NAME

bhnd - BHND driver programming interface

SYNOPSIS

#include <dev/bhnd/bhnd.h>

Bus Resource Functions

int
bhnd_activate_resource(*device_t dev, int type, int rid, struct bhnd_resource *r*);

*struct bhnd_resource **
bhnd_alloc_resource(*device_t dev, int type, int *rid, rman_res_t start, rman_res_t end, rman_res_t count, u_int flags*);

*struct bhnd_resource **
bhnd_alloc_resource_any(*device_t dev, int type, int *rid, u_int flags*);

int
bhnd_alloc_resources(*device_t dev, struct resource_spec *rs, struct bhnd_resource **res*);

int
bhnd_deactivate_resource(*device_t dev, int type, int rid, struct bhnd_resource *r*);

int
bhnd_release_resource(*device_t dev, int type, int rid, struct bhnd_resource *r*);

void
bhnd_release_resources(*device_t dev, const struct resource_spec *rs, struct bhnd_resource **res*);

Bus Space Functions

void
bhnd_bus_barrier(*struct bhnd_resource *r, bus_size_t offset, bus_size_t length, int flags*);

uint8_t
bhnd_bus_read_1(*struct bhnd_resource *r, bus_size_t offset*);

uint16_t
bhnd_bus_read_2(*struct bhnd_resource *r, bus_size_t offset*);

uint32_t

bhnd_bus_read_4(*struct bhnd_resource *r, bus_size_t offset*);

void
bhnd_bus_read_multi_1(*struct bhnd_resource *r, bus_size_t offset, uint8_t *datap, bus_size_t count*);

void
bhnd_bus_read_multi_2(*struct bhnd_resource *r, bus_size_t offset, uint16_t *datap, bus_size_t count*);

void
bhnd_bus_read_multi_4(*struct bhnd_resource *r, bus_size_t offset, uint32_t *datap, bus_size_t count*);

void
bhnd_bus_read_multi_stream_1(*struct bhnd_resource *r, bus_size_t offset, uint8_t *datap, bus_size_t count*);

void
bhnd_bus_read_multi_stream_2(*struct bhnd_resource *r, bus_size_t offset, uint16_t *datap, bus_size_t count*);

void
bhnd_bus_read_multi_stream_4(*struct bhnd_resource *r, bus_size_t offset, uint32_t *datap, bus_size_t count*);

void
bhnd_bus_read_region_1(*struct bhnd_resource *r, bus_size_t offset, uint8_t *datap, bus_size_t count*);

void
bhnd_bus_read_region_2(*struct bhnd_resource *r, bus_size_t offset, uint16_t *datap, bus_size_t count*);

void
bhnd_bus_read_region_4(*struct bhnd_resource *r, bus_size_t offset, uint32_t *datap, bus_size_t count*);

void
bhnd_bus_read_region_stream_1(*struct bhnd_resource *r, bus_size_t offset, uint8_t *datap, bus_size_t count*);

void
bhnd_bus_read_region_stream_2(*struct bhnd_resource *r, bus_size_t offset, uint16_t *datap, bus_size_t count*);

void
bhnd_bus_read_region_stream_4(*struct bhnd_resource *r, bus_size_t offset, uint32_t *datap, bus_size_t count*);

void
bhnd_bus_read_stream_1(*struct bhnd_resource *r, bus_size_t offset*);

void
bhnd_bus_read_stream_2(*struct bhnd_resource *r, bus_size_t offset*);

uint32_t
bhnd_bus_read_stream_4(*struct bhnd_resource *r, bus_size_t offset*);

void
bhnd_bus_set_multi_1(*struct bhnd_resource *r, bus_size_t offset, uint8_t value, bus_size_t count*);

void
bhnd_bus_set_multi_2(*struct bhnd_resource *r, bus_size_t offset, uint16_t value, bus_size_t count*);

void
bhnd_bus_set_multi_4(*struct bhnd_resource *r, bus_size_t offset, uint32_t value, bus_size_t count*);

void
bhnd_bus_set_region_1(*struct bhnd_resource *r, bus_size_t offset, uint8_t value, bus_size_t count*);

void
bhnd_bus_set_region_2(*struct bhnd_resource *r, bus_size_t offset, uint16_t value, bus_size_t count*);

void
bhnd_bus_set_region_4(*struct bhnd_resource *r, bus_size_t offset, uint32_t value, bus_size_t count*);

void
bhnd_bus_write_1(*struct bhnd_resource *r, uint8_t value*);

void
bhnd_bus_write_2(*struct bhnd_resource *r, uint16_t value*);

void
bhnd_bus_write_4(*struct bhnd_resource *r, uint32_t value*);

void
bhnd_bus_write_multi_1(*struct bhnd_resource *r, bus_size_t offset, uint8_t *datap, bus_size_t count*);

void
bhnd_bus_write_multi_2(*struct bhnd_resource *r, bus_size_t offset, uint16_t *datap, bus_size_t count*);

void
bhnd_bus_write_multi_4(*struct bhnd_resource *r, bus_size_t offset, uint32_t *datap, bus_size_t count*);

void
bhnd_bus_write_multi_stream_1(*struct bhnd_resource *r, bus_size_t offset, uint8_t *datap, bus_size_t count*);

void
bhnd_bus_write_multi_stream_2(*struct bhnd_resource *r, bus_size_t offset, uint16_t *datap, bus_size_t count*);

void
bhnd_bus_write_multi_stream_4(*struct bhnd_resource *r, bus_size_t offset, uint32_t *datap, bus_size_t count*);

void
bhnd_bus_write_region_1(*struct bhnd_resource *r, bus_size_t offset, uint8_t *datap, bus_size_t count*);

void
bhnd_bus_write_region_2(*struct bhnd_resource *r, bus_size_t offset, uint16_t *datap, bus_size_t count*);

void
bhnd_bus_write_region_4(*struct bhnd_resource *r, bus_size_t offset, uint32_t *datap, bus_size_t count*);

void
bhnd_bus_write_region_stream_1(*struct bhnd_resource *r, bus_size_t offset, uint8_t *datap, bus_size_t count*);

void
bhnd_bus_write_region_stream_2(*struct bhnd_resource *r, bus_size_t offset, uint16_t *datap, bus_size_t count*);

void
bhnd_bus_write_region_stream_4(*struct bhnd_resource *r, bus_size_t offset, uint32_t *datap, bus_size_t count*);

void
bhnd_bus_write_stream_1(*struct bhnd_resource *r, uint8_t value*);

void
bhnd_bus_write_stream_2(*struct bhnd_resource *r, uint16_t value*);

void
bhnd_bus_write_stream_4(*struct bhnd_resource *r, uint32_t value*);

Device Configuration Functions

int
bhnd_read_ioctl(*device_t dev, uint16_t *ioctl*);

int
bhnd_write_ioctl(*device_t dev, uint16_t value, uint16_t mask*);

int
bhnd_read_iost(*device_t dev, uint16_t *iost*);

uint32_t
bhnd_read_config(*device_t dev, bus_size_t offset, void *value, u_int width*);

int
bhnd_write_config(*device_t dev, bus_size_t offset, const void *value, u_int width*);

int
bhnd_reset_hw(*device_t dev, uint16_t ioctl, uint16_t reset_ioctl*);

int
bhnd_suspend_hw(*device_t dev, uint16_t ioctl*);

bool
bhnd_is_hw_suspended(*device_t dev*);

Device Information Functions

bhnd_attach_type

bhnd_get_attach_type(*device_t dev*);

*const struct bhnd_chipid **
bhnd_get_chipid(*device_t dev*);

bhnd_devclass_t
bhnd_get_class(*device_t dev*);

u_int
bhnd_get_core_index(*device_t dev*);

struct bhnd_core_info
bhnd_get_core_info(*device_t dev*);

int
bhnd_get_core_unit(*device_t dev*);

uint16_t
bhnd_get_device(*device_t dev*);

*const char **
bhnd_get_device_name(*device_t dev*);

uint8_t
bhnd_get_hwrev(*device_t dev*);

uint16_t
bhnd_get_vendor(*device_t dev*);

*const char **
bhnd_get_vendor_name(*device_t dev*);

int
bhnd_read_board_info(*device_t dev, struct bhnd_board_info *info*);

Device Matching Functions

bool
bhnd_board_matches(*const struct bhnd_board_info *board, const struct bhnd_board_match *desc*);

device_t

bhnd_bus_match_child(*device_t bus, const struct bhnd_core_match *desc*);

bool
bhnd_chip_matches(*const struct bhnd_chipid *chip, const struct bhnd_chip_match *desc*);

struct bhnd_core_match
bhnd_core_get_match_desc(*const struct bhnd_core_info *core*);

bool
bhnd_core_matches(*const struct bhnd_core_info *core, const struct bhnd_core_match *desc*);

bool
bhnd_cores_equal(*const struct bhnd_core_info *lhs, const struct bhnd_core_info *rhs*);

bool
bhnd_hwrev_matches(*uint16_t hwrev, const struct bhnd_hwrev_match *desc*);

*const struct bhnd_core_info **
bhnd_match_core(*const struct bhnd_core_info *cores, u_int num_cores,*
*const struct bhnd_core_match *desc*);

Device Table Functions

*const struct bhnd_device **
bhnd_device_lookup(*device_t dev, const struct bhnd_device *table, size_t entry_size*);

bool
bhnd_device_matches(*device_t dev, const struct bhnd_device_match *desc*);

uint32_t
bhnd_device_quirks(*device_t dev, const struct bhnd_device *table, size_t entry_size*);

BHND_BOARD_QUIRK(*board, flags*);

BHND_CHIP_QUIRK(*chip, hwrev, flags*);

BHND_CORE_QUIRK(*hwrev, flags*);

BHND_DEVICE(*vendor, device, desc, quirks, ...*);

BHND_DEVICE_IS_END(*struct bhnd_device *d*);

BHND_DEVICE_QUIRK_IS_END(*struct bhnd_device_quirk *q*);

BHND_PKG_QUIRK(*chip*, *pkg*, *flags*);

```
struct bhnd_device_quirk {
        struct bhnd_device_match      desc;
        uint32_t                      quirks;
};

struct bhnd_device {
   const struct bhnd_device_match        core;
   const char                                  *desc;
   const struct bhnd_device_quirk        *quirks_table;
   uint32_t                                    device_flags;
};

enum {
        BHND_DF_ANY   = 0,
        BHND_DF_HOSTB         = (1 << 0),
        BHND_DF_SOC   = (1 << 1),
        BHND_DF_ADAPTER       = (1 << 2)
};

#define BHND_DEVICE_END { { BHND_MATCH_ANY }, NULL, NULL, 0 }

#define BHND_DEVICE_QUIRK_END { { BHND_MATCH_ANY }, 0 }
```

DMA Address Translation Functions

int
bhnd_get_dma_translation(*device_t dev*, *u_int width*, *uint32_t flags*, *bus_dma_tag_t *dmat*, *struct bhnd_dma_translation *translation*);

```
struct bhnd_dma_translation {
        bhnd_addr_t         base_addr;
        bhnd_addr_t         addr_mask;
        bhnd_addr_t         addrext_mask;
        uint32_t   flags;
};

typedef enum {
```

```
        BHND_DMA_ADDR_30BIT     = 30,
        BHND_DMA_ADDR_32BIT     = 32,
        BHND_DMA_ADDR_64BIT     = 64
} bhnd_dma_addrwidth;

enum bhnd_dma_translation_flags {
        BHND_DMA_TRANSLATION_PHYSMAP            = (1<<0),
        BHND_DMA_TRANSLATION_BYTESWAPPED        = (1<<1)
};
```

Interrupt Functions

u_int
bhnd_get_intr_count(*device_t dev*);

int
bhnd_get_intr_ivec(*device_t dev*, *u_int intr*, *u_int *ivec*);

int
bhnd_map_intr(*device_t dev*, *u_int intr*, *rman_res_t *irq*);

void
bhnd_unmap_intr(*device_t dev*, *rman_res_t irq*);

NVRAM Functions

int
bhnd_nvram_getvar(*device_t dev*, *const char *name*, *void *buf*, *size_t *len*, *bhnd_nvram_type type*);

int
bhnd_nvram_getvar_array(*device_t dev*, *const char *name*, *void *buf*, *size_t size*,
bhnd_nvram_type type);

int
bhnd_nvram_getvar_int(*device_t dev*, *const char *name*, *void *value*, *int width*);

int
bhnd_nvram_getvar_int8(*device_t dev*, *const char *name*, *int8_t *value*);

int
bhnd_nvram_getvar_int16(*device_t dev*, *const char *name*, *int16_t *value*);

int
bhnd_nvram_getvar_int32(*device_t dev, const char *name, int32_t *value*);

int
bhnd_nvram_getvar_uint(*device_t dev, const char *name, void *value, int width*);

int
bhnd_nvram_getvar_uint8(*device_t dev, const char *name, uint8_t *value*);

int
bhnd_nvram_getvar_uint16(*device_t dev, const char *name, uint16_t *value*);

int
bhnd_nvram_getvar_uint32(*device_t dev, const char *name, uint32_t *value*);

int
bhnd_nvram_getvar_str(*device_t dev, const char *name, char *buf, size_t len, size_t *rlen*);

*const char **
bhnd_nvram_string_array_next(*const char *inp, size_t ilen, const char *prev, size_t *olen*);

```
typedef enum {
        BHND_NVRAM_TYPE_UINT8           = 0,
        BHND_NVRAM_TYPE_UINT16          = 1,
        BHND_NVRAM_TYPE_UINT32          = 2,
        BHND_NVRAM_TYPE_UINT64          = 3,
        BHND_NVRAM_TYPE_INT8            = 4,
        BHND_NVRAM_TYPE_INT16           = 5,
        BHND_NVRAM_TYPE_INT32           = 6,
        BHND_NVRAM_TYPE_INT64           = 7,
        BHND_NVRAM_TYPE_CHAR            = 8,
        BHND_NVRAM_TYPE_STRING          = 9,
        BHND_NVRAM_TYPE_BOOL            = 10,
        BHND_NVRAM_TYPE_NULL            = 11,
        BHND_NVRAM_TYPE_DATA            = 12
        BHND_NVRAM_TYPE_UINT8_ARRAY     = 16,
        BHND_NVRAM_TYPE_UINT16_ARRAY    = 17,
        BHND_NVRAM_TYPE_UINT32_ARRAY    = 18,
        BHND_NVRAM_TYPE_UINT64_ARRAY    = 19,
        BHND_NVRAM_TYPE_INT8_ARRAY      = 20,
```

```
        BHND_NVRAM_TYPE_INT16_ARRAY     = 21,
        BHND_NVRAM_TYPE_INT32_ARRAY     = 22,
        BHND_NVRAM_TYPE_INT64_ARRAY     = 23,
        BHND_NVRAM_TYPE_CHAR_ARRAY      = 24,
        BHND_NVRAM_TYPE_STRING_ARRAY    = 25,
        BHND_NVRAM_TYPE_BOOL_ARRAY      = 26
} bhnd_nvram_type;
```

Port/Region Functions

int
bhnd_decode_port_rid(*device_t dev*, *int type*, *int rid*, *bhnd_port_type *port_type*, *u_int *port*, *u_int *region*);

u_int
bhnd_get_port_count(*device_t dev*, *bhnd_port_type type*);

int
bhnd_get_port_rid(*device_t dev*, *bhnd_port_type type*, *u_int port*, *u_int region*);

int
bhnd_get_region_addr(*device_t dev*, *bhnd_port_type port_type*, *u_int port*, *u_int region*, *bhnd_addr_t *region_addr*, *bhnd_size_t *region_size*);

u_int
bhnd_get_region_count(*device_t dev*, *bhnd_port_type type*, *u_int port*);

bool
bhnd_is_region_valid(*device_t dev*, *bhnd_port_type type*, *u_int port*, *u_int region*);

```
typedef enum {
        BHND_PORT_DEVICE        = 0,
        BHND_PORT_BRIDGE        = 1,
        BHND_PORT_AGENT                 = 2
} bhnd_port_type;
```

Power Management Functions

int
bhnd_alloc_pmu(*device_t dev*);

int

bhnd_release_pmu(*device_t dev*);

int
bhnd_enable_clocks(*device_t dev, uint32_t clocks*);

int
bhnd_request_clock(*device_t dev, bhnd_clock clock*);

int
bhnd_get_clock_freq(*device_t dev, bhnd_clock clock, u_int *freq*);

int
bhnd_get_clock_latency(*device_t dev, bhnd_clock clock, u_int *latency*);

int
bhnd_request_ext_rsrc(*device_t dev, u_int rsrc*);

int
bhnd_release_ext_rsrc(*device_t dev, u_int rsrc*);

```
typedef enum {
        BHND_CLOCK_DYN      = (1 << 0),
        BHND_CLOCK_ILP      = (1 << 1),
        BHND_CLOCK_ALP      = (1 << 2),
        BHND_CLOCK_HT       = (1 << 3)
} bhnd_clock;
```

Service Provider Functions

int
bhnd_register_provider(*device_t dev, bhnd_service_t service*);

int
bhnd_deregister_provider(*device_t dev, bhnd_service_t service*);

device_t
bhnd_retain_provider(*device_t dev, bhnd_service_t service*);

void
bhnd_release_provider(*device_t dev, device_t provider, bhnd_service_t service*);

```
typedef enum {
        BHND_SERVICE_CHIPC,
        BHND_SERVICE_PWRCTL,
        BHND_SERVICE_PMU,
        BHND_SERVICE_NVRAM,
        BHND_SERVICE_GPIO,
        BHND_SERVICE_ANY       = 1000
} bhnd_service_t;
```

Utility Functions

*bhnd_erom_class_t **
bhnd_driver_get_erom_class(*driver_t *driver*);

bhnd_devclass_t
bhnd_find_core_class(*uint16_t vendor*, *uint16_t device*);

*const char **
bhnd_find_core_name(*uint16_t vendor*, *uint16_t device*);

bhnd_devclass_t
bhnd_core_class(*const struct bhnd_core_info *ci*);

*const char **
bhnd_core_name(*const struct bhnd_core_info *ci*);

int
bhnd_format_chip_id(*char *buffer*, *size_t size*, *uint16_t chip_id*);

void
bhnd_set_custom_core_desc(*device_t dev*, *const char *dev_name*);

void
bhnd_set_default_core_desc(*device_t dev*);

*const char **
bhnd_vendor_name(*uint16_t vendor*);

```
#define    BHND_CHIPID_MAX_NAMELEN    32
```

DESCRIPTION

bhnd provides a unified bus and driver programming interface for the on-chip interconnects and IP cores found in Broadcom Home Networking Division (BHND) devices.

The BHND device family consists of MIPS/ARM SoCs (System On a Chip) and host-connected chipsets based on a common library of Broadcom IP cores, connected via one of two on-chip backplane (hardware bus) architectures.

Hardware designed prior to 2009 used Broadcom's "SSB" backplane architecture, based on Sonics Silicon's interconnect IP. Each core on the Sonics backplane vends a 4 KiB register block, containing both device-specific CSRs, and SSB-specific per-core device management (enable/reset/etc) registers.

Subsequent hardware is based on Broadcom's "BCMA" backplane, based on ARM's AMBA IP. The IP cores used in earlier SSB-based devices were adapted for compatibility with the new backplane, with additional "wrapper" cores providing per-core device management functions in place of the SSB per-core management registers.

When BHND hardware is used as a host-connected peripheral (e.g., in a PCI Wi-Fi card), the on-chip peripheral controller core is configured to operate as an endpoint device, bridging access to the SoC hardware:

- Host access to SoC address space is provided via a set of register windows (e.g., a set of configurable windows into SoC address space mapped via PCI BARs)

- DMA is supported by the bridge core's sparse mapping of host address space into the backplane address space. These address regions may be used as a target for the on-chip DMA engine.

- Any backplane interrupt vectors routed to the bridge core may be mapped by the bridge to host interrupts (e.g., PCI INTx/MSI/MSI-X).

The **bhnd** driver programming interface -- and bhndb(4) host bridge drivers -- support the implementation of common drivers for Broadcom IP cores, whether attached via a BHND host bridge, or via the native SoC backplane.

Bus Resource Functions

The bhnd_resource functions are wrappers for the standard *struct resource* bus APIs, providing support for *SYS_RES_MEMORY* resources that, on bhndb(4) bridged chipsets, may require on-demand remapping of address windows prior to accessing bus memory.

These functions are primarily used in the implementation of BHND platform device drivers that, on

host-connected peripherals, must share a small set of register windows during initial setup and teardown.

BHND peripherals are designed to not require register window remapping during normal operation, and most drivers may safely use the standard *struct resource* APIs directly.

The **bhnd_activate_resource**() function activates a previously allocated resource.

The arguments are as follows:

dev The device holding ownership of the allocated resource.

type The type of the resource.

rid The bus-specific handle that identifies the resource being activated.

r A pointer to the resource returned by **bhnd_alloc_resource**().

The **bhnd_alloc_resource**() function allocates a resource from a device's parent bhnd(4) bus.

The arguments are as follows:

dev The device requesting resource ownership.

type The type of resource to allocate. This may be any type supported by the standard bus_alloc_resource(9) function.

rid The bus-specific handle identifying the resource being allocated.

start The start address of the resource.

end The end address of the resource.

count The size of the resource.

flags The flags for the resource to be allocated. These may be any values supported by the standard bus_alloc_resource(9) function.

To request that the bus supply the resource's default *start*, *end*, and *count* values, pass *start* and *end* values of 0ul and ~0ul respectively, and a *count* of 1.

The **bhnd_alloc_resource_any**() function is a convenience wrapper for **bhnd_alloc_resource**(), using the resource's default *start*, *end*, and *count* values.

The arguments are as follows:

dev The device requesting resource ownership.

type The type of resource to allocate. This may be any type supported by the standard bus_alloc_resource(9) function.

rid The bus-specific handle identifying the resource being allocated.

flags The flags for the resource to be allocated. These may be any values supported by the standard bus_alloc_resource(9) function.

The **bhnd_alloc_resources**() function allocates resources defined in resource specification from a device's parent bhnd(4) bus.

The arguments are as follows:

dev The device requesting ownership of the resources.

rs A standard bus resource specification. If all requested resources, are successfully allocated, this will be updated with the allocated resource identifiers.

res If all requested resources are successfully allocated, this will be populated with the allocated *struct bhnd_resource* instances.

The **bhnd_deactivate_resource**() function deactivates a resource previously activated by. **bhnd_activate_resource**(). The arguments are as follows:

dev The device holding ownership of the activated resource.

type The type of the resource.

rid The bus-specific handle identifying the resource.

r A pointer to the resource returned by bhnd_alloc_resource.

The **bhnd_release_resource**() function frees a resource previously returned by **bhnd_alloc_resource**().

The arguments are as follows:

dev The device holding ownership of the resource.

type The type of the resource.

rid The bus-specific handle identifying the resource.

r A pointer to the resource returned by bhnd_alloc_resource.

The **bhnd_release_resources**() function frees resources previously returned by **bhnd_alloc_resources**(). The arguments are as follows:

dev The device that owns the resources.

rs A standard bus resource specification previously initialized by **bhnd_alloc_resources**().

res The resources to be released.

The *bhnd_resource* structure contains the following fields:

res A pointer to the bus *struct resource*.

direct If true, the resource requires bus window remapping before it is MMIO accessible.

Bus Space Functions

The bhnd_bus_space functions wrap their equivalent bus_space(9) counterparts, and provide support for accessing bus memory via *struct bhnd_resource*.

bhnd_bus_barrier()
bhnd_bus_[read|write]_[1|2|4]()
bhnd_bus_[read_multi|write_multi]_[1|2|4]()
bhnd_bus_[read_multi_stream|write_multi_stream]_[1|2|4]()
bhnd_bus_[read_region|write_region]_[1|2|4]()
bhnd_bus_[read_region_stream|write_region_stream]_[1|2|4]()
bhnd_bus_[read_stream|write_stream]_[1|2|4]()
bhnd_bus_[set_multi|set_stream]_[1|2|4]()

Drivers that do not rely on *struct bhnd_resource* should use the standard *struct resource* and bus_space(9) APIs directly.

Device Configuration Functions

The **bhnd_read_ioctl**() function is used to read the I/O control register value of device *dev*, returning the current value in *ioctl*.

The **bhnd_write_ioctl**() function is used to modify the I/O control register of *dev*. The new value of the register is computed by updating any bits set in *mask* to *value*. The following I/O control flags are supported:

BHND_IOCTL_BIST	Initiate a built-in self-test (BIST). Must be cleared after BIST results are read via the IOST (I/O Status) register.
BHND_IOCTL_PME	Enable posting of power management events by the core.
BHND_IOCTL_CLK_FORCE	Force disable of clock gating, resulting in all clocks being distributed within the core. Should be set when asserting/deasserting reset to ensure the reset signal fully propagates to the entire core.
BHND_IOCTL_CLK_EN	If cleared, the core clock will be disabled. Should be set during normal operation, and cleared when the core is held in reset.
BHND_IOCTL_CFLAGS	The mask of IOCTL bits reserved for additional core-specific I/O control flags.

The **bhnd_read_iost**() function is used to read the I/O status register of device *dev*, returning the current value in *iost*. The following I/O status flags are supported:

BHND_IOST_BIST_DONE	Set upon BIST completion. Will be cleared when the BHND_IOCTL_BIST flag of the I/O control register is cleared using **bhnd_write_ioctl**().
BHND_IOST_BIST_FAIL	Set upon detection of a BIST error; the value is unspecified if BIST has not completed and BHND_IOST_BIST_DONE is not also set.
BHND_IOST_CLK	Set if the core has required that clocked be ungated, or cleared otherwise. The value is undefined if a core does not support clock gating.
BHND_IOST_DMA64	Set if this core supports 64-bit DMA.

BHND_IOST_CFLAGS The mask of IOST bits reserved for additional core-specific I/O status flags.

The **bhnd_read_config**() function is used to read a data item of *width* bytes at *offset* from the backplane-specific agent/config space of the device *dev*.

The **bhnd_write_config**() function is used to write a data item of *width* bytes with *value* at *offset* from the backplane-specific agent/config space of the device *dev*. The requested *width* must be one of 1, 2, or 4 bytes.

The agent/config space accessible via **bhnd_read_config**() and **bhnd_write_config**() is backplane-specific, and these functions should only be used for functionality that is not available via another **bhnd** function.

The **bhnd_suspend_hw**() function transitions the device *dev* to a low power "RESET" state, writing *ioctl* to the I/O control flags of *dev*. The hardware may be brought out of this state using **bhnd_reset_hw**().

The **bhnd_reset_hw**() function first transitions the device *dev* to a low power RESET state, writing *ioctl_reset* to the I/O control flags of *dev*, and then brings the device out of RESET, writing *ioctl* to the device's I/O control flags.

The **bhnd_is_hw_suspended**() function returns true if the device *dev* is currently held in a RESET state, or is otherwise not clocked. Otherwise, it returns false.

Any outstanding per-device PMU requests made using **bhnd_enable_clocks**(), **bhnd_request_clock**(), or **bhnd_request_ext_rsrc**() will be released automatically upon placing a device into a RESET state.

Device Information Functions

The **bhnd_get_attach_type**() function returns the attachment type of the parent bhnd(4) bus of device *dev*.

The following attachment types are supported:

BHND_ATTACH_ADAPTER The bus is resident on a bridged adapter, such as a PCI Wi-Fi device.

BHND_ATTACH_NATIVE The bus is resident on the native host, such as the primary or secondary bus of an embedded SoC.

The **bhnd_get_chipid**() function returns chip information from the parent bhnd(4) bus of device *dev*.

The returned *bhnd_chipid* struct contains the following fields:

chip_id The chip identifier.

chip_rev The chip's hardware revision.

chip_pkg The chip's semiconductor package identifier.

Several different physical semiconductor package variants may exist for a given chip, each of which may require driver workarounds for hardware errata, unpopulated components, etc.

chip_type The interconnect architecture used by this chip.

chip_caps The **bhnd** capability flags supported by this chip.

enum_addr The backplane enumeration address. On SSB devices, this will be the base address of the first SSB core. On BCMA devices, this will be the address of the enumeration ROM (EROM) core.

ncores The number of cores on the chip backplane, or 0 if unknown.

The following constants are defined for known *chip_type* values:

BHND_CHIPTYPE_SIBA	SSB interconnect.
BHND_CHIPTYPE_BCMA	BCMA interconnect.
BHND_CHIPTYPE_BCMA_ALT	BCMA-compatible variant found in Broadcom Northstar ARM SoCs.
BHND_CHIPTYPE_UBUS	UBUS interconnect. This BCMA-derived interconnect is found in Broadcom BCM33xx DOCSIS SoCs, and BCM63xx xDSL SoCs. UBUS is not currently supported by bhnd(4).

The following *chip_caps* flags are supported:

BHND_CAP_BP64	The backplane supports 64-bit addressing.
BHND_CAP_PMU	PMU is present.

Additional symbolic constants for known *chip_id*, *chip_pkg*, and *chip_type* values are defined in <*dev/bhnd/bhnd_ids.h*>.

The **bhnd_get_class**() function returns the BHND class of device *dev*, if the device's *vendor* and *device* identifiers are recognized. Otherwise, returns BHND_DEVCLASS_OTHER.

One of the following device classes will be returned:

BHND_DEVCLASS_CC	ChipCommon I/O Controller
BHND_DEVCLASS_CC_B	ChipCommon Auxiliary Controller
BHND_DEVCLASS_PMU	PMU Controller
BHND_DEVCLASS_PCI	PCI Host/Device Bridge
BHND_DEVCLASS_PCIE	PCIe Host/Device Bridge
BHND_DEVCLASS_PCCARD	PCMCIA Host/Device Bridge
BHND_DEVCLASS_RAM	Internal RAM/SRAM
BHND_DEVCLASS_MEMC	Memory Controller
BHND_DEVCLASS_ENET	IEEE 802.3 MAC/PHY
BHND_DEVCLASS_ENET_MAC	IEEE 802.3 MAC
BHND_DEVCLASS_ENET_PHY	IEEE 802.3 PHY
BHND_DEVCLASS_WLAN	IEEE 802.11 MAC/PHY/Radio
BHND_DEVCLASS_WLAN_MAC	IEEE 802.11 MAC
BHND_DEVCLASS_WLAN_PHY	IEEE 802.11 PHY
BHND_DEVCLASS_CPU	CPU Core
BHND_DEVCLASS_SOC_ROUTER	Interconnect Router
BHND_DEVCLASS_SOC_BRIDGE	Interconnect Host Bridge
BHND_DEVCLASS_EROM	Device Enumeration ROM
BHND_DEVCLASS_NVRAM	NVRAM/Flash Controller
BHND_DEVCLASS_SOFTMODEM	Analog/PSTN SoftModem Codec
BHND_DEVCLASS_USB_HOST	USB Host Controller
BHND_DEVCLASS_USB_DEV	USB Device Controller
BHND_DEVCLASS_USB_DUAL	USB Host/Device Controller
BHND_DEVCLASS_OTHER	Other / Unknown
BHND_DEVCLASS_INVALID	Invalid Class

The **bhnd_get_core_info**() function returns the core information for device *dev*. The returned *bhnd_core_info* structure contains the following fields:

vendor	Vendor identifier (JEP-106, ARM 4-bit continuation encoded)
device	Device identifier
hwrev	Hardware revision
core_idx	Core index
unit	Core unit

Symbolic constants for common vendor and device identifiers are defined in <*dev/bhnd/bhnd_ids.h*>. Common vendor identifiers include:

BHND_MFGID_ARM ARM
BHND_MFGID_BCM Broadcom
BHND_MFGID_MIPS MIPS

The **bhnd_get_core_index**(), **bhnd_get_core_unit**(), **bhnd_get_device**(), **bhnd_get_hwrev**(), and **bhnd_get_vendor**() functions are convenience wrappers for **bhnd_get_core_info**(), returning, respect the *core_idx*, *core_unit*, *device*, *hwrev*, or *vendor* field from the *bhnd_core_info* structure.

The **bhnd_get_device_name**() function returns a human readable name for device *dev*.

The **bhnd_get_vendor_name**() function returns a human readable name for the vendor of device *dev*.

The **bhnd_read_board_info**() function attempts to read the board information for device *dev*. The board information will be returned in the location pointed to by *info* on success.

The *bhnd_board_info* structure contains the following fields:

board_vendor Vendor ID of the board manufacturer (PCI-SIG assigned).

board_type Board ID.

board_devid Device ID.

board_rev Board revision.

board_srom_rev Board SROM format revision.

board_flags Board flags (1)

board_flags2 Board flags (2)

board_flags3 Board flags (3)

The *board_devid* field is the Broadcom PCI device ID that most closely matches the capabilities of the BHND device (if any).

On PCI devices, the *board_vendor*, *board_type*, and *board_devid* fields default to the PCI Subsystem Vendor ID, PCI Subsystem ID, and PCI device ID, unless overridden in device NVRAM.

On other devices, including SoCs, the *board_vendor*, *board_type*, and *board_devid* fields will be

populated from device NVRAM.

Symbolic constants for common board flags are defined in <*dev/bhnd/bhnd_ids.h*>.

Device Matching Functions

The bhnd device matching functions are used to match against core, chip, and board-level device attributes. Match requirements are specified using the *struct bhnd_board_match*, *struct bhnd_chip_match*, *struct bhnd_core_match*, *struct bhnd_device_match*, and *struct bhnd_hwrev_match* match descriptor structures.

The **bhnd_board_matches**() function returns true if *board* matches the board match descriptor *desc*. Otherwise, it returns false.

The **bhnd_chip_matches**() function returns true if *chip* matches the chip match descriptor *desc*. Otherwise, it returns false.

The **bhnd_core_matches**() function returns true if *core* matches the core match descriptor *desc*. Otherwise, it returns false.

The **bhnd_device_matches**() function returns true if the device *dev* matches the device match descriptor *desc*. Otherwise, it returns false.

The **bhnd_hwrev_matches**() function returns true if *hwrev* matches the hwrev match descriptor *desc*. Otherwise, it returns false.

The **bhnd_bus_match_child**() function returns the first child device of *bus* that matches the device match descriptor *desc*. If no matching child is found, NULL is returned.

The **bhnd_core_get_match_desc**() function returns an equality match descriptor for the core info in *core*. The returned descriptor will match only on core attributes identical to those defined by *core*.

The **bhnd_cores_equal**() function is a convenience wrapper for **bhnd_core_matches**() and **bhnd_core_get_match_desc**(). This function returns true if the *bhnd_core_info* structures *lhs* and *rhs* are equal. Otherwise, it returns false.

The **bhnd_match_core**() function returns a pointer to the first entry in the array *cores* of length *num_cores* that matches *desc*. If no matching core is found, NULL is returned.

A *bhnd_board_match* match descriptor may be initialized using one or more of the following macros:

BHND_MATCH_BOARD_VENDOR(*vendor*) Match on boards with a vendor equal to *vendor*.

BHND_MATCH_BOARD_TYPE(*type*) Match on boards with a type equal to BHND_BOARD_ ## *type*

BHND_MATCH_SROMREV(*sromrev*) Match on boards with a sromrev that matches BHND_HWREV_ ## *sromrev*.

BHND_MATCH_BOARD_REV(*hwrev*) Match on boards with hardware revisions that match BHND_ ## *hwrev*.

BHND_MATCH_BOARD(*vendor*, *type*) A convenience wrapper for **BHND_MATCH_BOARD_VENDOR**() and **BHND_MATCH_BOARD_TYPE**().

For example:

```
struct bhnd_board_match board_desc = {
        BHND_MATCH_BOARD_VENDOR(BHND_MFGID_BROADCOM),
        BHND_MATCH_BOARD_TYPE(BCM94360X52C),
        BHND_MATCH_BOARD_REV(HWREV_ANY),
        BHND_MATCH_SROMREV(RANGE(0, 10))
};
```

A *bhnd_chip_match* match descriptor may be initialized using one or more of the following macros:

BHND_MATCH_CHIP_ID(*id*) Match on chips with an ID equal to BHND_CHIPID_ ## *id*

BHND_MATCH_CHIP_REV(*hwrev*) Match on chips with hardware revisions that match BHND_ ## *hwrev*.

BHND_MATCH_CHIP_PKG(*pkg*) Match on chips with a package ID equal to BHND_PKGID_ ## *pkg*

BHND_MATCH_CHIP_TYPE(*type*) Match on chips with a chip type equal to BHND_CHIPTYPE_ ## *type*

BHND_MATCH_CHIP_IP(*id*, *pkg*) A convenience wrapper for

BHND_MATCH_CHIP_ID() and
BHND_MATCH_CHIP_PKG().

BHND_MATCH_CHIP_IPR(*id*, *pkg*, *hwrev*) A convenience wrapper for **BHND_MATCH_CHIP_ID**(), **BHND_MATCH_CHIP_PKG**(), and **BHND_MATCH_CHIP_REV**().

BHND_MATCH_CHIP_IR(*id*, *hwrev*) A convenience wrapper for **BHND_MATCH_CHIP_ID**() and **BHND_MATCH_CHIP_REV**().

For example:

```
struct bhnd_chip_match chip_desc = {
        BHND_MATCH_CHIP_IP(BCM4329, BCM4329_289PIN),
        BHND_MATCH_CHIP_TYPE(SIBA)
};
```

A *bhnd_core_match* match descriptor may be initialized using one or more of the following macros:

BHND_MATCH_CORE_VENDOR(*vendor*) Match on cores with a vendor ID equal to *vendor*

BHND_MATCH_CORE_ID(*id*) Match on cores with a device ID equal to *id*

BHND_MATCH_CORE_REV(*hwrev*) Match on cores with hardware revisions that match BHND_ ## *hwrev*.

BHND_MATCH_CORE_CLASS(*class*) Match on cores with a core device class equal to *class*

BHND_MATCH_CORE_IDX(*idx*) Match on cores with a core index equal to *idx*

BHND_MATCH_CORE_UNIT(*unit*) Match on cores with a core unit equal to *unit*

BHND_MATCH_CORE(*vendor*, *id*) A convenience wrapper for **BHND_MATCH_CORE_VENDOR**() and **BHND_MATCH_CORE_ID**().

For example:

```
struct bhnd_core_match core_desc = {
        BHND_MATCH_CORE(BHND_MFGID_BROADCOM, BHND_COREID_CC),
        BHND_MATCH_CORE_REV(HWREV_RANGE(0, 10))
};
```

The *bhnd_device_match* match descriptor supports matching on all board, chip, and core attributes, and may be initialized using any of the *bhnd_board_match*, *bhnd_chip_match*, or *bhnd_core_match* macros.

For example:

```
struct bhnd_device_match device_desc = {
        BHND_MATCH_CHIP_IP(BCM4329, BCM4329_289PIN),
        BHND_MATCH_BOARD_VENDOR(BHND_MFGID_BROADCOM),
        BHND_MATCH_BOARD_TYPE(BCM94329AGB),
        BHND_MATCH_CORE(BHND_MFGID_BROADCOM, BHND_COREID_CC),
};
```

A *bhnd_hwrev_match* match descriptor may be initialized using one of the following macros:

BHND_HWREV_ANY	Matches any hardware revision.
BHND_HWREV_EQ(*hwrev*)	Matches any hardware revision equal to *hwrev*
BHND_HWREV_GTE(*hwrev*)	Matches any hardware revision greater than or equal to *hwrev*
BHND_HWREV_LTE(*hwrev*)	Matches any hardware revision less than or equal to *hwrev*
BHND_HWREV_RANGE(*start*, *end*)	Matches any hardware revision within an inclusive range. If BHND_HWREV_INVALID is specified as the *end* value, will match on any revision equal to or greater than *start*

Device Table Functions

The bhnd device table functions are used to query device and quirk tables.

The **bhnd_device_lookup**() function returns a pointer to the first entry in device table *table* that matches the device *dev*. The table entry size is specified by *entry_size*.

The **bhnd_device_quirks**() function scan the device table *table* for all quirk entries that match the device *dev*, returning the bitwise OR of all matching quirk flags. The table entry size is specified by *entry_size*.

The *bhnd_device* structure contains the following fields:

core A *bhnd_device_match* descriptor.
desc A verbose device description suitable for use with device_set_desc(9), or NULL.
quirks_table The quirks table for this device, or NULL.
device_flags
The device flags required when matching this entry.

The following device flags are supported:

BHND_DF_ANY	Match on any device.
BHND_DF_HOSTB	Match only if the device is the bhndb(4) host bridge. Implies BHND_DF_ADAPTER.
BHND_DF_SOC	Match only if the device is attached to a native SoC backplane.
BHND_DF_ADAPTER	Match only if the device is attached to a bhndb(4) bridged backplane.

A *bhnd_device* table entry may be initialized using one of the following macros:

BHND_DEVICE(*vendor*, *device*, *desc*, *quirks*, *flags*)
Match on devices with a vendor ID equal to BHND_MFGID_ ## *vendor* and a core device ID equal to BHND_COREID_ ## *device*.

The device's verbose description is specified by the *desc* argument, a pointer to the device-specific quirks table is specified by the *quirks* argument, and any required device flags may be provided in *flags*. The optional *flags* argument defaults to BHND_DF_ANY if omitted.

BHND_DEVICE_END
Terminate the *bhnd_device* table.

For example:

```
struct bhnd_device bhnd_usb11_devices[] = {
        BHND_DEVICE(BCM, USB, "Broadcom USB1.1 Controller",
          bhnd_usb11_quirks),
        BHND_DEVICE_END
};
```

The *bhnd_device_quirk* structure contains the following fields:
desc A *bhnd_device_match* descriptor.
quirks Applicable quirk flags.

A bhnd_device_quirk table entry may be initialized using one of the following convenience macros:

BHND_BOARD_QUIRK(*board*, *flags*) Set quirk flags *flags* on devices with a board type equal to BHND_BOARD_ ## *board*.

BHND_CHIP_QUIRK(*chip*, *hwrev*, *flags*) Set quirk flags *flags* on devices with a chip ID equal to BHND_CHIPID_BCM ## *chip* and chip hardware revision that matches BHND_ ## *hwrev*.

BHND_PKG_QUIRK(*chip*, *pkg*, *flags"*) Set quirk flags *flags* on devices with a chip ID equal to BHND_CHIPID_BCM ## *chip* and chip package equal to BHND_ ## chip ## *pkg*.

BHND_CORE_QUIRK(*hwrev*, *flags"*) Set quirk flags *flags* on devices with a core hardware revision that matches BHND_ ## *hwrev*.

For example:

```
struct bhnd_device_quirk bhnd_usb11_quirks[] = {
        BHND_DEVICE(BCM, USB, "Broadcom USB1.1 Controller",
          bhnd_usb11_quirks),
        BHND_DEVICE_END
};
```

DMA Address Translation Functions

The **bhnd_get_dma_translation**() function is used to request a DMA address translation descriptor suitable for use with a maximum DMA address width of *width*, with support for the requested translation *flags*.

If a suitable DMA address translation descriptor is found, it will be stored in *translation*, and a bus DMA tag specifying the DMA translation's address restrictions will be stored in *dmat*. The *translation* and *dmat* arguments may be NULL if the translation descriptor or DMA tag are not desired.

The following DMA translation flags are supported:

BHND_DMA_TRANSLATION_PHYSMAP
The translation remaps the device's physical address space.

This is used in conjunction with BHND_DMA_TRANSLATION_BYTESWAPPED to define a DMA translation that provides byteswapped access to physical memory on big-endian MIPS SoCs.

BHND_DMA_TRANSLATION_BYTESWAPPED

The translation provides a byte-swapped mapping; write requests will be byte-swapped before being written to memory, and read requests will be byte-swapped before being returned.

This is primarily used to perform efficient byte swapping of DMA data on embedded MIPS SoCs executing in big-endian mode.

The following symbolic constants are defined for common DMA address widths:

BHND_DMA_ADDR_30BIT 30-bit DMA
BHND_DMA_ADDR_32BIT 32-bit DMA
BHND_DMA_ADDR_64BIT 64-bit DMA

The *bhnd_dma_translation* structure contains the following fields:

base_addr Host-to-device physical address translation. This may be added to a host physical address to produce a device DMA address.

addr_mask Device-addressable address mask. This defines the device DMA address range, and excludes any bits reserved for mapping the address within the translation window at *base_addr*.

addrext_mask Device-addressable extended address mask. If a the per-core BHND DMA engine supports the 'addrext' control field, it can be used to provide address bits excluded by *addr_mask*.

Support for DMA extended address changes -- including coordination with the core providing device-to-host DMA address translation -- is handled transparently by the DMA engine.

For example, on PCI Wi-Fi devices, the Wi-Fi core's DMA engine will (in effect) update the PCI host bridge core's DMA sbtopcitranslation base address to map the target address prior to performing a DMA transaction.

flags Translation flags.

Interrupt Functions

The **bhnd_get_intr_count**() function is used to determine the number of backplane interrupt lines assigned to the device *dev*. Interrupt line identifiers are allocated in monotonically increasing order, starting with 0.

The **bhnd_get_intr_ivec**() function is used to determine the backplane interrupt vector assigned to interrupt line *intr* on the device *dev*, writing the result to *ivec*. Interrupt vector assignments are backplane-specific: On BCMA devices, this function returns the OOB bus line assigned to the interrupt. On SIBA devices, it returns the target OCP slave flag number assigned to the interrupt.

The **bhnd_map_intr**() function is used to map interrupt line *intr* assigned to device *dev* to an IRQ number, writing the result to *irq*. Until unmapped, this IRQ may be used when allocating a resource of type SYS_RES_IRQ.

Ownership of the interrupt mapping is assumed by the caller, and must be explicitly released using *bhnd_unmap_intr*.

The **bhnd_unmap_intr**() function is used to unmap bus IRQ *irq* previously mapped using **bhnd_map_intr**() by the device *dev*.

NVRAM Functions

The **bhnd_nvram_getvar**() function is used to read the value of NVRAM variable *name* from the NVRAM provider(s) registered with the parent bhnd(4) bus of device *dev*, coerced to the desired data representation *type*, written to the buffer specified by *buf*.

Before the call, the maximum capacity of *buf* is specified by *len*. After a successful call -- or if ENOMEM is returned -- the size of the available data will be written to *len*. The size of the desired data representation can be determined by calling **bhnd_nvram_getvar**() with a NULL argument for *buf*.

The following NVRAM data types are supported:

BHND_NVRAM_TYPE_UINT8	unsigned 8-bit integer
BHND_NVRAM_TYPE_UINT16	unsigned 16-bit integer
BHND_NVRAM_TYPE_UINT32	unsigned 32-bit integer
BHND_NVRAM_TYPE_UINT64	signed 64-bit integer
BHND_NVRAM_TYPE_INT8	signed 8-bit integer
BHND_NVRAM_TYPE_INT16	signed 16-bit integer
BHND_NVRAM_TYPE_INT32	signed 32-bit integer
BHND_NVRAM_TYPE_INT64	signed 64-bit integer
BHND_NVRAM_TYPE_CHAR	UTF-8 character
BHND_NVRAM_TYPE_STRING	UTF-8 NUL-terminated string
BHND_NVRAM_TYPE_BOOL	uint8 boolean value
BHND_NVRAM_TYPE_NULL	NULL (empty) value
BHND_NVRAM_TYPE_DATA	opaque octet string
BHND_NVRAM_TYPE_UINT8_ARRAY	array of uint8 integers

BHND_NVRAM_TYPE_UINT16_ARRAY	array of uint16 integers
BHND_NVRAM_TYPE_UINT32_ARRAY	array of uint32 integers
BHND_NVRAM_TYPE_UINT64_ARRAY	array of uint64 integers
BHND_NVRAM_TYPE_INT8_ARRAY	array of int8 integers
BHND_NVRAM_TYPE_INT16_ARRAY	array of int16 integers
BHND_NVRAM_TYPE_INT32_ARRAY	array of int32 integers
BHND_NVRAM_TYPE_INT64_ARRAY	array of int64 integers
BHND_NVRAM_TYPE_CHAR_ARRAY	array of UTF-8 characters
BHND_NVRAM_TYPE_STRING_ARRAY	array of UTF-8 NUL-terminated strings
BHND_NVRAM_TYPE_BOOL_ARRAY	array of uint8 boolean values

The **bhnd_nvram_getvar_array**(), **bhnd_nvram_getvar_int**(), **bhnd_nvram_getvar_int8**(), **bhnd_nvram_getvar_int16**(), **bhnd_nvram_getvar_int32**(), **bhnd_nvram_getvar_uint**(), **bhnd_nvram_getvar_uint8**(), **bhnd_nvram_getvar_uint16**(), **bhnd_nvram_getvar_uint32**(), and **bhnd_nvram_getvar_str**() functions are convenience wrappers for **bhnd_nvram_getvar**().

The **bhnd_nvram_getvar_array**() function returns either a value of exactly *size* in *buf*, or returns an error code of ENXIO if the data representation is not exactly *size* in length.

The **bhnd_nvram_getvar_int**() and **bhnd_nvram_getvar_uint**() functions return the value of NVRAM variable *name*, coerced to a signed or unsigned integer type of *width* (1, 2, or 4 bytes).

The **bhnd_nvram_getvar_int8**(), **bhnd_nvram_getvar_int16**(), **bhnd_nvram_getvar_int32**(), **bhnd_nvram_getvar_uint**(), **bhnd_nvram_getvar_uint8**(), **bhnd_nvram_getvar_uint16**(), and **bhnd_nvram_getvar_uint32**() functions return the value of NVRAM variable *name*, coerced to a signed or unsigned 8, 16, or 32-bit integer type.

The **bhnd_nvram_getvar_str**() functions return the value of NVRAM variable *name*, coerced to a NUL-terminated string.

The **bhnd_nvram_string_array_next**() function iterates over all strings in the *inp* BHND_NVRAM_TYPE_STRING_ARRAY value. The size of *inp*, including any terminating NUL character(s), is specified using the *ilen* argument. The *prev* argument should be either a string pointer previously returned by **bhnd_nvram_string_array_next**(), or NULL to begin iteration. If *prev is not* NULL, the *olen* argument must be a pointer to the length previously returned by **bhnd_nvram_string_array_next**(). On success, the next string element's length will be written to this pointer.

Port/Region Functions

Per-device interconnect memory mappings are identified by a combination of *port type*, *port number*, and *region number*. Port and memory region identifiers are allocated in monotonically increasing order for each *port type*, starting with 0.

The following port types are supported:

BHND_PORT_DEVICE Device memory. The device's control/status registers are always mapped by the first device port and region, and will be assigned a SYS_RES_MEMORY resource ID of 0.

BHND_PORT_BRIDGE
Bridge memory.

BHND_PORT_AGENT Interconnect agent/wrapper.

The **bhnd_decode_port_rid**() function is used to decode the resource ID *rid* assigned to device *dev*, of resource type *type*, writing the port type to *port_type*, port number to *port*, and region number to *region*.

The **bhnd_get_port_count**() function returns the number of ports of type *type* assigned to device *dev*.

The **bhnd_get_port_rid**() function returns the resource ID for the SYS_RES_MEMORY resource mapping the *port* of *type* and *region* on device *dev*, or -1 if the port or region are invalid, or do not have an assigned resource ID.

The **bhnd_get_region_addr**() function is used to determine the base address and size of the memory *region* on *port* of *type* assigned to *dev*. The region's base device address will be written to *region_addr*, and the region size to *region_size*.

The **bhnd_get_region_count**() function returns the number of memory regions mapped to *port* of *type* on device *dev*.

The **bhnd_is_region_valid**() function returns true if *region* is a valid region mapped by *port* of *type* on device *dev*.

Power Management Functions

Drivers must ask the parent bhnd(4) bus to allocate device PMU state using **bhnd_alloc_pmu**() before calling any another bhnd PMU functions.

The **bhnd_alloc_pmu**() function is used to allocate per-device PMU state and enable PMU request handling for device *dev*. The memory region containing the device's PMU register block must be

allocated using bus_alloc_resource(9) or **bhnd_alloc_resource**() before calling **bhnd_alloc_pmu**(), and must not be released until after calling **bhnd_release_pmu**().

On all supported BHND hardware, the PMU register block is mapped by the device's control/status registers in the first device port and region.

The **bhnd_release_pmu**() function releases the per-device PMU state previously allocated for device *dev* using **bhnd_alloc_pmu**(). Any outstanding clock and external resource requests will be discarded upon release of the device PMU state.

The **bhnd_enable_clocks**() function is used to request that *clocks* be powered up and routed to the backplane on behalf of device *dev*. This will power any clock sources required (e.g., XTAL, PLL, etc) and wait until the requested clocks are stable. If the request succeeds, any previous clock requests issued by *dev* will be discarded.

The following clocks are supported, and may be combined using bitwise OR to request multiple clocks:

BHND_CLOCK_DYN Dynamically select an appropriate clock source based on all outstanding clock requests by any device attached to the parent bhnd(4) bus.

BHND_CLOCK_ILP Idle Low-Power (ILP) Clock. May be used if no register access is required, or long request latency is acceptable.

BHND_CLOCK_ALP Active Low-Power (ALP) Clock. Supports low-latency register access and low-rate DMA.

BHND_CLOCK_HT High Throughput (HT) Clock. Supports high bus throughput and lowest-latency register access.

The **bhnd_request_clock**() function is used to request that *clock* (or faster) be powered up and routed to device *dev*.

The **bhnd_get_clock_freq**() function is used to request the current clock frequency of *clock*, writing the frequency in Hz to *freq*.

The **bhnd_get_clock_latency**() function is used to determine the transition latency required for *clock*, writing the latency in microseconds to *latency*. The BHND_CLOCK_HT latency value is suitable for use as the D11 Wi-Fi core *fastpwrup_dly* value.

The **bhnd_request_ext_rsrc**() function is used to request that the external PMU-managed resource

assigned to device *dev*, identified by device-specific identifier *rsrc*, be powered up.

The **bhnd_release_ext_rsrc**() function releases any outstanding requests by device *dev* for the PMU-managed resource identified by device-specific identifier *rsrc*. If an external resource is shared by multiple devices, it will not be powered down until all device requests are released.

Service Provider Functions

The **bhnd_register_provider**() function is used to register device *dev* as a provider for platform *service* with the parent bhnd(4) bus.

The following service types are supported:

BHND_SERVICE_CHIPC	ChipCommon service. The providing device must implement the bhnd_chipc interface.
BHND_SERVICE_PWRCTL	Legacy PWRCTL service. The providing device must implement the bhnd_pwrctl interface.
BHND_SERVICE_PMU	PMU service. The providing device must implement the bhnd_pmu interface.
BHND_SERVICE_NVRAM	NVRAM service. The providing device must implement the bhnd_nvram interface.
BHND_SERVICE_GPIO	GPIO service. The providing device must implement the standard gpio(4) interface.
BHND_SERVICE_ANY	Matches on any service type. May be used with **bhnd_deregister_provider**() to remove all service provider registrations for a device.

The **bhnd_deregister_provider**() function attempts to remove provider registration for the device *dev* and *service*. If a *service* argument of BHND_SERVICE_ANY is specified, this function will attempt to remove *all service provider registrations for dev*.

The **bhnd_retain_provider**() function retains and returns a reference to the provider registered for *service* with the parent bhnd(4) bus of devce *dev*, if available. On success, the caller is responsible for releasing this provider reference using **bhnd_release_provider**(). The service provider is guaranteed to remain available until the provider reference is released.

The **bhnd_release_provider**() function releases a reference to a *provider* for *service*, previously retained by device *dev* using **bhnd_retain_provider**().

Utility Functions

The **bhnd_driver_get_erom_class**() function returns the bhnd_erom(9) class for the device enumeration table format used by bhnd(4) bus driver instance *driver*. If the driver does not support bhnd_erom(9) device enumeration, NULL is returned.

The **bhnd_find_core_class**() function looks up the BHND class, if known, for the BHND vendor ID *vendor* and device ID *device*.

The **bhnd_find_core_name**() function is used to fetch the human-readable name, if known, for the BHND core with a vendor ID of *vendor* and device ID of *device*.

The **bhnd_core_class**() and **bhnd_core_name**() functions are convenience wrappers for **bhnd_find_core_class**() and **bhnd_find_core_name**(), that use the *vendor* and *device* fields of the core info structure *ci*.

The **bhnd_format_chip_id**() function writes a NUL-terminated human-readable representation of the BHND *chip_id* value to the specified *buffer* with a capacity of *size*. No more than *size-1* characters will be written, with the *size'th* character set to '\0'. A buffer size of BHND_CHIPID_MAX_NAMELEN is sufficient for any string representation produced using **bhnd_format_chip_id**().

The **bhnd_set_custom_core_desc**() function uses the bhnd(4) device identification of *dev*, overriding the core name with the specified *dev_name*, to populate the device's verbose description using device_set_desc(9).

The **bhnd_set_default_core_desc**() function uses the bhnd(4) device identification of *dev* to populate the device's verbose description using device_set_desc(9).

The **bhnd_vendor_name**() function returns the human-readable name for the JEP-106, ARM 4-bit continuation encoded manufacturer ID *vendor*, if known.

RETURN VALUES

Bus Resource Functions

The **bhnd_activate_resource**(), **bhnd_alloc_resources**(), **bhnd_deactivate_resource**(), and **bhnd_release_resource**() functions return 0 on success, otherwise an appropriate error code is returned.

The **bhnd_alloc_resource**() and **bhnd_alloc_resource_any**() functions return a pointer to *struct resource* on success, a null pointer otherwise.

Device Configuration Functions

The **bhnd_read_config**() and **bhnd_write_config**() functions return 0 on success, or one of the following values on error:

[EINVAL]	The device is not a direct child of the bhnd(4) bus
[EINVAL]	The requested width is not one of 1, 2, or 4 bytes.
[ENODEV]	Accessing agent/config space for the device is unsupported.
[EFAULT]	The requested offset or width exceeds the bounds of the mapped agent/config space.

The **bhnd_read_ioctl**(), **bhnd_write_ioctl**(), **bhnd_read_iost**(), **bhnd_reset_hw**(), and **bhnd_suspend_hw**() functions return 0 on success, otherwise an appropriate error code is returned.

Device Information Functions

The **bhnd_read_board_info**() function returns 0 on success, otherwise an appropriate error code is returned.

DMA Address Translation Functions

The **bhnd_get_dma_translation**() function returns 0 on success, or one of the following values on error:

[ENODEV]	DMA is not supported.
[ENOENT]	No DMA translation matching the requested address width and translation flags is available.

If fetching the requested DMA address translation otherwise fails, an appropriate error code will be returned.

Interrupt Functions

The **bhnd_get_intr_ivec**() function returns 0 on success, or ENXIO if the requested interrupt line exceeds the number of interrupt lines assigned to the device.

The **bhnd_map_intr**() function returns 0 on success, otherwise an appropriate error code is returned.

NVRAM Functions

The **bhnd_nvram_getvar**(), **bhnd_nvram_getvar_array**(), **bhnd_nvram_getvar_int**(), **bhnd_nvram_getvar_int8**(), **bhnd_nvram_getvar_int16**(), **bhnd_nvram_getvar_int32**(),

bhnd_nvram_getvar_uint(), **bhnd_nvram_getvar_uint8**(), **bhnd_nvram_getvar_uint16**(), and **bhnd_nvram_getvar_uint32**() functions return 0 on success, or one of the following values on error:

[ENODEV] If an NVRAM provider has not been registered with the bus.

[ENOENT] The requested variable was not found.

[ENOMEM] If the buffer of size is too small to hold the requested value.

[EOPNOTSUPP] If the value's native type is incompatible with and cannot be coerced to the requested type.

[ERANGE] If value coercion would overflow (or underflow) the requested type

If reading the variable otherwise fails, an appropriate error code will be returned.

Port/Region Functions

The **bhnd_decode_port_rid**() function returns 0 on success, or an appropriate error code if no matching port/region is found.

The **bhnd_get_port_rid**() function returns the resource ID for the requested port and region, or -1 if the port or region are invalid, or do not have an assigned resource ID.

The **bhnd_get_region_addr**() function returns 0 on success, or an appropriate error code if no matching port/region is found.

PMU Functions

The **bhnd_alloc_pmu**() function returns 0 on success, otherwise an appropriate error code is returned.

The **bhnd_release_pmu**() function returns 0 on success, otherwise an appropriate error code is returned, and the core state will be left unmodified.

The **bhnd_enable_clocks**() and **bhnd_request_clock**() functions return 0 on success, or one of the following values on error:

[ENODEV] An unsupported clock was requested.

[ENXIO] No PMU or PWRCTL provider has been registered with the bus.

The **bhnd_get_clock_freq**() function returns 0 on success, or ENODEV if the frequency for the specified

clock is not available.

The **bhnd_get_clock_latency**() function returns 0 on success, or ENODEV if the transition latency for the specified clock is not available.

The **bhnd_request_ext_rsrc**() and **bhnd_release_ext_rsrc**() functions return 0 on success, otherwise an appropriate error code is returned.

Service Provider Functions

The **bhnd_register_provider**() function returns 0 on success, EEXIST if an entry for service already exists, or an appropriate error code if service registration otherwise fails.

The **bhnd_deregister_provider**() function returns 0 on success, or EBUSY if active references to the service provider exist.

The **bhnd_retain_provider**() function returns a pointer to *device_t* on success, a null pointer if the requested provider is not registered.

Utility Functions

The **bhnd_format_chip_id**() function returns the total number of bytes written on success, or a negative integer on failure.

SEE ALSO

bhnd(4), bhnd_erom(9)

AUTHORS

The **bhnd** driver programming interface and this manual page were written by Landon Fuller <*landonf@FreeBSD.org*>.