

Core Overview

Multiprocessor environments can use the mutex core with Avalon[®] interface to coordinate accesses to a shared resource. The mutex core provides a protocol to ensure mutually exclusive ownership of a shared resource.

The mutex core provides a hardware-based atomic test-and-set operation, allowing software in a multiprocessor environment to determine which processor owns the mutex. The mutex core can be used in conjunction with shared memory to implement additional interprocessor coordination features, such as mailboxes and software mutexes.

The mutex core is designed for use in Avalon-based processor systems, such as a Nios[®] II processor system. Altera provides device drivers for the Nios II processor to enable use of the hardware mutex.

The mutex core is SOPC Builder-ready and integrates easily into any SOPC Builder-generated system. This chapter contains the following sections:

- [“Functional Description”](#)
- [“Device and Tools Support”](#) on page 26–2
- [“Instantiating the Core in SOPC Builder”](#) on page 26–2
- [“Software Programming Model”](#) on page 26–2
- [“Mutex API”](#) on page 26–4

Functional Description

The mutex core has a simple Avalon Memory-Mapped (Avalon-MM) slave interface that provides access to two memory-mapped, 32-bit registers. [Table 26–1](#) shows the registers.

Table 26–1. Mutex Core Register Map

Offset	Register Name	R/W	Bit Description		
			31 ... 16	15 ... 1	0
0	mutex	RW	OWNER	VALUE	
1	reset	RW	–	–	RESET

The mutex core has the following basic behavior. This description assumes there are multiple processors accessing a single mutex core, and each processor has a unique identifier (ID).

- When the VALUE field is 0x0000, the mutex is available (i.e., unlocked). Otherwise, the mutex is unavailable (in other words, locked).
- The mutex register is always readable. A processor (or any Avalon-MM master peripheral) can read the mutex register to determine its current state.
- The mutex register is writable only under specific conditions. A write operation changes the mutex register only if one or both of the following conditions is true:
 - The VALUE field of the mutex register is zero.
 - The OWNER field of the mutex register matches the OWNER field in the data to be written.
- A processor attempts to acquire the mutex by writing its ID to the OWNER field, and writing a non-zero value to VALUE. The processor then checks if the acquisition succeeded by verifying the OWNER field.
- After system reset, the RESET bit in the reset register is high. Writing a one to this bit clears it.

Device and Tools Support

The mutex core supports all Altera device families supported by SOPC Builder, and provides device drivers for the Nios II hardware abstraction layer (HAL) system library.

Instantiating the Core in SOPC Builder

Hardware designers use the MegaWizard® Plug-In Manager for the mutex core in SOPC Builder to specify the core's hardware features. The MegaWizard Plug-In Manager provides the following options:

- **Initial Value**—the initial contents of the VALUE field after reset. If the **Initial Value** setting is non-zero, you must also specify **Initial Owner**.
- **Initial Owner**—the initial contents of the OWNER field after reset. When **Initial Owner** is specified, this owner must release the mutex before it can be acquired by another owner.

Software Programming Model

The following sections describe the software programming model for the mutex core, such as the software constructs used to access the hardware. For Nios II processor users, Altera provides routines to access the mutex core hardware. These functions are specific to the mutex core and directly manipulate low-level hardware. The mutex core cannot be accessed via

the HAL API or the ANSI C standard library. In Nios II processor systems, a processor locks the mutex by writing the value of its `cpuid` control register to the OWNER field of the mutex register.

Software Files

Altera provides the following software files accompanying the mutex core:

- **altera_avalon_mutex_regs.h**—this file defines the core’s register map, providing symbolic constants to access the low-level hardware.
- **altera_avalon_mutex.h**—this file defines data structures and functions to access the mutex core hardware.
- **altera_avalon_mutex.c**—this file contains the implementations of the functions to access the mutex core

Hardware Mutex

This section describes the low-level software constructs for manipulating the mutex core hardware.

The file **altera_avalon_mutex.h** declares a structure `alt_mutex_dev` that represents an instance of a mutex device. It also declares functions for accessing the mutex hardware structure, listed in [Table 26–2](#).

Function Name	Description
<code>altera_avalon_mutex_open()</code>	Claims a handle to a mutex, enabling all the other functions to access the mutex core.
<code>altera_avalon_mutex_trylock()</code>	Tries to lock the mutex. Returns immediately if it fails to lock the mutex.
<code>altera_avalon_mutex_lock()</code>	Locks the mutex. Will not return until it has successfully claimed the mutex.
<code>altera_avalon_mutex_unlock()</code>	Unlocks the mutex.
<code>altera_avalon_mutex_is_mine()</code>	Determines if this CPU owns the mutex.
<code>altera_avalon_mutex_first_lock()</code>	Tests whether the mutex has been released since reset.

These routines coordinate access to the software mutex structure using a hardware mutex core. For a complete description of each function, see section [“Mutex API” on page 26–4](#).

The code shown in [Example 26–1](#) demonstrates opening a mutex device handle and locking a mutex.

Example 26–1. Opening and Locking a mutex

```
#include <altera_avalon_mutex.h>

/* get the mutex device handle */
alt_mutex_dev* mutex = altera_avalon_mutex_open( "/dev/mutex" );

/* acquire the mutex, setting the value to one */
altera_avalon_mutex_lock( mutex, 1 );

/*
 * Access a shared resource here.
 */

/* release the lock */
altera_avalon_mutex_unlock( mutex );
```

Mutex API

This section describes the application programming interface (API) for the mutex core.

altera_avalon_mutex_is_mine()

Prototype: `int altera_avalon_mutex_is_mine(alt_mutex_dev* dev)`

Thread-safe: Yes.

Available from ISR: No.

Include: `<altera_avalon_mutex.h>`

Parameters: `dev`—the mutex device to test.

Returns: Returns non zero if the mutex is owned by this CPU.

Description: `altera_avalon_mutex_is_mine()` determines if this CPU owns the mutex.

altera_avalon_mutex_first_lock()

Prototype: `int altera_avalon_mutex_first_lock(alt_mutex_dev* dev)`

Thread-safe: Yes.

Available from ISR: No.

Include: `<altera_avalon_mutex.h>`

Parameters: `dev`—the mutex device to test.

Returns: Returns 1 if this mutex has not been released since reset, otherwise returns 0.

Description: `altera_avalon_mutex_first_lock()` determines whether this mutex has been released since reset.

altera_avalon_mutex_lock()

Prototype: `void altera_avalon_mutex_lock(alt_mutex_dev* dev, alt_u32 value)`

Thread-safe: Yes.

Available from ISR: No.

Include: `<altera_avalon_mutex.h>`

Parameters: `dev`—the mutex device to acquire.
`value`—the new value to write to the mutex.

Returns: —

Description: `altera_avalon_mutex_lock()` is a blocking routine that acquires a hardware mutex, and at the same time, loads the mutex with the `value` parameter.

altera_avalon_mutex_open()

Prototype: `alt_mutex_dev* alt_hardware_mutex_open(const char* name)`

Thread-safe: Yes.

Available from ISR: No.

Include: `<altera_avalon_mutex.h>`

Parameters: `name`—the name of the mutex device to open.

Returns: A pointer to the mutex device structure associated with the supplied name, or NULL if no corresponding mutex device structure was found.

Description: `altera_avalon_mutex_open()` retrieves a pointer to a hardware mutex device structure.

altera_avalon_mutex_trylock()

Prototype: `int altera_avalon_mutex_trylock(alt_mutex_dev* dev, alt_u32 value)`

Thread-safe: Yes.

Available from ISR: No.

Include: `<altera_avalon_mutex.h>`

Parameters: `dev`—the mutex device to lock.
`value`—the new value to write to the mutex.

Returns: Zero if the mutex was successfully locked, or non zero if the mutex was not locked.

Description: `altera_avalon_mutex_trylock()` tries once to lock the hardware mutex, and returns immediately.

altera_avalon_mutex_unlock()

Prototype: `void altera_avalon_mutex_unlock(alt_mutex_dev* dev)`

Thread-safe: Yes.

Available from ISR: No.

Include: `<altera_avalon_mutex.h>`

Parameters: `dev`—the mutex device to unlock.

Returns: -

Description: `altera_avalon_mutex_unlock()` releases a hardware mutex device. Upon release, the value stored in the mutex is set to zero. If the caller does not hold the mutex, the behavior of this function is undefined.

Document Revision History

Table 26–3 shows the revision history for this chapter.

Date and Document Version	Changes Made	Summary of Changes
May 2008 v8.0.0	No change from previous release.	—
October 2007 v7.2.0	No change from previous release.	—
May 2007 v7.1.0	Added table of contents to Overview section.	—
March 2007 v7.0.0	No change from previous release.	—
November 2006 v6.1.0	<ul style="list-style-type: none"> ● Updated Avalon terminology because of changes to Avalon technologies ● Changed old “Avalon switch fabric” term to “system interconnect fabric” ● Changed old “Avalon interface” terms to “Avalon Memory-Mapped interface” 	For the 6.1 release, Altera released the Avalon Streaming interface, which necessitated some rephrasing of existing Avalon terminology.
May 2006 v6.0.0	No change from previous release.	—
October 2005 v5.1.0	No change from previous release.	—
May 2005 v5.0.0	No change from previous release. Previously in the Nios II Processor Reference Handbook.	—
December 2004 v1.0	Initial release.	—

