lib_locks: Locks for concurrency

Publication Date: 2026/2/10
Document Number: XM-006390-UG v2.4.0

XMOS

IN THIS DOCUMENT

1 Introduction

This library provides access to hardware and software locks for use in concurrent C programs. In general it is not safe to use these to marshall within XC due to the assumptions XC makes about safe concurrent data access.

Three types of locks are provided. The first two support locking in the local tile scope only. Hardware locks are fast and power efficient but there are a limited number per tile. Software locks are slower but you can use an unlimited number of them.

In addition a single Intertile lock is available on `xcore.ai` devices only. This uses a shared peripheral register on the `xcore.ai` device and is suitable for protecting resources shared across tiles.

Examples include access to `xcore.ai` peripherals or locking printing from both tiles. It is slower than a software lock and so should only be used for device-wide locking requirements.

`lib_locks` is intended to be used with the XCommon CMake , the *XMOS* application build and dependency management system.

2 Basic use

2.1 Declaration & allocation

Before using a software or hardware lock, it must first be declared.

Software based locks should be initialised to a specific value:

```
swlock_t swlock = SWLOCK_INITIAL_VALUE;
```

Hardware locks relate to a physical resource in the device and so need to be properly allocated:

```
hwlock_t hwlock;

hwlock = hwlock_alloc();
```

Locks are typically used to protect critical sections of code when multiple threads are involved, so they are often declared globally for shared access.

The Intertile lock uses a fixed peripheral register which is initialised on power up and therefore does not require initialisation before use.

> **Warning**
>
> The Intertile lock uses a MIPI D-PHY register and so cannot be used at the same time as the MIPI D-PHY peripheral.

2.2 Acquisition and release

Hardware and software based locks use a similar API for acquisition and release. For software based locks the following is used:

```
swlock_acquire(&swlock);
// Perform critical code section..
swlock_release(&swlock);
```

Similarly, for hardware based locks:

```
hwlock_acquire(&hwlock);
// Perform critical code section..
hwlock_release(&hwlock);
```

The Intertile lock takes no arguments and only one lock per xcore.ai is available:

```
intertile_lock_acquire();
// Perform critical code section..
intertile_lock_release();
```

In all cases these are blocking calls.

2.3 Freeing

Once an application no longer requires a hardware lock it should be freed to allow the underlying hardware resource to be reused:

```
hwlock_free(hwlock);
```

Software and Intertile locks do not need to be freed.

3 Example application

An example is provided in *examples/app_locks_example* that demonstrates basic lock allocation, freeing, acquiring and releasing.

4 Hardware lock API

typedef unsigned **hwlock_t**

This type represents a hardware lock.

inline *hwlock_t* **hwlock_alloc**(void)

Allocate a hardware lock.

This function will allocate a new hardware lock from the pool of hardware locks available on the xCORE. The hardware has a limited number of hardware locks (for example, current L and S series devices have 4 locks per tile).

Returns

the allocated lock if allocation is successful or the value `HWLOCK_NOT_ALLOCATED` if not.

inline void **hwlock_free**(*hwlock_t* lock)

Free a hardware lock.

This function frees a given hardware lock and returns it to the hardware pool to be reallocated elsewhere.

Parameters

- **lock** – the hardware lock to be freed. If this is an invalid lock id or not an currently allocated lock then the function will trap.

inline void **hwlock_acquire**(*hwlock_t* lock)

Acquire a hardware lock.

This function acquires a lock for the current logical core. If another core holds the lock the function will pause until the lock is released.

Parameters

- **lock** – the hardware lock to acquire

inline void **hwlock_release**(*hwlock_t* lock)

Release a hardware lock.

This function releases a lock from the current logical core. The lock should have been previously claimed by hwlock_acquire().

Parameters

- **lock** – the hardware lock to release

5 Software lock API

typedef unsigned **swlock_t**

Type that represents a software lock

SWLOCK_INITIAL_VALUE

This define should be used to initialize a software lock e.g.

```
swlock_t my_lock = SWLOCK_INITIAL_VALUE;
```

If you intialize this way there is no need to call swlock_init().

void **swlock_init**(REFERENCE_PARAM(*swlock_t*, lock))

Initialize a software lock.

This function will initialize a software lock for use. Note that unlike hardware locks, there is no need to allocate or free a software lock from a limited pool.

int **swlock_try_acquire**(REFERENCE_PARAM(*swlock_t*, lock))

Try and acquire a software lock.

This function tries to acquire a lock for the current logical core. If another core holds the lock then the function will fail and return.

Parameters

- **lock** – the software lock to acquire.

Returns

a value that is equal to **SWLOCK_NOT_ACQUIRED** if the attempt fails. Any other value indicates that the acquisition has succeeded.

void **swlock_acquire**(REFERENCE_PARAM(*swlock_t*, lock))

Acquire a software lock.

This function acquires a lock for the current logical core. If another core holds the lock then the function will wait until it becomes available.

Parameters

- **lock** – the software lock to acquire.

void **swlock_release**(REFERENCE_PARAM(*swlock_t*, lock))

Release a software lock.

This function releases a previously acquired software lock for other cores to use.

Parameters

- **lock** – the software lock to release.

6 Intertile lock API

void **intertile_lock_acquire**(void)

Acquire a tile lock.

This function acquires the tile lock for the current logical core. If another logical core holds the lock then the function will wait until it becomes available.

void **intertile_lock_release**(void)

Release a tile lock.

This function releases a previously acquired tile lock for other cores to use. Note: the underlying register write operation (performed by write_sswitch_reg()) temporarily allocates a channel end. If many cores on the same tile wish to use this function at the same time, it may be sensible to use either a local tile lock to guard access to this function to avoid running out of chanend resources at runtime.

This must only be used AFTER a tile lock acquisition.