Character Device Driver Tutorial (Part 2)

BY JOHN BALDWIN

In the previous article in this three-part series, we built a simple character device driver that permitted I/O operations backed by a fixed buffer. In this article, we will extend this driver to support a FIFO data buffer along with support for non-blocking I/O and event reporting. The full source of each version of the device driver can be found at https://github.com/bsdjhb/cdev_tutorial.

Before moving forward though, we must address an unintended bit of unfinished business from the previous article. Alert reader Virus-V noted that the final version of the echo driver from the article did not destroy the `/dev/echo` device during unload and that accessing the device after unload triggered a kernel panic. One of the first clues to the bug lay in a warning message about leaked memory from the kernel emitted during module unload prior to the panic. As noted in the prior article, this warning is one of the reasons kernel modules should use a dedicated malloc type when possible. The bug lay in the `echodev_create()` function added in the last set of changes. We had failed to return a pointer to the newly allocated softc structure to the caller by storing the value in `*scp`. As a result, the `echo_softc` variable was always NULL and the softc was not destroyed during module unload. The fix was a one line addition to `echodev_create()` to store the pointer to the new softc in `*scp` on success.

The echo driver from the first article used a flat data buffer for I/O operations.

Using a FIFO Data Buffer

The echo driver from the first article used a flat data buffer for I/O operations. Reads and writes could access any region of the data buffer, and the entire range of the data buffer was always valid. These semantics are similar to accessing a file that does not grow when written beyond the end. A character device driver is free to implement a range of semantics, however. For this article, we will alter the echo driver to treat user I/O data like a FIFO stream device similar to a `pipe` or `fifo`. I/O write requests will append data to the tail of a logical data buffer and read requests will read data from the head of this data buffer. File offsets such as those used with `pread(2)` will be ignored. The driver will continue to use an in-kernel buffer to hold a temporary copy of user data. Writes will store data in this buffer and reads will consume data from this buffer. This means that the driver will now need to keep track of the amount of valid data in the buffer as well as the buffer's length. To simplify the implementation, the start of the data buffer will always be treated as the buffer's head. Read requests that read a subset of the available data will copy the remaining data to the front of the buffer.

This does raise several additional questions, however. First, how should reads that want to read more data from the buffer than is available be handled? Second, how should writes that want to store more data than the buffer can hold be handled? For simplicity, we will start by returning a short read of whatever bytes are available for read requests, and truncating write requests to only store the amount of data for which there is room in the buffer. Third, what should an `ECHODEV_SBUFSIZE` request do that shrinks a buffer smaller than the amount of valid data in the buffer? We have chosen to fail such a request with an error. One could choose to discard some of the data instead, but one would have to decide which data to discard. Listing 1 provides the updated read and write methods. Note that a new `valid` member has been added to the softc to track the amount of valid data in the buffer. Example 1 demonstrates a few scenarios of this updated driver. Initially, the device is empty, but data can be read once input is provided. The last few commands read a series of bytes across two requests.

Listing 1: Read and Write Using a FIFO Data Buffer

```
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_xlock(&sc->lock);
	todo = MIN(uio->uio_resid, sc->valid);
	error = uiomove(sc->buf, todo, uio);
	if (error == 0) {
		sc->valid -= todo;
		memmove(sc->buf, sc->buf + todo, sc->valid);
	}
	sx_xunlock(&sc->lock);
	return (error);
}

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

	sx_xlock(&sc->lock);
	todo = MIN(uio->uio_resid, sc->len - sc->valid);
```

```
        error = uiomove(sc->buf + sc->valid, todo, uio);
        if (error == 0)
                sc->valid += todo;
        sx_xunlock(&sc->lock);
        return (error);
}
```

Example 1: Simple FIFO I/O

```
# hd < /dev/echo

# echo "foo" > /dev/echo
# cat /dev/echo
foo
# echo "12345678" > /dev/echo
# dd if=/dev/echo bs=1 count=4 status=none | hd
00000000  31 32 33 34                                       |1234|
00000004
# cat /dev/echo
5678
```

Blocking I/O

While this version of the echo driver does implement a simple data stream, it has some limitations. If a process wants to use this device to share a block of data larger than the data buffer, it has to wait to write an additional buffer's worth of data until a reader has consumed the previous buffer's worth of data. This requires the writing process to either coordinate with the reader process(es) or to use a timer and retry write operations periodically. Neither of these solutions are very practical. Instead, the driver can permit larger writes by sleeping in the write request while the buffer is full until the request is completed. Readers would awaken the waiting writer when space is available permitting the writer to make additional progress. Similarly, readers could block waiting for data to return. To more closely match the semantics of pipes and sockets, we have chosen to make read requests only block at the start of a request and return a short read as soon as data is available. However, for writes, we attempt to drain the entire buffer. To handle blocking, we use the `sx_sleep(9)` function which atomically releases our device's lock while putting the current thread to sleep. Passing `PCATCH` to this function permits signals to interrupt this sleep in which case `sx_sleep()` will return a non-zero error value. Listing 2 shows the updated read method. The write method is similarly updated but with an extra loop to retry until the write has fully completed. Note that in the write method, we do not have to "hide" errors if the write partially completes. The generic write system call handling in the `dofilewrite()` function maps errors from `sx_sleep()` to success if at least some data was written. Some of the ioctl handlers also require updates to awaken sleeping writers when the buffer grows in size or has its contents cleared.

When testing this version of the driver, Example 2 shows some possibly surprising behavior. While the data previously written is returned, `cat(1)` continues to wait for additional data until killed with a signal.

Listing 2: Blocking Read Method

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

	if (uio->uio_resid == 0)
		return (0);

	sx_xlock(&sc->lock);

	/* Wait for bytes to read. */
	while (sc->valid == 0) {
		error = sx_sleep(sc, &sc->lock, PCATCH, "echord", 0);
		if (error != 0) {
		sx_xunlock(&sc->lock);
		return (error);
		}
	}

	todo = MIN(uio->uio_resid, sc->valid);
	error = uiomove(sc->buf, todo, uio);
	if (error == 0) {
		/* Wakeup any waiting writers. */
		if (sc->valid == sc->len)
		wakeup(sc);

		sc->valid -= todo;
		memmove(sc->buf, sc->buf + todo, sc->valid);
	}
	sx_xunlock(&sc->lock);
	return (error);
}
```

Example 2: Blocking I/O Hangs Forever

```
# echo "12345678" > /dev/echo
# cat /dev/echo
12345678
^C
```

Unloading Sanely

We will get to the surprising behavior from Example 2 shortly. The current driver has another surprise. Unloading the driver while a process is blocked in the read or write methods will hang the process unloading the module until the first process is killed with a signal. This is not the behavior an administrator expects when unloading a module. Instead, the echo driver should awaken any sleeping threads during device destruction and ensure they will return from the driver methods without sleeping again. To support this, the next change adds a `dying` flag to the `softc` and the read and write methods fail with an `ENXIO` error rather than blocking if this flag is set. During device destruction, the `dying` flag is set and sleeping threads are awakened before calling `destroy_dev()`. Listing 3 shows the changed lines in the read method and the updated `echodev_destroy()` function.

Listing 3: Waking Threads on Unload

```
static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
        ...
        /* Wait for bytes to read. */
        while (sc->valid == 0) {
                if (sc->dying)
                        error = ENXIO;
                else
                        error = sx_sleep(sc, &sc->lock, PCATCH, "echord", 0);
                if (error != 0) {
                        sx_xunlock(&sc->lock);
                        return (error);
                }
        }
        ...
}

...

static void
echodev_destroy(struct echodev_softc *sc)
{
        if (sc->dev != NULL) {
                /* Force any sleeping threads to exit the driver. */
                sx_xlock(&sc->lock);
                sc->dying = true;
                wakeup(sc);
                sx_xunlock(&sc->lock);

                destroy_dev(sc->dev);
        }
        free(sc->buf, M_ECHODEV);
        sx_destroy(&sc->lock);
        free(sc, M_ECHODEV);
}
```

Conditional Blocking on Read

In Example 2, it was surprising that `cat(1)` continued to block after reading the available data from the device given our prior examples. However, this behavior does follow naturally from our driver since `cat(1)` just calls `read(2)` in a loop until it receives EOF and the second call to `read(2)` blocks waiting for more data. The semantic used by other stream devices like pipes and fifos is that reads will return EOF instead of blocking if no process has the device open for writing. If there are devices open for writing, reads will block waiting for more data.

We can implement these semantics in the echo driver easily. We add a count of writers to the softc and only block in the read method if this count is non-zero. To detect writers, we add an open method that increments the counter for each open which requests write permission. This can be determined by checking for the `FWRITE` flag in the file flags passed to the open method. A new close method decrements count when a writer closes. By default, the close character device switch method is only called for the last close of a device when no remaining file descriptors remain. Instead, we set the `D_TRACKCLOSE` character device switch flag so that the close method is called each time a file descriptor is closed. The close method awakens any waiting readers if the last writer is closed. Listing 4 shows the new open and close methods as well as the changed line in the read method. Retrying the steps from Example 2 no longer results in the surprising behavior as `cat(1)` now exits after reading the available data.

Listing 4: Tracking Open Writers

```
static int
echo_open(struct cdev *dev, int fflag, int devtype, struct thread *td)
{
	struct echodev_softc *sc = dev->si_drv1;

	if ((fflag & FWRITE) != 0) {
		/* Increase the number of writers. */
		sx_xlock(&sc->lock);
		if (sc->writers == UINT_MAX) {
			sx_xunlock(&sc->lock);
			return (EBUSY);
		}
		sc->writers++;
		sx_xunlock(&sc->lock);
	}
	return (0);
}

static int
echo_close(struct cdev *dev, int fflag, int devtype, struct thread *td)
{
	struct echodev_softc *sc = dev->si_drv1;

	if ((fflag & FWRITE) != 0) {
		sx_xlock(&sc->lock);
		sc->writers--;
```

```
                if (sc->writers == 0) {
                        /* Wakeup any waiting readers. */
                        wakeup(sc);
                }
                sx_xunlock(&sc->lock);
        }
        return (0);
}

static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
        ...
        /* Wait for bytes to read. */
        while (sc->valid == 0 && sc->writers != 0) {
        ...
}
```

Non-blocking I/O

Now our echo device supports blocking I/O. However, some consumers may wish to use non-blocking I/O. A process can request non-blocking I/O either by passing the O_NONBLOCK flag to open(2), or toggling the O_NONBLOCK flag on an open file descriptor via fcntl(2). A character device driver can check if non-blocking I/O is enabled by checking for the O_NONBLOCK flag in the file flags passed to the read and write methods. If non-blocking I/O is requested, the error EWOULDBLOCK should be returned instead of blocking any time the driver would block. For the echo device, this means adding extra checks before blocking in the read and write methods. That alone is sufficient to handle non-blocking I/O requested at open time. However, to support toggling flags via fcntl(2), an additional step is required.

The I/O control handler should return zero if the requested flag setting is supported.

Every attempt to set file flags via the fcntl(2) F_SETFL operation invokes two I/O control commands on a character device: FIONBIO and FIOASYNC. Even requests that do not change the state of the associated O_NONBLOCK and O_ASYNC flags invoke these I/O control commands. If either I/O control command fails, the entire F_SETFL operation fails, and the file flags remain unchanged. As a result, a character device driver that wants to support F_SETFL must implement support for both I/O control commands.

FIONBIO and FIOASYNC pass an int as the command argument. This int value is zero if the associated file flag is clear in the new file flags or non-zero if the associated flag is set in the new file flags. The I/O control handler should return zero if the requested flag setting is supported, or an error if the requested setting is not supported. The echo device supports either setting for the O_NONBLOCK flag but does not support setting the O_ASYNC flag, so the FIOASYNC handler for the echo device fails if the int argument is non-zero.

Listing 5 shows the relevant changes to the read and I/O control methods to support non-blocking I/O.

Listing 5: Support for Non-Blocking I/O

```
static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
	...
	/* Wait for bytes to read. */
	while (sc->valid == 0 && sc->writers != 0) {
		if (sc->dying)
			error = ENXIO;
		else if (ioflag & O_NONBLOCK)
			error = EWOULDBLOCK;
		else
			error = sx_sleep(sc, &sc->lock, PCATCH, "echord", 0);
	...
}

...

static int
echo_ioctl(struct cdev *dev, u_long cmd, caddr_t data, int fflag,
    struct thread *td)
{
	...
	switch (cmd) {
	...
	case FIONBIO:
		/* O_NONBLOCK is supported. */
		error = 0;
		break;
	case FIOASYNC:
		/* O_ASYNC is not supported. */
		if (*(int *)data != 0)
			error = EINVAL;
		else
			error = 0;
		break;
	...
}
```

Polling I/O Status

Applications that use non-blocking I/O often use an event loop to service requests for multiple file descriptors. For each iteration of the loop, the application blocks waiting for one or more file descriptors to be ready (for example, having data available to read, or room for more data to be written). The application then services each of the ready file descriptors before waiting again. FreeBSD supports two system calls for this type of event loop: `select(2)` and `poll(2)`. In the kernel, `select(2)` and `poll(2)` are implemented using a common framework. Each requested file descriptor is polled individually to determine if it is ready. If no file descriptors are ready, the thread invoking the system call can sleep waiting for at least one of the file descriptors to become ready. If the file descriptor becomes ready while a thread is waiting, the file descriptor must awaken the sleeping thread.

The `selrecord(9)` family of functions manages the sleeping and awakening of threads. A file descriptor that supports polling must create a `struct selinfo` object for type of event it supports. The object should be initialized by clearing the entire object with zeroes (for example, using `memset()`). If the file descriptor's poll function finds that a file descriptor is not ready, it must call `selrecord()` on the associated `struct selinfo` object for each requested event. Any time an event occurs that could make a file descriptor ready, `selwakeup()` must be called on the `struct selinfo` object for that event. Finally, `seldrain()` should be used to awaken any remaining threads before destroying a `struct selinfo` object.

For the echo device we support both read and write events.

For character devices, the file descriptor polling function invokes the character device poll method. This method accepts a bitmask of `poll(2)` events as a function argument and must return a mask of those events that are currently true. In addition, this function is responsible for calling `selrecord()` if none of the requested events are true. Note that character devices do not support different types of priority data via the read and write methods, only normal data.

For the echo device we support both read and write events. We add two `struct selinfo` objects to the softc, one for each event. Since the entire softc is zeroed on creation, no further changes are required when initializing the softc. Each of the read and write methods calls `selwakeup()` for the *other* event to awaken any threads that might be waiting. A few other places can also make the echo device ready as well and need calls to `selwakeup()`. If an I/O control command grows the buffer or clears its contents, that may make the device ready to write. If the last writer closes the device, that can also make the device ready to read. A new poll method determines the current status of the device and calls `selrecord()` as needed. Finally, `seldrain()` is called for each event when destroying the device. Listing 6 shows the added call to `selwakeup()` in the read method as well as the new poll method. Note that for the read method, `selwakeup()` is used for the write event.

Listing 7: Device Polling

```
static int
echo_read(struct cdev *dev, struct uio *uio, int ioflag)
{
        ...
        error = uiomove(sc->buf, todo, uio);
        if (error == 0) {
                /* Wakeup any waiting writers. */
                if (sc->valid == sc->len)
                        wakeup(sc);

                sc->valid -= todo;
                memmove(sc->buf, sc->buf + todo, sc->valid);
                selwakeup(&sc->wsel);
        }
        ...
}

...

static int
echo_poll(struct cdev *dev, int events, struct thread *td)
{
        struct echodev_softc *sc = dev->si_drv1;
        int revents;

        revents = 0;
        sx_slock(&sc->lock);
        if (sc->valid != 0 || sc->writers == 0)
                revents |= events & (POLLIN | POLLRDNORM);
        if (sc->valid < sc->len)
                revents |= events & (POLLOUT | POLLWRNORM);
        if (revents == 0) {
                if ((events & (POLLIN | POLLRDNORM)) != 0)
                        selrecord(td, &sc->rsel);
                if ((events & (POLLOUT | POLLWRNORM)) != 0)
                        selrecord(td, &sc->wsel);
        }
        sx_sunlock(&sc->lock);
        return (revents);
}
```

A pair of generic I/O control commands are also useful for inspecting the status of a file descriptor. `FIONREAD` and `FIONWRITE` return the number of bytes that can be read or written without blocking, respectively. The byte count is returned in a control command argument of type int. Listing 8 shows the support for these I/O control commands in the echo device. Note that the returned value is clamped to `INT_MAX` to avoid overflow.

Listing 8: FIONREAD and FIONWRITE

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

```
        ...
        case FIONREAD:
                sx_slock(&sc->lock);
                *(int *)data = MIN(INT_MAX, sc->valid);
                sx_sunlock(&sc->lock);
                error = 0;
                break;
        case FIONWRITE:
                sx_slock(&sc->lock);
                *(int *)data = MIN(INT_MAX, sc->len - sc->valid);
                sx_sunlock(&sc->lock);
                error = 0;
                break;
        ...
}
```

To make this functionality easier to demonstrate, we have added a new **poll** command to the echoctl utility. This command uses **poll(2)** to query the echo device's current status. If the device is readable, it uses the **FIONREAD** I/O control command to output the number of bytes available to read. If the device is writable, it uses **FIONWRITE** to output the number of bytes that can be written. Example 3 shows a few invocations of this command along with other operations on the echo device. Note that since there is not another writer in this example, the device is readable even while it is empty.

Example 3: Polling I/O Status

```
# echoctl poll
Returned events: POLLIN|POLLOUT
0 bytes available to read
room to write 64 bytes
# echo "foo" > /dev/echo
# echoctl poll
Returned events: POLLIN|POLLOUT
4 bytes available to read
room to write 60 bytes
# cat /dev/echo
foo
# echoctl poll -r
Returned events: POLLIN
0 bytes available to read
```

I/O Status Reporting via kqueue(2)

FreeBSD provides the kqueue(2) kernel event notification facility as a separate API from **select(2)** and **poll(2)**. With **kqueue(2)**, applications register a persistent note in the kernel for each desired event. The kernel generates a stream of events that the application can consume and act upon. Unlike **select(2)** and **poll(2)**, an application does not need to register all the events it cares about each time it wants to wait for a new event. This reduces overhead in applications while also permitting more efficient tracking of desired events in the kernel.

A kernel event consists of a filter (event type) and identifier. The behavior of some events can be further customized via various flags. For I/O on file descriptors, the two primary filters are EVFILT_READ and EVFILT_WRITE to determine if a descriptor is readable or writable, respectively. The identifier field for these event filters is the integer file descriptor. In addition, for read and write events the kernel event structure returns the amount of data that can be read or written as a separate field. This avoids the need for separate invocations of the FIONREAD and FIONWRITE I/O control commands.

Inside the kernel, kernel events are described by a **struct knote** object. This structure contains copies of the event fields that are used to generate the events returned to the application. A list of active events is stored in a **struct knlist** object. Since I/O events handled by select(2) and poll(2) are usually associated with a kernel event, **struct selinfo** embeds a **struct knlist** as its **si_note** member. Each knote is also associated with a **struct filterops** object pointed to by the **kn_fop** member. This structure and the APIs for manipulating knotes and knote lists are described in kqueue(9).

For character devices, the kqfilter method is responsible for attaching a **struct filterops** object to a knote. This includes setting the **kn_fop** member and adding the knote onto the correct knote list. As a result, **struct filterops** objects for character devices do not use the **f_attach** member. The **kn_hook** member of **struct knote** is an opaque pointer that can be set by the kqfilter method to pass state to **struct filteropts** methods similar to the **si_drv1** field of struct cdev.

For the echo driver, we define two **struct filteropts** objects: one for read events and one for write events. Each event includes **f_detach** and **f_event** methods. We reuse the embedded knote list in the existing read and write **struct selinfo** objects from the softc. Since the echo driver uses an sx(9) lock, we define custom locking callbacks for use with **knlist_init()** when creating the echo device. The **f_detach** methods use **knlist_remove()** to remove a knote from the associated knote list. The **f_event** methods set the **kn_data** field to the appropriate byte count and mark the event ready if the byte count is non-zero. The **f_event** method for the read event also sets EV_EOF if there are no writers. A new kqfilter character device method attaches a knote to the new **struct filterops** objects for the EVFILT_READ and EVFILT_WRITE filters. It also sets the **kn_hook** member of the knote to the softc pointer. Finally, all the places in the driver that call **selrecord()** to awaken sleeping threads from **poll(2)** or **select(2)** now also call KNOTE_LOCKED() to report an event for knotes associated with the read or write event. Listing 9 shows the **struct filterops** object and its associated methods for the read filter. Listing 10 shows the new kqfilter character device method.

Listing 9: EVFILT_READ Filter

```
static struct filterops echo_read_filterops = {
      .f_isfd =   1,
      .f_detach = echo_kqread_detach,
      .f_event =  echo_kqread_event
};

...

static void
echo_kqread_detach(struct knote *kn)
```

```
{
	struct echodev_softc *sc = kn->kn_hook;

	knlist_remove(&sc->rsel.si_note, kn, 0);
}

static int
echo_kqread_event(struct knote *kn, long hint)
{
	struct echodev_softc *sc = kn->kn_hook;

	kn->kn_data = sc->valid;
	if (sc->writers == 0) {
		kn->kn_flags |= EV_EOF;
		return (1);
	}
	kn->kn_flags &= ~EV_EOF;
	return (kn->kn_data > 0);
}
```

Listing 10: kqfilter Device Method

```
static int
echo_kqfilter(struct cdev *dev, struct knote *kn)
{
	struct echodev_softc *sc = dev->si_drv1;

	switch (kn->kn_filter) {
	case EVFILT_READ:
		kn->kn_fop = &echo_read_filterops;
		kn->kn_hook = sc;
		knlist_add(&sc->rsel.si_note, kn, 0);
		return (0);
	case EVFILT_WRITE:
		kn->kn_fop = &echo_write_filterops;
		kn->kn_hook = sc;
		knlist_add(&sc->wsel.si_note, kn, 0);
		return (0);
	default:
		return (EINVAL);
	}
}
```

As with the `poll(2)` support, we have extended the echoctl utility with another command to demonstrate the `kevent(2)` support. The new events command registers read and write events for the echo device and outputs a line for each event that is received. Since read and write events are level-triggered by default, echoctl sets the `EV_CLEAR` flag when registering events for the echo device. This instead only reports events when the device state changes triggering a call to KNOTE_LOCKED() inside the driver. Example 4 shows the output of the events command across a series of actions. The first two events are reported as the initial state when the echo device is idle without any open readers or writers. In another shell we execute the command "jot -c -s "" 80 48 > /dev/echo" to write 81 bytes of data to the echo device. Since the default buffer size is 64 bytes, this command blocks in the write(2) system call after writing 64 bytes. The write of 64 bytes triggers the next EVFILT_READ event reporting 64 bytes available to read. Finally, in a third shell we execute the command "cat /dev/echo" to read all the data from the echo device. The first read(2) system call from cat(1) reads the 64 bytes of output and triggers an EVFILT_WRITE event. However, before the echoctl process can query the echo device's state, the jot(1) process has awakened and written the remaining 17 bytes of data to the buffer leaving room for 47 bytes. This accounts for the first EVFILT_WRITE event reported in the third block of events. The write of the remaining 17 bytes also triggered an EVFILT_READ event. However, by the time this event is reported, the jot(1) process has exited and cat(1) has read the remaining 17 bytes, so the EVFILT_READ event reports EV_EOF with zero bytes to read. The read of 17 bytes by cat(1) also triggered an EVFILT_WRITE event that is reported as the next to last event. Finally, cat(1) calls read(2) a third time which returns 0 to signal EOF. This read also triggers an EVFILT_WRITE event which is reported as the last event. This last sequence of events is not deterministic and may appear in a different order or with slightly different values across different runs (for example, the first EVFILT_WRITE may report 64 bytes available to write if jot(1) hasn't yet written the remaining 17 bytes).

Example 4: I/O Status via Kernel Events

```
# echoctl events -W
EVFILT_READ: EV_EOF 0 bytes
EVFILT_WRITE: 64 bytes
...
EVFILT_READ: 64 bytes
...
EVFILT_WRITE: 47 bytes
EVFILT_READ: EV_EOF 0 bytes
EVFILT_WRITE: 64 bytes
EVFILT_WRITE: 64 bytes
```

Conclusion

In this article we extended the echo device to support a FIFO data buffer with both blocking and non-blocking I/O. We also added support for querying device state via poll(2) and kevent(2). The final article in this series will describe how character devices can provide a backing store for memory mappings created via mmap(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.