Solving the synchronization problem in multi-core embedded real-time systems

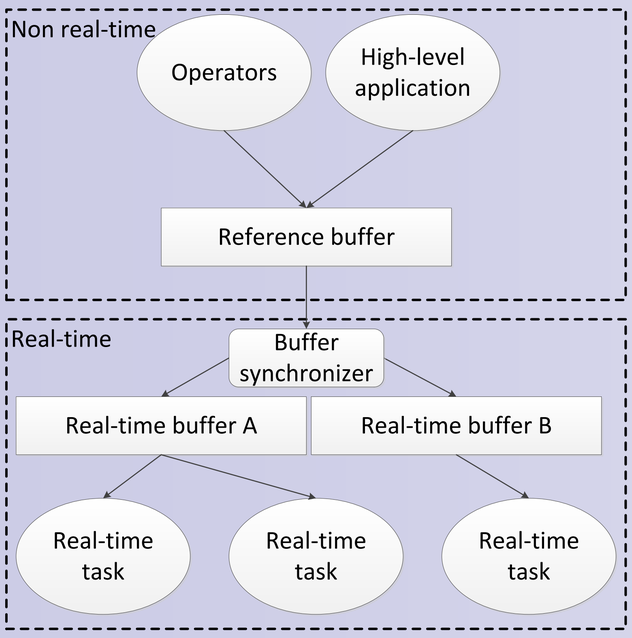
F. Hoguin, S. Deghaye, CERN, Geneva, Switzerland

Abstract

Multi-core CPUs have become the standard in embedded real-time systems. In such systems, where several tasks run simultaneously, developers can no longer rely on high priority tasks blocking low priority tasks. In typical control systems, low priority tasks are dedicated to receiving settings from the control room, and high priority real-time tasks, triggered by external events, control the underlying hardware based on these settings. Settings' correctness is of paramount importance and they must be modified atomically from a real-time task's point of view. This is not feasible in multi-core environments using classic double-buffer approaches, mainly because real-time tasks can overlap, preventing buffer swaps. Other common synchronization solutions involving locking critical sections introduce unpredictable jitter on real-time tasks, which is not acceptable in CERN's control system. We present a lock-free, wait-free solution to this problem based on a triple buffer, guaranteeing atomicity no matter the number of concurrent tasks. The only drawback is potential synchronization delay on contention. This solution has been implemented and tested in CERN's real-time C++ framework (FESA).

From a two-buffer to a three-buffer solution

The two-buffer approach, while suitable for single-core real-time systems, does not work on multi-core systems where, at any time, any number of real-time tasks (readers) can be reading settings values from the active buffer. We use a third buffer to ensure settings values' correctness while providing a strong real-time guarantee. Two real-time buffers, isolated from the non-real-time part, are updated alternatively with new setting values coming from the reference buffer. This work is orchestrated by the Buffer Synchronizer that ensures a real-time buffer is free of readers before updating it. Atomic operations available on all modern CPUs spare the use of locking mechanisms in the real-time part, ensuring a constant time between an event and the execution of the corresponding task.

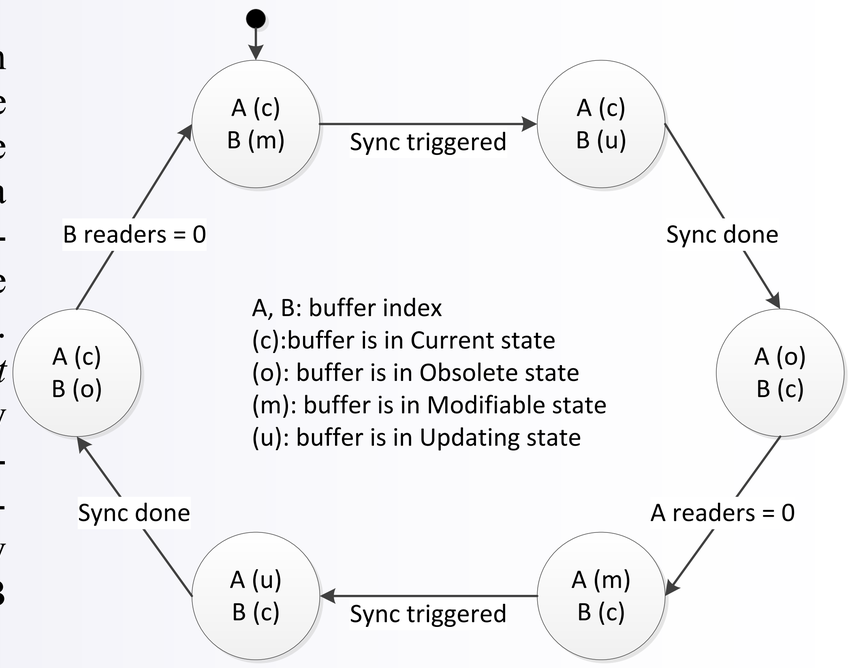


Buffer states and transitions

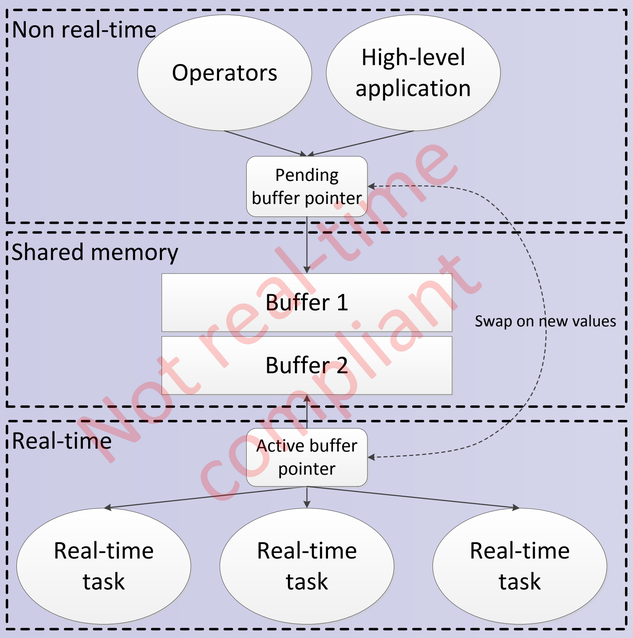
The real-time buffers can be in one of the following four states:

- Current: the buffer can safely be accessed and contains the current settings.
- Obsolete: the buffer can safely be accessed and contains old settings.
- Modifiable: the buffer is not in use and cannot be accessed. It is waiting for an update of setting values.
- Updating: new setting values are being copied in the buffer.

Buffers A and B are always in different but related states.



At start-up, the content of the reference buffer is copied to both real-time buffers A and B, and buffer A is in *Current* state while buffer B is in *Modifiable* state. Whenever the reference buffer is modified and the modification operation committed, a synchronization is triggered ("Sync triggered" transition). Since buffer B is in the *Modifiable* state, setting values are copied by the buffer synchronizer from the reference buffer to buffer B (*Updating*). Once the copy is done ("Sync done" transition), buffer A becomes *Obsolete* and buffer B becomes *Current*. This latter transition must be atomic to ensure that at any time, one and only one buffer is in the *Current* state. From now on, new readers will read B (the *Current* buffer). When buffer A has no readers any more, it becomes *Modifiable* ("A readers = 0" transition). The next time a synchronization will be triggered (second "Sync triggered" transition), the modified settings will be copied from the reference buffer to buffer A. When the copy is done (second "Sync done" transition), buffer A and buffer B atomically change state, going respectively to *Current* and *Obsolete*. When buffer B has no readers any more ("B readers = 0" transition), it goes back to the state *Modifiable*, which is the initial state.



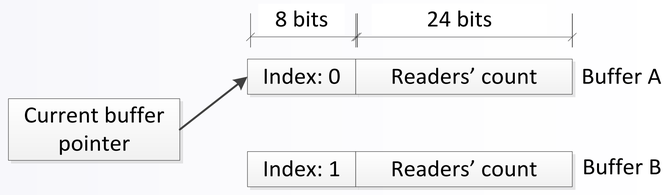
Abandoned solutions

Before developing this three-buffer solution, we analyzed different possibilities which were eventually discarded because of the impossibility to make them real-time compliant.

- Snapshot of the setting values: every real-time task uses its private snapshot. Not real-time compliant because memory allocation is needed, and the jitter depends on the size of the setting values.
- Reader-writer lock mechanism: real-time tasks get a reader lock on the setting values. Buffer swap is done when the writer lock is obtained. Not real-time compliant because real-time tasks would be blocked when copying new setting values.

Guarantee of atomicity in transitions

Our solution works provided transitions between states are atomic, as at any point in time, one and only one real-time buffer must be in the *Current* state. In order to guarantee atomicity of transitions, our implementation uses a structure for each buffer which stores the buffer index (8 bits) and its readers count (24 bits) and can be read and incremented in a single atomic operation (fetch and add). A pointer, whose value can also be changed atomically, designates the current buffer. When a reader starts its execution, it increments the readers count and reads the buffer index with a fetch and add instruction on the current buffer, using GCC's built-in *__sync_fetch_and_add*. When releasing the buffer, it uses an atomic decrement. Finally, the current buffer pointer's value is changed atomically when an *Updating* buffer becomes *Current.* The *Obsolete* buffer is deduced from the current buffer pointer, making the transition of both buffers atomic.



Strong real-time guarantees

- ✓ No memory allocation
- ✓ Consistent setting values throughout execution
- ✓ Constant, <5 ms jitter
- ✓ Pending setting values quickly visible