HighIntegritySystems

WITTENSTEIN

Checkpoints and Temporal Separation

Issue 1.1 - April 24, 2017

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 had 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, 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. The safety levels 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 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 would 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 developments times and lowering costs.

This paper discusses techniques to achieve temporal separation or time based partitioning within mixed safety level embedded systems. Temporal separation is concerned with ensuring that it is not possible for the other system software to compromise the processing demands of the safety critical software. While direct access to available processing resources is a primary part of ensuring temporal separation, system events and triggers may also require analysis in this regard.

1.2 Use Case – An 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 that 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.
- 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.

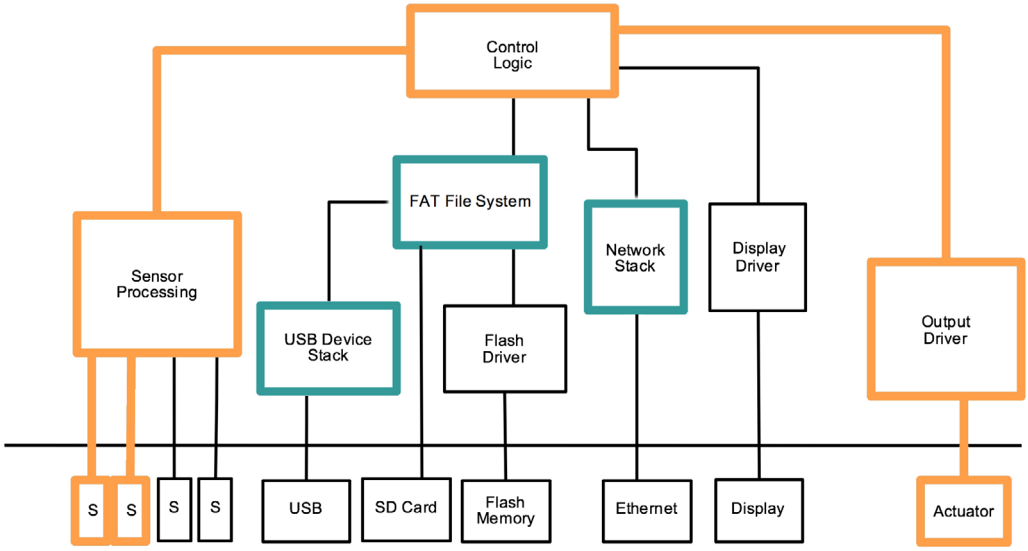

Figure 1-1 *A Typical Embedded System*

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 temporal separation only; spatial separation and data integrity are the subject of other white papers, available from www.highintegritysystems.com/white-papers/

CHAPTER 2 System Architecture and its Effect on Temporal Separation

2.1 Temporal Separation with a Multi-Processor System

There 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.

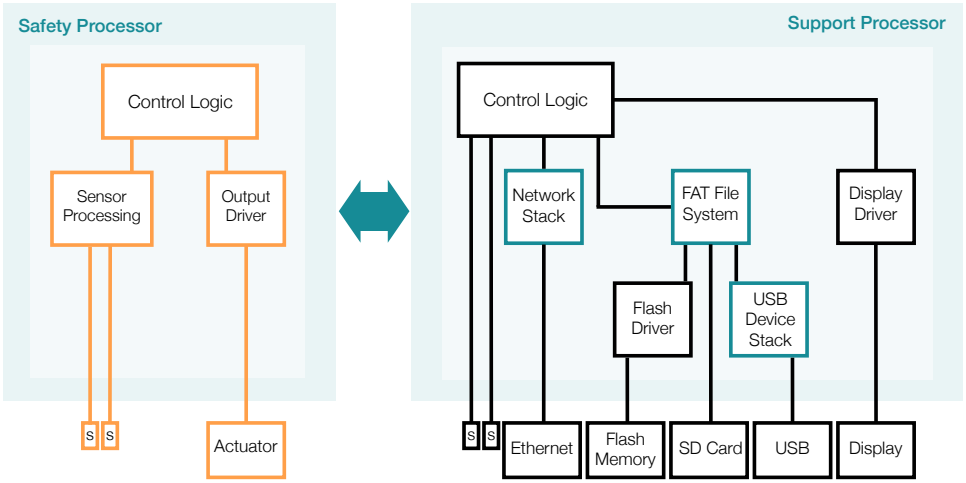

Figure 2-1 *A Multi-Processor System*

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

2.2 Temporal Separation with a Multi-Core System

Multi-core processors are becoming increasing prevalent 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.

From the perspective of demonstrating and proving temporal separation, this is identical to the multi-processor system, as there is clearly a physical separation between the CPU's.

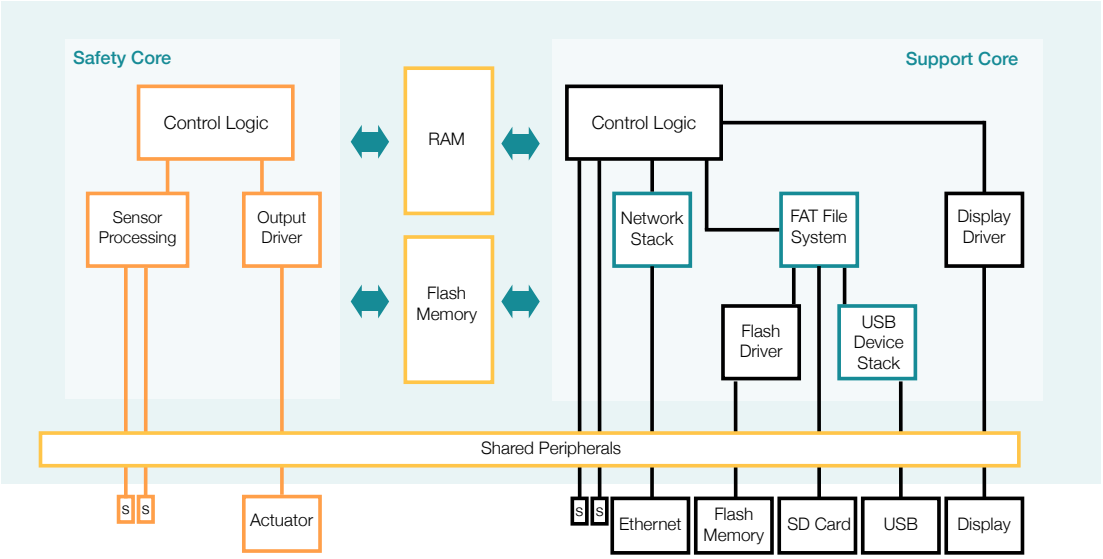

Figure 2-2 *A Multi-Core System*

2.3 Temporal 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 temporal separation between software modules. This means that any attempts to achieve temporal separation require a software solution.

A 'time separation' kernel can achieve temporal separation provided the kernel itself meets the requirements of a safety critical application and the system can accommodate the relatively inefficient processing model.

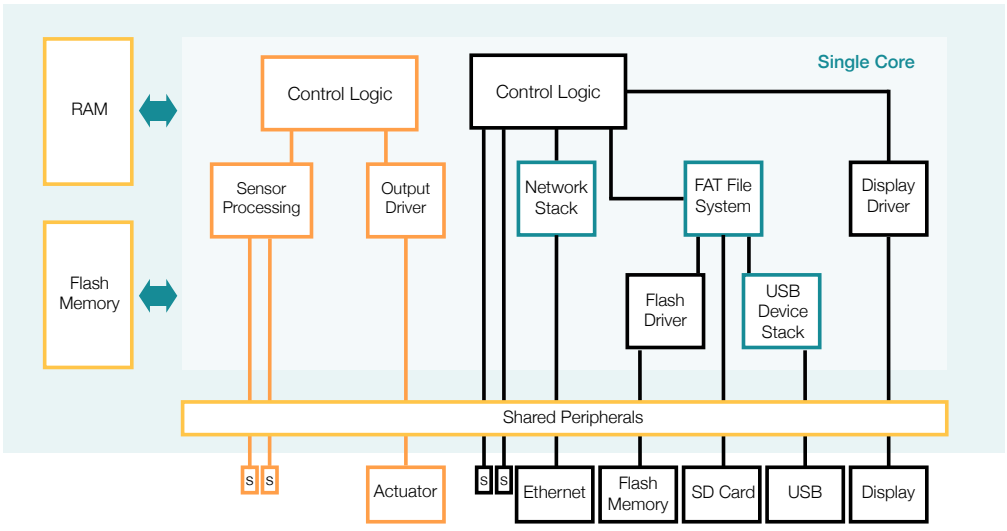

Figure 2-3 *A Single Core System*

Another option is to use a software component to detect infringements of the systems temporal requirements. Note that this does not enforce temporal separation but will allow for detection and implementation of recovery actions.

CHAPTER 3 Using Software Timers to Monitor Temporal Separation

3.1 Temporal Separation Problems

3.1.1 Temporal Scheduling Problems with Priority Based Pre-emptive Scheduling

In an embedded real time system, true temporal separation is hard to achieve as we are by definition responding to events and timely response to these events is crucial. Figure 3-1 shows a system that has two periodic tasks and an idle task. System operation is balanced and each task has sufficient run time even in worst-case execution conditions.

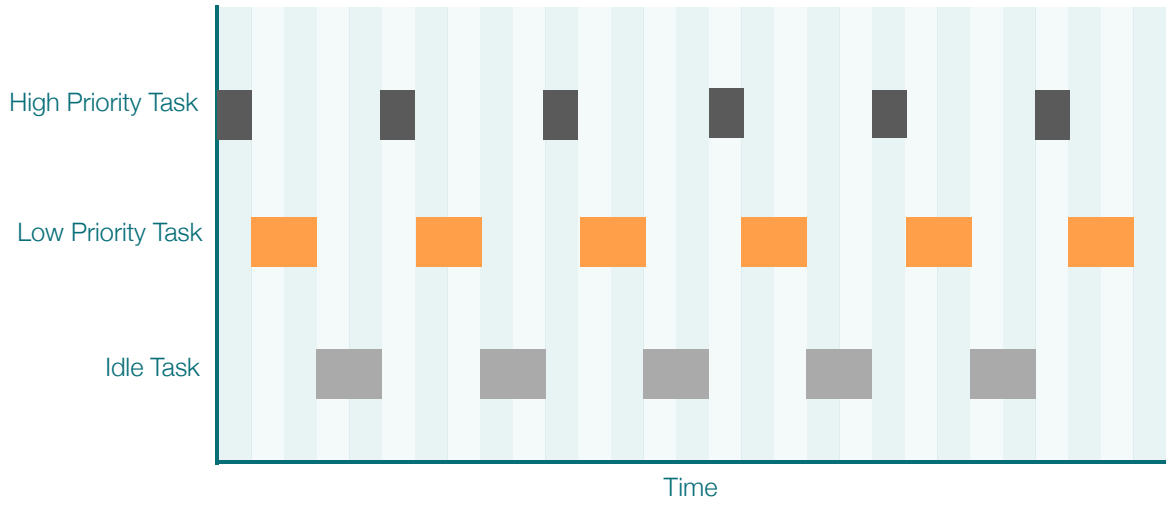

Figure 3-1 *A Balanced Prioritised System*

Figure 3-2 shows a system where the previously balanced system now has an interrupt that triggers a medium priority task. It is easy to see that there is serious jitter in both the start and duration of the low priority periodic task.

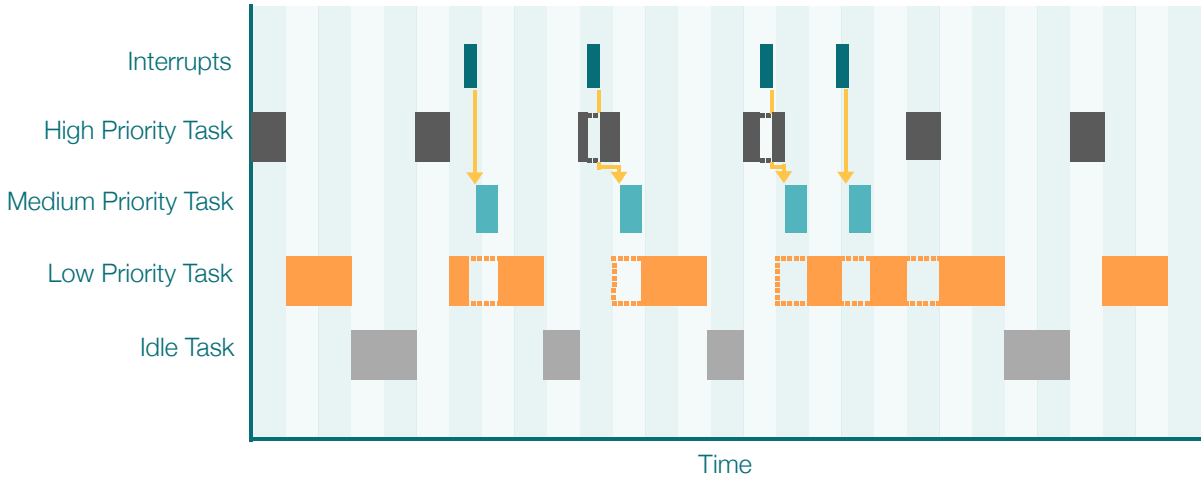

Figure 3-2 *Effects of Asynchronous Events on a Balanced System*

Finally, Figure 3-3 adds another interrupt and a low priority event driven task.

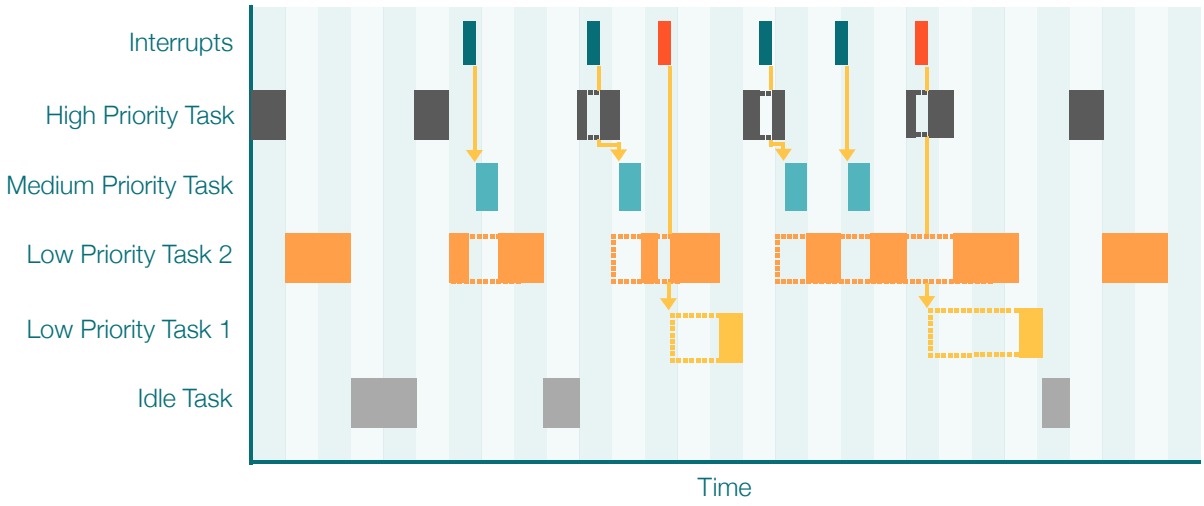

Figure 3-3 *A Severely Unbalanced System*

In this example, there is severe instability in the time taken to complete the low priority periodic processing. In one instance, the low priority task 2 periodic processing has not completed when the next pass is due to commence. In addition, there are severe delays in responding to the event that triggers the low priority event based task 1. Depending on the system in question, this may be acceptable and perhaps altering task priorities will yield some improvements; however, it may also be the case that a safety critical task has a defined time profile in order to maintain the correct operation of the system.

3.1.2 Avoiding Temporal Scheduling Problems

There are many established ways of handling the effect of scheduling overrun and thereby avoiding the extreme scenario presented in Figure 3-3. These include:

- Minimising the processing that occurs within interrupt handlers.
- Analysing the worst-case interrupt processing within periodic frames and if necessary taking steps to limit the maximum interrupt processing.
- Analysing the worst case processing within tasks and using this to determine average and peak CPU load.
- Wherever possible, making effective use of priority based scheduling so that essential tasks are performed in as deterministic manner as possible.

Even when all potential scheduling problems have been addressed and the system is working correctly, we still have not managed to prove temporal separation. For some systems, this may not be a problem; however, when dealing with a mixed SIL system then it is necessary to be able to prove that software operating at a lower SIL cannot interfere with the operation of software designed to a higher SIL.

3.2 Monitoring Temporal Separation Using Software Timers

Execution frequency (and jitter) of task execution can be monitored using software timers. Most commercial RTOSs offer this feature and it is relatively simple to detect scheduling issues and help to prove temporal separation. Note that the technique described here does not enforce temporal separation; it merely offers a means to detect when breaching of temporal requirements occurs.

3.2.1 Using a Software Timer to Monitor Task Execution

Figure 3-4 shows a simple system with two periodic tasks running. At a given point in a tasks' operation, the checkpoint occurs where the temporal performance is measured. The elapsed time is calculated and used to determine whether a scheduler underrun has occurred. The software timer used to detect scheduling overrun is reset (restarted). If at any time, the software timer expires then the task is in breach of its scheduling policy and the error reported. Depending on the nature of the time profiling, it may be more appropriate to implement the time checking at the beginning or end of the tasks run.

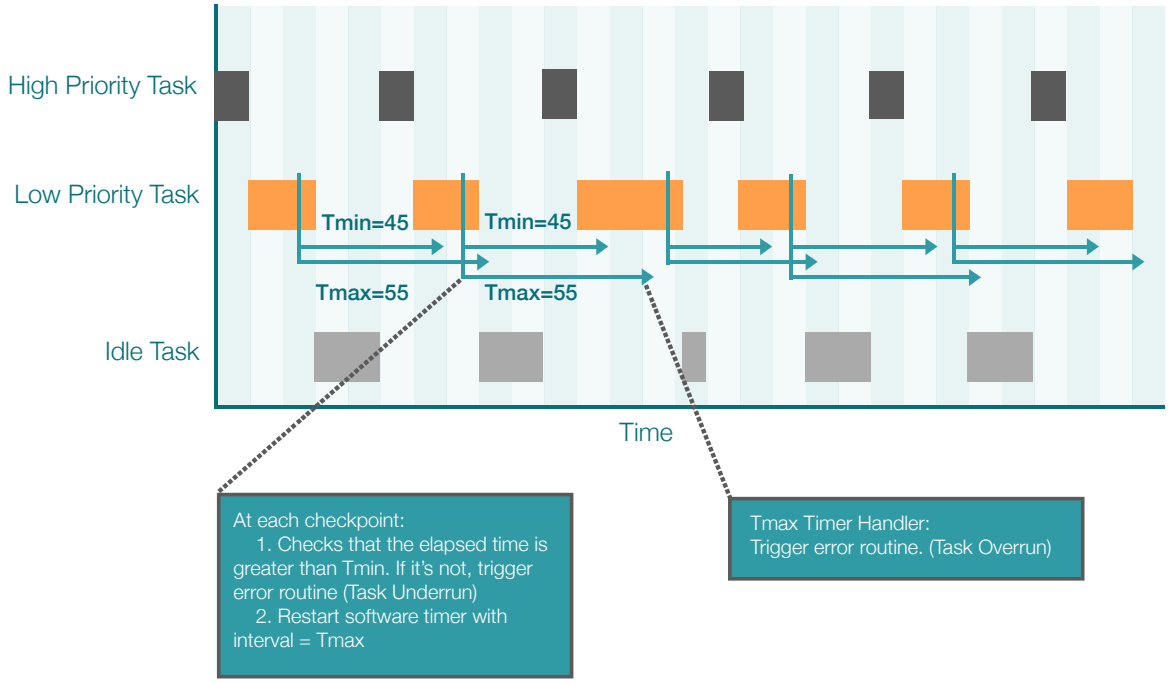

Figure 3-4 *Using a Software timer to Monitor Task Execution*

3.2.2 Using Software Timers to Monitor Interrupt Responses or Whole Safety Functions

These techniques can also be used to measure whole safety functions, where the final output may be the result of a number of events occurring and a number of tasks interacting to produce the final output.

Deferred interrupt processing is a term used to describe a scheme where events are recognised using interrupts and then passed into application tasks to perform the required processing. This has the advantage on minimising the time that the system is unresponsive to other events but can lead to indeterminate delays fully processing and responding to the events. Figure 3-3 illustrates this problem where the response to the low priority event is severely unstable. Figure 3-5 shows a scheme where a checkpoint timer is started within an ISR when the event is first recognised, in the first case the processing completes before the timer expires and all is well; however in the second case the timer expires and an error is reported, as the response to the event is too slow.

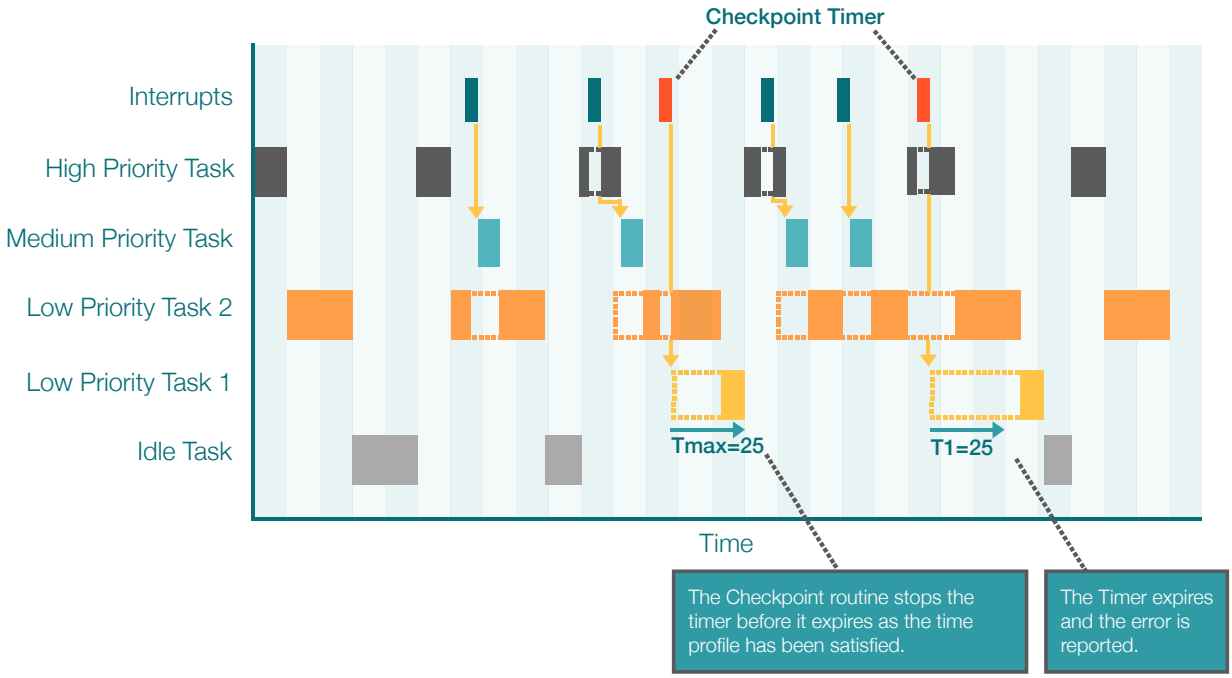

Figure 3-5 *Using a Software timer to Monitor Task Execution in an Unbalanced System*

3.2.3 Disadvantages of Using Software Timers for Task Monitoring

Using timers to implement monitoring checkpoints is a very simple concept and potentially very useful; however, there are a number of drawbacks.

- The monitoring functionality, which potentially is an important part of the safety case, is part of the general software timer mechanism. This means that any corruption or unanticipated behaviour with any software timer can affect the integrity of the entire monitoring mechanism.
- The priority of the timer mechanism is the priority of the monitoring mechanism. Therefore, potentially a fault condition can occur but go unreported as the CPU is performing higher priority processing.
- Using unique timer callbacks to identify the overrunning tasks requires much specialised code; however, a common timer callback handler may not provide the ability to differentiate the offending task.

The first two issues raised can be resolved by using an external windowed watchdog, which is refreshed providing no timing errors have been detected.

3.3 SAFE**RTOS**® and SAFE**Checkpoints**

Using a 'safety certified' RTOS such as SAFE**RTOS**® provides peace of mind and documentary evidence that the RTOS has been developed in accordance with the procedures necessary for inclusion within a safety certified product. Furthermore, SAFE**RTOS** also provides native support for a Memory Protection Unit (MPU), which allows the claiming of some degree of spatial separation. To supplement this SAFE**RTOS** also provides a dedicated checkpoints module, SAFE**Checkpoints**, which broadly operates in the manner described in this paper.

The SAFE**RTOS** SAFE**Checkpoints** mechanism operates within a dedicated kernel task (timer instance) that runs at the highest priority available within the system. This ensures that any misbehaviour in the regular timer callback handlers cannot affect the operation of the checkpoint monitoring and that other task processing cannot pre-empt or prevent the checkpoint task from running.

A dedicated checkpoint mechanism also offers extra features within the API and therefore the safety certified kernel code:

- Limited checkpoint API minimizes problems within the safety monitoring system due to operator misuse;
- The ability to select single shot or periodic checkpoints;
- Periodic checkpoints can run their timing from absolute or relative checkpoints (scheduling drift may or may not be allowable).
- Notification of checkpoint failures are by callback function; the system error handler or individual checkpoint callback routines can be specified.
- The checkpoints mechanism includes methods to identify individual callbacks even when the same handler manages multiple checkpoints.

Whatever the system architecture, SAFE**Checkpoints** module offers the host application developer the necessary tools to provide real time monitoring of the tasks within a safety system.

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