

# 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 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.

## FROM SINGLE-THREADED TO MULTI-THREADED EMBEDDED SYSTEMS

In typical control systems, accelerators settings are modified either manually by operators using graphical user interfaces or by high-level systems computing hardware settings from high-level values. These settings are then sent by the high-level applications to the computers in charge of the hardware real-time control. This control is done by real-time tasks in a limited time frame following an external trigger. A real-time task is nothing more than a piece of code executed with real-time priority. These tasks typically perform computations based on settings and drive the hardware with the values obtained. Hardware settings can be interdependent and modifications must be applied in a single operation. The real-time tasks must work with consistent sets of settings; from their point of view, settings must be modified atomically.

In single core embedded systems, developers can rely on the determinism of a real-time scheduler to guarantee consistency of settings with a simple double buffer system; one for active values (accessed by real-time tasks) and another one for pending values i.e. just modified values not yet accessible to real-time tasks. When the modification is done, the set of pending values is consistent and buffers may be swapped. A low priority task swaps pointers to the buffers atomically; the pending buffer becomes active, and vice versa. This is guaranteed

to be safe provided no real-time task is ever in a waiting state in the middle of its execution, ensuring that the buffer swapping task can never be executed while a real-time task is being executed.

As multi-core embedded systems become the norm, such assumptions cannot be made anymore. At CERN, most of the control system embedded systems use 2-core processors. In such setups, the buffer swapping task can be executed concurrently to a real-time task. As a result, swapping pending and active buffers without further checks could lead to a real-time task reading inconsistent settings. An example is shown in table 1.

Table 1: Example of Naive Buffer Swap with Multi-core CPU

Real-time task	Buffer swapping thread	Buffer 1	Buffer 2
Reads voltage => 200V	Swap triggered...	V = 200, A = 100	V = 10, A = 1000
Busy computing...	Swaps buffers	V = 200, A = 100	V = 10, A = 1000
Reads current => 1000A		V = 200, A = 100	V = 10, A = 1000
Tells power supply: 200V, 1000A		V = 200, A = 100	V = 10, A = 1000

## EXPLORED AND ABANDONED SOLUTIONS

In order to solve this problem, several solutions were explored. The goal was to implement an algorithm that fulfils the following requirements:

1. The solution shall be real-time compliant (in particular, no dynamic memory allocation is allowed).
2. Real-time tasks shall read consistent setting values throughout their execution.
3. Time interval between a trigger and the execution of the corresponding real-time task shall be constant and below 5 milliseconds (a jitter of 10% is acceptable).
4. Pending setting values shall be made available to real-time tasks within a reasonable time frame.

## Snapshot of the Setting Values

A possible solution is to make a snapshot of the setting values just as a real-time task is about to start. This snapshot is private to that execution of the task. While this would work, it is not real-time compliant as memory would be dynamically allocated to copy the setting values. Even a fixed size memory pool could be exhausted given enough simultaneous real-time tasks. Also, the jitter between the trigger and the task execution

depends on the size of the setting values. This solution fails requirements #1 and #3.

### Reader-Writer Lock Mechanism

In this implementation, real-time tasks acquire a reader lock just before starting their execution. Swapping settings buffers is done when the corresponding writer lock can be acquired. While this solution looks like it could work, there are unavoidable cases where a real-time task would be delayed beyond acceptable limits. This is typically the case when two real-time tasks overlap (see table 2 for an example).

Table 2: Reader-writer Example Preventing RT Task Execution

Real-time task A	Real-time task B	Buffer swapping task
Starts. Acquires reader mutex.		
Executing		Swap required. On hold waiting for writer mutex.
Executing	Starts. Blocks trying to acquire reader mutex because of “Buffer Swapping” task.	On hold
Executing	Blocked	On hold
Finishes. Releases reader mutex.	Blocked	Acquires writer mutex and swaps buffers.
	Acquires reader mutex.	Releases writer mutex.

In this situation, task B is blocked for as long as task A is executing, which is not acceptable as it fails requirement #3. To avoid real-time tasks blocking, readers could have priority over writers. But in this case, overlapping real-time tasks would prevent settings from ever being updated; the buffer swapping task would never acquire the writer mutex as, at any time, one of RTA or RTB would hold a reader mutex. This fails requirement #4.

## A LOCK-FREE, WAIT-FREE SYNCHRONIZATION SOLUTION

Requirement #3 means that real-time tasks should never block and never wait. We demonstrated earlier that these requirements cannot be satisfied with a traditional pointer swapping implementation, and that using any form of locking before executing a real-time task is impossible.

We introduce a solution that, by the use of an additional setting values buffer, guarantees settings consistency without using any locking mechanism in real-time tasks. Furthermore, we guarantee that the delay between an

event and the execution of the corresponding real-time task is fixed (requirement #3) by ensuring a fixed set of operations is executed in-between.

Instead of using an active settings buffer and a pending settings buffer, we use a reference settings buffer and two real-time settings buffers. The reference buffer contains the latest setting values, as modified by operators. This buffer is never accessed by real-time tasks. The real-time settings buffers are copies of the reference buffer at a certain point in time. From now on, we'll refer to the real-time buffers as buffer A and buffer B. They 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. Transition between states for a single buffer is presented in Fig. 1.

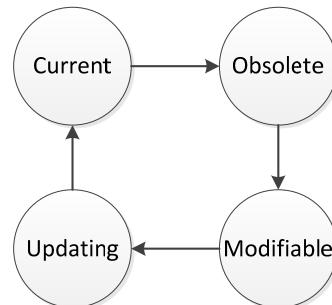


Figure 1 Buffer's states.

### Overview of the Behaviour

The following simplified description focuses on buffer A; buffer B follows the same pattern. Relations between buffer states are described in depth in the next section.

When a real-time task reacts to an event, it becomes a reader of the *Current* buffer (e.g. buffer A). Whenever newer values are available in the other buffer (buffer B), buffer A becomes *Obsolete*. Buffer A is guaranteed to have its readers count eventually reduced to 0 since it cannot receive new readers. As soon as buffer A has no longer any readers, it becomes *Modifiable*. At some point in time, new settings will be available and will be copied from the reference buffer to buffer A; it goes to the state *Updating*. When the copy is done, buffer A becomes *Current* at the same time as buffer B becomes *Obsolete*, coming back to the initial state. Operations available on buffers depending on their state are listed in table 3.

Table 3: Buffer Access Rights

	Can be accessed by readers	Can get new readers	Can be modified
Current	Yes	Yes	No
Obsolete	Yes	No	No
Modifiable	No	No	Yes
Updating	No	No	Yes

### States Transitions

A state machine representing this algorithm is presented in Fig. 2.

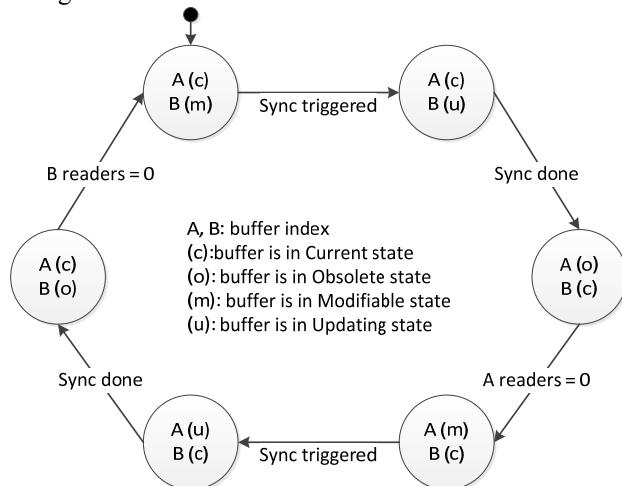


Figure 2: Complete two-buffer state machine.

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 a low priority task (the buffer synchronizer) from the reference buffer to buffer B; the latter goes to the state *Updating*. If no buffer is *Modifiable*, the copy operation is put on hold. To ensure consistency of settings, the buffer synchronizer holds a reader lock on the reference buffer during the copy. This time interval depends on the number of modified settings, their size and the bandwidth of the computer memory. On modern systems, it is typically very short, between a few microseconds to a few milliseconds. During that period of time, the reference buffer cannot be modified and the threads writing the settings coming from the high-level application are blocked. This delay is acceptable as the transfer of settings uses the Ethernet network, which is not real-time. In addition, the buffer synchronizer keeps a list of just-copied settings, the “settings to replicate”, which will be useful when the next synchronization occurs. 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. 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. This time though, this will not be sufficient as, at this point, buffer A is not up-to-date with respect to the settings that were copied earlier to buffer B. The buffer synchronizer needs to use the “settings to replicate” and copy them from buffer B to buffer A. When the copy is done (second “Sync done” transition), buffer A and buffer B atomically change state, going to *Current* and *Obsolete* respectively. When buffer B has no readers any more (“B readers = 0” transition), it goes back to the state *Modifiable*, which is the initial state.

## IMPLEMENTATION

This algorithm has been implemented in CERN’s real-time C++ framework. We present details of our implementation.

### Atomic Operations

Some operations need to be carried out atomically to ensure proper functioning of the algorithm. The list of required atomic operations is as follows:

- Fetch and increment or increment and fetch on 32 bits (can be reduced to 8 bits)
- Decrement on 32 bits (can be reduced to 8 bits)
- Compare and swap on a pointer (optional, for validation purposes only, can be replaced by a write operation)

As explained earlier, it is required to always have one and only one buffer in the *Current* state. This is achieved by using a pointer to the *Current* buffer, whose value can be changed atomically (supported by all modern CPU architectures [1][2]). Instead of a simple write, an atomic compare and swap is used in our implementation to ensure that the pointer value is as expected before modifying it; this is for validation purposes only and, in production, only the write is required.

When a new reader requests access to the *Current* buffer, its readers count is incremented, and its index is retrieved and assigned to the reader. Since the *Current* buffer pointer can be changed at any time, this sequence (increment and read) needs to be atomic as well. Our implementation uses a structure that can be modified atomically. It contains the buffer index in the most significant part and the number of readers in the least significant part. We use 8 bits for the buffer index and 24 bits for the reader index (see Fig. 3). This structure allows us to use a “fetch and increment” that will atomically increment the number of readers and read the buffer index. This primitive is again supported by all modern CPU architectures [3][4]. Note that on architectures with limited resources, the number of bits can be reduced from 8 to 1 and from 24 to 7 bits respectively, while still allowing a maximum of 128 readers.

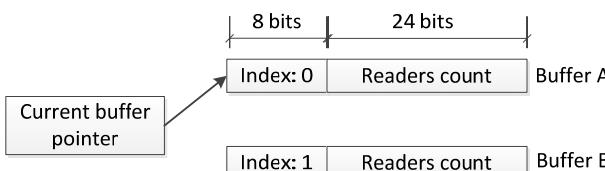


Figure 3: Internal structure for buffer management.

When a reader no longer needs access to a buffer, its readers count must be decremented. This also needs to be atomic because other readers may simultaneously modify this count. However, in this case, there is no need to retrieve the buffer index as it was retrieved when requesting access to the buffer. Therefore, an atomic decrement is sufficient and, again, this operation is supported by all modern architectures [3][4].

Our implementation uses GCC atomic built-ins `_sync_fetch_and_add`, `_sync_fetch_and_sub`, and `_sync_bool_compare_and_swap` [5], requiring no handwritten assembly at all. Such intrinsics exist on other systems [6] and, as of C++11, in the language itself [7].

### Synchronization Trigger

The synchronization of setting values is triggered when the reference buffer contains new consistent setting values. However, the synchronization can only happen if a real-time buffer is in the *Modifiable* state. Figure 2 shows that a real-time buffer is in the *Modifiable* state if it is not the *Current* buffer and it has no readers.

The buffer synchronizer thread (BST) ensures that a synchronization happens as soon as possible by waiting for the *Obsolete* buffer to go to *Modifiable*. Once a *Modifiable* buffer is available, the BST checks continuously whether a synchronization is needed. This is achieved in several steps. First, the BST deduces from the *Current* buffer pointer which buffer is *Obsolete*. The *Current* buffer pointer is guaranteed not to change since the BST is the only thread that can change it. To avoid a busy wait when the number of readers of the *Obsolete* buffer is greater than 0, we use a condition variable which is signalled by readers when a buffer's readers count reaches 0. A synchronization request is indicated to the BST via a boolean. Again, an expensive busy wait is avoided thanks to a condition variable which is signalled whenever the boolean is set to true.

## CONCLUSION

This synchronization algorithm fulfils all the requirements:

- Real-time compliant (no memory allocation)
- Consistent setting values (real-time buffers cannot be modified while they have readers)
- Constant jitter between event and task execution thanks to a fixed flow of execution and an O(1) algorithm
- Settings made available as soon as possible

Nevertheless, implementing the solution is not free and two major drawbacks have to be mentioned. First, there is an obvious additional memory consumption; a third buffer is required compared to the simpler double-buffer approach. In modern systems, this is probably not an issue as the amount of settings is typically small compared to the amount of available RAM. The second drawback is the lack of control on the delay between the synchronization request and the actual availability of the new settings to the real-time tasks. As the synchronization cannot occur before all the *Obsolete* buffer's readers have completed their execution, a slow real-time task can delay the synchronization. Therefore, it is possible to have new real-time tasks executions not using the latest settings. In our case, this is not a problem as we consider the sending of new settings a slow and non-deterministic operation. If this limitation is incompatible with the system to be controlled, the possible evolution based on the usage of additional real-time buffers is detailed in the next section.

### Possible Evolution

Our implementation uses two real-time buffers, but in practice, for busy real-time application with many tasks and frequent changes of settings, one can use as many buffers as the available memory allows. This would reduce the likelihood not to have any *Modifiable* buffer on synchronization request and therefore reduce significantly the delay between settings modification and settings availability. The management of *Modifiable* buffers would need to be adapted so that the first available *Modifiable* buffer can receive a copy of the newest setting values. In practice, two real-time buffers should be sufficient for most if not all real-time applications.

## REFERENCES

- [1] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, Section 8.1.1, Guaranteed atomic operations; <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architecture-software-developer-vol-3a-part-1-manual.pdf>
- [2] ARM Synchronization Primitives, Section 1.2, Exclusive accesses; [http://infocenter.arm.com/help/topic/com.arm.doc.dht0008a/DHT0008A\\_arm\\_synchronization\\_primitives.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.dht0008a/DHT0008A_arm_synchronization_primitives.pdf)
- [3] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, Section 8.1.2.2, Software Controlled Bus Locking; <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architecture-software-developer-vol-3a-part-1-manual.pdf>
- [4] ARM Synchronization Primitives, Section 1.3.3, Implementing a semaphore; [http://infocenter.arm.com/help/topic/com.arm.doc.dht0008a/DHT0008A\\_arm\\_synchronization\\_primitives.pdf](http://infocenter.arm.com/help/topic/com.arm.doc.dht0008a/DHT0008A_arm_synchronization_primitives.pdf)

- [5] Atomic Builtins – Using the GNU Compiler Collection (GCC), <http://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html>
- [6] InterlockedIncrement function (Windows),  
[https://msdn.microsoft.com/en-us/library/windows/desktop/ms683614\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms683614(v=vs.85).aspx)
- [7] std::atomic::fetch\_add, C++ reference,  
[http://en.cppreference.com/w/cpp/atomic/atomic/fetch\\_add](http://en.cppreference.com/w/cpp/atomic/atomic/fetch_add)