HighIntegritySystems

WITTENSTEIN

Using an MPU to Enforce Spatial Separation

Issue 1.3 - November 26, 2020

WITTENSTEIN high integrity systems
Americas: +1 408 625 4712
ROTW: +44 1275 395 600
Email: sales@high**integrity**systems.com
Web: www.high**integrity**systems.com

Contents

List of Figures

List of Notation

BSP Board Support Package
COTS Commercial off-the-shelf
DAP Design Assurance Pack
DHF Design History File
MCU Microcontroller Unit
MPU Memory Protection Unit
MMU Memory Management Unit
RTOS Real Time Operating System
SIL Safety Integrity Level
SOUP Software of Unknown Provenance

CHAPTER 1 Introduction

1.1 Introduction

In some industries, safety critical software has been in use for many years; however, increased regulation and the existence of domain specific safety development standards has led to a rapid growth in systems that use software classified as safety critical. The objective of all domain specific safety standards is to ensure that embedded system designs are robust, prevent harm or death occurring to users of the systems, and / or damage happening to surrounding equipment or the environment. Each application domain has slightly different use cases, which the safety standards take into account. The most used safety standards in embedded engineering are as follows:

- Industrial IEC 61508
- Medical IEC 62304 and FDA 510(k)
- Automotive ISO 26262
- Rail EN50128, EN50129
- Aerospace DO-178C

These safety standards typically define a range of safety levels. These safety levels classify the context the system is operating in, and define the amount of harm the system can potentially cause. The higher the safety level, the greater the potential harm, and therefore the more demanding the development life cycle becomes.

In many cases safety critical systems also have to support feature rich graphical interfaces, responsive networking communications, diagnostics, data storage and much more. For example, your typical medical device not only has to protect the patient and medical practitioner from harm, it must provide a good user experience, be easy to use, and communicate treatment data back to a healthcare center.

System designers face the challenge of providing safety and functionality as part of the same system. Due to the rigors of developing safety critical software the development costs are high and it would not be feasible to develop all the software used within the system to the highest safety level required. In addition, many software systems use third party components such as networking stacks and file systems - the development history of these components may be unknown, and hence these would only achieve a very low safety rating classification. In moderately complex systems, there may therefore be several different levels of safety software.

The software within the system needs partitioning to ensure that software from lower safety levels cannot interfere with software relating to the higher safety levels. Partitioning allows the safety related software to be small and concise, whilst allowing the use of third party software modules, thereby shortening development times and lowering costs.

This paper discusses techniques to achieve spatial separation and partitioning. Spatial separation is primarily concerned with ensuring that the accessing of physical system resources cannot lead to conflicts or corruption. Examples include access to Flash or RAM or system resources such as hardware peripherals. Other necessary forms of separation, such as temporal separation, will be the subject of a future white paper.

1.2 Use Case – A Simple Embedded System

For the purposes of this paper, we shall consider a moderately complex but typical embedded system as shown in Figure 1 1. From a software perspective, it includes components developed to different standards and Safety Integrity Levels (SIL), the system includes:

- Safety Critical Software (outlined in orange) – "Sensor Processing", "Control Logic" and the "Output Driver" contain the code that will implement the Safety Function of the application. Operation of this code must be entirely independent of the other code.
- Commercial grade third party software (outlined in green) – This is Software of Unknown Provenance (SOUP). We do not have access to the formal requirement or test documentation and it is either not possible or impractical to test this software to the required SIL of the product.
- Other software not developed to a required SIL.

It is important to note that the commercial components and 'other' software are not necessarily poor or functionally inadequate; however, when developing a safety system it is generally necessary to be able to prove that the software fully satisfies the requirements and that all software included in the project is necessary, complete and fully tested. This is typically not possible for third party components.

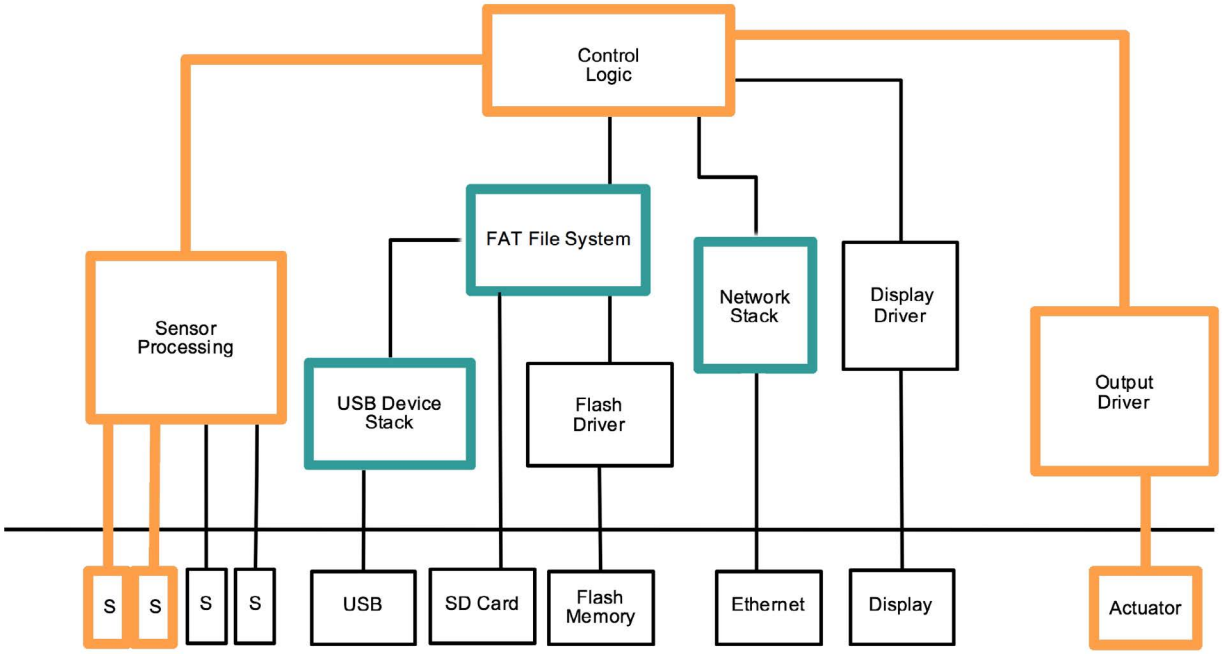

Figure 1-1 *A Typical Embedded System*

For this system to be implementable in a manner acceptable for a safety system, we need to be able to prove:

- Spatial separation between the safety code/data and the non-safety code/data. Spatial separation implies a clear separation between the safety and non-safety code and that the non-safety code cannot access any memory locations or peripheral components required by the safety code.
- Temporal separation between the safety code and the non-safety code. Temporal separation implies that the safety code has sufficient runtime to achieve its purpose and that this cannot be compromised by misbehaving or otherwise busy code.
- Data passed through non-safe stacks is either not safety related or protected. Where data received travels through unsafe channels (including software stacks and hardware communication busses), its integrity and validity must be assured before use in safety related processing.

This paper is concerned with the issues relating to spatial separation only, temporal separation and data integrity will be the subject of future white papers.

CHAPTER 2 System Architecture and its Effect on Spatial Separation

2.1 Spatial Separation with a Multi-Processor System

There are a number of different ways of designing the system described in the previous section. The classic approach is to split the software load across multiple processors, therefore we can have the arrangement shown in Figure 2-1 with the safety software hosted on one microprocessor and the support software hosted on a second 'non-safety' processor.

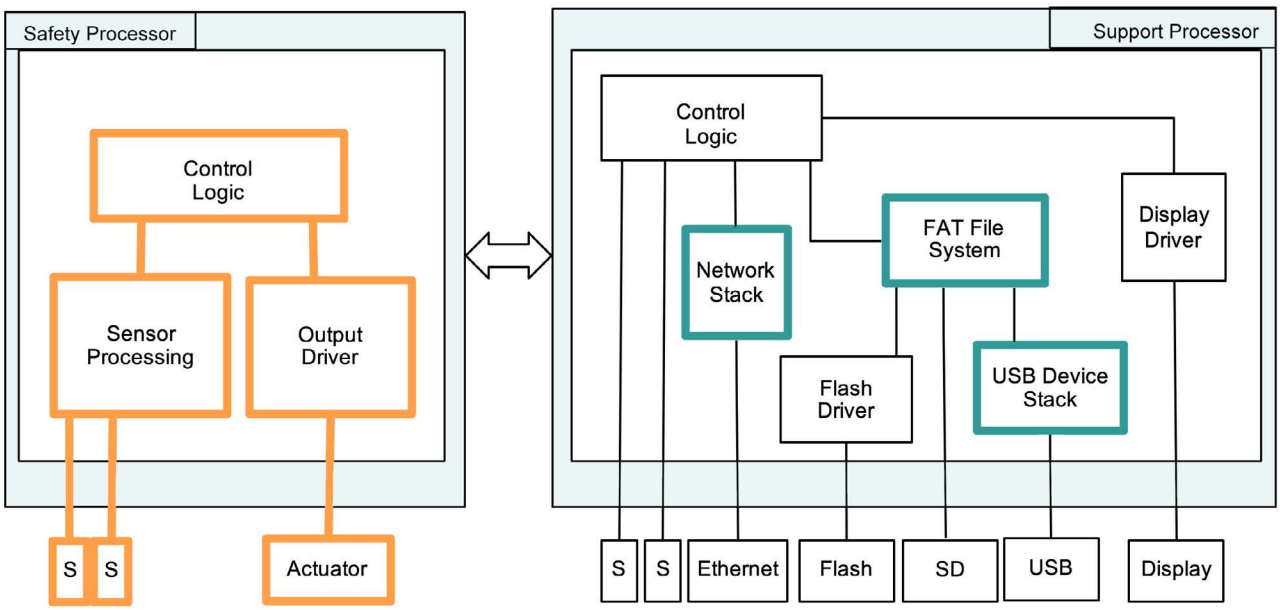

Figure 2-1 *A Multi-Processor System*

From the perspective of demonstrating and proving spatial separation, this is ideal, as there is clearly a physical separation between the CPU's and their associated memories. However, the increase in the cost of the hardware and the increased complexity of the hardware design may not be acceptable or desirable for all products.

2.2 Spatial Separation with a Multi-Core System

Multi-core processors are becoming increasingly available from many silicon vendors. These offer a solution where more processing power is available without increasing the complexity of the hardware design, as only one physical device has to be included. Figure 2-2 illustrates an architecture where the 'Safety Core' hosts the safety code and the 'Support' or 'Non-Safety' Core hosts the supporting software.

In a multi-core device, the system memories and peripherals that are located on the device are usually accessible or shared by all the CPU cores. This makes it difficult to claim any spatial separation between the safety and non-safety software as it is possible that misbehaving software in the non-safety core could affect the data used by the safety core. In this instance, a Memory Protection Unit (MPU) or Memory Management Unit (MMU) is useful to enforce spatial separation between the cores.

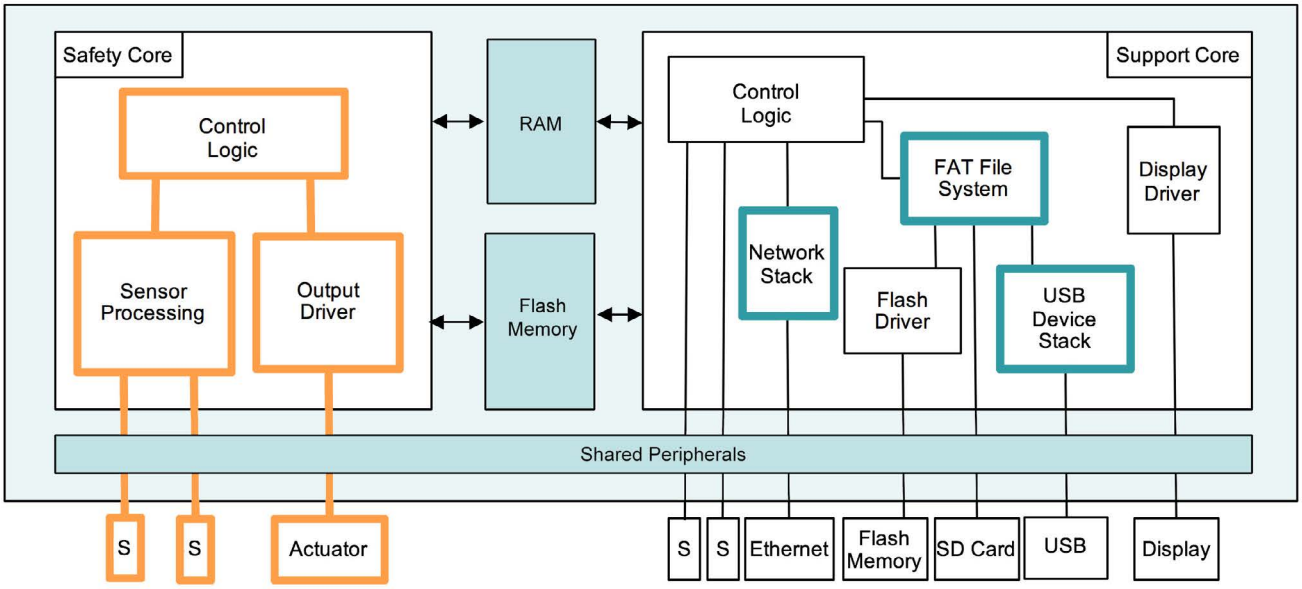

Figure 2-2 *A Multi-Core System*

2.3 Spatial Separation with a Single Core System

Where all the software is running on a single core, as shown in Figure 2-3, then clearly the architecture does not provide any spatial separation between software modules. However, with careful configuration of the MPU it may be possible to claim some spatial separation of safety and non-safety functionality.

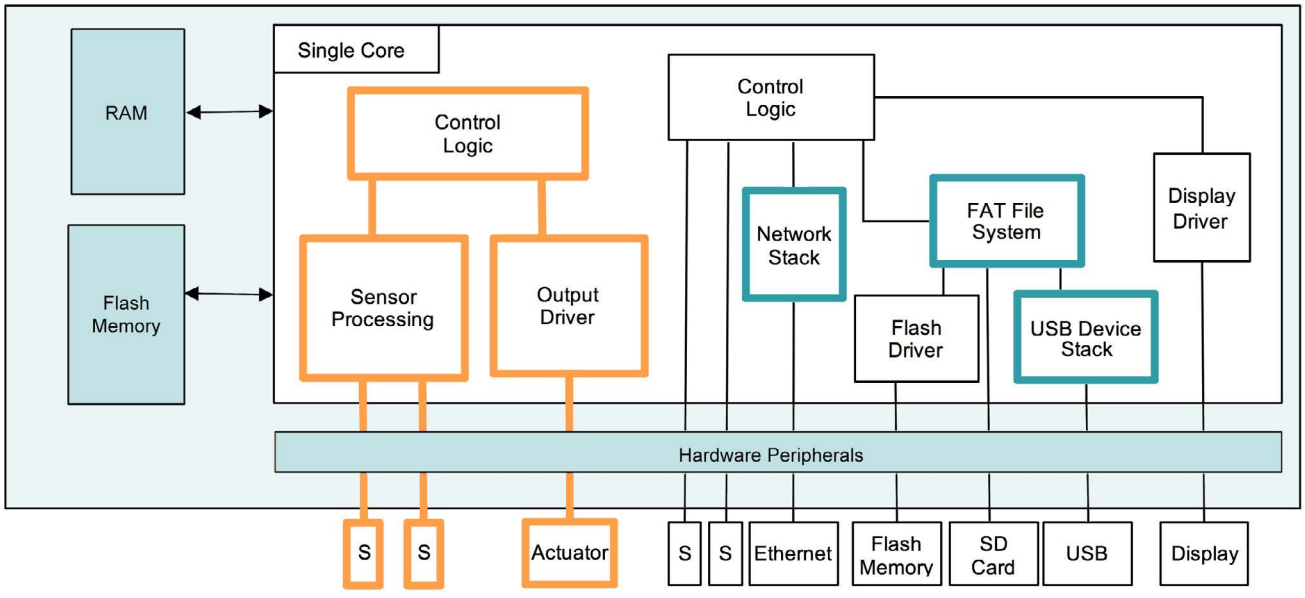

Figure 2-3 *A Single Core System*

CHAPTER 3 Achieving Spatial Separation using an MPU

3.1 Basic MPU Operation

Essentially the MPU monitors the CPU's access to the system memory space and triggers an exception if the attempted access is outside of some pre-defined memory range or if the current permissions settings are not sufficient to access the region. While different silicon manufacturers' MPU implementations vary slightly, most microprocessor MPUs allow the definition of a number of memory regions. An MPU region consists of a memory range and its associated access permissions.

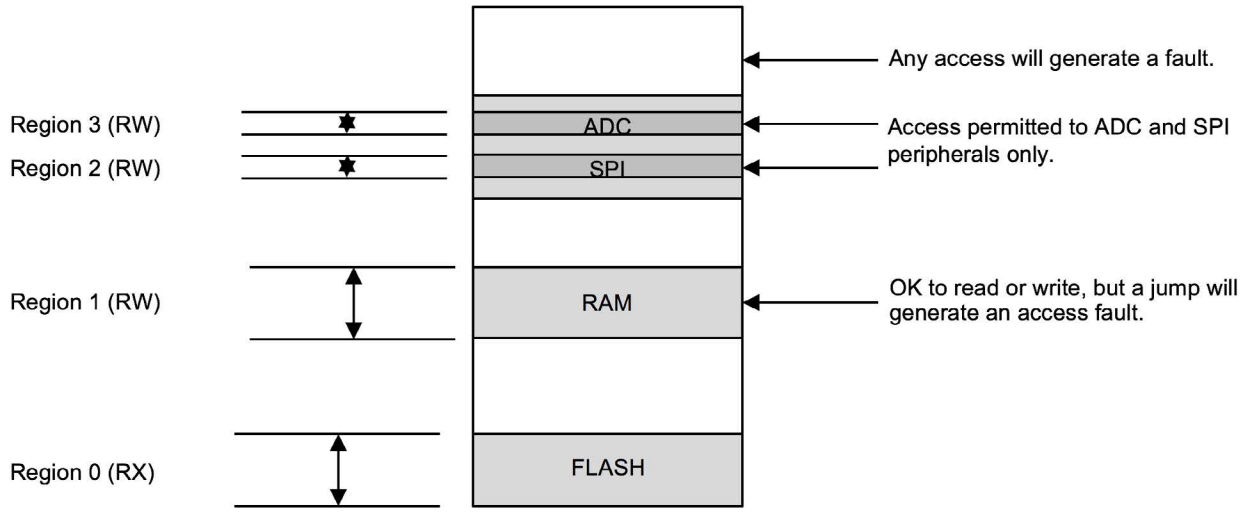

Figure 3-1 *Basic MPU Operation*

Figure 3-1 illustrates a basic MPU configuration. The configuration consists of four distinct MPU regions:

- A region corresponding to the Flash memory that has read and execute permissions;
- A region corresponding to the RAM that has read and write permissions;
- Two further regions that permit read and write access to the ADC and SPI peripheral registers.

Any access attempts to access outside of the defined regions will trigger an exception, and any attempts to write to flash memory or jump to the RAM region will also trigger exceptions.

3.2 Using a Basic MPU Configuration with a Multi-Core Architecture

Using separate (but complementary) configurations of the MPU for each core in a multi-core system, it is possible to achieve complete spatial separation between cores and therefore fulfil the requirements for the system configuration outlined in Section 2.2.

Figure 3-2 outlines a configuration where partitioning has been applied to the available memories in a multi-core device such that each core has dedicated access to its own memory space. Likewise, logical partitioning of the peripheral space ensures that each core has dedicated access to individual peripherals that are required. This arrangement satisfies partitioning requirements from a software perspective only and when considering the system as a whole other issues such as shared clocks, common power supplies etc. may need justifying.

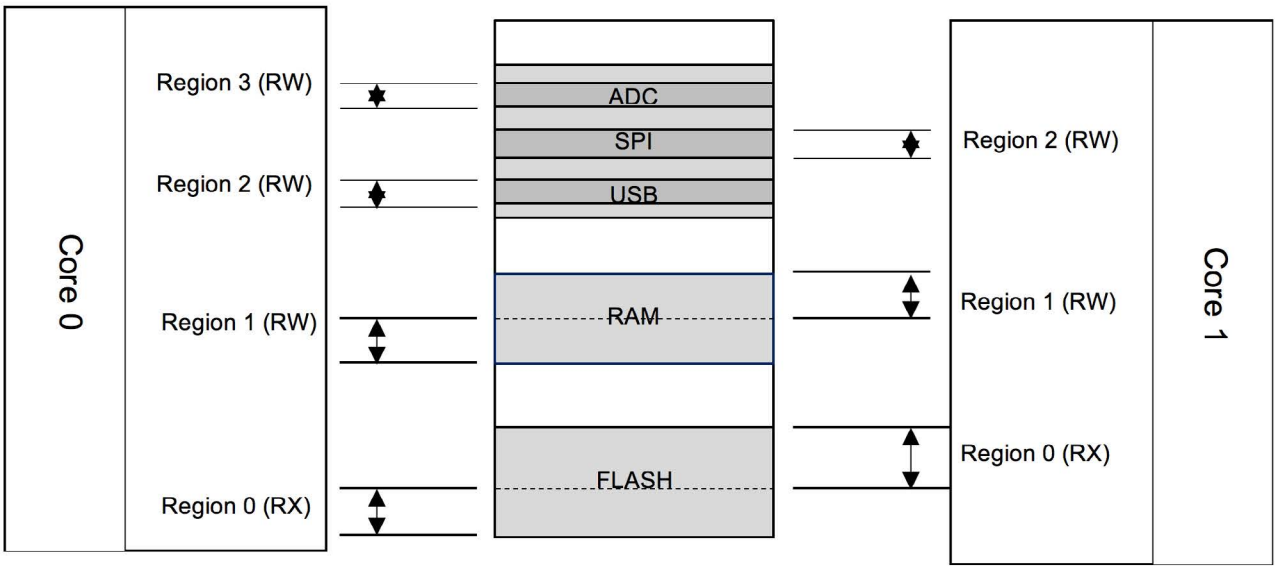

Figure 3-2 *Using Complementary MPU Settings to enforce Spatial Separation*

3.3 Simple Single Core Architecture

The use of the MPU in a single core architecture (see section 2.3) provides overall bounds checking of the software but a basic configuration will not provide any spatial separation between safety and non-safety software. This is because (by definition) all the software will share the same MPU settings or have the ability to modify the existing MPU settings.

Processor privilege modes are not used in many embedded systems because the added complexity of managing the modes and mode switching does not provide sufficient benefits; however when used with MPU configurations, this can be useful as different permissions can be granted to software of differing privilege levels.

Figure 3-3 illustrates a configuration where safety code runs with the processor in privileged mode and non-safety code runs with the processor in unprivileged mode. In this configuration, unprivileged code cannot spatially interfere with privileged code and data. Arranging the application in this manner is not always possible; in particular, interrupt handlers are a source of potential corruption to this scheme, as they will always operate in privileged mode.

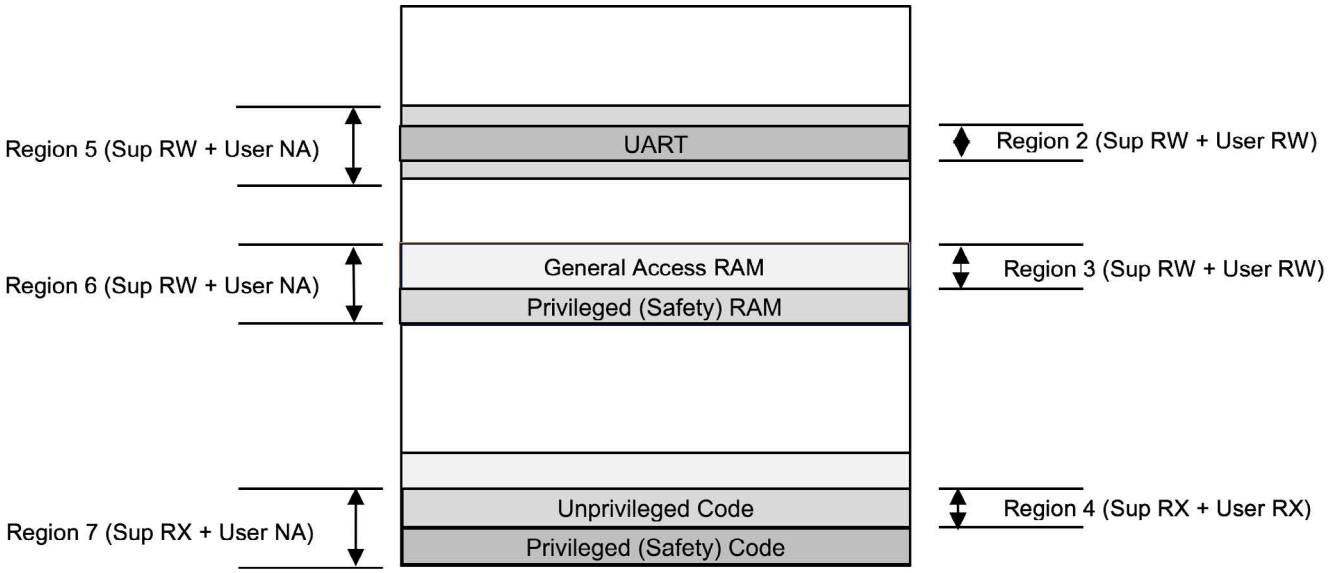

Figure 3-3 *Using Processor Privilege Modes to enforce Spatial Separation*

3.4 Using an RTOS in a Safety Application

An RTOS is different to most third party software components in that it inherently has the ability to affect the operation of the safety application. The RTOS typically manages the scheduling of any tasks or threads and is directly responsible for inter-task communication. The RTOS must therefore be certifiable to the required SIL of the application.

Unless the design and validation of the RTOS is specifically for use in a safety application, the onus for testing and proving suitability will fall on the application developer. For most applications the scope and costs involved with testing third party software for use in a safety application is prohibitive. Therefore when selecting an RTOS that will be used in a safety application, there is a clear benefit in selecting an RTOS such as SAFE**RTOS**® which has been designed to meet the needs of embedded safety applications. SAFE**RTOS** comprises a complete Design Assurance Pack (DAP) as well as the RTOS source code. The DAP can be presented to certifying bodies as evidence of the suitability of the RTOS for use within a safety application.

If the RTOS does not provide native support for the MPU, then use (or not) of an RTOS does not affect any claims to spatial separation. Without native support for the MPU, the RTOS is just another piece of privileged code and data as described in Figure 3-3 and indeed presents the usual problems with certifying third party software.

3.5 SAFE**RTOS**® a Safety Certified MPU aware RTOS

Using a 'safety certified' RTOS that also provides native MPU support, such as SAFE**RTOS**, allows the host application to use an RTOS in a safety application and still demonstrate a degree of spatial separation of the tasks within the system.

The Design Assurance Pack provided with SAFE**RTOS** contains full details of the design and testing that has been performed on the RTOS and can be used either as a model for developing to IEC61508 or related safety standards or submitted as evidence that the package has been developed in accordance with the mandated processes.

SAFE**RTOS**'s support for the MPU at kernel level ensures that the host application cannot perform unauthorized accesses of the kernel, authorized access to the kernel is only possible via the RTOS API. In addition, the design allows enforcement of spatial separation between user tasks. The design offers a compromise of usability and flexibility such that isolation and separation are achievable in a highly flexible manner.

The following sections show how a combination of fixed and reprogrammable MPU regions, together with effective use of processor modes, allows the development of systems that provide spatial separation whatever the physical hardware architecture.

3.5.1 Fixed MPU Regions

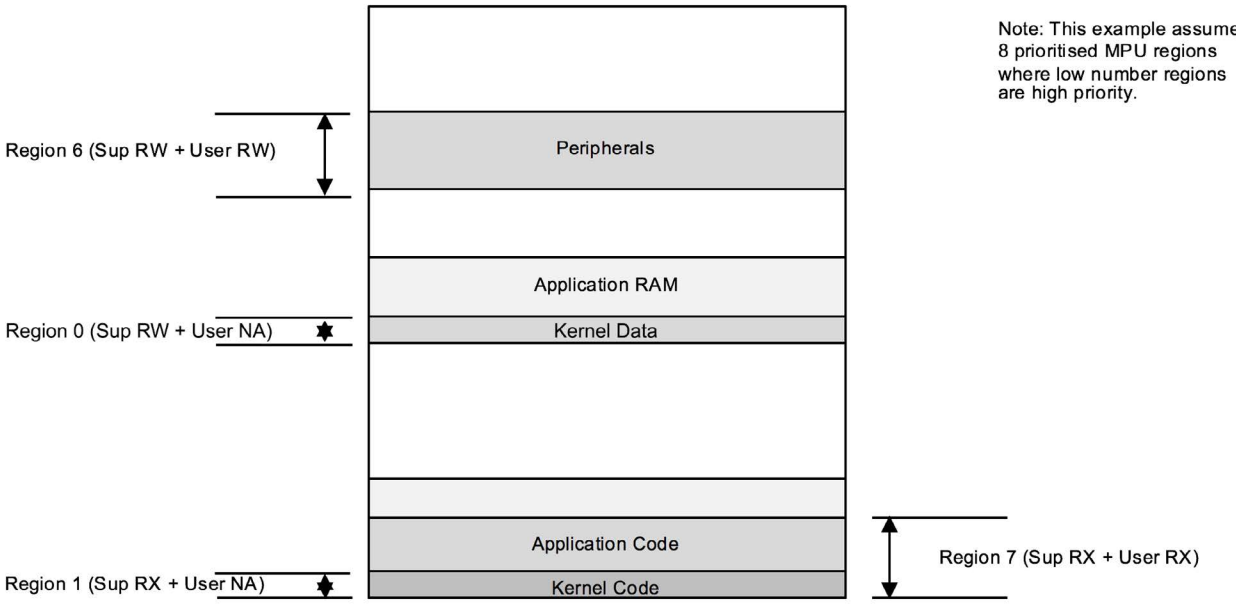

Figure 3-4 *SAFE**RTOS** Fixed MPU Regions*

- Application code can read and execute from the flash (or designated code execution region).
- User mode code is not permitted to access the kernel code except via the published API. This prevents unauthorised access to the RTOS' internal functions.
- User mode code cannot access the kernel data tables since these are critical to the correct operation of the system.
- Globally accessible memory regions allow access to memory ranges from all application code. The example below shows global access to the processor peripherals but this may not be an optimal way to demonstrate spatial separations as multiple tasks can potentially access the same peripheral.

3.5.2 Reprogrammed Regions

SAFE**RTOS** reprograms a number of regions on each context switch. This means that each task can have its own memory map of assigned memory ranges. In addition, an MPU region protects the task's stack. Any overflow or underflow condition immediately triggers an exception. Since the task's reprogrammable regions are programmed on each context switch, there is no limit to the number of tasks that can be defined.

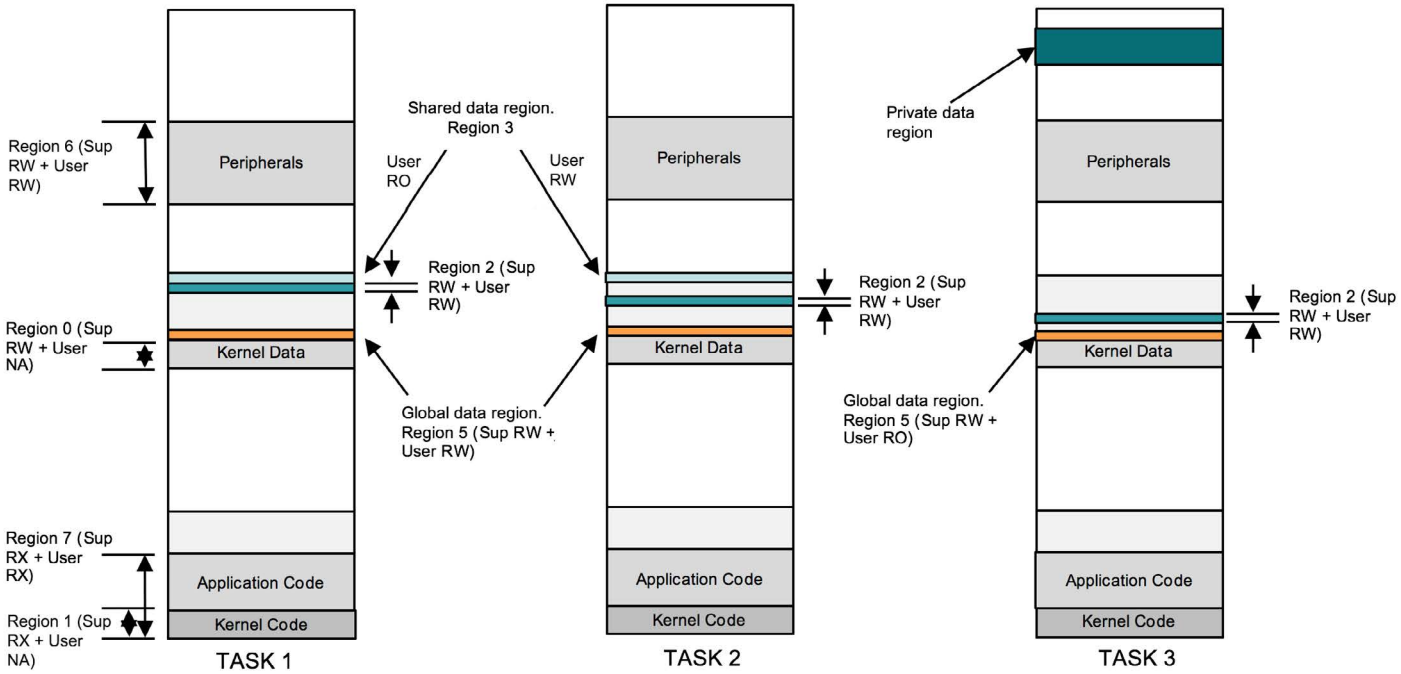

Figure 3-5 *SAFE**RTOS** Reprogrammable MPU Regions*

Figure 3-5 illustrates an example system with three tasks:

- Region 2 protects the task stack and therefore spans a different memory region depending on which task is running.
- Region 5 is a global data region. Tasks 1 and 2 have unrestricted access to this memory range but Task 3 has 'read' access only.
- Region 3 is a shared data region. Task 2 has unrestricted access, Task 1 has 'read' access and in this case, Task 3 has no access.

Using reprogrammable memory regions permits a flexible memory map where tasks that need to share data can, while tasks are denied access to memory that they do not require access to.

Contact Information

User feedback is essential to the continued maintenance and development of SAFERTOS. Please provide all software and documentation comments and suggestions via the most convenient of the contact points listed below.

Contact WITTENSTEIN high integrity systems

Address: WITTENSTEIN high integrity systems
Brown's Court, Long Ashton Business Park
Yanley Lane, Long Ashton
Bristol, BS41 9LB
England

Phone: +44 (0)1275 395 600
Email: support@HighIntegritySystems.com

Website www.HighIntegritySystems.com