Interrupts

Table of Contents

1 ModuleManagerInterrupt....2
1.1 Preview....2
1.2 Initialization....2
1.3 Interrupt handlers....2
1.4 Interrupt activation....2
2 PhysicalCriticalSection....3
2.1 Preview....3
2.2 Implementation....3

1 ModuleManagerInterrupt.

1.1 Preview.

1.1.1 *ModuleManagerInterrupt* is the main interrupts class.

1.1.2 Each interrupt handler must register with this main manager.

1.1.3 When a handler is registered in the main class, the main class also activates the related irqs.

1.1.4 A single handler may listen to several irqs invocations.

1.1.5 The main manager has no periodic actions.

1.2 Initialization.

1.2.1 Reset the handlers list.

1.2.2 During the system initialization phase, each module must register the interrupts' handlers through the main interrupts manager.

1.2.3 When a handler is registered, all related interrupts are enabled.

1.3 Interrupt handlers.

1.3.1 Interrupt handlers must implement the pure virtual '*Interrupt*' class.

1.3.2 Each handler has a one-to-many relation with '*TypeInterruptHandler*'. i.e. a single handler may handle multiple interrupts.

1.3.3 Each handler can register with as many irqs as needed.

1.4 Interrupt activation.

1.4.1 When an interrupt is invoked, the c handler redirects immediately to the main interrupts manager.

1.4.2 Each such invocation has two parameters

1.4.2.1 '*TypeInterruptHandler'* which determines the handler.

1.4.2.2 '*TypeInterruptAction'* allows for cases where a single handler handles multiple interrupts.

2 PhysicalCriticalSection.

2.1 Preview.

2.1.1 The *PhysicalCriticalSection* class is currently designed to protect a single processor critical sections from user interrupts.

2.1.2 The SysTick (as well as general failures) is not stopped by the CriticalSection object, as it is deemed a fast, and simple, essential interrupt.

2.1.3 The class is implemented as a simple, one level class, as it must be as efficient as possible.

2.2 Implementation.

2.2.1 Static ‘m_active_critical_section’

2.2.1.1 The class holds a static member holding a linked list of active critical section.

2.2.1.2 We need this mechanism, as setting, and resetting a group of interrupts is not an atomic action (setting a subgroup of interrupts is an atomic action, so it is safe).

2.2.1.3 The critical section protection might be preempted by higher priority interrupts.

2.2.1.4 We must protect against resuming the interrupts (when exiting critical section), while we activate a lower priority interrupt before higher priority (as interrupts are not resumed by priority order for code efficiency).

2.2.1.5 We must protect against a case where an interrupts is enabled by an (higher level) interrupt handler.

2.2.2 constructor:

2.2.2.1 Set the local state to CRITICAL_SECTION_*REGISTER*.

2.2.2.2 If the exists an active critical section, and if the active critical section is in ‘CRITICAL_SECTION_RESUME’ state, resume that critical section registered interrupts. This would handle the case of a low level enabled interrupt preempting a high level disabled interrupt.

2.2.2.3 Register locally the global active critical section.

2.2.2.4 Register this class as the global critical section.

2.2.2.5 Register locally active interrupts bitmap.

2.2.2.6 Disable all the active interrupts.

2.2.2.7 Set the local state to CRITICAL_SECTION_ACTIVE.

2.2.3 destructor:

2.2.3.1 Set local state to CRITICAL_SECTION_RESUME

2.2.3.2 Activate the interrupts according to the local state.

2.2.3.3 If the local active critical section is not NULL, and the state is CRITICAL_SECTION_REGISTER, register active interrupts (for that active object). This would handle the case where some interrupt updated the active interrupts list, and the active object has not reset that interrupt yet. (This is a valid program behavior, though not recommended).

2.2.3.4 Set the global active object to the local (previous) active object.