Embedded Systems

Design Patterns for Embedded Systems

Design Patterns for Embedded Systems

- Categories of Design Patterns include:
  - Accessing hardware
  - Concurrency and Resource Management
  - State Machine implementation and usage
  - Safety and Reliability

Cyclic Executive Pattern - Abstract

- used for very small embedded applications where it allows multiple tasks to be run pseudo-concurrently without the overhead of a complex scheduler or RTOS.

Cyclic Executive Pattern - Problem

- Many embedded systems are tiny applications that have extremely tight memory and time constraints and cannot possibly host even a scaled-down microkernel RTOS.
- The Cyclic Executive Pattern provides a low-resource means to accomplish Schedulability for a set of tasks.

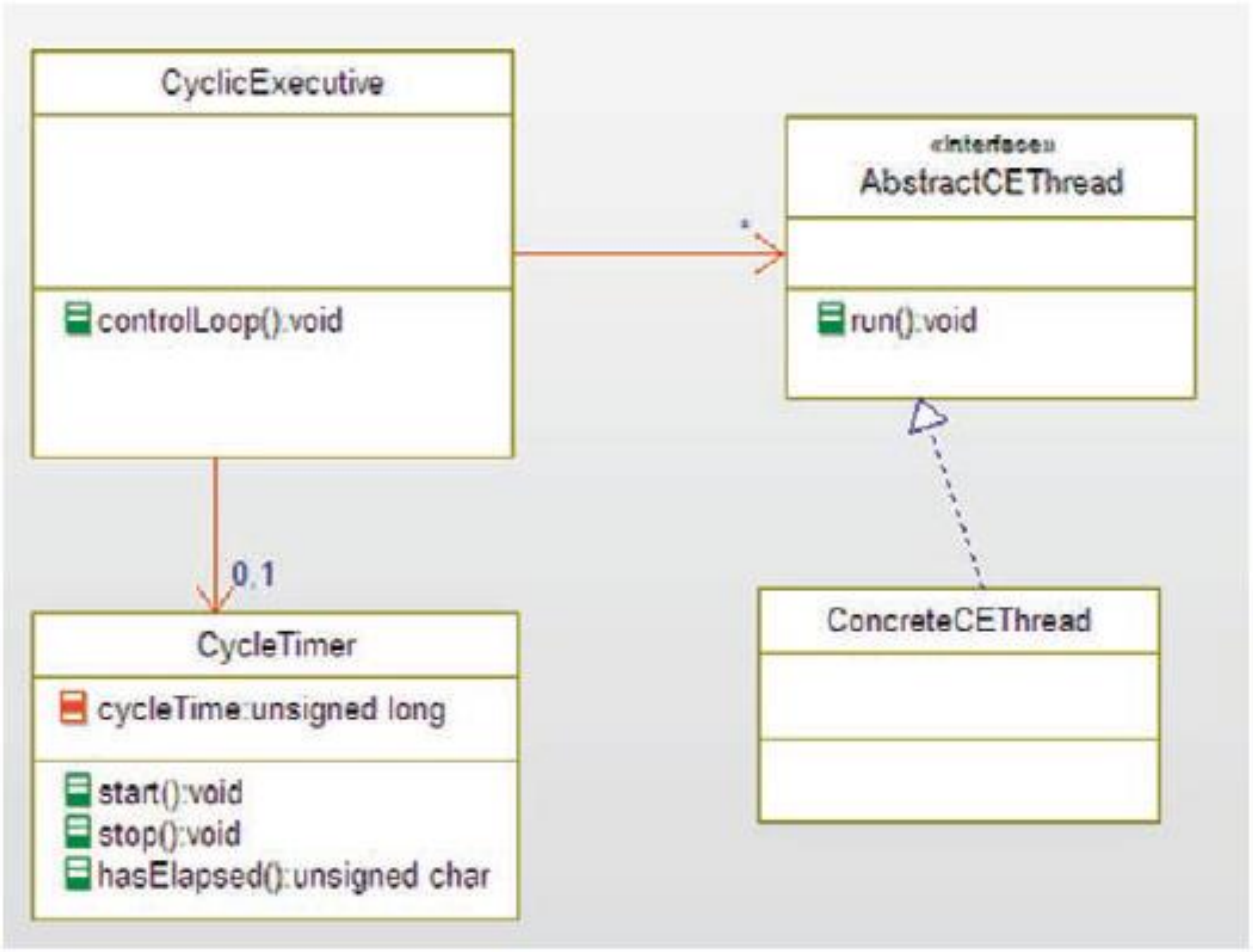
Cyclic Executive Pattern - Problem

- D_j is the deadline for task j
- C_j is the worst-case execution time for task j
- K is the cyclic executive loop overhead, including the overhead of the task invocation and return

for all tasks i, $D_i \geq \sum_{j=1}^{n} C_j + K$

Equation 4-1: Schedulability for Cyclic Executive with *n* tasks

Cyclic Executive Pattern - Structure



Cyclic Executive Pattern - Structure

- The *AbstractCEThread* provides a standard interface for the threads by declaring a run() function.
- It is this function that the Cyclic Executive will invoke to execute the task.
- When this function completes, the CyclicExecutive runs the next task in the list.

Cyclic Executive Pattern - Structure

- The *CyclicExecutive* contains the infinite loop that runs the tasks each in turn.
- The CyclicExecutive contains global stack and static data that are needed either by the tasks.
- The *time-triggered Cyclic Executive* will set up and use the CycleTimer to initiate each epoch (execution of the task list).

Cyclic Executive Pattern - Structure

- The *CycleTimer* isn't used in the most common variant of the CyclicExecutive.
- This is indicated with the "0,1" multiplicity on the directed association in the structure figure.
- This timer can invoke an interrupt when it expires or, even more simply, can return TRUE (non-zero) to the *hasElapsed()* function.
- The CyclicExecutive will then call *start()* to begin the timing interval for the next epoch.

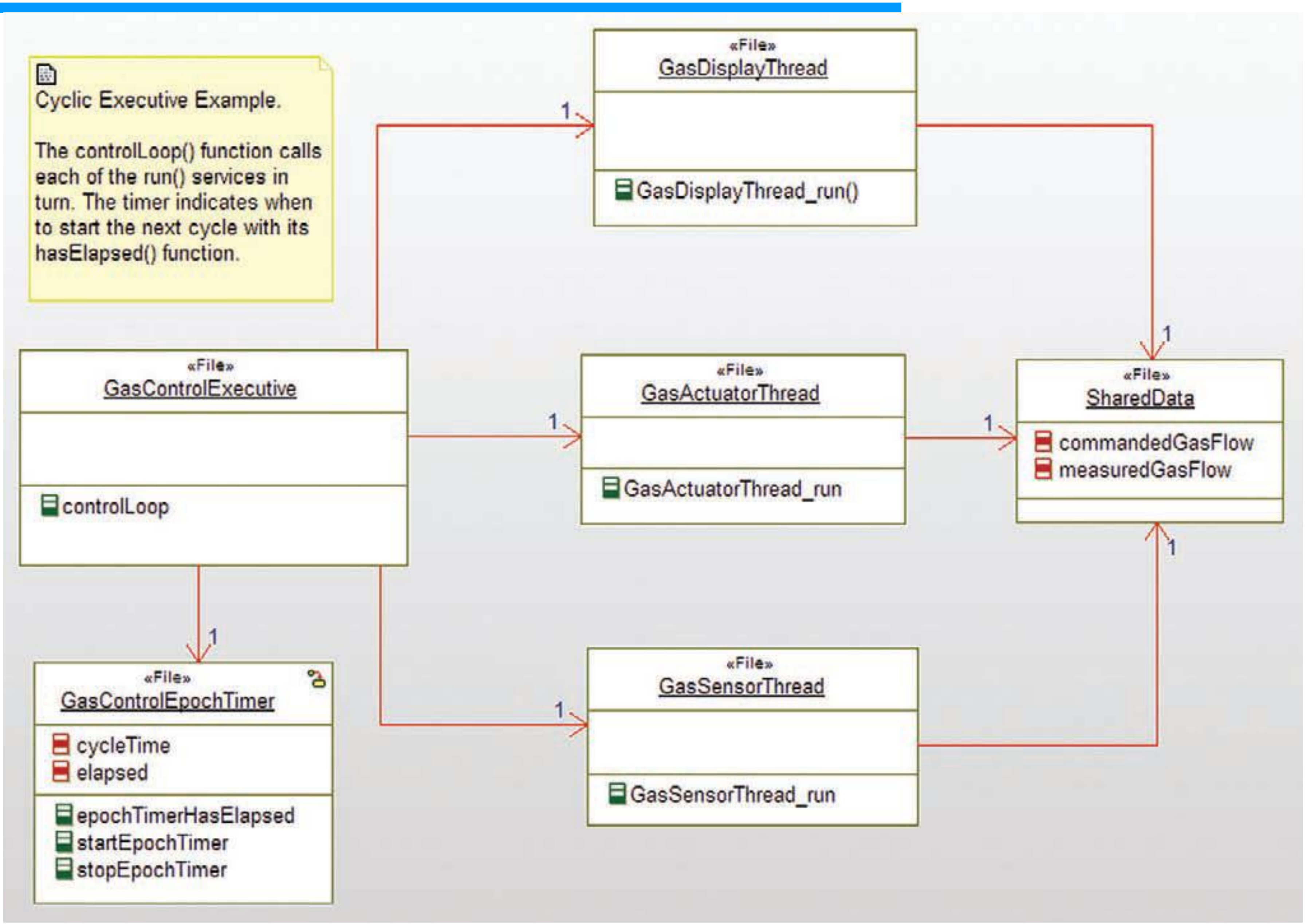
Cyclic Executive Pattern - Consequences

- The primary benefit of this pattern is its simplicity
- It is very lightweight in terms of required resources and so is appropriate for very small memory devices.
- because of the lack of preemption, unbounded blocking is not a concern.

Cyclic Executive Pattern - Consequences

- The pattern is less responsive to high-urgency events, and this makes the pattern inappropriate for applications that have high responsiveness requirements.
- A downside of using the pattern is that the interaction of threads is made more challenging than other patterns.
- If an output of one task is needed by another, the data must be retained in global memory or a shared resource until that task runs

Cyclic Executive Pattern - Example



Critical Region Pattern - Abstract

- The critical region pattern used for task coordination around.
- This happens by disabling task switching during a region where it is important that the current task executes without being preempted.
- This can be because it is accessing a resource that cannot be simultaneously accessed safely, or because it is crucial that the task executes for a portion of time without interruption.

Critical Region Pattern– Problem

- There are two primary circumstances under which a task should not be interrupted or preempted.
- First, it may be accessing a resource that may not be safely accessed by multiple clients simultaneously.
- Secondly, the task may be performing actions that must be completed within a short period of time or that must be performed in a specific order.
- This pattern allows the active task to execute without any potential interference from other tasks.

Critical Region Pattern– Structure

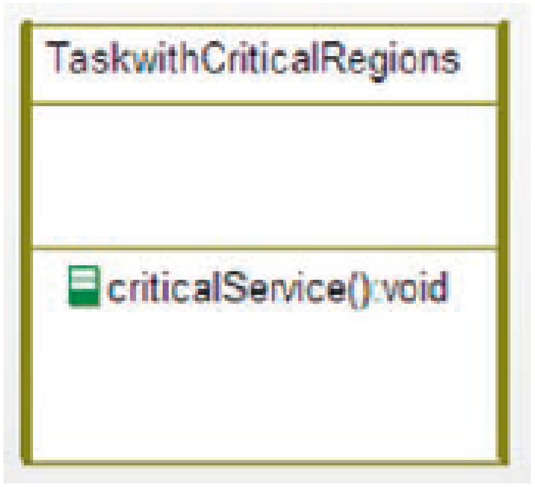
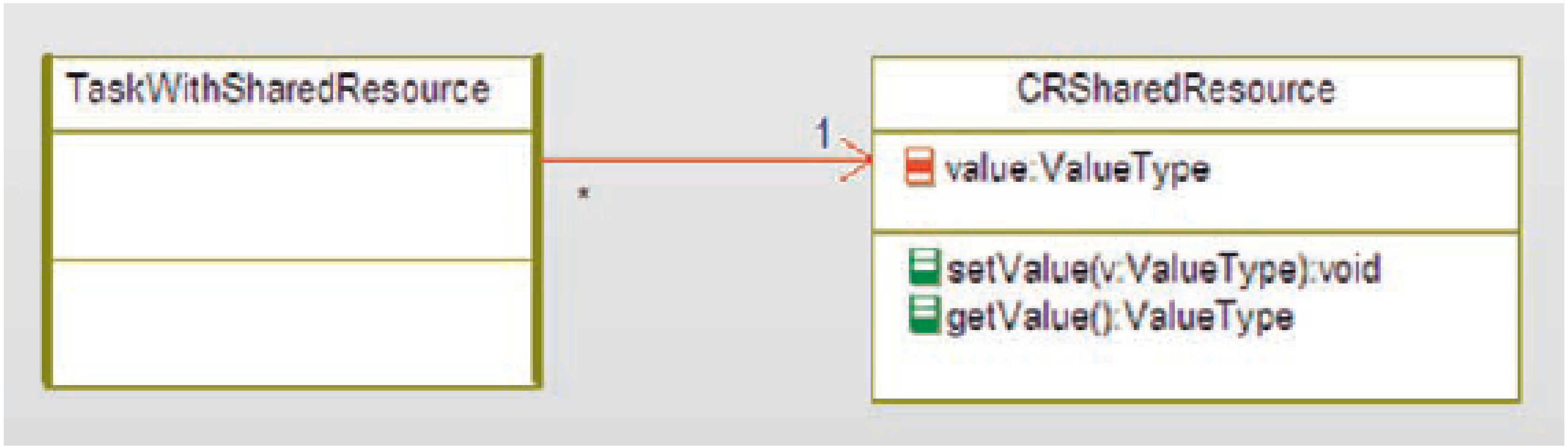


Figure 4-12: Critical Region Pattern



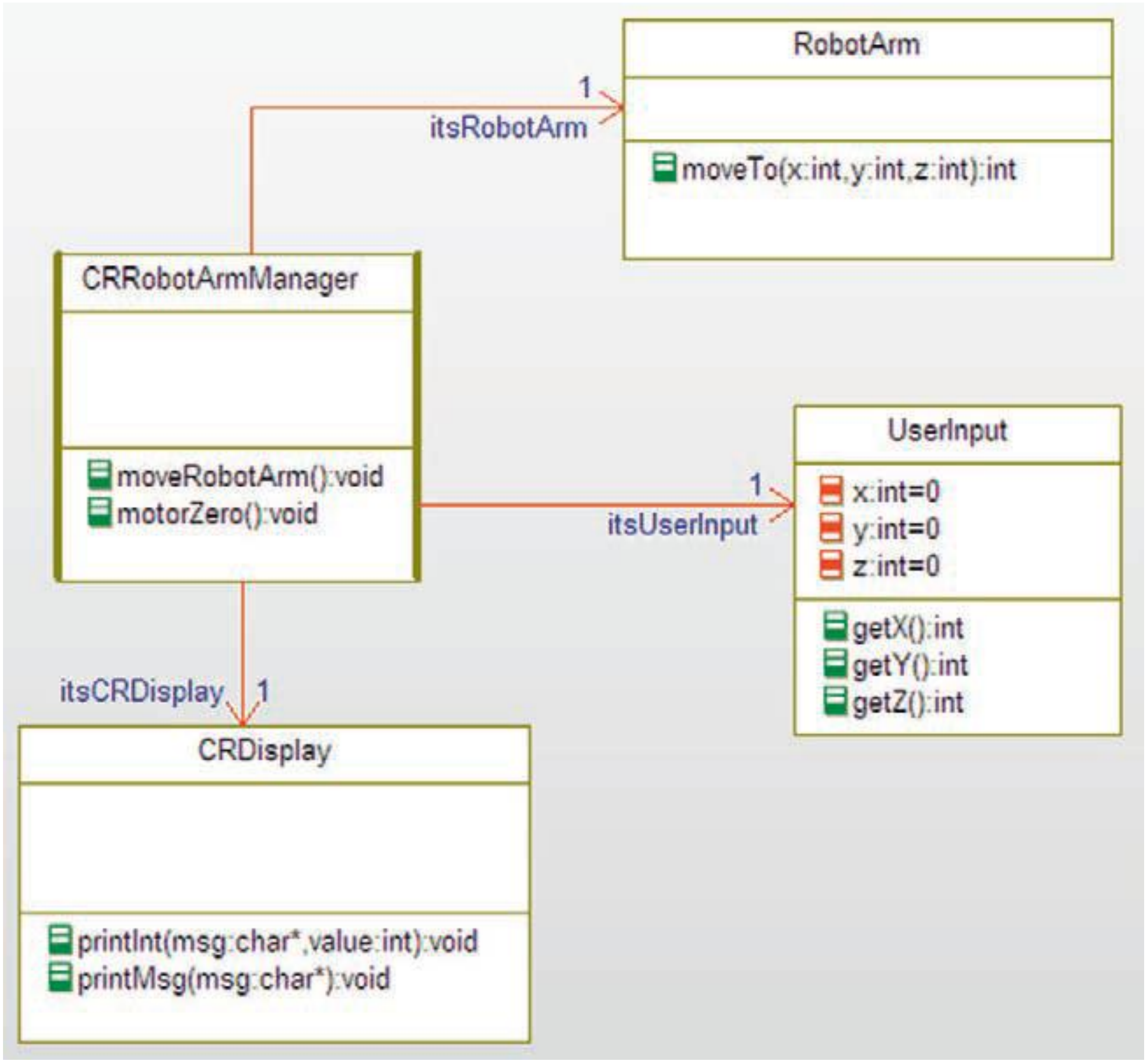
Critical Region Pattern - Consequences

- This pattern enforces the critical region policies well and can be used to turn off the scheduled task switches or, disable interrupts all together.
- Care must be taken to reenable task switching once the element is out of its critical region.
- Additionally, the use of this pattern can affect the timing of other tasks. Critical regions are usually short in duration for this reason.
- There is no problem of unbounded priority inversion because all task switching is disabled during the critical region.

Critical Region Pattern - Consequences

- Care must be taken when nesting calls to functions with their own critical regions.
- If such a function is called within the critical region of another, the nest function call will reenable task switching and end the critical region of the caller function.

Critical Region Pattern `- Example



References

- **Chapter 4**: Douglass, Bruce Powel. **Design patterns for embedded systems in C: an embedded software engineering toolkit**. Elsevier, 2010.