EB tresos® AutoCore OS Atomics documentation for TRICORE

product release 6.0

Elektrobit Automotive GmbH
Am Wolfsmantel 46
91058 Erlangen, Germany
Phone: +49 9131 7701 0
Fax: +49 9131 7701 6333
Email: info.automotive@elektrobit.com

Technical support

https://www.elektrobit.com/support

Legal disclaimer

Table of Contents

1. About this documentation

1.1. Typography and style conventions

Throughout the documentation you see that words and phrases are displayed in bold or italic font, or in Monospace font. To find out what these conventions mean, consult the following table. All default text is written in Arial Regular font without any markup.

Convention	Item is used	Example
Arial italics	to define new terms	The *basic building blocks* of a configuration are module configurations.
Arial italics	to emphasize	If your project's release version is mixed, all content types are available. It is thus called *mixed version*.
Arial italics	to indicate that a term is explained in the glossary	...exchanges *protocol data unit*s (*PDU*s) with its peer instance of other ECUs.
Arial boldface	for menus and submenus	Choose the **Options** menu.
Arial boldface	for buttons	Select **OK**.
Arial boldface	for keyboard keys	Press the **Enter** key
Arial boldface	for keyboard combination of keys	Press **Ctrl**+**Alt**+**Delete**
Arial boldface	for commands	Convert the XDM file to the newer version by using the **legacy convert** command.
Monospace font (Courier)	for file and folder names, also for chapter names	Put your script in the `function_name/abc-folder`
Monospace font (Courier)	for code	`for (i=0; i<5; i++) { /* now use i */ }`
Monospace font (Courier)	for function names, methods, or routines	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Monospace font (Courier)	for user input/indicates variable text	Enter a `three-digit prefix` in the menu line.
Square brackets [ ]	for optional parameters; for command syntax with optional parameters	`insertBefore [<opt>]`

<table>
<tr><th>Convention</th><th>Item is used</th><th>Example</th></tr>
<tr><td>Curly brackets {}</td><td>for mandatory parameters; for command syntax with mandatory parameters (in curly brackets)</td><td>insertBefore {<file>}</td></tr>
<tr><td>Three dots …</td><td>for further parameters</td><td>insertBefore [<opt>…]</td></tr>
<tr><td>A vertical bar |</td><td>to separate parameters in a list from which one parameters must be chosen or used; for command syntax, indicates a choice of parameters</td><td>allowinvalidmarkup {on|off}</td></tr>
<tr><td>Warning</td><td>to show information vital for the success of your configuration</td><td>WARNING
This is a warning
This is what a warning looks like.</td></tr>
<tr><td>Notice</td><td>to give additional important information on the subject</td><td>NOTE
This is a notice
This is what a notice looks like.</td></tr>
<tr><td>Tip</td><td>to provide helpful hints and tips</td><td>TIP
This is a tip
This is what a tip looks like.</td></tr>
</table>

2. EB tresos product line support

https://www.elektrobit.com/support

3. Atomics user's guide

This chapter deals with the atomic functions offered by EB tresos AutoCore OS. Its first section introduces the problem to be solved by them: concurrent memory accesses to shared memory locations by independent processor cores or other bus masters. With this background, the following two sections discuss the two important aspects of memory accesses in concurrent environments: *atomicity* and *consistency*. These constitute the vital basis for synchronization algorithms which are needed to coordinate the concurrent work.

3.1. Operating principles

3.1.1. Motivation

The need of *atomic functions* is generated by recent developments in microprocessor design. The trend to more concurrent architectures is evident. This means that multiple threads of execution may exists at any point in time.

Of course, these threads must be coordinated somehow to achieve proper synchronization with the serial parts of a program and that's one motivation for the atomic functions discussed in this document. Another motivation is to have the concurrent parts of a program working together without much friction. Other synchronization mechanisms offered by EB tresos AutoCore OS, for example, impose a high performance penalty, because context switches and dozens of instructions are executed when they are used. In some cases, this performance hit is not acceptable, and that's when specialized instructions offered by the hardware may come to the rescue. Although, these provide less complex synchronization mechanisms, if an algorithm makes efficient use of them, they can remedy the performance penalty mentioned above.

Key aspects in this realm are *atomicity* and *consistency* which are the subject of the next sections.

3.1.2. Atomicity of memory accesses

The term *atomicity* relates to a certain trait of memory accesses. Atomic accesses are never interrupted by other concurrent threads, even when they access the same memory location at the same time. They are executed either completely or not at all. The net result of an atomic access is as if there is no contention at all. This also extends to read-modify-write instructions which are naturally susceptible to interferences of other concurrent threads. This is because they first have to read from memory, process the data, and then write it back. This gives other threads the opportunity to intercept these activities so that the final value in memory is not as expected.

With guaranteed atomicity these issues can't occur and concurrent threads can work on shared data without the need of more expensive synchronization mechanisms provided by EB tresos AutoCore OS.

This statement would be true if there weren't further optimizations done in hardware and during compilation which may reorder memory accesses. This is the subject of the next section.

3.1.3. Consistency of memory accesses

The term *consistency* in this realm refers to the order in which concurrent threads perceive the write accesses of other threads to shared memory locations. This ordering may be affected by two different layers: hardware and compiler.

The microprocessor hardware may employ different kinds of buffering to avoid updating the system memory each time a write instruction is executed. Furthermore, read instructions may be executed speculatively ahead of time if this seems beneficial. The combined effects of these mechanisms must be taken into account if precise control of memory accesses is required, for example, when peripherals are operated or for synchronization algorithms.

Also, the compiler may *shuffle* read and write instructions for optimization purposes.

The atomic functions provided by EB tresos AutoCore OS enforce *sequential consistency*. This means that, firstly, hardware drains all of its buffers, so that the effects of past write instructions become visible to other threads. Furthermore, no later read instructions are executed speculatively. Additionally, no later write instructions are executed. All this gives an atomic function the properties of a *memory fence*. Such fences preclude reordering of memory accesses at the hardware level. Sequential consistent fences prevent read and write instructions from being moved either way across it.

Secondly, sequential consistency imposes a *total order* on all concurrent accesses to a shared memory location. The consequence is that all concurrent threads agree upon one and only one order in which they observe concurrent accesses to shared memory locations of all accessing threads. That is an important property of the atomic functions, because it is vital to implement synchronization algorithms.

3.2. API reference

This section lists the atomic functions and describes their behavior, inputs, and outputs.

All atomic functions operate on objects with platform-specific types. The type `os_atomic_t` is used for atomic objects which are accessed by multiple threads concurrently. It is *opaque* and thus you must access them only by the functions described in this section. Before you use an atomic object, you must initialize it. To do this, there is the function `OS_AtomicInit()` and the macro `OS_ATOMIC_OBJECT_INITIALIZER`. You can use the former at runtime and the latter at program load time. Atomic objects with static storage duration are automatically initialized at program load time with the initial value zero.

The value of an atomic object has the type `os_atomic_value_t`. This type is not opaque and hence you may use it in C language expressions as any other basic numerical type. It has no atomicity and memory ordering guarantees associated with it and is meant to be accessed by only one thread at any point in time. The maximum value that you can store in an object of type `os_atomic_value_t` is given by the macro `OS_ATOMICS_VALUE_MAX`.

Furthermore, *all* atomic functions (except `OS_AtomicInit()`) exhibit sequential consistency and preclude certain compiler optimizations which strive for moving read and write operations across them. Hence, one can think of this as an implicit call of `OS_AtomicThreadFence()` at the start and end of every atomic function.

3.2.1. OS_AtomicThreadFence()

NAME	**`OS_AtomicThreadFence`**
SYNOPSIS	A sequential-consistent memory fence.
SYNTAX	`void OS_AtomicThreadFence(void)`
DESCRIPTION	This function inserts a sequential-consistent memory fence into the program, where it is called. It prevents read and write instructions from being reordered across it either way. This restriction applies to both the hardware and compiler level. When it returns, all past memory accesses are finished with system-wide visibility.

3.2.2. OS_ATOMIC_OBJECT_INITIALIZER()

NAME	**`OS_ATOMIC_OBJECT_INITIALIZER`**
SYNOPSIS	Initializes an atomic object.
SYNTAX	`OS_ATOMIC_OBJECT_INITIALIZER(initialValue)`
DESCRIPTION	The macro expands to an initializer for an atomic object of type `os_atomic_t` and initializes it with the given initial value.

3.2.3. OS_AtomicInit()

NAME	**`OS_AtomicInit`**
SYNOPSIS	Initializes an atomic object.
SYNTAX	`void OS_AtomicInit(os_atomic_t volatile *object, os_atomic_value_t initialValue)`

NAME	**OS_AtomicInit**
DESCRIPTION	Initializes the atomic object with the given initial value.

3.2.4. OS_AtomicStore()

NAME	**OS_AtomicStore**
SYNOPSIS	Stores the given value atomically.
SYNTAX	`void OS_AtomicStore(os_atomic_t volatile *object, os_atomic_value_t newValue)`
DESCRIPTION	Atomically stores the value `newValue` into the atomic object at `object`.

3.2.5. OS_AtomicLoad()

NAME	**OS_AtomicLoad**
SYNOPSIS	Loads from the given memory location atomically.
SYNTAX	`os_atomic_value_t OS_AtomicLoad(os_atomic_t const volatile *object)`
DESCRIPTION	Atomically loads the value of the atomic object at `object`.
RETURN	The value of the atomic object at `object`.

3.2.6. OS_AtomicExchange()

NAME	**OS_AtomicExchange**
SYNOPSIS	Atomically exchanges values.
SYNTAX	`os_atomic_value_t OS_AtomicExchange(os_atomic_t volatile *object, os_atomic_value_t newValue)`
DESCRIPTION	Atomically exchanges the value of the atomic object at `object` with the value `newValue`.
RETURN	The value of the atomic object at `object` before the exchange.

NAME	OS_AtomicExchange

3.2.7. OS_AtomicCompareExchange()

NAME	OS_AtomicCompareExchange
SYNOPSIS	Atomically compares and exchanges values.
SYNTAX	`os_boolean_t OS_AtomicCompareExchange(os_atomic_t volatile *object, os_atomic_value_t *expected, os_atomic_value_t newValue)`
DESCRIPTION	Atomically exchanges the value of the atomic object at `object` with the value `newValue`, if and only if, its value is equal to the value at `expected`.
RETURN	The value `OS_TRUE`, if the atomic object at `object` was changed and `OS_FALSE` otherwise. In the latter case, the memory pointed to by `expected` is updated to contain the value of the atomic object at `object`, that it had at the point in time, when this function was called.

3.2.8. OS_AtomicFetchAdd()

NAME	OS_AtomicFetchAdd
SYNOPSIS	Atomically adds the given value to the atomic object.
SYNTAX	`os_atomic_value_t OS_AtomicFetchAdd(os_atomic_t volatile *object, os_atomic_value_t operand)`
DESCRIPTION	Atomically adds the value `operand` to the atomic object at `object` and updates it with the result.
RETURN	The value of the atomic object at `object` before the operation.

3.2.9. OS_AtomicFetchSub()

NAME	OS_AtomicFetchSub
SYNOPSIS	Atomically subtracts the given value from the atomic object.

NAME	OS_AtomicFetchSub
SYNTAX	`os_atomic_value_t OS_AtomicFetchSub(os_atomic_t volatile *object, os_atomic_value_t operand)`
DESCRIPTION	Atomically subtracts the value `operand` from the atomic object at `object` and updates it with the result.
RETURN	The value of the atomic object at `object` before the operation.

3.2.10. OS_AtomicFetchOr()

NAME	OS_AtomicFetchOr
SYNOPSIS	Atomically ORs the given value with the atomic object.
SYNTAX	`os_atomic_value_t OS_AtomicFetchOr(os_atomic_t volatile *object, os_atomic_value_t operand)`
DESCRIPTION	Atomically performs the boolean OR operation with the value `operand` and the value of the atomic object at `object` and updates it with the result.
RETURN	The value of the atomic object at `object` before the operation.

3.2.11. OS_AtomicFetchAnd()

NAME	OS_AtomicFetchAnd
SYNOPSIS	Atomically ANDs the given value with the atomic object.
SYNTAX	`os_atomic_value_t OS_AtomicFetchAnd(os_atomic_t volatile *object, os_atomic_value_t operand)`
DESCRIPTION	Atomically performs the boolean AND operation with the value `operand` and the value of the atomic object at `object` and updates it with the result.
RETURN	The value of the atomic object at `object` before the operation.

3.2.12. OS_AtomicFetchXor()

NAME	OS_AtomicFetchXor
SYNOPSIS	Atomically XORs the given value with the atomic object.

NAME	**OS_AtomicFetchXor**
SYNTAX	`os_atomic_value_t OS_AtomicFetchXor(os_atomic_t volatile *object, os_atomic_value_t operand)`
DESCRIPTION	Atomically performs the boolean XOR operation with the value `operand` and the value of the atomic object at `object` and updates it with the result.
RETURN	The value of the atomic object at `object` before the operation.

3.2.13. OS_AtomicTestAndSetFlag()

NAME	**OS_AtomicTestAndSetFlag**
SYNOPSIS	Atomically sets a flag in the atomic object.
SYNTAX	`os_boolean_t OS_AtomicTestAndSetFlag(os_atomic_t volatile *object, os_atomic_value_t flagSelectionMask)`
DESCRIPTION	Atomically sets the flag selected by `flagSelectionMask` in the atomic object at `object`. The selection mask may have only one bit set.
RETURN	The state of the selected flag before the operation.

3.2.14. OS_AtomicClearFlag()

NAME	**OS_AtomicClearFlag**
SYNOPSIS	Atomically clears a flag in the atomic object.
SYNTAX	`void OS_AtomicClearFlag(os_atomic_t volatile *object, os_atomic_value_t flagSelectionMask)`
DESCRIPTION	Atomically clears the flag selected by `flagSelectionMask` in the atomic object at `object`. The selection mask may have only one bit set.

3.3. Constraints for exclusive areas using `EB_FAST_LOCK`

The following constraints apply when you are inside an exclusive area with the type `EB_FAST_LOCK`.

- You must not call any OS APIs (except atomic functions).
- The execution time budget monitoring is ineffective. Thus, you're strongly advised to minimize the time spent inside an exclusive area.
- The time stamps returned by `OS_GetTimeStamp()` might become inaccurate. Thus, you're strongly advised to minimize the time span of being inside an exclusive area. Please note, you're not allowed to call `OS_GetTimeStamp()` from inside an exclusive area. Hence, this impact becomes evident only afterwards.

The following documents help you to evaluate the implications of using `EB_FAST_LOCK` for exclusive areas further when you face safety goals.

- EB tresos AutoCore OS safety application guide for ASIL-B applications
- EB tresos Safety OS user's guide for TRICORE family
- EB tresos Safety OS safety manual for TRICORE family

The Section 3.4, “Usage restrictions” examines the general constraints mentioned here specifically for TRICORE if there are any.

3.4. Usage restrictions

3.4.1. Atomic functions

The atomic functions on TRICORE do not take caches into account. Hence, it is necessary to put all atomic objects with type `os_atomic_t` into non-cacheable memory or to disable caches completely. This ensures that all threads of execution are able to observe each other's memory accesses.

Please note that the implementation uses specialized instructions (e.g., `cmpswap.w` or `swapmsk.w`) to work on atomic objects. These instructions require that objects of type `os_atomic_t` are naturally aligned.

3.4.2. Hardware specific constraints for `EB_FAST_LOCK`

For the TRICORE family, EB tresos AutoCore OS feature an optimized implementation to ease the overhead incurred by entering and leaving exclusive areas. This optimization is effective only for the processor modes `USER1` and `SUPERVISOR`. Consequently, this optimization doesn't affect program execution in processor mode `USER`. For further information about processor modes, please have a look at the configuration parameters `OsAppCpuMode` and `OsTrusted` of OS applications.

As far as timing is concerned, an exclusive area impairs monitoring of execution time budgets when it's based on a System Timer (STM). Thus, you can no longer rely on this mechanism when your application is inside an exclusive area. It might even become unreliable outside exclusive areas when your application spends too much time inside them.

The good news is, though, that the time stamps returned by `OS_GetTimeStamp()` aren't affected by exclusive areas at all.

Please also note that you must not call any OS API functions from inside exclusive areas.

Bibliography

Index