Linux Hardware I/O via `Mmap(2)`

Introduction to User-Space Hardware I/O

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer

Overview

Roadmap:

- Quick overview of *protected memory*
- “User mode” vs. “kernel mode”
- The `const` and `volatile` keywords
- The `mmap(2)` system call
- Security implications
- Code examples

What is “Protected Memory”?

Many Linux device drivers:

- Run from kernel memory
- Have no direct access to or from user memory

An *interface*:

- Bridges the gap between user and kernel memory
- Under Linux, implementation relies on a *device node*
- (We’ll come back to this in detail later)

What is “Protected Memory”?

Protected memory:

- The MMU keeps memory spaces separated
- A exception occurs if you go out-of-bounds

When an exception occurs:

- The kernel kills the user process, and/or,
- The kernel generates an OOPS message

What is “Protected Memory”?

What is “Protected Memory”?

User mode device drivers:

- User programs that access hardware

“But how can that be?!”

- Stay tuned! :)

Advantages and Disadvantages

Why run a driver in a user application?

- No kernel code required
- Potentially less risk to system stability
- Easier to manage driver development

Advantages and Disadvantages

Potentially less risk?

- You can still command the hardware to do something stoopid
- Interrupt hangs will (probably)(not) hang the kernel
- Linux will protect you from wild pointer dereferencing

Advantages and Disadvantages

Doesn't promote "mainlining":

- User applications don't go in kernel source code!

No support for interrupt handlers:

- Requires a small amount of kernel code
- The majority *can* be in user memory, however

Advantages and Disadvantages

Performance:

- Depends on specifics of implementation
- May be better or worse than kernel code

Security Implications

"Is this a security risk?"

- No.
- Only privileged users will get access to device memory
- Others will be denied access per Linux security models
- Your system is no more or less secure than with alternatives

Two Approaches

`/dev/mem` and `memmap(2)`

- Ye olde skool way
- Straightforward, works well
- Supported by even archaic kernel versions
- Perfect for polled, slow devices

Two Approaches

`#include<linux/uio_driver.h>`

- Sysfs API for device-related information
- Builds on the `memmap()` approach
- Ideal for pluggable devices, esp. PCI
- Relatively new addition to the kernel (2.6.18-ish)

(We’ll come back to this later)

Two Approaches

"Which one do I use?"

- Long-lived drivers probably need `uio_driver` API
- It's overkill for simple polling, however

The `/dev/mem` Device

The `/dev/mem` device:

- A *device node*
- Allows users to `mmap()` a physical address
- The `read()` and `write()` methods don't work

```
void *mmap(void *addr, size_t length, int prot
           int flags, int fd, off_t offset);
```

The `/dev/mem` Device

To control hardware:

- Call `open("/dev/mem", ...);`
- Use `mmap(2)` to map the device's control registers
- Use pointer dereferencing to drive the device as always

The /dev/mem Device

```
int main (int argc, char *argv[])
{
  /* mknod /dev/mem c 1 1 */
  /* crw-r--r--  1 root root 1, 1 2006-02-27 10:49 /dev/mem */
  int fd = open("/dev/mem", O_RDWR | O_SYNC);

  if (!fd)
    {
      perror("fd");
      return -1;
    }

  unsigned long gpio
    = (unsigned long)mmap(0, 0x1000, PROT_READ | PROT_WRITE,
                          MAP_SHARED, fd, GPIO);
```

The `/dev/mem` Device

```
unsigned long gpiod = gpio + GPIOD;
printf("gpiod mapped to %lx\n", gpiod);

volatile unsigned int* perd = PIO_PER(gpiod);
volatile unsigned int* psrd = PIO_PSR(gpiod);
volatile unsigned int* oerd = PIO_OER(gpiod);
volatile unsigned int* osrd = PIO_OSR(gpiod);
volatile unsigned int* sodrd = PIO_SODR(gpiod);
volatile unsigned int* codrd = PIO_CODR(gpiod);
volatile unsigned int* pdsrd = PIO_PDSR(gpiod);
```

The `/dev/mem` Device

```
#define LED (1 << 4) /* PD4 */

  *perd = LED;
  *oerd = LED;
  printf("psrd: %x osrd: %x\n", *psrd, *osrd);

  *codrd = LED;
  *sodrd = LED;
```

Manpage

```
MMAP(2)              Linux Programmer's Manual              MMAP(2)

NAME
    mmap, munmap - map or unmap files or devices into memory

SYNOPSIS
    #include <sys/mman.h>

    void  *mmap(void *start, size_t length, int prot,
                int flags, int fd, off_t offset);
    int munmap(void *start, size_t length);

DESCRIPTION
    The mmap() function creates a new mapping in the virtual address
    space of the calling process.  The starting address for the new
    mapping is specified in addr.  The length argument specifies the
    length of the mapping...

RETURN VALUE
    On success, mmap() returns a pointer to the mapped area.  On
    error, the value MAP_FAILED (that is, (void*) -1) is returned...
```

Parameters

`PROT_READ`

- Request read permissions to the mapped memory

`PROT_WRITE`

- Request write permissions to the mapped memory

Parameters

`MAP_SHARED`

- Request a “shared” mapping
- Updates are visible to other processes
- Updates are carried through to the underlying device
- (No other option works for device memory)

```
unsigned int *p;
p = mmap(0, 0x1000, PROT_READ | PROT_WRITE,
         MAP_SHARED, fd, 0xffff f000);
```

Parameters

`O_SYNC`

- Use in `open(1)`
- Indicates *nocache*, which is probably what you want

```
int fd;
fd = open("/dev/mem", O_RDWR | O_SYNC);
```

Reading and Writing the Hardware

Use normal C-style pointers:

- Dereference to read or write
- Model banks of registers as arrays
- Don't forget the `volatile` and `const` keywords!

```
int volatile const * volatile p;
```

Reading and Writing the Hardware

Watch out for endianness and alignment!

- Know the alignment restrictions of your host processor
- Know the alignment restrictions of the target device
- Be vigilant for data representation assumptions

- Scrutinize the compiler's assembly language carefully!

mmap-gpio-csb737.c

```
#define GPIO  0xfffff000UL
#define GPIOA 0x200UL
#define GPIOD 0x800UL
```

```
#define PIO_PER(p)  ((volatile unsigned int*)((unsigned long)(p) + 0))
#define PIO_PDR(p)  ((volatile unsigned int*)((unsigned long)(p) + 4))
#define PIO_PSR(p)  ((volatile unsigned int*)((unsigned long)(p) + 8))
#define PIO_OER(p)  ((volatile unsigned int*)((unsigned long)(p) + 0x10))
#define PIO_ODR(p)  ((volatile unsigned int*)((unsigned long)(p) + 0x14))
#define PIO_OSR(p)  ((volatile unsigned int*)((unsigned long)(p) + 0x18))
#define PIO_SODR(p) ((volatile unsigned int*)((unsigned long)(p) + 0x30))
#define PIO_CODR(p) ((volatile unsigned int*)((unsigned long)(p) + 0x34))
#define PIO_PDSR(p) ((volatile unsigned int*)((unsigned long)(p) + 0x3c))
```

mmap-gpio-csb737.c

```c
int main (int argc, char *argv[])
{
  /* mknod /dev/mem c 1 1 */
  /* crw-r--r--  1 root root 1, 1 2006-02-27 10:49 /dev/mem */
  int fd = open("/dev/mem", O_RDWR | O_SYNC);

  if (!fd)
    {
      perror("fd");
      return -1;
    }

  unsigned long gpio
    = (unsigned long)mmap(0, 0x1000, PROT_READ | PROT_WRITE,
                          MAP_SHARED, fd, GPIO);
```

mmap-gpio-csb737.c

```c
    unsigned long gpiod = gpio + GPIOD;
    printf("gpiod mapped to %lx\n", gpiod);

    volatile unsigned int* perd = PIO_PER(gpiod);
    volatile unsigned int* psrd = PIO_PSR(gpiod);
    volatile unsigned int* oerd = PIO_OER(gpiod);
    volatile unsigned int* osrd = PIO_OSR(gpiod);
    volatile unsigned int* sodrd = PIO_SODR(gpiod);
    volatile unsigned int* codrd = PIO_CODR(gpiod);
    volatile unsigned int* pdsrd = PIO_PDSR(gpiod);
```

mmap-gpio-csb737.c

```
#define LED (1 << 4) /* PD4 */

  *perd = LED;
  *oerd = LED;
  printf("psrd: %x osrd: %x\n", *psrd, *osrd);

  *codrd = LED;
```

mmap-gpio-csb737.c

```
    munmap((void*)gpio, 0x1000);
    close(fd);

    return 0;
```

mmap-gpio-csb737.c

```
# gcc -g -Wall -o csb737 mmap-gpio-csb737.c

# ./csb737
psrd: 1ea503b7 osrd: 1ea503b7
...
```

Caveat

The previous example is a Bad Idea:

- Other drivers are using GPIO
- We risk concurrent access issues

Well, not entirely true:

- The hardware naturally prevents some problems
- (Important details are left as an exercise)

Recap

Protected memory:

- Separates kernel and user memory spaces
- Prevents direct access to hardware
- “User mode” vs. “kernel mode”

Recap

The `mmap(2)` system call:

- Returns a pointer to physical memory
- Must be a privileged user process
- Use the `volatile` keyword

Code example:

- CSB737 GPIO controller

Recap

Advantages:

- “Device drivers” are ordinary applications
- Probably easier to develop and debug
- Potentially better application integration

Disadvantages:

- Modest performance hit

Linux Hardware I/O via `Mmap(2)`

Introduction to User-Space Hardware I/O

Bill Gatliff
bgat@billgatliff.com

Freelance Embedded Systems Developer

Demonstration

CSB737 LED:

- Blink from a user application
- (Using `mmap(2)`, not `gpiolib`)
- See `mmap-gpio-csb737.c` for code

Assignment

Questions:

- How does the AT91SAM9263 GPIO controller work?
- What features minimize the risks of concurrent access?

Questions:

- How do you configure pins as inputs, outputs?
- How do you prevent glitches during configuration?

Assignment

Repeat the demonstration:

- Build, run `mmap-gpio-csb737.c`
- Verify that it works as expected
- Review, understand the code

Assignment

Read the pushbutton:

- Use it to vary the blink rate of the LED