NAME

intro - introduction to kernel programming interfaces

DESCRIPTION

Welcome to the FreeBSD kernel documentation. Outside of the source code itself, this set of man(1) pages is the primary resource for information on usage of the numerous programming interfaces available within the kernel. In some cases, it is also a source of truth for the implementation details and/or design decisions behind a particular subsystem or piece of code.

The intended audience of this documentation is developers, and the primary authors are also developers. It is written assuming a certain familiarity with common programming or OS-level concepts and practices. However, this documentation should also attempt to provide enough background information that readers approaching a particular subsystem or interface for the first time will be able to understand.

To further set expectations, we acknowledge that kernel documentation, like the source code itself, is forever a work-in-progress. There will be large sections of the codebase whose documentation is subtly or severely outdated, or missing altogether. This documentation is a supplement to the source code, and can not always be taken at face value.

At its best, section 9 documentation will provide a description of a particular piece of code that, paired with its implementation, fully informs the reader of the intended and realized effects.

man(1) pages in this section most frequently describe functions, but may also describe types, global variables, macros, or high-level concepts.

CODING GUIDELINES

Code written for the FreeBSD kernel is expected to conform to the established style and coding conventions. Please see style(9) for a detailed set of rules and guidelines.

OVERVIEW

Below is presented various subsystems.

Data Structures

There are implementations for many well-known data structures available in the kernel.

bitstring(3) Simple bitmap implementation.

counter(9) An SMP-safe general-purpose counter implementation.

hash(9) Hash map implementation.

nv(9) Name/value pairs.

queue(3) Singly-linked and doubly-linked lists, and queues.

refcount(9) An SMP-safe implementation of reference counts.

sbuf(9) Dynamic string composition.

sglist(9) A scatter/gather list implementation.

Utility Functions

Functions or facilities of general usefulness or convenience. See also the *Testing and Debugging Tools* or *Miscellaneous* sub-sections below.

Formatted output and logging functions are described by printf(9).

Endian-swapping functions: byteorder(9).

Data output in hexadecimal format: hexdump(9).

A rich set of macros for declaring sysctl(8) variables and functions is described by sysctl(9).

Non-recoverable errors in the kernel should trigger a panic(9). Run-time assertions can be verified using the KASSERT(9) macros. Compile-time assertions should use **_Static_assert**().

The SYSINIT framework provides macros for declaring functions that will be executed during start-up and shutdown; see SYSINIT(9).

Deprecation messages may be emitted with gone_in(9).

A unit number facility is provided by unr(9).

Synchronization Primitives

The locking(9) man page gives an overview of the various types of locks available in the kernel and advice on their usage.

Atomic primitives are described by atomic(9).

The epoch(9) and smr(9) facilities are used to create lock-free data structures. There is also seqc(9).

Memory Management

Dynamic memory allocations inside the kernel are generally done using malloc(9). Frequently allocated objects may prefer to use uma(9).

Much of the virtual memory system operates on *vm_page_t* structures. The following functions are documented:

vm_page_advise(9), vm_page_alloc(9), vm_page_bits(9), vm_page_aflag(9), vm_page_alloc(9), vm_page_bits(9), vm_page_busy(9), vm_page_deactivate(9), vm_page_free(9), vm_page_grab(9), vm_page_insert(9), vm_page_lookup(9), vm_page_rename(9), vm_page_sbusy(9), vm_page_wire(9)

Virtual address space maps are managed with the vm_map(9) API.

The machine-dependent portion of the virtual memory stack is the pmap(9) module.

Allocation policies for NUMA memory domains are managed with the domainset(9) API.

File Systems

The kernel interface for file systems is VFS(9). File system implementations register themselves with vfsconf(9).

The abstract and filesystem-independent representation of a file, directory, or other file-like entity within the kernel is the vnode(9).

The implementation of access control lists for filesystems is described by acl(9). Also vaccess(9).

I/O and Storage

The GEOM framework represents I/O requests using the bio(9) structure.

Disk drivers connect themselves to GEOM using the disk(9) API.

The devstat(9) facility provides an interface for recording device statistics in disk drivers.

Networking

Much of the networking stack uses the mbuf(9), a flexible memory management unit commonly used to store network packets.

Network interfaces are implemented using the ifnet(9) API, which has functions for drivers and consumers.

A framework for managing packet output queues is described by altq(9).

To receive incoming packets, network protocols register themselves with netisr(9).

Virtualization of the network stack is provided by VNET(9).

The front-end for interfacing with network sockets from within the kernel is described by socket(9). The back-end interface for socket implementations is domain(9).

The low-level packet filter interface is described by pfil(9).

The bpf(9) interface provides a mechanism to redirect packets to userspace.

The subsystem for IEEE 802.11 wireless networking is described by ieee80211(9).

A framework for modular TCP implementations is described by tcp_functions(9).

A framework for modular congestion control algorithms is described by mod_cc(9).

Device Drivers

Consult the device(9) and driver(9) pages first.

Most drivers act as devices, and provide a set of methods implementing the device interface. This includes methods such as DEVICE_PROBE(9), DEVICE_ATTACH(9), and DEVICE_DETACH(9).

In addition to devices, there are buses. Buses may have children, in the form of devices or other buses. Bus drivers will implement additional methods, such as BUS_ADD_CHILD(9), BUS_READ_IVAR(9), or BUS_RESCAN(9).

Buses often perform resource accounting on behalf of their children. For this there is the rman(9) API.

Drivers can request and manage their resources (e.g. memory-space or IRQ number) from their parent using the following sets of functions:

bus_alloc_resource(9), bus_adjust_resource(9), bus_get_resource(9), bus_map_resource(9), bus_release_resource(9), bus_set_resource(9)

Direct Memory Access (DMA) is handled using the busdma(9) framework.

Functions for accessing bus space (i.e. read/write) are provided by bus_space(9).

Clocks and Timekeeping

The kernel clock frequency and overall system time model is described by hz(9).

A few global time variables, such as system up-time, are described by time(9).

Raw CPU cycles are provided by get_cyclecount(9).

Userspace Memory Access

Direct read/write access of userspace memory from the kernel is not permitted, and memory transactions that cross the kernel/user boundary must go through one of several interfaces built for this task.

Most device drivers use the uiomove(9) set of routines.

Simpler primitives for reading or writing smaller chunks of memory are described by casuword(9), copy(9), fetch(9), and store(9).

Kernel Threads, Tasks, and Callbacks

Kernel threads and processes are created using the kthread(9) and kproc(9) interfaces, respectively.

Where dedicated kernel threads are too heavyweight, there is also the taskqueue(9) interface.

For low-latency callback handling, the callout(9) framework should be used.

Dynamic handlers for pre-defined event hooks are registered and invoked using the EVENTHANDLER(9) API.

Thread Switching and Scheduling

The machine-independent interface to a context switch is mi_switch(9).

To prevent preemption, use a critical(9) section.

To voluntarily yield the processor, use kern_yield(9).

The various functions which will deliberately put a thread to sleep are described by sleep(9). Sleeping threads are removed from the scheduler and placed on a sleepqueue(9).

Processes and Signals

To locate a process or process group by its identifier, use pfind(9) and pgfind(9). Alternatively, the pget(9) function provides additional search specificity.

The "hold count" of a process can be manipulated with PHOLD(9).

The kernel interface for signals is described by signal(9).

Signals can be sent to processes or process groups using the functions described by psignal(9).

Security

See the generic security overview in security(7).

The basic structure for user credentials is *struct ucred*, managed by the ucred(9) API. Thread credentials are verified using priv(9) to allow or deny certain privileged actions.

Policies influenced by *kern.securelevel* must use the securelevel_gt(9) or securelevel_ge(9) functions.

The Mandatory Access Control (MAC) framework provides a wide set of hooks, supporting dynamically-registered security modules; see mac(9).

Cryptographic services are provided by the OpenCrypto framework. This API provides and interface for both consumers and crypto drivers; see crypto(9).

For information on random number generation, see random(9) and prng(9).

Kernel Modules

The interfaces for declaring loadable kernel modules are described by module(9).

Interrupts

The machine-independent portion of the interrupt framework supporting the registration and execution of interrupt handlers is described by intr_event(9).

Software interrupts are provided by swi(9).

Device drivers register their interrupt handlers using the bus_setup_intr(9) function.

Testing and Debugging Tools

A kernel test framework: kern_testfrwk(9)

A facility for defining configurable fail points is described by fail(9).

Commands for the ddb(4) kernel debugger are defined with the DB_COMMAND(9) family of macros.

The ktr(4) tracing facility adds static tracepoints to many areas of the kernel. These tracepoints are defined using the macros described by ktr(9).

Static probes for DTrace are defined using the SDT(9) macros.

Stack traces can be captured and printed with the stack(9) API.

Kernel sanitizers can perform additional compiler-assisted checks against memory use/access. These runtimes are capable of detecting difficult-to-identify classes of bugs, at the cost of a large overhead. Supported is the Kernel Address Sanitizer KASAN(9), and the Kernel Memory Sanitizer KMSAN(9).

The **LOCK_PROFILING** kernel config option enables extra code to assist with profiling and/or debugging lock performance; see LOCK_PROFILING(9).

Driver Tools

Defined functions/APIs for specific types of devices.

iflib(9) Programming interface for iflib(4) based network drivers.

pci(9) Peripheral Component Interconnect (PCI) and PCI Express (PCIe) programming API.

pwmbus(9)
Pulse-Width Modulation (PWM) bus interface methods.

usbdi(9) Universal Serial Bus programming interface.

superio(9) Functions for Super I/O controller devices.

Miscellaneous

Dynamic per-CPU variables: dpcpu(9).

CPU bitmap management: cpuset(9).

Kernel environment management: getenv(9).

Contexts for CPU floating-point registers are managed by the fpu_kern(9) facility.

For details on the shutdown/reboot procedure and available shutdown hooks, see reboot(9).

A facility for asynchronous logging to files from within the kernel is provided by alq(9).

The osd(9) framework provides a mechanism to dynamically extend core structures in a way that preserves KBI. See the hhook(9) and khelp(9) APIs for information on how this is used.

The kernel object implementation is described by kobj(9).

SEE ALSO

man(1), style(9)

The FreeBSD Architecture Handbook, https://docs.freebsd.org/en/books/arch-handbook/.