NAME

ddi_add_softintr, ddi_get_soft_iblock_cookie, ddi_remove_softintr, ddi_trigger_softintr – software interrupt handling routines

SYNOPSIS

#include <sys/types.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

int ddi_get_soft_iblock_cookie(dev_info_t **dip*, **int** *preference*,
ddi_iblock_cookie_t **iblock_cookiep***);**

int ddi_add_softintr(dev_info_t **dip*, **int** *preference*, **ddi_softintr_t** **idp*,
ddi_iblock_cookie_t **iblock_cookiep*, **ddi_idevice_cookie_t** **idevice_cookiep*,
u_int(**int_handler*)**(caddr_t** *int_handler_arg*), **caddr_t** *int_handler_arg***);**

void ddi_remove_softintr(ddi_softintr_t *id***);**

void ddi_trigger_softintr(ddi_softintr_t *id***);**

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

ARGUMENTS

ddi_get_soft_iblock_cookie( )

dip	Pointer to a **dev_info** structure.
preference	The type of soft interrupt to retrieve the cookie for.
iblock_cookiep	Pointer to a location to store the interrupt block cookie.

ddi_add_softintr( )

dip	Pointer to **dev_info** structure.
preference	A hint value describing the type of soft interrupt to generate.
idp	Pointer to a soft interrupt identifier where a returned soft interrupt identifier is stored.
iblock_cookiep	Optional pointer to an interrupt block cookie where a returned interrupt block cookie is stored.
idevice_cookiep	Optional pointer to an interrupt device cookie where a returned interrupt device cookie is stored (not used).
int_handler	Pointer to interrupt handler.
int_handler_arg	Argument for interrupt handler.

ddi_remove_softintr( )

id	The identifier specifying which soft interrupt handler to remove.

ddi_trigger_softintr( )

id	The identifier specifying which soft interrupt to trigger and which soft interrupt handler will be called.

DESCRIPTION

ddi_get_soft_iblock_cookie( )

ddi_get_soft_iblock_cookie( ) retrieves the interrupt block cookie associated with a particular soft interrupt preference level. This routine should be called before **ddi_add_softintr( )** to retrieve the interrupt block cookie needed to initialize locks (

mutex(9F), **rwlock**(9F)) used by the software interrupt routine. *preference* determines which type of soft interrupt to retrieve the cookie for. The possible values for *preference* are:

DDI_SOFTINT_LOW	**Low priority soft interrupt**
DDI_SOFTINT_MED	**Medium priority soft interrupt**
DDI_SOFTINT_HIGH	**High priority soft interrupt**

On a successful return, *iblock_cookiep* contains information needed for initializing locks associated with this soft interrupt (see **mutex_init**(9F) and **rw_init**(9F)). The driver can then initialize mutexes acquired by the interrupt routine before calling **ddi_add_softintr( )** which prevents a possible race condition where the driver's soft interrupt handler is called immediately *after* the driver has called **ddi_add_softintr( )** but *before* the driver has initialized the mutexes. This can happen when a soft interrupt for a different device occurs on the same soft interrupt priority level. If the soft interrupt routine acquires the mutex before it has been initialized, undefined behavior may result.

ddi_add_softintr( )

ddi_add_softintr( ) adds a soft interrupt to the system. The user specified hint *preference* identifies three suggested levels for the system to attempt to allocate the soft interrupt priority at. The value for *preference* should be the same as that used in the corresponding call to **ddi_get_soft_iblock_cookie( )**. Refer to the description of **ddi_get_soft_iblock_cookie( )** above.

The value returned in the location pointed at by *idp* is the soft interrupt identifier. This value is used in later calls to **ddi_remove_softintr( )** and **ddi_trigger_softintr( )** to identify the soft interrupt and the soft interrupt handler.

The value returned in the location pointed at by *iblock_cookiep* is an interrupt block cookie which contains information used for initializing mutexes associated with this soft interrupt (see **mutex_init**(9F) and **rw_init**(9F)). Note that the interrupt block cookie is normally obtained using **ddi_get_soft_iblock_cookie( )** to avoid the race conditions described above (refer to the description of **ddi_get_soft_iblock_cookie( )** above). For this reason, *iblock_cookiep* is no longer useful and should be set to **NULL**.

idevice_cookiep is not used and should be set to **NULL**.

The routine *int_handler*, with its argument *int_handler_arg*, is called upon receipt of a software interrupt. Software interrupt handlers must not assume that they have work to do when they run, since (like hardware interrupt handlers) they may run because a soft interrupt occurred for some other reason. For example, another driver may have triggered a soft interrupt at the same level. For this reason, before triggering the soft interrupt, the driver must indicate to its soft interrupt handler that it should do work. This is usually done by setting a flag in the state structure. The routine *int_handler* checks this flag, reachable through *int_handler_arg*, to determine if it should claim the interrupt and do its work.

The interrupt handler must return **DDI_INTR_CLAIMED** if the interrupt was claimed, **DDI_INTR_UNCLAIMED** otherwise.

If successful, **ddi_add_softintr( )** will return **DDI_SUCCESS**; if the interrupt information cannot be found, it will return **DDI_FAILURE**.

ddi_remove_softintr( )

ddi_remove_softintr( ) removes a soft interrupt from the system. The soft interrupt identifier *id*, which was returned from a call to **ddi_add_softintr( )**, is used to determine which soft interrupt and which soft interrupt handler to remove. Drivers must remove any soft interrupt handlers before allowing the system to unload the driver.

ddi_trigger_softintr( )

ddi_trigger_softintr( ) triggers a soft interrupt. The soft interrupt identifier *id* is used to determine which soft interrupt to trigger. This function is used by device drivers when they wish to trigger a soft interrupt which has been set up using **ddi_add_softintr( )**.

RETURN VALUES

ddi_add_softintr( ) and **ddi_get_soft_iblock_cookie( )** return:

DDI_SUCCESS	on success
DDI_FAILURE	on failure

CONTEXT

These functions can be called from user or kernel context. **ddi_trigger_softintr( )** may be called from high-level interrupt context as well.

EXAMPLES

In the following example, the device uses high level interrupts. High level interrupts are those that interrupt at the level of the scheduler and above. High level interrupts must be handled without using system services that manipulate thread or process states, because these interrupts are not blocked by the scheduler. In addition, high level interrupt handlers must take care to do a minimum of work because they are not preemptable. See **ddi_intr_hilevel**(9F).

In the example, the high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt handler. If the soft interrupt handler is not currently running, the high-level interrupt routine triggers a soft interrupt so the soft interrupt handler can process the data. Once running, the soft interrupt handler processes all the enqueued data before returning.

The state structure contains two mutexes. The high-level mutex is used to protect data shared between the high-level interrupt handler and the soft interrupt handler. The low-level mutex is used to protect the rest of the driver from the soft interrupt handler.

```
struct xxstate {
        . . .
        ddi_softintr_t          id;
        ddi_iblock_cookie_t     high_iblock_cookie;
        kmutex_t                high_mutex;
        ddi_iblock_cookie_t     low_iblock_cookie;
        kmutex_t                low_mutex;
        int                     softint_running;
        . . .
};
```

```
struct xxstate *xsp;
static u_int xxsoftintr(caddr_t);
static u_int xxhighintr(caddr_t);
. . .
```

The following code fragment would usually appear in the driver's **attach**(9E) routine. **ddi_add_intr**(9F) is used to add the high-level interrupt handler and **ddi_add_softintr( )** is used to add the low-level interrupt routine.

```
static u_int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        struct xxstate *xsp;
        . . .
        /* get high-level iblock cookie */
        if (ddi_get_iblock_cookie(dip, inumber,
                &xsp->high_iblock_cookie) != DDI_SUCCESS)  {
                        /* clean up */
                        return (DDI_FAILURE); /* fail attach */
        }

        /* initialize high-level mutex */
        mutex_init(&xsp->high_mutex, "xx high mutex", MUTEX_DRIVER,
                (void *)xsp->high_iblock_cookie);

        /* add high-level routine - xxhighintr() */
        if (ddi_add_intr(dip, inumber, NULL, NULL,
                xxhighintr, (caddr_t) xsp) != DDI_SUCCESS)  {
                        /* cleanup */
                        return (DDI_FAILURE); /* fail attach */
        }

        /* get soft iblock cookie */
        if (ddi_get_soft_iblock_cookie(dip, DDI_SOFTINT_MED,
                &xsp->low_iblock_cookie) != DDI_SUCCESS)  {
                        /* clean up */
                        return (DDI_FAILURE); /* fail attach */
        }

        /* initialize low-level mutex */
        mutex_init(&xsp->low_mutex, "xx low mutex", MUTEX_DRIVER,
                (void *)xsp->low_iblock_cookie);

        /* add low level routine - xxsoftintr() */
        if ( ddi_add_softintr(dip, DDI_SOFTINT_MED, &xsp->id,
                NULL, NULL, xxsoftintr, (caddr_t) xsp) != DDI_SUCCESS) {
                        /* cleanup */
```

```
                return (DDI_FAILURE);  /* fail attach */
        }

        . . .
}
```

The next code fragment represents the high-level interrupt routine. The high-level interrupt routine minimally services the device, and enqueues the data for later processing by the soft interrupt routine. If the soft interrupt routine is not already running, **ddi_trigger_softintr( )** is called to start the routine. The soft interrupt routine will run until there is no more data on the queue.

```
static u_int
xxhighintr(caddr_t arg)
{
        struct xxstate *xsp = (struct xxstate *) arg;
        int need_softint;
        . . .
        mutex_enter(&xsp->high_mutex);

        /*
         * Verify this device generated the interrupt
         * and disable the device interrupt.
         * Enqueue data for xxsoftintr() processing.
         */

        /* is xxsoftintr() already running ? */
        if (xsp->softint_running)
                need_softint = 0;
        else
                need_softint = 1;
        mutex_exit(&xsp->high_mutex);

        /* read-only access to xsp->id, no mutex needed */
        if (need_softint)
                ddi_trigger_softintr(xsp->id);
        . . .
        return (DDI_INTR_CLAIMED);
}

static u_int
xxsoftintr(caddr_t arg)
{
        struct xxstate *xsp = (struct xxstate *) arg;
        . . .
        mutex_enter(&xsp->low_mutex);
        mutex_enter(&xsp->high_mutex);
```

```
        /* verify there is work to do */
        if (work queue empty || xsp->softint_running ) {
              mutex_exit(&xsp->high_mutex);
              mutex_exit(&xsp->low_mutex);
              return (DDI_INTR_UNCLAIMED);
        }

        xsp->softint_running = 1;

        while ( data on queue ) {
              ASSERT(mutex_owned(&xsp->high_mutex));

              /* de-queue data */

              mutex_exit(&xsp->high_mutex);

              /* Process data on queue */

              mutex_enter(&xsp->high_mutex);
        }

        xsp->softint_running = 0;
        mutex_exit(&xsp->high_mutex);
        mutex_exit(&xsp->low_mutex);

        return (DDI_INTR_CLAIMED);
  }
```

SEE ALSO

ddi_add_intr(9F), **ddi_intr_hilevel**(9F), **ddi_remove_intr**(9F), **mutex_init**(9F)

Writing Device Drivers

NOTES

ddi_add_softintr( ) may not be used to add the same software interrupt handler more than once. This is true even if a different value is used for *int_handler_arg* in each of the calls to **ddi_add_softintr**( ). Instead, the argument passed to the interrupt handler should indicate what service(s) the interrupt handler should perform. For example, the argument could be a pointer to the device's soft state structure, which could contain a 'which_service' field that the handler examines. The driver must set this field to the appropriate value before calling **ddi_trigger_softintr()**.