Memory Protection

In a real-time operating system (RTOS), memory protection prevents one Task from overwriting the memory space of another, minimizing the possibility of corrupted memory causing failure of the application.

INTRODUCTION

In a real-time operating system (RTOS), memory protection is an aspect of memory management, and is especially important in safety-critical applications. In a microprocessor application, Tasks that are essential to safety-critical or high-integrity functions must not overwrite each other’s memory space, and also need to be protected from those parts of the system that are not safety-critical and may have lower integrity.

MEMORY PROTECTION

Many available microprocessors for embedded devices include an optional Memory Protection Unit (MPU) or, in some cases, a Memory Management Unit (MMU) to assist in the management of separate memory spaces. This technical note describes the use of an RTOS in implementing memory protection with a hardware MPU (or MMU).

USING THE MPU

The use of separate regions enables kernel code and data to be protected from unauthorized access by the application (this also requires correct operation of the kernel). Tasks can have their own separate, protected memory areas and these can be set at an access level appropriate to the Task (User or Privileged).

SPATIAL SEPARATION USING THE MPU OR MMU

An MPU is used to separate different areas of memory so that data from one area cannot overwrite or corrupt that in another area. The MPU can detect access to unauthorized regions within the memory map. An MMU serves the same function in protecting memory, but also includes more advanced memory control features that are required in complex systems.

MPUs allow a number of regions to be defined in memory. Each region consists of a memory range and associated access permissions. The way in which the MPU operates will vary depending on the manufacturer, but all generate a processor exception if an illegal access (reading, writing or executing the contents of a memory region without the necessary privilege) is detected. This can be used to protect access to both memory and peripherals.

The use of separate regions enables kernel code and data to be protected from unauthorized access by the application (this also requires correct operation of the kernel). Tasks can have their own separate, protected memory areas and these can be set at an access level appropriate to the Task (User or Privileged).

Some of the memory regions can be reprogrammed at each context switch, enabling Tasks to have regions that can be reconfigured according to the needs of the processing.

Figure 1. Memory Protection Regions
MEMORY ACCESS EXCEPTIONS

The application engineer is responsible for handling exceptions appropriately, although the RTOS may enforce particular conditions. In general, if an illegal access happens it is best to put the system into a safe state rather than to try to continue processing, as it will not be known what areas of memory have been corrupted as a result of the illegal access.

User mode Tasks can pass messages to each other using the standard queue and semaphore mechanisms. Shared memory regions can be explicitly created but this should be avoided.

Some of the Privileged memory is reserved by the RTOS for its own use. Calling an API function will require a temporary switch to Privileged mode. All non-stack data required by the RTOS kernel is also stored in Privileged memory areas.

Interrupt handlers run in Privileged mode and are therefore typically not constrained by the MPU, or run with the permissions in force when the interrupt occurs.

SAFERTOS MPU SUPPORT

SAFERTOS supports the definition and manipulation of MPU regions on a per task basis. This feature provides the tools allowing developers to add a degree of spatial separation between tasks, which used effectively, can help prevent tasks directly making unintentional or accidental access to incorrect memory regions.

The RTOS AND THE MPU

Memory partitioning and the use of it by Tasks must either be managed by an RTOS on behalf of the application, or by the application directly.

Direct management of the MPU by the application can be difficult, as the engineer is entirely responsible for coding the MPU’s use of the application at a lower level, and will need to specify the context for every switch. Care must be taken to ensure that the safety-critical regions of memory are protected from the non-certifiable code. This will inevitably make safety certification more difficult.

An RTOS that has native support for the MPU provides an API and the tools needed to split the application up into Tasks or threads and manages communication between Tasks with Queues or Events.

Use of an RTOS that provides native support for the MPU allows more opportunity for enforcing partitioning at a fundamental level. This makes it much easier to ensure integrity and to gain certification to a high Safety Integrity Level (SIL) for the application.