NAME

bus_space, **bus_space_barrier**, **bus_space_copy_region_1**, **bus_space_copy_region_2**, **bus_space_copy_region_4**, **bus_space_copy_region_8**, **bus_space_copy_region_stream_1**, **bus_space_copy_region_stream_2**, **bus_space_copy_region_stream_4**, **bus_space_copy_region_stream_8**, **bus_space_free**, **bus_space_map**, **bus_space_peek_1**, **bus_space_peek_2**, **bus_space_peek_4**, **bus_space_peek_8**, **bus_space_poke_1**, **bus_space_poke_2**, **bus_space_poke_4**, **bus_space_poke_8**, **bus_space_read_1**, **bus_space_read_2**, **bus_space_read_4**, **bus_space_read_8**, **bus_space_read_multi_1**, **bus_space_read_multi_2**, **bus_space_read_multi_4**, **bus_space_read_multi_8**, **bus_space_read_multi_stream_1**, **bus_space_read_multi_stream_2**, **bus_space_read_multi_stream_4**, **bus_space_read_multi_stream_8**, **bus_space_read_region_1**, **bus_space_read_region_2**, **bus_space_read_region_4**, **bus_space_read_region_8**, **bus_space_read_region_stream_1**, **bus_space_read_region_stream_2**, **bus_space_read_region_stream_4**, **bus_space_read_region_stream_8**, **bus_space_read_stream_1**, **bus_space_read_stream_2**, **bus_space_read_stream_4**, **bus_space_read_stream_8**, **bus_space_set_multi_1**, **bus_space_set_multi_2**, **bus_space_set_multi_4**, **bus_space_set_multi_8**, **bus_space_set_multi_stream_1**, **bus_space_set_multi_stream_2**, **bus_space_set_multi_stream_4**, **bus_space_set_multi_stream_8**, **bus_space_set_region_1**, **bus_space_set_region_2**, **bus_space_set_region_4**, **bus_space_set_region_8**, **bus_space_set_region_stream_1**, **bus_space_set_region_stream_2**, **bus_space_set_region_stream_4**, **bus_space_set_region_stream_8**, **bus_space_subregion**, **bus_space_unmap**, **bus_space_write_1**, **bus_space_write_2**, **bus_space_write_4**, **bus_space_write_8**, **bus_space_write_multi_1**, **bus_space_write_multi_2**, **bus_space_write_multi_4**, **bus_space_write_multi_8**, **bus_space_write_multi_stream_1**, **bus_space_write_multi_stream_2**, **bus_space_write_multi_stream_4**, **bus_space_write_multi_stream_8**, **bus_space_write_region_1**, **bus_space_write_region_2**, **bus_space_write_region_4**, **bus_space_write_region_8**, **bus_space_write_region_stream_1**, **bus_space_write_region_stream_2**, **bus_space_write_region_stream_4**, **bus_space_write_region_stream_8**, **bus_space_write_stream_1**, **bus_space_write_stream_2**, **bus_space_write_stream_4**, **bus_space_write_stream_8** - bus space manipulation functions

SYNOPSIS

#include <machine/bus.h>

int
bus_space_map(*bus_space_tag_t space*, *bus_addr_t address*, *bus_size_t size*, *int flags*, *bus_space_handle_t *handlep*);

void
bus_space_unmap(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t size*);

int
bus_space_subregion(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*,

bus_size_t size, *bus_space_handle_t *nhandlep*);

int
bus_space_alloc(*bus_space_tag_t space*, *bus_addr_t reg_start*, *bus_addr_t reg_end*, *bus_size_t size*, *bus_size_t alignment*, *bus_size_t boundary*, *int flags*, *bus_addr_t *addrp*, *bus_space_handle_t *handlep*);

void
bus_space_free(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t size*);

int
bus_space_peek_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*);

int
bus_space_peek_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*);

int
bus_space_peek_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*);

int
bus_space_peek_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*);

int
bus_space_poke_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*);

int
bus_space_poke_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*);

int
bus_space_poke_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*);

int
bus_space_poke_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*,

*uint8_t *datap*);

uint8_t
bus_space_read_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*);

uint16_t
bus_space_read_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*);

uint32_t
bus_space_read_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*);

uint64_t
bus_space_read_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*);

uint8_t
bus_space_read_stream_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*);

uint16_t
bus_space_read_stream_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*);

uint32_t
bus_space_read_stream_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*);

uint64_t
bus_space_read_stream_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*);

void
bus_space_write_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*,
uint8_t value);

void
bus_space_write_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*,
uint16_t value);

void
bus_space_write_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*,
uint32_t value);

void
bus_space_write_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*,

uint64_t value);

void
bus_space_write_stream_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t value*);

void
bus_space_write_stream_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t value*);

void
bus_space_write_stream_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t value*);

void
bus_space_write_stream_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t value*);

void
bus_space_barrier(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *bus_size_t length*, *int flags*);

void
bus_space_read_region_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*, *bus_size_t count*);

void
bus_space_read_region_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*, *bus_size_t count*);

void
bus_space_read_region_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t *datap*, *bus_size_t count*);

void
bus_space_read_region_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t *datap*, *bus_size_t count*);

void
bus_space_read_region_stream_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*,

*uint8_t *datap*, *bus_size_t count*);

void
bus_space_read_region_stream_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*, *bus_size_t count*);

void
bus_space_read_region_stream_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t *datap*, *bus_size_t count*);

void
bus_space_read_region_stream_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t *datap*, *bus_size_t count*);

void
bus_space_write_region_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*, *bus_size_t count*);

void
bus_space_write_region_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*, *bus_size_t count*);

void
bus_space_write_region_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t *datap*, *bus_size_t count*);

void
bus_space_write_region_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t *datap*, *bus_size_t count*);

void
bus_space_write_region_stream_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*, *bus_size_t count*);

void
bus_space_write_region_stream_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*, *bus_size_t count*);

void
bus_space_write_region_stream_4(*bus_space_tag_t space*, *bus_space_handle_t handle*,

*bus_size_t offset, uint32_t *datap, bus_size_t count*);

void
bus_space_write_region_stream_8(*bus_space_tag_t space, bus_space_handle_t handle, bus_size_t offset, uint64_t *datap, bus_size_t count*);

void
bus_space_copy_region_1(*bus_space_tag_t space, bus_space_handle_t srchandle, bus_size_t srcoffset, bus_space_handle_t dsthandle, bus_size_t dstoffset, bus_size_t count*);

void
bus_space_copy_region_2(*bus_space_tag_t space, bus_space_handle_t srchandle, bus_size_t srcoffset, bus_space_handle_t dsthandle, bus_size_t dstoffset, bus_size_t count*);

void
bus_space_copy_region_4(*bus_space_tag_t space, bus_space_handle_t srchandle, bus_size_t srcoffset, bus_space_handle_t dsthandle, bus_size_t dstoffset, bus_size_t count*);

void
bus_space_copy_region_8(*bus_space_tag_t space, bus_space_handle_t srchandle, bus_size_t srcoffset, bus_space_handle_t dsthandle, bus_size_t dstoffset, bus_size_t count*);

void
bus_space_copy_region_stream_1(*bus_space_tag_t space, bus_space_handle_t srchandle, bus_size_t srcoffset, bus_space_handle_t dsthandle, bus_size_t dstoffset, bus_size_t count*);

void
bus_space_copy_region_stream_2(*bus_space_tag_t space, bus_space_handle_t srchandle, bus_size_t srcoffset, bus_space_handle_t dsthandle, bus_size_t dstoffset, bus_size_t count*);

void
bus_space_copy_region_stream_4(*bus_space_tag_t space, bus_space_handle_t srchandle, bus_size_t srcoffset, bus_space_handle_t dsthandle, bus_size_t dstoffset, bus_size_t count*);

void
bus_space_copy_region_stream_8(*bus_space_tag_t space, bus_space_handle_t srchandle, bus_size_t srcoffset, bus_space_handle_t dsthandle, bus_size_t dstoffset, bus_size_t count*);

void
bus_space_set_region_1(*bus_space_tag_t space, bus_space_handle_t handle, bus_size_t offset,*

uint8_t value, *bus_size_t count*);

void
bus_space_set_region_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t value*, *bus_size_t count*);

void
bus_space_set_region_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t value*, *bus_size_t count*);

void
bus_space_set_region_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t value*, *bus_size_t count*);

void
bus_space_set_region_stream_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t value*, *bus_size_t count*);

void
bus_space_set_region_stream_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t value*, *bus_size_t count*);

void
bus_space_set_region_stream_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t value*, *bus_size_t count*);

void
bus_space_set_region_stream_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t value*, *bus_size_t count*);

void
bus_space_read_multi_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*,

*uint32_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_stream_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_stream_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_stream_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t *datap*, *bus_size_t count*);

void
bus_space_read_multi_stream_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_stream_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*,

*uint8_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_stream_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_stream_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t *datap*, *bus_size_t count*);

void
bus_space_write_multi_stream_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t *datap*, *bus_size_t count*);

void
bus_space_set_multi_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t value*, *bus_size_t count*);

void
bus_space_set_multi_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t value*, *bus_size_t count*);

void
bus_space_set_multi_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint32_t value*, *bus_size_t count*);

void
bus_space_set_multi_8(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint64_t value*, *bus_size_t count*);

void
bus_space_set_multi_stream_1(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint8_t value*, *bus_size_t count*);

void
bus_space_set_multi_stream_2(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*, *uint16_t value*, *bus_size_t count*);

void
bus_space_set_multi_stream_4(*bus_space_tag_t space*, *bus_space_handle_t handle*, *bus_size_t offset*,

uint32_t value, bus_size_t count);

void
bus_space_set_multi_stream_8(*bus_space_tag_t space, bus_space_handle_t handle, bus_size_t offset, uint64_t value, bus_size_t count*);

DESCRIPTION

The **bus_space** functions exist to allow device drivers machine-independent access to bus memory and register areas. All of the functions and types described in this document can be used by including the <*machine/bus.h*> header file.

Many common devices are used on multiple architectures, but are accessed differently on each because of architectural constraints. For instance, a device which is mapped in one system's I/O space may be mapped in memory space on a second system. On a third system, architectural limitations might change the way registers need to be accessed (e.g. creating a non-linear register space). In some cases, a single driver may need to access the same type of device in multiple ways in a single system or architecture. The goal of the **bus_space** functions is to allow a single driver source file to manipulate a set of devices on different system architectures, and to allow a single driver object file to manipulate a set of devices on multiple bus types on a single architecture.

Not all buses have to implement all functions described in this document, though that is encouraged if the operations are logically supported by the bus. Unimplemented functions should cause compile-time errors if possible.

All of the interface definitions described in this document are shown as function prototypes and discussed as if they were required to be functions. Implementations are encouraged to implement prototyped (type-checked) versions of these interfaces, but may implement them as macros if appropriate. Machine-dependent types, variables, and functions should be marked clearly in <*machine/bus.h*> to avoid confusion with the machine-independent types and functions, and, if possible, should be given names which make the machine-dependence clear.

CONCEPTS AND GUIDELINES

Bus spaces are described by bus space tags, which can be created only by machine-dependent code. A given machine may have several different types of bus space (e.g. memory space and I/O space), and thus may provide multiple different bus space tags. Individual buses or devices on a machine may use more than one bus space tag. For instance, ISA devices are given an ISA memory space tag and an ISA I/O space tag. Architectures may have several different tags which represent the same type of space, for instance because of multiple different host bus interface chipsets.

A range in bus space is described by a bus address and a bus size. The bus address describes the start of

the range in bus space. The bus size describes the size of the range in bytes. Buses which are not byte addressable may require use of bus space ranges with appropriately aligned addresses and properly rounded sizes.

Access to regions of bus space is facilitated by use of bus space handles, which are usually created by mapping a specific range of a bus space. Handles may also be created by allocating and mapping a range of bus space, the actual location of which is picked by the implementation within bounds specified by the caller of the allocation function.

All of the bus space access functions require one bus space tag argument, at least one handle argument, and at least one offset argument (a bus size). The bus space tag specifies the space, each handle specifies a region in the space, and each offset specifies the offset into the region of the actual location(s) to be accessed. Offsets are given in bytes, though buses may impose alignment constraints. The offset used to access data relative to a given handle must be such that all of the data being accessed is in the mapped region that the handle describes. Trying to access data outside that region is an error.

Because some architectures' memory systems use buffering to improve memory and device access performance, there is a mechanism which can be used to create "barriers" in the bus space read and write stream. There are three types of barriers: read, write, and read/write. All reads started to the region before a read barrier must complete before any reads after the read barrier are started. (The analogous requirement is true for write barriers.) Read/write barriers force all reads and writes started before the barrier to complete before any reads or writes after the barrier are started. Correctly-written drivers will include all appropriate barriers, and assume only the read/write ordering imposed by the barrier operations.

People trying to write portable drivers with the **bus_space** functions should try to make minimal assumptions about what the system allows. In particular, they should expect that the system requires bus space addresses being accessed to be naturally aligned (i.e., base address of handle added to offset is a multiple of the access size), and that the system does alignment checking on pointers (i.e., pointer to objects being read and written must point to properly-aligned data).

The descriptions of the **bus_space** functions given below all assume that they are called with proper arguments. If called with invalid arguments or arguments that are out of range (e.g. trying to access data outside of the region mapped when a given handle was created), undefined behaviour results. In that case, they may cause the system to halt, either intentionally (via panic) or unintentionally (by causing a fatal trap of by some other means) or may cause improper operation which is not immediately fatal. Functions which return *void* or which return data read from bus space (i.e., functions which do not obviously return an error code) do not fail. They could only fail if given invalid arguments, and in that case their behaviour is undefined. Functions which take a count of bytes have undefined results if the specified *count* is zero.

TYPES

Several types are defined in <*machine/bus.h*> to facilitate use of the **bus_space** functions by drivers.

bus_addr_t

The *bus_addr_t* type is used to describe bus addresses. It must be an unsigned integral type capable of holding the largest bus address usable by the architecture. This type is primarily used when mapping and unmapping bus space.

bus_size_t

The *bus_size_t* type is used to describe sizes of ranges in bus space. It must be an unsigned integral type capable of holding the size of the largest bus address range usable on the architecture. This type is used by virtually all of the **bus_space** functions, describing sizes when mapping regions and offsets into regions when performing space access operations.

bus_space_tag_t

The *bus_space_tag_t* type is used to describe a particular bus space on a machine. Its contents are machine-dependent and should be considered opaque by machine-independent code. This type is used by all **bus_space** functions to name the space on which they are operating.

bus_space_handle_t

The *bus_space_handle_t* type is used to describe a mapping of a range of bus space. Its contents are machine-dependent and should be considered opaque by machine-independent code. This type is used when performing bus space access operations.

MAPPING AND UNMAPPING BUS SPACE

This section is specific to the NetBSD version of these functions and may or may not apply to the FreeBSD version.

Bus space must be mapped before it can be used, and should be unmapped when it is no longer needed. The **bus_space_map**() and **bus_space_unmap**() functions provide these capabilities.

Some drivers need to be able to pass a subregion of already-mapped bus space to another driver or module within a driver. The **bus_space_subregion**() function allows such subregions to be created.

bus_space_map(*space***,** *address***,** *size***,** *flags***,** *handlep***)**

The **bus_space_map**() function maps the region of bus space named by the *space*, *address*, and *size* arguments. If successful, it returns zero and fills in the bus space handle pointed to by *handlep* with the handle that can be used to access the mapped region. If unsuccessful, it will return non-zero and leave the bus space handle pointed to by *handlep* in an undefined state.

The *flags* argument controls how the space is to be mapped. Supported flags include:

BUS_SPACE_MAP_CACHEABLE Try to map the space so that accesses can be cached and/or prefetched by the system. If this flag is not specified, the implementation should map the space so that it will not be cached or prefetched.

This flag must have a value of 1 on all implementations for backward compatibility.

BUS_SPACE_MAP_LINEAR Try to map the space so that its contents can be accessed linearly via normal memory access methods (e.g. pointer dereferencing and structure accesses). This is useful when software wants to do direct access to a memory device, e.g. a frame buffer. If this flag is specified and linear mapping is not possible, the **bus_space_map**() call should fail. If this flag is not specified, the system may map the space in whatever way is most convenient.

BUS_SPACE_MAP_NONPOSTED Try to map the space using non-posted device memory. This is to support buses and devices where mapping with posted device memory is unsupported or broken. This flag is currently only available on arm64.

Not all combinations of flags make sense or are supported with all spaces. For instance, BUS_SPACE_MAP_CACHEABLE may be meaningless when used on many systems' I/O port spaces, and on some systems BUS_SPACE_MAP_LINEAR without BUS_SPACE_MAP_CACHEABLE may never work. When the system hardware or firmware provides hints as to how spaces should be mapped (e.g. the PCI memory mapping registers' "prefetchable" bit), those hints should be followed for maximum compatibility. On some systems, requesting a mapping that cannot be satisfied (e.g. requesting a non-cacheable mapping when the system can only provide a cacheable one) will cause the request to fail.

Some implementations may keep track of use of bus space for some or all bus spaces and refuse to allow duplicate allocations. This is encouraged for bus spaces which have no notion of slot-specific space addressing, such as ISA, and for spaces which coexist with those spaces (e.g. PCI memory and I/O spaces co-existing with ISA memory and I/O spaces).

Mapped regions may contain areas for which there is no device on the bus. If space in those areas is accessed, the results are bus-dependent.

bus_space_unmap(*space***,** *handle***,** *size***)**

The **bus_space_unmap**() function unmaps a region of bus space mapped with **bus_space_map**(). When unmapping a region, the *size* specified should be the same as the size given to **bus_space_map**() when mapping that region.

After **bus_space_unmap**() is called on a handle, that handle is no longer valid. (If copies were made of the handle they are no longer valid, either.)

This function will never fail. If it would fail (e.g. because of an argument error), that indicates a software bug which should cause a panic. In that case, **bus_space_unmap**() will never return.

bus_space_subregion(*space***,** *handle***,** *offset***,** *size***,** *nhandlep***)**

The **bus_space_subregion**() function is a convenience function which makes a new handle to some subregion of an already-mapped region of bus space. The subregion described by the new handle starts at byte offset *offset* into the region described by *handle*, with the size give by *size*, and must be wholly contained within the original region.

If successful, **bus_space_subregion**() returns zero and fills in the bus space handle pointed to by *nhandlep*. If unsuccessful, it returns non-zero and leaves the bus space handle pointed to by *nhandlep* in an undefined state. In either case, the handle described by *handle* remains valid and is unmodified.

When done with a handle created by **bus_space_subregion**(), the handle should be thrown away. Under no circumstances should **bus_space_unmap**() be used on the handle. Doing so may confuse any resource management being done on the space, and will result in undefined behaviour. When **bus_space_unmap**() or **bus_space_free**() is called on a handle, all subregions of that handle become invalid.

ALLOCATING AND FREEING BUS SPACE

This section is specific to the NetBSD version of these functions and may or may not apply to the FreeBSD version.

Some devices require or allow bus space to be allocated by the operating system for device use. When the devices no longer need the space, the operating system should free it for use by other devices. The **bus_space_alloc**() and **bus_space_free**() functions provide these capabilities.

bus_space_alloc(*space***,** *reg_start***,** *reg_end***,** *size***,** *alignment***,** *boundary***,** *flags***,** *addrp***,** *handlep***)**

The **bus_space_alloc**() function allocates and maps a region of bus space with the size given by *size*, corresponding to the given constraints. If successful, it returns zero, fills in the bus address pointed to by *addrp* with the bus space address of the allocated region, and fills in the bus space handle pointed to by *handlep* with the handle that can be used to access that region. If unsuccessful, it returns non-zero

and leaves the bus address pointed to by *addrp* and the bus space handle pointed to by *handlep* in an undefined state.

Constraints on the allocation are given by the *reg_start*, *reg_end*, *alignment*, and *boundary* parameters. The allocated region will start at or after *reg_start* and end before or at *reg_end*. The *alignment* constraint must be a power of two, and the allocated region will start at an address that is an even multiple of that power of two. The *boundary* constraint, if non-zero, ensures that the region is allocated so that *first address in region* / *boundary* has the same value as *last address in region* / *boundary*. If the constraints cannot be met, **bus_space_alloc**() will fail. It is an error to specify a set of constraints that can never be met (for example, *size* greater than *boundary*).

The *flags* parameter is the same as the like-named parameter to **bus_space_map**(), the same flag values should be used, and they have the same meanings.

Handles created by **bus_space_alloc**() should only be freed with **bus_space_free**(). Trying to use **bus_space_unmap**() on them causes undefined behaviour. The **bus_space_subregion**() function can be used on handles created by **bus_space_alloc**().

bus_space_free(*space***,** *handle***,** *size***)**

The **bus_space_free**() function unmaps and frees a region of bus space mapped and allocated with **bus_space_alloc**(). When unmapping a region, the *size* specified should be the same as the size given to **bus_space_alloc**() when allocating the region.

After **bus_space_free**() is called on a handle, that handle is no longer valid. (If copies were made of the handle, they are no longer valid, either.)

This function will never fail. If it would fail (e.g. because of an argument error), that indicates a software bug which should cause a panic. In that case, **bus_space_free**() will never return.

READING AND WRITING SINGLE DATA ITEMS

The simplest way to access bus space is to read or write a single data item. The **bus_space_read_N**() and **bus_space_write_N**() families of functions provide the ability to read and write 1, 2, 4, and 8 byte data items on buses which support those access sizes.

bus_space_read_1(*space***,** *handle***,** *offset***)**

bus_space_read_2(*space***,** *handle***,** *offset***)**

bus_space_read_4(*space***,** *handle***,** *offset***)**

bus_space_read_8(*space***,** *handle***,** *offset***)**
The **bus_space_read_N**() family of functions reads a 1, 2, 4, or 8 byte data item from the offset specified by *offset* into the region specified by *handle* of the bus space specified by *space*. The location being read must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data item being read. On some systems, not obeying this requirement may cause incorrect data to be read, on others it may cause a system crash.

Read operations done by the **bus_space_read_N**() functions may be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function.

These functions will never fail. If they would fail (e.g. because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

bus_space_write_1(*space***,** *handle***,** *offset***,** *value***)**

bus_space_write_2(*space***,** *handle***,** *offset***,** *value***)**

bus_space_write_4(*space***,** *handle***,** *offset***,** *value***)**

bus_space_write_8(*space***,** *handle***,** *offset***,** *value***)**
The **bus_space_write_N**() family of functions writes a 1, 2, 4, or 8 byte data item to the offset specified by *offset* into the region specified by *handle* of the bus space specified by *space*. The location being written must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data item being written. On some systems, not obeying this requirement may cause incorrect data to be written, on others it may cause a system crash.

Write operations done by the **bus_space_write_N**() functions may be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function.

These functions will never fail. If they would fail (e.g. because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

PROBING BUS SPACE FOR HARDWARE WHICH MAY NOT RESPOND

One problem with the **bus_space_read_N**() and **bus_space_write_N**() family of functions is that they

provide no protection against exceptions which can occur when no physical hardware or device responds to the read or write cycles. In such a situation, the system typically would panic due to a kernel-mode bus error. The **bus_space_peek_N**() and **bus_space_poke_N**() family of functions provide a mechanism to handle these exceptions gracefully without the risk of crashing the system.

As with **bus_space_read_N**() and **bus_space_write_N**(), the peek and poke functions provide the ability to read and write 1, 2, 4, and 8 byte data items on busses which support those access sizes. All of the constraints specified in the descriptions of the **bus_space_read_N**() and **bus_space_write_N**() functions also apply to **bus_space_peek_N**() and **bus_space_poke_N**().

In addition, explicit calls to the **bus_space_barrier**() function are not required as the implementation will ensure all pending operations complete before the peek or poke operation starts. The implementation will also ensure that the peek or poke operations complete before returning.

The return value indicates the outcome of the peek or poke operation. A return value of zero implies that a hardware device is responding to the operation at the specified offset in the bus space. A non-zero return value indicates that the kernel intercepted a hardware exception (e.g., bus error) when the peek or poke operation was attempted. Note that some busses are incapable of generating exceptions when non-existent hardware is accessed. In such cases, these functions will always return zero and the value of the data read by **bus_space_peek_N**() will be unspecified.

Finally, it should be noted that at this time the **bus_space_peek_N**() and **bus_space_poke_N**() functions are not re-entrant and should not, therefore, be used from within an interrupt service routine. This constraint may be removed at some point in the future.

bus_space_peek_1(*space*, *handle*, *offset*, *datap*)
bus_space_peek_2(*space*, *handle*, *offset*, *datap*)
bus_space_peek_4(*space*, *handle*, *offset*, *datap*)
bus_space_peek_8(*space*, *handle*, *offset*, *datap*)

The **bus_space_peek_N**() family of functions cautiously read a 1, 2, 4, or 8 byte data item from the offset specified by *offset* in the region specified by *handle* of the bus space specified by *space*. The data item read is stored in the location pointed to by *datap*. It is permissible for *datap* to be NULL, in which case the data item will be discarded after being read.

bus_space_poke_1(*space*, *handle*, *offset*, *value*)
bus_space_poke_2(*space*, *handle*, *offset*, *value*)
bus_space_poke_4(*space*, *handle*, *offset*, *value*)
bus_space_poke_8(*space*, *handle*, *offset*, *value*)

The **bus_space_poke_N**() family of functions cautiously write a 1, 2, 4, or 8 byte data item specified by *value* to the offset specified by *offset* in the region specified by *handle* of the bus space specified by *space*.

BARRIERS

In order to allow high-performance buffering implementations to avoid bus activity on every operation, read and write ordering should be specified explicitly by drivers when necessary. The **bus_space_barrier**() function provides that ability.

bus_space_barrier(*space***,** *handle***,** *offset***,** *length***,** *flags***)**

The **bus_space_barrier**() function enforces ordering of bus space read and write operations for the specified subregion (described by the *offset* and *length* parameters) of the region named by *handle* in the space named by *space*.

The *flags* argument controls what types of operations are to be ordered. Supported flags are:

BUS_SPACE_BARRIER_READ Synchronize read operations.

BUS_SPACE_BARRIER_WRITE Synchronize write operations.

Those flags can be combined (or-ed together) to enforce ordering on both read and write operations.

All of the specified type(s) of operation which are done to the region before the barrier operation are guaranteed to complete before any of the specified type(s) of operation done after the barrier.

Example: Consider a hypothetical device with two single-byte ports, one write-only input port (at offset 0) and a read-only output port (at offset 1). Operation of the device is as follows: data bytes are written to the input port, and are placed by the device on a stack, the top of which is read by reading from the output port. The sequence to correctly write two data bytes to the device then read those two data bytes back would be:

```
/*
 * t and h are the tag and handle for the mapped device's
 * space.
 */
bus_space_write_1(t, h, 0, data0);
bus_space_barrier(t, h, 0, 1, BUS_SPACE_BARRIER_WRITE);  /* 1 */
bus_space_write_1(t, h, 0, data1);
bus_space_barrier(t, h, 0, 2,
   BUS_SPACE_BARRIER_READ|BUS_SPACE_BARRIER_WRITE);     /* 2 */
```

```
ndata1 = bus_space_read_1(t, h, 1);
bus_space_barrier(t, h, 1, 1, BUS_SPACE_BARRIER_READ);   /* 3 */
ndata0 = bus_space_read_1(t, h, 1);
/* data0 == ndata0, data1 == ndata1 */
```

The first barrier makes sure that the first write finishes before the second write is issued, so that two writes to the input port are done in order and are not collapsed into a single write. This ensures that the data bytes are written to the device correctly and in order.

The second barrier makes sure that the writes to the output port finish before any of the reads to the input port are issued, thereby making sure that all of the writes are finished before data is read. This ensures that the first byte read from the device really is the last one that was written.

The third barrier makes sure that the first read finishes before the second read is issued, ensuring that data is read correctly and in order.

The barriers in the example above are specified to cover the absolute minimum number of bus space locations. It is correct (and often easier) to make barrier operations cover the device's whole range of bus space, that is, to specify an offset of zero and the size of the whole region.

REGION OPERATIONS

Some devices use buffers which are mapped as regions in bus space. Often, drivers want to copy the contents of those buffers to or from memory, e.g. into mbufs which can be passed to higher levels of the system or from mbufs to be output to a network. In order to allow drivers to do this as efficiently as possible, the **bus_space_read_region_N**() and **bus_space_write_region_N**() families of functions are provided.

Drivers occasionally need to copy one region of a bus space to another, or to set all locations in a region of bus space to contain a single value. The **bus_space_copy_region_N**() family of functions and the **bus_space_set_region_N**() family of functions allow drivers to perform these operations.

bus_space_read_region_1(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

bus_space_read_region_2(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

bus_space_read_region_4(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

bus_space_read_region_8(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

The **bus_space_read_region_N**() family of functions reads *count* 1, 2, 4, or 8 byte data items from bus space starting at byte offset *offset* in the region specified by *handle* of the bus space specified by *space*

and writes them into the array specified by *datap*. Each successive data item is read from an offset 1, 2, 4, or 8 bytes after the previous data item (depending on which function is used). All locations being read must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data items being read and the data array pointer should be properly aligned. On some systems, not obeying these requirements may cause incorrect data to be read, on others it may cause a system crash.

Read operations done by the **bus_space_read_region_N**() functions may be executed in any order. They may also be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function. There is no way to insert barriers between reads of individual bus space locations executed by the **bus_space_read_region_N**() functions.

These functions will never fail. If they would fail (e.g. because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

bus_space_write_region_1(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

bus_space_write_region_2(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

bus_space_write_region_4(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

bus_space_write_region_8(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

The **bus_space_write_region_N**() family of functions reads *count* 1, 2, 4, or 8 byte data items from the array specified by *datap* and writes them to bus space starting at byte offset *offset* in the region specified by *handle* of the bus space specified by *space*. Each successive data item is written to an offset 1, 2, 4, or 8 bytes after the previous data item (depending on which function is used). All locations being written must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data items being written and the data array pointer should be properly aligned. On some systems, not obeying these requirements may cause incorrect data to be written, on others it may cause a system crash.

Write operations done by the **bus_space_write_region_N**() functions may be executed in any order. They may also be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function. There is no way to insert barriers between writes of individual bus space locations executed by the **bus_space_write_region_N**() functions.

These functions will never fail. If they would fail (e.g. because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

bus_space_copy_region_1(*space***,** *srchandle***,** *srcoffset***,** *dsthandle***,** *dstoffset***,** *count***)**

bus_space_copy_region_2(*space***,** *srchandle***,** *srcoffset***,** *dsthandle***,** *dstoffset***,** *count***)**

bus_space_copy_region_4(*space***,** *srchandle***,** *srcoffset***,** *dsthandle***,** *dstoffset***,** *count***)**

bus_space_copy_region_8(*space***,** *srchandle***,** *srcoffset***,** *dsthandle***,** *dstoffset***,** *count***)**

The **bus_space_copy_region_N**() family of functions copies *count* 1, 2, 4, or 8 byte data items in bus space from the area starting at byte offset *srcoffset* in the region specified by *srchandle* of the bus space specified by *space* to the area starting at byte offset *dstoffset* in the region specified by *dsthandle* in the same bus space. Each successive data item read or written has an offset 1, 2, 4, or 8 bytes after the previous data item (depending on which function is used). All locations being read and written must lie within the bus space region specified by their respective handles.

For portability, the starting addresses of the regions specified by the each handle plus its respective offset should be a multiple of the size of data items being copied. On some systems, not obeying this requirement may cause incorrect data to be copied, on others it may cause a system crash.

Read and write operations done by the **bus_space_copy_region_N**() functions may be executed in any order. They may also be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function. There is no way to insert barriers between reads or writes of individual bus space locations executed by the **bus_space_copy_region_N**() functions.

Overlapping copies between different subregions of a single region of bus space are handled correctly by the **bus_space_copy_region_N**() functions.

These functions will never fail. If they would fail (e.g. because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

bus_space_set_region_1(*space***,** *handle***,** *offset***,** *value***,** *count***)**

bus_space_set_region_2(*space***,** *handle***,** *offset***,** *value***,** *count***)**

bus_space_set_region_4(*space***,** *handle***,** *offset***,** *value***,** *count***)**

bus_space_set_region_8(*space***,** *handle***,** *offset***,** *value***,** *count***)**

The **bus_space_set_region_N**() family of functions writes the given *value* to *count* 1, 2, 4, or 8 byte data items in bus space starting at byte offset *offset* in the region specified by *handle* of the bus space specified by *space*. Each successive data item has an offset 1, 2, 4, or 8 bytes after the previous data item (depending on which function is used). All locations being written must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data items being written. On some systems, not obeying this requirement may cause incorrect data to be written, on others it may cause a system crash.

Write operations done by the **bus_space_set_region_N**() functions may be executed in any order. They may also be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function. There is no way to insert barriers between writes of individual bus space locations executed by the **bus_space_set_region_N**() functions.

These functions will never fail. If they would fail (e.g. because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

READING AND WRITING A SINGLE LOCATION MULTIPLE TIMES

Some devices implement single locations in bus space which are to be read or written multiple times to communicate data, e.g. some ethernet devices' packet buffer FIFOs. In order to allow drivers to manipulate these types of devices as efficiently as possible, the **bus_space_read_multi_N**(), **bus_space_set_multi_N**(), and **bus_space_write_multi_N**() families of functions are provided.

bus_space_read_multi_1(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

bus_space_read_multi_2(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

bus_space_read_multi_4(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

bus_space_read_multi_8(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

The **bus_space_read_multi_N**() family of functions reads *count* 1, 2, 4, or 8 byte data items from bus space at byte offset *offset* in the region specified by *handle* of the bus space specified by *space* and writes them into the array specified by *datap*. Each successive data item is read from the same location in bus space. The location being read must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data items being read and the data array pointer should be properly aligned. On some systems, not obeying these requirements may cause incorrect data to be read, on others it may cause a system crash.

Read operations done by the **bus_space_read_multi_N**() functions may be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function. Because the **bus_space_read_multi_N**() functions read the same bus space location multiple times, they place an implicit read barrier between each successive read of that bus space location.

These functions will never fail. If they would fail (e.g. because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

bus_space_write_multi_1(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

bus_space_write_multi_2(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

bus_space_write_multi_4(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

bus_space_write_multi_8(*space***,** *handle***,** *offset***,** *datap***,** *count***)**

The **bus_space_write_multi_N**() family of functions reads *count* 1, 2, 4, or 8 byte data items from the array specified by *datap* and writes them into bus space at byte offset *offset* in the region specified by *handle* of the bus space specified by *space*. Each successive data item is written to the same location in bus space. The location being written must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data items being written and the data array pointer should be properly aligned. On some systems, not obeying these requirements may cause incorrect data to be written, on others it may cause a system crash.

Write operations done by the **bus_space_write_multi_N**() functions may be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function. Because the **bus_space_write_multi_N**() functions write the same bus space location multiple times, they place an implicit write barrier between each successive write of that bus space location.

These functions will never fail. If they would fail (e.g. because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

bus_space_set_multi_1(*space***,** *handle***,** *offset***,** *value***,** *count***)**

bus_space_set_multi_2(*space***,** *handle***,** *offset***,** *value***,** *count***)**

bus_space_set_multi_4(*space***,** *handle***,** *offset***,** *value***,** *count***)**

bus_space_set_multi_8(*space***,** *handle***,** *offset***,** *value***,** *count***)**

The **bus_space_set_multi_N**() writes *value* into bus space at byte offset *offset* in the region specified by *handle* of the bus space specified by *space*, *count* times. The location being written must lie within the bus space region specified by *handle*.

For portability, the starting address of the region specified by *handle* plus the offset should be a multiple of the size of data items being written and the data array pointer should be properly aligned. On some systems, not obeying these requirements may cause incorrect data to be written, on others it may cause a system crash.

Write operations done by the **bus_space_set_multi_N**() functions may be executed out of order with respect to other pending read and write operations unless order is enforced by use of the **bus_space_barrier**() function. Because the **bus_space_set_multi_N**() functions write the same bus space location multiple times, they place an implicit write barrier between each successive write of that bus space location.

These functions will never fail. If they would fail (e.g. because of an argument error), that indicates a software bug which should cause a panic. In that case, they will never return.

STREAM FUNCTIONS

Most of the **bus_space** functions imply a host byte-order and a bus byte-order and take care of any translation for the caller. In some cases, however, hardware may map a FIFO or some other memory region for which the caller may want to use multi-word, yet untranslated access. Access to these types of memory regions should be with the **bus_space_*_stream_N**() functions.

bus_space_read_stream_1()
bus_space_read_stream_2()
bus_space_read_stream_4()
bus_space_read_stream_8()
bus_space_read_multi_stream_1()
bus_space_read_multi_stream_2()
bus_space_read_multi_stream_4()
bus_space_read_multi_stream_8()
bus_space_read_region_stream_1()
bus_space_read_region_stream_2()
bus_space_read_region_stream_4()
bus_space_read_region_stream_8()
bus_space_write_stream_1()
bus_space_write_stream_2()
bus_space_write_stream_4()

bus_space_write_stream_8()
bus_space_write_multi_stream_1()
bus_space_write_multi_stream_2()
bus_space_write_multi_stream_4()
bus_space_write_multi_stream_8()
bus_space_write_region_stream_1()
bus_space_write_region_stream_2()
bus_space_write_region_stream_4()
bus_space_write_region_stream_8()
bus_space_copy_region_stream_1()
bus_space_copy_region_stream_2()
bus_space_copy_region_stream_4()
bus_space_copy_region_stream_8()
bus_space_set_multi_stream_1()
bus_space_set_multi_stream_2()
bus_space_set_multi_stream_4()
bus_space_set_multi_stream_8()
bus_space_set_region_stream_1()
bus_space_set_region_stream_2()
bus_space_set_region_stream_4()
bus_space_set_region_stream_8()

These functions are defined just as their non-stream counterparts, except that they provide no byte-order translation.

COMPATIBILITY

The current NetBSD version of the **bus_space** interface specification differs slightly from the original specification that came into wide use and FreeBSD adopted. A few of the function names and arguments have changed for consistency and increased functionality.

SEE ALSO

bus_dma(9)

HISTORY

The **bus_space** functions were introduced in a different form (memory and I/O spaces were accessed via different sets of functions) in NetBSD 1.2. The functions were merged to work on generic "spaces" early in the NetBSD 1.3 development cycle, and many drivers were converted to use them. This document was written later during the NetBSD 1.3 development cycle, and the specification was updated to fix some consistency problems and to add some missing functionality.

The manual page was then adapted to the version of the interface that FreeBSD imported for the CAM SCSI drivers, plus subsequent evolution. The FreeBSD **bus_space** version was imported in FreeBSD 3.0.

AUTHORS

The **bus_space** interfaces were designed and implemented by the NetBSD developer community. Primary contributors and implementors were Chris Demetriou, Jason Thorpe, and Charles Hannum, but the rest of the NetBSD developers and the user community played a significant role in development.

Justin Gibbs ported these interfaces to FreeBSD.

Chris Demetriou wrote this manual page.

Warner Losh modified it for the FreeBSD implementation.

BUGS

This manual may not completely and accurately document the interface, and many parts of the interface are unspecified.