Gnu Rabbit

Code for GENNUM and White Rabbit
August 2010

Alessandro Rubini (rubini@gnudd.com)
Work sponsored by CERN (www.cern.ch)

Introduction

This package includes a device driver for the GN4124 PCI-E board and a module for raw PCI I/O, used at CERN as development tool to help prototyping White Rabbit hardware and software.

1 Driver for GN4124

Still to be done, I'm sorry. Stay tuned.

2 Raw PCI I/O

The kernel module for raw I/O is called *rawrabbit*. After running *make* you'll find a file called `rawrabbit.ko`.

To compile you may optionally set the following three variables in your environment:

`CROSS_COMPILE`
: The variable defaults to the empty string, used for native compilation

`ARCH`
: The variable defaults to the build architecture

`LINUX`
: This is the location of the kernel source against which you are compiling. It defaults to the place where the currently running kernel has been compiled (assuming it was compiled on this same system).

The code has been run-tested on version 2.6.34, and compile-tested on 2.6.24.7-rt27, 2.6.29.4-rt15, 2.6.31.6-rt19, 2.6.31.12-rt21.

The module creates a *misc* char device driver, with major number 10 and minor number 42. If you are running *udev* the special file `/dev/rawrabbit` will be created automatically.

Warning: future releases of this package may change the device number or switch to several devices, I'm yet undecided on this choice.

2.1 General features of rawrabbit

The driver is designed to act as a misc device (i.e. a char device) towards user programs and as a PCI driver towards hardware, declaring the pair *vendor*/*device* it is able to drive.

The pair of identifiers is predefined at compile time but can be changed at run time. The defaults (set forth in `rawrabbit.h` refer to the GN4124 evaluation board. The values can also be changed at module load time by setting the `vendor` and `device` arguments. For example the following command sets *rawrabbit* to look for a 3com Ethernet device:

```
insmod rawrabbit.ko vendor=0x10b7 device=0x9055
```

When the driver is loaded it registers as a PCI driver for the preselected vendor/device pair, but loading succeeds even if no matching peripheral exists on the system, as user space programs can request a different vendor/device pair at runtime. Since a single bus might host several instances of the same peripheral, user space programs can also specify the *bus* and *devfn* values in order to select a specific instance of the hardware device. Similarly, the pair *subvendor*/*subdevice* may be specified.

User programs can use *read* and *write*, *mmap* and *ioctl* as described later (**Note:** mmap is not currently supported). Each and every command refers to the device currently selected by means of the vendor/device pair as well as bus/devfn and/or subvendor/subdevice if specified.

The driver allows access to the PCI memory regions for generic I/O operations, as well as some limited interrupt management in user space. Programs can also access a DMA buffer, for which they can know the physical address on a page-by-page basis.

In the source file, each global function or variable declared in the file itself or in the associated header has `rr_` as prefix in the name, even if its scope is static. Local variables have simple names with no prefix, like `i` or `dev`. This convention is followed so when reading random parts of the source you can immediately know whether the symbol is defined in the same file (like `rr_dev`) or is an external Linux resource (like `pci_driver`).

2.2 Interrupt management

The driver is able to handle the interrupt for the active device. User space is allowed to wait for an interrupt and acknowledge it later at will. To allow this latency, the driver disables the interrupt as soon as it is reported, so user-space can do the board-specific I/O before asking to re-enable the interrupt.

The interrupt handler is registered as a shared handler, as most PCI cards must share the interrupt request line with other peripherals. In particular, on my development motherboard both the PCI-E and the PCI slot share the interrupt with other core peripherals and I couldn't test stuff if I didn't enable sharing.

Unfortunately, the *rawrabbit* interrupt handler can't know if the interrupt source is its own board or another peripherals, so all it can do is saying it handled to the interrupt (by returning `IRQ_HANDLED`) and disable it. If you are running other peripherals on the same interrupt line, you'll need to acknowledge the interrupt pretty often, to avoid a system lock or data loss in your storage or network device.

2.3 Bugs and misfeatures

This version of *rawrabbit* creates a single device and can act on a single PCI peripheral at a time. This limitation will be removed in later versions, as time permits.

The *read* and *write* implementations don't enforce general data-size constraints: reading or writing 1, 2, 4, 8 bytes at a time forces 8, 16, 32, 64 bit accesses respectively, while bigger transfers use unpredictable access patterns to I/O memory, as the driver uses *copy_from_user* and *copy_to_user*.

The interrupt line is always requested and handled (by disabling it). This means that if the line is shared with other devices, you can't avoid it being disabled thus breaking the other devices. A specific ioctl to request/release the handler is needed.

The driver assumes to work with PCI-E so odd BAR areas are not supported. This limitation may be lifted in future versions if needed.

Important: please note that there may be bugs related to cache memory. When using DMA you may encounter incorrect data due to missing flush or invalidate instructions. If the problem is real please report the bug to me, with as much information as possible about the inconsistency, and I'll do my best to find the proper solution. One solution might be adding two *ioctl* commands: one to flush the buffer after it has been written and one to invalidate it before reading; however better solutions, with no API changes, may be viable. Or the problem may just not appear as things are already correct, I can't tell for sure.

2.4 The DMA buffer

At module load time, a 1MB buffer is allocated. The actual size can be changed by means of a module parameter, but it currently can't be bigger than 4MB.

The buffer is allocated with *vmalloc*, so it is contiguous in virtual space but not in physical space. User space can read and write the buffer like it was BAR 12 (0xc) of the device, using contiguous offsets from 0 up to the buffer size.

In order to DMA data to/from the buffer, the peripheral device must be told the physical address to use. Since allocation is page-grained, you need a different physical address for each 4kB page of data. The driver can thus return the list of *page frame numbers* that make up the *vmalloc* buffer. A PFN is a 32-bit number that identifies the position of the page in physical memory. With 4kB pages, you can shift by 12 bits to have the physical address, and a 32-bit PFN can span up to 44 bits of physical address space.

The details about how PFNs are returned to user space are described later where the *ioctl* commands are discussed. A working example is in the *rrcmd* user space tool.

Unlikely what happens with I/O memory, reading and writing the DMA buffer uses the *copy_*_user* functions for all accesses, so the pattern of actual access to memory can't be controlled, but this is not a problem for RAM (as opposed to registers).

2.5 System calls implemented

The following system calls are implemented in *rawrabbit*:

open
close These system calls are used to keep a refcount of device use. If the device has been opened more than once, it will refuse to change the active device, to prevent possible confusion in another process using *rawrabbit* at the same time. Please note that after *fork* the device is still opened twice but the driver can't know about it, so in this case changing the active device is allowed, but it can be confusing nonetheless.

llseek The *seek* family of system calls is implemented using the default kernel implementation. A process may seek the device to access specific registers in specific BAR areas, or the DMA buffer. The offset being used selects the BAR and the offset within the BAR at the same time. Each BAR is limited to an extension of 512MB: so BAR0 starts at 0, BAR 2 starts at 0x2000.0000 and BAR 4 starts at offset 0x4000.0000; if you prefer symbolic names, `RR_BAR_0`, `RR_BAR_2` and `RR_BAR_4` are defined in `rawrabbit.h`. The DMA buffer is accessed like it was BAR 12 (`RR_BAR_BUF`), so `0xc` or `c` can be used in *rrcmd* (see Section 2.7.1 [rrcmd], page 5).

read
write By reading and writing the device, a process can access on-board I/O space. The file position (set through *llseek* or by sequential access of file data) is used to specify both the BAR and the offset within the BAR as described above. Access to an inexistent BAR returns `EINVAL`, access outside the BAR size returns `EIO`. If the hardware device offers I/O ports (instead of I/O memory), the system calls are not supported and you must use *ioctl* – *read* and *write* will return `EINVAL` like the BAR was not existent.

As a special case, reading past the DMA buffer size returns 0 (EOF), and writing retunrs `ENOSPC`, since the DMA buffer is a memory region and a file-like interface is best suited for command-line tools like `dd`.

mmap **Warning:** mmap is not yet implemented in this version.

The *mmap* system call allows direct user-space access to the I/O memory. The device offset has the same meaning as for *read*, but accesses to undefined pages cause a `SIGBUS` to be sent. If the device offers I/O ports (instead of I/O memory), the *mmap* method can't be used on such BAR areas.

ioctl A number of *ioctl* commands are supported, they are listed in the next section. Note that the commamnds to read and write can act both on memory and "I/O ports" areas.

2.6 Ioctl commands

The following *ioctl* commands are currently implemented. The type of the third argument is shown in parentheses after each command:

`RR_DEVSEL (struct rr_devsel *)`
The command copies device selection information to kernel space. If the device has been opened more than once the command fails with `EBUSY`; otherwise the pci driver is unregistered and re-registered with a new `pci_id` item. If no device matches the new selection `ENODEV` is returned after a timeout of 100ms.

`RR_DEVGET (struct rr_devsel *)`
The command returns to user space device information: vendor/device, subvendor/subdevice and bus/devfn. If no device is currently managed by the driver, `ENODEV` is returned.

`RR_READ (struct rr_iocmd *)`
`RR_WRITE (struct rr_iocmd *)`
The commands can read or write one register from an even BAR area (BAR 0, 2, 4) of within the DMA buffer (BAR 12, 0xc). The `address` field of the structure specifies both the BAR and the offset (see `rawrabbit.h` or the description of *llseek* above for the details). Access outside the size of the area returns `ENOMEDIUM`. The `datasize` field of `rr_iocmd` can be 1, 2, 4 or 8 and is a byte count. The other fields, `data8` through `data64` are used to host the register value; these fields are collapsed together in an unnamed union (see the *gcc* documentation about unnamed unions), so the same code works with little-endian and big-endian systems.

`RR_IRQWAIT (no third argument)`
The command waits for an interrupt to happen on the device. If an interrupt did already happen, `EAGAIN` is returned, otherwise an interrupt is waited for and 0 is returned. After the interrupt fired, the interrupt line is disabled by the kernel handler. Please note that this may be a serious problem if the line is shared with other peripherals, like your hard drive o ethernet card.

`RR_IRQENA (no third argument)`
The command re-enables the interrupt. The user is assumed to have acknowledged the interrupt in the board itself, or another interrupt will immediately fire. If the interrupt did not happen, `EAGAIN` is returned, otherwise the command returns the number of nanoseconds that elapsed since the interrupt occurred. If more than one second elapsed, the command returns 1000000000 (one billion), to avoid overflowing the signed integer return value of *ioctl*.

`RR_GETDMASIZE (no third argument)`
The command simply returns the size, in bytes, of the DMA buffer, Currently such size can only be changed at module load time and is fixed for the lifetime of the module.

`RR_GETPLIST (array of 1024 32-bit values)`
The command returns the PFNs for the current DMA buffer. The initial part of the page passed as third argument is filled with 32-bit values. The array must be a complete 1024-entry array, even if only part of it is used. Each value written represents a *page frame number* that can be shifted by 12 bits to obtain the physical

address for the associated page. The *rawrabbit* module can only work with 4kB pages, and a compile-time check is built into the code to prevent compilation with a different page size; at least not before a serious audit of the code.

2.7 User space demo programs

The subdirectory `user/` of this package includes the user-space sample tools. The helper for *rawrabbit* (*rr*) is called *rrcmd*.

2.7.1 rrcmd

The *rrcmd* program can do raw I/O and change the active binding of the device.

Every command line can change the binding and issue a command. Since binding is persistent, you can issue commands without specifying a new binding. The initial binding is defined by module parameters, or by default as a GN4124 device.

To specify a new binding, the syntax is "*vendor*`:`*device*`/`*subvendor*`:`*subdevice*`@`*bus*`:`*devfn*" where the first pair is mandatory and the following ones are optional.

The following is an example session with *rrcmd*, from the compilation directory, note that in this case I'm using the GN4124 device and an ethernet port without active driver.

```
tornado% sudo insmod kernel/rawrabbit.ko
tornado% ./user/rrcmd info
/dev/rawrabbit: bound to 1a39:0004/1a39:0004@0001:0000
tornado% ./user/rrcmd 10b7:9055
tornado% ./user/rrcmd info
/dev/rawrabbit: bound to 10b7:9055/10b7:9055@0004:0000
tornado% ./user/rrcmd 1a39:0004 info
/dev/rawrabbit: bound to 1a39:0004/1a39:0004@0001:0000
tornado% ./user/rrcmd 10b7:9055@01:0
./user/rrcmd: /dev/rawrabbit: ioctl(DEVSEL): No such device
tornado% ./user/rrcmd info
/dev/rawrabbit: not bound
```

The "no such device" error above depends on the chosen bus:devfn parameter. Please note that trying to bound to a device already driven by a kernel driver returns `ENODEV` in the same way, as the probe function of the PCI driver registered by *rawrabbit* will not be called.

To read and write data with *rrcmd* you can use the `r` and `w` commands. The syntax of the commands is as follows:

```
r[<sz>] <bar>:<addr>
w[<sz>] <bar>:<addr> <val>
<sz> = 1, 2, 4, 8 (default = 4)
<bar> = 0, 2, 4
```

Actually, since an interactive user often reads and writes the same register, the `r` and `w` commands are the same, and a read or write is selected according to the number of arguments. You can think of `r` as "register" and `w` as "word" if you prefer.

In this example two Gennum leds are turned off, and the value is read back. Address 0xa08 in BAR 4 is the "output drive enable" register for the GPIO signals from the GN4124 chip, and enabling the drive without any other change from default settings is enough to turn the leds off.

```
tornado% ./user/rrcmd r 4:a08
0x00000000
tornado% ./user/rrcmd r 4:a08 0x3000
tornado% ./user/rrcmd r 4:a08
0x00003000
```

Note, in the example above, that "`r`" is used for writing as well as reading. If you forget the `r` or `w` command name, however, the program will understand the argument as a *vendor*:*device* pair, and will unbind the driver. This can be construed as a design bug and you can blame me at will.

Reading data with a different-from-default size returns the right number of hex digits, to make clear what data size that has been read:

```
tornado% ./user/rrcmd r1 4:a08
0x00
tornado% ./user/rrcmd r2 4:a08
0x3000
tornado% ./user/rrcmd r4 4:a08
0x00003000
tornado% ./user/rrcmd r8 4:a08
0x0000000000003000
```

Interrupt management with *rrcmd* can be performed using two commands: `irqwait` and `irqena`. The former is used to wait for an interrupt to happen; the latter re-enables the interrupt in the controller. You should probably acknowledge the interrupt in the device between these two operations. The `irqwait` command returns `EAGAIN` if the interrupt has already happened; the `irqena` command returns `EAGAIN` if the interrupt has not happened yet.

For example, this script waits for an interrupt in a BT878 frame grabber and acknowledges it for 100 times:

```
# select device and enable vsync interrupt (bit 1, value 0x2)
./user/rrcmd 109e:036e w 0:104 2
# now wait for irq, acknowledging bit 1 for vsync
for n in $(seq 1 100); do
   ./user/rrcmd irqwait
   ./user/rrcmd w 0:100 2
   ./user/rrcmd irqena
done
# finally, disable the interrupt in the device, ack and enable
./user/rrcmd w 0:104 0
./user/rrcmd w 0:100 2
./user/rrcmd irqena
```

The other commands are *getdmasize* and *getplist*, that work as follows:

```
tornado% ./user/rrcmd getdmasize
dmasize: 1048576 (0x100000 -- 1 MB)
tornado% ./user/rrcmd getplist | head
buf 0x00000000: pfn 0x00029a3c, addr 0x000029a3c000
buf 0x00001000: pfn 0x0002dbb1, addr 0x00002dbb1000
buf 0x00002000: pfn 0x00029a34, addr 0x000029a34000
buf 0x00003000: pfn 0x00029839, addr 0x000029839000
buf 0x00004000: pfn 0x00029838, addr 0x000029838000
buf 0x00005000: pfn 0x000298ed, addr 0x0000298ed000
buf 0x00006000: pfn 0x000298ec, addr 0x0000298ec000
buf 0x00007000: pfn 0x00029843, addr 0x000029843000
buf 0x00008000: pfn 0x00029842, addr 0x000029842000
buf 0x00009000: pfn 0x0002dbab, addr 0x00002dbab000
```

2.8 User space benchmarks

The package includes a few trivial programs used to benchmark performance of the various I/O primitives.

2.8.1 bench/ioctl

The program tests how many ioctl output operations can be performed per second. It issues a number of register writes assuming the driver is currently accessing the Gennum evaluation board.

The data written makes the 4 GPIO leds blink with different duty cycles, so you should see them lit at different light levels.

On my system, the program reports more than 3 million operations per second:

```
tornado% ./bench/ioctl 1000000
1000000 ioctls in 303611 usecs
3293688 ioctls per second
tornado% ./bench/ioctl 10000000
10000000 ioctls in 3068384 usecs
3259044 ioctls per second
```

2.8.2 bench/irq878

Warning: this program is missing The program does the same kind of operation as the script shown earlier: it handles BT878 interrupts in user space, and prints the delays from actual interrupt to end-of-acknowledge. While the script shown earlier report times in the order of 10ms, since several processes are executed between the interrupt and the final `irqena`, this shows the system call overhead which is just a few microseconds:

```
tornado% ./bench/irq878 100
got 100 interrupts, average delay 6389ns
```

2.8.3 Benchmarking read and write

No specific program is provided to check access to the DMA buffer, as *dd* is enough to verify read and write speed. A script like the following will work:

```
IF="if=/dev/rawrabbit"
OF="of=/dev/rawrabbit"
# test dmabuf read
for BS in 1 2 4 8 16 32 64 128 256 512 1024 2048 4096; do
   dd bs=$BS skip=$(expr $(printf %i 0xc0000000) / $BS) $IF of=/dev/null \
2>&1 | grep MB/s
done
# test dmabuf write
for BS in 1 2 4 8 16 32 64 128 256 512 1024 2048 4096; do
   dd bs=$BS seek=$(expr $(printf %i 0xc0000000) / $BS) $OF if=/dev/null \
2>&1 | grep MB/s
done
```

To benchmark access to I/O memory, the *rdwr* utility is offered. It repeatedly accesses the GPIO register (bar 4, offset 0xa08 of the GN4124 board) as a 32bit register and measures the time it takes:

```
tornado% ./bench/rdwr 1000000
1000000 writes in 361487 usecs
2766351 writes per second
1000000 reads in 1041681 usecs
959986 reads per second
```

It's interesting to note that reads are slower than writes, but mostly that writes are smaller than writing *ioctls* (compare with `bench/ioctl`). The difference is probably due to the need to

lseek between one *read* or *write* and the next, so for an *ioctl*-based I/O operation you need one system call, while to achieve the same using *read* or *write* you need two system calls.

Table of Contents

Introduction 1

1 Driver for GN4124 1

2 Raw PCI I/O 1

2.1 General features of rawrabbit 1
2.2 Interrupt management 2
2.3 Bugs and misfeatures 2
2.4 The DMA buffer 2
2.5 System calls implemented 3
2.6 Ioctl commands 4
2.7 User space demo programs 5
2.7.1 rrcmd 5
2.8 User space benchmarks 7
2.8.1 bench/ioctl 7
2.8.2 bench/irq878 7
2.8.3 Benchmarking read and write 7