the pragma name. Such a label prefix limits the effect of the pragma to the statement following a label with the specified name. The `restore` argument on a pragma with a label prefix has a special meaning: it removes the most recent definition of the pragma for that label.

You can see a label pragma as a kind of macro mechanism that inserts a pragma in front of the statement after the label, and that adds a corresponding `#pragma ... restore` after the statement.

Compared to regular pragmas, label pragmas offer the following advantages:

- The pragma text does not clutter the code, it can be defined anywhere before a function, or even in a header file. So, the pragma setting and the source code are uncoupled. When you use different header files, you can experiment with a different set of pragmas without altering the source code.
- The pragma has an implicit end: the end of the statement (can be a loop) or block. So, no need for pragma restore / endoptimize etc.

Example:

```
#pragma lab1:optimize P

volatile int v;

void f( void )
{
      int i, a;
      a = 42;

lab1: for( i=1; i<10; i++ )
      {
        /* the entire for loop is part of the pragma optimize */
        a += i;
      }
      v = a;
}
```

Supported pragmas

The Safety Checker recognizes the following pragmas, other pragmas are ignored. On the command line you can use **csaf --help=pragmas** to get a list of all supported pragmas. Pragmas marked with (*) support a label prefix.

STDC FP_CONTRACT [on | off | default | restore] (*)

This is a standard ISO C99 pragma. With this pragma you can control the **+contract** flag of Safety Checker option **--fp-model**.

boolean [on | off | default | restore] (*)

This pragma is used to mark the macros "false" and "true" from the library header file `stdbool.h` as "essentially BOOLEAN", which is a concept from the MISRA C:2012 standard.

extension isuffix [on | off | default | restore] (*)

Enables a language extension to specify imaginary floating-point constants. With this extension, you can use an "i" suffix on a floating-point constant, to make the type `_Imaginary`.

```
float 0.5i
```

fp_negzero [on | off | default | restore] (*)

With this pragma you can control the **+negzero** flag of Safety Checker option **--fp-model**.

fp_nonan [on | off | default | restore] (*)

With this pragma you can control the **+nonan** flag of Safety Checker option **--fp-model**.

fp_rewrite [on | off | default | restore] (*)

With this pragma you can control the **+rewrite** flag of Safety Checker option **--fp-model**.

macro / nomacro [on | off | default | restore] (*)

Turns macro expansion on or off. By default, macro expansion is enabled.

message "*message*" ...

Print the message string(s) on standard output.

nomisrac [*nr*,...] [default | restore] (*)

Without arguments, this pragma disables MISRA C checking. Alternatively, you can specify a comma-separated list of MISRA C rules to disable.

See Safety Checker option **--misrac** and Section 4.8.2, *C Code Checking: MISRA C*.

stdinc [on | off | default | restore] (*)

This pragma changes the behavior of the `#include` directive. When set, the Safety Checker options **--include-directory** and **--no-stdinc** are ignored.

warning [*number*,...] [default | restore] (*)

With this pragma you can disable warning messages. If you do not specify a warning number, all warnings will be suppressed.

3.5. Predefined Preprocessor Macros

The TASKING Safety Checker supports the predefined macros as defined in the table below. The macros are useful to create conditional C code.

Macro	Description
__BUILD__	Identifies the build number of the Safety Checker, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the Safety Checker, __BUILD__ expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.
__CSAF__	Identifies the Safety Checker. You can use this symbol to flag parts of the source which must be recognized by the Safety Checker by TASKING only. It expands to 1.
__DATE__	Expands to the compilation date: “mmm dd yyyy”.
__DOUBLE_FP__	Expands to 1 if you used option **--fp-model=-float**, otherwise unrecognized as macro.
__FILE__	Expands to the current source file name.
__LINE__	Expands to the line number of the line where this macro is called.
__MISRAC_VERSION__	Expands to the MISRA C version used 1998, 2004 or 2012 (option **--misrac-version**). The default is 2004.

Macro	Description
__REVISION__	Expands to the revision number of the Safety Checker. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -> 1, v1.0rb -> -1

__SAFETY_CLASS_ACCESS_RIGHTS__	Identifies the safety class access rights table. It expands to:
	
	`__export__ static __used__ const struct`
	`{`
	`  __uint_least64_t safety_class_src;`
	`  __uint_least64_t safety_class_dst;`
	`  __uint_least64_t access_rights;`
	`} __safety_class_access_rights_table[] =`

__SAFETY_CLASS_ACCESS_RIGHTS_READ__	Identifies the safety class read access rights. Expands to 0x1.
__SAFETY_CLASS_ACCESS_RIGHTS_WRITE__	Identifies the safety class write access rights. Expands to 0x2.
__SAFETY_CLASS_ACCESS_RIGHTS_CALL__	Identifies the safety class call access rights. Expands to 0x4.

__SAFETY_CLASS_AREAS__	Identifies the safety class areas table. It expands to:
	
	`__export__ static __used__ const struct`
	`{`
	`  __uintptr_t address;`
	`  __uint_least64_t size;`
	`  __uint_least64_t safety_class;`
	`} __safety_class_areas_table[] =`

__SAFETY_CLASS_ATTRIBUTES__	Identifies the safety class attributes table. It expands to:
	
	`__export__ static __used__ const struct`
	`{`
	`  __uint_least64_t safety_class;`
	`  char * name;`
	`} __safety_class_attributes_table[] =`

__SAFETY_CLASS_SELECTIONS__	Identifies the safety class selections table. It expands to:
	
	`__export__ static __used__ const struct`
	`{`
	`  char * file_pattern;`
	`  char * name_pattern;`
	`  __uint_least64_t safety_class;`
	`} __safety_class_selections_table[] =`

__SINGLE_FP__	Expands to 1 if you used option **--fp-model=+float** (Treat 'double' as 'float'), otherwise unrecognized as macro.

Macro	Description
__STDC__	Identifies the level of ANSI standard. The macro expands to 1 if you set option **--language** (Control language extensions), otherwise expands to 0.
__STDC_HOSTED__	Always expands to 0, indicating the implementation is not a hosted implementation.
__STDC_VERSION__	Identifies the ISO-C version number. Expands to 199901L for ISO C99 or 199409L for ISO C90.
__TASKING__	Identifies the Safety Checker as a TASKING product. Expands to 1 if the Safety Checker by TASKING is used.
__TIME__	Expands to the compilation time: “hh:mm:ss”
__VERSION__	Identifies the version number of the Safety Checker. For example, if you use version 2.1r1 of the Safety Checker, __VERSION__ expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).

Example

```
#if __MISRAC_VERSION__ == 2012
/* this part is only valid for MISRA C:2012 */
...
#endif
```

3.6. Configuring the Safety Checker for Non-Standard C Code

The Safety Checker by TASKING fully supports the ISO-C standard, and additional generic language extensions as explained in the previous sections, but it does not add extra possibilities to program the special functions of a target.

It is required to configure the Safety Checker for non-standard C code. The Safety Checker generates a syntax error when a target specific feature is used.

Bit size

The bit size of the integer types must conform to the target used. You can use the bit size configuration options (**--bit-size-<type>**) to set the bit size of type `char`, `short`, `long`, `long long` and pointer. (e.g. **--bit-size-int=16**)

Non-standard keywords

All TASKING non-standard keywords have two leading underscores (__).

TASKING compiler language extensions

When you use the Safety Checker on C code with TASKING compiler C language extensions[1], configurations are needed as explained in the following sections.

[1] Other C compilers can have equivalent additions to the C language that require the same configurations.

3.6.1. Data Types

For non-standard data types you must define an ISO C99 data type with a bit size that is equal or larger than the non-standard data type and with a matching arithmetic. You can add a `typedef` declaration that is a synonym or add a macro that defines the keyword of the extra data type. For example:

```
typedef _Bool __bit;
```

or:

```
#define __bit _Bool
```

3.6.2. Data Type Qualifiers, Memory Qualifiers and Function Qualifiers

Non-standard data type qualifiers, memory qualifiers and function qualifiers are superfluous for the Safety Checker. You cannot use them in your C code. You can remove the keyword by adding an empty macro definition for it. For example:

```
#define __far
```

For qualifiers with arguments use macros with the same number of arguments or use variable arguments `"..."`. For example:

```
#define __interrupt(vector)
#define __vector_table(...)
```

3.6.3. Attributes

The attributes supported by the Safety Checker are listed in Section 3.3, *Attributes*. Attributes in non-standard C code can be used with or without the `__attribute__(())` keyword. You can remove the attribute by adding an empty macro definition. For example:

```
#define __align(n)
```

The Safety Checker issues a warning on unknown attributes in the form `__attribute__((name))` and ignores the attribute.

Attributes to specify alignment and absolute addresses are superfluous for the Safety Checker and cannot be used in your C code.

3.6.4. Intrinsic Functions

The Safety Checker does not support target specific functions that generate assembly instructions.

You can define function prototypes for these intrinsic functions. For example:

```
void __nop( void );
```

This is required to avoid warnings that are generated for the implicit declaration of intrinsic functions (**-w505**) and the warning for calling an intrinsic function without a prototype (**-w577**).

Without any function definition for the intrinsic function the Safety Checker generates in the report file that no further information is available. For example:

```
__nop() // No further information
```

Intrinsic functions that generate inline assembly instructions that do not access global data or special function registers cannot cause a safety violation. You can safely ignore the "No further information" informational message for these intrinsic functions.

When it is required to check the global data access from within an intrinsic function, you have to declare a function that mimics the data access of the intrinsic function. For example:

```
void __intrinsic_read_write_global( int * p ) { *p++; }
```

No safety class is defined for target specific intrinsic functions, they are marked as class 0, which is conform the default. When intrinsic functions are executed from functions which have restricted execute access you can assign a safety class for the intrinsic functions to allow execute access.

You have to add the target specific intrinsic function names to a safety class selection table of your application. The safety class values assigned to the intrinsic function should match the safety class access rights table of your application to allow execute access.

For example:

```
__SAFETY_CLASS_SELECTIONS__
{
    /*  File Mask  Name Mask    Class */
    {   "*",       "__nop",     INTRINSIC_NOP_CLASS }
};

__SAFETY_CLASS_ACCESS_RIGHTS__
{
    /*
     * An ASIL A object is allowed to execute the intrinsic
     * function __nop().
     */

    /*  Src     Dst                  Rights  */
    {   ASIL_A, INTRINSIC_NOP_CLASS, __SAFETY_CLASS_ACCESS_RIGHTS_CALL__ }
};
```

3.6.5. Pragmas

The pragmas to control the Safety Checker are defined in Section 3.4, *Pragmas to Control the Safety Checker*. All other non-STDC pragmas are ignored. The Safety Checker generates a diagnostic message `W509: ignored unrecognized "#pragma <name>"`. You can disable this warning with command line option **--no-warnings=509**.

3.6.6. Predefined Typedefs

The Safety Checker generates a syntax error when target specific typedefs are used. You should add `typedef` declarations that are a synonym for the target specific type. For example:

```
typedef unsigned long long int __ull;
```

3.6.7. Predefined Macros

You can add target specific predefined macros that have a fixed value to configure the non-standard C code. You can add these target specific macros in an include file or define them at the command line. For example:

```
#define __CTC__ 1
```

or

```
-D__CTC__=1
```

Target specific predefined macros that are derived from target specific options should be defined on the command line. For example the predefined macro name of a core `__CORE__`*`name`*`__` could be defined on the command line. For example:

```
-D__CORE_TC13__
```

3.6.8. Include Files

Target specific standard include files are not located in the Safety Checker's default `include` directory relative to the installation directory. You can use the command line option **--include-directory** to specify the target specific standard include directory. For example:

```
--include-directory=$(PRODDIR)\include
```

3.6.9. Assembly Instructions in the C Source

With the TASKING keyword `__asm()` you can use assembly instructions in the C source and pass C variables as operands to the assembly code. The `__asm()` syntax is supported, but the Safety Checker does not check assembly blocks; they are regarded as a black box. When the assembly blocks have no side effects, and will not access global data, no checking is required.

When it is required to check the global data access from within an assembly block, you have to replace it with equivalent C code.

For example:

```
#if __CSAF__
    write = read;
#else
__asm( "ld.w d15,read\n"
       "st.w write,d15\n" );
#endif
```

You can also replace or remove non-TASKING compliant inline assembly blocks with the predefined macro __CSAF__.

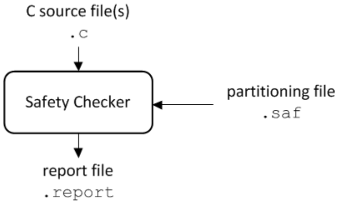
3.6.10. Target Configuration

The Safety Checker support website contains configuration files for several compilers. You can configure the Safety Checker for a compiler by including the configuration file at the beginning of each C source file, before other includes with the option **--include-file**.

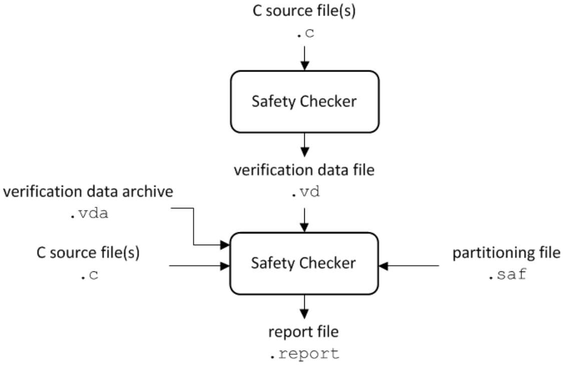
Chapter 4. Using the Safety Checker

This chapter describes the Safety Checker phases and explains how to call the Safety Checker.

The Safety Checker accepts one or more C files as input and a file containing safety class definitions and safety class access rights. The output is a report file.



In a more advanced flow, checking can be done in two runs. An intermediate verification data file (`.vd`) is created in the first run. This is the input for the second run. The intermediate files can be stored in an archive, which is also possible as input to the Safety Checker.



4.1. Safety Checker Phases

During the analysis of a C program, the Safety Checker runs through a number of phases.

1. Preprocessor phase:

   The preprocessor includes files and substitutes macros by C source. It uses only string manipulations on the C source. The syntax for the preprocessor is independent of the C syntax but is also described in the ISO/IEC 9899:1999(E) standard.

2. Static analysis phase:

   The scanner converts the preprocessor output to a stream of tokens which are fed to a parser for the C grammar. The parser performs a syntactic and semantic analysis of the program and generates an intermediate representation of the program.

3. Partitioning checking phase:

   The intermediate (verification) data is used for partitioning checking and a Safety Checker report file is generated.

4.2. Invocation Syntax

You can call the Safety Checker from the command line. The invocation syntax is:

```
csaf [ [option]... [file]... ]...
```

file can be a C source file (`.c`), verification data file (`.vd`), VD archive (`.vda`) or safety class partitioning file (`.saf`).

You can find a detailed description of all Safety Checker options in Section 7.2, *Safety Checker Options*.

4.3. Verification Data

The following entities and actions on entities can be present in C source files:

Entity action	Global variable / function	Argument	Function return value	Local variable	Fixed address
read / write / call	1.1	1.2	1.3	Always allowed	1.5
pass as parameter to a function	2.1	2.2	2.3	2.4	2.5
return	3.1	3.2	3.3	Illegal	3.5

The numbers refer to the following cases:

Case	Description
1.1	Access (read/write/call) of a variable/function
1.2	Access (read/write/call) of an argument
1.3	Access (read/write/call) of a function return value
1.5	Access (read/write/call) of a fixed address
2.1	Pass (the address of) a variable/function as parameter to a function
2.2	Pass an argument as parameter to a function
2.3	Pass a function return value as parameter to a function
2.4	Pass (the address of) a variable with local lifetime as parameter to a function
2.5	Pass a fixed address as parameter to a function
3.1	Return (the address of) a variable/function
3.2	Return an argument
3.3	Return a function return value

Case	Description
3.5	Return a fixed address

Variable may represent a basic type (like `int`), a structure (member) or the element of an array.

For example (`safety.c`):

```
int x;  // ASIL A (safety class 3)
int y;  // ASIL A (safety class 3)
int z;  // ASIL A (safety class 3)

int * f1(void)  // safety class 7
{
  return &z;    // case 3.1 - No safety issue.
}

int f2(int *n)  // safety class 7
{
  x = 0;        // case 1.1 - Write access for x can be checked with a safety class.
  *f1() = 0;    // case 1.3 - Write access for z can be checked with a safety class.
  return *n;    // case 1.2 - Read access for x, y, and z can be checked with a safety class.
}

void f3(int *n) // safety class 7
{
  f2(&y);       // case 2.1 - Call access can be checked with a safety class.
  f2(n);        // case 2.2 - Call access can be checked with a safety class.
  f2(f1());     // case 2.3 - Call access can be checked with a safety class.
}

void main(void)
{
  f3(&x);
}
```

4.4. Define Partitioning Information

Safety class access rights

With safety class access rights you define the access allowed between safety class areas. You can specify the access rights table as an initialized predefined C structure and you can insert this structure anywhere in the source code. Most obvious you will store it in a separate file with a specific name, for example `partitioning.saf`. This file is simply passed to the Safety Checker just as if it was an ordinary source file.

You can use the following predefined macros as access rights:

```
__SAFETY_CLASS_ACCESS_RIGHTS_READ__    // read access
__SAFETY_CLASS_ACCESS_RIGHTS_WRITE__   // write access
__SAFETY_CLASS_ACCESS_RIGHTS_CALL__    // call access
```

For example:

```
#define ASIL_A 3   //
#define ASIL_B 4
#define ASIL_D 7

__SAFETY_CLASS_ACCESS_RIGHTS__
{
    // safety class '7' only has read access in safety class '3'

    // Src      Dst      Rights
    {  ASIL_D,  ASIL_A,  __SAFETY_CLASS_ACCESS_RIGHTS_READ__ }
};
```

Note that you can combine access rights with the 'binary OR' operator (|).

Safety class selections

With the safety class selection table you specify which part of the code belongs to which safety class. If you do not specify a class, class 0 is the default. The safety class numbers are just arbitrary numbers. It is up to you to assign a particular number to an (A)SIL level. The safety class selection table is also an initialized predefined C structure. For example,

```
__SAFETY_CLASS_SELECTIONS__
{
    // f1(), f2() and f3() are part of safety class 7
    // x, y and z are part of safety class 3
    // everything else is part of safety class 0

    // File Mask     Name Mask   Class
    {  "safety.c",   "f*",       ASIL_D     },
    {  "safety.c",   "x",        ASIL_A     },
    {  "safety.c",   "y",        ASIL_A     },
    {  "safety.c",   "z",        ASIL_A     }
};

__SAFETY_CLASS_AREAS__
{
    // Address    Size      Class
    {  0x8004000, 0x200,    ASIL_B     },
    {  0x8008000, 0x100,    ASIL_B     }
};
```

Safety class areas

With the safety class areas table you specify which memory areas belongs to which safety class. Those areas are used to check violations of direct memory access, i.e. in case of SFR access. This table is also an initialized predefined C structure. For example:

```
__SAFETY_CLASS_AREAS__
{
    // Address    Size      Class
    {  0x8004000, 0x200,    ASIL_B     },
    {  0x8008000, 0x100,    ASIL_B     }
};
```

Safety class attributes

With the safety class attributes table you can define other names for the default class numbers. The names are used in the report file and in diagnostic messages. For example,

```
__SAFETY_CLASS_ATTRIBUTES__
{
    // Class    Name
    {  3,       "ASIL A"    },
    {  4,       "ASIL B"    },
    {  7,       "ASIL D"    }
};
```

4.5. Output of the Safety Checker

This section contains an example output of the Safety Checker. The Safety Checker analyzes the application for possible violations of safety restrictions and generates a report file. You can redirect any errors and warnings to a file.

For example,

```
csaf safety.c partitioning.saf --error-file=safety.err --keep-output-files --output-format=2
```

Errors and warnings are sent to `safety.err`.

```
csaf E497: ["safety.c" 12] safety violation writing "x" (ASIL A) from "f2" (ASIL D)
csaf E497: ["safety.c" 13] safety violation writing "z" (ASIL A) from "f2" (ASIL D)
csaf I896: ["safety.c" 7] the address of "z" is returned by function "f1" to function "f2"
csaf E499: ["safety.c" 26] safety violation calling "f3" (ASIL D) from "main" (class 0)
3 errors, 0 warnings
```

The output of the Safety Checker is sent to `safety.report`. If there are errors the output file is removed, unless you specify option **--keep-output-files**.

```
* Safety Class Areas
=====================
+------------------------------------------+
| Address  |    Size | Class        | Used |
|==========================================|
| 08004000 |     512 | ASIL B       | No   |
| 08008000 |     256 | ASIL B       | No   |
+------------------------------------------+


* Safety Class Selections
==========================

+------------------------------------------------------------------------+
| File pattern              | Name pattern          | Class         | Used |
|========================================================================|
| safety.c                  | f*                    | ASIL D        | Yes  |
| safety.c                  | x                     | ASIL A        | Yes  |
| safety.c                  | y                     | ASIL A        | Yes  |
| safety.c                  | z                     | ASIL A        | Yes  |
| *                         | *                     | 0             | Yes  |
+------------------------------------------------------------------------+


* Safety Class Access Rights Table
===================================

      Target:    0      ASIL A ASIL B ASIL D
              +-----------------------------
Source: 0     |  rwx    ---    ---    ---
        ASIL A|  ---    rwx    ---    ---
        ASIL B|  ---    ---    rwx    ---
        ASIL D|  ---    r--    ---    rwx

* Call/Data Graph
==================

         +--f1() -> &z
         |   :
         |   :.. [act] Returns static variable "z"
         |
         |   +--f1() -> &z
         |   |   :
         |   |   :.. [act] Returns static variable "z"
         |   |
         +--f2(&y/&x/&z)
         |   :
         |   :.. [act] Reads from parameter #1
         |   :.. [chk] Check access for reading "y" (ASIL A) from "f2" (ASIL D)
         |   :.. [chk] Check access for reading "x" (ASIL A) from "f2" (ASIL D)
         |   :.. [chk] Check access for reading "z" (ASIL A) from "f2" (ASIL D)
         |   :.. [act] Writes to static variable "x"
         |   :.. [chk] Check access for writing "x" (ASIL A) from "f2" (ASIL D)
                       // ACCESS VIOLATION!
         |   :.. [act] Writes to the return value of function "f1"
         |   :.. [chk] Check access for writing "z" (ASIL A) from "f2" (ASIL D)
                       // ACCESS VIOLATION!
         |   :.. [act] Calls function "f1"
         |   :.. [chk] Check access for calling "f1" (ASIL D) from "f2" (ASIL D)
         |
     +--f3(&x)
     |   :
     |   :.. [act] Calls function "f1"
     |   :.. [chk] Check access for calling "f1" (ASIL D) from "f3" (ASIL D)
     |   :.. [act] Calls function "f2"
     |   :.. [chk] Check access for calling "f2" (ASIL D) from "f3" (ASIL D)
     |   :.. [act] Passes static variable "y" as parameter #1 to function "f2"
     |   :.. [act] Passes parameter #1 as parameter #1 to function "f2"
     |   :.. [act] Passes the return value of function "f1" as parameter #1 to function "f2"
     |
 *--main()
     :
     :.. [act] Calls function "f3"
     :.. [chk] Check access for calling "f3" (ASIL D) from "main" (class 0)
               // ACCESS VIOLATION!
     :.. [act] Passes static variable "x" as parameter #1 to function "f3"


* Access table (sorted on address)
===================================

+------------------------------------------------------------------------------------------------+
| Address  | Name                          | Acting function               | Address  | Access |
|================================================================================================|
| xxxxxxxx | f1                            | f2                            | xxxxxxxx | --CA   |
| xxxxxxxx | f1                            | f3                            | xxxxxxxx | --CA   |
| xxxxxxxx | f2                            | f3                            | xxxxxxxx | --CA   |
| xxxxxxxx | f3                            | main                          | xxxxxxxx | --CA   |
| xxxxxxxx | x                             | f2                            | xxxxxxxx | RW-A   |
| xxxxxxxx | x                             | main                          | xxxxxxxx | ---A   |
| xxxxxxxx | y                             | f2                            | xxxxxxxx | R---   |
| xxxxxxxx | y                             | f3                            | xxxxxxxx | ---A   |
| xxxxxxxx | z                             | f1                            | xxxxxxxx | ---A   |
| xxxxxxxx | z                             | f2                            | xxxxxxxx | RW--   |
+------------------------------------------------------------------------------------------------+
Access: R=read, W=write, C=call and A=address taken

* MISRA C report
=================


Modules compiled with MISRA C:2004 checks:
```

```
-----------------------------------------

                                             1     1                2           2
1    2   3     4 5      6    78           9  0     9                0           1
.    .   .     . .      .    ..           .  .     .                .           .
123451234123456121234567123451123456789012123123456123456789012345671234567890121

........................X....................X................................... TOTAL

........................X....................X................................... partitioning.saf
........................X....................X................................... safety.c


Modules compiled without MISRA C checks:
----------------------------------------

None
```

Notes on the output:

- A '-' in the Safety Class Access Rights Table means 'no rights'.
- `[act]` in the Call/Data Graph means an action.
- `[chk]` in the Call/Data Graph means a check.
- The X in the MISRA C report indicates that the MISRA C check for that rule was enabled for the specified module. The `TOTAL` row shows the rules that were enabled for all modules. Note that some rules are not displayed in this example output to fit on the page.

4.6. Safety Checker Remarks

- The Safety checker cannot keep track of addresses stored in (global) variables. For example,

  ```
  int *my_global_ptr;
  void foo( void )
  {
      my_global_ptr = &my_global_int;
      *my_global_ptr = 3;
  }
  ```

  This is also true in case of global data initialization:

  ```
  int *my_global_ptr = &my_global_int;
  void foo( void )
  {
      *my_global_ptr = 3;
  }
  ```

- Passing the address of an object as argument to a function is supported:

  ```
  int my_global_int;
  void foo( void )
  {
      bar( &my_global_int );
  }
  ```

  Those parameters can also be passed to another function:

  ```
  int my_global_int;
  void foo( int *my_par_int )
  {
      bar( my_par_int );
  }
  ```

The Call/Data Graph of the output report shows how addresses of objects 'flows' through the application.

4.7. How the Safety Checker Searches Include Files

When you use include files (with the `#include` statement), you can specify their location in several ways. The Safety Checker searches the specified locations in the following order:

1. If the `#include` statement contains an absolute pathname, the Safety Checker looks for this file. If no path or a relative path is specified, the Safety Checker looks in the same directory as the source file. This is only possible for include files that are enclosed in "".

   > This first step is not done for include files enclosed in <>.

2. When the Safety Checker did not find the include file, it looks in the directories that are specified with option **--include-directory (-I)**).

3. When the Safety Checker did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable `CSAFINC`.

4. When the Safety Checker still did not find the include file, it finally tries the default include directory relative to the installation directory (unless you specified option **--no-stdinc**).

Example

Suppose that the C source file `test.c` contains the following lines:

```
#include <stdio.h>
#include "myinc.h"
```

You can call the Safety Checker as follows:

```
csaf -Imyinclude test.c
```

First the Safety Checker looks for the file `stdio.h` in the directory `myinclude` relative to the current directory. If it was not found, the Safety Checker searches in the environment variable `CSAFINC` and then in the default `include` directory.

The Safety Checker now looks for the file `myinc.h`, in the directory where `test.c` is located. If the file is not there the Safety Checker searches in the directory `myinclude`. If it was still not found, the Safety Checker searches in the environment variable `CSAFINC` and then in the default `include` directory.

4.8. Static Code Analysis

Various approaches and algorithms exist to perform static code analysis (SCA), each having specific pros and cons.

SCA Implementation Design Philosophy

SCA is implemented in the Safety Checker based on the following design criteria:

- An SCA phase does not take up an excessive amount of execution time. Therefore, the SCA can be performed during a normal edit-analyze cycle.
- SCA is implemented in the parser phase. Therefore, no new makefiles or work procedures have to be developed to perform SCA.
- The number of emitted false positives is kept to a minimum. A false positive is a message that indicates that a correct code fragment contains a violation of a rule/recommendation. A number of warnings is issued in two variants, one variant when it is *guaranteed* that the rule is violated when the code is executed, and the other variant when the rule is *potentially* violated, as indicated by a preceding warning message.

  For example see the following code fragment:

  ```
  extern int some_condition(int);
  void f(void)
  {
      char buf[10];
      int i;

      for (i = 0; i <= 10; i++)
      {
          if (some_condition(i))
          {
              buf[i] = 0; /* subscript may be out of bounds */
          }
      }
  }
  ```

  As you can see in this example, if `i=10` the array `buf[]` might be accessed beyond its upper boundary, depending on the result of `some_condition(i)`. If the Safety Checker cannot determine the result of this function at run-time, the Safety Checker issues the warning "subscript is *possibly* out of bounds" preceding the CERT warning "ARR35: do not allow loops to iterate beyond the end of an array". If the Safety Checker can determine the result, or if the `if` statement is omitted, the Safety Checker can guarantee that the "subscript is out of bounds".

- The SCA implementation has real practical value in embedded system development. There are no real objective criteria to measure this claim. Therefore, the Safety Checker supports well known standards for safety critical software development such as the MISRA guidelines for creating software for safety critical automotive systems and secure "CERT C Secure Coding Standard" released by CERT. CERT is founded by the US government and studies internet and networked systems security vulnerabilities, and develops information to improve security.

4.8.1. C Code Checking: CERT C

The CERT C Secure Coding Standard provides rules and recommendations for secure coding in the C programming language. The goal of these rules and recommendations is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. The application of the secure coding standard will lead to higher-quality systems that are robust and more resistant to attack.

For details about the standard, see the CERT C Secure Coding Standard web site. For general information about CERT secure coding, see www.cert.org/secure-coding.

Versions of the CERT C standard

Version 1.0 of the CERT C Secure Coding Standard is available as a book by Robert C. Seacord [Addison-Wesley]. Whereas the web site is a wiki and reflects the latest information, the book serves as a fixed point of reference for the development of compliant applications and source code analysis tools.

The rules and recommendations supported by the Safety Checker reflect the version of the CERT web site as of June 1 2009.

The following rules/recommendations implemented by the Safety Checker, are not part of the book: PRE11-C, FLP35-C, FLP36-C, MSC32-C

For a complete overview of the supported CERT C recommendations/rules by the Safety Checker, see Chapter 8, *CERT C Secure Coding Standard*.

Priority and Levels of CERT C

Each CERT C rule and recommendation has an assigned *priority*. Three values are assigned for each rule on a scale of 1 to 3 for

- severity - how serious are the consequences of the rule being ignored

  1. low (denial-of-service attack, abnormal termination)
  2. medium (data integrity violation, unintentional information disclosure)
  3. high (run arbitrary code)

- likelihood - how likely is it that a flaw introduced by ignoring the rule could lead to an exploitable vulnerability

1. unlikely
2. probable
3. likely

- remediation cost - how expensive is it to comply with the rule

  1. high (manual detection and correction)
  2. medium (automatic detection and manual correction)
  3. low (automatic detection and correction)

The three values are then multiplied together for each rule. This product provides a measure that can be used in prioritizing the application of the rules. These products range from 1 to 27. Rules and recommendations with a priority in the range of 1-4 are level 3 rules (low severity, unlikely, expensive to repair flaws), 6-9 are level 2 (medium severity, probable, medium cost to repair flaws), and 12-27 are level 1 (high severity, likely, inexpensive to repair flaws).

The Safety Checker checks most of the level 1 and some of the level 2 CERT C recommendations/rules.

For a complete overview of the supported CERT C recommendations/rules by the Safety Checker, see Chapter 8, *CERT C Secure Coding Standard*.

To apply CERT C code checking to your application

On the command line use option **--cert**.

```
csaf --cert={all | name [-name],...]
```

With **--diag=cert** you can see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, **--diag=pre** lists all supported checks in the preprocessor category.

4.8.2. C Code Checking: MISRA C

The C programming language is a standard for high level language programming in embedded systems, yet it is considered somewhat unsuitable for programming safety-related applications. Through enhanced code checking and strict enforcement of best practice programming rules, TASKING MISRA C code checking helps you to produce more robust code.

MISRA C specifies a subset of the C programming language which is intended to be suitable for embedded automotive systems. It consists of a set of rules, defined in *MISRA C:2004, Guidelines for the Use of the C Language in Critical Systems* (Motor Industry Research Association (MIRA), 2004).

The Safety Checker also supports MISRA C:1998, the first version of MISRA C and MISRA C: 2012, the latest version of MISRA C. You can select the version with the following option:

```
--misrac-version=1998
--misrac-version=2004
--misrac-version=2012
```

In your C source files you can check against the MISRA C version used. For example:

```
#if __MISRAC_VERSION__ == 1998
  ...
#elif __MISRAC_VERSION__ == 2004
  ...
#elif __MISRAC_VERSION__ == 2012
  ...
#endif
```

For a complete overview of all MISRA C rules, see Chapter 9, *MISRA C Rules*.

Implementation issues

The MISRA C implementation in the Safety Checker supports nearly all rules. Only a few rules are not supported because they address documentation, run-time behavior, or other issues that cannot be checked by static source code inspection, or because they require an application-wide overview.

During analysis of the code, violations of the enabled MISRA C rules are indicated with error messages.

MISRA C rules are divided in mandatory rules, required rules and advisory rules. If rules are violated, errors are generated causing the Safety Checker to stop. With the following options warnings, instead of errors, are generated:

```
--misrac-mandatory-warnings
--misrac-required-warnings
--misrac-advisory-warnings
```

> Note that not all MISRA C violations will be reported when other errors are detected in the input source. For instance, when there is a syntax error, all semantic checks will be skipped, including some of the MISRA C checks.

Quality Assurance report

To ensure compliance to the MISRA C rules throughout the entire project, the Safety Cheker can generate a MISRA C Quality Assurance report. This report lists the various modules in the project with the respective MISRA C settings at the time of analyzing. You can use this in your company's quality assurance system to provide proof that company rules for best practice programming have been applied in the particular project.

To apply MISRA C code checking to your application

On the command line use option **--misrac**.

```
csaf --misrac={all | number [-number],...]
```

4.9. Safety Checker Error Messages

The Safety Checker reports the following types of error messages.

F (Fatal errors)

After a fatal error the Safety Checker immediately aborts.

E (Errors)

Errors are reported, but the Safety Checker continues analysis. No output files are produced unless you have set the Safety Checker option **--keep-output-files** (the resulting output file may be incomplete).

W (Warnings)

Warning messages do not result into an erroneous output. They are meant to draw your attention to possible vulnerabilities which requires special attention. You can control warnings with Safety Checker option **--no-warnings**.

I (Information)

Information messages are always preceded by an error message. Information messages give extra information about the error.

S (System errors)

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

```
S9##: internal consistency check failed - please report
```

please report the error number and as many details as possible about the context in which the error occurred.

Display detailed information on diagnostics

On the command line you can use the Safety Checker option **--diag** to see an explanation of a diagnostic message:

```
csaf --diag=[format:]{all | number,...]
```

Chapter 5. Tutorial

The `examples\partitioning` directory of the Safety Checker installation contains several examples of partitioning analysis of the Safety Checker. They serve as a good starting point for your own safety critical project.

- `basic` - A basic example using functions and global variables with various safety configurations.
- `rebel` - A rebel module tries to access critical information, but is caught by the partitioning analysis of the Safety Checker.
- `sfr` - A simple example showing how safety classes can be applied to memory-mapped Special Function Registers (SFRs).
- `ivar` - A basic example using functions passing the address of a global variable and disallowing access to that variable.
- `istruct` - A basic example using functions passing the address of a `struct` member and disallowing access to that `struct` member.

In this tutorial we will use the `basic` example to go through the process of preparing your project for safety checking. After this `basic` example, you can inspect the other examples.

Step 1: Write the source files

The source files delivered with the `basic` example are:

- `module1.c`

  ```
  #include "globdef.h"

  int my_global;
  ```

- `module2.c`

  ```
  #include "globdef.h"

  int f2( void )
  {
      return my_global;
  }
  ```

- `module3.c`

  ```
  #include "globdef.h"

  void f3( void )
  {
  ```

```
    my_global = 1;
}
```

- `module4.c`

  ```
  #include "globdef.h"

  void f4( void )
  {
      f3();
  }
  ```

- `globdef.h`

  ```
  extern int my_global;

  extern int f2( void );
  extern void f3( void );
  extern void f4( void );
  ```

Step 2: Define the safety classes

Several safety class configuration files are delivered with the `basic` example:

- `none.saf`

  This is an empty file which triggers the default settings. Use this to get your code working without having to worry about safety classes.

- `default.saf`

  This file contains safety class settings that are equivalent to the default settings. This file has the same purpose as `none.saf`.

- `minimal.saf`

  Each module gets a different safety class. The minimum set of access rights has been defined so that the Safety Checker approves the program.

- `maximal.saf`

  Each module gets a different safety class. The maximum set of access rights has been defined so that every possible access is allowed. This is just for illustration purposes. If you want to allow everything it is easier to just use `none.saf`.

We will use the file `minimal.saf` in this tutorial:

```
/*
 * Minimal settings using different safety classes
 */
```

```
#define R  (__SAFETY_CLASS_ACCESS_RIGHTS_READ__)
#define W  (__SAFETY_CLASS_ACCESS_RIGHTS_WRITE__)
#define X  (__SAFETY_CLASS_ACCESS_RIGHTS_CALL__)

#define QM     (0U)
#define ASIL_A (1U)
#define ASIL_B (2U)
#define ASIL_C (3U)
#define ASIL_D (4U)

/*

  Safety Class Access Rights Table
===================================

    Target:    0    1    2    3    4
            +-------------------------
Source: 0   |  ---  ---  ---  ---  ---
        1   |  ---  rwx  --x  ---  ---
        2   |  ---  ---  rwx  ---  -w-
        3   |  ---  ---  ---  rwx  r--
        4   |  ---  ---  ---  ---  rwx
*/

/*
 * Minimum rights to get example to work.
 */
__SAFETY_CLASS_ACCESS_RIGHTS__
{
    /* Src     Dst     Rights  */

    {  QM,     QM,     0U    }, /* Zero, as this is not used. */
    {  ASIL_B, ASIL_D, W     }, /* An ASIL B object is allowed to write an ASIL D object */
    {  ASIL_A, ASIL_B, X     }, /* An ASIL A object is allowed to execute an ASIL B object */
    {  ASIL_C, ASIL_D, R     }  /* An ASIL C object is allowed to read an ASIL D object */
};


/*
** Safety Class selections:
*/

__SAFETY_CLASS_SELECTIONS__
{
    /* File Mask     Name Mask  Class */

    {  "module1.c",  "*",       ASIL_D  },
    {  "module2.c",  "*",       ASIL_C  },
    {  "module3.c",  "*",       ASIL_B  },
    {  "module4.c",   "*",       ASIL_A  }
};

#if 0 /* No safety class areas defined for this example */

/*
** Safety Class areas:
*/

__SAFETY_CLASS_AREAS__
{
 /* Address    Size    Class */
};

#endif
```

Note that in this example ASIL A is assigned to safety class 1, ASIL B to safety class 2, etc. But you can use any number you want.

Step 3: Call the Safety Checker

You can call the Safety Checker in a separate invocation or you can use a `makefile`. The makefiles delivered in the `examples` directory work with both **amk** (delivered with the Safety Checker) and GNU make.

Call the Safety Checker by hand :

```
install_dir\bin\csaf --output-format=2 -k -o minimal.report module1.c
                     module2.c module3.c module4.c minimal.saf
```

This invocation creates the report file `minimal.report` in one call.

Alternatively, call the make utility **amk**:

```
install_dir\bin\amk minimal.report_passed
```

This call to **amk** results in the following invocations:

```
install_dir\bin\csaf --vd --dep-file=module1.vd.d --make-target=module1.vd -o module1.vd module1.c
install_dir\bin\csaf --vd --dep-file=module2.vd.d --make-target=module2.vd -o module2.vd module1.c
install_dir\bin\csaf --vd --dep-file=module3.vd.d --make-target=module3.vd -o module3.vd module1.c
install_dir\bin\csaf --vd --dep-file=module4.vd.d --make-target=module4.vd -o module4.vd module1.c
install_dir\bin\csaf --output-format=2 -k --dep-file=minimal.report_passed.d
                     --make-target=minimal.report -o minimal.report minimal.saf
                     module1.vd module2.vd module3.vd module4.vd
```

This invocation uses verification data files (`.vd`) as intermediate results and dependency files for the make utility. This invocation also creates the report file `minimal.report`. All options are explained in Section 7.2, *Safety Checker Options*.

Step 4: Inspect the report file

The report file `minimal.report` contains an overview of the safety class selections, the safety class access rights and a call/data graph with remarks on which checks have been performed.

```
* Safety Class Selections
==========================

+------------------------------------------------------------------------+
| File pattern               | Name pattern          | Class          | Used |
|========================================================================|
| module1.c                  | *                     | 4              | Yes  |
| module2.c                  | *                     | 3              | Yes  |
| module3.c                  | *                     | 2              | Yes  |
| module4.c                  | *                     | 1              | Yes  |
+------------------------------------------------------------------------+


* Safety Class Access Rights Table
===================================

    Target:    0    1    2    3    4
            +--------------------------
Source: 0   |  ---  ---  ---  ---  ---
        1   |  ---  rwx  --x  ---  ---
        2   |  ---  ---  rwx  ---  -w-
        3   |  ---  ---  ---  rwx  r--
        4   |  ---  ---  ---  ---  rwx


* Call/Data Graph
==================

 *--f2()
     :
     :.. [act] Reads from static variable "my_global"
     :.. [chk] Check access for reading "my_global" (class 4) from "f2" (class 3)


     +--f3()
     |   :
     |   :.. [act] Writes to static variable "my_global"
     |   :.. [chk] Check access for writing "my_global" (class 4) from "f3" (class 2)
     |
 *--f4()
     :
     :.. [act] Calls function "f3"
     :.. [chk] Check access for calling "f3" (class 2) from "f4" (class 1)
```

```
* Access table (sorted on address)
===================================

+------------------------------------------------------------------------------------------------+
| Address  | Name                           | Acting function                | Address  | Access |
|================================================================================================|
| xxxxxxxx | my_global                      | f2                             | xxxxxxxx | R--A   |
| xxxxxxxx | my_global                      | f3                             | xxxxxxxx | -W-A   |
| xxxxxxxx | f3                             | f4                             | xxxxxxxx | --CA   |
+------------------------------------------------------------------------------------------------+
Access: R=read, W=write, C=call and A=address taken
```

Chapter 6. Using the Utilities

The Safety Checker by TASKING comes with a number of utilities:

amk	A make utility to maintain, update, and reconstruct groups of programs. The make utility looks whether files are out of date, rebuilds them and determines which other files as a consequence also need to be rebuilt. It supports parallelism which utilizes the multiple cores found on modern host hardware.
arsaf	An archiver. With this utility you create and maintain library files with verification data files (`.vd`) generated by the Safety Checker.

6.1. Make Utility amk

amk is a make utility that you can use to maintain, update, and reconstruct groups of programs. **amk** features parallelism which utilizes the multiple cores found on modern host hardware, hardening for path names with embedded white space and it has an (internal) interface to provide progress information for updating a progress bar. It does not use an external command shell (`/bin/sh`, `cmd.exe`) but executes commands directly.

The primary purpose of any make utility is to speed up the edit-build-test cycle. To avoid having to build everything from scratch even when only one source file changes, it is necessary to describe dependencies between source files and output files and the commands needed for updating the output files. This is done in a so called "makefile".

6.1.1. Makefile Rules

A makefile dependency rule is a single line of the form:

```
[target ...] : [prerequisite ...]
```

where *target* and *prerequisite* are path names to files. Example:

```
test.report : test.c
```

This states that target `test.report` depends on prerequisite `test.c`. So, whenever the latter is modified the first must be updated. Dependencies accumulate: prerequisites and targets can be mentioned in multiple dependency rules (circular dependencies are not allowed however). The command(s) for updating a target when any of its prerequisites have been modified must be specified with leading white space after any of the dependency rule(s) for the target in question. Example:

```
test.report :
  csaf test.c   # leading white space
```

Command rules may contain dependencies too. Combining the above for example yields:

```
test.report : test.c
  csaf test.c
```

White space around the colon is not required. When a path name contains special characters such as '**:**', '**#**' (start of comment), '**=**' (macro assignment) or any white space, then the path name must be enclosed in single or double quotes. Quoted strings can contain anything except the quote character itself and a newline. Two strings without white space in between are interpreted as one, so it is possible to embed single and double quotes themselves by switching the quote character.

When a target does not exist, its modification time is assumed to be very old. So, **amk** will try to make it. When a prerequisite does not exist possibly after having tried to make it, it is assumed to be very new. So, the update commands for the current target will be executed in that case. **amk** will only try to make targets which are specified on the command line. The default target is the first target in the makefile which does not start with a dot.

Static pattern rules

Static pattern rules are rules which specify multiple targets and construct the prerequisite names for each target based on the target name.

```
[target ...] : target-pattern : [prerequisite-patterns ...]
```

The *target* specifies the targets the rules applies to. The *target-pattern* and *prerequisite-patterns* specify how to compute the prerequisites of each target. Each target is matched against the *target-pattern* to extract a part of the target name, called the *stem*. This stem is substituted into each of the *prerequisite-patterns* to make the prerequisite names (one from each *prerequisite-pattern*).

Each pattern normally contains the character '%' just once. When the *target-pattern* matches a target, the '%' can match any part of the target name; this part is called the *stem*. The rest of the pattern must match exactly. For example, the target `foo.report` matches the pattern '`%.report`', with '`foo`' as the stem. The target `foo.c` does not match that pattern.

The prerequisite names for each target are made by substituting the stem for the '%' in each prerequisite pattern.

Example:

```
reports = test.report filter.report

all: $(reports)

$(reports): %.report: %.c
    csaf $< -o $@
    echo the stem is $*
```

Here '$<' is the automatic variable that holds the name of the prerequisite, '$@' is the automatic variable that holds the name of the target and '`$*`' is the stem that matches the pattern. Internally this translates to the following two rules:

```
test.report: test.c
    csaf test.c -o test.report
    echo the stem is test

filter.report: filter.c
```

```
    csaf filter.c -o filter.report
    echo the stem is filter
```

Each target specified must match the target pattern; a warning is issued for each target that does not.

Special targets

There are a number of special targets. Their names begin with a period.

Target	Description
`.DONE`	When the make utility has finished building the specified targets, it continues with the rules following this target.
`.INIT`	The rules following this target are executed before any other targets are built.

`.PHONY`	The prerequisites of this target are considered to be phony targets. A phony target is a target that is not really the name of a file. The rules following a phony target are executed unconditionally, regardless of whether a file with that name exists or what its last-modification time is.
	For example:
	`.PHONY: clean`
	`clean:`
	`        rm *.report`
	With `amk clean`, the command is executed regardless of whether there is a file named `clean`.

6.1.2. Makefile Directives

Directives inside makefiles are executed while reading the makefile. When a line starts with the word "`include`" or "`-include`" then the remaining arguments on that line are considered filenames whose contents are to be inserted at the current line. "`-include`" will silently skip files which are not present. You can include several files. Include files may be nested.

Example:

```
include makefile2 makefile3
```

White spaces (tabs or spaces) in front of the directive are allowed.

6.1.3. Macro Definitions

A *macro* is a symbol name that is replaced with its definition before the makefile is executed. Although the macro name can consist of lowercase or uppercase characters, uppercase is an accepted convention. When a line does not start with white space and contains the assignment operator '**=**', '**:=**' or '**+=**' then the line is interpreted as a macro definition. White space around the assignment operator and white space at the end of the line is discarded. Single character macro evaluation happens by prefixing the name with '**$**'. To evaluate macros with names longer than one character put the name between parentheses '**()**' or curly braces '**{}**'. Macro names may contain anything, even white space or other macro evaluations. Example:

```
DINNER = $(FOOD) and $(BEVERAGE)
FOOD = pizza
BEVERAGE = sparkling water
FOOD += with cheese
```

With the **+=** operator you can add a string to an existing macro. An extra space is inserted before the added string automatically.

Macros are evaluated recursively. Whenever `$(DINNER)` or `${DINNER}` is mentioned after the above, it will be replaced by the text "`pizza with cheese and sparkling water`". The left hand side in a macro definition is evaluated before the definition takes place. Right hand side evaluation depends on the assignment operator:

=	Evaluate the macro at the moment it is used.
:=	Evaluate the replacement text before defining the macro.

Subsequent '**+=**' assignments will inherit the evaluation behavior from the previous assignment. If there is none, then '**+=**' is the same as '**=**'. The default value for any macro is taken from the environment. Macro definitions inside the makefile overrule environment variables. Macro definitions on the **amk** command line will be evaluated first and overrule definitions inside the makefile.

Predefined macros

Macro	Description
$	This macro translates to a dollar sign. Thus you can use "$$" in the makefile to represent a single "$".
@	The name of the current target. When a rule has multiple targets, then it is the name of the target that caused the rule commands to be run.
*	The basename (or stem) of the current target. The stem is either provided via a static pattern rule or is calculated by removing all characters found after and including the last dot in the current target name. If the target name is '`test.c`' then the stem is '`test`' (if the target was not created via a static pattern rule).
<	The name of the first prerequisite.
MAKE	The **amk** path name (quoted if necessary). Optionally followed by the options **-n** and **-s**.
ORIGIN	The name of the directory where **amk** is installed (quoted if necessary).
SUBDIR	The argument of option **-G**. If you have nested makes with **-G** options, the paths are combined. This macro is defined in the environment (i.e. default macro value).

The @, * and < macros may be suffixed by '**D**' to specify the directory component or by '**F**' to specify the filename component. `$(@D)` evaluates to the directory name holding the file`$(@F)`. `$(@D)/$(@F)` is equivalent to $@. Note that on MS-Windows most programs accept forward slashes, even for UNC path names.

The result of the predefined macros @, * and < and 'D' and 'F' variants is not quoted, so it may be necessary to put quotes around it.

Note that stem calculation can cause unexpected values. For example:

```
$@                     $*
/home/.wine/test       /home/
/home/test/.project    /home/test/
/../file               /.
```

Macro string substitution

When the macro name in an evaluation is followed by a colon and equal sign as in

```
$(MACRO:string1=string2)
```

then **amk** will replace *string1* at the end of every word in `$(MACRO)` by *string2* during evaluation. When `$(MACRO)` contains quoted path names, the quote character must be mentioned in both the original string and the replacement string[1]. For example:

```
$(MACRO:.report"=.d")
```

6.1.4. Makefile Functions

A function not only expands but also performs a certain operation. The following functions are available:

$(filter *pattern* ...,*item* ...)

The `filter` function filters a list of items using a pattern. It returns *items* that do match any of the *pattern* words, removing any items that do not match. The patterns are written using '%',

```
    ${filter %.c %.h, test.c test.h test.report readme.txt output.c}
```

results in:

```
    test.c test.h output.c
```

$(filter-out *pattern* ...,*item* ...)

The `filter-out` function returns all *items* that do not match any of the *pattern* words, removing the items that do match one or more. This is the exact opposite of the `filter` function.

```
    ${filter-out %.c %.h, test.c test.h test.report readme.txt output.c}
```

results in:

```
    test.report readme.txt
```

[1] Internally, **amk** tokenizes the evaluated text, but performs substitution on the original input text to preserve compatibility here with existing make implementations and POSIX.

$(foreach *var-name*, *item* ..., *action*)

The `foreach` function runs through a list of items and performs the same *action* for each *item*. The *var-name* is the name of the macro which gets dynamically filled with an item while iterating through the *item* list. In the *action* you can refer to this macro. For example:

```
    ${foreach T, test filter output, ${T}.c ${T}.h}
```

results in:

```
    test.c test.h filter.c filter.h output.c output.h
```

6.1.5. Conditional Processing

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

```
ifdef macro-name
if-lines
else
else-lines
endif
```

The *if-lines* and *else-lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else-lines* following it. White spaces (tabs or spaces) in front of preprocessing directives are allowed.

First the *macro-name* after the `ifdef` command is checked for definition. If the macro is defined then the *if-lines* are interpreted and the *else-lines* are discarded (if present). Otherwise the *if-lines* are discarded; and if there is an `else` line, the *else-lines* are interpreted; but if there is no else line, then no lines are interpreted.

When you use the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested to any level.

You can also add tests based on strings. With `ifeq` the result is true if the two strings match, with `ifneq` the result is true if the two strings do not match. They are used in the following way:

```
ifeq(string1,string2)
if-lines
else
else-lines
endif
```

6.1.6. Makefile Parsing

amk reads and interprets a makefile in the following order:

1. When the last character on a line is a backslash (****) (i.e. without trailing white space) then that line and the next line will be concatenated, removing the backslash and newline.

2. The unquoted '#' character indicates start of comment and may be placed anywhere on a line. It will be removed in this phase.

   ```
   # this comment line is continued\
   on the next line
   ```

3. Trailing white space is removed.
4. When a line starts with white space and it is not followed by a directive or preprocessing directive, then it is interpreted as a command for updating a target.
5. Otherwise, when a line contains the unquoted text '**=**', '**+=**' or '**:=**' operator, then it will be interpreted as a macro definition.
6. Otherwise, all macros on the line are evaluated before considering the next steps.
7. When the resulting line contains an unquoted '**:**' the line is interpreted as a dependency rule.
8. When the first token on the line is "`include`" or "`-include`" (which by now must start on the first column of the line), **amk** will execute the directive.
9. Otherwise, the line must be empty.

Macros in commands for updating a target are evaluated right before the actual execution takes place (or would take place when you use the **-n** option).

6.1.7. Makefile Command Processing

A line with leading white space (tabs or spaces) without a (preprocessing) directive is considered as a command for updating a target. When you use the option **-j** or **-J**, **amk** will execute the commands for updating different targets in parallel. In that case standard input will not be available and standard output and error output will be merged and displayed on standard output only after the commands have finished for a target.

You can precede a command by one or more of the following characters:

@	Do not show the command. By default, commands are shown prior to their output.
-	Continue upon error. This means that **amk** ignores a non-zero exit code of the command.
+	Execute the command, even when you use option **-n** (dry run).
|	Execute the command on the foreground with standard input, standard output and error output available.

Built-in commands

Command	Description
`true`	This command does nothing. Arguments are ignored.
`false`	This command does nothing, except failing with exit code 1. Arguments are ignored.

Command	Description
`echo` *arg*...	Display a line of text.
`exit` *code*	Exit with defined code. Depending on the program arguments and/or the extra rule options '-' this will cause **amk** to exit with the provided code. Please note that '`exit 0`' has currently no result.
`argfile` *file arg*...	Create an argument file suitable for the **--option-file** (**-f**) option of all the other tools. The first `argfile` argument is the name of the file to be created. Subsequent arguments specify the contents. An existing argument file is not modified unless necessary. So, the argument file itself can be used to create a dependency to options of the command for updating a target.
`rm` [*option*]... *file*...	Remove the specified file(s). The following options are available: **-r**, **--recursive** Remove directories and their contents recursively. **-f**, **--force** Force deletion. Ignore non-existent files, never prompt. **-i**, **--interactive** Interactive. Prompt before every removal. **-v**, **--verbose** Verbose mode. Explain what is being done. **-m** *file* Read options from *file*.. **-?**, **--help** Show usage.

6.1.8. Calling the amk Make Utility

The invocation syntax of **amk** is:

```
amk [option]... [target]... [macro=def]...
```

For example:

```
amk test.report
```

target You can specify any target that is defined in the makefile. A target can also be one of the intermediate files specified in the makefile.

macro=*def* Macro definition. This definition remains fixed for the **amk** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is not inherited by subordinate **amk**'s

option For a complete list and description of all **amk** make utility options, see Section 7.3, *Parallel Make Utility Options*.

Exit status

The make utility returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

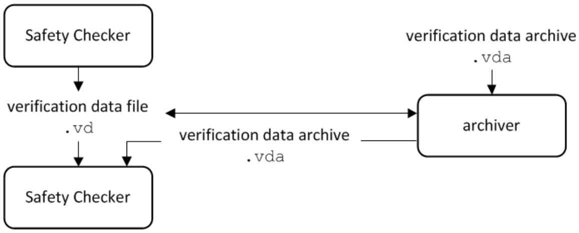
6.2. Archiver

The archiver **arsaf** is a program to build and maintain your own library files. A library file is a verification data archive with extension `.vda` and contains one or more verification data files (`.vd`) that may be used by the Safety Checker.

The archiver has five main functions:

- Deleting a module from the library
- Moving a module to another position in the library file
- Replacing a module in the library or add a new module
- Showing a table of contents of the library file
- Extracting a module from the library

The archiver takes the following files for input and output:



6.2.1. Calling the Archiver

You can call the archiver from the command line. The invocation syntax is:

```
arsaf key_option [sub_option...] library [module]
```

key_option With a key option you specify the main task which the archiver should perform. You must *always* specify a key option.

sub_option Sub-options specify into more detail how the archiver should perform the task that is specified with the key option. It is not obligatory to specify sub-options.

library The name of the library file on which the archiver performs the specified action. You must always specify a library name, except for the options **-?** and **-V**. When the library is not in the current directory, specify the complete path (either absolute or relative) to the library.

module The name of a verification data file. You must always specify a module file name when you add, extract, replace or remove a module from the library.

Options of the archiver utility

The following archiver options are available:

Description	**Option**	**Sub-option**
Main functions (key options)		
Replace or add a module	**-r**	**-a -b -c -n -u -v**
Extract a module from the library	**-x**	**-o -v**
Delete a module from library	**-d**	**-v**
Move a module to another position	**-m**	**-a -b -v**
Print a table of contents of the library	**-t**	**-s0 -s1**
Print the contents of a module to standard output	**-p**	
Sub-options		
Append or move new modules after existing module *name*	**-a** *name*	
Append or move new modules before existing module *name*	**-b** *name*	
Suppress the message that is displayed when a new library is created	**-c**	
Create a new library from scratch	**-n**	
Preserve last-modified date from the library	**-o**	
Print symbols in library modules	**-s**{**0**|**1**}	
Replace only newer modules	**-u**	
Verbose	**-v**	
Miscellaneous		
Display options	**-?**	
Display description of one or more diagnostic messages	**--diag**	
Display version header	**-V**	
Read options from *file*	**-f** *file*	
Suppress warnings above level *n*	**-w***n*	

For a complete list and description of all archiver options, see Section 7.4, *Archiver Options*.

6.2.2. Archiver Examples

Create a new library

If you add modules to a library that does not yet exist, the library is created. To create a new library with the name `mylib.vda` and add the modules `mod1.vd` and `mod2.vd` to it:

```
arsaf -r mylib.vda mod1.vd mod2.vd
```

Add a new module to an existing library

If you add a new module to an existing library, the module is added at the end of the module. (If the module already exists in the library, it is replaced.)

```
arsaf -r mylib.vda mod3.vd
```

Print a list of modules in the library

To inspect the contents of the library:

```
arsaf -t mylib.vda
```

The library has the following contents:

```
mod1.vd
mod2.vd
mod3.vd
```

Move a module to another position

To move `mod3.vd` to the beginning of the library, position it just before `mod1.vd`:

```
arsaf -mb mod1.vd mylib.vda mod3.vd
```

Delete a module from the library

To delete the module `mod1.vd` from the library `mylib.vda`:

```
arsaf -d mylib.vda mod1.vd
```

Extract all modules from the library

Extract all modules from the library `mylib.vda`:

```
arsaf -x mylib.vda
```

Chapter 7. Tool Options

This chapter provides a detailed description of the options for the Safety Checker, make utility and archiver.

7.1. Configuring the Command Line Environment

If you want to use the tools on the command line (using a Windows command prompt), you can set *environment variables*.

You can set the following environment variables:

Environment variable	Description
CSAFINC	With this variable you specify one or more additional directories in which the Safety Checker looks for include files. See Section 4.7, *How the Safety Checker Searches Include Files*.
PATH	With this variable you specify the directory in which the executables reside. This allows you to call the executables when you are not in the `bin` directory. Usually your system already uses the PATH variable for other purposes. To keep these settings, you need to add (rather than replace) the path. Use a semicolon (;) to separate path names.
TMPDIR	With this variable you specify the location where programs can create temporary files. Usually your system already uses this variable. In this case you do not need to change it.

See the documentation of your operating system on how to set environment variables.

7.2. Safety Checker Options

This section lists all Safety Checker options.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
csaf -Eil test.c
csaf --preprocess=+includes,+list test.c
```

When you do not specify an option, a default value may become active.

Safety Checker option: --absolute

Command line syntax

```
--absolute
```

Description

With this option the Safety Checker always shows the complete path name of the source file in diagnostic messages.

Related information

-

Safety Checker option: --bit-size-<type>

Command line syntax

```
--bit-size-char=size
--bit-size-short=size
--bit-size-int=size
--bit-size-long=size
--bit-size-llong=size
--bit-size-ptr=size
```

Default and minimum size:

Integer type	Default size	Minimum size
char	8	8
short	16	16
int	32	16
long	32	32
long long	64	64
pointer	32	8

Description

With these options you can configure the sizes of the integer types. The default sizes conform to the sizes as defined by ISO C99. The sizes should not be less than the minimum size.

Related information

-

Safety Checker option: --cert

Command line syntax

```
--cert={all | name[-name],...}
```

Default format: all

Description

With this option you can enable one or more checks for CERT C Secure Coding Standard recommendations/rules. When you omit the argument, all checks are enabled. *name* is the name of a CERT recommendation/rule, consisting of three letters and two digits. Specify only the three-letter mnemonic to select a whole category. For the list of names you can use, see Chapter 8, *CERT C Secure Coding Standard*.

On the command line you can use **--diag=cert** to see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, **--diag=pre** lists all supported preprocessor checks.

Example

To enable the check for CERT rule STR30-C, enter:

```
csaf --cert=str30 test.c
```

Related information

Chapter 8, *CERT C Secure Coding Standard*

Safety Checker option **--diag** (Explanation of diagnostic messages)

Safety Checker option: --define (-D)

Command line syntax

--define=*macro_name*[**=***macro_definition*]

-D*macro_name*[**=***macro_definition*]

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

On the command line, you can use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the Safety Checker with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the C source) is, for example, useful for conditional C source as shown in the example below.

> Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example

Consider the following C program with conditional code to check a demo program and a real program:

```
void main( void )
{
#if DEMO
    demo_func();   /* check the demo program */
#else
    real_func();   /* check the real program */
#endif
}
```

You can now use a macro definition to set the DEMO flag:

```
csaf --define=DEMO test.c
csaf --define=DEMO=1 test.c
```

Note that both invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

```
csaf --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.c
```

Related information

Safety Checker option **--undefine** (Remove preprocessor macro)

Safety Checker option **--option-file** (Specify an option file)

Safety Checker option: --dep-file

Command line syntax

`--dep-file``[=`*`file`*`]`

Description

With this option you tell the Safety Checker to generate dependency lines that can be used in a Makefile. In contrast to the option **--preprocess=+make**, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension `.d` (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

```
csaf --dep-file=test.dep test.c
```

The Safety Checker analyzes the file `test.c`, which results in the output file `test.report`, and generates dependency lines in the file `test.dep`.

Related information

Safety Checker option **--preprocess=+make** (Generate dependencies for make)

Safety Checker option **--make-target** (Specify target name for **--dep-file** output)

Safety Checker option: --diag

Command line syntax

```
--diag=[format:]{all | msg[-msg],...}
```

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. The Safety Checker does not analyze any files. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given (except for the CERT checks). If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

With **--diag=cert** you can see a list of the available CERT checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, **--diag=pre** lists all supported preprocessor checks.

Example

To display an explanation of message number 282, enter:

```
csaf --diag=282
```

This results in the following message and explanation:

```
E282: unterminated comment

Make sure that every comment starting with /* has a matching */.
Nested comments are not possible.
```

To write an explanation of all errors and warnings in HTML format to file `cerrors.html`, use redirection and enter:

```
csaf --diag=html:all > cerrors.html
```

Related information

Section 4.9, *Safety Checker Error Messages*

Safety Checker option **--cert** (Enable individual CERT checks)

Safety Checker option: --error-file

Command line syntax

`--error-file``[=`*`file`*`]`

Description

With this option the Safety Checker redirects error messages to a file. If you do not specify a filename, the error file will be named after the output file with extension `.err`.

Example

To write errors to `errors.err` instead of `stderr`, enter:

```
csaf --error-file=errors.err test.c
```

Related information

-

Safety Checker option: --error-limit

Command line syntax

```
--error-limit=number
```

Default: 42

Description

With this option you limit the number of error messages in one invocation run to the specified number. When the limit is exceeded, the Safety Checker aborts with fatal error message F105. Warnings and informational messages are not included in the count. When 0 (zero) or a negative number is specified, the Safety Checker emits all errors. Without this option the maximum number of errors is 42.

Related information

Section 4.9, *Safety Checker Error Messages*

Safety Checker option: --fp-model

Command line syntax

```
--fp-model=flags
```

You can set the following flags:

+/-contract	**c/C**	allow expression contraction
+/-float	**f/F**	treat 'double' as 'float'
+/-nonan	**n/N**	allow optimizations to ignore NaN/Inf
+/-rewrite	**r/R**	allow expression rewriting
+/-negzero	**z/Z**	ignore sign of -0.0
	0	alias for **--fp-model=CFNRZ** (strict)
	1	alias for **--fp-model=cFNRZ** (precise)
	2	alias for **--fp-model=cFnrz** (fast double)
	3	alias for **--fp-model=cfnrz** (fast single)

Default: **`--fp-model=cFnrz`**

Description

With this option you select the floating-point execution model.

With **--fp-model=+contract** you allow the Safety Checker to contract multiple float operations into a single operation, with different rounding results. A possible example is fused multiply-add.

With **--fp-model=+float** you tell the Safety Checker to treat variables and constants of type `double` as `float`. Because the `float` type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

With **--fp-model=+nonan** you allow the Safety Checker to ignore NaN or Inf input values. An example is to replace multiply by zero with zero.

With **--fp-model=+rewrite** you allow the Safety Checker to rewrite expressions by reassociating. This might result in rounding differences and possibly different exceptions. An example is to rewrite (a*c)+(b*c) as (a+b)*c.

With **--fp-model=+negzero** you allow the Safety Checker to ignore the sign of -0.0 values. An example is to replace (a-a) by zero.

Related information

Pragmas `STDC FP_CONTRACT`, `fp_negzero`, `fp_nonan` and `fp_rewrite` in Section 3.4, *Pragmas to Control the Safety Checker*.

Safety Checker option: --help (-?)

Command line syntax

--help[**=***item*]

-?

You can specify the following arguments:

options	**o**	Show extended option descriptions
pragmas	**p**	Show the list of supported pragmas
typedefs	**t**	Show the list of predefined typedefs

Description

Displays an overview of all command line options. With an argument you can specify which extended information is shown.

Example

The following invocations all display a list of the available command line options:

```
csaf -?
csaf --help
csaf
```

The following invocation displays a list of the available pragmas:

```
csaf --help=pragmas
```

Related information

-

Safety Checker option: --include-directory (-I)

Command line syntax

--include-directory=*path*,...

-I*path*,...

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory,

The order in which the Safety Checker searches for include files is:

1. The pathname in the C source file and the directory of the C source (only for #include files that are enclosed in "")
2. The path or paths that are specified with this option. Multiple paths/options are handled by the Safety Checker from left to right.
3. The path that is specified in the environment variable `CSAFINC` when the product was installed.
4. The default directory `$(PRODDIR)\include` (unless you specified option **--no-stdinc**).

Example

Suppose that the C source file `test.c` contains the following lines:

```
#include <stdio.h>
#include "myinc.h"
```

You can call the Safety Checker as follows:

```
csaf --include-directory=myinclude test.c
```

First the Safety Checker looks for the file `stdio.h` in the directory `myinclude` relative to the current directory. If it was not found, the Safety Checker searches in the environment variable and then in the default include directory.

The Safety Checker now looks for the file `myinc.h` in the directory where `test.c` is located. If the file is not there the Safety Checker searches in the directory `myinclude`. If it was still not found, the Safety Checker searches in the environment variable and then in the default include directory.

Related information

Safety Checker option **--include-file** (Include file at the start of an analysis)

Safety Checker option **--no-stdinc** (Skip standard include files directory)

Section 3.6.8, *Include Files*

Safety Checker option: --include-file (-H)

Command line syntax

`--include-file=``file,...`

`-H``file,...`

Description

With this option you include one or more extra files at the beginning of each C source file, before other includes. This is the same as specifying `#include "file"` at the beginning of *each* of your C sources.

Example

```
csaf --include-file=csaf.h test1.c test2.c
```

The file `csaf.h` is included at the beginning of both `test1.c` and `test2.c`.

Related information

Safety Checker option **--include-directory** (Add directory to include file search path)

Section 3.6.10, *Target Configuration*

Safety Checker option: --iso (-c)

Command line syntax

`--iso=``{`**`90`**`|`**`99`**`}`

`-c``{`**`90`**`|`**`99`**`}`

Default: **`--iso=99`**

Description

With this option you select the ISO C standard. C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard. C99 is the default.

Example

To select the ISO C90 standard on the command line:

```
csaf --iso=90 test.c
```

Related information

Section 2.2, *C Syntax Checking*

Safety Checker option **--language** (Language extensions)

Safety Checker option: --keep-output-files (-k)

Command line syntax

`--keep-output-files`

`-k`

Description

If an error occurs during safety checking, the resulting `.report` file may be incomplete or incorrect. With this option you keep the generated output file (`.report`) when an error occurs.

By default the Safety Checker removes the generated output file (`.report`) when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to inspect the generated report file. Even if it is incomplete or incorrect.

Example

```
csaf --keep-output-files test.c
```

When an error occurs, the generated output file `test.report` will *not* be removed.

Related information

Safety Checker option **--warnings-as-errors** (Treat warnings as errors)

Safety Checker option: --language (-A)

Command line syntax

--language=[*flags*]

-A[*flags*]

You can set the following flags:

+/-gcc	**g/G**	enable a number of gcc extensions
+/-kanji	**k/K**	support for Shift JIS Kanji in strings
+/-comments	**p/P**	// comments in ISO C90 mode
+/-strings	**x/X**	relaxed const check for string literals

Default: **-AGKpx**

Default (without flags): **-AGKPX**

Description

With this option you control the language extensions the Safety Checker can accept.

The option **--language** (**-A**) without flags disables all language extensions.

GNU C extensions

The **--language=+gcc** (**-Ag**) option enables the following gcc language extensions:

- The identifier __FUNCTION__ expands to the current function name.
- Alternative syntax for variadic macros.
- Alternative syntax for designated initializers.
- Allow zero sized arrays.
- Allow empty struct/union.
- Allow unnamed struct/union fields.
- Allow empty initializer list.
- Allow initialization of static objects by compound literals.
- The middle operand of a ? : operator may be omitted.
- Allow a compound statement inside braces as expression.
- Allow arithmetic on void pointers and function pointers.
- Allow a range of values after a single case label.

- Additional preprocessor directive `#warning`.
- Allow comma operator, conditional operator and cast as lvalue.
- An inline function without "`static`" or "`extern`" will be global.
- An "`extern inline`" function will not be analyzed on its own.
- An `__attribute__` directly following a struct/union definition relates to that tag instead of to the objects in the declaration.

For a more complete description of these extensions, you can refer to the UNIX gcc info pages (**info gcc**).

Shift JIS Kanji support

With **--language=+kanji** (**-Ak**) you enable support for Shift JIS encoded Kanji multi-byte characters in strings, (wide) character constants and `//` comments. Without this option, encodings with 0x5c as the second byte conflict with the use of the backslash as an escape character. Shift JIS in `/*...*/` comments is supported regardless of this option. Note that Shift JIS also includes Katakana and Hiragana.

Comments in ISO C90 mode

With **--language=+comments** (**-Ap**) you enable support for C++ style comments (//) in ISO C90 mode (option **--iso=90**). In ISO C99 mode this style of comments is always accepted.

Check assignment of string literal to non-const string pointer

With **--language=+strings** (**-Ax**) you disable warnings about discarded `const` qualifiers when a string literal is assigned to a non-`const` pointer.

```
char *p;
int main( void )
{
    p = "hello"; // with -AX the Safety Checker issues warning W525
    return 0;
}
```

Related information

Safety Checker option **--iso** (ISO C standard)

Section 3.2, *Shift JIS Kanji Support*

Safety Checker option: --make-target

Command line syntax

`--make-target=`*`name`*

Description

With this option you can overrule the default target name in the make dependencies generated by the options **--preprocess=+make** (**-Em**) and **--dep-file**. The default target name is the basename of the input file, with extension `.vd`.

Example

```
csaf --preprocess=+make --make-target=test.report test.c
```

The Safety Checker generates dependency lines with the target name `test.report` instead of the default name `test.vd`.

Related information

Safety Checker option **--preprocess=+make** (Generate dependencies for make)

Safety Checker option **--dep-file** (Generate dependencies in a file)

Safety Checker option: --misrac

Command line syntax

```
--misrac={all | nr[-nr]},...
```

Description

With this option you specify to the Safety Checker which MISRA C rules must be checked. With the option **--misrac=all** the Safety Checker checks for all supported MISRA C rules.

Example

```
csaf --misrac=9-13 test.c
```

The Safety Checker generates an error for each MISRA C rule 9, 10, 11, 12 or 13 violation in file `test.c`.

Related information

Section 4.8.2, *C Code Checking: MISRA C*

Safety Checker option **--misrac-mandatory-warnings**

Safety Checker option **--misrac-advisory-warnings**

Safety Checker option **--misrac-required-warnings**

Safety Checker option: --misrac-advisory-warnings / --misrac-required-warnings / --misrac-mandatory-warnings

Command line syntax

```
--misrac-advisory-warnings
--misrac-required-warnings
--misrac-mandatory-warnings
```

Description

Normally, if an advisory rule, required rule or mandatory rule is violated, the Safety Checker generates an error. As a consequence, no report file is generated. With this option, the Safety Checker generates a warning instead of an error.

Related information

Section 4.8.2, *C Code Checking: MISRA C*

Safety Checker option **--misrac**

Safety Checker option: --misrac-version

Command line syntax

`--misrac-version=`{**`1998`**|**`2004`**|**`2012`**}

Default: 2004

Description

MISRA C rules exist in three versions: MISRA C:1998, MISRA C:2004 and MISRA C:2012. By default, the C source is checked against the MISRA C:2004 rules. With this option you can select which version to use.

Related information

Section 4.8.2, *C Code Checking: MISRA C*

Safety Checker option **--misrac**

Safety Checker option: --no-stdinc

Command line syntax

```
--no-stdinc
```

Description

With this option you tell the Safety Checker not to look in the default `include` directory relative to the installation directory, when searching for include files. This way the Safety Checker only searches in the include file search paths you specified.

Related information

Safety Checker option **--include-directory** (Add directory to include file search path)

Section 4.7, *How the Safety Checker Searches Include Files*

Safety Checker option: --no-warnings (-w)

Command line syntax

```
--no-warnings[=number[-number],...]
```

```
-w[number[-number],...]
```

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number or a range, only the specified warnings are suppressed. You can specify the option **--no-warnings=***number* multiple times.

Example

To suppress warnings 537 and 538, enter:

```
csaf test.c --no-warnings=537,538
```

Related information

Safety Checker option **--warnings-as-errors** (Treat warnings as errors)

Pragma `warning`

Safety Checker option: --option-file (-f)

Command line syntax

`--option-file=`*`file`*`,...`

`-f` *`file`*`,...`

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the Safety Checker.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"
  ```

  ```
  'This has a double quote " embedded'
  ```

  ```
  'This has a double quote " and a single quote '"' embedded"
  ```

- When a text line reaches its length limit, use a \ to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"
  ```

  ```
           -> "This is a continuation line"
  ```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--output-format=acDrt
--define=DEMO=1
test.c
```

Specify the option file to the Safety Checker:

```
csaf --option-file=myoptions
```

This is equivalent to the following command line:

```
csaf --output-format=acDrt --define=DEMO=1 test.c
```

Related information

-

Safety Checker option: --output-file (-o)

Command line syntax

`--output-file=`*`file`*

`-o` *`file`*

Description

With this option you can specify another filename for the output file of the Safety Checker. Without this option the basename of the C source file is used with extension `.report`.

Example

To create the file `output.report` instead of `test.report`, enter:

```
csaf --output-file=output.report test.c
```

Related information

-

Safety Checker option: --output-format

Command line syntax

```
--output-format[=flag,...]
```

You can specify the following format flags:

+/-areas	**a/A**	Safety class areas
+/-callgraph	**c/C**	Call/data graph of the application
+/-details	**d/D**	Call graph details
+/-misra	**m/M**	MISRA C Quality Assurance report
+/-rights	**r/R**	Safety class access rights
+/-selection	**s/S**	Safety class selections
+/-table	**t/T**	Access table
	0	Alias for **ACDMRST** (nothing)
	1	Alias for **acDmrsT** (default)
	2	Alias for **acdmrst** (everything)

Default: **`--output-format=acDmrsT`**

Description

With this option you can control which parts of the report output you want to see.

1. Safety class areas table
2. Safety class selections table
3. Safety class access rights table
4. Call/data graph
5. Call graph details (actions and checks)
6. Access table (sorted on address)
7. MISRA C Quality Assurance report

By default, all parts are reported, except for parts 5 and 6.

The MISRA C Quality Assurance report lists the various modules in the project with the respective MISRA C settings at the time of analysis. This report is only shown if you specify the option **--misrac** with one or more rules.

Related information

Section 4.5, *Output of the Safety Checker*

Safety Checker option: --preprocess (-E)

Command line syntax

```
--preprocess[=flags]

-E[flags]
```

You can set the following flags:

+/-comments	**c/C**	keep comments
+/-includes	**i/I**	generate a list of included source files
+/-list	**l/L**	generate a list of macro definitions
+/-make	**m/M**	generate dependencies for make
+/-noline	**p/P**	strip #line source position information

Default: **`-ECILMP`**

Description

With this option you tell the Safety Checker to preprocess the C source. The Safety Checker sends the preprocessed file to `stdout`. To capture the information in a file, specify an output file with the option **--output**.

With **--preprocess=+comments** you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With **--preprocess=+includes** the Safety Checker will generate a list of all included source files. The preprocessor output is discarded.

With **--preprocess=+list** the Safety Checker will generate a list of all macro definitions. The preprocessor output is discarded.

With **--preprocess=+make** the Safety Checker will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension `.vd`. With the option **--make-target** you can specify a target name which overrules the default target name.

With **--preprocess=+noline** you tell the preprocessor to strip the #line source position information (lines starting with `#line`). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

Example

```
csaf --preprocess=+comments,+includes,-list,-make,-noline test.c
     --output=test.pre
```

The Safety Checker preprocesses the file `test.c` and sends the output to the file `test.pre`. Comments and a list of all included source files are included but no list of macro definitions and no dependencies are generated and the line source position information is not stripped from the output file.

Related information

Safety Checker option **--dep-file** (Generate dependencies in a file)

Safety Checker option **--make-target** (Specify target name for **-Em** output)

Safety Checker option: --signed-bitfields

Command line syntax

`--signed-bitfields`

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default an `int` bit-field is treated as `unsigned int`. With this option you tell the Safety Checker to treat `int` bit-fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

Related information

Section 3.1, *Data Types*

Safety Checker option: --stdout (-n)

Command line syntax

`--stdout`

`-n`

Description

With this option you tell the Safety Checker to send the output to `stdout` (usually your screen). No files are created. This option is for example useful to quickly inspect the output or to redirect the output to other tools.

Related information

-

Safety Checker option: --uchar (-u)

Command line syntax

`--uchar`

`-u`

Description

By default `char` is the same as specifying `signed char`. With this option `char` is the same as `unsigned char`.

Related information

-

Safety Checker option: --undefine (-U)

Command line syntax

--undefine=*macro_name*

-U*macro_name*

Description

With this option you can undefine an earlier defined macro as with `#undef`. This option is for example useful to undefine predefined macros.

The following predefined ISO C standard macros cannot be undefined:

`__FILE__`	current source filename
`__LINE__`	current source line number (int type)
`__TIME__`	hh:mm:ss
`__DATE__`	Mmm dd yyyy
`__STDC__`	level of ANSI standard

Example

To undefine the predefined macro `__TASKING__`:

```
csaf --undefine=__TASKING__ test.c
```

Related information

Safety Checker option **--define** (Define preprocessor macro)

Section 3.5, *Predefined Preprocessor Macros*

Safety Checker option: --vd

Command line syntax

```
--vd
```

Description

With option **--vd** you can split up the analysis phase of the Safety Checker into smaller steps. The Safety Checker skips partitioning checking and reporting and writes the verification data (VD) to a file with the suffix `.vd`. You can use those `.vd` files as input files again for the Safety Checker.

Related information

Section 4.1, *Safety Checker Phases*

Safety Checker option: --version (-V)

Command line syntax

```
--version
```

```
-V
```

Description

Display version information. The Safety Checker ignores all other options or input files.

Related information

-

Safety Checker option: --warnings-as-errors

Command line syntax

`--warnings-as-errors``[=`*`number`*`[-`*`number`*`],...]`

Description

If the Safety Checker encounters an error, it stops analyzing. When you use this option without arguments, you tell the Safety Checker to treat all warnings not suppressed by option **--no-warnings** (or `#pragma warning`) as errors. This means that the exit status of the Safety Checker will be non-zero after one or more Safety Checker warnings. As a consequence, the Safety Checker now also stops after encountering a warning.

You can limit this option to specific warnings by specifying a comma-separated list of warning numbers or ranges. In this case, this option takes precedence over option **--no-warnings** (and `#pragma warning`).

Related information

Safety Checker option **--no-warnings** (Suppress some or all warnings)

Pragma `warning`

7.3. Parallel Make Utility Options

The invocation syntax is:

```
amk [option...] [target...] [macro=def]
```

This section describes all options for the parallel make utility.

For detailed information about the parallel make utility and using makefiles see Section 6.1, *Make Utility amk*.

Parallel make utility option: --always-rebuild (-a)

Command line syntax

`--always-rebuild`

`-a`

Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild *all* files, without checking whether they are out of date.

Example

```
amk -a
```

Rebuilds all your files, regardless of whether they are out of date or not.

Related information

-

Parallel make utility option: --change-dir (-G)

Command line syntax

`--change-dir=`*`path`*

`-G` *`path`*

Description

Normally you must call the make utility from the directory where your makefile and other files are stored.

With the option **-G** you can call the make utility from within another directory. The *path* is the path to the directory where your makefile and other files are stored and can be absolute or relative to your current directory.

The macro `SUBDIR` is defined with the value of *path*.

Example

Suppose your makefile and other files are stored in the directory `..\myfiles`. You can call the make utility, for example, as follows:

```
amk -G ..\myfiles
```

Related information

-

Parallel make utility option: --diag

Command line syntax

--diag=[*format*:]{**all** | *msg*[-*msg*],...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

Example

To display an explanation of message number 169, enter:

```
amk --diag=451
```

This results in the following message and explanation:

```
E451: make stopped

An error has occured while executing one of the commands
of the target, and -k option is not specified.
```

To write an explanation of all errors and warnings in HTML format to file `amkerrors.html`, use redirection and enter:

```
amk --diag=html:all > amkerrors.html
```

Related information

-

Parallel make utility option: --dry-run (-n)

Command line syntax

```
--dry-run
```

```
-n
```

Description

With this option you tell the make utility to perform a dry run. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

Example

```
amk -n
```

The make utility does not perform any tasks but displays what it would do if called without the option **-n**.

Related information

Parallel make utility option **-s** (Do not print commands before execution)

Parallel make utility option: --help (-? / -h)

Command line syntax

--help[**=***item*]

-h[*item*]

-?

You can specify the following arguments:

options	**o**	Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

The following invocations all display a list of the available command line options:

```
amk -?
amk -h
amk --help
```

To see a detailed description of the available options, enter:

```
amk --help=options
```

Related information

-

Parallel make utility option: --jobs (-j) / --jobs-limit (-J)

Command line syntax

```
--jobs[=number]
-j[number]
```

```
--jobs-limit[=number]
-J[number]
```

Description

When these options you can limit the number of parallel jobs. The default is 1. Zero means no limit. When you omit the *number*, **amk** uses the number of cores detected.

Option **-J** is the same as **-j**, except that the number of parallel jobs is limited by the number of cores detected.

Example

```
amk -j3
```

Limit the number of parallel jobs to 3.

Related information

-

Parallel make utility option: --keep-going (-k)

Command line syntax

```
--keep-going
```

```
-k
```

Description

When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option **-k**, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

Example

```
amk -k
```

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

Related information

-

Parallel make utility option: --list-targets (-l)

Command line syntax

`--list-targets`

`-l`

Description

With this option, the make utility lists all "primary" targets that are out of date.

Example

```
amk -l
list of targets
```

Related information

-

Parallel make utility option: --makefile (-f)

Command line syntax

--makefile=*my_makefile*

-f *my_makefile*

Description

By default the make utility uses the file `makefile` to build your files.

With this option you tell the make utility to use the specified file instead of the file `makefile`. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

If you use '-' instead of a makefile name it means that the information is read from `stdin`.

Example

```
amk -f mymake
```

The make utility uses the file `mymake` to build your files.

Related information

-

Parallel make utility option: --no-warnings (-w)

Command line syntax

```
--no-warnings[=number,...]
```

```
-w[number,...]
```

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=***number* multiple times.

Example

To suppress warnings 751 and 756, enter:

```
amk --no-warnings=751,756
```

Related information

Parallel make utility option **--warnings-as-errors** (Treat warnings as errors)

Parallel make utility option: --silent (-s)

Command line syntax

`--silent`

`-s`

Description

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

Example

```
amk -s
```

The make utility rebuilds your files but does not print the commands it executes during the make process.

Related information

Parallel make utility option **-n** (Perform a dry run)

Parallel make utility option: --version (-V)

Command line syntax

```
--version
```

```
-V
```

Description

Display version information. The make utility ignores all other options or input files.

Related information

-

Parallel make utility option: --warnings-as-errors

Command line syntax

`--warnings-as-errors[=number,...]`

Description

If the make utility encounters an error, it stops. When you use this option without arguments, you tell the make utility to treat all warnings as errors. This means that the exit status of the make utility will be non-zero after one or more warnings. As a consequence, the make utility now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Parallel make utility option **--no-warnings** (Suppress some or all warnings)

7.4. Archiver Options

The archiver and library maintainer **arsaf** is a tool to build library files and it offers the possibility to replace, extract and remove modules from an existing library.

The invocation syntax is:

```
arsaf key_option [sub_option...] library [module]
```

This section describes all options for the archiver. Some suboptions can only be used in combination with certain key options. They are described together. Suboptions that can always be used are described separately.

For detailed information about the archiver, see Section 6.2, *Archiver*.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Overview of the options of the archiver utility

The following archiver options are available:

Description	Option	Sub-option
Main functions (key options)		
Replace or add a module	**-r**	**-a -b -c -n -u -v**
Extract a module from the library	**-x**	**-o -v**
Delete a module from library	**-d**	**-v**
Move a module to another position	**-m**	**-a -b -v**
Print a table of contents of the library	**-t**	**-s0 -s1**
Print the contents of a module to standard output	**-p**	
Sub-options		
Append or move new modules after existing module *name*	**-a** *name*	
Append or move new modules before existing module *name*	**-b** *name*	
Suppress the message that is displayed when a new library is created	**-c**	
Create a new library from scratch	**-n**	
Preserve last-modified date from the library	**-o**	
Print symbols in library modules	**-s**{**0**|**1**}	
Replace only newer modules	**-u**	
Verbose	**-v**	

Description	**Option**	**Sub-option**
Miscellaneous		
Display options	**-?**	
Display description of one or more diagnostic messages	**--diag**	
Display version header	**-V**	
Read options from *file*	**-f** *file*	
Suppress warnings above level *n*	**-w***n*	

Archiver option: --delete (-d)

Command line syntax

```
--delete [--verbose]
```

```
-d [-v]
```

Description

Delete the specified modules from a library. With the suboption **--verbose** (**-v**) the archiver shows which modules are removed.

--verbose	**-v**	Verbose: the archiver shows which modules are removed.

Example

```
arsaf --delete mylib.vda mod1.vd mod2.vd
```

The archiver deletes `mod1.vd` and `mod2.vd` from the library `mylib.vda`.

```
arsaf -d -v mylib.vda mod1.vd mod2.vd
```

The archiver deletes `mod1.vd` and `mod2.vd` from the library `mylib.vda` and displays which modules are removed.

Related information

-

Archiver option: --diag

Command line syntax

--diag=[*format*:]{**all** | *msg*[-*msg*],...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. The archiver does not perform any actions. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

Example

To display an explanation of message number 102, enter:

```
arsaf --diag=102
```

This results in the following message and explanation:

```
F102: cannot create "<file>"

The output file or a temporary file could not be created. Check if you have
sufficient disk space and if you have write permissions for the specified
file.
```

To write an explanation of all errors and warnings in HTML format to file `arerrors.html`, use redirection and enter:

```
arsaf --diag=html:all > arerrors.html
```

Related information

-

Archiver option: --dump (-p)

Command line syntax

`--dump`

`-p`

Description

Print the contents of the specified module(s) in the library to standard output.

This option is only useful when you redirect or pipe the output to other files or tools that serve your own purposes. Normally you do not need this option.

Example

```
arsaf --dump mylib.vda mod1.vd > file.vd
```

The archiver prints the file `mod1.vd` to standard output where it is redirected to the file `file.vd`. The effect of this example is very similar to extracting a file from the library but in this case the 'extracted' file gets another name.

Related information

-

Archiver option: --extract (-x)

Command line syntax

```
--extract [--modtime] [--verbose]
```

```
-x [-o] [-v]
```

Description

Extract an existing module from the library.

--modtime	**-o**	Give the extracted module the same date as the last-modified date that was recorded in the library. Without this suboption it receives the last-modified date of the moment it is extracted.
--verbose	**-v**	Verbose: the archiver shows which moules are extracted.

Example

To extract the file `mod1.vd` from the library `mylib.vda`:

```
arsaf --extract mylib.vda mod1.vd
```

If you do not specify a module, all modules are extracted:

```
arsaf -x mylib.vda
```

Related information

-

Archiver option: --help (-?)

Command line syntax

```
--help[=item]

-?
```

You can specify the following argument:

options	Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

The following invocations all display a list of the available command line options:

```
arsaf -?
arsaf --help
arsaf
```

To see a detailed description of the available options, enter:

```
arsaf --help=options
```

Related information

-

Archiver option: --move (-m)

Command line syntax

```
--move [-a posname] [-b posname]

-m [-a posname] [-b posname]
```

Description

Move the specified modules to another position in the library.

The ordering of modules in a library can make a difference in how programs are linked if a symbol is defined in more than one module.

By default, the specified modules are moved to the end of the archive. Use the suboptions **-a** or **-b** to move them to a specified place instead.

--after=*posname*	**-a** *posname*	Move the specified module(s) after the existing module *posname*.
--before=*posname*	**-b** *posname*	Move the specified module(s) before the existing module *posname*.

Example

Suppose the library `mylib.vda` contains the following modules (see option **--print**):

```
mod1.vd
mod2.vd
mod3.vd
```

To move `mod1.vd` to the end of `mylib.vda`:

```
arsaf --move mylib.vda mod1.vd
```

To move `mod3.vd` just before `mod2.vd`:

```
arsaf -m -b mod3.vd mylib.vda mod2.vd
```

The library `mylib.vda` after these two invocations now looks like:

```
mod3.vd
mod2.vd
mod1.vd
```

Related information

Archiver option **--print** (-t) (Print library contents)

Archiver option: --option-file (-f)

Command line syntax

`--option-file=`*`file`*

`-f` *`file`*

Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the archiver.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** (**-f**) multiple times.

If you use '-' instead of a filename it means that the options are read from `stdin`.

Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  ```
  "This has a single quote ' embedded"
  ```

  ```
  'This has a double quote " embedded'
  ```

  ```
  'This has a double quote " and a single quote '"' embedded"
  ```

- When a text line reaches its length limit, use a \ to continue the line. Whitespace between quotes is preserved.

  ```
  "This is a continuation \
  line"
  ```

  ```
           -> "This is a continuation line"
  ```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
-x mylib.vda mod1.vd
-w5
```

Specify the option file to the archiver:

```
arsaf --option-file=myoptions
```

This is equivalent to the following command line:

```
arsaf -x mylib.vda mod1.vd -w5
```

Related information

-

Archiver option: --print (-t)

Command line syntax

```
--print [--symbols=0|1]
```

```
-t [-s0|-s1]
```

Description

Print a table of contents of the library to standard output. With the suboption **-s0** the archiver displays all symbols per module.

--symbols=0	**-s0**	Displays per module the name of the module itself and all symbols in the module.
--symbols=1	**-s1**	Displays the symbols of all modules in the library in the form *library_name*:*module_name*:*symbol_name*

Example

```
arsaf --print mylib.vda
```

The archiver prints a list of all modules in the library `mylib.vda`:

```
arsaf -t -s0 mylib.vda
```

The archiver prints per module all symbols in the library. For example:

```
hello.vd
   symbols:
        _main
        _world
```

Related information

-

Archiver option: --replace (-r)

Command line syntax

```
--replace [--after=posname] [--before=posname]
          [--create] [--new] [--newer-only] [--verbose]
```

```
-r [-a posname] [-b posname][-c] [-n] [-u] [-v]
```

Description

You can use the option **--replace** (**-r**) for several purposes:

- Adding new modules to the library
- Replacing modules in the library with the same module of a newer date
- Creating a new library

The option **--replace** (**-r**) normally *adds* a new module to the library. However, if the library already contains a module with the specified name, the existing module is *replaced*. If you specify a library that does not exist, the archiver creates a new library with the specified name.

If you add a module to the library without specifying the suboption **-a** or **-b**, the specified module is added at the end of the archive. Use the suboptions **-a** or **-b** to insert them after/before a specified place instead.

--after=*posname*	**-a** *posname*	Insert the specified module(s) after the existing module *posname*.
--before=*posname*	**-b** *posname*	Insert the specified module(s) before the existing module *posname*.
--create	**-c**	Suppress the message that is displayed when a new library is created.
--new	**-n**	Create a new library from scratch. If the library already exists, it is overwritten.
--newer-only	**-u**	Insert the specified module only if it is newer than the module in the library.
--verbose	**-v**	Verbose: the archiver shows which modules are replaced.

The suboptions **-a** or **-b** have no effect when a module is added to the library.

Example

Suppose the library `mylib.vda` contains the following module (see option **--print**):

```
mod1.vd
```

To add `mod2.vd` to the end of `mylib.vda`:

```
arsaf --replace mylib.vda mod2.vd
```

To insert `mod3.vd` just before `mod2.vd`:

```
arsaf -r -b mod2.vd mylib.vda mod3.vd
```

The library `mylib.vda` after these two invocations now looks like:

```
mod1.vd
mod3.vd
mod2.vd
```

Creating a new library

To *create a new library file*, add a module and specify a library that does not yet exist:

```
arsaf --replace newlib.vda mod1.vd
```

The archiver creates the library `newlib.vda` and adds the module `mod1.vd` to it.

To *create a new library file and overwrite an existing library*, add a module and specify an existing library with the suboption **--new** (**-n**):

```
arsaf -r -n mylib.vda mod1.vd
```

The archiver overwrites the library `mylib.vda` and adds the module `mod1.vd` to it. The new library `mylib.vda` only contains `mod1.vd`.

Related information

Archiver option **--print** (**-t**) (Print library contents)

Archiver option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The archiver ignores all other options or input files.

Example

```
arsaf -V
```

The archiver displays the version information but does not perform any tasks.

Related information

-

Archiver option: --warning (-w)

Command line syntax

`--warning=``*level*`

`-w``*level*`

Description

With this suboption you tell the archiver to suppress all warnings above the specified level. The level is a number between 0 - 9.

The level of a message is printed between parentheses after the warning number. If you do not use the **-w** option, the default warning level is 8.

Example

To suppress warnings above level 5:

```
arsaf --extract --warning=5 mylib.vda mod1.vd
```

Related information

-

Chapter 8. CERT C Secure Coding Standard

The CERT C Secure Coding Standard provides rules and recommendations for secure coding in the C programming language. The goal of these rules and recommendations is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. The application of the secure coding standard will lead to higher-quality systems that are robust and more resistant to attack.

This chapter contains an overview of the CERT C Secure Coding Standard recommendations and rules that are supported by the Safety Checker by TASKING.

For details see the CERT C Secure Coding Standard web site. For general information about CERT secure coding, see www.cert.org/secure-coding.

Identifiers

Each rule and recommendation is given a unique identifier. These identifiers consist of three parts:

- a three-letter mnemonic representing the section of the standard
- a two-digit numeric value in the range of 00-99
- the letter "C" indicates that this is a C language guideline

The three-letter mnemonic is used to group similar coding practices and to indicate to which category a coding practice belongs.

The numeric value is used to give each coding practice a unique identifier. Numeric values in the range of 00-29 are reserved for recommendations, while values in the range of 30-99 are reserved for rules.

Safety Checker invocation

With the Safety Checker option **--cert** you can enable one or more checks for the CERT C Secure Coding Standard recommendations/rules. With **--diag=cert** you can see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, **--diag=pre** lists all supported checks in the preprocessor category.

8.1. Preprocessor (PRE)

PRE01-C Use parentheses within macros around parameter names

Parenthesize all parameter names in macro definitions to avoid precedence problems.

PRE02-C Macro replacement lists should be parenthesized

Macro replacement lists should be parenthesized to protect any lower-precedence operators from the surrounding expression. The example below is syntactically correct, although the `!=` operator was omitted. Enclosing the constant `-1` in parenthesis will prevent the incorrect interpretation and force an error message:

```
#define EOF -1  // should be (-1)
int getchar(void);
void f(void)
{
  if (getchar() EOF) // != operator omitted
  {
    /* ... */
  }
}
```

PRE10-C Wrap multi-statement macros in a do-while loop

When multiple statements are used in a macro, enclose them in a `do-while` statement, so the macro can appear safely inside `if` clauses or other places that expect a single statement or a statement block. Braces alone will not work in all situations, as the macro expansion is typically followed by a semicolon.

PRE11-C Do not conclude a single statement macro definition with a semicolon

Macro definitions consisting of a single statement should not conclude with a semicolon. If required, the semicolon should be included following the macro expansion. Inadvertently inserting a semicolon can change the control flow of the program.

8.2. Declarations and Initialization (DCL)

DCL30-C Declare objects with appropriate storage durations

The lifetime of an automatic object ends when the function returns, which means that a pointer to the object becomes invalid.

DCL31-C Declare identifiers before using them

The ISO C90 standard allows implicit typing of variables and functions. Because implicit declarations lead to less stringent type checking, they can often introduce unexpected and erroneous behavior or even security vulnerabilities. The ISO C99 standard requires type identifiers and forbids implicit function declarations. For backwards compatibility reasons, the Safety Checker assumes an implicit declaration and continues translation after issuing a warning message (W505 or W535).

DCL32-C Guarantee that mutually visible identifiers are unique

The Safety Checker encountered two or more identifiers that are identical in the first 31 characters. The ISO C99 standard allows a compiler to ignore characters past the first 31 in an identifier. Two distinct identifiers that are identical in the first 31 characters may lead to problems when the code is ported to a different compiler.

DCL35-C Do not invoke a function using a type that does not match the function definition

This warning is generated when a function pointer is set to refer to a function of an incompatible type. Calling this function through the function pointer will result in undefined behavior. Example:

```
void my_function(int a);
int main(void)
{
  int (*new_function)(int a) = my_function;
  return (*new_function)(10); /* the behavior is undefined */
}
```

8.3. Expressions (EXP)

EXP01-C Do not take the size of a pointer to determine the size of the pointed-to type

The size of the object(s) allocated by malloc(), calloc() or realloc() should be a multiple of the size of the base type of the result pointer. Therefore, the sizeof expression should be applied to this base type, and not to the pointer type.

EXP12-C Do not ignore values returned by functions

The Safety Checker gives this warning when the result of a function call is ignored at some place, although it is not ignored for other calls to this function. This warning will not be issued when the function result is ignored for all calls, or when the result is explicitly ignored with a (void) cast.

EXP30-C Do not depend on order of evaluation between sequence points

Between two sequence points, an object should only be modified once. Otherwise the behavior is undefined.

EXP32-C Do not access a volatile object through a non-volatile reference

If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.

EXP33-C Do not reference uninitialized memory

Uninitialized automatic variables default to whichever value is currently stored on the stack or in the register allocated for the variable. Consequently, uninitialized memory can cause a program to behave in an unpredictable or unplanned manner and may provide an avenue for attack.

EXP34-C Ensure a null pointer is not dereferenced

Attempting to dereference a null pointer results in undefined behavior, typically abnormal program termination.

EXP37-C Call functions with the arguments intended by the API

When a function is properly declared with function prototype information, an incorrect call will be flagged by the Safety Checker. When there is no prototype information available at the call, the Safety Checker cannot check the number of arguments and the types of the arguments. This message is issued to warn about this situation.

EXP38-C Do not call offsetof() on bit-field members or invalid types

The behavior of the offsetof() macro is undefined when the member designator parameter designates a bit-field.

8.4. Integers (INT)

INT30-C Ensure that unsigned integer operations do not wrap

A constant with an unsigned integer type is truncated, resulting in a wrap-around.

INT34-C Do not shift a negative number of bits or more bits than exist in the operand

The shift count of the shift operation may be negative or greater than or equal to the size of the left operand. According to the C standard, the behavior of such a shift operation is undefined. Make sure the shift count is in range by adding appropriate range checks.

INT35-C Evaluate integer expressions in a larger size before comparing or assigning to that size

If an integer expression is compared to, or assigned to a larger integer size, that integer expression should be evaluated in that larger size by explicitly casting one of the operands.

8.5. Floating Point (FLP)

FLP30-C Do not use floating point variables as loop counters

To avoid problems with limited precision and rounding, floating point variables should not be used as loop counters.

FLP35-C Take granularity into account when comparing floating point values

Floating point arithmetic in C is inexact, so floating point values should not be tested for exact equality or inequality.

FLP36-C Beware of precision loss when converting integral types to floating point

Conversion from integral types to floating point types without sufficient precision can lead to loss of precision.

8.6. Arrays (ARR)

ARR01-C Do not apply the sizeof operator to a pointer when taking the size of an array

A function parameter declared as an array, is converted to a pointer by the Safety Checker. Therefore, the sizeof operator applied to this parameter yields the size of a pointer, and not the size of an array.

ARR34-C Ensure that array types in expressions are compatible

Using two or more incompatible arrays in an expression results in undefined behavior.

ARR35-C Do not allow loops to iterate beyond the end of an array

Reading or writing of data outside the bounds of an array may lead to incorrect program behavior or execution of arbitrary code.

8.7. Characters and Strings (STR)

STR30-C Do not attempt to modify string literals

Writing to a string literal has undefined behavior, as identical strings may be shared and/or allocated in read-only memory.

STR33-C Size wide character strings correctly

Wide character strings may be improperly sized when they are mistaken for narrow strings or for multi-byte character strings.

STR34-C Cast characters to unsigned types before converting to larger integer sizes

A signed character is sign-extended to a larger signed integer value. Use an explicit cast, or cast the value to an unsigned type first, to avoid unexpected sign-extension.

STR36-C Do not specify the bound of a character array initialized with a string literal

The Safety Checker issues this warning when the character buffer initialized by a string literal does not provide enough room for the terminating null character.

8.8. Memory Management (MEM)

MEM00-C Allocate and free memory in the same module, at the same level of abstraction

The Safety Checker issues this warning when the result of the call to malloc(), calloc() or realloc() is discarded, and therefore not free()d, resulting in a memory leak.

MEM08-C Use realloc() only to resize dynamically allocated arrays

Only use realloc() to resize an array. Do not use it to transform an object to an object of a different type.

MEM30-C Do not access freed memory

When memory is freed, its contents may remain intact and accessible because it is at the memory manager's discretion when to reallocate or recycle the freed chunk. The data at the freed location may appear valid. However, this can change unexpectedly, leading to unintended program behavior. As a result, it is necessary to guarantee that memory is not written to or read from once it is freed.

MEM31-C Free dynamically allocated memory exactly once

Freeing memory multiple times has similar consequences to accessing memory after it is freed. The underlying data structures that manage the heap can become corrupted. To eliminate double-free vulnerabilities, it is necessary to guarantee that dynamic memory is freed exactly once.

MEM32-C Detect and handle memory allocation errors

The result of realloc() is assigned to the original pointer, without checking for failure. As a result, the original block of memory is lost when realloc() fails.

MEM33-C Use the correct syntax for flexible array members

Use the ISO C99 syntax for flexible array members instead of an array member of size 1.

MEM34-C Only free memory allocated dynamically

Freeing memory that is not allocated dynamically can lead to corruption of the heap data structures.

MEM35-C Allocate sufficient memory for an object

The Safety Checker issues this warning when the size of the object(s) allocated by malloc(), calloc() or realloc() is smaller than the size of an object pointed to by the result pointer. This may be caused by a sizeof expression with the wrong type or with a pointer type instead of the object type.

8.9. Environment (ENV)

ENV32-C All atexit handlers must return normally

The Safety Checker issues this warning when an atexit() handler is calling a function that does not return. No atexit() registered handler should terminate in any way other than by returning.

8.10. Signals (SIG)

SIG30-C Call only asynchronous-safe functions within signal handlers

SIG32-C Do not call longjmp() from inside a signal handler

Invoking the longjmp() function from within a signal handler can lead to undefined behavior if it results in the invocation of any non-asynchronous-safe functions, likely compromising the integrity of the program.

8.11. Miscellaneous (MSC)

MSC32-C Ensure your random number generator is properly seeded

Ensure that the random number generator is properly seeded by calling srand().

Chapter 9. MISRA C Rules

This chapter contains an overview of the supported and unsupported MISRA C rules.

9.1. MISRA C:1998

This section lists all supported and unsupported MISRA C:1998 rules.

See also Section 4.8.2, *C Code Checking: MISRA C*.

A number of MISRA C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the Safety Checker by TASKING. (R) is a required rule, (A) is an advisory rule.

1. (R) The code shall conform to standard C, without language extensions.

x 2. (A) Other languages should only be used with an interface standard.

3. (A) Inline assembly is only allowed in dedicated C functions.

x 4. (A) Provision should be made for appropriate run-time checking.

5. (R) Only use characters and escape sequences defined by ISO C.

x 6. (R) Character values shall be restricted to a subset of ISO 106460-1.

7. (R) Trigraphs shall not be used.

8. (R) Multibyte characters and wide string literals shall not be used.

9. (R) Comments shall not be nested.

10. (R) Sections of code should not be "commented out".

In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment:

- a line ends with ';', or
- a line starts with '}', possibly preceded by white space

11. (R) Identifiers shall not rely on significance of more than 31 characters.

12. (A) The same identifier shall not be used in multiple name spaces.

13. (A) Specific-length typedefs should be used instead of the basic types.

14. (R) Use `unsigned char` or `signed char` instead of plain `char`.

x 15. (A) Floating-point implementations should comply with a standard.

16. (R) The bit representation of floating-point numbers shall not be used.
A violation is reported when a pointer to a floating-point type is converted to a pointer to an integer type.

17. (R) `typedef` names shall not be reused.

18. (A) Numeric constants should be suffixed to indicate type.
A violation is reported when the value of the constant is outside the range indicated by the suffixes, if any.

19. (R) Octal constants (other than zero) shall not be used.

20. (R) All object and function identifiers shall be declared before use.

21. (R) Identifiers shall not hide identifiers in an outer scope.

22. (A) Declarations should be at function scope where possible.

x 23. (A) All declarations at file scope should be static where possible.

24. (R) Identifiers shall not have both internal and external linkage.

x 25. (R) Identifiers with external linkage shall have exactly one definition.

26. (R) Multiple declarations for objects or functions shall be compatible.

x 27. (A) External objects should not be declared in more than one file.

28. (A) The `register` storage class specifier should not be used.

29. (R) The use of a tag shall agree with its declaration.

30. (R) All automatics shall be initialized before being used .
This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.

31. (R) Braces shall be used in the initialization of arrays and structures.

32. (R) Only the first, or all enumeration constants may be initialized.

33. (R) The right hand operand of && or `||` shall not contain side effects.

34. (R) The operands of a logical && or `||` shall be primary expressions.

35. (R) Assignment operators shall not be used in Boolean expressions.

36. (A) Logical operators should not be confused with bitwise operators.

37. (R) Bitwise operations shall not be performed on signed integers.

38. (R) A shift count shall be between 0 and the operand width minus 1.
This violation will only be checked when the shift count evaluates to a constant value at compile time.

39. (R) The unary minus shall not be applied to an unsigned expression.

40. (A) `sizeof` should not be used on expressions with side effects.

x 41. (A) The implementation of integer division should be documented.

42. (R) The comma operator shall only be used in a `for` condition.

43. (R) Don't use implicit conversions which may result in information loss.

44. (A) Redundant explicit casts should not be used.

45. (R) Type casting from any type to or from pointers shall not be used.

46. (R) The value of an expression shall be evaluation order independent.
    This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.
47. (A) No dependence should be placed on operator precedence rules.
48. (A) Mixed arithmetic should use explicit casting.
49. (A) Tests of a (non-Boolean) value against 0 should be made explicit.
50. (R) F.P. variables shall not be tested for exact equality or inequality.
51. (A) Constant unsigned integer expressions should not wrap-around.
52. (R) There shall be no unreachable code.
53. (R) All non-null statements shall have a side-effect.
54. (R) A null statement shall only occur on a line by itself.
55. (A) Labels should not be used.
56. (R) The `goto` statement shall not be used.
57. (R) The `continue` statement shall not be used.
58. (R) The `break` statement shall not be used (except in a `switch`).
59. (R) An `if` or loop body shall always be enclosed in braces.
60. (A) All `if`, `else if` constructs should contain a final `else`.
61. (R) Every non-empty `case` clause shall be terminated with a `break`.
62. (R) All `switch` statements should contain a final `default` case.
63. (A) A `switch` expression should not represent a Boolean case.
64. (R) Every `switch` shall have at least one `case`.
65. (R) Floating-point variables shall not be used as loop counters.
66. (A) A `for` should only contain expressions concerning loop control.
    A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.
67. (A) Iterator variables should not be modified in a `for` loop.
68. (R) Functions shall always be declared at file scope.
69. (R) Functions with variable number of arguments shall not be used.
70. (R) Functions shall not call themselves, either directly or indirectly.
    A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.
71. (R) Function prototypes shall be visible at the definition and call.
72. (R) The function prototype of the declaration shall match the definition.
73. (R) Identifiers shall be given for all prototype parameters or for none.
74. (R) Parameter identifiers shall be identical for declaration/definition.
75. (R) Every function shall have an explicit return type.

76. (R) Functions with no parameters shall have a `void` parameter list.

77. (R) An actual parameter type shall be compatible with the prototype.

78. (R) The number of actual parameters shall match the prototype.

79. (R) The values returned by `void` functions shall not be used.

80. (R) Void expressions shall not be passed as function parameters.

81. (A) `const` should be used for reference parameters not modified.

82. (A) A function should have a single point of exit.

83. (R) Every exit point shall have a `return` of the declared return type.

84. (R) For `void` functions, `return` shall not have an expression.

85. (A) Function calls with no parameters should have empty parentheses.

86. (A) If a function returns error information, it should be tested.
A violation is reported when the return value of a function is ignored.

87. (R) `#include` shall only be preceded by other directives or comments.

88. (R) Non-standard characters shall not occur in `#include` directives.

89. (R) `#include` shall be followed by either `<filename>` or `"filename"`.

90. (R) Plain macros shall only be used for constants/qualifiers/specifiers.

91. (R) Macros shall not be `#define`'d and `#undef`'d within a block.

92. (A) `#undef` should not be used.

93. (A) A function should be used in preference to a function-like macro.

94. (R) A function-like macro shall not be used without all arguments.

95. (R) Macro arguments shall not contain pre-preprocessing directives.
A violation is reported when the first token of an actual macro argument is '#'.

96. (R) Macro definitions/parameters should be enclosed in parentheses.

97. (A) Don't use undefined identifiers in pre-processing directives.

98. (R) A macro definition shall contain at most one # or ## operator.

99. (R) All uses of the `#pragma` directive shall be documented.
This rule is really a documentation issue. The Safety Checker will flag all `#pragma` directives as violations.

100. (R) `defined` shall only be used in one of the two standard forms.

101. (A) Pointer arithmetic should not be used.

102. (A) No more than 2 levels of pointer indirection should be used.
A violation is reported when a pointer with three or more levels of indirection is declared.

103. (R) No relational operators between pointers to different objects.
In general, checking whether two pointers point to the same object is impossible. The Safety Checker will only report a violation for a relational operation with incompatible pointer types.

104. (R) Non-constant pointers to functions shall not be used.

105. (R) Functions assigned to the same pointer shall be of identical type.

106. (R) Automatic address may not be assigned to a longer lived object.

107. (R) The null pointer shall not be de-referenced.
A violation is reported for every pointer dereference that is not guarded by a NULL pointer test.

108. (R) All `struct`/`union` members shall be fully specified.

109. (R) Overlapping variable storage shall not be used.
A violation is reported for every `union` declaration.

110. (R) Unions shall not be used to access the sub-parts of larger types.
A violation is reported for a `union` containing a `struct` member.

111. (R) Bit-fields shall have type `unsigned int` or `signed int`.

112. (R) Bit-fields of type `signed int` shall be at least 2 bits long.

113. (R) All `struct`/`union` members shall be named.

114. (R) Reserved and standard library names shall not be redefined.

115. (R) Standard library function names shall not be reused.

x 116. (R) Production libraries shall comply with the MISRA C restrictions.

x 117. (R) The validity of library function parameters shall be checked.

118. (R) Dynamic heap memory allocation shall not be used.

119. (R) The error indicator `errno` shall not be used.

120. (R) The macro `offsetof` shall not be used.

121. (R) `<locale.h>` and the `setlocale` function shall not be used.

122. (R) The `setjmp` and `longjmp` functions shall not be used.

123. (R) The signal handling facilities of `<signal.h>` shall not be used.

124. (R) The `<stdio.h>` library shall not be used in production code.

125. (R) The functions `atof`/`atoi`/`atol` shall not be used.

126. (R) The functions `abort`/`exit`/`getenv`/`system` shall not be used.

127. (R) The time handling functions of library `<time.h>` shall not be used.

9.2. MISRA C:2004

This section lists all supported and unsupported MISRA C:2004 rules.

See also Section 4.8.2, *C Code Checking: MISRA C*.

A number of MISRA C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the Safety Checker by TASKING. (R) is a required rule, (A) is an advisory rule.

Environment

1.1 (R) All code shall conform to ISO 9899:1990 "Programming languages - C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.

1.2 (R) No reliance shall be placed on undefined or unspecified behavior.

x 1.3 (R) Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.

x 1.4 (R) The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.

x 1.5 (A) Floating-point implementations should comply with a defined floating-point standard.

Language extensions

2.1 (R) Assembly language shall be encapsulated and isolated.

2.2 (R) Source code shall only use `/* ... */` style comments.

2.3 (R) The character sequence `/*` shall not be used within a comment.

2.4 (A) Sections of code should not be "commented out". In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment: - a line ends with ';', or - a line starts with '}', possibly preceded by white space

Documentation

x 3.1 (R) All usage of implementation-defined behavior shall be documented.

x 3.2 (R) The character set and the corresponding encoding shall be documented.

x 3.3 (A) The implementation of integer division in the chosen compiler should be determined, documented and taken into account.

3.4 (R) All uses of the `#pragma` directive shall be documented and explained. This rule is really a documentation issue. The Safety Checker will flag all `#pragma` directives as violations.

3.5 (R) The implementation-defined behavior and packing of bit-fields shall be documented if being relied upon.

x 3.6 (R) All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.

Character sets

4.1 (R) Only those escape sequences that are defined in the ISO C standard shall be used.

4.2 (R) Trigraphs shall not be used.

Identifiers

5.1 (R) Identifiers (internal and external) shall not rely on the significance of more than 31 characters.

5.2 (R) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

5.3 (R) A `typedef` name shall be a unique identifier.

5.4 (R) A tag name shall be a unique identifier.

5.5 (A) No object or function identifier with static storage duration should be reused.

5.6 (A) No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.

5.7 (A) No identifier name should be reused.

Types

6.1 (R) The plain `char` type shall be used only for storage and use of character values.

6.2 (R) `signed` and `unsigned char` type shall be used only for the storage and use of numeric values.

6.3 (A) `typedefs` that indicate size and signedness should be used in place of the basic types.

6.4 (R) Bit-fields shall only be defined to be of type `unsigned int` or `signed int`.

6.5 (R) Bit-fields of type `signed int` shall be at least 2 bits long.

Constants

7.1 (R) Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and definitions

8.1 (R) Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.

8.2 (R) Whenever an object or function is declared or defined, its type shall be explicitly stated.

8.3 (R) For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.

8.4 (R) If objects or functions are declared more than once their types shall be compatible.

8.5 (R) There shall be no definitions of objects or functions in a header file.

8.6 (R) Functions shall be declared at file scope.

8.7 (R) Objects shall be defined at block scope if they are only accessed from within a single function.

8.8 (R) An external object or function shall be declared in one and only one file.

8.9 (R) An identifier with external linkage shall have exactly one external definition.

x 8.10 (R) All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.

8.11 (R) The `static` storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.

8.12 (R) When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

9.1 (R) All automatic variables shall have been assigned a value before being used. This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.

9.2 (R) Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

9.3 (R) In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic type conversions

10.1 (R) The value of an expression of integer type shall not be implicitly converted to a different underlying type if:
a) it is not a conversion to a wider integer type of the same signedness, or
b) the expression is complex, or
c) the expression is not constant and is a function argument, or
d) the expression is not constant and is a return expression.

10.2 (R) The value of an expression of floating type shall not be implicitly converted to a different type if:
a) it is not a conversion to a wider floating type, or
b) the expression is complex, or
c) the expression is a function argument, or
d) the expression is a return expression.

10.3 (R) The value of a complex expression of integer type may only be cast to a type of the same signedness that is no wider than the underlying type of the expression.

10.4 (R) The value of a complex expression of floating type may only be cast to a type that is no wider than the underlying type of the expression.

10.5 (R) If the bitwise operators ~ and << are applied to an operand of underlying type `unsigned char` or `unsigned short`, the result shall be immediately cast to the underlying type of the operand.

10.6 (R) A "U" suffix shall be applied to all constants of `unsigned` type.

Pointer type conversions

11.1 (R) Conversions shall not be performed between a pointer to a function and any type other than an integral type.

11.2 (R) Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.

11.3 (A) A cast should not be performed between a pointer type and an integral type.

11.4 (A) A cast should not be performed between a pointer to object type and a different pointer to object type.

11.5 (R) A cast shall not be performed that removes any `const` or `volatile` qualification from the type addressed by a pointer.

Expressions

12.1 (A) Limited dependence should be placed on C's operator precedence rules in expressions.

12.2 (R) The value of an expression shall be the same under any order of evaluation that the standard permits. This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.

12.3 (R) The `sizeof` operator shall not be used on expressions that contain side effects.

12.4 (R) The right-hand operand of a logical && or `||` operator shall not contain side effects.

12.5 (R) The operands of a logical && or `||` shall be *primary-expressions*.

12.6 (A) The operands of logical operators (&&, `||` and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (&&, `||` and `!`).

12.7 (R) Bitwise operators shall not be applied to operands whose underlying type is signed.

12.8 (R) The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand. This violation will only be checked when the shift count evaluates to a constant value at compile time.

12.9 (R) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

12.10 (R) The comma operator shall not be used.

12.11 (A) Evaluation of constant unsigned integer expressions should not lead to wrap-around.

12.12 (R) The underlying bit representations of floating-point values shall not be used. A violation is reported when a pointer to a floating-point type is converted to a pointer to an integer type.

12.13 (A) The increment (++) and decrement (`--`) operators should not be mixed with other operators in an expression.

Control statement expressions

13.1 (R) Assignment operators shall not be used in expressions that yield a Boolean value.

13.2 (A) Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.

13.3 (R) Floating-point expressions shall not be tested for equality or inequality.

13.4 (R) The controlling expression of a `for` statement shall not contain any objects of floating type.

13.5 (R) The three expressions of a `for` statement shall be concerned only with loop control. A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.

13.6 (R) Numeric variables being used within a `for` loop for iteration counting shall not be modified in the body of the loop.

13.7 (R) Boolean operations whose results are invariant shall not be permitted.

Control flow

14.1 (R) There shall be no unreachable code.

14.2 (R) All non-null statements shall either:
a) have at least one side effect however executed, or
b) cause control flow to change.

14.3 (R) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.

14.4 (R) The `goto` statement shall not be used.

14.5 (R) The `continue` statement shall not be used.

14.6 (R) For any iteration statement there shall be at most one break statement used for loop termination.

14.7 (R) A function shall have a single point of exit at the end of the function.

14.8 (R) The statement forming the body of a `switch`, `while`, `do ... while` or `for` statement be a compound statement.

14.9 (R) An `if (`*`expression`*`)` construct shall be followed by a compound statement. The `else` keyword shall be followed by either a compound statement, or another `if` statement.

14.10 (R) All `if ... else if` constructs shall be terminated with an `else` clause.

Switch statements

15.1 (R) A switch label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.

15.2 (R) An unconditional `break` statement shall terminate every non-empty `switch` clause.

15.3 (R) The final clause of a switch statement shall be the `default` clause.

15.4 (R) A `switch` expression shall not represent a value that is effectively Boolean.

15.5 (R) Every `switch` statement shall have at least one `case` clause.

Functions

16.1 (R) Functions shall not be defined with variable numbers of arguments.

16.2 (R) Functions shall not call themselves, either directly or indirectly. A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.

16.3 (R) Identifiers shall be given for all of the parameters in a function prototype declaration.

16.4 (R) The identifiers used in the declaration and definition of a function shall be identical.

16.5 (R) Functions with no parameters shall be declared with parameter type `void`.

16.6 (R) The number of arguments passed to a function shall match the number of parameters.

16.7 (A) A pointer parameter in a function prototype should be declared as pointer to `const` if the pointer is not used to modify the addressed object.

16.8 (R) All exit paths from a function with non-void return type shall have an explicit `return` statement with an expression.

16.9 (R) A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.

16.10 (R) If a function returns error information, then that error information shall be tested. A violation is reported when the return value of a function is ignored.

Pointers and arrays

x 17.1 (R) Pointer arithmetic shall only be applied to pointers that address an array or array element.

x 17.2 (R) Pointer subtraction shall only be applied to pointers that address elements of the same array.

17.3 (R) >, >=, <, <= shall not be applied to pointer types except where they point to the same array. In general, checking whether two pointers point to the same object is impossible. The Safety Checker will only report a violation for a relational operation with incompatible pointer types.

17.4 (R) Array indexing shall be the only allowed form of pointer arithmetic.

17.5 (A) The declaration of objects should contain no more than 2 levels of pointer indirection. A violation is reported when a pointer with three or more levels of indirection is declared.

17.6 (R) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Structures and unions

18.1 (R) All structure or union types shall be complete at the end of a translation unit.

18.2 (R) An object shall not be assigned to an overlapping object.

x 18.3 (R) An area of memory shall not be reused for unrelated purposes.

18.4 (R) Unions shall not be used.

Preprocessing directives

19.1 (A) `#include` statements in a file should only be preceded by other preprocessor directives or comments.

19.2 (A) Non-standard characters should not occur in header file names in `#include` directives.

x 19.3 (R) The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence.

19.4 (R) C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

19.5 (R) Macros shall not be `#define`'d or `#undef`'d within a block.

19.6 (R) `#undef` shall not be used.

19.7 (A) A function should be used in preference to a function-like macro.

19.8 (R) A function-like macro shall not be invoked without all of its arguments.

19.9 (R) Arguments to a function-like macro shall not contain tokens that look like preprocessing directives. A violation is reported when the first token of an actual macro argument is '#'.

19.10 (R) In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of # or ##.

19.11 (R) All macro identifiers in preprocessor directives shall be defined before use, except in `#ifdef` and `#ifndef` preprocessor directives and the `defined()` operator.

19.12 (R) There shall be at most one occurrence of the # or ## preprocessor operators in a single macro definition.

19.13 (A) The # and ## preprocessor operators should not be used.

19.14 (R) The `defined` preprocessor operator shall only be used in one of the two standard forms.

19.15 (R) Precautions shall be taken in order to prevent the contents of a header file being included twice.

19.16 (R) Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

19.17 (R) All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.

Standard libraries

20.1 (R) Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.

20.2 (R) The names of standard library macros, objects and functions shall not be reused.

x 20.3 (R) The validity of values passed to library functions shall be checked.

20.4 (R) Dynamic heap memory allocation shall not be used.

20.5 (R) The error indicator `errno` shall not be used.

20.6 (R) The macro `offsetof`, in library `<stddef.h>`, shall not be used.

20.7 (R) The `setjmp` macro and the `longjmp` function shall not be used.

20.8 (R) The signal handling facilities of `<signal.h>` shall not be used.

20.9 (R) The input/output library `<stdio.h>` shall not be used in production code.

20.10 (R) The library functions `atof`, `atoi` and `atol` from library `<stdlib.h>` shall not be used.

20.11 (R) The library functions `abort`, `exit`, `getenv` and `system` from library `<stdlib.h>` shall not be used.

20.12 (R) The time handling functions of library `<time.h>` shall not be used.

Run-time failures

x 21.1 (R) Minimization of run-time failures shall be ensured by the use of at least one of:
a) static analysis tools/techniques;
b) dynamic analysis tools/techniques;
c) explicit coding of checks to handle run-time faults.

9.3. MISRA C:2012

This section lists all supported and unsupported MISRA C:2012 rules.

See also Section 4.8.2, *C Code Checking: MISRA C*.

A number of MISRA C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the Safety Checker by TASKING. (M) is a mandatory rule, (R) is a required rule, (A) is an advisory rule.

A standard C environment

1.1 (R) The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.

1.2 (A) Language extensions should not be used.

1.3 (R) There shall be no occurrence of undefined or critical unspecified behavior.

Unused code

2.1 (R) A project shall not contain unreachable code.

2.2 (R) There shall be no dead code.

2.3 (A) A project should not contain unused type declarations.

2.4 (A) A project should not contain unused tag declarations.

2.5 (A) A project should not contain unused macro declarations.

2.6 (A) A function should not contain unused label declarations.

2.7 (A) There should be no unused parameters in functions.

Comments

3.1 (R) The character sequences `/*` and `//` shall not be used within a comment.

3.2 (R) Line-splicing shall not be used in `//` comments.

Character sets and lexical conventions

4.1 (R) Octal and hexadecimal escape sequences shall be terminated.

4.2 (A) Trigraphs should not be used.

Identifiers

x 5.1 (R) External identifiers shall be distinct.

x 5.2 (R) Identifiers declared in the same scope and name space shall be distinct.

x 5.3 (R) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.

x 5.4 (R) Macro identifiers shall be distinct.

x 5.5 (R) Identifiers shall be distinct from macro names.

x 5.6 (R) A `typedef` name shall be a unique identifier.

x 5.7 (R) A tag name shall be a unique identifier.

x 5.8 (R) Identifiers that define objects or functions with external linkage shall be unique.

x 5.9 (A) Identifiers that define objects or functions with internal linkage should be unique.

Types

6.1 (R) Bit-fields shall only be declared with an appropriate type.

6.2 (R) Single-bit named bit-fields shall not be of a signed type.

Literals and constants

7.1 (R) Octal constants shall not be used.

7.2 (R) A "u" or "`U`" suffix shall be applied to all integer constants that are represented in an `unsigned` type.

7.3 (R) The lowercase character "`l`" shall not be used in a literal suffix trivial.

7.4 (R) A string literal shall not be assigned to an object unless the object's type is "pointer to `const`-qualified `char`".

Declarations and definitions

8.1 (R) Types shall be explicitly specified.

8.2 (R) Function types shall be in prototype form with named parameters.

8.3 (R) All declarations of an object or function shall use the same names and type qualifiers.

8.4 (R) A compatible declaration shall be visible when an object or function with external linkage is defined.

8.5 (R) An external object or function shall be declared once in one and only one file.

8.6 (R) An identifier with external linkage shall have exactly one external definition.

8.7 (A) Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.

8.8 (R) The `static` storage class specifier shall be used in all declarations of objects and functions that have internal linkage.

8.9 (A) An object should be defined at block scope if its identifier only appears in a single function.

8.10 (R) An inline function shall be declared with the `static` storage class.

8.11 (A) When an array with external linkage is declared, its size should be explicitly specified.

8.12 (R) Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.

8.13 (A) A pointer should point to a `const`-qualified type whenever possible.

8.14 (R) The `restrict` type qualifier shall not be used.

Initialization

9.1 (M) The value of an object with automatic storage duration shall not be read before it has been set.

9.2 (R) The initializer for an aggregate or union shall be enclosed in braces.

9.3 (R) Arrays shall not be partially initialized.

9.4 (R) An element of an object shall not be initialized more than once.

9.5 (R) Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The essential type model

10.1 (R) Operands shall not be of an inappropriate essential type.

10.2 (R) Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.

10.3 (R) The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.

10.4 (R) Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.

10.5 (A) The value of an expression should not be cast to an inappropriate essential type.

10.6 (R) The value of a composite expression shall not be assigned to an object with wider essential type.

10.7 (R) If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.

10.8 (R) The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer type conversions

11.1 (R) Conversions shall not be performed between a pointer to a function and any other type.

11.2 (R) Conversions shall not be performed between a pointer to an incomplete type and any other type.

11.3 (R) A cast shall not be performed between a pointer to object type and a pointer to a different object type.

11.4 (A) A conversion should not be performed between a pointer to object and an integer type.

11.5 (A) A conversion should not be performed from pointer to `void` into pointer to object.

11.6 (R) A cast shall not be performed between pointer to `void` and an arithmetic type.

11.7 (R) A cast shall not be performed between pointer to object and a non-integer arithmetic type.

11.8 (R) A cast shall not remove any `const` or `volatile` qualification from the type pointed to by a pointer.

11.9 (R) The macro `NULL` shall be the only permitted form of integer null pointer constant.

Expressions

12.1 (A) The precedence of operators within expressions should be made explicit.

12.2 (R) The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.

12.3 (A) The comma operator should not be used.

12.4 (A) Evaluation of constant expressions should not lead to unsigned integer wrap-around.

Side effects

13.1 (R) Initializer lists shall not contain persistent side effects.

13.2 (R) The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.

13.3 (A) A full expression containing an increment (`++`) or decrement (`--`) operator should have no other potential side effects other than that caused by the increment or decrement operator.

13.4 (A) The result of an assignment operator should not be used.

13.5 (R) The right hand operand of a logical `&&` or `||` operator shall not contain persistent side effects.

13.6 (M) The operand of the `sizeof` operator shall not contain any expression which has potential side effects.

Control statement expressions

14.1 (R) A loop counter shall not have essentially floating type.

14.2 (R) A `for` loop shall be well-formed.

14.3 (R) Controlling expressions shall not be invariant.

14.4 (R) The controlling expression of an `if` statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control flow

15.1 (A) The `goto` statement should not be used.

15.2 (R) The `goto` statement shall jump to a label declared later in the same function.

15.3 (R) Any label referenced by a `goto` statement shall be declared in the same block, or in any block enclosing the `goto` statement.

15.4 (A) There should be no more than one `break` or `goto` statement used to terminate any iteration statement.

15.5 (A) A function should have a single point of exit at the end.

15.6 (R) The body of an iteration-statement or a selection-statement shall be a compound-statement.

15.7 (R) All `if ... else if` constructs shall be terminated with an `else` statement.

Switch statements

16.1 (R) All `switch` statements shall be well-formed.

16.2 (R) A switch label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.

16.3 (R) An unconditional `break` statement shall terminate every switch-clause.

16.4 (R) Every `switch` statement shall have a `default` label.

16.5 (R) A `default` label shall appear as either the first or the last switch label of a `switch` statement.

16.6 (R) Every `switch` statement shall have at least two switch-clauses.

16.7 (R) A switch-expression shall not have essentially Boolean type.

Functions

17.1 (R) The features of `<stdarg.h>` shall not be used.

17.2 (R) Functions shall not call themselves, either directly or indirectly.

17.3 (M) A function shall not be declared implicitly.

17.4 (M) All exit paths from a function with non-`void` return type shall have an explicit `return` statement with an expression.

17.5 (A) The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

17.6 (M) The declaration of an array parameter shall not contain the `static` keyword between the `[ ]`.

17.7 (R) The value returned by a function having non-`void` return type shall be used.

17.8 (A) A function parameter should not be modified.

Pointers and arrays

18.1 (R) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

18.2 (R) Subtraction between pointers shall only be applied to pointers that address elements of the same array.

18.3 (R) The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object.

18.4 (A) The `+`, `-`, `+=` and `-=` operators should not be applied to an expression of pointer type.

18.5 (A) Declarations should contain no more than two levels of pointer nesting.

18.6 (R) The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.

18.7 (R) Flexible array members shall not be declared.

18.8 (R) Variable-length array types shall not be used.

Overlapping storage

19.1 (M) An object shall not be assigned or copied to an overlapping object.

19.2 (A) The `union` keyword should not be used.

Preprocessing directives

20.1 (A) `#include` directives should only be preceded by preprocessor directives or comments.

20.2 (R) The `'`, `"` or `\` characters and the `/*` or `//` character sequences shall not occur in a header file name.

20.3 (R) The `#include` directive shall be followed by either a `<filename>` or `"filename"` sequence.

20.4 (R) A macro shall not be defined with the same name as a keyword.

20.5 (A) `#undef` should not be used.

20.6 (R) Tokens that look like a preprocessing directive shall not occur within a macro argument

20.7 (R) Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.

20.8 (R) The controlling expression of a `#if` or `#elif` preprocessing directive shall evaluate to 0 or 1.

20.9 (R) All identifiers used in the controlling expression of `#if` or `#elif` preprocessing directives shall be `#define`'d before evaluation.

20.10 (A) The # and ## preprocessor operators should not be used.

20.11 (R) A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.

20.12 (R) A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.

20.13 (R) A line whose first token is # shall be a valid preprocessing directive.

20.14 (R) All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if`, `#ifdef` or `#ifndef` directive to which they are related.

Standard libraries

21.1 (R) `#define` and `#undef` shall not be used on a reserved identifier or reserved macro name.

21.2 (R) A reserved identifier or macro name shall not be declared.

21.3 (R) The memory allocation and deallocation functions of `<stdlib.h>` shall not be used.

21.4 (R) The standard header file `<setjmp.h>` shall not be used.

21.5 (R) The standard header file `<signal.h>` shall not be used.

21.6 (R) The Standard Library input/output functions shall not be used.

21.7 (R) The `atof`, `atoi`, `atol` and `atoll` functions of `<stdlib.h>` shall not be used.

21.8 (R) The library functions `abort`, `exit`, `getenv` and `system` of `<stdlib.h>` shall not be used.

21.9 (R) The library functions `bsearch` and `qsort` of `<stdlib.h>` shall not be used.

21.10 (R) The Standard Library time and date functions shall not be used

21.11 (R) The standard header file `<tgmath.h>` shall not be used.

21.12 (A) The exception handling features of `<fenv.h>` should not be used.

Resources

x 22.1 (R) All resources obtained dynamically by means of Standard Library functions shall be explicitly released.

x	22.2	(M)	A block of memory shall only be freed if it was allocated by means of a Standard Library function.
x	22.3	(R)	The same file shall not be open for read and write access at the same time on different streams.
x	22.4	(M)	There shall be no attempt to write to a stream which has been opened as read-only.
x	22.5	(M)	A pointer to a `FILE` object shall not be dereferenced.
x	22.6	(M)	The value of a pointer to a `FILE` shall not be used after the associated stream has been closed.