



Adam Koehler <adam.t.koehler@gmail.com>

attempting to understand etherWrite()

4 messages

Paul Hinze <paul.t.hinze@gmail.com>

Sat, Mar 29, 2008 at 4:52 PM

To: XINU Brain Trust <brylow-xinu@mscs.mu.edu>

XBT,

As I'm catching up with the work my group in embedded systems group has done in the past few days, I found myself going through etherWrite() and trying to understand it. It's really pretty tricky, and the documentation is minimal, so I figured I would bring my exploration into a discussion on The List out of which we can hopefully get some improved comments/docs in the ethernet driver. So if those of you in the know could look this over, address the question marks, and see where my mistakes are, I'd appreciate it.

So, etherWrite(), does some error checking, and then allocates memory for an ethPktBuffer:

[device/ether/etherWrite.c]

```
> epb = (struct ethPktBuffer *)getmem(sizeof(struct ethPktBuffer) + len);
> ASSERT((ulong)SYSERR != (ulong)epb);
```

Given the definition of an ethPktBuffer:

[include/ether.h]

```
> struct ethPktBuffer
> {
>   uchar    *buf;          /**< Pointer to buffer space    */
>   uchar    *data;        /**< Start of data within buffer */
>   ushort   length;       /**< Length of packet data    */
> };
```

This means that epb should point to an area of memory that looks like this:

```
+-----+ 0x00
| buf           |
+-----+ 0x04
| data          |
+-----+-----+ 0x08
| length   |   |
+-----+-----+ ^
|           | | len-sizeof(short)
.           .
.           .
.           .
+-----+-----+ v
```

So we have the buffer information directly in front of the space needed for the packet data. Note that because length is a short, the space after the three fields of the ethPktBuffer is not word aligned. (Right?)

So next we have this mysterious line:

```
[device/ether/etherWrite.c]
```

```
> epb = (struct ethPktBuffer *)(((ulong)epb) | 0x20000000);
```

ORing on one bit of a freshly `getmem()`ed pointer, interesting...

IIRC, the address returned from `getmem()` will be in the `0x8--` range, and therefore this line will convert it into `0xA--` space. The effect of this statement is to have the driver address `KSEG1` (uncached memory), instead of `KSEG0`, correct? I seem to remember this being a key breakthrough in the ethernet driver development. Why, again?

Next we fill in the fields of the `ethPktBuffer`:

```
[device/ether/etherWrite.c]
```

```
> epb->length = len;
> epb->buf = (uchar *) (epb + 1);
> epb->data = epb->buf;
> memcpy(epb->buf, buf, len);
```

Now this part is confusing to me. Storing the length is straightforward, but why does `buf` point one character into the struct? And then `data` points to the same place as `buf`? Or does `data` get the value stored at the modified `buf` pointer? My pointer headache is getting to me, so let me see if I can illustrate what I think is happening and let you guys correct me. After this block, as far as I can tell, the pointers look like this:

```
0x01
+--+-----+
|  ||  |
| vv  |
+-----+-----+-----+-----...+
| buf  | data | le |          |
+-----+-----+-----+-----...+
|      |      | |<-- len ...>
0x00  0x04  0x08 0x10
```

Seems rather odd to me... why have two pointers pointing to the middle of a field? In my mind it would make more sense to have `buf` point to `(ethPktBuffer *) (epb + 1)` which would be the start of the allocated buffer. Thoughts?

Anyways, moving on to the final chunk of code:

```
[device/ether/etherWrite.c]
```

```
> ps = disable();
> // Check for buffer space
```

Firstly, the "Check for buffer space" comment doesn't seem to apply. What happens if there is no buffer space? Either way we end up returning 0.

```
[device/ether/etherWrite.c]
```

```
> entry = peth->txTail;
> peth->txBufs[entry] = epb;
```

So `txTail` is an index into the `txBufs` table which is an array of `ethPktBuffer` POINTERS (`ethPktBuffer **`). We store the value of our pointer in that array.

Next the control field is set up:

```
[device/ether/etherWrite.c]
```

```
> control = len & DESC_CTRL_LEN;
> // Mark as start and end of frame, interrupt on completion.
> control |= DESC_CTRL_IOC | DESC_CTRL_SOF | DESC_CTRL_EOF;
```

From the header file I got a definition for the field that looks like this:

```
+---+-----+-----+
|bits|core spfc bits | length  |
+---+-----+-----+
 ^^^^
||||
|||+--- End of Table
||+---- Interrupt on Completion
|+----- End of Frame
+----- Start of Frame
```

My only concern here is the DESC_CTRL_LEN mask which is 0x00001fff. If the field can properly store a short, shouldn't this be 0x0000ffff?

```
[device/ether/etherWrite.c]
> if (RING_MASK == entry)
> { control |= DESC_CTRL_EOT; }
```

RING_MASK is 0x01FF (511d, the same as the TX_RING_ENTRIES) and is labeled "Field size for ring indices." I would assume that this tells the card when to wrap the txTail index. If this is the case, why use RING_MASK when you could just use TX_RING_ENTRIES?

```
[device/ether/etherWrite.c]
> /* Add this buffer to the Tx ring. */
> peth->txRing[entry].control = control;
> peth->txRing[entry].address =
> ((ulong)epb) + 2 * EP_ALEN & 0x1FFFFFFFUL;
```

txRing is an array of dmaDescriptor structures, which have two pointers: control and address. The address assignment here is particularly tricky. We jump (2 * EP_ALEN) = 12 longs past the epb pointer and then mask off the top 3 bits. I would guess that the mask is because the card will translate the address into the appropriate KSEG?

```
[device/ether/etherWrite.c]
> peth->txTail = (entry + 1) & RING_MASK;
```

Here we increment the txTail pointer, with a rather cheap (or clever, depending on your leanings) trick that relies on the fact that once the limit is reached (512), the rest of the bits will all be 0s. This invalidates my guess above about telling the card about the end of the ring... if we wrap the ring then why does the card have to know?

```
[device/ether/etherWrite.c]
> entry = peth->txTail;
> pccsr->dmaTxLast = entry * sizeof(struct dmaDescriptor);
>
> restore(ps);
> return 0;
```

dmaTxLast is located in the card's control and status registers and labeled "Tx Last Posted Desc", so this would be a byte index into the descriptor ring that references (now that we just incremented entry) the next available descriptor? Crazy.

So, as you can see... lots of questions. I'm sure almost all of them have simple "Duh, Paul" answers, but hey, that's what mailing lists

Adam Koehler <adam.t.koehler@gmail.com>**Sat, Mar 29, 2008 at 5:09 PM**

To: Paul Hinze <paul.t.hinze@gmail.com>

Cc: XINU Brain Trust <brylow-xinu@mscs.mu.edu>

Next we fill in the fields of the ethPktBuffer:

```
[device/ether/etherWrite.c]
```

```
> epb->length = len;
> epb->buf = (uchar *)(epb + 1);
> epb->data = epb->buf;
> memcpy(epb->buf, buf, len);
```

Now this part is confusing to me. Storing the length is straightforward, but why does buf point one character into the struct? And then data points to the same place as buf? Or does data get the value stored at the modified buf pointer? My pointer headache is getting to me, so let me see if I can illustrate what I think is happening and let you guys correct me.

Okay so I will just put it out there that I could be drop dead wrong. But shoot me for trying :). For now I will just tackle some of these questions that pertain to this block of code.

Since epb is a point to an ether packet buffer the addition is pointer addition. It doesn't go one character into the struct it goes a whole ether packet buffer past the address where epb is currently. And then that memory address is cast as a uchar *. From the drawings you put up this is where the data should start, which makes the next line valid because it assigns the location to the epb->data. After which the contents of the buffer are memcopied into the location where data starts.

I don't know if you already understood this, the diagrams didn't translate well into my mail I don't think... or I just can't read them correctly. Either way feel free to comment on the validity of my conclusion.

Adam

Paul Hinze <paul.t.hinze@gmail.com>**Sat, Mar 29, 2008 at 5:13 PM**

To: Adam Koehler <adam.t.koehler@gmail.com>

On Sat, Mar 29, 2008 at 5:09 PM, Adam Koehler <adam.t.koehler@gmail.com> wrote:

```
>
> > Next we fill in the fields of the ethPktBuffer:
> >
> > [device/ether/etherWrite.c]
> > > epb->length = len;
> > > epb->buf = (uchar *)(epb + 1);
> > > epb->data = epb->buf;
> > > memcpy(epb->buf, buf, len);
```

```
> Since epb is a point to an ether packet buffer the addition is pointer
> addition. It doesn't go one character into the struct it goes a whole ether
> packet buffer past the address where epb is currently. And then that memory
> address is cast as a uchar *. From the drawings you put up this is where
> the data should start, which makes the next line valid because it assigns
> the location to the epb->data. After which the contents of the buffer are
> memcopied into the location where data starts.
```

You're right, Adam. I misread the parentheses. So both data and buf point to the beginning of memory just after the struct. Thanks for catching that. The question that remains for that chunk is then: what is the difference between data and buf here?

pt

Dennis Brylow <brylow@mscs.mu.edu>

Sat, Mar 29, 2008 at 6:16 PM

To: XINU Brain Trust <brylow-xinu@mscs.mu.edu>

Quoting Paul Hinze <paul.t.hinze@gmail.com>:

> So we have the buffer information directly in front of the space
> needed for the packet data. Note that because length is a short, the
> space after the three fields of the ethPktBuffer is not word aligned.
> (Right?)

Actually, no. On Mips, sizeof(struct ethPktBuffer) == 12. The compiler pads out the structure for word alignment. On other architectures, (I'm talking about you, Pentium,) that might not be the case, but then those architectures care much less about word alignment so it still works out.

> So next we have this mysterious line:

>
> [device/ether/etherWrite.c]
> > epb = (struct ethPktBuffer *)(((ulong)epb | 0x20000000);
>
> ORing on one bit of a freshly getmem()ed pointer, interesting...
>
> IIRC, the address returned from getmem() will be in the 0x8-- range,
> and therefore this line will convert it into 0xA-- space. The effect
> of this statement is to have the driver address KSEG1 (uncached
> memory), instead of KSEG0, correct? I seem to remember this being a
> key breakthrough in the ethernet driver development. Why, again?

If the packet buffer pointer is in cached memory, then it is possible that the contents of a posted packet will still be (partially) loitering in the cache when the DMA engine is reading memory to transmit the packet. By having the pointer in uncached memory, the contents we write to that memory are guaranteed to have actually been written to memory when we hand the packet buffer over to the DMA engine.

> Now this part is confusing to me. Storing the length is
> straightforward, but why does buf point one character into the struct?

See Adam's answer -- he is correct.

> And then data points to the same place as buf?

Yes.

> Firstly, the "Check for buffer space" comment doesn't seem to apply.

No. The comment is a placeholder for a check that does not yet exist. At present, a demanding process could fill up all of the transmit buffers faster than the NIC could transmit them. There should be a check for that, and it should go before we march on ahead and start putting the packet on the ring.

> What happens if there is no buffer space? Either way we end up
> returning 0.

I'd say it (the driver) should drop the packet and increment an overflow counter.

> My only concern here is the DESC_CTRL_LEN mask which is 0x00001fff.
> If the field can properly store a short, shouldn't this be 0x0000ffff?

The 0x1FFF mask came from the Linux source. Not sure why the NIC designers would go for a 13-bit field here, but we have no evidence of it being any other size, larger or smaller.

think this field could store a short -- Ethernet MTU is generally 1500, and

> RING_MASK is 0x01FF (511d, the same as the TX_RING_ENTRIES) and is
> labeled "Field size for ring indices." I would assume that this
> tells the card when to wrap the txTail index. If this is the case,
> why use RING_MASK when you could just use TX_RING_ENTRIES?

RING_MASK is when the card *must* wrap, because it can store no ring index larger than RING_MASK. TX_RING_ENTRIES could be a different value, and in fact is substantially smaller in OpenWRT Linux. TX_RING_ENTRIES must be less than or equal to RING_MASK.

```
>
> [device/ether/etherWrite.c]
> > /* Add this buffer to the Tx ring. */
> > peth->txRing[entry].control = control;
> > peth->txRing[entry].address =
> >     (((ulong)epb) + 2 * EP_ALEN) & 0x1FFFFFFFUL;
>
> txRing is an array of dmaDescriptor structures, which have two
> pointers: control and address. The address assignment here is
> particularly tricky. We jump (2 * EP_ALEN) = 12 longs past the epb
> pointer and then mask off the top 3 bits.
```

You're wrong, but so am I. Because the typecast of epb is to ulong, (not ulong *,) it is just a number, so it is an unscaled 12 bytes past the epb pointer. That happens to be the correct answer in this case, but the calculation is wrong. The address field should be assigned simply

```
peth->txRing[entry].address = epb->data & 0x1FFFFFFFUL;
```

The address should point to the start of the packet data, which also happens to be 12 bytes (2 * EP_ALEN) past the start of the packet buffer structure.

> I would guess that the mask is because the card will translate the address
> into the appropriate KSEG?

I would say that the DMA engine requires a physical address, so we strip off all of the KSEG* bits.

```
> [device/ether/etherWrite.c]
> > peth->txTail = (entry + 1) & RING_MASK;
>
> Here we increment the txTail pointer, with a rather cheap (or clever,
> depending on your leanings) trick that relies on the fact that once
> the limit is reached (512), the rest of the bits will all be 0s. This
> invalidates my guess above about telling the card about the end of the
> ring... if we wrap the ring then why does the card have to know?
```

Our internal tail pointer and the card's tail pointer need to wrap in the same way, or they get out of sync. The "& RING_MASK" thing is rather cheap, but actually models correctly how the card represents this field. We cannot post a tail pointer with more bits than RING_MASK.

> dmaTxLast is located in the card's control and status registers and
> labeled "Tx Last Posted Desc", so this would be a byte index into the
> descriptor ring that references (now that we just incremented entry)
> the next available descriptor? Crazy.

If the DMA engine is already caught up (i.e. waiting for the next packet

to transmit,) this last assignment to `pecsr->dmaTxLast` apparently tells the card that it may advance to the next-to-last descriptor.

Looking at this originally, I was pretty sure this was an off-by-one error, but without first incrementing "entry", the card does not transmit my most recent packet until I post the next one. (Ping times in the 1 second range, as a result.) It is a strange device.

-D