HighIntegritySystems

WITTENSTEIN

Embedded Architectures Supporting Mixed Safety Integrity Software

Issue 1.2 - July 8, 2016

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

The rapid growth of safety critical software has been driven by regulation and underpinned by the existence of domain specific safety development standards. Their objective is to ensure embedded systems are designed robustly, to prevent harm or death occurring to users of the systems, 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 are used to classify the context the system is operating in, and define the amount harm the system can potentially cause. The higher the safety level, the greater the potential harm, and the more rigorous and 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 users experience, be easy to use, and communicate treatment data back to a healthcare center.

System designers are now faced with 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. Also, 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 would have to be classified with a very low safety rating. This means that within a single system there may be several different levels of safety software.

The software within the system needs to be partitioned, grouping software of the same safety level together, and assuring that software from lower safety levels can not interfere with software relating to the higher safety levels. Partitioning allows the safety related software to be kept small and concise, whilst allowing the use of third party software modules, which shortens developments times and lowers costs.

This paper discusses in detail partitioning techniques used in mixed safety level embedded systems.

CHAPTER 2 Use Case

2.1 Use Case – A Simple Embedded System

Figure 2-1 shows a simple embedded system. A number of input sensors are processed by control logic and some output is driven in response. There is a display with associated software and some memory storage with its own software. Not all of the elements shown are critical to the successful running of the system. The mission critical components are outlined in green

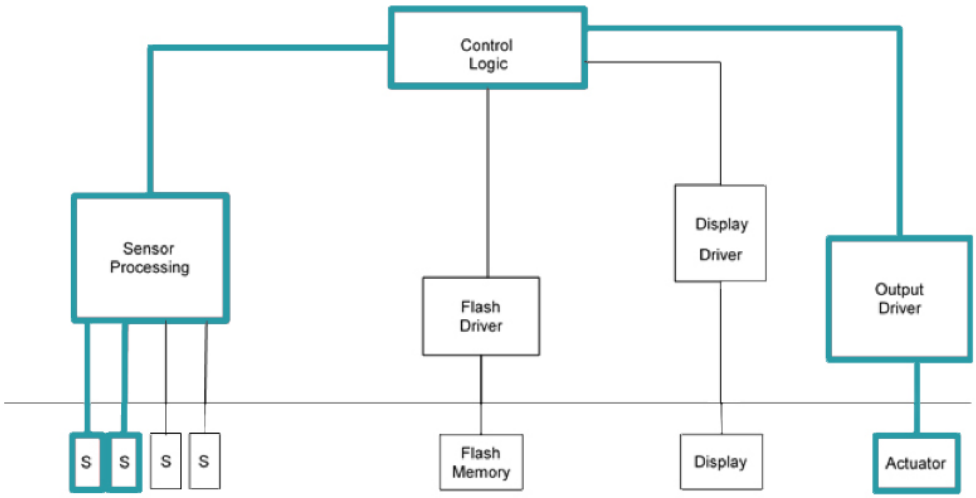

Figure 2-1 *A Simple Embedded System*

Things inevitably get more complex. The requirement to reprogram in the field, manage logging and configuration files and allow remote monitoring give us a considerably more complex block diagram, see Figure 2-2.

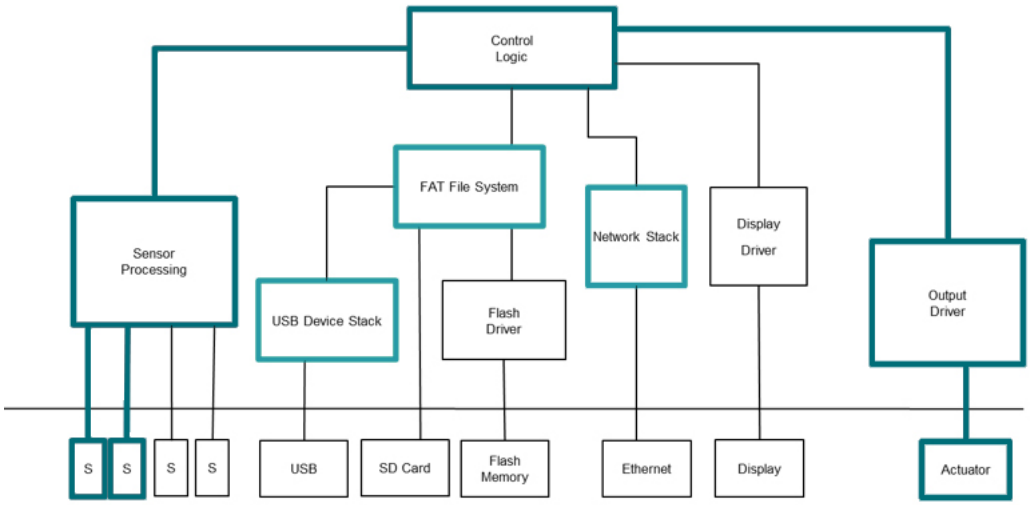

Figure 2-2 *A more complex system*

We now have a system with:

- Critical Safety Software
- Commercial third party software that we have no control over.
- Other software not developed to the required Safety Integrity Level (SIL).
- Grey areas where potentially critical data is passing through 3rd party stacks.

To successfully achieve a SIL rating for our product we need to be able to demonstrate both temporal and spatial separation between the code responsible for the safety critical parts of the application and the code that has not been developed with the required rigor and the software of unknown provenance.

One approach to achieving this separation is to spread the software across multiple processors so that one (or more) processors are responsible for implementing the safety application and the non-safety functions are located other processors. This architecture is discussed in section CHAPTER 3.

Another approach is to use software techniques to attempt to guarantee the necessary separation. This involves:

- Breaking the application into a number of threads or tasks and using an Memory Protection Unit (MPU) or Memory Management Unit (MMU) aware Real Time Operating System (RTOS) to enforce spatial separation.
- Using software techniques to monitor the timing profiles of the tasks and report deviations to the temporal separation.
- Using communication protocols to protect the data communications with other tasks and system elements rather than relying on the integrity of the data channel and the software servicing that channel.

Each of these techniques will be the subject of a separate white paper, but are introduced in CHAPTER 4.

CHAPTER 3 Achieving Functional Separation with Hardware

3.1 System Architecture

The example system architecture, shown in Figure 3 1 achieves the objectives set out above for the device (in this case a medical device). It is constructed around three processing elements. Two processors are allocated to implementing safety critical functionality, and the third processor is reserved for non-safety related operations including the users interface, networking and other aspects of the application.

Generally, it is desirable to limit the scope of the safety critical functionality as much as possible. The simpler the overall design is, the easier and cheaper the design and testing and hopefully results in reduced development timescales, and an easier path to achieving approval/certification.

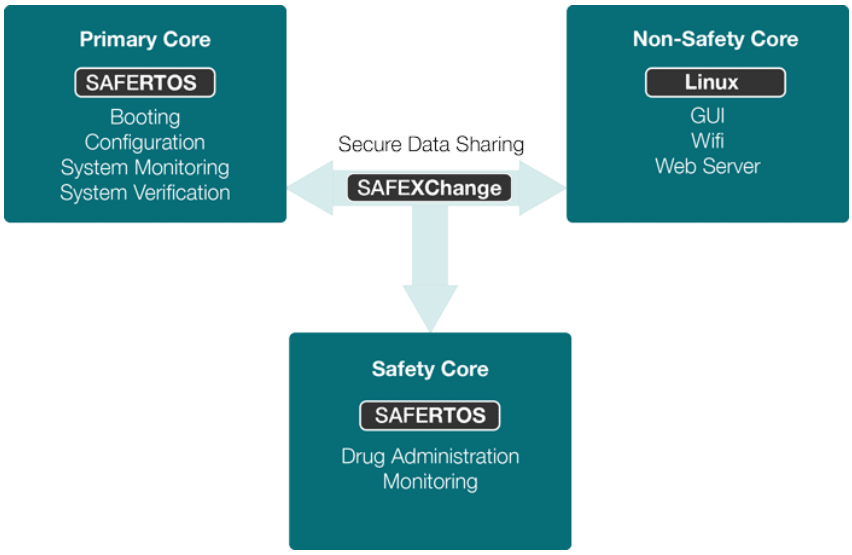

Figure 3 1 *Multi Processor Architecture*

By allocating two microprocessors to perform the safety function, it allows one microprocessor to implement the safety function and the other to verify the operation. The resulting safety case is simple, if the two microprocessors agree the treatment continues, if there is a difference then the treatment is stopped and the system is placed in a safe state. This architecture provides protection from single hardware failures, and inherently provides isolation from interference from actions taken by the non-safety processor.

One of the microprocessors would also be allocated the 'power on' task, ensuring the system is placed within a safe state upon leaving its initialisation state, and configuring the system architecture so that the non-safety processor cannot interfere with the safety microprocessors. This microprocessor would also be allocated the task of performing any power on self-tests and periodic built in tests that effect the safety of the system.

3.2 Software Architecture

As the architecture has two different roles to perform, supporting safety and non-safety functionality, two different types of software Board Support Packages (BSPs) are required, coupled with a means of transferring data seamlessly between them.

The safety related microprocessors will typically require a real time operating system suitable for use within a safety device. The RTOS is at the heart of the system and controls the scheduling of the safety software. Therefore the RTOS needs to be rated at the highest SIL of the software on the processor, and cannot be treated as a 'Commercial off the Shelf' (COTS) or Software of Unknown Provenance (SOUP) component.

One such example of a SIL rated RTOS is SAFE**RTOS**® from WITTENSTEIN high integrity systems. With an imperceptible boot time, SAFE**RTOS** can quickly and effectively bring the system up, configure the safety partitions and execute critical safety functionality/testing, before enabling other processors which may require longer to boot. Due to its high safety classification, SAFE**RTOS** can safely be used on both primary and monitoring microprocessors, removing the need to use differential software.

In this example, there is a separate application processor, which handles the non-safety critical aspects of the system. This may include networking, Wi-Fi stacks, and user interfaces. In this example the application processor is shown as using a BSP built around the Linux operating system. Linux is supported by a wide range of middleware components and drivers and provides the ideal platform for building a complex application upon. An alternative to Linux would be to use an RTOS such as SAFE**RTOS** and a selection of middleware components.

It should be noted that careful attention should be paid to the classification of software/system functions. In this example we are assuming the display and networking functionality are non-safety critical components, this is frequently not the case. When dealing with medical devices the information displayed may affect clinical decisions or the input data affects dose rates, this functionality then becomes a safety component.

3.3 Data Sharing

When sharing Data within a safety critical system, the SIL value of the individual data items needs to be preserved during transmission. The distribution of safety critical data becomes more complex when the transmission mechanism is shared with data generated from unknown or non-safety rated sources.

Generally communication channels can be considered as white channels with known properties or black channel with unknown properties. As designers of the system, initially it would seem more logical to consider the communication link as a white channel, but this requires implementing in-depth bit error rate probability calculations, and an analysis to determine the effects the unknown data sources may have on the communication channel. In many ways it's easier to treat the channel as a black channel, and protect the data sent over it. Here the system will need to guard against the 8 known failure modes of a black channel, as detailed within EN/IEC61784-3 Functional safety fieldbuses – general rules and profile definitions. These failure modes include, Data Corruption, Unwanted Repetition, Wrong Sequencing, Loss, Inacceptable Delay, Insertion, Masquerade and incorrect addressing (see section 4.5).

3.4 Safety Considerations

The current trend is to build safety systems from pre-certified modules. This reduces overall development times and lowers the risk of certification issues. It also allows companies to benefit from the know-how and expertise of suppliers who have their components in safety critical devices already.

Processors can be selected according to their ability to be safety certified. Many silicon vendors are now offering safety design packages for their MCUs. These packages typically contain a Safety Manual that details the list of safety requirements (conditions of use) and provides examples to guide users on how to achieve the required SIL according to IEC 61508.

SAFERTOS from WITTENSEIN high integrity systems is supplied fully integrated with the selected processor/compiler combination, and accompanied by a comprehensive Design Assurance Pack (DAP) or Design History File (DHF). These packages contain the design and testing artefacts as well as instructions on how to install and integrate SAFERTOS into a safety critical development without the need for further retesting upon the target hardware. The safety manual also details how to generate the evidence required to demonstrate the integration and installation process has been correctly followed.

3.5 Conclusion

Referring back to the original use case and the requirements for spatial and temporal separation, this has been achieved by physically separating the safety and non-safety functionality onto different processors and using a third processor to monitor the correctness of the safety processor. A strong case for certification can be made if:

- The software on the safety and monitoring processors has been developed in accordance with the required SIL rating for the industry sector.
- Any third party software (such as RTOS or communications stacks) has also been developed and tested to the required SIL level.
- The selection of the hardware is suitable for use in a safety application at the required SIL, this potentially includes software to monitor the correct operation of the hardware, redundancy of sensors/actuators, error checking of RAM and ROM.
- Data exchange between the safety and monitoring processors is implemented using hardware or protocols to guarantee safety to the required SIL.

By using the separation and isolation features of SAFERTOS, which allows software of different SILs to safety share the same linear memory range, it may be possible to combine the functionality of two of the processors. For example, the functionality of the safety microprocessor monitoring and verifying the safety function could be integrated upon the Application Processor. Likewise the functionality of the Application Processor could be integrated upon the safety microprocessor implementing the monitoring of the safety function. This is further discussed in the next section.

CHAPTER 4 Achieving Functional Separation with a Single Processor

4.1 System Architecture

The architecture discussed in the previous section potentially provides the necessary separation and is suitable for larger devices. When dealing with smaller devices or battery operated devices a smaller hardware footprint is frequently necessary in terms of both cost and power consumption. In these cases, it is desirable to use a single processor for all the functionality with an external watchdog or small monitoring micro to provide the necessary redundancy for failure detection of a safety device. This is shown in Figure 4-1. The acceptability of a single core architecture for a mixed SIL application should be reviewed on a case-by-case basis and the design and safety mitigations need to be discussed with the relevant certification body at an early stage.

Figure 4-1 *Single Processor Architecture*

4.2 Software Architecture

Conventionally, all software running on a microprocessor must be written to the Safety Integrity Level (SIL) that is required for the system. This means that the use of third party COTS or SOUP code is problematic as it is most unlikely to be tested or documented to the required standards.

When using third party software that is pre-certified as suitable for inclusion in a SIL rated safety system, advantages can be gained in that the effort necessary to design, test and certify the software has been performed externally by people who are domain experts in that functionality. SAFERTOS from WITTENSTEIN high integrity systems provides a fully pre-certified scheduling kernel which is designed for use with safety applications up to SIL3.

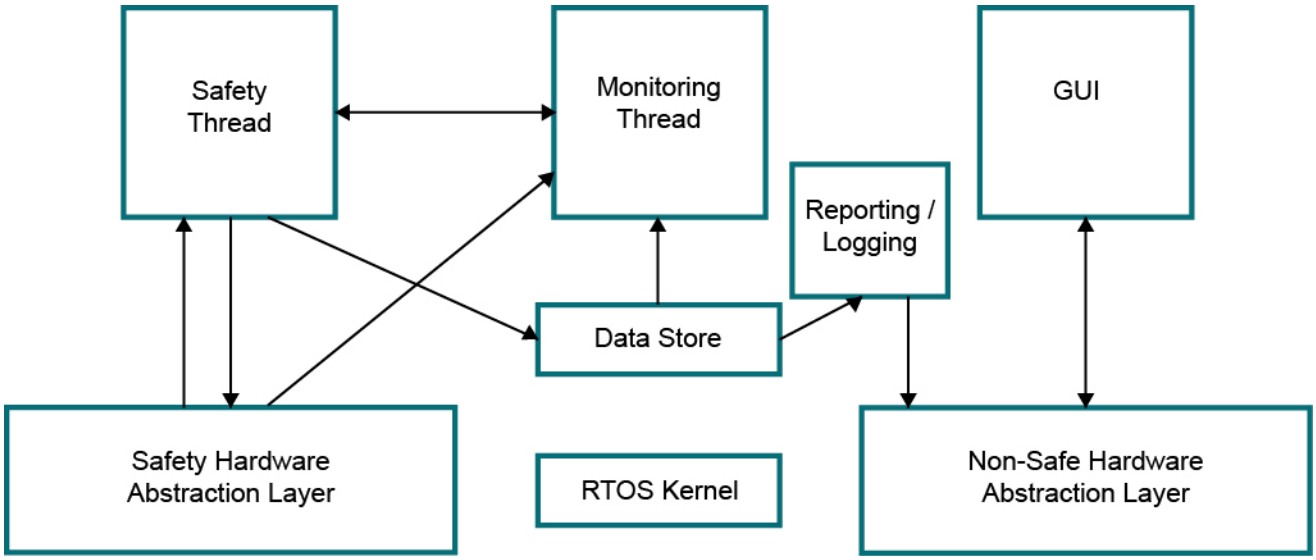

Figure 4-2 *Mixed SIL Software Architecture*

4.3 Spatial Separation using MPU or MMU

An MPU or MMU is used to detect access to unauthorised regions within the memory map. The rest of this section refers to the use of the MPU; however it is equally applicable to the use of an MMU with a flat memory map.

Microprocessors that have an MPU typically allow a number of “memory regions” to be defined. This consists of a memory range and associated access permissions. There are some differences in the operation depending on the silicon manufacturer (e.g. ARM processors feature prioritised MPU regions whereas others are additive with respect to granting of permissions); however whatever the flavour, the action is the same, a processor exception will be generated if an illegal access is detected.

When not using an RTOS, the uses of the MPU are somewhat limited unless the application performs significant reconfiguration of the MPU during run time. Figure 4-3 shows a memory map where the MPU is used to restrict access to the used FLASH, used RAM and certain peripherals. This is a useful diagnostic but not enough to prove any spatial separation of processes within the system.

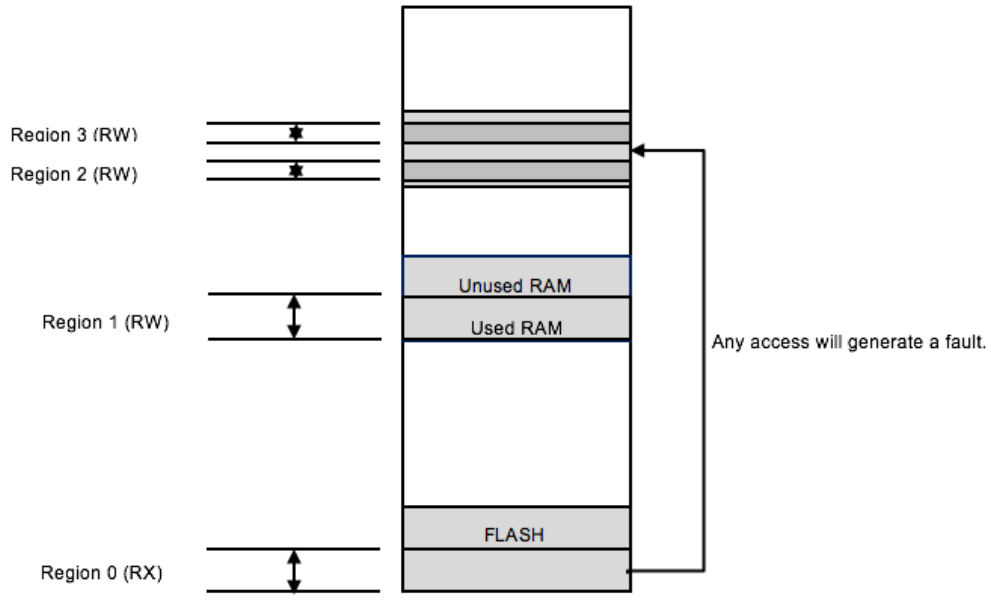

Figure 4-3 *Basic MPU configuration*

Using processor privilege modes (e.g. System or User) can provide more options to define more precise memory regions but this is still generally insufficient to fully segregate safety and non-safety critical code.

When using an RTOS, the application is broken up into tasks or threads. Communication between tasks is frequently accomplished with Queues or Events that are managed by the RTOS. This gives much more opportunity for enforcing partitioning at a fundamental level if the RTOS provides native support for the MPU. If the RTOS in use does not provide native support for the MPU, then we have a similar scenario to the previous arrangement except that we have a new problem, the RTOS code and data. The RTOS is managing the task switching and is responsible for scheduling, therefore from a Function Safety viewpoint this must be at the highest SIL level of the application and we cannot use protection techniques to partition COTS or SOUP since the operation of this implicitly affects the operation of the application. This means that we must either use a certified RTOS that natively supports the MPU or test it ourselves. Many commercial RTOS's have a long history and there is some 'proof in use' claims but this is notoriously hard to quantify in a functional safety environment.

4.3.1 SAFERTOS a Certified RTOS with Integrated MPU Support

When using an RTOS with integrated MPU support, we can implement a number of safety measures. Firstly, it is possible to protect the kernel code and data from unauthorised access by the application. Obviously the correct operation of the kernel is necessary to ensure that the safety critical software is scheduled correctly. Figure 4-4 shows the memory map of a system where:

- All flash code has read and execute permissions.
- Kernel code has supervisor read and execute permissions only.
- Kernel data has supervisor read and write permissions only.
- General access is permitted to the processor peripherals.
- No other access is granted to the RAM (this means that any necessary access must be explicitly granted in an individual task basis

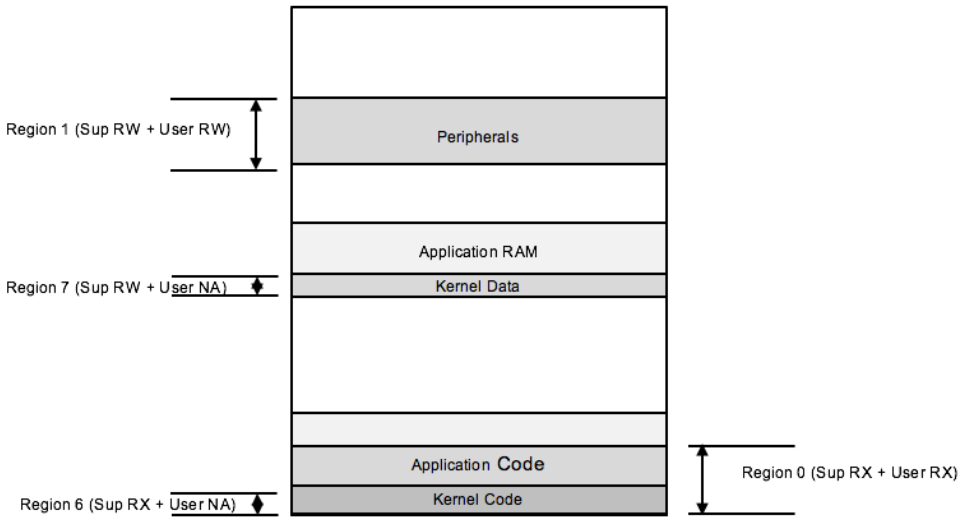

Figure 4-4 *Protecting the Scheduler Kernel with MPU regions*

Secondly, integrated MPU Support allows us to provide a degree of task isolation by protecting user task stacks. Figure 4-5 shows a system where each task has an MPU region that covers the stack allocated to the task. This region is reprogrammed on each context switch so that the active task cannot corrupt other task stacks or memory buffers.

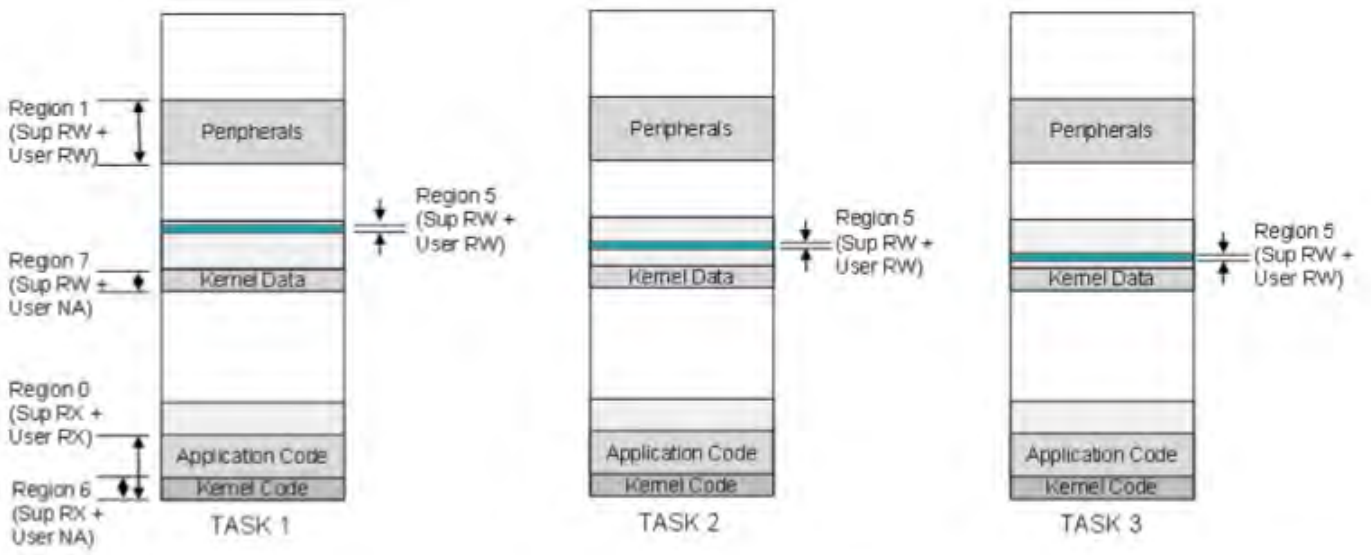

Figure 4-5 *Protecting Task Stacks with MPU regions*

Finally, integrated MPU Support allows us to reprogram some of the MPU during each context switch and therefore allow each task to have its own set of configurable regions. Figure 4-6 shows a system where:

- There is a global data region that tasks 1 and 2 have full access to but task 3 has only read access.
- There is a shared region between tasks 1 and 2.
- Task 3 has a private data region.

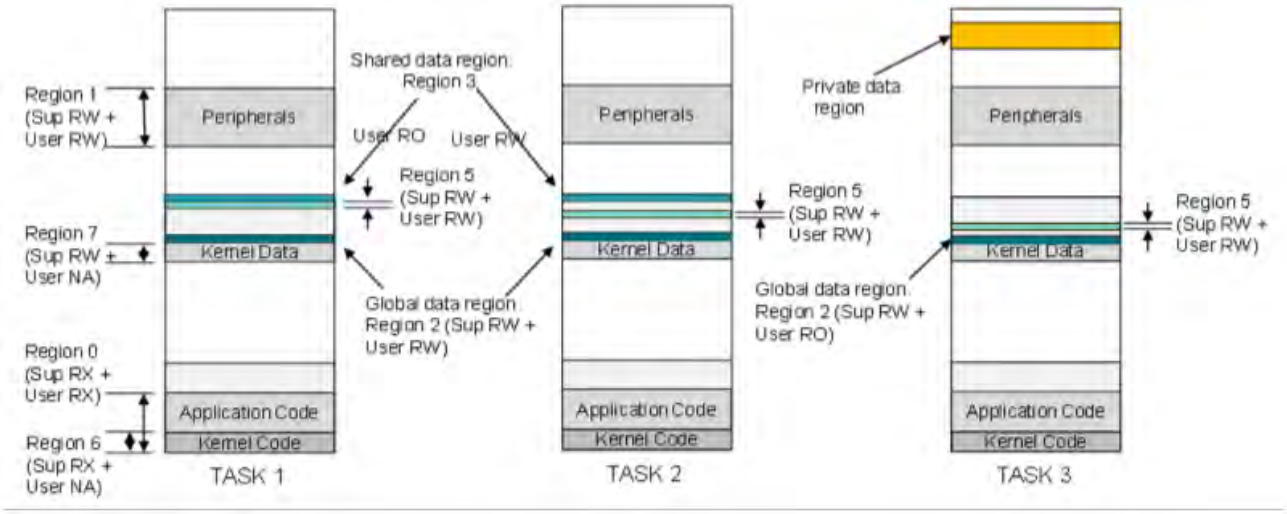

Figure 4-6 *Configurable MPU regions for each Task*

4.4 Temporal Separation using Checkpoints

Temporal separation is much harder in an embedded real time system, by definition we are responding to events and timely response to these events is crucial. Checkpoints is a feature that can be used to detect scheduling issues and help to prove temporal separation. Note that this does not enforce temporal separation, it merely offers a means to detect when temporal separation has been breached.

4.4.1 Temporal Scheduling Problems

Figure 4-7 shows a system that has two interrupts that each trigger a task to process the event. In addition, there is a high priority periodic task and a low priority periodic task.

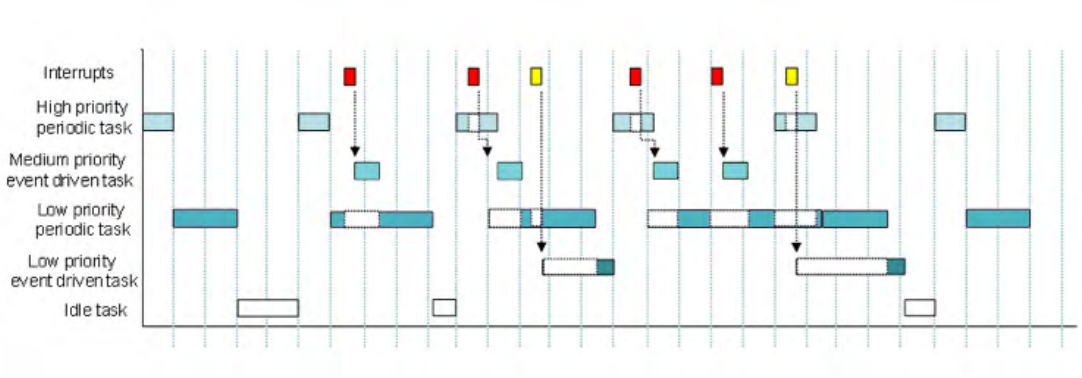

Figure 4-7 *Temporal Scheduling Problems*

In this example there is severe instability in the time taken to complete the low priority periodic processing and in one instance the processing has not completed when the pass is due to commence. In addition, there are severe delays in responding to the event that triggers the low priority event based task. Depending on the system in question this may be acceptable and perhaps some improvements can be made by altering task priorities; however it may be the case that a safety critical task has a define time profile that must be adhered to to maintain the correct operation of the system.

4.4.2 Using Software Timers to Monitor Task Execution

Checkpoints is a very simple concept where two software timers are used to monitor the execution of a task. Figure 4-8 shows a task that has two software timers associated with it. There is a checkpoint monitoring point within the task, if T1 has not expired at that point then there is a task underrun situation. The checkpoint monitoring point also resets both T1 and T2; therefore if at any point T2 expires then there is a task overrun condition. The precise action to take in the case of a scheduling breach is application specific and an error callback hook can be triggered when an error is detected.

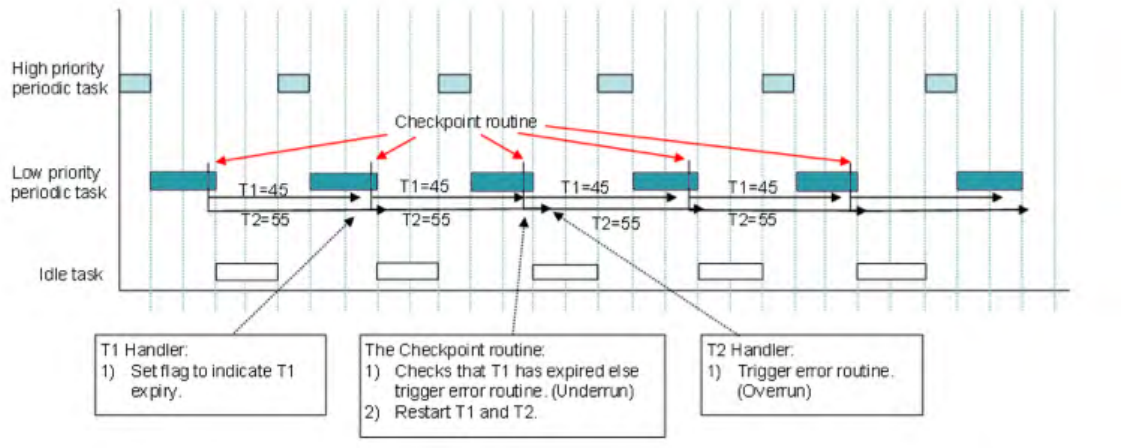

Figure 4 8 *Checkpoint Timers*

Variations on this approach can be used to monitor event driven tasks, or whole safety functions that may involve the interaction of multiple tasks and events.

4.5 Data Protection with SAFEXchange

The SAFEXchange Safety Component preserves the SIL of data shared across black channel communication buses within a multi-processor, multi-core environment. As such it forms a safety communication layer between the application and communication software that adds integrity information to data packets before transmission. Upon reception, the additional integrity information is used to verify the correctness of the data packet before making it available to the application layer.

SAFEXchange uses the principles defined in the Functional Safety Fieldbus Standard, IEC 61784-3 to determine the correctness of the data. This standard defines the eight error types that occur in a 'Black Channel' communication system and outlines the safety measures required to protect against them. The potential failure modes (Data Corruption, Unwanted Repetition, Wrong Sequencing, Loss, Inacceptable Delay, Insertion, Masquerade and incorrect addressing) are shown in Figure 4-9 together with the mitigation strategy included within the SAFEXchange protocol layer.

	Mitigations			
Failures	**Identifier**	**Sequence Counter**	**Timestamp**	**Checksum**
Incorrect Addressing	X			X
Corruption				X
Delay			X	
Repetition		X		
Incorrect Sequence		X	X	
Loss		X		
Insertion		X	X	
Masquerade	X	X	X	X

Figure 4-9 *Communication Failure Modes and Mitigations*

Contact Information

User feedback is essential to the continued maintenance and development of SAFERTOS. Please provide all software and documentation comments and suggestions to the most convenient contact point 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
Fax: +44 (0)1275 393 630
Email: support@HighIntegritySystems.com

Website www.HighIntegritySystems.com