Reprinted from the

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Device discovery and power management in embedded systems

David Gibson
OzLabs, IBM Linux Technology Center
`dwg@au1.ibm.com, ols2003@gibson.dropbear.id.au`

Abstract

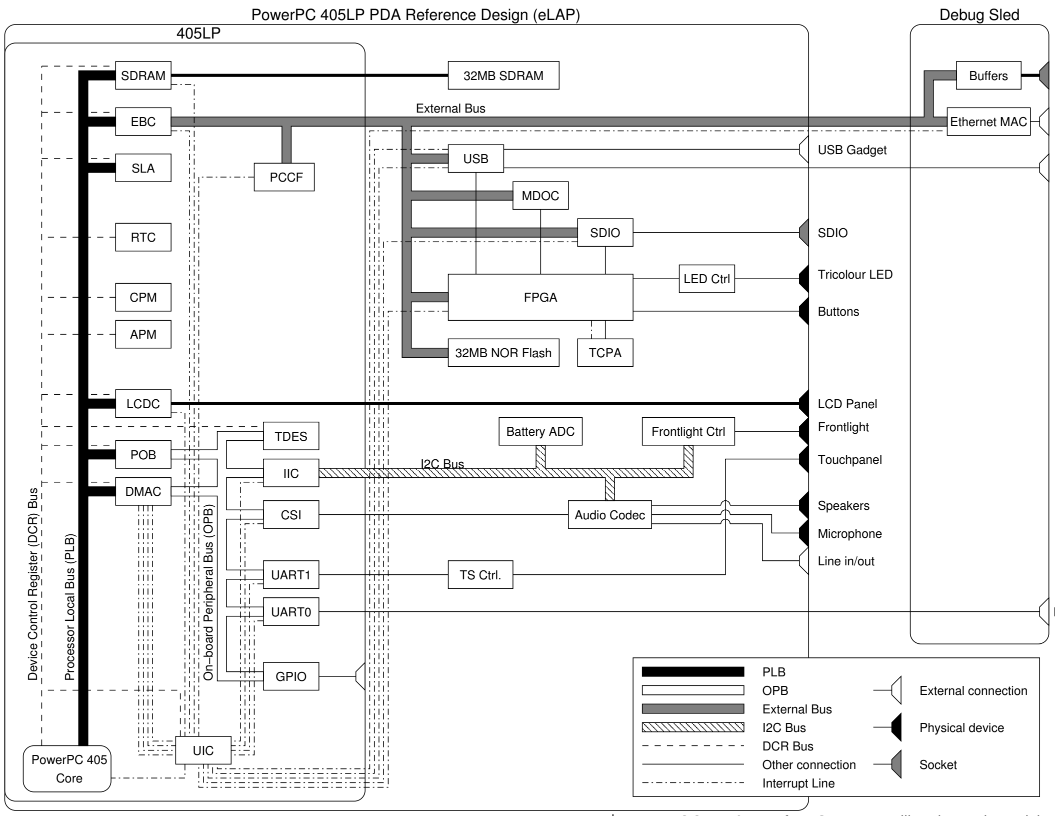
This paper covers issues in device discovery and power management in embedded Linux systems. In particular, we focus on the IBM® PowerPC® 405LP (a "system-on-chip" CPU designed for handheld applications) and IBM's PDA reference design based upon it. Peripherals in embedded systems are often connected in an ad-hoc manner and are not on a bus which can be scanned or probed. Thus the kernel must have knowledge of what devices are present built in at compile time. We examine how the new unified device model provides a clean method for representing this information, while allowing good re-use of code from machine to machine. The 405LP includes a number of novel power management features, in particular the ability to very rapidly change CPU and bus frequencies. We also examine how the device model provides a framework for representing constraints the peripherals and their interconnections place upon allowable frequencies and other information relevant to power management.

1 Introduction: the device discovery problem

Device discovery is the process the kernel and its device drivers use to determine what peripheral devices are present in a machine and how to communicate with them. Generally this means determining what IO addresses, interrupt lines and/or other bus specific addresses and resources are associated with each device.

Usually there are a few peripherals that are present in every machine of a particular type. Then there are optional devices that may or may not be installed in a particular machine. Some of these may be added or removed only from one boot to the next, and some may be hot-pluggable, added or removed while the machine is running.

The peripheral devices in an embedded machine often look very different to those in a conventional desktop or server. Even when a similar peripheral is used, differences in the way it is connected into the system can mean that it must be accessed and initialised quite differently. Many assumptions that are made about devices in a "normal" machine cannot be made in embedded machines, and the hardware and firmware of embedded machines generally provides much less assistance to the kernel for device discovery. All these things require different approaches to device discovery to be used.



APM	Power management unit, implements the CPU sleep/suspend states (NB: this is not related to the APM BIOS in PC laptops)
Audio Codec	Texas Instruments TLV320AIC23 stereo audio codec
Battery ADC	ADS7823 ADC monitoring battery voltage
CPM	Clock and Power Management unit
CSI	Codec Serial Interface (interface to audio codec devices)
DMAC	DMA controller (can DMA to/from devices on both the PLB and OPB)
EBC	External Bus Controller
Ethernet MAC	RTL8109 Ethernet controller
Frontlight Ctrl	DS1050 LCD frontlight controller
GPIO	General Purpose IO interface
IIC	I^2C bus interface (both master and slave capable)
LED Ctrl	BU8770KN tricolour LED driver
LCDC	LCD display controller
MDOC	64M of M-Systems Millennium Plus Disk-on-Chip (NAND flash with specialised controller)
PCCF	PCMCIA and Compact Flash controller
POB	PLB-to-OPB bridge
RTC	Real Time Clock
SDIO	Toshiba TC6380AF Secure Digital and SDIO interface
SDRAM	SDRAM controller
SLA	Speech Label Accelerator (hardware implementation of a particular speech recognition algorithm)
TCPA	Atmel AT97SC3201 TPM
TDES	Triple-DES accelerator
TS Ctrl	Semtech UR7HCT52_S840L touchscreen controller
UARTx	NS16550 compatible UARTs
UIC	Universal Interrupt Controller
USB	Phillips ISP1161 USB interface (includes both host controller and gadget-side interface)

Figure 1: Block diagram of the eLAP

2 The PowerPC 405LP PDA reference design

Most of the issues we discuss in this paper apply to many different embedded machines. However, for simplicity we focus on one example machine, the PowerPC 405LP PDA reference design, also known as the Embedded Linux Application Platform or eLAP. As the name suggests, this is a prototype reference design for a PDA based on the PowerPC 405LP CPU.

The IBM PowerPC 405LP is a CPU from the PowerPC 4xx family. This series of CPUs is designed for "system-on-chip" embedded applications. As the name suggests these processors are implementations of the PowerPC Architecture™, however they have some notable differences from "classic" PPC CPUs (as used in IBM pSeries™ servers and Apple workstations). The 4xx CPUs operate at much lower clock rates (and hence are cooler and cheaper), although they are in the high end by embedded standards. They have a much simpler MMU (just a software loaded TLB) and they have no

FPU. More interestingly, they include a number of peripheral devices built into the CPU die itself (hence the term "system-on-chip").

Different CPUs in the 4xx family are designed for different applications and have different collections of on-chip peripherals. The 405LP is designed for handheld, battery-powered applications. Figure 1 shows a block diagram of the eLAP, including the various built-in peripherals of the 405LP. The chip includes no less than three internal buses: the high-bandwidth Processor Local Bus (PLB) connected directly to the CPU core, the slower On-Board Peripheral bus (OPB), and the special DCR bus. The latter is used to implement Device Control Registers: rather than using normal memory-mapped IO, some of the on-chip devices use these special registers which are accessed using special machine instructions. As shown, the 405LP's peripherals include amongst other things, an LCD controller, a real-time clock, an I^{2}C interface and two serial ports. Other 4xx chips can include devices such as Ethernet controllers, HDLC interfaces, PCI host bridges, IDE and USB controllers. The 405LP also includes a number of novel power management features, which we'll examine in §5.

In addition to the devices within the 405LP, the eLAP includes 32MB of RAM, 32MB of NOR Flash and a number of additional peripherals, also shown in Figure 1. Most of these are connected via a minimal bus driven by the 405LP's on-chip External Bus Controller (EBC) unit. An extra debug and development sled can be attached to the eLAP, again shown in Figure 1. It includes an Ethernet controller, the physical PCMCIA slot driven by the 405LP's PCCF core and the physical connectors for the USB host port and serial port. Of course, as well as the peripherals shown, further devices can be attached via the PCMCIA and SDIO slots and the USB host interface.

3 Current approaches

3.1 Conventional machines

On normal server or workstation machines, device discovery is mostly quite straightforward.[1] Nearly all modern machines are based on PCI, which (like most modern buses) is designed so that devices can be queried and configured in a standard way. This makes it easy for the kernel to scan the PCI bus (or buses), determine what devices are present and their addresses, and pass this information to the appropriate device drivers. USB devices provide similar functionality, as do PCMCIA and ISA/PnP devices.[2]

The few remaining devices (including the PCI host bridge itself) are usually standard—to be found on all machines of this type and often nearly all machines of this architecture. They can be found at well-known addresses, so the drivers for these devices simply hardcode this information. On PCs, non-PnP ISA devices do introduce some problems. In fact they demonstrate a subset of the problems with embedded hardware that we will examine in the next section.

Many non-x86 machines make device discovery even simpler with firmware support. Open Firmware (on IBM and Apple PowerPC machines) and likewise its ancestor OpenPROM (on Sun machines) builds a tree with information about each of the peripherals on the machine. At boot time the kernel queries this information, making a copy of the device tree which can later be used by drivers to find devices. The ACPI BIOS found on recent Intel® machines provides some similar information, although neither the ACPI implementations nor Linux's use of them is very well es-

[1] Although on big servers keeping track of the devices once they're discovered can be another matter.

[2] At least in theory; many ISA/PnP implementations are buggy in practice.

tablished as yet.

3.2 Embedded weirdness

On embedded machines all the assumptions that are made on "normal" machines break down. Embedded machines can and do have arbitrarily peculiar combinations of peripherals connected in a more-or-less ad-hoc manner.

Often, many of an embedded machine's peripherals are connected via an unconventional bus which provides no facilities for systematic scanning or probing of devices. On the 405LP this is true of both the on-chip buses and the main external bus. Devices can appear essentially anywhere within the CPU's physical address space. Some times the address or other behaviour of a device is affected by a custom FPGA or other programmable logic device with its own control registers. Device interrupt lines introduce even more problems, being routed in complex and arbitrary ways that are often controlled or masked by a board-specific FPGA or CPLD. Devices can sometimes have multiple dependencies on other devices: for example on the eLAP, audio is driven by the TI codec, which is controlled and configured via the I^2C bus. However, the actual audio data is delivered to the codec via a serial connection to the on-chip Codec Serial Interface (CSI). The CSI in turn depends on the on-chip DMA controller to supply data from RAM.

Sometimes machines also have a more conventional bus such as PCI or PCMCIA, but it may have to be accessed via a bridge which is not configured in the same way as one would expect on a conventional machine. Worst of all, there are dozens or hundreds of different types of embedded machine, each with its own completely different set of devices and connections.

Under these circumstances, it is tempting to turn to each board's firmware to provide information about which devices are present. Unfortunately the firmware on most embedded machines is very primitive, providing little more than a boot loader. Usually it will provide a few useful pieces of information, such as the amount of RAM on the system or the board revision, but it certainly won't give comprehensive device information. Furthermore, what information the firmware does provide usually can't be used without already knowing something about the machine in question, since embedded firmwares are almost as varied as embedded machines themselves.

Since embedded machines can and do break any assumption one might care to make about how devices are attached, there is no magical way that the kernel can detect what devices are present. So, the only approach is to have the kernel "just know" the device setup for a particular board by building the knowledge into the kernel at compile time.

Although it is impossible to completely avoid hardwired knowledge of boards in the kernel, we do want to keep this information in as clean a way as possible. Specifically, we want to isolate the direct knowledge of board specific details to as small a section of the kernel as possible, and we want to make it easy to add the details of new boards and their peripherals.

In this paper, we generally assume that the kernel must be configured for one particular type of embedded machine, since this is the simplest case. Building a kernel which will support multiple machines is certainly possible, and most of the methods we discuss can still be applied. In this case the kernel need to include device information about all supported boards. Early in boot, the kernel will identify the machine it is running on (by some ad-hoc method), and select which information to use on that basis.

Now, we examine some of the existing methods by which embedded devices are supported in the kernel.

3.3 Hardcoded hacks

The naïve approach to handling embedded devices that aren't on a conventional bus is to treat them all like the "system" devices on a PC or other conventional machine. That is, simply hardcode knowledge of the device into the relevant device driver or into the kernel initialisation code for the machine in question. This approach is currently used for quite a number of embedded devices—unsurprisingly since, as we'll see, a comprehensive better approach has yet to be implemented.

This method has some serious shortcomings. The most obvious problems come with embedded peripherals that are similar to ones also found in conventional machines. For obvious reasons, it is normal in this case to adapt the existing conventional driver for use in the embedded machine.

Sometimes this has been done by adding `#ifdef`s to the driver for the board specific code. For example, this has been done with the `cs89x0` driver for the CrystalLAN CS8900 Ethernet chip. This chip is used on some ISA cards, but is also found on the "Beech" embedded board (another IBM reference board based on the 405LP). Apart from the fact that `#ifdef`s make the driver code ugly, this clearly causes a problem if the embedded machine can also have a normal ISA or PCMCIA version of the device: the kernel can't support both versions of the peripheral on the same machine.

Another method is to copy, then modify the existing driver to make a version specific to a particular embedded machine. The approach was taken for the `arctic_enet` driver for the Ethernet on the eLAP's debug sled. The sled's Ethernet is based on an RTL8019 chip, which is used in a number of ISA cards, as well as several other embedded machines. This method allows multiple versions of the device to be simultaneously supported, but incurs the obvious maintenance problems of having several almost-but-not-quite identical drivers present in the kernel. The situation is aggravated as more embedded machines are supported.

The fundamental problem with this approach is that there are many more types of embedded machine than there are of normal machines. Indeed there is often only one dominant type of conventional machine per architecture (PC for x86, CHRP for PowerPC, Sun server/workstation for SPARC, etc.). With the large number of different types of embedded machine, direct hardcoding quickly becomes messy: there is a lot of duplicated code, and it is inconvenient to add new machines and peripherals.

3.4 The OCP subsystem

Since we can't entirely avoid hardcoded information, the obvious approach to isolating the messiness is to encode information about devices into a data structure which is statically compiled into the kernel, but parsed to provide data to the drivers at runtime. This solution is conceptually similar to using device information from firmware except that the device tree is supplied by the kernel itself, rather than read at boot time.

The "OCP" subsystem (standing for On-Chip Peripheral) is a partial implementation of this approach. It only covers the on-chip devices on PPC 4xx chips, and is quite limited in the sorts of device information it can represent, but it iss still a substantial improvement over hardcoded drivers.

The subsystem has been through several significant rewrites before reaching its present form. The initial implementation in the `linuxppc_2_4_devel` BK tree had a number of serious design and interface problems. It was then rewritten in 2.5 based on the new Linux unified device model, and using the PCI subsystem as a reference. This version is considerably cleaner, but still contains some poorly thought out elements, in particular some things have been copied from PCI which make little sense in their new context. It has now been rewritten again by Benjamin Herrenschmidt in the `linuxppc-2.4` BK tree. This latest rewrite is conceptually similar to the 2.5 version, but considerably simpler and cleaner. This final version still needs to be forward ported to 2.5, re-introducing the integration with the unified device model.

For each CPU with OCP devices, there is a table of definitions like that in Figure 2. This is found in a C file specific to the particular CPU, along with any initialisation or support code for that CPU. The example in Figure 2 doesn't include all the 405LP's on-chip devices, since not all the drivers have been adapted to use the OCP infrastructure yet. The table consists of `ocp_def` structures, shown in Figure 3. At boot time, the OCP system scans the `core_ocp` table to produce a list of OCP devices present, making an `ocp_device` structure (also in Figure 3) for each to keep track of it at runtime.

The `vendor` and `function` fields between them identify the type of device. This mimics the vendor/function pairs used to identify PCI and USB devices. However in this case the ID values are not built into the device but are simply arbitrary values allocated in `include/asm/ocp_ids.h`. Drivers for the on-chip peripherals register themselves when loaded, using the `ocp_register_driver` function and a table of OCP device

from
`arch/ppc/platforms/ibm405lp.c`

```
struct ocp_def core_ocp[]
 __initdata = {
  { .vendor    = OCP_VENDOR_IBM,
    .function  = OCP_FUNC_OPB,
    .index     = 0,
    .irq       = OCP_IRQ_NA,
    .pm        = OCP_CPM_NA,
  },
  { .vendor    = OCP_VENDOR_IBM,
    .function  = OCP_FUNC_16550,
    .index     = 0,
    .paddr     = UART0_IO_BASE,
    .irq       = UART0_INT,
    .pm        = IBM_CPM_UART0
  },
  { .vendor    = OCP_VENDOR_IBM,
    .function  = OCP_FUNC_16550,
    .index     = 1,
    .paddr     = UART1_IO_BASE,
    .irq       = UART1_INT,
    .pm        = IBM_CPM_UART1
  },
  { .vendor    = OCP_VENDOR_IBM,
    .function  = OCP_FUNC_IIC,
    .paddr     = IIC0_BASE,
    .irq       = IIC0_IRQ,
    .pm        = IBM_CPM_IIC0
  },
  { .vendor    = OCP_VENDOR_IBM,
    .function  = OCP_FUNC_GPIO,
    .paddr     = GPIO0_BASE,
    .irq       = OCP_IRQ_NA,
    .pm        = IBM_CPM_GPIO0
  },
  { .vendor    = OCP_VENDOR_INVALID
  }
};
```

Figure 2: OCP device table for 405LP

from `include/asm/ocp.h`

```
struct ocp_def {
  unsigned int   vendor;
  unsigned int   function;
  int            index;
  phys_addr_t    paddr;
  int            irq;
  unsigned long  pm;
  void           *additions;
};

struct ocp_device {
  struct list_head     link;
  char                 name[80];
  const struct ocp_def *def;
  void                 *drvdata;
  struct ocp_driver    *driver;
  u32                current_state;
};
```

Figure 3: OCP device structure

from `drivers/i2c/i2c-ibm_iic.c`

```
static struct ocp_device_id
 ibm_iic_ids[] = {
  { .vendor = OCP_ANY_ID,
    .function = OCP_FUNC_IIC },
  { .vendor = OCP_VENDOR_INVALID }
};

static struct ocp_driver
 ibm_iic_driver = {
   .name          = "iic",
   .id_table      = ibm_iic_ids,
   .probe         = iic_probe,
   .remove        = iic_remove,
};

⋮

  ocp_register_driver(
      &ibm_iic_driver);
```

Figure 4: OCP driver registration (for the IIC driver)

IDs like that shown in Figure 4. This again is analogous to a PCI driver. The OCP subsystem matches the driver against the list of OCP devices, calling the driver's probe routine for each relevant device.

The `index` field is used to distinguish between multiple devices of the same type. The `paddr` and `irq` fields, unsurprisingly, give the device's physical base address (on PPC all IO is memory-mapped) and its IRQ line. The `pm` field is used for power management, we'll look at it in §5.3. Finally, the `additions` field is a hack used to supply extra device-specific information. It is not needed for any of the devices on the 405LP but it is used on some other chips: for example, some 4xx CPUs, such as the 405GP and NPe405H include one or more Ethernet MAC controller (EMAC) units. These make use of a specialised DMA controller known as the Memory Access Layer (MAL). The `additions` field is used to identify which MAL channels are associated with each EMAC—another piece of information that the kernel has to "just know."

The 2.5 version of the OCP system is integrated with the unified device model. At bootup the OCP code registers an OCP `bus_type` and one instance of it—all the OCP devices are registered as devices on this bus.[3] The `ocp_device` and `ocp_driver` structures become wrappers around the device model's `device` and `device_driver` structures.

4 Future approaches

As yet, there is no really convenient and comprehensive way of dealing with embedded "unprobeable" peripherals. The OCP system is

[3] This ignores the distinction between the two on-chip buses, PLB and OPB. We can get away with this because the POB is always enabled and has a fixed configuration, so in practice we can ignore the distinction for the purposes of device discovery.

probably the closest thing to such a system, but it has significant limitations: it only covers PPC 4xx on-chip devices, and its data structure is a flat table so it cannot represent peripherals behind a bus-to-bus bridge or other more complex interconnections.

The fact that some peripherals are built into the CPU chip is interesting from a hardware point of view. However, for the purposes of device discovery, there is little inherent difference between on-chip devices, and devices which are on a separate chip, but which still can't be straightforwardly scanned or probed. It seems worthwhile, then, to extend the idea of the OCP system to cover embedded devices more generally.

The Linux unified device model provides the obvious place to represent information about these devices at runtime: it already provides the code for matching devices to drivers and its tree structure allows multiple buses with configurable bridges between them to be represented.

It is not immediately clear how to represent everything that's needed in the device tree, though. While for many devices the physical address and IRQ number is all the information that is needed, some devices have multiple IRQs and/or IO windows, at discontinuous addresses. Some buses require different resource addresses: for example many of the 4xx on-chip devices need DCR numbers, and I^2C devices need I^2C addresses rather than physical IO addresses. Hence, devices on different buses are likely to need different wrappers around `struct device` providing different address information.

Even less obvious is how to represent devices with multiple connections, such as the audio codec on the eLAP, connected both the the I^2C bus and to the CSI.[4] As yet, the device model does not have a clear way to represent this.

Another as yet unanswered question is how the device information should be represented in the kernel image. In fact, we gain a little flexibility if this information is removed from the kernel proper by having it as a blob of data which is passed to the kernel by the bootstrap loader (the shim between the firmware bootloader and the kernel proper which handles decompressing the kernel and moving it to the correct address in memory). This has the advantage that on those machines which do have a reasonable sophisticated firmware or bootloader, such as PPCBoot/U-Boot (see [7]), the device information can be taken from there.

from `include/asm/bootinfo.h`

```
struct bi_record {
   unsigned long tag;
   unsigned long size;
   unsigned long data[0];
};

#define BI_FIRST               0x1010
#define BI_LAST                0x1011
#define BI_CMD_LINE            0x1012
#define BI_BOOTLOADER_ID       0x1013
#define BI_INITRD              0x1014
#define BI_SYSMAP              0x1015
#define BI_MACHTYPE            0x1016
#define BI_MEMSIZE             0x1017
#define BI_BOARD_INFO          0x1018
```

Figure 5: Boot info records

On PPC systems, there already exists a flexible method of passing data from the bootstrap to the kernel proper through "boot info records." The bootstrap passes to the kernel a list of `bi_record` structures, shown in Figure 5. Each

[4]Note that this is a different problem to multipath IO. That deals with the case where a device can be accessed by any of several routes, here we have devices that requires several connections simultaneously.

`bi_record` is a blob of data with a length and tag, the internal format of the information being determined by the tag (some tag values are also shown in Figure 5). Currently this method is used for passing information such as the size of memory and the board and bootloader versions. This system could be extended to pass an entire set of device information to the kernel (there is no reason a particular `bi_record` couldn't contain a list of further `bi_record`s, giving a tree structure).

A related question is how to represent the device information in the kernel source. The simplest approach would be to directly include the data structure used to represent the information at boot time. However, that's likely to be quite inconvenient to edit and extend, especially if the format includes length fields (like `bi_record`s) or internal pointers. It might be worthwhile, then, to create a *device tree compiler*: a program used during the kernel build to take a text file describing the device layout and generate code or data to be included in the kernel image.

5 Power management

In a battery powered device such as the eLAP, it is clearly important to minimise power consumption. The most obvious way to do this is to power down sections of the system when they are not in use. Obviously, this means that the kernel needs to know when a device is in use, including when it is in use indirectly because another device relies on it.

The topics of power management and device discovery are therefore related: device discovery is about providing exactly the sort of information about the interconnection of devices that effective power management requires.

As yet the integration of power management techniques with detailed device information is very much a work-in-progress, even on conventional systems and doubly so on embedded machines. So, we can only give an overview here of what the major issues are: most of the hard cases remain to be investigated, let alone implemented.

Again we will use the eLAP as our example: the 405LP's power management features introduce some new (and largely unsolved) problems in providing device information for power management. So, we first examine these features, then in §5.2, §5.3 and §5.4 we examine several different methods of reducing power consumption and the device information problems they introduce.

5.1 eLAP power management features

The 405LP CPU is designed especially for low power operation and as such it has some novel and interesting power management features. Most of the on-chip peripherals can be powered on and off under software control. Some also provide more detailed power control to allow power savings when only parts of the peripheral are in use, or when it is in use intermittently. It also allows for several methods of saving the CPU state while shutting down the chip as a whole (i.e., "sleep" modes).

More interestingly, the 405LP includes a clock generation core that allows the clock frequency of the CPU core and also the PLB, OPB and EBC buses to be altered dynamically. The ratios between the CPU and various bus frequencies are not fixed, so the chip can be adjusted differently for IO versus compute performance. While a number of different CPUs allow the frequency to be changed while running, the 405LP can change frequency exceptionally quickly (microseconds) which enables new power management techniques based on dynamically adjusting frequency based on workload and idle periods. The 405LP can also operate at a variety of different voltages, which can provide much greater power savings than just adjusting the frequency (power consumption varies roughly linearly with frequency and cubicly with voltage, maximum frequency is roughly linear with voltage). Another novel feature of the 405LP is that it can continue to operate, albeit slowly in some cases, while a voltage transition is in progress.

As well as supporting the 405LP's features, the eLAP board has some extra power management features of its own. A number of the on-board devices, such as the audio codec, include the ability to power down some or all of their operations while not in use. Other devices, such as the USB and SDIO chips can be powered down under the control of an FPGA register.

5.2 Static power management

We use the term static power management for the process of suspending or sleeping a machine, i.e. saving the machine's state while turning most or all of the machine off, then restoring the state when the machine is powered on again. This of course is normal in everyday laptops, and handling this for embedded machines is not a great deal different.

Embedded devices do introduce some extra complexities, though. On PC laptops, the BIOS (either APM or ACPI) provides some support to the kernel on how to properly suspend the machine (indeed, in the APM case the BIOS handles most of the work itself). Embedded machines, on the other hand, usually require the kernel to know how to suspend and resume the machine directly. For example the suspend code for the eLAP knows how to use the 405LP's features to save the CPU state, how to configure the RAM to enter self-refresh mode, how to use the board's FPGA registers to turn off the board, and how to rebuild the state when the machine is resumed.

In addition, static power management on all machines requires knowledge of what devices' dependencies on each other are, so they can be shut down and later restarted in the correct order—this was one of the major motivations for the creation of the unified device model. Hence, all the complexities of obtaining detailed device information for embedded systems impact on static power management. However static power management doesn't really add further difficulties beyond those we have already discussed for device discovery.

5.3 Peripheral power management

We use the term peripheral power management to refer to disabling and powering down peripheral devices when they are not in use. This is often relatively straightforward, since it can be handled directly by the driver for the device in question. This also delegates the question of when the device is "in use" to the driver. Sometimes it is sufficient to enable power to the device when it is open, and disable it when closed, other times more fine-grained control of the power is desirable, e.g. to take advantage of idle periods.

When the device depends on other devices being enabled, the situation is a little more complex. However the driver will generally know what the other devices are and their drivers, so it is usually quite simple to create an ad-hoc interface whereby one driver can ask the other to enable the device it requires.

Difficulties do arise where power to one device is controlled by another: for example the 4xx on-chip devices are controlled by a central clock and power management unit (CPM). Similarly many boards have devices which are powered on and off by board-specific FPGA registers.

The 4xx CPM has a simple interface, allowing the OCP subsystem to support peripheral power management quite easily. The `pm` field in the OCP device definitions (see Figures 2 & 3) is a mask describing which bit in the CPM registers controls power to this peripheral. Drivers can then use functions from the OCP subsystem to switch the device on and off.

Obviously, for peripherals that are unique to a particular board it is also easy for the driver to directly control power to the device. So far however, little work has been done on the general case where common devices may be powered on and off by board specific controls.

5.4 Dynamic power management

Dynamic power management (DPM) refers to dynamically adjusting CPU frequencies and voltage during operation. This approach is quite new, at least in Linux, and very much under development. The details of the motivations and approaches to dynamic power management are outside the scope of this paper, for more information see [4]. IBM and MontaVista software are collaborating on further development in this area.

DPM does introduces some new problems related to device information. To work properly, devices in operation may have to impose constraints on what frequency or other settings are allowed. For example, a device may require a certain amount of bus bandwidth, and hence impose a minimum bus frequency, or it may require interrupts to be handled without too high a latency, and hence impose a minimum CPU frequency.

These constraint details are somewhat like the basic information about device interconnection that we have already examined, but clearly require even more detailed information about the devices. Since these constraints may well depend on details of the hardware interconnects, this is yet more information which the kernel must "just know."

Again, the unified device model provides an obvious framework in which to represent this information. [4] discusses some methods for setting constraints within drivers. A general approach to representing constraints in a way that is easily extensible to new boards is yet to be implemented, and is likely to take considerable further investigation and development.

References

[1] IBM Corporation, *PowerPC® 405LP Embedded Processor User's Manual*, Preliminary, 2002.

[2] IBM Corporation, *PowerPC® 405GP Embedded Processor User's Manual*, Seventh Preliminary Edition, 2000.

[3] IBM Corporation, *PowerNP NPe405H Network Processor User's Manual*, Preliminary, 2002.

[4] IBM and MontaVista Software, *Dynamic Power Management for Embedded Systems*, Version 1.0, `http://www.research.ibm.com/arl/projects/papers/DPM_V1.1.pdf`, 2002.

[5] `linuxppc_2_4_devel` kernel tree, `bk://ppc@ppc.bkbits.net/linuxppc_2_4_devel`.

[6] `linuxppc-2.4` kernel tree, `bk://ppc@ppc.bkbits.net/linuxppc-2.4`.

[7] PPCBoot homepage, `http://ppcboot.sourceforge.net/`.

About the author

David Gibson is an employee of the IBM Linux Technology Center, working from Canberra, Australia. Most recently he has been working on board and device bringup for Linux on embedded PowerPC machines, along with various bits of kernel infrastructure for cleanly supporting PowerPC 4xx and other system-on-chip CPUs. He is also the author and maintainer of the orinoco driver for Prism II based 802.11b NICs. In the past he has worked on ramfs (as included in the -ac kernel tree), and "esky," a userspace implementation of checkpoint/resume.

Legal Statement