Power Management in the Sharp Zaurus

Miquel Pericàs Gleim

26th January 2004

1 Introduction

Power Management has become an integral part of any portable system or handheld. Battery technology has not evolved at the same rate as microelectronics, and as a consequence of increased power consumption by chips, battery life has shortened. In both notebooks and handhelds a trade-off needs to be established between battery life and battery size. Large batteries can be huge and heavy, a property that does not fit well with portable systems. Smaller batteries, on the other side, have short lifetimes.

This is especially critical in handhelds, where batteries tend to be very small, usually storing less than 1000mAh. Such a system, despite using all sorts of low-power electronics, has an expected lifetime that is no longer than two hours, with all features turned on\footnote{this data refers to a Sharp Zaurus model 5600 while reproducing an mpeg1 movie with frontlight at its maximum contrast.}. Although I chose to give this value, reporting battery life is actually very hard because each vendor uses different benchmarks to measure this and, thus, the result should be taken with a bit of caution. A lifetime of two hours is clearly unacceptable. In the case of notebook computers, even though they qualify as mobile equipment, the choice of connecting the hardware to the power grid is often available. With handhelds, as with mobile phones, this is often not a possible. As a result, handheld computers offer a great deal of power management capabilities that will give the system a longer life expectancy.

Which techniques allow the system to have longer life expectancy? Basically, the idea is that most features in such a system are mostly not used and that a lot of power can be saved by turning these units "off". In fact, modern handhelds are mostly completely disabled waiting for some sort of interrupt to wake them up.

In this work we will analyze the power management features of an handheld device such as the Sharp Zaurus SL5500. In section 2 the working environment will be described. Section 3 will describe the power management features of the StrongARM 1110, the core of the SL5500. Finally, in section 4 we will describe how the linux kernel handles power management in the sharp Zaurus SL5500 and what can be done to improve power management.
2 The Environment

This work has been done focusing on the Sharp Zaurus model SL5500. This PDA is equipped with a StrongARM 1100 processor clocked at 206MHz. The system consists of a keyboard, a touchscreen, two expansion slots (Compact Flash and MultiMedia Card) and several other minor IO devices. The system is operated by a linux kernel configured and patched by Lineo, the maintainer of the Embedix Linux project. The kernel version is 2.4.18.

During this project, the kernel has been analyzed and recompiled using a gcc crosscompiler for the ARM4 platform. Two gcc versions have been tested: gcc-2.95.3 and gcc-3.3.2. For stability reasons, version 2.95.3 has been used. The kernel has not been tested using the ROM that Sharp provides. Instead, the OpenZaurus ROM (release 3.3.6-pre1) has been used for the tests. The ROM contains the root filesystem, which consists mostly of applications, but also of kernel modules, some of them closed. Thus, no kernels other than the provided 2.4.18-rmk7-pxa3-embedix could be tested. Some of the modules provide important functionalities. For example, without the ROM it was not possible to change the brightness of the frontlight. The frontlight has significant power consumption, so it was imperative to have this feature available.

Tests are now being conducted to port the linux-2.6 kernel to the Zaurus SL5500. This kernel includes some interesting features like clock scaling, which is not part of the 2.4 series. However, the feature has been backported, and so the 2.4.18 kernel of the Embedix project, already includes the corresponding patches for this feature to work. Clock scaling is an important feature because power consumption is proportional to the operating frequency of a device. From VLSI theory we know that the power consumption of a electronic device is more or less \( P = CVdd^2af \), where \( C \) is the equivalent capacitance, \( Vdd \) is the supply voltage, \( a \) is the activity factor and \( f \) is the frequency. This gives an idea of the possible power reductions that can be derived from an efficient use of clock scaling. We will come back to this topic at the end of the fourth chapter.

3 Power Management features of the SA1100

We will now describe the power management features of the SA1100. We referred to the documentation of the SA1110 model in this document because the it is more widely available. Note that this is not important because the core of the SA1110 is a SA1100 with some minor additions that are unrelated to power management.

At each instant, the SA1110 can be in one of three modes of operation: run, sleep and idle.

The chip contains power management logic that controls the transitions between these three modes. These modes are used to reduce processor power consumption when some functionalities are not needed, or when the power supply is low or out of regulation. Figure 1 shows a diagram of the modes along with the possible transitions. We will now comment on the features of each mode afterwards.

3.1 Run Mode

Run mode is the normal operation mode of the SA1100. In this mode, all features are enabled and no power management features are active. Particularly, all power supplies are enabled,
Figure 1: The SA1100: diagram of operation modes and the transitions between them
all clocks are running and all on-chip components are functional. This is the normal mode of operation when the SA1100 is executing code. The processor enters run mode usually after reset or power-up of the system.

### 3.2 Idle Mode

The SA1100 allows a program to stop the CPU. As a result, the chip enters idle mode. Once in this mode, the processor continues to monitor interrupt requests from both on-chip and off-chip devices. An interrupt reactivates the processor. Several units remain fully functional in this mode: the System Control Module (SCM), the Memory and PCMCIA Control Module (MPCM) and the Power Manager (PM).

In idle mode the CPU is stopped, but since the SA1110 is static, no information is lost. This is an important feature because it enables the processor to switch back into run mode and resume operation exactly at the same place where it has been stopped. During idle mode, all other on-chip resources are active, including: all system unit modules (real-time clock, operating system timer, interrupt controller, general-purpose IO, and power manager), all peripheral unit modules (DMA controller, LCD controller, and serial controllers 0-4); and all memory controller resources. The PLL (phase locked loop) also remains in lock so that the part can be quickly brought back into operation when an interrupt occurs.

Idle mode is entered while in run mode by issuing a series of three instructions consecutively:

1. ‘disable clock switching’ instruction
2. a load from non-cacheable location
3. ‘wait for interrupt’ instruction

The following assembler code is an extract from the cpu_sa1100_idle procedure in the linux kernel. It implements the three instruction sequence that puts the SA1100 into idle mode:

```assembly
ldr r1, =0xf0000000
mcr p15, 0, r0, c15, c2, 2 // disable clock switching
ldr r0,[r1]
mcr p15, 0, r0, c15, c8, 2 // wait for interrupt
```

Any enabled interrupt brings the processor back from idle mode into run mode. An interrupt is masked/unmasked using the Interrupt Controller Mask Register (ICMR). When the interrupt arrives, the ‘wait for interrupt’ instruction is completed and the execution resumes. In any case, clock switching needs to be reenabled after the interrupt arrives.

Asserting the nRESET pin also triggers a transition from idle mode into run mode. This can also be achieved by having the OSMR3 (Operating System Timer Match Register 3) configured as watchdog.
3.3 Sleep Mode

When in sleep mode, the SA1100 offers the lowest level of functionality. In the transition from run/idle to sleep, the SA1100 performs an orderly shutdown of on-chip activity and applies an internal reset to the processor. There are two events that transition the processor into sleep mode: software control operations or power supply faults.

Entry into sleep mode through software control is accomplished by setting the force sleep bit in the power manager control register (PMCR). This bit is set once before sleep by the software and cleared by the hardware during sleep mode. When the system wakes up from the sleep the bit is already cleared.

Entry into sleep via a power supply fault is caused by the assertion of the VDD_FAULT pin or the BATT_FAULT pin. VDD_FAULT indicates that the main power supply is out of regulation. BATT_FAULT, on the other side, should be used to indicate that the battery has been removed or is low.

The shutdown sequence consists of three steps:

1. The following actions occur during the first step:
   a. First the power manager switches the GPIO (general purpose input/output) pins to their sleep state (which has been previously programmed with the GPIO Sleep State Register, the PGSR).
   b. The DRAMs are placed into self-refresh mode. This operation is critical because no data in the DRAMs can be lost during sleep if the processor has to be able to continue execution from the same machine state.
   c. If the sleep state was entered due to the assertion of VDD_FAULT or BATT_FAULT, the possible wake-up sources are reset from what has been programmed by software to their “fault state.”

2. In the second step happen the following actions:
   a. All wake-up sources are cleared. In case of sudden power faults, the processor cannot prepare for the event, so the resulting state will be random. Clearing all wake-up sources is necessary to prevent an immediate-wakeup due to some event that is related to the system state and is not an intentional interrupt.
   b. An internal reset is applied to the SA1100

3. In the third step the following occurs
   a. Depending on the status of the OPDE (Oscillator Power-Down Bit) the 3.686 MHz oscillator is stopped. If this action is performed, the sleep results in greater power savings, but the wake-up sequence is also slower.
   b. The processor signals the external system to disable the VDDI power supply. This further prevents power consumption.
An equivalent sequence takes the SA-1110 from sleep mode back to run/idle. During sleep mode the system waits for preprogrammed wake-up events, either programmed by the CPU prior to setting the force bit, or programmed by the power manager when a fault condition is detected. When the nRESET line is asserted all saved contents (including DRAM are lost). Thus, after such an event the SA1100 will produce a common boot sequence.

### 3.4 Power Management Registers

The SA-1100 has eight 32-bit registers to perform power management related operations. The following list shows the functionality of each register:

- **PMCR (Power Manager Control Register):** allows software invocation of sleep mode
- **PSSR (Power Manager Sleep Status Register):** tells the processor which event brought the cpu to sleep
- **PSPR (Power Management Scratch Pad Register):** stores the processor state during sleep
- **POWER (Power Manager Wake-up Enable Register):** used to program the desired wake-up sources in the system
- **PCRF (Power Manager General Configuration Register):** controls various configurable features within the SA-1100
- **PPCR (Power Manager PLL Configuration Register):** allows the user to change the PLL operating frequency
- **PGSR (Power Manager GPIO Sleep State Register):** used to program the value loaded onto GPIO outputs when the SA-1100 transitions into sleep mode
- **POSR (Power Manager Oscillator Status Register):** contains a single bit that indicates whether the 32KHz oscillator has stabilized after a hardware reset

This set of registers is memory mapped and can be accessed using pointers from within the kernel memory space.

### 3.5 Other hardware in the Sharp Zaurus SL-5500

The SA1100 is the heart of the Zaurus' processing system and probably the most power-consuming device. Alone it is not very usable. Around this CPU the Zaurus includes a lot of IO devices that provide the interface to the handheld. There are many such devices, like the compact flash controller or the SanDisk (or MMC) controller, but here we will only be interested in the touchscreen frontlight, another component that requires considerable power. The frontlight is controlled by an integrated controller from Philips, the UCB 1200, whose functionality includes the touchscreen (and the frontlight), the audio interface and an analog-digital converter. We will not study the UCB in detail as we did with the SA1100 power management interface. We will refer to the UCB in the next section when dealing with functionalities inside the linux kernel.
4 Power management support in the linux kernel for the SA1100-based Sharp Zaurus SL5500

In the previous section we have commented on the hardware support that the SA1100 contains for power management. We presented the three operation modes and commented the possible transitions among them. Hardware-triggered transitions are triggered by signals that are sent to pins. Theoretically we could generate these signals by hand, for example, by removing the battery when in run mode. Practically this is not possible, because the Zaurus sends an reset signal while the battery pod is open.

Thus we can only generate software signals to switch among operation modes. A software control signal is generated, for example, when pressing the On/Off button in the front of the Zaurus device for a couple of seconds. This event is captured and interpreted as a poweroff request. The SA1100 is then sent into sleep mode. This events can only take place under the control of a software piece. In the case of the Zaurus this piece of code is the linux kernel.

Although the task of sending the Zaurus into sleep mode may seem a task that offers little reconfigurability, the kernel has control of many more modules where specific functionalities to reduce power consumption can be applied. A trivial example, and a very important one, is what to do in case of user inactivity. For instance, imagine a user who is checking his TODO list for the present day and after reading a long list of appointments finally leaves his Zaurus handheld on the table and keeps staring into the void. The Zaurus device still shows the TODO list...to nobody. And possibly the frontlight may be at its brightest position. If the kernel takes no action, the user may find that once he returns to his crude reality his Zaurus is out of battery, and he has not even started working.

This example could not take place today because all handhelds include functionality to turn off the device after some time of inactivity. How much time and which modes of operation is up to the kernel, and often can be configured by the user.

Hardware support without kernel support is thus meaningless. This section will cover the possibilities that the linux kernel\textsuperscript{2} offers to control power management and reduce power consumption.

4.1 How Linux manages the operation modes in the Zaurus SL-5500

We will start with an analysis on how linux performs the transitions between the different operation modes. We will only look at how linux switches from run mode to either idle mode or sleep mode because these are the transitions that actually can save power. The process of going from a low-power mode to a higher power mode will not be covered.

The relevant code regarding operation modes can be found in the arch/arm/mach-sa1100/ directory. The Sharp Zaurus is normally referred to as collie, so this name will be appearing frequently in file names. The more interesting files in this directory are: collie_power.c, collie_battery.c, collie_apm.c and collie_suspend.S.

In the previous section we commented on the code sequences necessary to go into idle mode and sleep mode. In the Linux kernel, these functionalities are available using the functions

\textsuperscript{2}the kernel as distributed by Embedix
cpu_sa1100_do_suspend() and cpu_sa1100_idle(), which respectively transition the SA1100 into sleep and idle mode. Both functions are defined in the assembler file collie_suspend.S.

We will assume for now a Zaurus device in run mode with all functionalities available and every on-chip device turned on. The following two sections analyze how the Zaurus switches to sleep mode and how it switches to idle mode.

4.1.1 Going into sleep mode

A Zaurus transitions only to sleep mode when it is not supposed to continue any activity soon. In sleep mode, there is no possibility to store information in the SA-1100 because it performs an internal reset and turns off the power. Thus, after a sleep mode transition the Zaurus needs to perform some additional actions to recover the system state. Fortunately, thanks to the self-refresh option, the DRAMs do not lose information, thus the kernel does not need to be rebooted after sleep mode.

In the tested kernel the power management was performed completely through the advanced power management interface (APM). Only three events will signal the linux kernel to bring the Zaurus into sleep mode:

- The battery level is too low to continue operation, no AC connector is online and an error happened with the battery. This triggers a CRITICAL_SUSPEND event inside the APM module which brings the Zaurus into sleep. This event cannot be aborted.

- A sysctl can send the system into sleep mode. The sysctl has the name pm.suspend and can be accessed using /proc/sys/pm/suspend. Accessing this file brings the Zaurus to sleep\(^3\). As we'll see, this feature is used by OpenZaurus to turn off the Zaurus after a period of inactivity.

- Finally, a series of 150 consecutive IRQs generated by the poweroff button will generate an APM_USER_SUSPEND event in the APM module that will put the Zaurus to sleep.

It's expected that the hardware conditions VDD_FAULT and BATT_FAULT will also bring the device to sleep but this cannot be checked manually because the Zaurus architecture makes this impossible. Opening the battery compartment automatically generates a hardware reset that is not deasserted until the compartment is closed again. From the design point of view, the Zaurus should always be brought to sleep via software, never via hardware.

4.1.2 Going into idle mode

The kernel transitions to idle mode by calling cpu_sa1100_idle(), which is defined in collie_suspend.S. As discussed in section 3, only software can put the SA1100 into idle mode. The linux kernel calls the idle feature only from the main apm loop. Within the kernel, apm executes as a kernel thread\(^4\) that waits on external events or on self-programmed alarms/timeout. The execution flow of this thread is the following\(^5\):

\(^3\)the actual command that we used for the tests was: echo 1 > /proc/sys/pm/suspend
\(^4\)the name of the thread being kapm-idled
\(^5\)in simplified form
void apm_mainloop(){
  for(;;){
    timeout = APM_CHECK_TIMEOUT;
    schedule_timeout(timeout); // schedule timeout and sleep
    apm_event_handler();       // check all events
    if (!system_idle())        // if the system is not idle
        continue;             // then continue
    apm_do_idle();             // otherwise enter idle mode
  }
}

This procedure has two important functions: handling events and checking for inactivity. The `apm_event_handler()` checks for the event that woke up the apm thread. There are various events that will wake up the apm daemon, such as standby/suspend/resume events, a low battery event, update time and also critical suspend, which we have already commented in the previous section. But our interest here is in the other type of event. It’s possible that `apm_event_handler()` returns without having processed any event. In this case, what happened is that the programmed alarm timed out. The kernel then checks for any activity in the system with `system_idle()`. If no activity is detected the kernel puts the device into idle mode by calling `apm_do_idle()`. The amount of time that passes between the last event and a transition to idle mode can be controlled by modifying `APM_CHECK_TIMEOUT` which for the tested version was equivalent to the number of clocks per second (ie, 100). The timeout parameter is also given in clocks per second. Thus the system goes into idle mode after 1 second of inactivity.

From the point of view of the user, idle mode does not exist. There are no appreciable differences between run mode and idle mode and the transition also does not cause any event on the screen. The idle mode just stops the CPU in a way in which it can reboot almost instantly. When the user generates a new event the system resumes instantly without any appreciable change. The power saving exist, but they are not very big. Sleep mode is required to minimize power. But on the other side, the wake-up sequence of the sleep mode is much more costly, and, during sleep mode, the screen is off and does not display any information.

4.2 Managing the operation modes and frontlight from the user software

We have now commented how the APM daemon transitions the Zaurus into various states. But power management is not limited to kernel control. The kernel also offers services so that user software can modify the operation modes. Further, other power reductions are available to the user via driver access.

We commented in the previous section how the kernel offers a sysctl `pm.suspend` that allows the system to be sent into sleep mode by user request. The OpenZaurus’ desktop environment Opie includes functionality to turn the Zaurus into sleep mode when the user has been away for too much time. This is detected as inactivity for a large period. The functionality is accessed through the Light and Power applet in the Settings tab. One of the several parameters that you can modify is the Suspend after option. This allows to set the period of inactivity after which Opie
will call the `pm.suspend` `sysctl` to enter sleep mode. This feature is very important. Otherwise, if a user forgets to turn off his device, it would remain in idle mode consuming a great deal of power.

We have now seen methods to reduce power by operating the operation modes through software. Although these techniques are very effective, there are still other options for reducing the power consumption even further. Of course we can always reduce the `Suspend after` parameter, but having the Zaurus enter sleep mode after say, 10 seconds, is excessive and not very user friendly. One option to reduce power during idle mode is to reduce the brightness of the screen.

The touchscreen frontlight is a source of huge power consumption in a portable device. A user learns soon in how far reducing the brightness of his screen will increment the battery life of his device. In the previous section we saw how Opie puts the system into sleep mode after a period of inactivity. As we will see, the approach with the frontlight is exactly the same. At kernel level, the only difference is that this functionality is available through the `ioctl()` interface instead of the `sysctl()` interface.

The relevant source files related to frontlight management are in the `drivers` directory, particularly `char/collie_ucb1200_ts.c` (the touchscreen driver) and `video/collie_frontlight.c` (the frontlight driver and ioctl interface). When the Linux kernel boots it attaches the frontlight driver to the device `/dev/collie-fl`. This device can be opened using the `open()` interface and later it can be operated using `ioctl` operations. The `ioctl` interface defines five operations:

- `COLLIE_FL_IOCTL_ON`: turns the frontlight on
- `COLLIE_FL_IOCTL_OFF`: turns the frontlight off
- `COLLIE_FL_IOCTL_STEP_CONTRAST`: sets a specific contrast level
- `COLLIE_FL_IOCTL_GET_STEP_CONTRAST`: gets the current step
- `COLLIE_FL_IOCTL_GET_STEP`: gets the range of steps

The frontlight is operated completely using a single function to set the contrast level: `colliefl_step_contrast_setting()`. This function, defined in `collie_frontlight.c`, receives one parameter, the `step`, and modifies the frontlight level accordingly. Note that in the context of the Zaurus, the brightness level and the contrast level are the same.

In OpenZaurus, the user is allowed to modify the contrast level by using the same `Light and Power` applet. This application allows to modify three parameters related to the frontlight management: the time before the light is dimmed, the period before the light is completely turned off and the contrast level during normal operation. Dimming the light is accomplished by reducing the contrast in one step. The period before dimming has to be smaller than the period before the light is completely turned off. Equally, the period before suspend has to be larger than the period before the light is turned completely off. Note that turning the light off is still different from putting the Zaurus to sleep. First, returning into run mode is instantaneous and second, even though the light has been turned off, the LCD is still active and shows the current screen. This can be seen when the environment is bright enough. However, it’s hard to work in these conditions, thus operating the Zaurus without frontlight is very uncommon. Finally the applet also allows to set the level of brightness during normal operation. In the Zaurus model there are
five modes of contrast available. Any of the modes can be set. The frontlight can even be turned completely off. In this mode, the dimming has no effect, but its not very usable so this is more or less anecdotal. The look of the Light and Power applet can be seen in Figure 2.

4.3 CPU frequency scaling

We commented before how the \texttt{linux-2.6} kernel includes functionality for scaling the speed of the CPU. This feature is called \textit{CPU frequency scaling}. Because frequency is in direct relationship with power, this feature is very interesting. If the frequency can be scaled down 50\% we will get approximated power savings of also 50\%.

In The Zaurus SL5500, frequency scaling is by changing the lower bits of the \textit{Power Manager PLL Configuration Register} (PPCR). DRAM timings are closely related to the core clock speed so these need to be changed, too. The used registers are:

- \texttt{MDCNFG 0xA0000000} DRAM config
- \texttt{MDCAS0 0xA0000004} Access waveform
- \texttt{MDCAS1 0xA0000008} Access waveform
- \texttt{MDCAS2 0xA000000C} Access waveform
This feature is interesting enough so that it has been backported to some trees of the linux-2.4 development branch. Among others, the kernels distributed by Embedix includes this feature, although it is not enabled by default. To test this feature you have to access the General Setup menu and the enable the option Support CPU clock change (EXPERIMENTAL). As you see, this feature is classified experimental. Changing the clock is not a simple task. Many side effects can happen that can result in malfunction if done without care. For example, changing the clock frequency in the Zaurus requires to change the DRAM timings. If this is not done carefully, DRAM and CPU may run out of sync and the system will stop working.

As with the suspend feature, accessing the clock changing code is done via the sysctl interface. The SA1100 CPU frequency scaling code can be located in arch/arm/mach-sa1100/cpu-sa1100.c, while the global frequency scaling code plus sysctl managing is located in the source file kernel/cpufreq.c. The code defines three sysctl nodes:

- /proc/sys/cpu/0/speed: contains the current frequency in Hz. In the SL5500 this value is by default 206400
- /proc/sys/cpu/0/speed-min: contains the minimum value of the frequency in Hz. By default it is 59000
- /proc/sys/cpu/0/speed-max: contains the maximum value of the frequency in Hz. By default it is 287000

The file arch/arm/mach-sa1100/cpu-sa1100.c contains all valid cpu frequencies along with the associated DRAM timings.

After recompiling and booting the kernel the three sysctl files are available in the sysctl file system. We tested this feature by writing other allowed frequency values into the speed file, but currently without positive results. Writing values does not affect the speed file, and unfortunately the busybox unix utilities do not include a sysctl tool\(^6\) so we were limited in our tests.

5 Conclusions

Although the limitations of battery life are expected to be present for a long time we are not completely out of luck. Many hardware techniques exist that can reduce the power consumption of devices. In the SA1100 these techniques consist in various operation modes with various functionality/power–consumption trade–offs. Although we were not able to test this, frequency scaling is another very interesting feature of modern embedded hardware. Other options for reducing the power in handhelds include better management of the frontlight. We tested these features in the Sharp Zaurus SL5500 handheld and studied how the linux kernel manages them. We saw that many actions can be taken to reduce power consumption, but we also noted how adding more and more features decreases the level of user–friendliness of the device. First, many users probably do not want to struggle with power management issues and second, the constant changes in the frontlight can be annoying.

\(^6\)Trying to write a sysctl command tool ourselves in C with arm-linux-gcc resulted in gcc complaining that _sysctl() was not a valid symbol.