Lion’s Chapters 16 & 17

By Pavel Babenko and Michael Buchoff

The RK Disk Driver
Introduction

- RK disk storage consists of
  - disk controller RK11-D
  - a number of RK disk drives, up to eight for each disk controller
  - Removable disk cartridges
- This disk storage is most used in PDP11 systems

Disk format

- Surfaces: 2
- Tracks/surface: 200
- Sectors/track: 12
- Bytes/Sector: 512
- Total of 2.4M Capacity
RK11 Hardware controller

- Contains total of 7 hardware registers

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk address</td>
<td>in</td>
</tr>
<tr>
<td>Word count</td>
<td>in</td>
</tr>
<tr>
<td>Bus address</td>
<td>in</td>
</tr>
<tr>
<td>Control status</td>
<td>in/out</td>
</tr>
<tr>
<td>Drive status</td>
<td>out</td>
</tr>
<tr>
<td>Error</td>
<td>out</td>
</tr>
<tr>
<td>Data buffer</td>
<td>in/out</td>
</tr>
</tbody>
</table>

Hardware registers

- Disk address (in) – address of block on disk to read/write
- Word count (in) – number of 2-bytes to read/write
- Bus address (in) – memory location for data used in read/write operations
Hardware registers (cont.)

Control status register

- (in) Operation flags
  - The type of operation: read/write/reset
  - Generate interrupt upon completion or not
  - Operation start bit
- (out) Status information
  - Ready flag
  - Error flag

There are two more registers used by UNIX:

- Drive status register holds information on drive condition after operation
- Error status register holds drive error code

In case of error, UNIX prints error status and drive status
PDP11 memory model

- In PDP11 hardware registers use the same address space as memory.
- For example, memory may be located at addresses 0-010000 and hardware registers located at 012000.
- Hardware registers for RK11-D controller are located at memory address base of 0177400 (octadecimal).

UNIX mounted device concept

- After removable cartridge is plugged in, UNIX operator has to 'mount' this drive with mount command.
- After this, drive gets device number and device parameter records. Files and directories on drive are linked to some UNIX filesystem subdirectory.
- Drive has to be dismounted after use.
RK driver software model

- Each UNIX block device has a queue of pending IO operations. Each operation is defined by one IO buffer.
- Each block device has an associated devtab structure.

```
struct devtab {
    char d_active;
    struct buf **d_actf;
    struct buf **d_actl;
    ... 
}
```

d_active indicates if device is currently busy.
d_actf and d_actl point to the beginning and end of pending IO queue.

Overview of IO buffer structure

- IO buffer contains:
  - Device name
  - Memory address to read/write data
  - Number of bytes to read/write
  - Number of block on device to access
  - Operation flags: operation type (read/write), whether operation asynchronous or not etc.
  - Pointer to the next buffer in queue
IO operation start: rkstrategy()

- Function rkstrategy(buf *bp) starts IO operation
- It adds buffer bp to IO queue
- Then it checks if device is now busy
- If it is busy, it does nothing
- If it is not, it starts device operation with rkstart() function

```c
rkstrategy(abp)
struct buf *abp;
{
    register struct buf * bp;
    bp = abp;
    ...
    /* add buffer bp to IO queue */
    if (rktab.d_actf == 0)
        rktab.d_actf = bp;
    else
        rktab.d_actl->av_forw = bp;
        rktab.d_actl = bp;
    /* start device operation */
    if (rktab.d_active == 0)
        rkstart();
}
```
rkstart() and devstart() functions

- **rkstart()** checks if there are IO operations in queue
  - If yes, **rkstart()** sets busy flag and calls devstart()
- **devstart()** is responsible for executing one IO operation.
  - It takes next buffer from IO queue.
  - It loads controller registers with data from buffer.
  - It sets interrupt flag of status register.
  - It sets GO flag of status register => Hardware begins executing this IO operation

```c
#define RKADDR 0177400
struct {
    int rksr;
    int rker;
    int rksr;
    int rkcr;
    int rkwcr;
    int rbak;
    int rkda;
} rkstart() {
    register struct buf * bp;
    /* if queue is empty then return */
    if ((bp = rktab.d_actfrktab.d_actf) == 0) return;
    /* set busy flag and call devstart() */
    rktab.d_active ++;
    devstart(bp, &RKADDR->rkda, rkaddr(bp), 0);
}
/* rkaddr is an auxiliary function that, given linear block number on disk, returns coded sector/track number information in device format */
```
devstart(bp, devloc, devblk, hbcom)
struct buf *bp;
int *devloc;
{
    register int *dp;
    register struct buf *rbp;
    register int com;

dp = devloc;    /* contains the upper, rkda port number */
*dp = devblk;   /* rbp now points to the next buffer */
*--dp = rbp->b_addr; /* track/sector number sent to device */
*--dp = rbp->b_wcount; /* number of bytes to transfer sent to device */
com = (hbcom << 8) | IENABLE | GO | ((rbp->b_xmem & 03) << 4);
if (rbp->bflags & B_READ)
    com |= RCOM;
else
    com |= WCOM;
    /* status register is set, operation is started! */
}
Operation completed

- UNIX function devstart() always requests interrupt to occur upon completion of IO operation
- RK hardware uses interrupt vector 220 from interrupt vector table
- RK interrupts are handled by function rkintr()

rkintr() function

- clears busy flag from device
- checks for IO errors
- If error happened, it executes the same IO operation for up to 10 times
- Otherwise, it removes used buffer from IO queue and calls iodone() function on removed buffer
- Then, it AGAIN calls rkstart() to process all other IO requests left in IO queue
rkintr()
{
    register struct buf *bp;

    if (rktab.d_active == 0) /* if operation is not started, exit */
        return;

    bp = rktab.d_actf; /* bp points to executed buffer */
    rktab.d_active = 0; /* clear busy flag */

    if (RKADDR->rkc < 0) { /* if error bit is set */
        ...
        RKADDR->rkc = RESET | GO;
        ...
        if (++rktab.d_errcnt <= 10) { /* try to repeat faulty operation up to 10 times */
            rkstart();
            return;
        }
    }
    bp->b_flags |= B_ERROR; /* if still experiencing an error, give it up */
    rktab.d_errcnt = 0; /* clear error repeat count flag */

    rktab.d_actf = bp->av_forw; /* remove this IO buffer from queue */
    iodone(bp); /* do some postprocessing on removed buffer */
    rkstart(); /* start operation again, for the next buffer */
}

iodone() function

- If operation was asynchronous, it releases used buffer by adding it to unused buffers list
- If operation was synchronous, it awakes the process that requested the operation
- If not, the process is responsible for releasing the buffer
io-done(bp)
struct buf *bp;
{
    register struct buf *rbp;

    rbp = bp;               /* rbp is a buffer which operation is finished */
    ...
    rbp->b_flags |= B_DONE; /* mark operation as done */

    if (rbp->b_flags & B_ASYNC)
        brelse(rbp);        /* in asynchronous mode, release buffer */
    else {
        rbp->b_flags &= ~B_WANTED;
        wakeup(rbp);        /* in synchronous mode, wakeup a process */
    }
}

Operation flowchart

rkstrategy(bp)  rkstart()
                devstart()
                        IO is in process...
                                Interrupt: rkintr()
Chapter 17

IO buffers

Chapter 17 - Buffers

- buf structure
- Buffer functions
  - clrbuf
  - incore
  - getblk
  - Init
  - bread, bwrite, bflush
What is a buffer?

A buffer is an area of memory used for storing messages.

How are buffers useful?

Example: We wish to write 5 bytes to a disk.

Method 1:

Method 2 (using buffers):
Major drawback of buffers

- Heavy memory requirements
- (Although a few buffers no big deal, a few hundred is)

Programming newbie – 1 buffer for every possible use

Experienced UNIX programmers – Have a pool of buffers ready for arbitrary use

(avn-list)

How are do we implement a buffer?

- Technically, we need nothing more than a chunk of data and its length.

- We will use chunks of 514 bytes
  
  \[ \text{4720} \ \text{char buffers[NBUF][514]} \]

  (Where NBUF = 15)
Linked Lists

The complicated part

- Each buffer will have a header
  
  ```c
  struct buf {
    // Buffer & length
    // Double-linked lists
    // Flags (reading, writing, etc.)
    // Which device and where on device
    // Error information
  } buf[NBUF];
  ```
The complicated part

- Each buffer will have a header

```c
struct buf {
    int b_flags; /* see defines below */
    struct buf *b_forw; /* headed by devtab of b_dev */
    struct buf *b_back; /* */
    struct buf *av_forw; /* position on free list */
    struct buf *av_back; /* if not BUSY */
    int b_dev; /* major+minor device name */
    int b_wcount; /* transfer count (usu. words) */
    char *b_addr; /* low order core address */
    char *b_xmem; /* high order core address */
    char *b_blkno; /* block # on device */
    char *b_error; /* returned after I/O */
    char *b_resid; /* words not transferred after error */
} buf[NBUF];
```

- `clrbuf(struct buf *bp)`

  Clears out the first 512 bytes (256 words) of the buffer
**clrbuf(struct buf *bp)**

```c
5038 clrbuf (bp)
5039 int *bp;
5040 {
5041     register *p;
5042     register c;
5043     p = bp->b_addr;
5044     c = 256;
5045     do
5046         *p++ = 0;
5047     while (--c);
5049 }
```

**incore(adev, char *blkno)**

- Searches for a buffer with a matching device number of *adev* and a block of *blkno*.
- Returns the buffer if it finds a match
- Else, returns 0
incore(ad, char *blkno)

```c
incore(ad, blkno)
{
    register int dev;
    register struct buff *bp;
    register struct devtab *dp;

    dev = ad;
    dp = bdevsw[ad.d_major].d_tab;
    for (bp = dp->b_forw;
        bp != dp;
        bp = bp->b_forw)
    {
        if (bp->b_blkno == blkno &
            bp->b_dev == dev)
            return(bp);
    }
    return(0);
}
```

getblk(dev, char *blkno)

- If there is a match, return it.
- Otherwise, search for the oldest non-busy block and allocate it.
getblk(dev, char *blkno)

1. Start from DEVTAB and search each buffer until we end up with a match (or back where we started from)

2. If there are no free nodes, wait until there is one

3. Use the next free buffer. (this one)

DEVTAB

buffer

buffer

bfreelist

...
getblk(dev, char *blkno)

Code will return here every time something changes

Get the devtab from the device

Iterate through the circularly linked list
If there is no match, check the next one
If the match we found is busy, mark it as wanted and sleep until something has happened to it. Then try again.

Mark it as ours and return

If there are no free elements, tell the OS we want one and sleep until it becomes available. Then, try again.

Mark the free element as ours.
If we need to write out, go ahead and try again.
Mark this flag as busy and unused.
Remove this from the av-list and insert it into the device linked list
Set the device information
Return the new buffer
binit()

```c
binit()
{
    register struct buf *dp;
    register struct devtab *dp;
    register int i;
    struct bdevsw *bdp;

    bfreelist_b_forw = bfreelist_b_back = 
bfreelist_av_forw = bfreelist_av_back = &bfreelist;
    for (i = 0; i < NBUF; i++) {
        bp = &buf[i];
        bp->a_dev = -1;
        bp->a_addr = buffers[i];
        bp->a_back = &bfreelist;
        bp->a_bsize = bfreelist.b_bsize;
        bfreelist_b_forw = &bp;
        bfreelist_av_forw = &bp;
        brelse(bp);
    }
    for (bdp = bdevsw; bdp->d_opend; bdp++) {
        dp = bdp->d_tab;
        if(dp) {
            dp->b_forw = dp;
            dp->b_back = dp;
        } ++;
    }
    nblkdev = i;
}
```

Other Functions

- **bread (buffer read, not the food)** – Pass it a device number and an address in that device, it will read it and return the buffer

- **bwrite (buffer write)** – Pass it a buffer and it writes it out to the device

- **bflush (buffer flush)** – Pass it a device and it writes out all the buffers
Conclusions

- RK disk driver, devtab structure
  - rkstrategy, rkstart, devstart, rkintr, iodone
- buf structure
- Buffer functions
  - clrbuf
  - incore
  - getblk
  - Init
  - bread, bwrite, bflush

Works Cited

http://www.dictionary.com
http://www.snopes.com/common/computer/disk.gif

Lions’ Commentary on UNIX 6th Edition
(With Source Code)
Questions?