BY JOHN BALDWIN

IMPLEMENTING SYSTEM CONTROL NODES (sysctl)

The FreeBSD kernel provides a set of system control nodes that can be used to query and set state information. These nodes can be used to obtain a wide variety of statistics and configure parameters. The information accepted and provided by each node can use one of several types-including integers, strings, and structures.

THE NODES are organized as a tree. Each node is assigned a unique number within its level. Nodes are identified internally by an array of node numbers that traverses the tree from the root down to the node being requested. Most nodes in the tree have a name, and named nodes may be identified by a string of names separated by dots. For example, there is a top-level node named "kern" that is assigned the number one (1), and it contains a child node named "ostype" that is assigned the number one (1). This node's address is 1.1, but it may also be referred to by its full name: "kern.ostype". Users and applications generally use node names rather than node numbers.

Accessing System Control Nodes

The standard C library on FreeBSD provides several routines for accessing system control nodes. The sysctl(3) and sysctlbyname(3) functions access a single node. The sysctl(3) function uses the internal array of numbers to identify a node while sysctlbyname(3) accepts a string containing the node's full name. Each sysctl access can

query the current state of the node, set the state of the node, or perform both operations.

The sysctlnametomib(3) function maps a node's full name to its internal address. This operation uses an internal sysctl node and is a bit expensive, so a program that queries a control node frequently can use this routine to cache the address of a node. It can then query the node using sysctl(3) rather than sysctlbyname(3).

Some control nodes have a named prefix with unnamed leaves. An example of this is the "kern.proc.pid" node. It contains a child node for each process. The internal address of a given process's node consists of the address of "kern.proc.pid" and a fourth number which corresponds to the pid of the process.

Example 1 demonstrates using this to fetch information about the current process.

```
struct kinfo_proc kp;
int i, mib[4];
size_t len;

/* Fetch the address of the "kern.proc.pid"
   prefix. */
len = 3;
sysctlnametomib("kern.proc.pid", mib, &len);

/* Fetch the process information for the
   current process. */
len = sizeof(kp);
mib[3] = getpid();
sysctl(mib, 4, &kp, &len, NULL, 0);
```

Example 1

Simple Control Nodes

In the kernel, the <sys/sysctl.h> header provides several macros to declare control nodes. Each declaration includes the name of the parent node, a number to assign to this node, a name for the node, flags to control the node's behavior, and a description of the node. Some declarations require additional arguments. The parent node is identified by its full name, but with a single underscore as a prefix and dots replaced by underscores. For example, the "foo.bar" parent node would be identified by "_foo_bar". To declare a top-level node, use an empty parent name. The number should use the macro `OID_AUTO` to request that the system assign a unique number. (Some nodes use hardcoded numbers for legacy reasons, but all new nodes should use system-assigned numbers.) The flags argument must indicate which types of access the node supports (read, write, or both) and can also include other optional flags. The description should be a short string describing the node. It is displayed instead of the value when the "-d" flag is passed to sysctl(8).

Integer Nodes

The simplest and most common control node is a leaf node that controls a single integer. This type of node is defined by the `SYSCTL_INT()` macro. It accepts two additional arguments; a pointer and a value. If the pointer is non-`NULL`, it should point to an integer variable which will be read and written by the control node (as specified in the flags argument). If the pointer is `NULL`, then the node must be a read-only node that returns the value argument when read.

```
SYSCTL_INT(_kern, OID_AUTO, one, CTLFLAG_RD,
    NULL, 1, "Always returns one");

int frob = 500;
SYSCTL_INT(_kern, OID_AUTO, frob, CTLFLAG_RW,
    &frob, 0, "The \"frob\" variable");
```

Example 2

Example 2 defines two integer sysctl nodes: "kern.one" is a read-only node that always returns the value one and "kern.frob" is a read-write node that reads and writes the value of the global "frob" integer.

Additional macros are available for several integer types including: `SYSCTL_UINT()` for unsigned integers, `SYSCTL_LONG()` for signed long integers, `SYSCTL_ULONG()` for unsigned long integers, `SYSCTL_QUAD()` for signed 64-bit integers, `SYSCTL_UQUAD()` for 64-bit unsigned integers, and `SYSCTL_COUNTER_U64()` for 64-bit unsigned integers managed by the counter(9) API. Only the `SYSCTL_INT()` and `SYSCTL_UINT()` macros may be used with a `NULL` pointer. The other macros require a non-`NULL` pointer and ignore the value parameter.

Other Node Types

The `SYSCTL_STRING()` macro is used to define a leaf node with a string value. This macro accepts two additional arguments: a pointer and a length. The pointer should point to the start of the string. If the length is zero, the string is assumed to be a constant string and attempts to write to the string will fail (even if the node allows write access). If the length is non-zero, it specifies the maximum length of the string buffer (including a terminating null char-

```
static SYSCTL_NODE(, OID_AUTO, demo, 0, NULL,
    "Demonstration tree");

static char name_buffer[64] = "initial name";
SYSCTL_STRING(_demo, OID_AUTO, name,
    CTLFLAG_RW, name_buffer,
    sizeof(name_buffer), "Demo name");

static struct demo_stats {
        int demo_reads;
        int demo_writes;
} stats;
SYSCTL_STRUCT(_demo, OID_AUTO, status,
     CTLFLAG_RW, &stats, demo_stats,
     "Demo statistics");

SYSCTL_OPAQUE(_demo, OID_AUTO, mi_switch,
    CTLFLAG_RD, &mi_switch, 64, "Code",
    "First 64 bytes of mi_switch()");
```

Example 3

acter) and attempts to write a string longer than the buffer's size will fail.

The `SYSCTL_STRUCT()` macro is used to define a leaf node whose value is a single C struct. This macro accepts one additional pointer argument which should point to the structure to be controlled. The size of the structure is inferred from the type.

The `SYSCTL_OPAQUE()` macro is used to define a leaf node whose value is a data buffer of unspecified type. The macro accepts three additional arguments: a pointer to the start of the data buffer, the length of the data buffer, and a string describing the format of the data buffer.

The `SYSCTL_NODE()` macro is used to define a branch node. This macro accepts one additional argument which is a pointer to a function handler. For a branch node with explicit leaf nodes (declared by other `SYSCTL_*()` macros) the pointer should be `NULL`. The macro may be prefixed with `static` to declare a branch node private to the current file. A public node can be forward declared in a header for use by other files via the `SYSCTL_DECL()` macro. This macro accepts a single argument which is the full name of the node specified in the format used for a parent node in the other macro invocations. Example 3 defines a top level node with three leaf nodes describing a string buffer, a structure, and an opaque data buffer.

Node Flags

Each node definition requires a flags argument. All leaf nodes and branch nodes with a non-NULL function handler must specify the permitted access (read and/or write) in the flags field. The flags field can also include zero or more of the flags listed in Table 1.

Complex Control Nodes

System control nodes are not limited to simply reading and writing existing variables. Each leaf node includes a pointer to a handler function that is invoked when the node is accessed. This function is responsible for returning the "old" value of a node as well as accepting "new" values assigned to a node. The standard node macros such as `SYSCTL_INT()` use predefined handlers in `sys/kern/kern_sysctl.c`.

A leaf node with a custom handler function is defined via the `SYSCTL_PROC()` macro. In addition to the standard arguments accepted by the other macros, `SYSCTL_PROC()` accepts a

Table 1

FLAG	PURPOSE
CTLFLAG_ANYBODY	All users can write to this node. Normally only the superuser can write to a node.
CTLFLAG_SECURE	Can only be written if securelevel is less than or equal to zero.
CTLFLAG_PRISON	Can be written to by a superuser inside of a prison created by jail(2).
CTLFLAG_SKIP	Hides this node from iterative walks of the tree such as when sysctl(8) lists nodes.
CTLFLAG_MPSAFE	Handler routine does not require Giant. All of the simple node types set this flag already. It is only required explicitly for nodes that use a custom handler.
CTLFLAG_VNET	Can be written to by a superuser inside of a prison if that prison contains its own virtual network stack.

FLAG	MEANING
CTLTYPE_NODE	This node is a branch node and does not have an associated value.
CTLTYPE_INT	This node describes one or more signed integers.
CTLTYPE_UINT	This node describes one or more unsigned integers.
CTLTYPE_LONG	This node describes one or more signed long integers.
CTLTYPE_ULONG	This node describes one or more unsigned long integers.
CTLTYPE_S64	This node describes one or more signed 64-bit integers.
CTLTYPE_U64	This node describes one or more unsigned 64-bit integers.
CTLTYPE_STRING	This node describes one or more strings.
CTLTYPE_OPAQUE	This node describes an arbitrary data buffer.
CTLTYPE_STRUCT	This node describes one or more data structures.

Table 2

pointer argument named "arg1", an integer argument named "arg2", a pointer to the handler function, and a string describing the format of the node's value. The flags argument is also required to specify the type of the node's value.

Node Types

The type of a node's value is specified in a field in the node's flags. The standard node macros all work with a specific type and adjust the flags argument to include the appropriate type. The `SYSCTL_PROC()` macro does not imply a specific type, so the type must be specified explicitly. Note that all nodes are allowed to return or accept an array of values and the type simply specifies the type of one array member. The standard node macros all return or accept a single value rather than an array. The available types are listed in Table 2.

Note that since `SYSCTL_PROC()` only defines leaf nodes, `CTLTYPE_NODE` should not be used. Branch nodes with custom handlers are described below.

Node Format Strings

Each node has a format string in addition to a type. The sysctl(8) utility uses this string to format the node's value. As with the node type, most of the standard macros specify the format implicitly. The `SYSCTL_OPAQUE` and `SYSCTL_PROC` macros require the format to be specified explicitly. Most format strings are tied to a specific type and most types only have a single format string. The available format strings are listed in Table 3.

Handler Functions

A system control node handler can be used to provide additional behavior beyond reading and writing an existing variable. Handlers can be used to provide input validation such as range checks on new node values. Handlers can also generate temporary data structures to return to userland. This is commonly done for handlers which return a snapshot of system state such as a list of open network connections or the process table.

Handler functions accept four arguments and return an integer error code. The `<sys/sysctl.h>` header provides a macro to define the function arguments: `SYSCTL_HANDLER_ARGS`. It defines four arguments: "oidp", "arg1", "arg2", and "req". The

FORMAT	MEANING
"A"	An ASCII string. Used with CTLTYPE_STRING.
"I"	A signed integer. Used with CTLTYPE_INT.
"IU"	An unsigned integer. Used with CTLTYPE_UINT.
"IK"	An integer whose value is in units of one-tenth of a degree Kelvin. The sysctl(8) utility will convert the value to Celsius before displaying. Used with CTLTYPE_UINT.
"L"	A signed long integer. Used with CTLTYPE_LONG.
"LU"	An unsigned long integer. Used with CTLTYPE_ULONG.
"Q"	A signed 64-bit integer. Used with CTLTYPE_S64.
"QU"	An unsigned 64-bit integer. Used with CTLTYPE_U64.
"S,<foo>"	A C structure of type `struct foo`. Used with CTLTYPE_STRUCT. The sysctl(8) utility understands a few structure types such as `struct timeval` and `struct loadavg`.

Table 3

"oidp" argument points to the `struct sysctl_oid` structure that describes the node whose handler is being invoked. The "arg1" and "arg2" arguments hold the values assigned to the "arg1" and "arg2" arguments to the `SYSCTL_PROC()` invocation that defined this node. The "req" argument points to a `struct sysctl_req` structure that describes the specific request being made. The return value should be zero on success or an error number from `<sys/errno.h>` on failure. If `EAGAIN` is returned, then the request will be retried within the kernel without returning to userland or checking for signals.

Example 4 defines two integer nodes with a custom handler that rejects attempts to set an invalid value. It uses the predefined handler function `sysctl_handle_int()` that is used to implement `SYSCTL_INT()` to update a local variable. If the request attempts to set a new value, it validates the new value and only updates the associated variable if the new value is accepted.

This example uses a predefined handler (`sysctl_handle_int()`) to publish the old value and accept a new value. Some custom handlers need to manage these steps directly. The macros `SYSCTL_IN()` and `SYSCTL_OUT()` are provided for this purpose. Both macros accept three arguments: a pointer to the current request ("req") from `SYSCTL_HANDLER_ARGS`, a pointer to a buffer in the kernel's address space, and a length. The `SYSCTL_IN()` macro copies data from the caller's "new" buffer into the kernel buffer. The `SYSCTL_OUT()` macro copies data from the kernel buffer into the caller's "old" buffer. These macros return zero if the copy is successful and an error number if it fails. In particular, if the caller buffer is too small, the macros will fail and return `ENOMEM`. These macros can be invoked multiple times. Each invocation advances an internal offset into the caller's buffer. Multiple invocations of `SYSCTL_OUT()` append the kernel buffers passed to the macro to the caller's "old" buffer, and multiple invocations of `SYSCTL_IN()` will read sequential blocks of data from the caller's "new" buffer.

One of the values returned to userland after a sysctl(3) invocation is the amount of data returned in the "old" buffer. The count is advanced by the full length passed to `SYSCTL_OUT()` even if the copy fails with an error. This can be used to allow userland to query the necessary length of an "old" buffer

Example 4

```
/*
 * 'arg1' points to the variable being exported, and 'arg2' specifies a
 * maximum value.  This assumes that negative values are not permitted.
 */
static int
sysctl_handle_int_range(SYSCTL_HANDLER_ARGS)
{
        int error, value;

        value = *(int *)arg1;
        error = sysctl_handle_int(oidp, &value, 0, req);
        if (error != 0 || req->newptr == NULL)
                return (error);
        if (value < 0 || value >= arg2)
                return (EINVAL);
        *(int *)arg1 = value;
        return (0);
}

static int foo;
SYSCTL_PROC(_debug, OID_AUTO, foo, CTLFLAG_RW | CTLTYPE_INT, &foo, 100,
    sysctl_handle_int_range, "I", "Integer between 0 and 99");

static int bar;
SYSCTL_PROC(_debug, OID_AUTO, bar, CTLFLAG_RW | CTLTYPE_INT, &bar, 0x100,
    sysctl_handle_int_range, "I", "Integer between 0 and 255");
```

for a node that returns a variable-sized buffer. If it is expensive to generate the data copied to the "out" buffer and a handler is able to estimate the amount of space needed, then the handler can treat this case specially. A caller queries length by using a `NULL` pointer for the "old" buffer. The handler can detect this case by comparing `req->oldptr` against `NULL`. The handler can then make a single call to `SYSCTL_OUT()` passing `NULL` as the kernel buffer and the total estimated length as the length. If the size of the data changes frequently, then the handler should overestimate the size of the buffer so that the caller is less likely to get an `ENOMEM` error on the subsequent call to query the node's state.

The `SYSCTL_OUT()` and `SYSCTL_IN()` macros can access memory in a user process. These accesses can trigger page faults if a user page is not currently mapped. For this reason, non-sleepable locks such as mutexes and reader/writer locks cannot be held when invoking these macros. Some control nodes return an array of state objects that correspond to a list of objects inside the kernel where the list is protected by a non-sleepable lock. One option such handlers can use is to allocate a temporary buffer in the kernel that is large enough to hold all of the output. The handler can populate the kernel buffer while it walks the list under the lock and then pass the populated buffer to `SYSCTL_OUT()` at the end after releasing the lock. Another option is to drop the lock around each invocation of `SYSCTL_OUT()` while walking the list. Some handlers may not want to allocate a temporary kernel buffer because it would be too large, and they may wish to avoid dropping the lock because the resulting races are too painful to handle. The system provides a third option for these handlers: the "old" buffer of a request can be wired by calling `sysctl_wire_old_buffer()`. Wiring the buffer guarantees that no accesses to the buffer will fault allowing `SYSCTL_OUT()` to be used while holding a non-sleepable lock. Note that this option is only available for the "old" buffer. There is no corresponding function for the "new" buffer. The `sysctl_wire_old_buffer()` function returns zero if it succeeds and an error number if it fails.

If a sysctl node wishes to work properly in a 64-bit kernel when it is acccessed by a 32-bit process, it can detect this case by checking for the `SCTL_MASK32` flag in `req->flags`. For example, a node that returns a long value should return a 32-bit integer in this case. A node that returns an array of structures corresponding to an internal list of objects may need to return an array of structures with an alternate 32-bit layout.

If a node allows the caller to alter its state via a "new" value, the handler should compare `req->newptr` against `NULL` to determine if a "new" value is supplied. A handler should only invoke `SYSCTL_IN()` and attempt to set a new value if `req->newptr` is non-`NULL`.

An example of a custom node handler that uses many of these features is the implementation of the "kern.proc.proc" node. The in-kernel implementation is more complex, but a simplified version is provided in Example 5.

Complex Branch Nodes

A branch node declared via `SYSCTL_NODE()` can specify a custom handler. If a handler is specified, then it is always invoked when any node whose address begins with the address of the branch node is accessed. The handler functions similarly to the custom handlers described above. Unlike `SYSCTL_PROC()`, the "arg1" and "arg2" parameters are not configurable. Instead, "arg1" points to an integer array containing the address of the node being accessed, and "arg2" contains the length of the address. Note that the address specified by "arg1" and "arg2" is relative to the branch node whose handler is being invoked. For example, if a branch node has the address 1.2 and node 1.2.3.4 is accessed, the handler for the branch node will be invoked with "arg1" pointing to an array containing "3, 4" and "arg2" set to 2. A simplified version of the "kern.proc.pid" handler is given below as Example 6. Recall that this is the node invoked by Example 1.

Dynamic Control Nodes

The control nodes described previously are static control nodes. Static nodes are defined in a source file with a fixed name and are created either when the kernel initializes the system control node subsystem or when a kernel module is loaded. Static nodes in a kernel module are removed when the kernel module is unloaded. The arguments passed to handlers for static nodes are also resolved at link time. This means that static nodes generally operate

Example 5

```
static int
sysctl_kern_proc_proc(SYSCTL_HANDLER_ARGS)
{
#ifdef COMPAT_FREEBSD32
        struct kinfo_proc32 kp32;
#endif
        struct kinfo_proc kp;
        struct proc *p;
        int error;

        if (req->oldptr == NULL) {
#ifdef COMPAT_FREEBSD32
                if (req->flags & SCTL_MASK32)
                        return (SYSCTL_OUT(req, NULL, (nprocs + 5) *
                            sizeof(struct kinfo_proc32)));
#endif
                return (SYSCTL_OUT(req, NULL, (nprocs + 5) *
                    sizeof(struct kinfo_proc)));
        }

        error = sysctl_wire_old_buffer(req, 0);
        if (error != 0)
                return (error);
        sx_slock(&allproc_lock);
        LIST_FOREACH(p, &allproc, p_list) {
                PROC_LOCK(p);
                fill_kinfo_proc(p, &kp);
#ifdef COMPAT_FREEBSD32
                if (req->flags & SCTL_MASK32) {
                        freebsd32_kinfo_proc_out(&kp, &kp32);
                        error = SYSCTL_OUT(req, &kp32, sizeof(kp32));
                } else
#endif
                        error = SYSCTL_OUT(req, &kp, sizeof(kp));
                PROC_UNLOCK(p);
                if (error != 0)
                        break;
        }
        sx_sunlock(&allproc_lock);
        return (error);
}
SYSCTL_PROC(_kern_proc, KERN_PROC_PROC, proc, CTLFLAG_RD |
    CTLFLAG_MPSAFE | CTLTYPE_STRUCT, NULL, 0, sysctl_kern_proc_proc,
    "S,kinfo_proc", "Process table");
```

on global variables.

The kernel also provides support for dynamic control nodes. Unlike static nodes, dynamic nodes can be created or destroyed at any time. They can also use dynamically generated names and reference dynamically allocated variables. Dynamic nodes can be created as new children of both static and dynamic nodes.

`sysctl` Contexts

To safely remove dynamic control nodes, each node must be explicitly tracked and removed in a safe order (leaves before branches). Doing this by hand is tedious and error prone, so the kernel provides a sysctl context abstraction. A sysctl context is a container that tracks zero or more dynamic control nodes. It allows all of the control nodes it contains to be safely removed in an atomic transaction.

The typical practice is to create one context for each group of related nodes via a call to `sysctl_ctx_init()`. All of the nodes are added to the context during initialization (such

as when a driver attaches to a device). Only a reference to the context has to be maintained. A single call to `sysctl_ctx_free()` during teardown (such as when a driver detaches from a device) is sufficient to remove the entire group of control nodes.

Adding Dynamic Control Nodes

Dynamic control nodes are created by using one of the `SYSCTL_ADD_*()` macros from `<sys/sysctl.h>`. Each of these macros corresponds to a macro used to create static node with the following differences:

- The dynamic macros are invoked within a code block rather than at the top level. The dynamic macros return a pointer to the created node.
- The dynamic macros add an additional argument that is a pointer to the sysctl context the new node should be associated with. This is given as the first argument to the macro.
- The parent argument is specified as a pointer to the node list belonging to the parent node. Two helper macros are provided to locate these pointers. The `SYSCTL_STATIC_CHILDREN()` macro should be used when the parent node is a static control node. It takes the parent node's name as the sole argument. The name is formatted in the same manner that a parent is specified when declaring a static node. For parents that are dynamic nodes, the `SYSCTL_CHILDREN()` macro should be used instead. It accepts a pointer to the parent node as returned by a previous invocation of `SYSCTL_ADD_NODE()` as its sole argument.
- The name argument is specified as a pointer to a C string rather than an unquoted identifier. The kernel will create a duplicate of this string to use as the name of the node. This allows the name to be constructed in a temporary buffer if needed.
- The kernel will also create a duplicate of the description argument so that it can be constructed in a temporary buffer if needed.

Example 7 defines two functions to manage a dynamic sysctl node. The first function initializes a sysctl context and creates the new node. The second function destroys the node and destructs the context.

Tunables

Another kernel API that is often used in conjunction with control nodes is the tunable API. Tunables are values stored in the kernel's environment. This environment is populated by the boot loader and can also be modified at run-time by the kenv(1) command. The kernel environment consists of named variables with string values similar to the environment of user processes. The API provides two sets of macros in `<sys/kernel.h>`.

```
static int
sysctl_kern_proc_pid(SYSCTL_HANDLER_ARGS)
{
        struct kinfo_proc kp;
        struct proc *p;
        int *mib;

        if (arg2 == 0)
                return (EISDIR);
        if (arg2 != 1)
                return (ENOENT);
        mib = (int *)arg1;
        p = pfind(mib[0]);
        if (p == NULL)
                return (ESRCH);
        fill_kinfo_proc(p, &kp);
        PROC_UNLOCK(p);
        return (SYSCTL_OUT(req, &kp,
            sizeof(kp)));
}
static SYSCTL_NODE(_kern_proc, KERN_PROC_PID,
    pid, CTLFLAG_RD | CTLFLAG_MPSAFE,
    sysctl_kern_proc_pid,
    "Process information");
```

Example 6

The first set (`TUNABLE_*()`) are declared at the top-level similar to static control nodes and fetch the value of a tunable at either boot time or when a module is loaded. The second set of macros (`TUNABLE_*_FETCH()`) can be used in a code block to fetch a tunable at runtime. Each macro accepts a C string name as the first argument that specifies the name of the tunable to read. When using tunables in conjunction with control nodes, the convention is to use the name of the control node as the tunable's name.

The tunable API supports several integer types. Each macro accepts a pointer to an integer variable of the corresponding type as the second argument. Each macro invocation searches the kernel's environment for the requested tunable. If the tunable is found and the entire string value is parsed successfully, the integer variable is changed to

SUFFIX	VALUE
k	2^10
m	2^20
g	2^30
t	2^40

Table 4

Example 7

```
static struct sysctl_ctx_list ctx;

static int
load(void)
{
        static int value;
        int error;

        error = sysctl_ctx_init(&ctx);
        if (error)
                return (error);
        if (SYSCTL_ADD_INT(&ctx, SYSCTL_STATIC_CHILDREN(_debug), OID_AUTO,
            "dynamic", CTLFLAG_RW, &value, 0, "An integer") == NULL)
                return (ENXIO);
        return (0);
}

static int
unload(void)
{

        return (sysctl_ctx_free(&ctx));
}
```

the parsed value. Note that overflows are silently ignored. If the tunable is not found or contains invalid characters, the integer variable is left unchanged. The macros provided for integers are: `TUNABLE_INT()` for signed integers, `TUNABLE_LONG()` for signed long integers, `TUNABLE_ULONG()` for unsigned long integers, and `TUNABLE_QUAD()` for signed 64-bit integers.

The string value of an integer tunable is parsed in the same manner as strtol(3) with a base of zero. Specifically, a string that begins with "0x" is interpreted as a hexadecimal value, a string that begins with "0" is interpreted as an octal value, and all other strings are interpreted as a decimal value. In addition, the string may contain an optional single character suffix that specifies a unit. The value is scaled by the size of the unit. The unit is case-insensitive. Supported units are described in Table 4.

String tunables are also supported by the `TUNABLE_STR()` macro. This macro accepts three arguments: the name of the tunable, a pointer to a character buffer, and the length of the character buffer. If the tunable does not exist in the kernel environment, the character buffer is left unchanged. If the tunable does exist, its value is copied into the buffer. The string in the buffer is always terminated with a null character. The value will be truncated if it is too long to fit into the buffer.

The `TUNABLE_*_FETCH()` macros accept the same arguments as the corresponding `TUNABLE_*()` macro. They also have the same semantics with one additional behavior. These macros return an integer value of zero if the tunable is found and successfully parsed, and non-zero otherwise.

System control nodes that have a corresponding tunable should use either the `CTLFLAG_RDTUN` or `CTLFLAG_RWTUN` flag to specify the allowed access to the node. Note that this does not cause the system to implicitly fetch a tunable based on the node's name. The tunable must be fetched explicitly. However, it does provide a hint to the sysctl(8) utility that is used in diagnostic messages.

Example 8 demonstrates the use of a tunable in a device driver to fetch a default parameter. The parameter is available as a read-only control node that can be queried by the user (this is helpful for the user when determining the default value). It also includes a portion of the attach routine where the global tunable is used to set the initial value of a per-device control variable. A dynamic sysctl is created for each device to allow the variable to be changed for each device independently. The sysctl is stored in the per-device sysctl tree

Example 8

```
static SYSCTL_NODE(_hw, OID_AUTO, foo, CTLFLAG_RD, NULL,
    "foo(4) parameters");

static int foo_widgets = 5;
TUNABLE_INT("hw.foo.widgets", &foo_widgets);
SYSCTL_INT(_hw_foo, OID_AUTO, widgets, CTLFLAG_RDTUN, &foo_widgets, 0,
   "Initial number of widgets for each foo(4) device");

static int
foo_attach(device_t dev)
{
        struct foo_softc *sc;
        char descr[64];

        sc = device_get_softc(dev);
        sc->widgets = foo_widgets;
        snprintf(descr, sizeof(descr), "Number of widgets for %s",
            device_get_nameunit(dev));
        SYSCTL_ADD_INT(device_get_sysctl_ctx(dev),
            SYSCTL_CHILDREN(device_get_sysctl_tree(dev)), OID_AUTO,
            "widgets", CTLFLAG_RW, &sc->widgets, 0, descr);
        ...
}
```

created by the new-bus subsystem. It also uses the per-device sysctl context so that the sysctl is automatically destroyed when the device is detached.

The interface for system control nodes is defined in `<sys/sysctl.h>` and the implementation can be found in `sys/kern/kern_sysctl.c`. It may be particularly useful to examine the implementation of the predefined handlers. First, they demonstrate typical uses of `SYSCTL_IN()` and `SYSCTL_OUT()`. Second, they can be used to marshal data in custom handlers.

The interface for tunables is defined in `<sys/kernel.h>` and the implementation can be found in `sys/kern/kern_environment.c`.

John Baldwin joined the FreeBSD Project as a committer in 1999. He has worked in several areas of the system, including SMP infrastructure, the network stack, virtual memory, and device driver support. John has served on the Core and Release Engineering teams and organizes an annual FreeBSD developer summit each spring.

SUBSCRIBE TO

FreeBSD® JOURNAL

AVAILABLE AT THESE APP STORES NOW

amazon apps Available at amazon

Download on the App Store

ANDROID APP ON Google play