Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

Linux, Open Source, and System Bring-up Tools

How to make bring-up hurt less

Tim Hockin
Google, Inc.
thockin@google.com

Abstract

System bring-up is a complex and difficult task. Modern hardware is incredibly complex and it's not getting any simpler. Bringup engineers spend far too much time hunting through specs and register dumps, trying to find the source of their problems. There are very few tools available to make this easier.

This paper introduces three open source tools that can help ease the pain, and hopefully shorten bring-up cycles for new platforms. SGABIOS is a legacy option ROM which implements a simple serial console interface for BIOS. Iotools is a suite of low-level utilities which enables rapid prototyping of register settings. Prettyprint is a powerful library and toolset which allows users to easily examine and manipulate device settings.

Introduction

Sometimes you get lucky on a bring-up, and things just work the way they are supposed to. More often than not, though, something goes wrong. Unfortunately, it's usually many "somethings" that go wrong. When things do go wrong, someone has to figure out what happened and how to fix it.

Platforms today are vastly more complicated than they were just a few years back. Almost nothing works when the system powers on. It all needs to be configured. When the inevitable "something" goes wrong, determining the cause can be an overwhelming task.

Of course, there are no magic bullets, but there are tools that can help to make solving some of these problems easier.

1 Terminology

Before diving in, it's important that we are all speaking the same language:

Platform: Sometimes used as a synonym for motherboard, a platform is really the combination of components that make up a computer system. This includes the CPU or CPU family, the memory controller, the IO controller, the DRAM, the IO devices, and usually the system firmware.

Bring-up: The process of evolving a platform from an expensive *objet d'art* into a fully operational computer system. This process usually involves debugging and/or working around the hardware, configuring the system in the BIOS, and hacking the drivers and kernel into shape. It often includes superstitious rituals, cynical prayers, and lots of cussing.

BIOS: Basic Input Output System. The BIOS is the software that executes when a PC powers on, and is primarily responsible for configuring the hardware.

Device: A piece of hardware that is logically self-contained. While a typical southbridge is a single chip, it is usually viewed as a collection of devices, such as disk controllers, network interfaces, and bridges.

Chipset: A hardware chip or chips that provide the bulk of the IO on a platform. Chipsets are typically tested and sold as a single unit. These generally include a *north bridge* which contains one or more memory controllers as well as high-speed IO bridges, and a *south bridge* which contains lower-speed devices such as storage and legacy bus interfaces.

Register: An addressable set of bits exposed by a device. Most devices contain many registers. Registers generally hold control and status bits for the device, and can be mapped into a multitude of address spaces such as PCI config space, memory, or IO space.

2 Serial Console for the Unwashed Masses

An obvious place to start is to get the BIOS output as it boots. Just about anyone who has ever booted up a PC has seen the BIOS output on the screen. This is, however, not very useful. Most servers do not have a monitor plugged in to them at all times. VGA-capable chips, while not particularly high-tech, are not free to buy or run. Why require one on every server?

It is a sad fact that many platforms available today *still* do not have serial console support. Those that do offer it usually offer it as an up-sell on the BIOS, and the implementation quality is often questionable.

Some implementations provide side-band interfaces, which only get used to print certain information. This is not particularly useful to anyone, and is fortunately not seen much any more. Some implementations do what is called *screen scraping* which depends on a real VGA device with real VGA memory to store the screen contents. They periodically scan the VGA memory and send updates on the serial port. Some implementations support text output but completely break down in the face of "advanced" features like cursor movement or color.

2.1 Solving It Once and for All

In order to provide a consistent feature set, one Google engineer chose to solve this once and (hopefully) for all. Thus was born `SGABIOS`—the *Serial Graphics Adapter*. `SGABIOS` is a stand-alone option ROM which can be loaded on a platform to provide serial console support. It provides a robust set of VGA-compatible features which allow most BIOS output to be converted to serial-safe output. It supports basic cursor movement, color, text input, and large serial consoles.

The easiest way to use `SGABIOS` is to make your BIOS load it as an option ROM. You can try to convince your board vendor to include it as an option ROM in the BIOS build, or you can use tools (usually provided by the BIOS vendor) to load an option ROM into a BIOS image. If this is not an option for you, all is not lost. There are commercially available add-in debug cards which have option-ROM sockets. In a pinch, many network and other cards have programmable ROMs which can be made to load an arbitrary option ROM.

When started, `SGABIOS` attempts to detect if there is a terminal attached. If detected, `SGABIOS` will adapt its internal structures to the detected terminal size. `SGABIOS` then traps `INT 10h`, the legacy "print something" BIOS function, and `INT 16h`, the legacy "read keyboard" function. The final result is that any well-behaved BIOS, option ROM, or legacy OS will now be using the serial port transparently. However, there are some badly behaved programs which attempt to write to VGA memory directly. `SGABIOS` can not fix those applications. Fortunately, this does not seem to be a very big problem.

We have successfully run `SGABIOS` with LILO and GRUB, as well as DOS. It works wonderfully for the uses we have found, though it does have its limitations. Some applications, such as LILO, query `INT 10h` for previously displayed data. Because there is no VGA memory backing it, `SGABIOS` only stores a small amount of the most recently printed output. This has been good enough to handle the applications we have found to do this, but it does have the potential to fail. As with so many things, it is a tradeoff of memory size vs. functionality.

You can find `SGABIOS` at `http://sgabios.googlecode.com`.

3 Simple Access to Registers

A recurring situation in my office is that you can boot, but something is not right. You might have some ideas on what it could be, but you need to run some additional tests. You need to modify some registers.

You could have the board vendor build some test BIOSes with the various settings. That's not going to be an effective, scalable, or timely solution.

You could build a custom kernel which programs the desired changes; at least you control that part. It's still a pretty heavyweight answer, and the hardware test team folks are not really kernel hackers. This approach is better than the last one, but not good.

One might ask "Hold on, doesn't the kernel expose some APIs that let me fiddle with registers?" Why yes, it does. Now you only have to write some simple programs to do these tests. But again, the test team is not really C programmers. There must be something simpler.

3.1 Introducing Iotools

A simple, scriptable interface to device registers allows anyone who can do basic programming to deal with this problem. Almost anyone is now able to trivially read and write registers, thereby enabling a whole new debugging army.

This is the goal of `iotools`. The `iotools` package provides a suite of simple command-line tools which enable various forms of register accesses. They are mostly thin wrappers around Linux kernel interfaces such as sysfs and device nodes. `Iotools` also includes a number of simple logical operation tools, which make manipulating register data easier.

The `iotools` "suite" is actually a single binary, a la `busybox`. This allows for simple distribution and installation on target systems. The `iotools` binary is less than 30 kilobytes in size when built with shared libraries. Building it as a static binary obviously increases the size, depending on the libc it is linked against. This should make `iotools` suitable for use in most size-sensitive environments, such as flash or initramfs.

A note of caution is warranted. Writing to registers on a running system can crash the system. You should always understand exactly what you are changing, and whether there might be a kernel driver managing those same registers. Sometimes it is enough to simply unload a driver before making your changes. Other times you just have to go for it.

3.2 What's in Iotools?

At the time of writing, the `iotools` suite includes tools to access the following register spaces:

- **PCI**: Read and write registers in PCI config space. This includes both traditional config space (256 bytes per device) and extended config space (4 Kbytes per device) for those devices which support it. Access is provided by `sysfs` or `procfs` and is supported as 8-bit, 16-bit, and 32-bit operations.

- **IO**: Read and write registers in x86 IO ports. This covers the 64-Kbyte space only. Access is provided by `IN` and `OUT` instructions and is supported as 8-bit, 16-bit, and 32-bit operations.

- **MMIO**: Read and write memory-mapped registers or physical memory. This provides access to the entire 64-bit physical memory space via `/dev/mem`. It supports 8-bit, 16-bit, and 32-bit operations.

- **MSR**: Read and write x86 model-specific registers on any CPU. This provides access to the full 32-bit MSR space via `/dev/cpu/*/msr`. It supports only 64-bit operations (all MSRs are 64 bits).

- **TSC**: Read the CPU timestamp counter on the current CPU. This is provided by the `RDTSC` instruction and is always a 64-bit operation.

- **CPUID**: Read data from the `CPUID` instruction on any CPU. This provides access to the full 32-bit CPUID space via `/dev/cpu/*/cpuid`.

- **SMBus**: Read and write registers on `SMBus` devices. This is provided by the `/dev/i2c-*` drivers and supports 8-bit, 16-bit, and block operations.

- **CMOS**: Read and write legacy CMOS memory. Most PCs have around 100 bytes of non-volatile memory that is accessed via the real-time clock. Access is provided by the `/dev/nvram` driver, and only supports 8-bit operations. This should be used with caution. CMOS memory is often used by the system BIOS, and changing it can have unintended side effects.

In addition to the register access tools, `iotools` also includes several tools to perform logical operations on numbers. These tools are important because they support 64-bit operations and treat all numbers as unsigned, which can be a problem in some shell environments.

- **AND**: Produce the logical AND of all arguments.

- **OR**: Produce the logical inclusive OR of all arguments.

- **XOR**: Produce the logical exclusive OR of all arguments.

- **NOT**: Produce the bitwise NOT of the argument.

- **SHL**: Shift the argument left by a specified number of bits, zero-filling at the right.

- **SHR**: Shift the argument right by a specified number of bits, zero-filling at the left (no sign extension).

3.3 A Simple Example

Suppose you need to test the behavior of enabling SERR reporting on your platform. This is controlled by bit 8 of the 16-bit register at offset 4 of each PCI device. You could whip up a quick script:

```
#!/bin/bash

function set_serr {
  # SERR is bit8 (0x100) of
  #     16-bit register 0x4
  OLD=$(pci_read16 $1 $2 $3 0x4)
  NEW=$(or $OLD 0x100)
  pci_write32 $1 $2 $3 4 $NEW
}

# hardcoded list of PCI addresses
set_serr 0 0 0
set_serr 0 0 1
set_serr 0 0 2
```

You can do better than this, though. You can trivially make this script loop for each PCI device:

```
#!/bin/bash

function set_serr {
  # SERR is bit8 (0x100) of
  #     16-bit register 0x4
  OLD=$(pci_read16 $1 $2 $3 0x4)
  NEW=$(or $OLD 0x100)
  pci_write32 $1 $2 $3 4 $NEW
}

# for each bus, dev, func
for B in $(seq 0 255); do
  for D in $(seq 0 31); do
    for F in $(seq 0 7); do
      pci_read32 $B $D $F 0 \
          >/dev/null 2>&1
      if [ $? != 0 ]; then
        # does not exist
        continue;
      fi
      set_serr $B $D $F
    done
  done
done
```

This version takes a bit longer to run, but works regardless of the devices in the system. You can shorten the run time significantly by putting a sane upper bound on the number of buses. Few systems have more than 20 or 30 buses, even in this era of point-to-point PCI Express buses.

This is the sort of tool that someone with very basic shell scripting skills can produce in just a few minutes with `iotools`.

You can find `iotools` at `http://iotools.googlecode.com`.

4 Making it Simpler

The previous section shows just one example of the sorts of problems that arise during bring-up. Frankly, it wasn't a particularly complicated problem, and the solution is bordering on real programming. Worse than that, it requires that the person doing the work remember several "magic" numbers. Which register is this bit in? How wide is that register? Which bit is it? Taken further, the problem quickly becomes very difficult.

Suppose you want to examine or configure something more complicated, like PCI Express advanced error reporting (AER). AER is a *capability* in PCI terminology. That means that some devices will support it and some will not. The only way to find out is to ask each device. Further, each device might put the AER registers at a different offset in their PCI register set. As if that is not enough, some devices have different AER register layouts, depending on what kind of device they are and which version of the the specification they support.

Doing this in an `iotools` script is certainly possible; it just isn't so simple anymore. Google needed something that internalizes and hides even more of the details. This gave rise to `prettyprint`.

4.1 An Unfortunate Name

The original goal of `prettyprint` was this: to dump the state of all the registers in the system in a `diff`-friendly format. This would allow us to use one of our favorite debug tools, which we call *"Did you try rolling back the BIOS?"* Boot with BIOS A, `prettyprint` the system. Boot with BIOS B, `prettyprint` the system. Then `diff` the results.

Like the previous examples, there are other ways of doing this. They all resulted in a screenful of numbers, followed by a few hours of digging through datasheets to find what each bit of each differing register means. The only thing worse than going through this process and finding that the difference is undocumented is going through this process multiple times.

Instead, `prettyprint` attaches a datatype to field values, allowing it to produce output which is not only `diff`-friendly, but which is also human-friendly.

4.2 Fundamentals of Prettyprint

`Prettyprint` has two fundamental constructs: *registers* and *fields*. In keeping with the common vernacular, a register is a single addressable set of bits. Registers have a defined, fixed width, but they have no intrinsic meaning.

Fields, on the other hand, are of arbitrary width and are the only entity with meaning. Fields can be defined as a set of register bits (*regbits*), constant bits, or even as procedures. Every field has a datatype, and the result of reading a field is a value that can be evaluated against that datatype to produce a human-readable string.

4.3 The Power of Fields

Let's look at a the simple example from Section 3.3. For each PCI device there is a 16-bit register at offset 4 called `%command` (the `%` is a convention to indicate a name is register). For each PCI device there is also a field called `serr`. This field is exactly 1 bit wide, and is composed of bit 8 of `%command`. When accessing this field, one can interpret its value as a boolean, where a value of 1 = *"yes"* and a value of 0 = *"no"*.

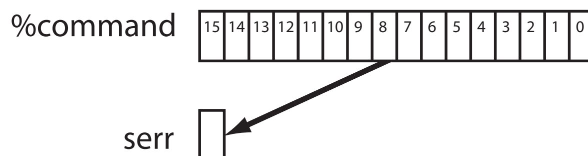

Figure 1: A simple field

Now, when you dump the state of a device, you can see a line item that says `serr: yes`.

This is vastly more useful than a hexadecimal number about which I have to remember that bit 8 being set means `SERR` is enabled. Even better, since I now have a system that understands `serr` directly, I can write to it just as easily as I can read from it.

4.4 Binding Fields to Devices

The previous example glossed over the details of "for each PCI device." This is a key aspect of `prettyprint`'s power. Registers are defined in an abstract way, divorced of exactly which device or access method they employ. They simply have *addresses*. When it comes time to use these registers, `prettyprint` passes control to the drivers which enable each class of device. A *binding* is used to map which abstract registers belong to which driver.

When starting up, `prettyprint` can find hardware devices in one of two ways. Firstly, you can tell it where a device is found. This is the only option for some devices, especially legacy devices. For example, to tell `prettyprint` about the serial port, you would have to tell it something to the effect of, *"There exists a serial port in IO space, at address 0x3f8."* In so doing, you have given `prettyprint` enough information to bind the serial port registers and fields to a driver and address.

Better still, you can let `prettyprint` discover some devices. Many modern devices can be discovered either through the hardware itself, such as PCI, or through simple interfaces, such as ACPI. In this case, the driver has a discovery routine which will find devices and bind them as it finds them. This is how we are able to define things like `serr` as something that exists "for each PCI device."

4.5 About the Implementation

`Prettyprint` is written in C++. I can hear the cries of frustration already. Why C++? Because I thought that the problem decomposed nicely into an object-oriented model, and because I wanted to improve my C++.

`Prettyprint` has been designed from the start as a library to be used as a backend by various applications. From state dumping utilities to interactive shells to `FUSE` filesystems, anything is possible.

4.6 Defining Registers And Fields

So how does one go about defining a device? One of the things that the choice of C++ brought to the project was a way to manipulate the language syntax. The end goal is to have an actual interpreted language which is used to define devices. Until then, we have a set of C++ classes, functions, and templates which define a pseudo-language.

This pseudo language is intended to make the definition of registers and fields as simple as possible. Let's look at the SERR example:

```
REG16("%command", 0x04);
FIELD("serr", "yesno_t",
    BITS("%command", 8));
```

That's pretty straightforward. We define %command as a 16-bit register at address 0x04. We define serr as a field with datatype yesno_t, composed of bit 8 from %command.

Frequently, a field maps directly to a register. To simplify this, prettyprint understands *regfields*. For example, the PCI "intpin" field is the only consumer of the %intpin register.

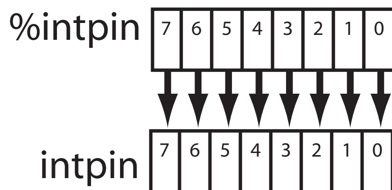

Figure 2: A regfield

We can express that as:

```
REG8("%intpin", 0x3d);
FIELD("intpin", "int_t",
    BITS("%intpin", 7, 0));
```

Or we can take the equivalent regfield shortcut:

```
REGFIELD8("intpin", 0x3d, "int_t");
```

Let's consider a more complicated example. In a PCI-PCI bridge, there are several registers which control the address ranges which are decoded by the bridge. They are implemented as two different registers, which combine to form a logical 64-bit address. The low 20 bits of both the base and limit register are fixed to 0 and 1, respectively.

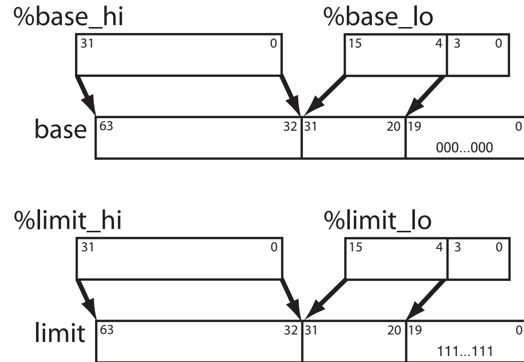

Figure 3: Complex fields

In prettyprint, this is expressed as:

```
REG16("%base_lo", 0x24);
REG32("%base_hi", 0x28);
REG16("%limit_lo", 0x26);
REG32("%limit_hi", 0x2c);

FIELD("base", "addr64_t",
      BITS("%base_hi", 31, 0) +
      BITS("%base_lo", 15, 4) +
      BITS("%0", 19, 0));
FIELD("limit", "addr64_t",
      BITS("%limit_hi", 31, 0) +
      BITS("%limit_lo", 15, 4) +
      BITS("%1", 19, 0));
```

Notice the use of %0 and %1 as registers. These are the *magic registers*. When read, %0 always returns all logic 0 bits. Likewise, %1 always returns all logic 1 bits. Also notice that the bits in a field are defined from most significant to least significant. A field can be arbitrarily long, and can be composed of any number of regbits.

4.7 Scopes and Paths

The examples so far have been relatively small. In reality the %command register has a number of fields that derive from it. All told, there are thousands of fields in

each PCI device. prettyprint provides *scopes* as a mechanism for grouping related things together.

Think of scopes like directories in a filesystem. Each scope has a name and a set of contents. A scope can contain registers, fields, or other scopes. Like the filesystem metaphor, prettyprint has paths. There is a conceptual *root* of the path tree, and each register, field, and scope can be named by a unique path. Also like a UNIX directory tree, path elements are seperated by a forward slash (/), and two dots (..) means the parent scope.

The %command register from our previous examples actually looks something like this:

```
REG16("%command", 0x04);
OPEN_SCOPE("command");
  FIELD("io", "yesno_t",
        BITS("../%command", 0));
  FIELD("mem", "yesno_t",
        BITS("../%command", 1));
  FIELD("bm", "yesno_t",
        BITS("../%command", 2));
  FIELD("special", "yesno_t",
        BITS("../%command", 3));
  FIELD("mwinv", "yesno_t",
        BITS("../%command", 4));
  FIELD("vgasnoop", "yesno_t",
        BITS("../%command", 5));
  FIELD("perr", "yesno_t",
        BITS("../%command", 6));
  FIELD("step", "yesno_t",
        BITS("../%command", 7));
  FIELD("serr", "yesno_t",
        BITS("../%command", 8));
  FIELD("fbb", "yesno_t",
        BITS("../%command", 9));
  FIELD("intr", "yesno_t",
        BITS("../%command", 10));
CLOSE_SCOPE();
```

4.8 Datatypes

Each field can be evaluated against its datatype. Prettyprint defines a number of primitives:

- **int**: a decimal number
- **hex**: a hexadecimal number
- **enum**: an enumerated value
- **bool**: a binary enum
- **bitmask**: a set of name bits

These primitives are used to create several pre-defined datatypes:

- **int_t**: a number
- **hex_t**: a hexadecimal number
- **hex4_t**: a 4-bit hexadecimal number
- **hex8_t**: a 8-bit hexadecimal number
- **hex12_t**: a 12-bit hexadecimal number
- **hex16_t**: a 16-bit hexadecimal number
- **hex20_t**: a 20-bit hexadecimal number
- **hex32_t**: a 32-bit hexadecimal number
- **hex64_t**: a 64-bit hexadecimal number
- **hex128_t**: a 128-bit hexadecimal number
- **addr16_t**: a 16-bit address
- **addr32_t**: a 32-bit address
- **addr64_t**: a 64-bit address
- **yesno_t**: a boolean, 1 = "yes", 0 = "no"
- **truefalse_t**: a boolean, 1 = "true", 0 = "false"
- **onoff_t**: a boolean, 1 = "on", 0 = "off"
- **enabledisable_t**: a boolean, 1 = "enabled", 0 = "disabled"
- **bitmask_t**: a simple bitmask

Without doubt, any reasonably complex device will need to define its own datatypes. Prettyprint allows datatypes to be defined at any level of scope, and to be used in any scope below the definition—similar to C.

- **INT(name, units?)**: define a new int type with optional units.
- **HEX(name, width?, units?)**: define a new hex type with optional width and units.

- **ENUM(name, KV(name, value), ...)**: define a new enum type with the specified named values.
- **BOOL(name, true, false)**: define a new bool type with the specified true and false strings.
- **BITMASK(name, KV(name, value), ...)**: define a new bitmask type with the specified named bits.

Sometimes you want to define a new datatype for exactly one field. Rather than come up with a good name for it, each of the datatype definitions supports an `ANON_` prefix, which removes the name argument and produces an anonymous datatype. For example, the previous PCI `intpin` example used `int_t` as the datatype. In reality, we want an enumerated type. This is the only field that will use this type, so we want to declare it anonymously:

```
REGFIELD8("intpin", 0x3d, ANON_ENUM(
        KV("none", 0),
        KV("inta", 1),
        KV("intb", 2),
        KV("intc", 3),
        KV("intd", 4)));
```

4.9 Advanced Techniques

So far, we've seen how `prettyprint` can be used to define simple registers and fields. Unfortunately, few hardware devices are so simple. Because the `prettyprint` "language" is actually a dialect of C++, there is a lot of power at your fingertips.

Hardware registers are at a premium. Often the hardware will overload the meaning of some bits depending on the state of other bits. `Prettyprint` supports the conditional definition of registers and fields.

Let's look at another example. In a PCI bridge's IO decode window, there is a `width` field. That field determines whether the high half of the `base` field is valid.

```
REG8("%base_lo", 0x1c);
REG16("%base_hi", 0x30);

FIELD("width", ANON_ENUM(
        KV("bits16", 0),
        KV("bits32", 1)),
    BITS("%base_lo", 3, 0));

if (FIELD_EQ("width", "bits16")) {
    FIELD("base", "addr16_t",
            BITS("%base_lo", 7, 4) +
            BITS("%0", 11, 0));
} else {  // bits32
    FIELD("base", "addr32_t",
            BITS("%base_hi", 15, 0) +
            BITS("%base_lo", 7, 4) +
            BITS("%0", 11, 0));
}
```

In this example you see the usage of `FIELD_EQ()`. This performs a read of the `width` field and compares the result against the value specified. Comparisons can be done by string or by number, thanks to function overloading in C++. The above example could have just as easily (though less maintainably) used:

```
FIELD_EQ("width", 0)
```

The actual evaluation of the a comparison is done by the specific datatype, which is the only place that can actually determine what it means to compare values. `Prettyprint` supports the following comparison operations:

- **FIELD_EQ**: the field is equal to the specified comparator.
- **FIELD_NE**: the field is not equal to the comparator.
- **FIELD_LT**: the field is less than the comparator.
- **FIELD_LE**: the field is less than or-equal-to the comparator.
- **FIELD_GT**: the field is greater than the comparator.
- **FIELD_GE**: the field is greater than or-equal-to the comparator.
- **FIELD_BOOL**: the field is boolean TRUE, equivalent to NE 0.

- **FIELD_AND**: the field matches the comparator.

Again, because the prettyprint "language" is really just C++, almost any native construct will work. There are some limitations, though.

To start with, C++ will not allow a switch statement on a non-integer value, so you can not switch on enumerated strings. In the eventual prettyprint language implementation, this will be supported.

Secondly, control statements are evaluated just once, as the tree of registers and fields is being built. Later changes to control bits do not change the tree structure. This is something we want to enable in the prettyprint language, but we do not have support for it yet.

4.10 Discovering Specific Devices

Throughout these examples, we have looked at standard PCI fields and registers. The PCI standard covers only a fraction of the available PCI register space. Almost every PCI device defines its own non-standard register set. What about those extra registers and fields?

In the same way that prettyprint can discover generic devices, such as PCI, it can also discover specific devices. A device definition can register itself for discovery through a specific driver. When the driver's discovery mechanism detects the registered device, as determined by a driver-specific signature, it invokes the specific device code, rather than the generic.

For example, a device definition for an AMD Opteron might register with the PCI driver for the vendor and device pair (0x1022, 0x1100). When the PCI driver finds that vendor and device pair, the Opteron-specific device code would be invoked, rather than the generic PCI device code.

Rather than re-encoding the entire PCI specification, the generic PCI code can be invoked from the Opteron code. This allows device code to extend standard devices with very little effort.

4.11 The Directory Tree

The prettyprint code is broken into four components. The top-level directory contains the core classes and functions that make up the prettyprint library. This includes pp_register, pp_field, pp_datatype, etc.

The drivers subdirectory contains the driver modules. Currently prettyprint only supports Linux, though it would not be hard to add support for other operating systems. At the time of writing, the drivers directory contains drivers for:

- **PCI**: via /sys or /proc
- **MEM**: via /dev/mem
- **IO**: via IN and OUT instructions
- **MSR**: via /dev/cpu/*/msr
- **CPUID**: via /dev/cpu/*/cpuid

The devices subdirectory contains device code, written in the prettyprint "language." When we have a real language parser, this is the code that will be rewritten in the new language. Prettyprint currently has support for:

- **PCI**: most of the fields for generic PCI and PCI Express devices, including many capabilities.
- **CPUID**: very basic CPUID fields

Lastly, the examples subdirectory contains example programs which use the prettyprint library.

4.12 FUSE and Prettyprint

One of the more exciting examples is the pp_fs application. This is a FUSE filesystem which allows direct access to registers and fields.

Using pp_fs, the example of setting SERR on all devices becomes trivial:

```
$ find /pp -wholename \*command\/serr \
   | while read X; do
     echo -n "$X: $(cat $X) -> "
     echo "yes" > $X
     cat $X
done
/pp/pci.0.0.0.2/command/serr: no -> yes
/pp/pci.0.0.1.0/command/serr: no -> yes
/pp/pci.0.0.0.1/command/serr: no -> yes
/pp/pci.0.0.0.0/command/serr: no -> yes
```

4.13 Current Status

The current examples demonstrate the capabilities of `prettyprint`. `pp_discover` has already proven to be a useful tool at Google. But there is still a lot of work to do in many areas.

`Prettyprint` is under active development. A great way to get involved is to encode new hardware devices into the `prettyprint` language. The larger our device repository gets, the more useful it becomes.

You can find `prettyprint` at `http://prettyprint.googlecode.com`.

5 Acknowledgements

Thanks to Nathan Laredo for pushing to make `SGABIOS` a reality.

Thanks to Aaron Durbin for busybox-ifying `iotools` on a whim, and to all the Google platforms folks who have added tools to it.

Thanks to Aaron Durbin, Mike Waychison, Jonathan Mayer, and Lesley Northam for all their help on `prettyprint`.

Thanks to Google for letting us hack on fun systems and release our work back to the world.