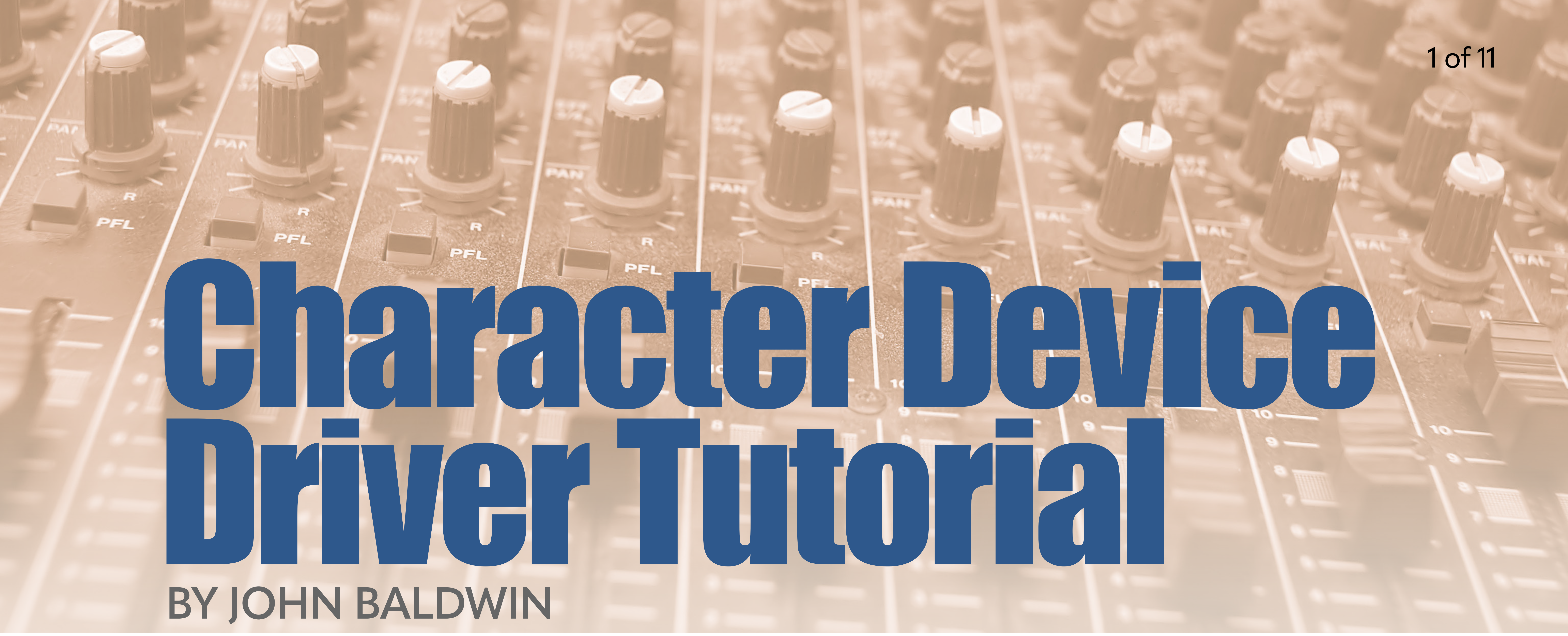

Character Device Driver Tutorial

BY JOHN BALDWIN

Character devices provide pseudo files exported to userspace applications by the device filesystem (devfs(5)). Unlike standard filesystems where the semantics of various operations such as reading and writing are the same across all files within a filesystem, each character device defines its own semantics for each file operation. Character device drivers declare a character device switch (`struct cdevsw`) which includes function pointers for each file operation.

Character device switches are often implemented as part of a hardware device driver. FreeBSD's kernel provides several wrapper APIs which implement a character device on top of a simpler set of operations. The disk(9) API implements an internal character device switch on top of the methods in struct disk for example. Several device drivers provide a character device to export device behavior that doesn't map to an existing in-kernel subsystem to userspace.

Character device switches are often implemented as part of a hardware device driver.

Other character device switches are implemented purely as a software construct. For example, the `/dev/null` and `/dev/zero` character devices are not associated with any hardware device.

In a series of three articles, the first of which is this one, we will build a simple character device driver progressively adding new functionality to explore character device switches and several of the operations character device drivers can implement. The full source of each version of device driver can be found at https://github.com/bsdjhb/cdev_tutorial. We will start with a barebones driver which creates a single character device.

Lifecycle Management

A character device driver is responsible for explicitly creating and destroying character devices. Active character devices are represented by instances of `struct cdev`. Character devices are created by the make_dev_s(9) function. This function accepts a pointer to an arguments structure, a pointer to a character device object pointer, and a printf-style format string and following arguments. The format string and following arguments are used to construct the name of the character device.

The arguments structure contains a few mandatory fields and several optional fields. The structure must be initialized by a call to `make_dev_args_init()` before setting any fields.

The **mda_devsw** member must point to the character device switch. The **mda_uid**, **mda_gid**, and **mda_mode** fields should be set to the initial user ID, group ID, and permissions of the device node. Most character devices are owned by root:wheel, and the constants `UID_ROOT` and `GID_WHEEL` can be used for this. The **mda_flags** field should also be set to either `MAKEDEV_NOWAIT` or `MAKEDEV_WAITOK`. Additional flags can be included via the C or operator if needed. For our sample driver, we set `MAKEDEV_CHECKNAME` so that we can fail gracefully with an error if an echo device already exists rather than panicking the system.

Character devices are destroyed by passing a pointer to the character device to **destroy_dev()**. This function will block until all references to the character device have been removed. This includes waiting for any threads currently executing in character device switch methods for this device to return from those methods. Once **destroy_dev()** returns, it is safe to release any resources used by the character device. Alternatively, character devices can be destroyed asynchronously via either **destroy_dev_sched()** or **destroy_dev_sched_cb()**. These functions schedule destruction of the character device on an internal kernel thread. For **destroy_dev_sched_cb()**, the supplied callback is invoked with the supplied argument after the character device has been destroyed. This can be used to release resources used by the character device. Keep in mind that one of the resources a character device uses are the character device switch methods. This means, for example, that module unloading must wait for any character devices using functions defined in that module to be destroyed.

Alternatively, character devices can be destroyed asynchronously.

For our initial driver (Listing 1), we use a module event handler to create a /dev/echo device when the module is loaded and destroy it when the module is unloaded. After building and loading this module, the device exists but isn't able to do much as shown in Example 1. The character device switch for this driver (**echo_cdevsw**) is initialized with only two required fields: **d_version** must always be set to the constant `D_VERSION`, and **d_name** should be set to the driver name.

Listing 1: Barebones Driver

```
#include <sys/param.h>
#include <sys/conf.h>
#include <sys/kernel.h>
#include <sys/module.h>

static struct cdev *echodev;

static struct cdevsw echo_cdevsw = {
	.d_version =	D_VERSION,
	.d_name =	"echo"
};

static int
```

```
echodev_load(void)
{
	struct make_dev_args args;
	int error;

	make_dev_args_init(&args);
	args.mda_flags = MAKEDEV_WAITOK | MAKEDEV_CHECKNAME;
	args.mda_devsw = &echo_cdevsw;
	args.mda_uid = UID_ROOT;
	args.mda_gid = GID_WHEEL;
	args.mda_mode = 0600;
	error = make_dev_s(&args, &echodev, "echo");
	return (error);
}

static int
echodev_unload(void)
{
	if (echodev != NULL)
		destroy_dev(echodev);
	return (0);
}

static int
echodev_modevent(module_t mod, int type, void *data)
{
	switch (type) {
	case MOD_LOAD:
		return (echodev_load());
	case MOD_UNLOAD:
		return (echodev_unload());
	default:
		return (EOPNOTSUPP);
	}
}

DEV_MODULE(echodev, echodev_modevent, NULL);
```

Example 1: Using the Barebones Driver

```
# ls -l /dev/echo
crw-------  1 root wheel 0x39 Oct 25 13:06 /dev/echo
# cat /dev/echo
cat: /dev/echo: Operation not supported by device
```

Reading and Writing

Now that we have a character device, let's add some behavior. As the name "echo" implies, this device should accept input by writing to the device and echo that input back out by reading from the device. To provide this, we will add read and write methods to the character device switch.

Read and write requests for character devices are described by a **`struct uio`** object. Two of the fields in this structure are useful for character device drivers: **`uio_offset`** is the logical file offset (e.g. from lseek(2)) for the start of the request and **`uio_resid`** is the number of bytes to transfer. Data is transferred between the application buffer and an in-kernel buffer by the uiomove(9) function. This function updates members of the uio object including **`uio_offset`** and **`uio_resid`** and can be called multiple times. A request can be completed as a short operation by moving a subset of bytes to or from the application buffer.

The second version of the echo driver adds a global static buffer to use as the backing store for read and write requests. The logical file offset is treated as an offset into the global buffer. Requests are truncated to the size of the buffer, so that reading beyond the end of the buffer triggers a zero-byte read indicating EOF. Writes beyond the end of the buffer fail with the error **`EFBIG`**. To protect against concurrent access, a global sx(9) lock is used to protect the buffer. An sx(9) lock is used instead of a regular mutex since **`uiomove()`** might sleep while it faults in a page backing an application buffer. Listing 2 shows the read and write character device methods.

Listing 2: Read and Write Using a Global Buffer

```
static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
	size_t todo;
	int error;

	if (uio->uio_offset >= sizeof(echobuf))
		return (0);

	sx_slock(&echolock);
	todo = MIN(uio->uio_resid, sizeof(echobuf) - uio->uio_offset);
	error = uiomove(echobuf + uio->uio_offset, todo, uio);
	sx_sunlock(&echolock);
	return (error);
}

static int
echo_write(struct cdev *dev, struct uio *uio, int ioflag)
{
	size_t todo;
	int error;

	if (uio->uio_offset >= sizeof(echobuf))
		return (EFBIG);

	sx_xlock(&echolock);
```

```
        todo = MIN(uio->uio_resid, sizeof(echobuf) - uio->uio_offset);
        error = uiomove(echobuf + uio->uio_offset, todo, uio);
        sx_xunlock(&echolock);
        return (error);
}
```

The body of these methods are mostly identical. One reason for this is that the arguments to **`uiomove()`** are the same for both read and write. This is because the uio object encodes the direction of the data transfer as part of its state.

If we load this version of the driver, we can now interact with the device by reading and writing to it. Example 2 shows a few interactions demonstrating the echo behavior. Note that the output of jot exceeded the size of the driver's 64-byte buffer, so the subsequent read of the device was truncated.

Example 2: Echoing Data Using a Global Buffer

```
# cat /dev/echo
# echo foo > /dev/echo
# cat /dev/echo
foo
# jot -c -s "" 70 48 > /dev/echo
# cat /dev/echo
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmno#
```

Device Configuration via ioctl()

The fixed size of the global buffer is a weird quirk of this device. We can permit changing the buffer size by adding a custom ioctl(2) command for this device. I/O control commands are named by a command constant and accept an optional argument.

Command constants are defined by one of the `_IO`, `_IOR`, `_IOW`, or `_IOWR` macros from the <sys/ioccom.h> header. All these macros accept a group and number as the first two arguments. Both values are 8 bits. Typically, an ASCII alphabetical character is used as the group, and all commands for a given driver use the same group. FreeBSD's kernel defines several existing sets of I/O control commands. A set of generic commands that can be used with any file descriptor are defined in <sys/filio.h> using the group 'f'. Other sets are intended for use with specific types of file descriptors such as the commands in <sys/sockio.h> which are defined for sockets. For custom commands for a character device driver, do not use the 'f' group to avoid potential conflicts with the generic commands in <sys/filio.h>. Each command should use a different value for the number argument. If a command accepts an optional argument, the type of the argument must be given as the third argument to the `_IOR`, `_IOW`, or `_IOWR` macro. The `_IOR` macro defines a command that returns a value from the driver to the userspace application (the command "reads" the argument from the driver). The `_IOW` macro defines a command that passes a value to the driver (the command "writes" the argument to the driver). The `_IOWR` macro defines a command that is both read and written by the driver. The size of the argument is encoded in the command constant. This means that commands with the same group and number, but a different sized argument, will have different command constants. This is useful when implementing support for alternate userspace ABIs (for example, supporting a 32-bit userspace application on a 64-bit kernel) as the alternate ABIs will use a different command constant.

BSD kernels such as FreeBSD manage the copying of the I/O control command argument in the generic system call layer. This differs from Linux where the kernel passes the raw userspace pointer to the device driver, requiring the device driver to copy data to and from userspace. Instead, BSD kernels use the size argument encoded in the command constant to allocate an in-kernel buffer of the requested size. If the command was defined with `_IOW` or `_IOWR`, the buffer is initialized by copying the argument value in from the userspace application. If the command was defined with `_IOR`, the buffer is cleared with zeroes. After the device driver's ioctl routine completes, if the command was defined with `_IOR` or `_IOWR`, the buffer's contents are copied out to the userspace application.

For the echo driver, let's define three new control commands. The first command returns the current size of the global buffer. The second command permits setting a new size of the global buffer. The third command clears the contents of the buffer by resetting all the bytes to zero.

These commands are defined in a new echodev.h header shown in Listing 3. A header is used so that the constants can be shared with userspace applications as well as the driver. Note that the first command reads the buffer size into a size_t argument in userspace, the second command writes a new buffer size from a size_t argument in userspace, and the third command does not accept an argument. All three commands use the 'E' group and are assigned unique command numbers.

Listing 3: I/O Control Command Constants

```
#define     ECHODEV_GBUFSIZE   _IOR('E', 100, size_t)   /* get buffer size */
#define     ECHODEV_SBUFSIZE   _IOW('E', 101, size_t)   /* set buffer size */
#define     ECHODEV_CLEAR      _IO('E', 102)            /* clear buffer */
```

Supporting a dynamically sized buffer requires several driver changes. The global buffer is replaced with a global pointer to a dynamically allocated buffer, and a new global variable contains the buffer's current size. The pointer and length are initialized during module load, and the current buffer is freed during module unload. Since the buffer's size is no longer a constant, the checks for out-of-bounds reads and writes must now be done while holding the lock.

The in-kernel malloc(9) for FreeBSD requires an additional malloc type argument for both the allocation and free routines. Malloc types track allocation requests providing fine-grained statistics. These statistics are available via the -m flag to the vmstat(8) command which displays a separate line for each type. The kernel does include a general device buffer malloc type (`M_DEVBUF`) that drivers can use. However, it is best practice for drivers to define a dedicated malloc type. This is especially true for drivers in kernel modules. When a module is unloaded, malloc types defined in a kernel module are destroyed. If any allocations still reference those malloc types, the kernel emits a warning about the leaked allocations. The finer-grained statistics are also useful for debugging and performance analysis. New malloc types are defined via the `MALLOC_DEFINE` macro. The first argument provides the variable name of the new type. By convention, types are named in all uppercase and use a leading prefix of "M_". For this driver, we will use the name `M_ECHODEV`. The second argument is a short string name displayed by utilities such as vmstat(8). It is best practice to avoid whitespace characters in the short name. The third argument is a string description of the type.

Driver support for the custom control commands is implemented in the new function in Listing 4. The `cmd` argument contains the command constant for the requested command and the `data` argument points to the in-kernel buffer containing the optional command argument. The overall structure of the function is a switch statement on the `cmd` argument. The default error value for unknown commands is `ENOTTY`, even for non-tty devices. The two commands which accept a size argument cast `data` to the correct pointer type before dereferencing. The `ECHODEV_GBUFSIZE` command writes the current size to `*data`, while `ECHODEV_SBUFSIZE` reads the desired new size from `*data`.

For commands which alter the device state, the driver requires a writable file descriptor (that is, a file descriptor opened with `O_RDWR` or `O_WRONLY`). To enforce this, the `ECHODEF_SBUFSIZE` and `ECHODEV_CLEAR` commands require the `FWRITE` flag to be set in `fflag`. The `fflag` argument contains the file descriptor status flags defined in <sys/fcntl.h>. These flags map `O_RDONLY`, `O_WRONLY`, and `O_RDWR` to a combination of the `FREAD` and `FWRITE` flags. All other flags from open(2) are included directly in the file descriptor status flags. Note that a subset of these flags can be changed on an open file descriptor by fcntl(2).

Listing 4: I/O Control Handler

```
static int
echo_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int fflag,
    struct thread *td)
{
      int error;

      switch (cmd) {
      case ECHODEV_GBUFSIZE:
            sx_slock(&echolock);
            *(size_t *)data = echolen;
            sx_sunlock(&echolock);
            error = 0;
            break;
      case ECHODEV_SBUFSIZE:
      {
            size_t new_len;

            if ((fflag & FWRITE) == 0) {
                  error = EPERM;
                  break;
            }

            new_len = *(size_t *)data;
            sx_xlock(&echolock);
            if (new_len == echolen) {
                  /* Nothing to do. */
            } else if (new_len < echolen) {
                  echolen = new_len;
            } else {
                  echobuf = reallocf(echobuf, new_len, M_ECHODEV,
                        M_WAITOK | M_ZERO);
                    echolen = new_len;
            }
            sx_xunlock(&echolock);
            error = 0;
            break;
      }
      case ECHODEV_CLEAR:
            if ((fflag & FWRITE) == 0) {
                  error = EPERM;
                  break;
            }

            sx_xlock(&echolock);
            memset(echobuf, 0, echolen);
            sx_xunlock(&echolock);
            error = 0;
            break;
      default:
            error = ENOTTY;
            break;
      }
      return (error);
}
```

To invoke these commands from userspace, we need a new user application. The repository contains an **`echoctl`** program used in Example 3. The size command outputs the current size of the buffer, the resize command sets a new buffer size, and the clear command clears the buffer contents. Note that in this example, the output from jot is no longer truncated. The last command in this example displays the dynamic allocation statistics for the driver's allocations using `M_ECHODEV`.

Example 3: Resizing the Global Buffer

```
# echoctl size
64
# echo foo > /dev/echo
# echoctl clear
# cat /dev/echo
# echoctl resize 80
# jot -c -s "" 70 48 > /dev/echo
# cat /dev/echo
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstu
# vmstat -m | egrep 'Type|echo'
           Type  Use Memory Req Size(s)
        echodev    1   128    2 64,128
```

Per-Instance Data

So far, our device driver has used global variables to hold its state. For a simple demonstration driver with a single device instance this is ok. However, most character devices are part of a hardware device driver and need to support multiple instances of a device within a single system. To support this, drivers define a structure containing the software context for a single device instance. In BSD kernels, this software context is named a "softc". Drivers typically define a structure type whose name uses a "_softc" suffix, and variables holding pointers to softc structures are usually named "sc".

Character devices provide straightforward support for per-instance data. `struct cdev` contains three members available for storing driver-specific data. `si_drv0` contains an integer value while `si_drv1` and `si_drv2` store arbitrary pointers. Device drivers are free to set these variables while creating character devices using the `mda_unit`, `mda_si_drv1`, and `mda_si_drv2` fields of `struct make_dev_args`. These values can then be accessed as members of the `struct cdev` argument to character device switch methods. Historically, device drivers used a unit number to track per-instance data. Modern device drivers in FreeBSD store a softc pointer in the `si_drv1` field and rarely use the other two fields.

Most character devices are part of a hardware device driver and need to support multiple instances of a device within a single system.

For our echo device driver, we define a `struct echodev_softc` type containing all of the state needed for an instance of the echo device. The device driver still stores a single global holding the softc of the single instance for use during module load and unload, but the rest of the driver accesses state via the softc pointer. These changes do not change any of the driver's functionality but do require refactoring various parts of the driver. Listing 5 shows the new softc structure type. Listing 6 demonstrates the type of refactoring needed for each character device switch method by showing the updated read method. Lastly, Listing 7 shows the updated routines used during module load and unload.

Listing 5: softc Structure

```
struct echodev_softc {
      struct cdev *dev;
      char *buf;
      size_t len;
      struct sx lock;
};
```

Listing 6: Driver Method Using softc Structure

```
static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
      struct echodev_softc *sc = dev->si_drv1;
      size_t todo;
      int error;
```

```
        sx_slock(&sc->lock);
        if (uio->uio_offset >= sc->len) {
                error = 0;
        } else {
                todo = MIN(uio->uio_resid, sc->len - uio->uio_offset);
                error = uiomove(sc->buf + uio->uio_offset, todo, uio);
        }
        sx_sunlock(&sc->lock);
        return (error);
}
```

Listing 7: Module Load and Unload Using softc Structure

```
static int
echodev_create(struct echodev_softc **scp, size_t len)
{
        struct make_dev_args args;
        struct echodev_softc *sc;
        int error;

        sc = malloc(sizeof(*sc), M_ECHODEV, M_WAITOK | M_ZERO);
        sx_init(&sc->lock, "echo");
        sc->buf = malloc(len, M_ECHODEV, M_WAITOK | M_ZERO);
        sc->len = len;
        make_dev_args_init(&args);
        args.mda_flags = MAKEDEV_WAITOK | MAKEDEV_CHECKNAME;
        args.mda_devsw = &echo_cdevsw;
        args.mda_uid = UID_ROOT;
        args.mda_gid = GID_WHEEL;
        args.mda_mode = 0600;
        args.mda_si_drv1 = sc;
        error = make_dev_s(&args, &sc->dev, "echo");
        if (error != 0) {
                free(sc->buf, M_ECHODEV);
                sx_destroy(&sc->lock);
                free(sc, M_ECHODEV);
        }
        return (error);
}

static void
echodev_destroy(struct echodev_softc *sc)
{
        if (sc->dev != NULL)
                destroy_dev(sc->dev);
        free(sc->buf, M_ECHODEV);
        sx_destroy(&sc->lock);
        free(sc, M_ECHODEV);
}

static int
echodev_modevent(module_t mod, int type, void *data)
{
        static struct echodev_softc *echo_softc;

        switch (type) {
        case MOD_LOAD:
                return (echodev_create(&echo_softc, 64));
        case MOD_UNLOAD:
                if (echo_softc != NULL)
                        echodev_destroy(echo_softc);
                return (0);
        default:
                return (EOPNOTSUPP);
        }
}
```

Conclusion

Thanks for reading this far. The next article in this series will extend this driver to implement a FIFO buffer including support for non-blocking I/O and I/O event reporting via poll(2) and kevent(2).

JOHN BALDWIN is a systems software developer. He has directly committed changes to the FreeBSD operating system for over twenty years across various parts of the kernel (including x86 platform support, SMP, various device drivers, and the virtual memory subsystem) and userspace programs. In addition to writing code, John has served on the FreeBSD core and release engineering teams. He has also contributed to the GDB debugger. John lives in Ashland, Virginia with his wife, Kimberly, and three children: Janelle, Evan, and Bella.