Hardware Spinlock Framework

Introduction

Hardware spinlock modules provide hardware assistance for synchronization and mutual exclusion between heterogeneous processors and those not operating under a single, shared operating system.

For example, OMAP4 has dual Cortex-A9, dual Cortex-M3 and a C64x+ DSP, each of which is running a different Operating System (the master, A9, is usually running Linux and the slave processors, the M3 and the DSP, are running some flavor of RTOS).

A generic hwspinlock framework allows platform-independent drivers to use the hwspinlock device in order to access data structures that are shared between remote processors, that otherwise have no alternative mechanism to accomplish synchronization and mutual exclusion operations.

This is necessary, for example, for Inter-processor communications: on OMAP4, cpu-intensive multimedia tasks are offloaded by the host to the remote M3 and/or C64x+ slave processors (by an IPC subsystem called Syslink).

To achieve fast message-based communications, a minimal kernel support is needed to deliver messages arriving from a remote processor to the appropriate user process.

This communication is based on simple data structures that is shared between the remote processors, and access to it is synchronized using the hwspinlock module (remote processor directly places new messages in this shared data structure).

A common hwspinlock interface makes it possible to have generic, platform- independent, drivers.

User API

```
struct hwspinlock *hwspin_lock_request(void);
```

Dynamically assign an hwspinlock and return its address, or NULL in case an unused hwspinlock isn't available. Users of this API will usually want to communicate the lock's id to the remote core before it can be used to achieve synchronization.

Should be called from a process context (might sleep).

```
struct hwspinlock *hwspin_lock_request_specific(unsigned int id);
```

Assign a specific hwspinlock id and return its address, or NULL if that hwspinlock is already in use. Usually board code will be calling this function in order to reserve specific hwspinlock ids for predefined purposes.

Should be called from a process context (might sleep).

```
int of_hwspin_lock_get_id(struct device_node *np, int index);
```

Retrieve the global lock id for an OF phandle-based specific lock. This function provides a means for DT users of a hwspinlock module to get the global lock id of a specific hwspinlock, so that it can be requested using the normal hwspin_lock_request_specific() API.

The function returns a lock id number on success, -EPROBE_DEFER if the hwspinlock device is not yet registered with the core, or other error values.

Should be called from a process context (might sleep).

```
int hwspin_lock_free(struct hwspinlock *hwlock);
```

Free a previously-assigned hwspinlock; returns 0 on success, or an appropriate error code on failure (e.g. -EINVAL if the hwspinlock is already free).

Should be called from a process context (might sleep).

```
int hwspin_lock_timeout(struct hwspinlock *hwlock, unsigned int timeout);
```

Lock a previously-assigned hwspinlock with a timeout limit (specified in msecs). If the hwspinlock is already taken, the function will busy loop waiting for it to be released, but give up when the timeout elapses. Upon a successful return from this function, preemption is disabled so the caller must not sleep, and is advised to release the hwspinlock as soon as possible, in order to minimize remote cores polling on the hardware interconnect.

Returns 0 when successful and an appropriate error code otherwise (most notably -ETIMEDOUT if the hwspinlock is still busy after timeout msecs). The function will never sleep.

```
int hwspin_lock_timeout_irq(struct hwspinlock *hwlock, unsigned int timeout);
```

Lock a previously-assigned hwspinlock with a timeout limit (specified in msecs). If the hwspinlock is already taken, the function will busy loop waiting for it to be released, but give up when the timeout elapses. Upon a successful return from this function, preemption and the local interrupts are disabled, so the caller must not sleep, and is advised to release the hwspinlock as soon as possible.

Returns 0 when successful and an appropriate error code otherwise (most notably -ETIMEDOUT if the hwspinlock is still busy after timeout msecs). The function will never sleep.

```
int hwspin_lock_timeout_irqsave(struct hwspinlock *hwlock, unsigned int to,
                                unsigned long *flags);
```

Lock a previously-assigned hwspinlock with a timeout limit (specified in msecs). If the hwspinlock is already taken, the function will busy loop waiting for it to be released, but give up when the timeout elapses. Upon a successful return from this function, preemption is disabled, local interrupts are disabled and their previous state is saved at the given flags placeholder. The caller must not sleep, and is advised to release the hwspinlock as soon as possible.

Returns 0 when successful and an appropriate error code otherwise (most notably -ETIMEDOUT if the hwspinlock is still busy after timeout msecs).

The function will never sleep.

```
int hwspin_lock_timeout_raw(struct hwspinlock *hwlock, unsigned int timeout);
```

Lock a previously-assigned hwspinlock with a timeout limit (specified in msecs). If the hwspinlock is already taken, the function will busy loop waiting for it to be released, but give up when the timeout elapses.

Caution: User must protect the routine of getting hardware lock with mutex or spinlock to avoid dead-lock, that will let user can do some time-consuming or sleepable operations under the hardware lock.

Returns 0 when successful and an appropriate error code otherwise (most notably -ETIMEDOUT if the hwspinlock is still busy after timeout msecs).

The function will never sleep.

```
int hwspin_lock_timeout_in_atomic(struct hwspinlock *hwlock, unsigned int to);
```

Lock a previously-assigned hwspinlock with a timeout limit (specified in msecs). If the hwspinlock is already taken, the function will busy loop waiting for it to be released, but give up when the timeout elapses.

This function shall be called only from an atomic context and the timeout value shall not exceed a few msecs.

Returns 0 when successful and an appropriate error code otherwise (most notably -ETIMEDOUT if the hwspinlock is still busy after timeout msecs).

The function will never sleep.

```
int hwspin_trylock(struct hwspinlock *hwlock);
```

Attempt to lock a previously-assigned hwspinlock, but immediately fail if it is already taken.

Upon a successful return from this function, preemption is disabled so caller must not sleep, and is advised to release the hwspinlock as soon as possible, in order to minimize remote cores polling on the hardware interconnect.

Returns 0 on success and an appropriate error code otherwise (most notably -EBUSY if the hwspinlock was already taken). The function will never sleep.

```
int hwspin_trylock_irq(struct hwspinlock *hwlock);
```

Attempt to lock a previously-assigned hwspinlock, but immediately fail if it is already taken.

Upon a successful return from this function, preemption and the local interrupts are disabled so caller must not sleep, and is advised to release the hwspinlock as soon as possible.

Returns 0 on success and an appropriate error code otherwise (most notably -EBUSY if the hwspinlock was already taken).

The function will never sleep.

```
int hwspin_trylock_irqsave(struct hwspinlock *hwlock, unsigned long *flags);
```

Attempt to lock a previously-assigned hwspinlock, but immediately fail if it is already taken.

Upon a successful return from this function, preemption is disabled, the local interrupts are disabled and their previous state is saved at the given flags placeholder. The caller must not sleep, and is advised to release the hwspinlock as soon as possible.

Returns 0 on success and an appropriate error code otherwise (most notably -EBUSY if the hwspinlock was already taken). The function will never sleep.

```
int hwspin_trylock_raw(struct hwspinlock *hwlock);
```

Attempt to lock a previously-assigned hwspinlock, but immediately fail if it is already taken.

Caution: User must protect the routine of getting hardware lock with mutex or spinlock to avoid dead-lock, that will let user can do some time-consuming or sleepable operations under the hardware lock.

Returns 0 on success and an appropriate error code otherwise (most notably -EBUSY if the hwspinlock was already taken). The function will never sleep.

```
int hwspin_trylock_in_atomic(struct hwspinlock *hwlock);
```

Attempt to lock a previously-assigned hwspinlock, but immediately fail if it is already taken.

This function shall be called only from an atomic context.

Returns 0 on success and an appropriate error code otherwise (most notably -EBUSY if the hwspinlock was already taken). The function will never sleep.

```
void hwspin_unlock(struct hwspinlock *hwlock);
```

Unlock a previously-locked hwspinlock. Always succeed, and can be called from any context (the function never sleeps).

> **Note**
>
> code should **never** unlock an hwspinlock which is already unlocked (there is no protection against this).

```
void hwspin_unlock_irq(struct hwspinlock *hwlock);
```

Unlock a previously-locked hwspinlock and enable local interrupts. The caller should **never** unlock an hwspinlock which is already unlocked.

Doing so is considered a bug (there is no protection against this). Upon a successful return from this function, preemption and local interrupts are enabled. This function will never sleep.

```
void
hwspin_unlock_irqrestore(struct hwspinlock *hwlock, unsigned long *flags);
```

Unlock a previously-locked hwspinlock.

The caller should **never** unlock an hwspinlock which is already unlocked. Doing so is considered a bug (there is no protection against this). Upon a successful return from this function, preemption is reenabled, and the state of the local interrupts is restored to the state saved at the given flags. This function will never sleep.

```
void hwspin_unlock_raw(struct hwspinlock *hwlock);
```

Unlock a previously-locked hwspinlock.

The caller should **never** unlock an hwspinlock which is already unlocked. Doing so is considered a bug (there is no protection against this). This function will never sleep.

```
void hwspin_unlock_in_atomic(struct hwspinlock *hwlock);
```

Unlock a previously-locked hwspinlock.

The caller should **never** unlock an hwspinlock which is already unlocked. Doing so is considered a bug (there is no protection against this). This function will never sleep.

```
int hwspin_lock_get_id(struct hwspinlock *hwlock);
```

Retrieve id number of a given hwspinlock. This is needed when an hwspinlock is dynamically assigned: before it can be used to achieve mutual exclusion with a remote cpu, the id number should be communicated to the remote task with which we want to synchronize.

Returns the hwspinlock id number, or -EINVAL if hwlock is null.

Typical usage

```
#include <linux/hwspinlock.h>
#include <linux/err.h>

int hwspinlock_example1(void)
{
        struct hwspinlock *hwlock;
        int ret;

        /* dynamically assign a hwspinlock */
        hwlock = hwspin_lock_request();
        if (!hwlock)
                ...

        id = hwspin_lock_get_id(hwlock);
        /* probably need to communicate id to a remote processor now */

        /* take the lock, spin for 1 sec if it's already taken */
        ret = hwspin_lock_timeout(hwlock, 1000);
        if (ret)
                ...
        /*
        * we took the lock, do our thing now, but do NOT sleep
        */

        /* release the lock */
        hwspin_unlock(hwlock);

        /* free the lock */
        ret = hwspin_lock_free(hwlock);
        if (ret)
                ...

        return ret;
}

int hwspinlock_example2(void)
{
        struct hwspinlock *hwlock;
        int ret;

        /*
        * assign a specific hwspinlock id - this should be called early
        * by board init code.
        */
        hwlock = hwspin_lock_request_specific(PREDEFINED_LOCK_ID);
        if (!hwlock)
                ...

        /* try to take it, but don't spin on it */
        ret = hwspin_trylock(hwlock);
        if (!ret) {
                pr_info("lock is already taken\n");
                return -EBUSY;
        }

        /*
        * we took the lock, do our thing now, but do NOT sleep
        */

        /* release the lock */
        hwspin_unlock(hwlock);

        /* free the lock */
        ret = hwspin_lock_free(hwlock);
        if (ret)
                ...

        return ret;
}
```

API for implementors

```
int hwspin_lock_register(struct hwspinlock_device *bank, struct device *dev,
              const struct hwspinlock_ops *ops, int base_id, int num_locks);
```

To be called from the underlying platform-specific implementation, in order to register a new hwspinlock device (which is usually a bank of numerous locks). Should be called from a process context (this function might sleep).

Returns 0 on success, or appropriate error code on failure.

```
int hwspin_lock_unregister(struct hwspinlock_device *bank);
```

To be called from the underlying vendor-specific implementation, in order to unregister an hwspinlock device (which is usually a bank of numerous locks).

Should be called from a process context (this function might sleep).

Returns the address of hwspinlock on success, or NULL on error (e.g. if the hwspinlock is still in use).

Important structs

struct hwspinlock_device is a device which usually contains a bank of hardware locks. It is registered by the underlying hwspinlock implementation using the hwspin_lock_register() API.

```
/**
* struct hwspinlock_device - a device which usually spans numerous hwspinlocks
* @dev: underlying device, will be used to invoke runtime PM api
* @ops: platform-specific hwspinlock handlers
* @base_id: id index of the first lock in this device
```

```
 * @num_locks: number of locks in this device
 * @lock: dynamically allocated array of 'struct hwspinlock'
 */
struct hwspinlock_device {
        struct device *dev;
        const struct hwspinlock_ops *ops;
        int base_id;
        int num_locks;
        struct hwspinlock lock[0];
};
```

struct hwspinlock_device contains an array of hwspinlock structs, each of which represents a single hardware lock:

```
/**
 * struct hwspinlock - this struct represents a single hwspinlock instance
 * @bank: the hwspinlock_device structure which owns this lock
 * @lock: initialized and used by hwspinlock core
 * @priv: private data, owned by the underlying platform-specific hwspinlock drv
 */
struct hwspinlock {
        struct hwspinlock_device *bank;
        spinlock_t lock;
        void *priv;
};
```

When registering a bank of locks, the hwspinlock driver only needs to set the priv members of the locks. The rest of the members are set and initialized by the hwspinlock core itself.

Implementation callbacks

There are three possible callbacks defined in 'struct hwspinlock_ops':

```
struct hwspinlock_ops {
        int (*trylock)(struct hwspinlock *lock);
        void (*unlock)(struct hwspinlock *lock);
        void (*relax)(struct hwspinlock *lock);
};
```

The first two callbacks are mandatory:

The ->trylock() callback should make a single attempt to take the lock, and return 0 on failure and 1 on success. This callback may **not** sleep.

The ->unlock() callback releases the lock. It always succeed, and it, too, may **not** sleep.

The ->relax() callback is optional. It is called by hwspinlock core while spinning on a lock, and can be used by the underlying implementation to force a delay between two successive invocations of ->trylock(). It may **not** sleep.