

## NAME

**freebsd\_device\_driver** – How to write a FreeBSD Device Driver

## DRIVER DECLARATION

```
#include <sys/param.h>
#include <sys/bus.h>

static device_method_t foo_methods[] = {
    DEVMETHOD(device_probe,          foo_probe),
    DEVMETHOD(device_attach,        foo_attach),
    DEVMETHOD(device_detach,        foo_detach),

    DEVMETHOD(bar_baz,              foo_baz),

    { 0, 0 }
};

static driver_t foo_driver = {
    "foo",
    foo_methods,
    sizeof(struct foo_softc),
};

static devclass_t foo_devclass;

/* Module Definitions */
DRIVER_MODULE(foo, bar, foo_driver, foo_devclass, 0, 0);
MODULE_VERSION(foo, 1);
```

The supported device interfaces:

device_identify	device_probe	device_attach	device_detach
device_shutdown	device_quiesce	device_suspend	device_resume

The support bus interfaces:

bus_read_ivar	bus_write_ivar	bus_print_child	bus_add_child
bus_child_detached	bus_child_present	bus_child_pnpinfo_str	bus_child_location_str
bus_driver_added	bus_setup_intr	bus_tearardown_intr	bus_config_intr
bus_alloc_resource	bus_activate_resource	bus_deactivate_resource	bus_release_resource
bus_set_resource	bus_get_resource	bus_delete_resource	bus_get_resource_list
bus_probe_nomatch			

The method `device_probe` shall return either a positive error, or `BUS_PROBE_SPECIFIC` for an exact match, or a negative weight. Some values that can be used are:

<code>BUS_PROBE_SPECIFIC</code>	<code>BUS_PROBE_VENDOR</code>	<code>BUS_PROBE_DEFAULT</code>
<code>BUS_PROBE_LOW_PRIORITY</code>	<code>BUS_PROBE_GENERIC</code>	<code>BUS_PROBE_HOOVER</code>

See `_if.m` files for other bus specifications. For additional information see `driver(9)`.

## RESOURCE ALLOCATION

```
#include <sys/param.h>
#include <sys/bus.h>

#include <machine/bus.h>
#include <sys/rman.h>
#include <machine/resource.h>

struct resource_spec {
    int    type; /* SYS_RES_* */
    int    rid;
    int    flags;
};

int bus_alloc_resources(device_t, struct resource_spec *, struct resource **);
void bus_release_resources(device_t, const struct resource_spec * , struct resource, **);
```

Available resource types:

<code>SYS_RES_IOPORT</code>	<code>SYS_RES_MEMORY</code>	<code>SYS_RES_IRQ</code>	<code>SYS_RES_DRQ</code>
-----------------------------	-----------------------------	--------------------------	--------------------------

The rid is bus/device dependant. For PCI devices, it is `PCIR_BAR(x)`, where x is the xth BAR of the PCI device.

Flags:

RF\_ALLOCATED                      RF\_ACTIVE                      RF\_SHAREABLE                      RF\_TIMESHARE

For additional information see `bus_alloc_resource(9)`.

## USING RESOURCES

```
void bus_read_N(struct resource *, bus_size_t offset);
```

```
void bus_{read,write}_{multi,region}_N(struct resource *, bus_size_t offset, uintN_t *data, bus_size_t count);
```

```
void bus_write_N(struct resource *, bus_size_t offset, uintN_t val);
```

```
void bus_set_{multi,region}_N(struct resource *, bus_size_t offset, uintN_t val, bus_size_t count);
```

The `N` is one of 1, 2, 4 or 8. The functions handle converting to native byte order automatically unless the `_stream` version is called. The `_multi` version of the functions is useful for IO ports where you need to read/write from the same port multiple times. The `_region` version of the functions is useful for memory buffers when you need to copy a range to/from the device from/to memory.

For additional information see `bus_space(9)`.

## USING INTERRUPTS

```
typedef void (*driver_intr_t) (void *);
```

```
int bus_setup_intr(device_t, struct resource *, int flags, driver_intr_t, void *, void **cookiep);
```

```
int bus_teardown_intr(device_t, struct resource *, void *cookiep);
```

Flags:

INTR\_FAST                      INTR\_EXCL                      INTR\_MPSAFE                      INTR\_ENTROPY

A `INTR_TYPE_xxx` must be specified. The available types are: TTY, BIO, NET, CAM, MISC, CLK, AV. If `INTR_EXCL` is not specified, the driver must be able to handle when the `driver_intr_t` is called before `bus_setup_intr` returns. For additional information see `bus_setup_intr(9)`.

## CHARACTER DEVICE

```
#include <sys/conf.h>
```

```
static struct cdevsw dv_cdevsw = {
    .d_version = D_VERSION,
    .d_flags = D_TRACKCLOSE,
    .d_open = dv_os_open,
    .d_close = dv_os_close,
    .d_name = "dv",
};
```

```
struct cdev * make_dev(struct cdevsw *cdevsw, int minor, uid_t uid, gid_t gid, int perms, const char *fmt, ...);
```

```
int minor(struct cdev *);
```

```
void destroy_dev(struct cdev *);
```

The `make_dev()` makes a device file with `minor`, `uid`, `gid` and `perms`. The name will be `fmt` formatted with the kernel equivalent of `sprintf(3)`. Available types for `d_flags`:

D\_TAPE                      D\_DISK                      D\_TTY                      D\_MEM

Available flags for `d_flags`:

D\_TRACKCLOSE                      D\_MMAP\_ANON                      D\_PSEUDO                      D\_NEEDGIANT

The functions that can be provided are:

<code>d_open</code>	<code>d_fdopen</code>	<code>d_close</code>	<code>d_read</code>
<code>d_write</code>	<code>d_ioctl</code>	<code>d_poll</code>	<code>d_mmap</code>
<code>d_strategy</code>	<code>d_dump</code>	<code>d_kqfilter</code>	<code>d_purge</code>

The returned `struct cdev *` has the fields `si_drv0`, `si_drv1`, and `si_drv2` which are available for driver assignment. `si_drv0` is an `int` while the other two are `void *`.

For additional information see `make_dev(9)`.

## IOCTL

```
#include <sys/ioctl.h>
```

```
typedef int (*d_ioctl_t) (struct cdev *dev, u_long cmd, caddr_t data, int fflag, struct thread *td);
```

```

#define NOARGIOCTL      _IO(c, n)
#define READIOCTL      _IOR(c, n, type)
#define WRITEIOCTL     _IOW(c, n, type)
#define READWRITEIOCTL _IOWR(c, n, type)

```

Ioctl's are defined using one of the `_IO`, `_IOR`, `_IOW` and `_IOWR` macros. The kernel transfers the data to userland based upon the size of the type. `c` is a `char` that is used to differentiate groups of ioctls. Within each group, `n` is an `int` used to tell the differences between ioctls within the group.

## BUS\_DMA

```

#include <machine/bus.h>

int bus_dma_tag_create(bus_dma_tag_t parent, bus_size_t alignment, bus_size_t boundary,
bus_addr_t lowaddr, bus_addr_t highaddr, bus_dma_filter_t *filtfunc, void *filtfuncarg,
bus_size_t maxsize, int nsegments, bus_size_t maxsegsz, int flags, bus_dma_lock_t *lockfunc, void
*lockfuncarg, bus_dma_tag_t *dmat);

int bus_dma_tag_destroy(bus_dma_tag_t dmat);

int bus_dmamap_create(bus_dma_tag_t dmat, int flags, bus_dmamap_t *mapp);

int bus_dmamap_destroy(bus_dma_tag_t dmat, bus_dmamap_t map);

int bus_dmamap_load(bus_dma_tag_t dmat, bus_dmamap_t map, void *buf, bus_size_t buflen,
bus_dmamap_callback_t *callback, void *callback_arg, int flags);

int bus_dmamap_load_mbuf(bus_dma_tag_t dmat, bus_dmamap_t map, struct mbuf *mbuf,
bus_dmamap_callback2_t *callback, void *callback_arg, int flags);

int bus_dmamap_load_mbuf_sg(bus_dma_tag_t dmat, bus_dmamap_t map, struct mbuf *mbuf,
bus_dma_segment_t *segs, int *nsegs, int flags);

int bus_dmamap_load_uio(bus_dma_tag_t dmat, bus_dmamap_t map, struct uio *uio,
bus_dmamap_callback2_t *callback, void *callback_arg, int flags);

void bus_dmamap_unload(bus_dma_tag_t dmat, bus_dmamap_t map);

void bus_dmamap_sync(bus_dma_tag_t dmat, bus_dmamap_t map, op);

int bus_dmamem_alloc(bus_dma_tag_t dmat, void **vaddr, int flags, bus_dmamap_t *mapp);

void bus_dmamem_free(bus_dma_tag_t dmat, void *vaddr, bus_dmamap_t map);

```

A tag is what describes the restrictions on the capabilities of the device's dma engine. If the dma engine can only write to 8 byte aligned addresses, then alignment shall be 8.

Flags:

```

BUS_DMA_WAITOK          BUS_DMA_NOWAIT          BUS_DMA_ALLOCNOW          BUS_DMA_COHERENT

```

`bus_dmamap_sync()` ops are `BUS_DMASYNC_{PRE,POST}{READ,WRITE}`. The operations are from the driver's point of view. If you are reading from the device into memory, it is a read op. If you are writing to the device, it is a write op.

For additional information see `bus_dma(9)`.

## PCI BUS

Each bus exports a set of ivars that is accessible to each of the children. They are normally accessed via `(bus_name)_get_(ivar)(dev)`. Use `(bus_name)_set_(ivar)(dev, ivar)` to change the value. Available ivars:

subvendor	subdevice	vendor	device
devid	class	subclass	progif
revid	intpin	irq	bus
slot	function	ether	cmdreg
cachelnsz	mingnt	maxlat	lattimer

These are normally used for `_probe` functions:

```

static int
dv_probe(device_t dev)
{
    switch (pci_get_vendor(dev)) {
        case PCI_VENDOR_FOOBAR: /* FooBar */
            switch (pci_get_device(dev)) {
                case PCI_PRODUCT_FOOBAR_F9000:
                    device_set_desc(dev, "FooBar F9000");
                    return 0;
            }
    }

    return (ENXIO);
}

```

For additional information see `pci(9)`.

## SYSCALL

```
SYSCALL_NODE(_parent, OID_AUTO, name, NULL, const char *desc);
```

```
SYSCALL_type(_parent, OID_AUTO, name, flag, ctype *, ctype, const char *desc);
```

These are used to define compile time sysctls. There are `_ADD` versions that can be used at run time. The argument *type* is be one of INT, UINT, LONG, ULONG, STRUCT or STRING. For STRING, *ctype* is *char*.

Flags:

CTLFLAG_RD	CTLFLAG_WR	CTLFLAG_RW	CTLFLAG_NOLOCK
CTLFLAG_ANYBODY	CTLFLAG_SECURE	CTLFLAG_PRISON	CTLFLAG_DYN
CTLFLAG_SKIP	CTLFLAG_TUN	CTLFLAG_RDTUN	

## SYSINIT

```
#include <sys/kernel.h>
```

```
typedef void (*sysinit_nfunc_t) (void *);
```

```
SYSINIT(uniquifier, subsystem, order, sysinit_nfunc_t, void *);
```

For a complete list of subsystems see `sys/kernel.h` (order top to bottom, left to right):

SI_SUB_TUNABLES	SI_SUB_LOCK	SI_SUB_DRIVERS	SI_SUB_PSEUDO
SI_SUB_KMEM	SI_SUB_KLD	SI_SUB_CONFIGURE	SI_SUB_KTHREAD_IDLE
SI_SUB_WITNESS	SI_SUB_DEVFS	SI_SUB_CLOCKS	SI_SUB_RUN_SCHEDULER

Orders:

SI_ORDER_FIRST	SI_ORDER_SECOND	SI_ORDER_MIDDLE	SI_ORDER_ANY
----------------	-----------------	-----------------	--------------

## CALLOUT

```
void callout_init_mtx(struct callout *, struct mtx *, int flags);
```

```
int callout_stop(struct callout *);
```

```
int callout_drain(struct callout *);
```

```
int callout_reset(struct callout *, int ticks, timeout_t *func, void *arg);
```

Flags may be `CALLOUT_RETURNUNLOCKED`. The function `callout_stop()` will stop any pending callout and must be called with the mutex held. The function `callout_drain()` will stop any pending callouts and wait for any running callouts to finish. It must not be called with locks that the callout depends upon.

For additional information see `timeout(9)`.

## TASKQUEUE

```
typedef void (*task_fn_t) (void *, int pending);
```

```
TASK_INIT(struct task *, int priority, task_fn_t *, void *);
```

```
TASKQUEUE_DECLARE(name);
```

```
TASKQUEUE_DEFINE_THREAD(name);
```

```
int taskqueue_enqueue(struct taskqueue *, struct task *);
```

```
int taskqueue_enqueue_fast(struct taskqueue *, struct task *);
```

```
taskqueue_drain(struct taskqueue *, struct task *);
```

The system provides four global taskqueues:

<code>taskqueue_fast</code>	<code>taskqueue_swi</code>	<code>taskqueue_swi_giant</code>	<code>taskqueue_thread</code>
-----------------------------	----------------------------	----------------------------------	-------------------------------

Use `TASKQUEUE_DEFINE_THREAD()` to create your own queue that has a dedicated dispatch thread. For the global queue `taskqueue_fast`, use the function `taskqueue_enqueue_fast()`. The `taskqueue_fast` queue is safe for use with fast interrupt handlers.

For additional information see `taskqueue(9)`.

## SEE ALSO

`bus_alloc_resource(9)`, `bus_dma(9)`, `BUS_SETUP_INTR(9)`, `bus_space(9)`, `DECLARE_GEOM_CLASS(9)`, `driver(9)`, `DRIVER_MODULE(9)`, `ifnet(9)`, `make_dev(9)`, `module(9)`, `pci(9)`, `resource_int_value(9)`, `selrecord(9)`, `taskqueue(9)`, `timeout(9)`

## AUTHORS

John-Mark Gurney <jmg@FreeBSD.org>