Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

Power reduction techniques for TFT LCD displays

Decostre, Alain

Award date:
2003

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 2002-2003

Power Reduction Techniques for TFT LCD Displays

Alain DECOSTRE

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique

UBS 10027165

Abstract

Energy consumption has become a major concern in portable devices design. The power minimization has traditionally focused on circuit design. More recently, software based techniques, named Dynamic Power Management (DPM), have emerged as attractive approaches to save energy.
The display is one of the main contributor to the overall power budget. In this dissertation we show how to integrate such DPM techniques in a Linux device driver controlling a TFT liquid crystal display. We also detail the implementation of the techniques "Variable dot clock" and "Variable frame refresh" on a Personal Digital Assistant (PDA) using the Linux operating system. A power consumption reduction of respectively 6.4 % and 10 % is observed. Finally we outline a solution for a third technique : the "Liquid crystal orientation shift".
By this work we demonstrate that a software adaptation in the operating system, without modifying the hardware, can lead to an appreciable reduction of the power consumption in a portable system. Therefore the DPM techniques can be applied to improve the lifetime of existing portable system batteries.

Résumé

La consommation d'énergie est devenue un centre d'intérêt majeur dans la conception d'appareils portables. La minimisation de l'énergie s'est traditionnellement concentrée sur la conception de circuits. Plus récemment, des techniques orientées logiciel, appelées gestion dynamique de la consommation d'énergie (en anglais DPM), se sont révélées être une approche intéressante pour économiser de l'énergie.
L'écran est l'un des principaux consommateurs d'énergie du système. Dans ce mémoire, nous montrons comment intégrer de telles techniques DPM dans un pilote de périphérique Linux contrôlant un écran TFT à cristaux liquides. Nous détaillons aussi l'implémentation des techniques "Variable dot clock" et "Variable frame refresh" sur un Assistant Personnel Digital (en anglais PDA) utilisant le système d'exploitation Linux. Une réduction de la consommation électrique de respectivement 6.4 % et 10 % est observée. En dernier lieu, nous esquissons aussi une solution pour une troisième technique appelée "Liquid crystal orientation shift".
Par ce travail, nous démontrons qu'une adaptation logicielle du système d'exploitation, sans modification du matériel, peut apporter une réduction appréciable de la consommation électrique dans un système portable. Les techniques peuvent donc être appliquées pour améliorer la durée de vie des batteries des systèmes portables existants.

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to realize this work. First, I want to thank my supervisor, Professor Jean Ramaekers, for his very relevant advice and comments, and the reading of my successive drafts.

I would like also to thank all the members of the Micrel laboratory of the University of Bologna for their welcome and their help throughout my stay in Bologna. In particular, I am grateful to Professor Luca Benini who facilitated my integration in his team and supervised my training, and Assistant Professor Andrea Acquaviva whose help, stimulating advice and encouragement were greatly useful.

Finally, I would like to express a special thank you to my family for its support during my studies and particularly during the preparation of my dissertation.

Contents

Introduction **13**

1 Dynamic power management **17**

1.1 Introduction 17
1.2 Dynamic power management at component level 18
1.2.1 Power state model 19
1.2.2 Under the black box 20
1.2.3 Internal controller and external controller 20
1.3 Dynamic power management at system level 21
1.3.1 Structure of a system-level power manager 21
1.3.2 Power manager controller 22
1.3.3 Power manager observer 24
1.3.4 Power manager policies 25
1.4 Dynamic power management at network level 28
1.5 Examples of dynamic power management techniques 28
1.5.1 Battery-driven dynamic power management 28
1.5.2 Power management for the processor: Dynamic voltage scaling . . . 29
1.5.3 Selective instruction compression for memory energy reduction . . . 32
1.5.4 Power management for hard disks 35
1.5.5 Transport protocol optimization for energy efficient wireless embedded system 36

2 System description **39**

2.1 Overview of the Assabet 39
2.2 Frame buffer and palette memory organization 39
2.3 Refreshing the screen 42
2.4 The LCD controller 43
2.4.1 The LCD Controller registers 43
2.4.2 The LCD Controller Control Registers (LCCR) 44
2.4.3 The LCD Controller Status Register ($LCSR$) 45

2.4.4 The LCD Controller DMA Registers (*DBAR* and *DCAR*) 45

3 The Linux operating system **47**
3.1 Introduction 47
3.2 User mode and kernel mode 48
3.3 Filesystem 49
3.4 Device driver and device file 52
3.4.1 Device driver 52
3.4.2 Device file 52
3.5 The concept of module 54

4 The Linux device drivers **57**
4.1 General description of a Linux device driver 57
4.1.1 Definition and roles of a device driver 57
4.1.2 Level of kernel support to hardware device 58
4.1.3 Classes of devices 59
4.2 Char drivers 61
4.3 A methodology for device driver designing 66
4.3.1 The ideal device driver 66
4.3.2 Design methodology 66

5 Interrupts **69**
5.1 Polling mode and interrupt mode 69
5.2 Interrupts handling 70
5.2.1 Interrupts detection 70
5.2.2 Interrupt handler selection 71
5.2.3 Interrupt handler role and working 72
5.3 The initialization 73
5.4 Bottom and top half parts of an interrupt handler 74
5.5 Task queues 75
5.6 Kernel task queues 77
5.7 Tasklets 78

6 The LCD controller frame buffer driver **79**
6.1 Device dependent and independent levels 79
6.2 The initialization 80
6.2.1 Initializing the main device-related data structures 80
6.2.2 Inserting the "controller task" into the task queue 82
6.2.3 Initializing the frame buffer memory 83
6.2.4 Installing the interrupt service routine 83

6.2.5 Defining the content of the LCD controller registers 83
6.2.6 Registering the frame buffer driver 85
6.2.7 Registering the device power management function 85
6.2.8 Updating the controller registers and enabling the LCD screen and the controller 85
6.3 Updating the palette 88

7 Driver adaptation for power reduction of a TFT LCD display 91
7.1 Software techniques for low-power TFT LCD displays 91
7.1.1 Variable dot clock 92
7.1.2 Variable frame refresh 92
7.1.3 Liquid crystals orientation shift 92
7.1.4 Backlight shifting 93
7.2 Implementation 93
7.2.1 Variable dot clock 93
7.2.2 Variable frame refresh 95
7.2.3 Liquid crystals orientation shift 96
7.3 Experimental results 97
7.3.1 Variable dot clock 97
7.3.2 Variable frame refresh 99

Conclusion 101

Bibliography 103

Table of acronyms 107

Glossary 109

Appendix 115

A LCD Controller Registers 117
A.1 LCD Controller Control Register 0 (LCCR0) 117
A.2 LCD Controller Control Register 1 (LCCR1) 118
A.3 LCD Controller Control Register 2 (LCCR2) 118
A.4 LCD Controller Control Register 3 (LCCR3) 119
A.5 DMA Channel 1 Base Address Register 120
A.6 DMA Channel 1 Current Address Register 120
A.7 DMA Channel 2 Base Address Register 120
A.8 DMA Channel 2 Current Address Register 120
A.9 LCD Controller Status Register 121

B Applications **123**
B.1 PCD.c 123
B.2 LENON.c 124
B.3 LENOFF.c 125

C fbmem.c **127**

D sa1100fb.c **147**

E fb.h **187**

F sa1100fb.h **199**

List of Figures

1.1 Example of power state machine. 19
1.2 Structure of a system-level power manager. 22
1.3 ACPI interface. 23
1.4 The power manager observes requests. 24
1.5 Applications control power states directly. 25
1.6 Timeout-based policy. 27
1.7 Predictive shutdown policy. 27
1.8 Processor-memory normal architecture (a) and power-saving architecture (b) 33
1.9 Processor-Memory second power-saving architecture 34
1.10 Example of sessions for different thresholds. 36

2.1 The Assabet. 40
2.2 Frame buffer organization with a palette. 41
2.3 Frame buffer organization without palette. 41
2.4 The display timings. 42

3.1 A view of the kernel. 50
3.2 An example of filesystem. 50
3.3 The device driver interface. 53
3.4 Access to the modules symbols. 56

4.1 Level of kernel support (Left: No support at all. Central: Minimal support. Right: Extended support). 59

5.1 Detection of an interrupt by the processor. 71
5.2 Selection of the interrupt handler and access to the ISRs. 73
5.3 A task queue. 75
5.4 Initialization and execution of a task queue. 76

6.1 Defining the content of the LCD controller register. 84
6.2 Updating the controller registers and enabling the LCD screen and the controller 87

6.3 Updating the palette. 89

7.1 Updating the fb_var_screeninfo structure. 94
7.2 The active window coordinates. 96

Introduction

Energy consumption has become one of the primary concerns in electronic systems design due to the recent popularity of portable devices such as laptop computers, Personal Digital Assistants (PDAs) and cellular phones. The battery capacity has improved very slowly while the computational demand (for instance multimedia applications) has drastically increased over the same time. Some low-power circuit design techniques have helped to increase battery lifetime. On the other hand, managing power dissipation at higher levels can significantly reduce energy consumption and increase battery lifetime. That is the reason why power management techniques, under the name of Dynamic Power Management (DPM), have recently emerged as an attractive alternative to inflexible hardware solutions.

Dynamic Power Management is in fact a design methodology aiming at controlling performance and power levels of digital circuits and systems, with the goals of extending the operational autonomy time of battery-powered systems, providing acceptable performance degradation when energy supply is limited, and adapting power dissipation to satisfy environmental constraints. DPM encompasses a set of techniques that achieve energy-efficient computation by selectively turning off (or reducing the performance of) system components when they are idle (or partially unexploited).

DPM applies to the various components of a system, in particular to the Liquid Crystal Display (LCD) of portable systems, such as laptop computers and PDAs. Display power consumption is often one of the most significant contributor to the overall power budget for many portable devices, especially when multimedia applications are run. There are four main software techniques to reduce power consumption of a LCD display. These techniques do not require hardware changes. Therefore they are applicable on existing systems by simply modifying the driver in the operating system.

The first technique, called *Variable dot clock*, reduces the dot clock frequency used for processor-display communication, decreasing the display refresh rate as a side effect. The second technique is named *Variable frame refresh* and exploits the property of liquid crystals of the display, which maintains their orientation for a while when they are not polarized.

Advantage of this time is taken to reduce power consumption by disabling the LCD controller. The third method, the *liquid crystal orientation shift* technique, is based on the observation that LCD power consumption is proportional to the image luminance. The luminance of useless parts of the screen can therefore be reduced in order to save energy. Finally, the fourth method, called *Backlight shifting*, proposes to adapt the luminance intensity of the display according to the environment luminance. This method derives from the observation that the display has a high visibility in poorly illuminated environment because of the high contrast.

A driver including these techniques is implemented in the ECOS real-time embedded operating system [1]. The present dissertation reports our implementation work of the first two techniques in a device driver for the Linux operating system and outlines a solution for the third technique. Implementing these techniques was performed during my training period at the Department of Electronics, Computer Science and Systems (DEIS) of the University of Bologna (Italy), from September 2002 to January 2003.

This dissertation is organized as follows. The first chapter describes the basic concepts of Dynamic Power Management. Three DPM levels are considered: the component level, the system level and the network level. Examples of power reduction techniques are also included. Each example concerns one of the main elements of a classical electronic system. This chapter has been written in cooperation with Benjamin Briquet.

The second chapter gives an overview of the hardware platform, the Assabet, used to implement the power reduction techniques. We explain the frame buffer memory organization and the LCD controller registers, because these elements are driven by the driver we have adapted.

In the third chapter we present the Linux operating system which was installed on the Assabet. In particular we describe how Linux allows applications to manage I/O devices and emphasize the notion of file, which is the basic concept of the Linux system.

The fourth chapter is related to the basic concepts of a Linux devices driver. We introduce the three main classes of device drivers and describe the basic functions of a char driver, since the LCD driver we have adapted owns to this class. We also propose a methodology for device driver designing.

The fifth chapter describes how the processor detects an interrupt and launches the appropriate interrupt handler. It also explains the concept of task queue, which is used to implement the Variable frame refresh technique.

In the sixth chapter we give a description of the sa1100fb.c driver, i.e. the driver we have enhanced for power management. We explain how a configuration change is effectively performed. This is another personal contribution to this dissertation subject.

The last chapter develops the power reduction techniques for a TFT[1] LCD display. The implementation and observed measurements are described for the Variable dot clock and the Variable frame refresh techniques. A solution is outlined for the Liquid crystal orientation shift technique.

In conclusion we emphasize the effectiveness of DPM methodology for reducing power consumption in electronic systems.

[1]Thin-Film Transistor.

Chapter 1

Dynamic power management

1.1 Introduction

One of the most important technical evolution of the last decade has been the emergence of portable systems, such as laptop computers, cellular phones and PDAs. The increasing popularity of such systems encourages the development of more and more sophisticated devices. Designing a portable system requires to tackle about the problem of delivering high performance with a limited consumption of electric power. High performance is required to support complex applications (for instance multimedia) that are running on these portable systems. Low-power consumption is required to achieve acceptable autonomy in battery-powered systems, as well as to decrease battery weight. Stationary systems[1] are also concerned with power conservation, because of the cost and noise of cooling systems, the cost of electric power (especially for large systems) and stricter environmental impact regulations.

The battery capacity has improved very slowly while the demands of computation capacity have drastically increased over the same time. Better low-power circuit design techniques have helped to increase battery-lifetime, but benefits obtained by such techniques do not compensate all the needs. In addition, the pressure for fast time-to-market has become extremely high, and it is often unacceptable to completely redesign a system merely to reduce its power dissipation.

Electronic systems are generally designed to deliver peak performance, but in many cases peak performance levels are not needed for most of the operation time. Cellular phones and portable computers are two examples of systems with non-uniform workload. When the user is sending or receiving a call with a cellular phone (or is running an application on

[1] i.e. non portable systems.

a laptop computer), he wants to have the maximum performance. However, when the user is carrying the phone in his pocket (or is thinking to what to write during a text-editing session on a laptop computer), he does not need the full computational power of the system.

Power management for computer systems is traditionally focused on regulating the power consumption by switching the system in a low-power state, which does not allow to use the system. This state is a de-activating state, generally requiring a user action to re-activate the system. More recently, some researches focus on the development of power management techniques performed while programs are running.

Dynamic power management (DPM) is a design methodology that dynamically reconfigures an electronic system to provide the requested services and performance levels with a minimum number of active components or a minimum load of such components. Dynamic power management encompasses a set of techniques that achieve energy-efficient computation by selectively turning off (or reducing the performance of) system components when they are idle (or partially unexploited). A component is in an *idle* state if it has no request to serve; it can then be put in a *sleep* state to reduce its power consumption. When a request arrives, the component wakes up and switches into a *run* state in order to serve this request[2]. Moreover, a component can be completely shut down in order to not consume power. The component is thus in the *off* state.

The fundamental premise for the applicability of DPM is that systems (and their components) have to support a non uniform workload during their operation times. Such an assumption is valid for most systems, both when considered in isolation and when internetworked. A second assumption of DPM is that it is possible to predict, with a certain degree of confidence, the fluctuations of workload. Finally, a third assumption is that the workload observation and prediction should not consume significant energy.

We examine hereafter the three levels where dynamic power management (DPM) can be applied: first at component level, then at system level and finally at network level.

1.2 Dynamic power management at component level

We can see a system like a set of interacting components. The definition of a component is general and abstract. It may be a chip (such as the CPU) or a board (hard disk, memory, wireless interface, video display,...) but in the current context, it is a black box: no detailed knowledge of its internal structure is needed.

[2]Other names for these states are sometimes used in the literature, such as active, disable, on, work,....

A power manageable component (PMC) is characterized by multiple states of operation that span the power-performance trade-off. This allows to distinguish these components from those always operating at a given performance level and power consumption. The ideal case for a PMC is always to have a lot of states of operation in order to minimize the power by calibrating perfectly the performance needed by the requests served by the component. Nevertheless, a problem appears when the number of states increases because the hardware complexity and overhead become more pronounced. In fact, the PMC's cannot switch from a state to another one without a cost. The cost may be a performance lost, a delay or even a power transition cost. The relation that we often note is: low-power states (such as sleep) have lower performances and larger transition costs, with respect to states with higher power. The transition cost has an important impact and has to be taken into account in the power management models.

1.2.1 Power state model

We can thus define for each component a power/performance behavior. In other words, it can be defined for each PMC the power states it accepts (for instance run, off, sleep, and idle), the associated performances, the transition costs and the power consumptions when it switches from one power-state to another one. We can represent this information with a power state machine for example (see figure 1.1). To summarize, we can say that a PMC

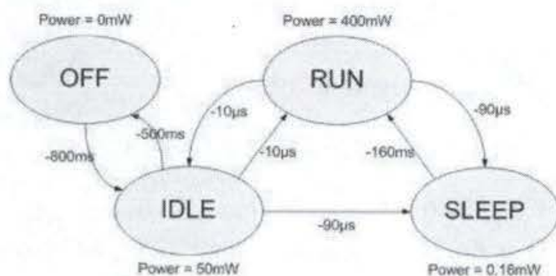

Figure 1.1: Example of power state machine.

is needed to develop some power management techniques and that a component cannot switch from one of its state to another one without a cost. This leads us to consider the power management model as a non-trivial optimization problem.

1.2.2 Under the black box

We have defined a component as a black box. It is time to see now how DPM is implemented inside the *box*. We examine in this section two of the most dynamic techniques implemented in components themselves: the *clock gating* technique and the *power off supply* technique. In reality, these techniques are generally implemented in the hardware circuit of the component. That is why they are also called physical mechanisms.

One of the most common DPM techniques at the component level is the clock gating. This addresses digital components that are clocked (CPU, display,...). Power can be saved by reducing the frequency of the component clock (or some component clocks, if the component uses several clocks), and at the limit by stopping the clock. Clock gating can be applied both during idleness periods or activity periods. During activity periods, slowing the clock decreases the performance and, for some components (like the CPU), extends the execution time to perform a task. In this case, power saving is generally weak. For this reason, power saving can be more important during idleness period. The main challenge is then efficient idleness detection. Moreover, to go back to normal system activity, clock gating requires a short transition time: the clock should be re-initialized in one or a few clock periods.

Even by stopping the clock, power dissipation is not completely eliminated. Power consumption of an idle component can be avoided by the technique of powering off the component. This radical solution requires controllable switches inserted in the electrical line supplying the component. A major disadvantage of this method is the recovery time, which is typically higher than in the case of clock gating because the component's operation must be re-initialized.

1.2.3 Internal controller and external controller

We have said above that the transition latency can have an important impact on the power management model. In reality, when it is possible to switch a component to a *sleep* state without compromising the performance (or with little performance degradation) because the transition between the *sleep* state and the *run* state is nearly instantaneous, an *internal controller* - internal to the component - can be implemented. This internal controller decides for example to decrease the frequency (clock gating) or shut down the component (supply shutdown) if there are no requests to serve. Internally managed components are also called self-managed components. The main drawback is the lack of observability of the overall system operation and of the need of tolerating little or no performance degradation, since no assumptions can be made on how demanding the component's environment will be.

Clock gating is often implemented in a component with an *internal controller* while supply shutdown will be rather implemented with an *external controller*. The difference between the two techniques resides in the transition time to switch the component into its normal state. Indeed, when power transitions take a long time or consume a lot of energy, it is needed to take into account the workload of the system and decide when it is worthwhile to switch to a low-power state. Otherwise, the performance could be disastrous. The example that we have in mind is the limit case of workload with no idle periods longer than the time required to enter and exit the *sleep* state. If we decide to shut down the component as soon as an idle period is detected and if the power consumption associated with state transitions is of the same order of that of the *run* state, this could reduce the performance without saving any power. Workload information is thus required for all *advanced* power management approaches. The goal is even to predict the workload to identify exactly the idle periods. Several approaches and models can be produced to capture the workload information. Diverse techniques based on predictive approach or stochastic control have already been developed. We can easily imagine that these models are very complex. It is thus the role of an *external controller* - external to the component - to control transitions based on the workload of the system.

Let's see now how an external controller can be implemented at the system level by a power manager.

1.3 Dynamic power management at system level

1.3.1 Structure of a system-level power manager

The power management idea is to profit from idle periods (e.g., when the component is not used) to put the component in its *sleep* state, or in one of its *sleep* states if multiple low-power states are available, without compromising too much the performance of the whole system. Regarding the system, the assumption made here is that it is not always entirely active; this means there are some periods during which some components are idle.

The activity of components (PMC and also non-PMC) is coordinated by a *system controller*, which is generally implemented in a software routine of the operating system. That is why the control of the system consumption is also implemented in software and especially in operating system as a module of the system controller.

We call *power manager* (PM) the system part (hardware or software) that performs DPM at system level. A power manager is composed of an *observer* monitoring the workload of

the system and its resources, a *controller* issuing commands for forcing state transitions in system resources, and at least one *policy*, which is a control algorithm for deciding when and how to force state transitions (based on information provided by the observer). The figure 1.2 depicts the general structure of a power manager.

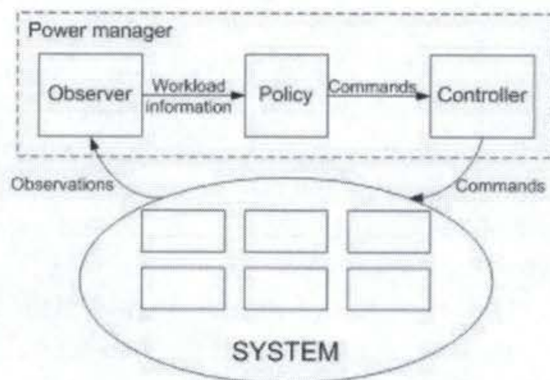

Figure 1.2: Structure of a system-level power manager.

There are some reasons for implementing the power manager at software level. Software power managers are easy to write and to reconfigure. Then, in most cases, the designer cannot or does not want to interfere with and modify the underlying hardware platform. Finally, as DPM implementations are still a novel art, the experimentation with software is easier than with hardware.

1.3.2 Power manager controller

The power manager must be able to control the state of each PMC. In order to develop a power manager as generic as possible, it should be interesting to dispose of an interface between the power manager and a PMC. This interface should be able to understand each command from the power manager and to effectively perform it. The way to control the state of a component is very hardware dependent. For this reason, the interface should be implemented in the component-closest software part, which is generally the *driver* of the component.

Industrial designs have been also proposed to encourage the standardization of interfaces. For instance, Intel, Microsoft and Toshiba propose a standard called *advanced configuration and power interface*, in short ACPI. ACPI is an OS-independent general specification that defines the interface between the operating system and an ACPI-compliant hardware. This interface can be used both for hardware configuration and power management. The

front-end of the ACPI is the *ACPI driver*, which is OS-specific. The OS kernel interacts with hardware through the ACPI driver, which maps each OS request to an ACPI command and each ACPI response/message to a signal/interrupt (see figure 1.3).

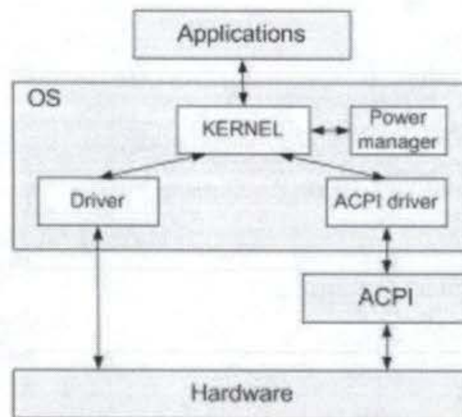

Figure 1.3: ACPI interface.

The ACPI allows the operating system to put the system in five possible *global power states*:

- *Working*: The system is ON and fully usable.
- *Sleeping*: The system appears to be OFF and the power consumption is reduced. The system returns to the working state in an amount of time inversely proportional to the power consumption.
- *Soft off*: The system appears to be OFF and power consumption is very low. A full OS reboot is needed to restore the working state.
- *Mechanical off*: The system is OFF, with no power consumption. It needs to be reconnected to the power supply to go back to the working state after a full reboot.
- *Legacy*: This state is entered when the system does not comply with ACPI.

Additionally, the ACPI specifications define power states for system components. There are two types of system components, *devices* and *processor*, for which four power states are defined in the Table 1.1:

	Device power states		Processor power states
D0	This state has the highest level of power consumption. The device is fully active.	*C0*	The processor is fully operational and executes instructions.
D1, *D2*	The details of these states are device dependent. A device in D1 is expected to save less power than in D2, but it preserves more context (hence, wake-up is faster).	*C1*, *C2*	The processor is not executing instructions. The processor in C1 is expected to save less power than in C2, but switching from C1 to C0 is performed in a negligible time.
D3	Power has been fully removed from the device. The device cannot be used in this state and has the longest restore time. The OS will re-initialize the device when powering back on.	*C3*	This state offers improved power savings with respect to C2. The hardware latency to resuming execution is larger than that in C2.

Table 1.1: Devices and processor power states

1.3.3 Power manager observer

A power manager needs information to predict future workloads. Two extreme approaches allow to collect this information. In the first approach, the power manager observes the requests soliciting the managed component and try to predict the future idleness length, and on this basis determines the best power state. It selects the power state without direct interaction with requesting application. The figure 1.4 depicts this approach.

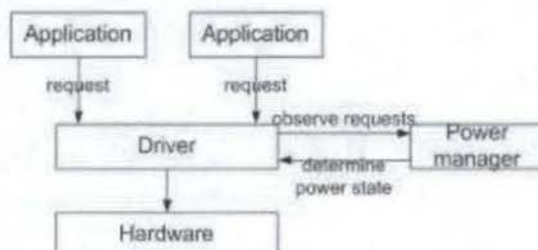

Figure 1.4: The power manager observes requests.

In reality, however, requests may be generated by multiple requesters. For example, requests for a network interface card may come from different programs (such as ftp, telnet or netscape). These programs work differently and have different performance requirements. Without information about requester programs, the power manager cannot precisely predict the exact moment when the component is not used anymore and thus wastes energy to

maintain performance unnecessarily or cause delays and performance waste to save power.

At the other extreme, requester programs directly control power management through an application programming interface (API). For example, programs can keep a component in a run state, wake up a component in the sleep state or know the current state of a component. The figure 1.5 describes this approach. The main disadvantage of this ap-

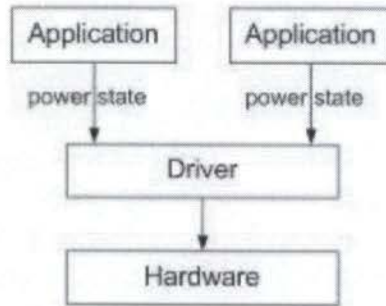

Figure 1.5: Applications control power states directly.

proach is that programs must include specific power management instructions. Moreover, programs do not know very well the components, especially the transition costs of each of them. They are not aware if changing state saves power.

Other solutions have been proposed, like in [2], which are generally situated between the two previous extreme approaches. Generally, in these other solutions, programs provide to the power manager an information about their needs of components. The power manager decides, according to the programs needs and its knowledge of the components transition costs, to put some components in sleep state if their are idle for a sufficient long period.

1.3.4 Power manager policies

A power management *policy* is an algorithm that selectively shuts down idle resources based on the observation of present and past workload and operating conditions. We survey here two different approaches to policy optimization, the *predictive techniques* and the *stochastic control.*

The rationale in all *predictive approaches* is to take DPM decisions based on predictions concerning the duration of idle periods. A generic predictive method observes the time-varying workload, and, based on this observation, computes a predicted duration T_{pred} of

the upcoming idle time. The power manager then decides the transition to the sleep state if $T_{pred} \geq T_{BE}$, where T_{BE} is the *break-even time*, the minimum idle time amortizing the state transition cost.

Good predictive approaches should minimize the time for which the system wastes power because it does not immediately detect the beginning of an idle period. They also should minimize the mis-predictions $T_{pred} \neq T_{idle}$, where T_{idle} is the actual duration of an idle period. Predicting a too small T_{pred} ($T_{pred} < T_{idle}$) wastes power, while a too long T_{pred} ($T_{pred} > T_{idle}$) decreases performance.

The most common predictive PM policy is the *fixed timeout*. The policy can be summarized as follows: when an idle period begins, a timer is started with a certain duration T_{T0}. If after T_{T0} the system is still idle, then the PM forces the transition to the sleep state. The system remains in sleep state until it receives a request from the environment that signals the end of the idle period. The fundamental assumption in the fixed timeout policy is that the probability of T_{idle} being longer than $T_{BE} + T_{T0}$, given that $T_{idle} > T_{T0}$, is close to one. Hence, this policy assumes that if the idleness duration is longer than the timeout duration ($T_{idle} > T_{T0}$), the idleness duration will continue a sufficiently long time to allow the system to save power by changing the power state ($T_{idle} > T_{T0} + T_{BE}$). Timeouts are "implicitly" predictive even if they never generate an actual T_{pred}, in the sense that they predict a long idle time if the system has been idle for a while. The critical design decision is obviously the choice of the timeout value T_{T0}.

Timeouts have three main limitations: fixed timeouts may be ineffective when the workload is non-constant. Moreover, power is wasted while waiting for the expiration of the timeout. Finally, performance penalty is always paid upon wake-up. *Adaptive timeouts* have been developed to improve effectiveness, by dynamically reducing/increasing the timeout value when idleness duration increases/decreases. The figure 1.6 illustrates the timeout-based policies.

Another predictive policy consists of shutting down or putting in sleep state the system or a component as soon as it becomes idle, if the policy predicts $T_{pred} > T_{BE}$. A prediction of idle time duration is made available as soon as the idle period begins. Predictions are made based on past idle and activity period durations. The system wakes up either only upon arrival of a request from the environment, or at the end of the predicted idle period. The *predictive shutdown policy* is illustrated in the figure 1.7.

Predictive approaches have some limitations: first, they do not precisely take into account the workload variations. Then, predictive algorithms are based on a two-state system

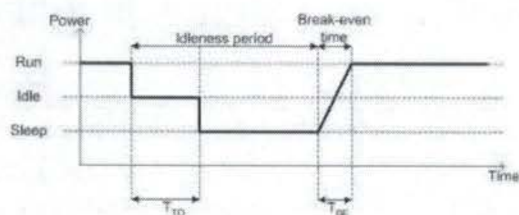

Figure 1.6: Timeout-based policy.

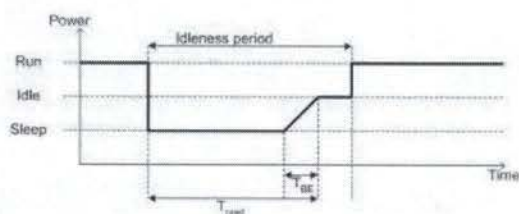

Figure 1.7: Predictive shutdown policy.

model, while real-life systems have multiple power states. Policy involves not only the choice of when to perform state transitions, but also the choice of which transition should be performed.

The *stochastic control* approach considers the workload as a *Markov chain* of R states. A Markov chain is a probability function of the evolution of the system state, according to the state at present time. A Markov chain defines probabilities for the system to enter into a state at a moment $t + 1$, according to the system state at the moment t. Thus, the workload can be represented by a two-state Markov chain, where the two states are: $R0$ when no request is issued by the environment and $R1$ when a request is issued.

In the same way, power states are also represented by a Markov chain of S states. The transitions are probabilistic, and probabilities are controlled by commands issued by the power manager. A power manager is then a function $R \times S \rightarrow A$ where A is a command to control the future state of the system. Such function is an abstract representation of a decision process: the PM observes the power state S of the system and the state R representing the workload, takes a decision, and issues a command A to select the next state of the system.

1.4 Dynamic power management at network level

In many cases, the systems are not isolated and interact between them. We call network a set of communicating systems. We can thus imagine a power manager that tries to minimize the global consumption of the whole network, rather than the individual consumption of each system. The power manager is so implemented as a *distributed* algorithm. This distributed algorithm takes autonomous decisions for each system in the network based either on local information, or on incomplete global network status data.

Currently, the DPM in such networked systems is not really still well developed.

1.5 Examples of dynamic power management techniques

This section describes some examples where dynamic power management techniques are used. Each subsection is focused on one of the main components of a portable computing system. Thus, we present here low-power techniques concerning respectively the battery, the processor, the memory, the hard disk and the network card.

1.5.1 Battery-driven dynamic power management

Battery lifetime extension is a primary design objective for portable systems. Traditionally, battery life-time has been prolonged mainly by reducing average power consumption of system components. An analysis of battery discharge characteristics has allowed to develop new opportunities for life-time extension. [3] and [4] propose a class of policies, whose decision rules controlling the system operation state are based on the observation of both system workload and battery output voltage.

The proposed policies are based on three battery physical properties. First, the effective voltage of a battery decreases as the state of charge decreases. As a matter of fact, a battery is considered exhausted when its output voltage falls below a given voltage threshold (such as 80% of the nominal voltage). The second property is that the actual usable capacity (Number of Amps × Number of hours) depends on the discharge current. More clearly, in a theoretical battery, requiring more power (for example twice more power) reduces battery lifetime proportionally (for example a reduction of a factor two). In a real battery, the lifetime reduction is more accentuated. The reason is that at higher discharge current, a battery is less efficient to convert its chemically stored energy into available electrical energy. The third property says that a battery can recover some of its deliver-

able charge if discharge periods are interleaved with rest periods (i.e. periods in which no current is drawn).

The simplest policy is threshold-based. It aims at maximizing battery lifetime by lowering the performance when the battery is almost discharged. If the battery is fully charged, the system is kept in a normal-performance state. When the battery output voltage falls below a threshold, the system is forced into a low-performance and low-power state until the battery is fully discharged. The rationale for this policy is to provide acceptable degradation of system performance as the battery discharges.

Some modern portable appliances can accommodate two batteries in the same case. The batteries are normally used sequentially: The second battery starts supplying the current only when the first battery is totally discharged. As a battery can recover some of its deliverable charge if it is let at the rest, a dual-battery system can alternatively use each battery to draw its energy. In this way, the battery temporarily disconnected from the load can recover, while the other one powers the system. An efficient policy to manage the batteries alternation consists of defining three regions of operation. In the first region, the switching between the two batteries has constant frequency, and the system works in a normal-performance state. The second region is entered when the output voltage of one battery first reaches a threshold. The system still works in normal state, but switching between batteries is voltage-controlled. When the output voltage of the loaded battery reaches the threshold, it is disconnected from the load (to give it some recovery time). The second region is exited when the output voltage of the battery temporarily disconnected from the load does not reach a level close enough to the threshold during the recovery time. In the third region, the fixed frequency-switching scheme is restored, and the system is transitioned into a low-performance state until both batteries are fully discharged.

The battery-driven policies proposed here, applied in isolation or altogether with another technique(s), allow a very important lifetime extension in some systems.

1.5.2 Power management for the processor: Dynamic voltage scaling

DPM aims at reducing energy consumption at the system level by selectively placing components into low-power states during idle periods. The power manager can completely turn off the component (power off supply) or disable some clocks to do that (clock gating). In the microprocessors, other techniques have been developed to decrease significantly the energy consumption. If clock gating is an efficient technique to reduce the energy in clocked components, *dynamic voltage scaling* (DVS) is often an additional technique that can more

reduce the power consumption. DVS can be used simultaneously to the clock gating.

We can compare in fact DVS to a kind of DPM at the component level except that DVS is targeting specifically the processor. DVS is close to clock gating because both act on the clock frequency except that DVS does not turn off the clock but changes the frequency of the processor (the speed of the processor) and voltage at run-time, depending on the needs of the running application. Thus DVS do not thus detect idleness period but rather estimate the needs of the running application. When peak performance is needed (e.g. peak computational loads), the processor operates at its normal voltage and frequency (which is also its maximum frequency). During the rest of the time, when the load is lower, the operating frequency is reduced to meet the computational requirements (performance). In fact, once again, it is assumed implicitly that the *workload* is not constant and that the average computational throughput is often much lower than the peak computational capacity needed for adequate performance.

DVS algorithms modify the frequency according to the needs of the running application and go beyond that because they scale the operating voltage of the processor along with the frequency. In fact, the vast majority of microprocessors today has a voltage-dependent maximum operating frequency, so that when they are used at a reduced frequency, the processor can operate at a lower supply voltage. In other words, the maximum operating frequency increases (within certain limits) with the increased operating voltage; when the processor is run slower, a reduced operating voltage suffices. The interest of reducing the operating voltage along the frequency is obviously to save energy. There is a second characteristic also shared by the vast majority of microprocessors today, allowing saving energy: the energy consumed by the processor per clock cycle scales quadratically with the operating voltage, so that even a small change in voltage can have a significant impact on energy consumption. Moreover, important energy savings can be got because high performance is needed only for a small fraction of the time, while for the rest of the time, a low-performance and low-power processor would suffice. By dynamically scaling both voltage and frequency of the processor based on computation load, DVS can provide the performance to meet peak computational demands, while on average, providing the reduced power consumption benefits typically available on low-power performance processors.

To decrease the frequency along with the needs of the application, the processor must operate over a range of frequencies. For that, it must be specifically hardware-designed to support dynamic clock frequency adjustment. The number of frequencies is thus predefined by the design of the microprocessor. As we have already pointed it out in previous sections for the different *states* in the DPM model, there is also some overhead here when switching from a frequency to another one. This overhead is also called transition time, like in the

DPM model. Nevertheless, even if it depends a lot from the hardware, the transition time overhead is often very short.

If DVS relies on a specific hardware design, we can say that it relies also on software at the system level, or even at the task level. DVS algorithms have in fact two main functions: the first one identifies the needs of the application and the second one adjusts effectively the CPU frequency and voltage. To accomplish the first "mission", the execution time of the application has to be predicted (by analyzing the workload). It is important to note that this execution time can be increased (performance decreases) if the frequency is decreased, but the main challenge is to respect the needs of the running application or, in other words, to keep an *adequate* performance. In fact, "respect the needs" or "keep an adequate performance" can be formulated like the respect of the deadline of a task in a real-time environment where tasks must be completed before a deadline, or like keeping a frame delay constant for a MPEG or MP3 streaming application. There are in fact a lot of models implemented in software (at the system level or at the task level) that try to get accurate information on the *workload* in order to adapt the frequency of the processor on the basis of this information while keeping an adequate performance . We can also define for DVS the notion of *policy*, which is an algorithm that selectively adjusts the clock frequency and voltage of the processor based on the observation of the *workload*.

In fact, we can easily integrate the DVS technique into the DPM model presented in previous sections at the system level. We can consider that the processor *remains* in a *run* state when the DVS algorithm is applied. It allows reducing the energy consumption of the processor when the processor is in a *run* state. Let us remind that, when in previous sections, we talk about the *run* state, the component consumes a lot of energy because it is fully powered on. In reality, we can see here the *run* state like a set of sub-states.

If the model describing the *run* state is the same as the one characterizing the other states, the transformation, from the "original" DPM *run* state into multiple "new" *run* states taking into account the different performance and power levels of DVS, is completely compatible with the rest of the model and so, the power management *policy* that we develop can make decisions for both dynamic voltage setting and the transition into the low-power states. For example, in the case of a MPEG streaming application where keeping an adequate performance consists in maintaining the frame delay constant, the power manager can check if the rate of incoming or decoding frames has changed, and then adjusts the CPU frequency and voltage accordingly. Once the decoding is completed, the system enters in an *idle* state. At this point, the power manager observes the time spent in the idle state, and depending on the policy, it decides when transiting into one of the *sleep* states. When a request arrives for video decoding (after receiving a new frame on

the network interface), the power manager switches the system back into the active state and starts the decoding process. That is actually the ratio between the rate of incoming frames and the rate of decoding frames that determines the frequency and the voltage levels of the processor.

It is to be noted that a dilemma appears and can be stated as follow: lowering the threshold voltage to reduce *run* power can increase *sleep* power. In fact, increasing the execution time of applications by reducing the frequency causes the system to be more often in the *run* state than in the *sleep* state. In other words, if a system is frequently *idle*, it could be more interesting to save power by putting the system in the *sleep* state instead of decreasing the frequency and the voltage of the processor.

1.5.3 Selective instruction compression for memory energy reduction

We have already seen that power management can be applied to several embedded components. It is time now to see how we can save power in the memory management.

The first developed techniques are bus encoding techniques and memory organization techniques. Both are based on a reduced switching activity on the processor-memory bus. In fact, the first technique changes the format of the information transmitted on the processor-memory bus. Thus, it reduces the switching activity on the bus and so, in the same way, reduces the power consumption. The second techniques change the way information is stored in memory so that the address streams between the processor and the memory cause a low-transition activity on the bus. The processor-memory interface is a major contributor in the power consumption and so, these techniques can be efficient.

Other techniques have been additionally developed through instruction memory bandwidth. These techniques use a set of special instructions which are smaller in size (e.g., contain less bits than normal instructions) and hence, achieve to reduce the bandwidth needed to run the program. Either these special instructions are another set of instructions supported by the processor but in this case it often requires additional software tools to allow users to generate these special machine instructions from the task level, or these special instructions are simply a subset of the original instructions supported by the processor. In this latter case, if we consider only a subset of the original instructions, we decrease the number of potentially instructions used and so, we do not need the entirely bit-width associated to all original instructions. Thus this subset of instructions can be replaced by binary patterns of limited width. For example, if we identify in an application the utilization of 512 distinct instructions among the 8192 instructions offered by the processor, we

can then only use 9 bits to encode the 512 instructions (log_2 512 = 9) instead of 13 bits needed to encode 8192 distinct instructions (log_2 8192 = 13). The 9-bit instructions take less place in memory than 13-bit instructions. It thus reduces the memory bandwidth usage and in consequence the total energy because the unused memory banks can be disabled.

We suppose in fact that the processor can disable the memory banks that are not currently used in order to save energy. It is an hardware requirement to save energy. For example, if the memory consists of four 8-bit banks and that we have succeeded in reducing the size of the instructions from 32 to 8 bits, it requires only one memory bank instead of 4. The 3 unused banks can be disabled to save energy. Moreover, if we assume that the memory access is 8-bit wide, the number of fetching between the processor and the memory passes from 4 to 1: the total bus utilization is thus also reduced, which improves performance (i.e., the program utilization time decreases).

We can simply implement a table whose the role is to match the original instructions of the subset with the compressed instructions. In reality, the program is stored in memory in compressed format, i.e., each instruction is replaced with a $[log_2 N]$-bit binary pattern which is in one-to-one correspondence with the original instruction. Every time an instruction is fetched from the memory, it is first decompressed (i.e, the original format is restored) by means of the *instruction decompression table* and then passed to the processor's decoding logic. The advantage of the table is that we do not modify the architecture of the processor. The figure 1.8(b) depicts the solution and especially the *instruction decompression table* (IDT). The main drawback of this technique is the energy saving depends on

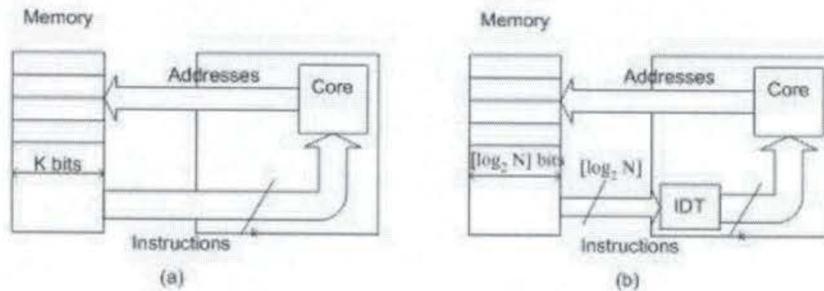

Figure 1.8: Processor-memory normal architecture (a) and power-saving architecture (b)

the ratio between the number of different instructions that forms the subset (compressed instructions) and the total number of machine instructions. The more distinct instructions are used, the more there are instructions in the subset and therefore, fewer energy is saved. Indeed, the bit-width of these instructions may become similar to the bit-width of

the original instructions, thus making negligible the reduction in memory bandwidth. The second drawback is the implementation of the table because this can be very complicated to encode and decode instructions especially if the size of the compressed instructions is not compatible with the byte-addressable memory. Indeed, usually memories can be only accessed for example by a multiple of 8 bits. If we refer to the example in the previous paragraph where the application uses only 512 instructions among the 8192 instructions supported by the processor, the 9-bit instruction is not compatible with the 8-bit access scheme of the memory. In fact, two bytes are needed to store each compressed instruction and so, it results in a waste of space and consequently in a waste of energy. Moreover, it requires two operations to store this 9-bit instruction in memory and so, it does not decrease the program utilization time.

One solution consists of taking a subset of fixed cardinality. That does not depend anymore on the number of distinct instructions used by each application running but rather on the probabilities of the instructions to be used. In fact, an assumption is indirectly made: the number of machine instructions used by most software programs, although limited with respect to the total number of instructions supported by the processor, has a highly non-uniform statistical distribution. In other words, some instructions are usually much more used than others. We can hence consider a subset of original instructions that are executed more often whatever the applications; less probable instructions are left unchanged and stored as they are in memory. As there are as well non-compressed (few) instructions as compressed (many) instructions in memory, that requires a controller which properly handles instruction fetching. The figure 1.9 depicts the second power-saving architecture in the case where the compressed instruction has 8-bit width.

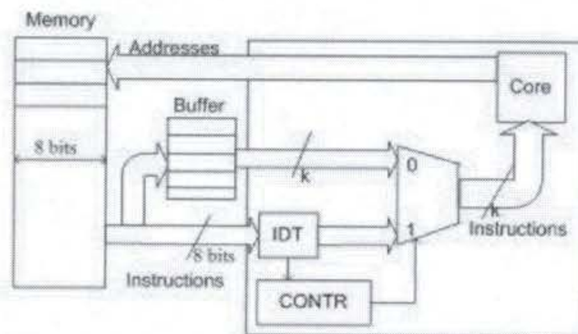

Figure 1.9: Processor-Memory second power-saving architecture

The percentage of energy that can be saved depends mainly on the architecture and espe-

cially on the way in which the memory is organized and accessed. The technique presented here is indeed a new way of managing and accessing the memory since there are different bit-width instructions. We have not presented all the implementation and changes that this technique required in the memory management. However, it is directly implemented in hardware and so, it is application-independent. It is important to say also that, like in all other power management techniques, there are some *costs* to take into account in this power management technique. In fact, the *instruction decompression table* (IDT) consumes a certain time to decode and to encode instructions, and therefore it consumes power. This cost in power and performance of the decompression block is not still well-known and is currently under investigation. In addition to that, to distinguish compressed and original instructions, it is needed to put sometimes some bits in memory and this causes also some overheads in the number of memory accesses, as well as a waste of space. Thus, the costs in term of performance or energy exist also for these techniques aiming at reducing the consumption of energy by the memories. Like in all other power management techniques, there is always a trade-off between performance and power.

1.5.4 Power management for hard disks

The hard disk drive is one of the major power consuming subsystems in a computer, since it can consume more than one fifth of the total power used by the computer. The main way to reduce power consumption of hard disks consists of stopping plates spinning during the periods when no disk requests are made. However, this approach encounters three problems: first, when a request occurs while plates are not spinning, it can not be performed before the plates have taken back a sufficient speed. This delay strongly decreases the performance of the system. Moreover, accelerating the plates generates extra power dissipation. Finally, too frequent on/off cycles tend to accelerate the degradation of the hardware systems, causing a problem of reliability. Now a desirable power management scheme should save energy while providing high performance and low failure rates.

Disks accesses generally occur in burst: the activity of a disk is often characterized by sets of close requests, separated by long idle periods. It can be made use of these idle periods to reduce power consumption. In [5], a method is proposed to do that, on the basis of the concept of *sessions*, which are time intervals when requests frequently occur. A session starts with a disk access, and is separated from the previous one by a long period of inactivity. Requests close in time are regrouped in the same session. A threshold τ is used to separate sessions. If no requests occurs during τ seconds after a request occurrence, the current session ends. The figure 1.10 depicts an example of sessions with different τ.

A first observation is that, during inter-session periods, there are no disk activities, and

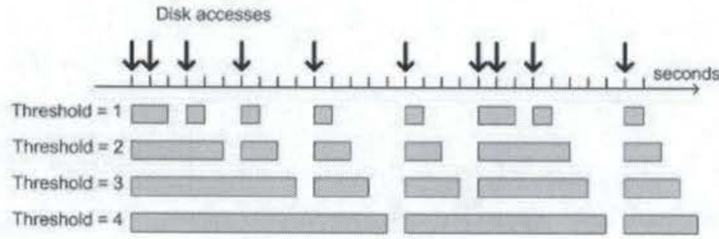

Figure 1.10: Example of sessions for different thresholds.

then the system can spin down the plates to save power. Moreover, smaller τ may separate adjacent requests into two sessions, while a larger value can regroup them into one. Hence, a large τ allows to reduce the number of inter-sessions, while a small τ increases their occurrences. Thus, a variable τ can be used to improve performance according to the access frequency: by increasing τ during access bursts, one increases the waiting time before the session ends, and thus reduces the risk to spin down the plates before the end of the burst. In the same way, τ can be decreased when the disk activity is characterized by long idleness periods separated by short bursts.

1.5.5 Transport protocol optimization for energy efficient wireless embedded system

For wireless portable systems, the power consumption in the network interface plays an important role for the battery lifetime. Although there are already some DPM techniques allowing to put the network interface card in several possible states (receive state, transmit state, idle state, sleep state,...), the power consumption can be optimized by developing techniques at an upper level, and in particular at the network protocols level. Indeed, the higher sub-layers of the protocol (IP, TCP, UDP,...) allow to estimate the long-term workload, while the Medium Access Control (MAC) sub-layer allows to know the channel state. This optimization is especially needed since transition costs are very important. Hence, the challenge in developing an effective power management strategy is to identify the times during which the sub-system can be placed in a particular power mode, by observing the workload.

The IEEE 802.11 protocol describes how the physical and MAC sub-layers must be implemented. IEEE 802.11 defines two modes: a normal mode and a low-power mode. When the network card is in low-power mode, it is shut down and the received data packets are lost. Fortunately, these packets are kept by the access point. IEEE 802.11 specifies that the network card must be periodically awaken to check if the access point keeps data

packets. The main drawback of this technique is that the card wakes up to check the access point, even if no packets are waiting. It generates thus a power consumption overhead.

A transport protocol optimization of wireless LAN interfaces has been proposed in [6]. The proposed technique is based on the TCP buffer. It consists of monitoring the TCP receive buffer occupancy and exploiting in-built signaling mechanisms of TCP, such as *window advertisement*[3] and *silly window syndrome*[4] (SWS) avoidance, to identify scenarios where low power modes can be triggered with minimal overheads. Two extreme cases can occur: TCP receive buffer full and TCP receive buffer empty.

TCP disposes of a mechanism preventing the receive buffer from overflowing. When the receive buffer is full, the receiver sends a *zero window*[5] to the sender. This message signals to the sender that it has to stop sending packets. As the application reads data out of the receive buffer, the window becomes non-zero, but it is not immediately advertised to avoid the silly window syndrome. In fact, the receiver has to wait until its window has considerably increased, to prevent the inefficient exchange of small amounts of data across the connection (instead of full-sized segments). Hence, in order to avoid the silly window syndrome, the receiver waits before requesting to the sender to resend data packets. The receiver can take advantage of this waiting time to save power by switching off the network card. Indeed, the receiver knows that it will not receive data packet during this period. Generally, this period can last several seconds before the receive buffer is sufficiently empty to allowing the sender to resend data packets.

Another idleness condition that can be exploited at the transport layer occurs when the buffer is empty. However, power management actions have to be taken carefully when the buffer is empty because, even if no packets are presently stored, they might be in transit over the network. Ensuring that incoming packets are not dropped can be performed by a predictive approach. Thus, the traffic flow can be predicted considering the status of the outstanding TCP sockets. If no socket is actively used by the receiver, the network card is considered as idle and can be shut down. When the buffer of the receiver is empty, a timer is started. If the timer expires, the card is shut down until a read/write operation

[3]To avoid overflowing the buffer, TCP sets a Window Size field in each packet it transmits. This field contains the amount of data that may be transmitted into the buffer. If this number falls to zero, the remote TCP can send no more data. It must wait until buffer space becomes available and it receives a packet announcing a non-zero window size.

[4]The silly window syndrome occurs when the receiver authorizes too early the sender to continue the transmission. "Too early" means that the receive buffer is not sufficiently empty yet. As a consequence, the receiver signals a "small window", so that the sender will send small data packets in burst, a situation to be avoided.

[5]Message signaling to the sender that the receive buffer is full.

on the buffer occurs. At this moment, the card is switched on. If a read/write operation on the buffer occurs before the timer expiration, the timer is cancelled and then restarted.

Chapter 2

System description

This chapter describes the system Assabet used for this work. The first section presents the PDA concerned by the power management. The second section explains what is and how is organized the frame buffer memory. The third section gives more details about the displaying of pixels, and the last section describes the LCD controller registers.

2.1 Overview of the Assabet

LCD power management has been implemented on a PDA plateform, the Assabet, supplied by HP. The Assabet disposes of a StrongARM 1110 Microprocessor [7], which can operate with an internal clock frequency up to 221 MHz. The Assabet has 8 MBytes FLASH and 32 MBytes DRAM. The screen used for the experimentation is a Sharp 3.9" TFT LCD touch screen which supports either 8-bit or 16-bit color at 320 × 240 pixels.

The operating system is the version 2.4.6-rmk1-np1 of the Linux kernel.

2.2 Frame buffer and palette memory organization

The *pixel* is the smallest monochromatic element of an image. An image can be seen as a matrix of pixels, so that the image size is defined by the horizontal and vertical numbers of pixels. In memory, a pixel is simply represented by a number, which is generally coded with 4, 8, 12 or 16 bits. A set of pixels corresponding to an entire image on the screen is called a *frame*.

The *frame buffer* is a memory area used to contain the pixels to display on the screen. It is able to contain enough pixels to fill the entire screen one or more times. The frame buffer provides an abstraction to the graphic hardware: it allows the software (applications or the

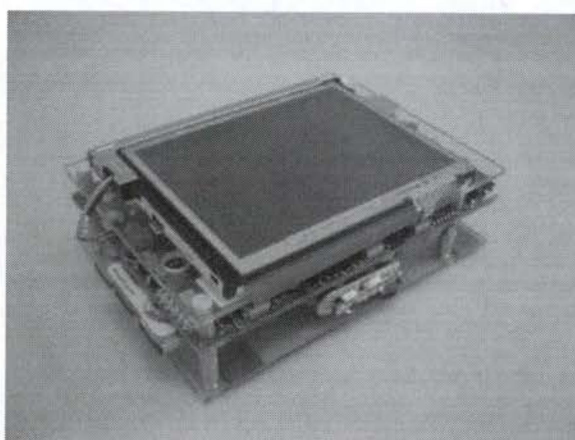

Figure 2.1: The Assabet.

operating system) to display images on the screen. Indeed, the software has just to write the image's pixels into the frame buffer and they will be displayed on the screen. Thus, the software does not need to know anything about the low-level details. In the case of the Assabet, the frame buffer is located in DRAM. There are several ways to organize the pixels inside the frame buffer. We can distinguish two main approaches.

In the first approach, a frame is composed of two parts: the *palette* and the *pixel data*. At the lowest addresses of the frame buffer is stored the palette, also called the *color map*, which contains the possible colors for a pixel of the frame. Each possible color is represented by a *palette entry*. The palette generally contains 16 or 256 palette entries. The first palette entry (palette entry 0) contains an extra field, called PBS for *Pixel Bit Size*, allowing to select the number of bits per pixel data in the frame.

Behind the palette are stored the *pixel data*, which indicate the color of each pixel of the frame. The pixel data are used as pointers to index into the palette. A pixel data must be able to select each entry of the palette. It is thus coded with 4 bits for a 16 entries palette and with 8 bits for a 256 entries palette. In monochrome mode, the palette is composed of 16 palette entries. Each palette entry is 16 bits long, but only four of these bits are used to define the gray-scale level. In color mode, the palette is composed of 16 or 256 palette entries, each of them corresponding to a possible color. A palette entry is encoded with 16 bits and defines a color as a combination of three basic colors, red, green and blue, whose intensities vary. The intensity of a basic color is coded on 4 bits,

so that 15 intensities[1] are possible for each basic color. Hence, it is possible to represent $15^3 = 3375$ different colors. However, since the palette only stores 16 or 256 entries, an entire frame can only contain 16 or 256 different colors. The figure 2.2 depicts the frame buffer organization in this approach, both for color and monochrome modes.

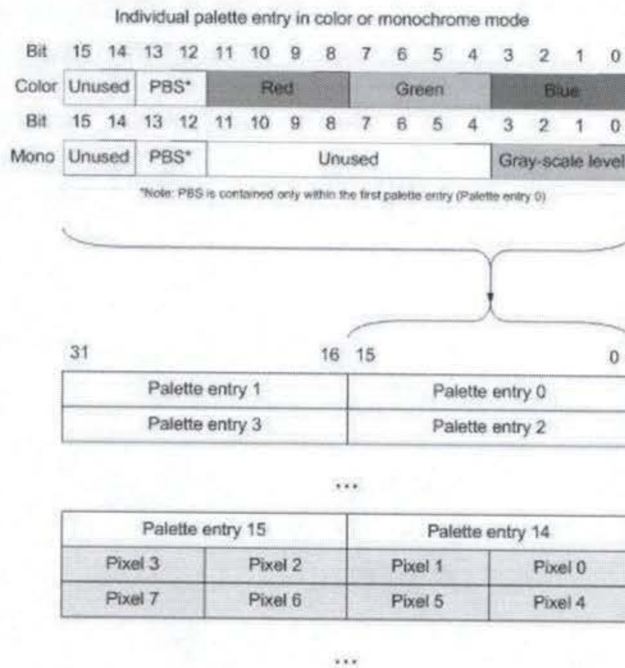

Figure 2.2: Frame buffer organization with a palette.

In the second approach, no palette is used. The color of a pixel is simply represented by a number coded with 8, 12 or 16 bits. The figure 2.3 depicts the frame buffer organization in this approach.

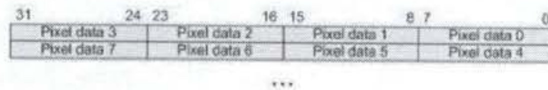

Figure 2.3: Frame buffer organization without palette.

[1] 15 and not 16 color intensities because the two sets of bits '1110' and '1111' correspond to the same maximal intensity

2.3 Refreshing the screen

A screen receives and draws an image pixel by pixel. Pixels are successively displayed on the screen from left to right to draw a line, and pixel lines are successively drawn from the top to the bottom of the screen. Pixels are transmitted by means of data pins to the screen at a frequency determined by a regular signal called *pixel clock*. This signal is emitted on a specific pin simply called *pixel clock pin*. At each tick on the pixel clock pin, a pixel is read from the data pins and displayed on the screen. The display also needs to know when a pixels line and a frame ends. Two other signals are thus sent on two pins: the *line clock* which occurs at the end of the transmission of a pixel line, and the *frame clock* which notifies the end of a frame. At each line/frame clock pulse, a new line/frame begins. The line and frame clock periods are multiple of the pixel clock period.

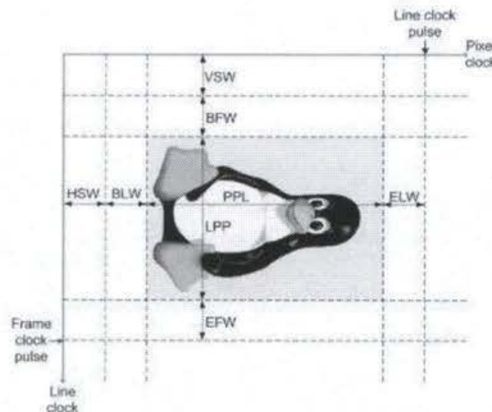

Figure 2.4: The display timings.

The figure 2.4 presents all the display timings that must be taken into account in the line and frame clock period calculation. PPL (Pixel Per Line) is the number of pixels in each line, i.e. the horizontal resolution of the screen. Since one pixel is displayed each pixel clock pulse, it takes PPL pixel clock pulses to display a line. BLW (Beginning-of-Line pixel clock Wait count) and ELW (End-of-Line pixel clock Wait count) specify the number of wait pixel clock pulses to insert at the beginning and at the end of each line. HSW (Horizontal Sync pulse Width) defines the wait pixel clock pulses between two line clock pulses, in addition to BLW and ELW. Hence, the line clock period is the sum of BLW, PPL, ELW and HSW. Likewise, LPP (Lines Per Panel) specifies the vertical resolution of the screen. BFW (Beginning-of-Frame line clock Wait count) and EFW (End-of-Frame line clock Wait count) specify the number of wait line clock pulses to insert at the beginning and at the

end of each frame. Finally, VSW (Vertical Sync pulse Width) defines the wait line clock pulses between two frame clock pulses, in addition to BFW and EFW. The frame clock period is hence the sum of BFW, LPP, EFW and VSW.

From these values, it is possible to calculate the *refresh rate* of the screen, by means of the following equations:

$$Pixel\ clock\ pulses\ per\ line = BLW + ELW + HSW + PPL$$

$$Lines\ per\ frame = BFW + EFW + VSW + LPP$$

$$Pixel\ clock\ pulses\ per\ frame = Pixel\ clock\ pulses\ per\ line \times Lines\ per\ frame$$

$$Refresh\ rate = \frac{Pixel\ clock\ frequency}{Pixel\ clock\ pulses\ per\ frame}$$

Once the pixel clock frequency is defined, the LCD controller automatically refreshes the screen by repeatedly sending pixel to the screen. It is to be noted that refreshing the screen is not performed by the core processor, which is free to perform other operations.

2.4 The LCD controller

A complex device, like a LCD display, needs a device controller to drive it. The *LCD controller* has three main roles: first, it has to transmit data sent by the processor to the display. Secondly, it has to interpret commands it receives and force the display to perform the corresponding changes. Finally, it has to receive electric signals, called *interrupts*, from the device and transmit them to the processor. Rather than describing the architecture of the LCD controller, it is more useful to present its interface, which is composed of registers.

2.4.1 The LCD Controller registers

In the Assabet, most of chips (controllers, caches,...) are integrated in the processor, in order to minimize power consumption by reducing distances. The processor disposes of its own address space, which is subdivided into several areas. Each of these areas is associated with a chip of the system and is subdivided into different memory registers, called *I/O registers* or *I/O ports*. To act on a chip, such as LCD controller, the processor has to write values in one or several I/O registers associated to the chip. Likewise, the processor can be notified about the device state by reading an I/O register. An I/O register can be decomposed in one or more sets of bits, called *fields*. Each of these fields commands a

function of the chip or provides information about the chip's status.

I/O registers are the basic way for drivers to communicate with the different external devices. There are four main types of I/O registers: control registers, status registers, input and output registers. The *control registers* allow to send commands to the device. A command is given by writing bits inside some fields of a control register. The *status registers* allow to be notified about the device state. For that, the processor simply reads the content of a status register. The driver can also fetch data from the device by reading bytes from an *input register* and push data to the device by writing bytes in an *output register*. In order to limit their number, some registers are often used for different purposes. For instance, some bits of a register describe the device state, while others specify the command to be issued to the device. Similarly, the same I/O register may be used as an input register or an output register.

The LCD controller of the Assabet includes four control registers ($LCCR0$, $LCCR1$, $LCCR2$ and $LCCR3$), one status register ($LCSR$) and four input/output DMA registers ($DBAR1$, $DBAR2$, $DCAR1$ and $DCAR2$). An exhaustive description of these registers can be found in Appendix A. We simply present here an overview of these registers and the main fields that are used in the next chapters.

2.4.2 The LCD Controller Control Registers (LCCR)

The first LCD controller control register, called $LCCR0$, contains ten fields mainly concerned with display modes and interrupt requests. In a power management context, the most interesting field in $LCCR0$ is the field LEN, since it allows to disable the LCD controller. All the other registers must be initialized before enabling the LCD controller. If LEN is modified when the LCD controller is active, the current frame will be completely transmitted before disabling the controller.

The second LCD controller control register, called $LCCR1$, regroups four fields used to control the number of pixels or pixel clock per line.

The third control register, $LCCR2$, regroups four fields that are used to control the number of pixels line or line clock per frame.

Finally, the LCD Controller Control Register 3, or $LCCR3$, contains seven fields allowing to control various functions within the LCD controller. For power management implementation, the most interesting field is PCD, which allows to select the frequency of the pixel clock. PCD allows to generate a frequency situated between $CCLK/514$ and $CCLK/6$,

where $CCLK$ is the CPU clock frequency. From the PCD value, it is possible to calculate the pixel clock frequency by using the following formula:

$$Pixel\ clock\ frequency = \frac{CCLK}{2 * (PCD + 2)}$$

2.4.3 The LCD Controller Status Register ($LCSR$)

This register contains 32 fields used to signal interrupts. These interrupts can be generated by events such as the disabling of the LCD controller or a DMA transfer. An interrupt is lifted when its corresponding bit is re-initialized. Thus, this register contains two kinds of bits: the *flags*, that are set and cancelled by the hardware, and the *status bits*, that are set by the hardware and cancelled by the software.

2.4.4 The LCD Controller DMA Registers ($DBAR$ and $DCAR$)

The LCD controller disposes of two DMA channels to transmit frame buffer data to the LCD controller. Each of these DMA channels needs a *base address pointer*, stored in the $DBAR1$ or $DBAR2$ register, and a *current address pointer*, stored in the $DCAR1$ or $DCAR2$ register.

When the LCD display is switched on, both DBAR and DCAR contain the base address. The LCD controller requires a DMA transfer, and the DMA reads four words from the frame buffer memory by using the current address pointer DCAR. These words are sent to the controller and DCAR is incremented of four. Each time the entry buffer of the controller contains four empty places, a DMA request is sent and four new words are loaded. When the current address pointer reaches the end of the frame buffer, the base address value in DBAR is copied in DCAR and the transfer continues from the start of the frame buffer.

Chapter 3

The Linux operating system

3.1 Introduction

The Linux operating system, in short Linux, is a member of the Unix operating systems family. A first version of Linux was developed in 1991 by a Finnish student, Linus Torvalds, by referring to Minix, a version of the Unix operating system developed by Andrew Tanenbaum. This first Linux version was only available on IBM/PC compatible computers using an Intel 80386 microprocessor, but was adapted afterwards with the help of other developers, in order to be supported by various architectures, such as Alpha, SPARC, ARM or MIPS. Linux was also upgraded with many additional functionalities and finally converged to a stable version released in 1993. Year after year, new enhanced versions were released, taking into account hardware evolutions and new peripherals. All those evolutions resulted from the contribution of many developers around the world, their activities still being coordinated by Torvalds.

One of the advantages of Linux is that it is not a commercial operating system: its source code is under GNU[1] Public License (GPL), that allows anybody to consult it freely, execute it or make a copy of the source files, adapt it or improve it, and distribute it, modified or not. Nevertheless, any software product derived from a product covered by the GPL must, if it is redistributed, be released under the GPL. The main goal of this philosophy is to allow the growth of knowledge by allowing everybody to modify programs at will.

The main part of the Linux operating system is called the *kernel*. The kernel is loaded into RAM at boot time and contains many needful procedures for an efficient running of the system. It must achieve the two following purposes:

- Interact with the hardware components.

[1] GNU is the recursive acronym of "GNU's Not Unix".

- Provide an operating environment for the applications implemented on the computer system (the so-called user programs).

Some operating systems (such as MS-DOS) allow all applications to directly communicate with the hardware components. On the contrary, Unix operating systems and in particular Linux, hide to the user all low-level details concerning the physical organization of the computer. When a user program wants to use a hardware resource, it sends a request to the operating system. The kernel analyzes the request and allocates the resource by interacting with the hardware resource on behalf of the user program. In order to reinforce this approach, Linux introduces two execution modes for the processor: a non privileged mode for the user programs (the *User Mode*) and a privileged mode for the kernel (the *Kernel Mode*).

3.2 User mode and kernel mode

A *process* can be defined as "an instance of a program in execution". When a process runs in user mode, it doesn't dispose of the privilege to directly access to data structures or functions of the kernel. When a process runs in kernel mode, these restrictions no longer apply: kernel data structures and functions can be used without any restriction. Each processor proposes special instructions to switch from the user mode to the kernel mode and vice versa.

An user application runs essentially in user mode and switches to kernel mode only when it requests a service provided by the kernel. When the request is satisfied, the processor puts the process back in user mode. For example, an applicative process cannot normally directly access to a peripheral device. It has to switch to kernel mode before that. When the I/O operations are completed, the processor has to go back to the user mode.

There are several situations to switch in kernel mode:

- A process invokes a *system call.*
- The processor running the process signals an *exception*, which is an unusual condition such as an invalid instruction. Then, the kernel handles the exception on behalf of the process that has caused it.
- A peripheral device issues an *interrupt signal* to the processor to notify it about of an event, such as the completion of an I/O operation. Each interrupt signal is managed by an element of the kernel called an *interrupt handler.*

The computer memory is subdivided into user space and kernel space, and this separation obviously corresponds to the two execution modes. Thus, the user space is the place where applicative processes are running; the kernel space is the one where the kernel code is executed after loading, and where kernel data structures are hosted.

Operations executed in the kernel space include (figure 3.1):

- **Process management:** To achieve the effect of an apparent simultaneous execution of multiple processes, Linux has to switch from one process to another one in a very short time. Process management is concerned with the creation and the destruction of processes, as well as when to switch and which process to choose.

- **Memory management:** In general, the memory is not large enough to contain all the running processes. Before loading a program in memory, it could be needful to liberate space, by releasing the memory of inactive processes. Memory management is concerned with the allocation of the memory space to processes and liberation of space.

- **Device control:** Generally a process needs data from outside (keyboard, disks) and produces a result to be exported outside (screen, disk, printer). Most of the device control operations are performed by code; this code, specific to the device being addressed, is called *device driver*. The kernel must dispose of a device driver for every peripheral present in the system (hard disk, keyboard, screen and so on).

- **Filesystem:** It is concerned with the creation, the access, the move, the destruction and the organization of files or directories and also their storage on volumes.

- **Networking:** Networking must be managed by the operating system because most network operations are not specific to a process: incoming data packets are asynchronous events. The packets must be collected, identified and dispatched to the processes. More, the system is in charge of delivering data packets from programs to the network. Additionally, all the routing and address resolution issues are implemented within the kernel.

3.3 Filesystem

Linux, as any Unix operating system, uses the concept of *file*. An Unix file is a container of information, structured as a sequence of bytes. It does not include any control information such as its length, or an End-Of-File (EOF) delimiter. Each file belongs to one of these following types:

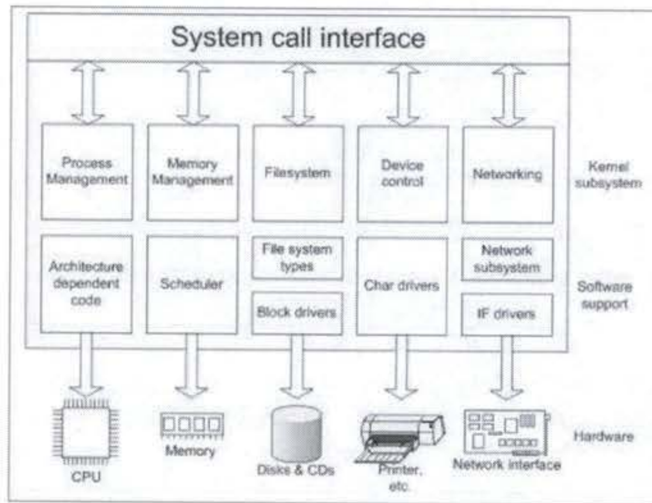

Figure 3.1: A view of the kernel.

- regular file: a set of jointly stored data.
- directory: a set of files regrouped in the filesystem.
- symbolic link: a short file containing the access path to another file.
- pipe and named pipe: a special file used for interprocess communication.
- socket: a software mechanism allowing programs to communicate with a local or remote application.
- device file: file allowing user programs to access hardware devices of the system.

For the user, files are regrouped according to a structure in tree: the *filesystem*. Each node of the tree, except terminal nodes, represents a directory and contains information about files located under this node. A terminal node represents a file of another type (See figure 3.2).

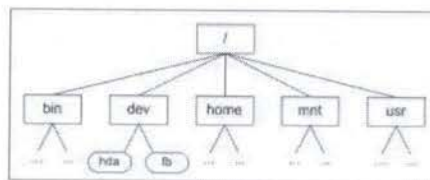

Figure 3.2: An example of filesystem.

At each file of the filesystem corresponds a special structure, an *inode*, used by the system to identify it. The inode structure contains all relevant information for the filesystem to handle a file, such as the type of file, its size, its inode number, the owner id, the access rights, the pointers to data blocks and other type-specific information. Thus, file's content is stored in data blocks, pointed by the inode.

As mentioned in the previous section, the user programs are not authorized to access directly to devices and have to require them through system calls. Because the filesystem is stored on a disk, user programs cannot directly manipulate files and must also use system calls. Therefore, Linux uses several system calls related to file handling. Whenever a process wants to perform an operation on a specific file, it uses the appropriate system call and passes the file pathname as a parameter.

Although all the files seem identical in the filesystem, accessing to the data blocks of a regular file is very different from accessing to a device through a device file, or communicating through a socket. Nevertheless, from the user program point of view, system calls are standardized. Thus, a *read()*[2] system call applied on a regular file allows to read the content of the data blocks, whereas applied on a device file, it allows to extract data provided by the device (for example, the movements of the mouse). Effective differences between regular files and device files are hidden from user programs by the kernel.

Concretely, a system call is not implemented directly: it uses a pointer for each operation. By default, this pointer references the default file operations, which are operations used to manage regular files. For the other files, the kernel updates the pointer in order to replace default file operations by the appropriate functions for the particular file to be accessed.

If a process wants to perform an operation, like read or write a file, it has to open the file before. In order to open a file, a process has to invoke the *open()* system call. This system call creates an "open file" object, and returns a *file descriptor*. This descriptor stores information about the interaction between the open file and the process, and contains pointers to the functions associated to this file. When the file is opened, processes can realize some operations on the file like moving into the file (by using the pointer to the *lseek()* function), reading some part of the file (*read()*) or writing into it (*write()*). When a process doesn't need anymore to access the contents of a file, it can invoke the system call "*close()*", which releases the open file object corresponding to the file descriptor. When a

[2]The function names are postfixed with ().

process terminates, the kernel closes all its still opened files.

3.4 Device driver and device file

3.4.1 Device driver

The kernel interacts with I/O devices by means of special programs, called *device drivers*. Device drivers are part of the kernel and consist of data structures and functions that control the various devices connected to the computer system, such as hard disk, keyboard, mouse, monitor, network interface, and devices linked to a SCSI[3] bus.

Each driver interacts with the kernel in an uniform way. This approach has the following advantages:

- Device-specific code can be encapsulated in a specific module (see below).
- Vendors can add new devices without knowing the kernel source code: only the interface specifications must be known.
- The kernel deals with all devices in a uniform way and accesses them through the same interface.
- It is possible to write a device driver as a module that can be dynamically loaded in the kernel without rebooting the system. It is also possible to dynamically unload a module that is no longer needed, thus minimizing the size of the kernel image stored in RAM.

The figure 3.3 illustrates how device drivers interface with the rest of the kernel and with the processes. If some user programs want to operate on hardware devices, they make requests to the kernel using the usual file-related system calls and the device files normally found in the /dev directory. As a matter of fact, the device files are the user-visible portion of the device driver interface. Each device file refers to a specific device driver, which is invoked by the kernel in order to perform the requested operation on the hardware component.

3.4.2 Device file

Linux considers I/O devices as files; thus, the filesystem includes specific files, the *device files*, that allow user programs to access hardware devices. These device files represent in the filesystem the I/O devices supported by Linux. When a process accesses to a device file, it activates the functions specific to the device. These functions, implemented in the

[3] Small Computer System Interface

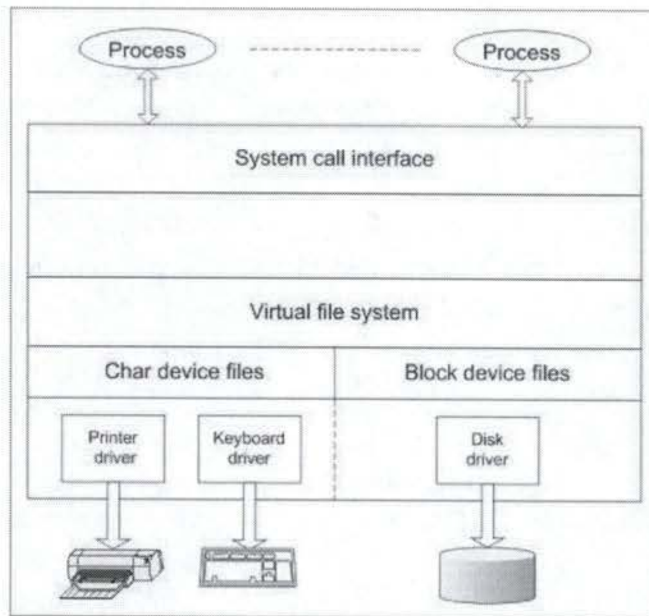

Figure 3.3: The device driver interface.

device driver, are linked to the device file by the kernel. With few exceptions, all device files are located in the /dev directory.

In addition of its name, each device file has three main attributes:

- the class of device;
- the *major number*;
- the *minor number.*

The class of a device concerns the way to exchange data from and to the device. *Char device*, *block device* and *network interface* are the three main types, which are described in the chapter "General description of a Linux device driver".

The major number is a number ranking from 1 to 255 that identifies the device type. Usually, all device files having the same major number and the same class share the same set of file operations, since they are handled by the same device driver. For example, /dev/null and /dev/zero are both handled by the driver number 1. Another example is the hard disk: although each of its partition is considered as a specific device, the partitions own the same major number. The kernel uses this major number when an *open()* is called to request the execution by an appropriate driver.

The minor number, also ranking from 1 to 255, is used by the driver to identify a specific device among a group of devices that share the same major number. For example, each partition of a disk uses a different minor number.

Every time the kernel calls a device driver, it tells the driver which device is being acted upon. The major and minor numbers are paired in a single data type that the driver uses to identify a particular device. A combined device number (the major and minor number concatenated together) resides in the field `i_rdev` of the `inode` structure. Thus, the driver can identify the device by extracting the device number located at `inode->i_rdev`.

3.5 The concept of module

The Linux kernel is monolithic: each kernel function is integrated into the whole kernel program. When a programmer wants to add a new functionality to the Linux kernel, he has two possible approaches: he can decide either to include the new function into the kernel (static approach), or to write the new function as a module, compile it separately and integrate the compiled code to the kernel (dynamic approach).

A *module* usually regroups additional functions to be realized by a filesystem, a device driver, a protocol or other features. A module is an object file, its code can be dynamically linked to (and unlinked from) the kernel at any time during the runtime.

The module is integrated in the kernel in order to serve future requests and proposes two entry points. The first one, the `module_init()` (or sometimes `init_module()`) function, contains all needful code to respond to later invocation of the module's functions (memory allocation, hardware initialization,...). The second entry point, `module_exit()` (or `cleanup_module()`), must be able to undo what was realized by `module_init()`. The two commands `insmod ./<file>.o` and `rmmod <file>` allow respectively to link and unlink a module.

Generally, programmers have the tendency to implement new functions as modules. Because modules can be linked on request, the kernel will not be bloated with hundreds of rarely-used functions and will have a reasonable size. However, some code cannot be added by modularization to the kernel: this happens typically when a new functionality requires a modification of some data structures or functions statically linked in the kernel. As an example, suppose that the functionality needs to introduce new fields into the inode structure. Because the kernel is running, it is impossible to modify the declaration of the

`struct inode` type. The only way to implement the modification is then to add statically the fields in the type declaration and re-compile the kernel.

The link between the kernel and a module implies two main tasks to be performed. First, the kernel must be able to access to global symbols (variables and functions) of the module, such as the entry point of the module. In the same way, a module must be able to access to the kernel symbols or to the symbols of other modules. The second task consists in keeping track of the use of a module, so that no module is unloaded while another module or another part of the kernel is using it.

When a module is loaded, the references to the kernel symbols that are used in the module must be replaced with suitable addresses. The kernel uses a special table to store the symbols that can be accessed by modules together with their corresponding addresses. This table is called the *kernel symbol table.* At load time, all references to the global kernel symbols are replaced by the effective addresses.

A linked module can also export its own symbols, so that the kernel or other modules can access to the global variables and use functions from a module loaded earlier. For that purpose, the module collects all its exportable symbols and includes them into the kernel symbol table.

The figure 3.4 shows how the kernel accesses to the module's functions and how the module accesses to the kernel's functions.

The kernel keeps an usage counter for each module, so that it can determine whether this module can be removed without problem. The system needs this information because a module can't be unloaded if it is used by a process. The counter is incremented when an operation involving the module's functions is started and decremented when the operation terminates. A module can be unlinked only if its usage counter is null.

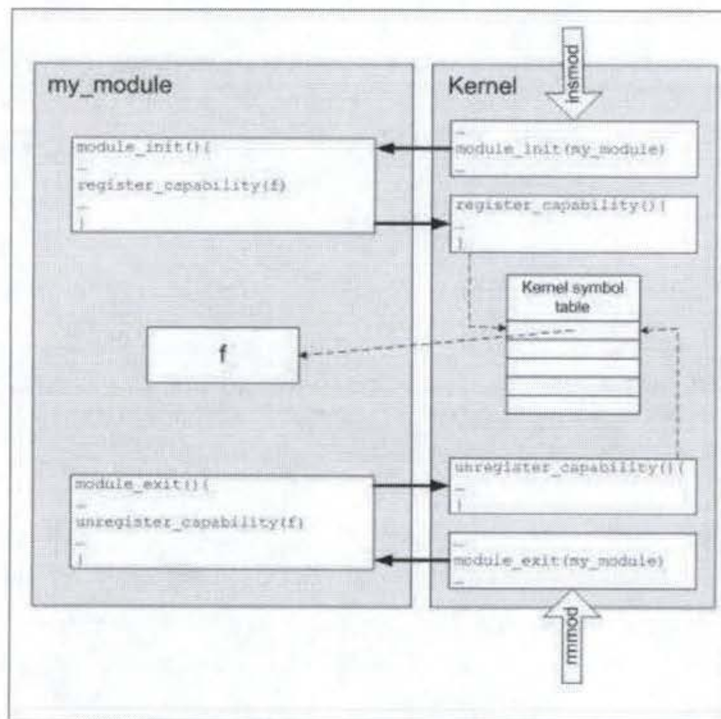

Figure 3.4: Access to the modules symbols.

Chapter 4

The Linux device drivers

4.1 General description of a Linux device driver

4.1.1 Definition and roles of a device driver

Definition

A *device driver* is the part of the operating system that manages communication with a specific device. It is a "black box" that hides the details of a particular device and provides a high level programming interface to it.

Roles

User actions are executed by means of a set of standardized calls that are independent of the specific driver; mapping these calls to device-specific operations that really act on the hardware is then the role of the device driver. Because each device requires a driver running with each type of an operating system, there are several hundred drivers on Linux.

The main functions of a driver are:

- Perform I/O management;
- Provide transparent device management;
- Avoid low-level programming;
- Increase I/O speed because usually it is optimized;
- Include software and hardware error management;
- Allow current access to the software by several processes.

A device driver is a software layer that lies between the applications and the device. Because several applications can use the same device by different ways, it is important for a driver to be flexible. This is the reason why a driver provides mechanisms, and not a policy. Moreover, it is also interesting to keep complexity out of the kernel, in order to maintain the small size of this one. For example, a hard disk can generally be subdivided into several partitions, which can support different files organizations (FAT, Ext, Ext2, NTFS). The hard disk driver cannot impose a files organization, but must let the choice of this organization.

4.1.2 Level of kernel support to hardware device

The kernel can support the access to hardware devices in three possible ways (see figure 4.1):

- **No support at all:** applications interact directly with device I/O registers by using assembly language instructions. The most famous example of this approach is how the *X Window System*, the most common used windows manager under Unix and Linux, handles the graphic display. It is composed of a *X server*, managing a terminal (mouse, keyboard and display) and providing graphic functions, and of one or several *X clients* calling these functions to display windows or drawings. These programs are not included in the kernel: they run in user mode like usual applications.

- **Minimal support:** the kernel does not recognize the hardware device but only its I/O interface. It considers this I/O interface as the one of a device capable of reading and/or writing sequences of bytes. Consequently, the kernel invites the application program to read and/or write bytes and supplies to the device a device driver to handle these bytes. The minimal support approach is used to handle external hardware devices connected to a general-purpose I/O interface, like serial ports.

- **Extended support:** The kernel recognizes the device and handles itself the I/O interface. All hardware devices directly connected to the I/O bus, like internal hard disk, are handled according to the extended support approach: the kernel has to integrate a driver for each device of this type.

Minimal support is preferable to extended support because it keeps small the kernel size. But this approach has also a disadvantage: because of the general-purpose I/O interface, that only offers generic functions, specific interactions between the device and functions or data structures of the kernel are impossible. For example, in the case of a hard disk, it is not possible with the minimal support approach, to recognize the disk and set up its filesystem. In this case, an extended support will be compulsory.

Each device proposes a number of functionalities that the driver has to be able to execute.

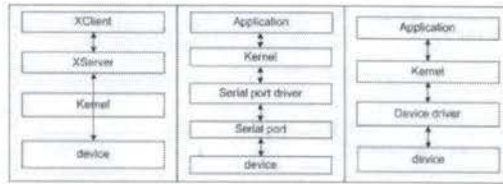

Figure 4.1: Level of kernel support (Left: No support at all. Central: Minimal support. Right: Extended support).

Although the working and the implementation differ for each device, these functionalities often present common characteristics that allow to get together existing devices in several classes.

4.1.3 Classes of devices

Linux distinguishes three device classes: *char devices*, *block devices* and *network interfaces*. At each class of devices corresponds a class of drivers, which allow to access to the devices of this class.

Char devices

A char (or character) device is a device that can be accessed in the same way as a stream of bytes, where bytes are addressed sequentially. Thus, a char device generally produces, and/or expects to receive, an aleatory-sized stream of bytes. The mouse, the keyboard, the display, the serial ports and the printer are examples of char devices, as they are well characterized by a stream of bytes. The main difference between a char device and a regular file is that with the regular file, it is possible to move back and forth, whereas most char devices are just seen as data channels which can only be accessed sequentially.

Char devices are accessed through names in the filesystem, called filesystem nodes. Those names are conventionally located in the `/dev` directory (for example, `/dev/lp0` is the device file corresponding to the printer).

Block devices

A block device is a device that can host a file system, accessible only by block of bytes. Hard disks, floppy disks and CD-ROM are the most usual examples of block devices. These kinds of devices have a very high average access time. Each operation requires several milliseconds to complete, mainly because the hard disk controller must move the heads on the disk surface and turn the disk to reach the exact position of the data. However, when

the heads are correctly placed, data transfer is performed at rates of tens of megabytes per second.

In order to achieve acceptable performance, block devices transfer several adjacent bytes at once. Groups of bytes are *adjacent* when they are recorded on the disk surface in such a way that a single search operation can access to them. The hardware imposes as transfer unit the *sector*, which is generally a set of 512 adjacent bytes.

However, at the driver level, each transfer request concerns a fixed-size set of adjacent bytes, called *block*, in a single operation. In Linux, block size must be a power of 2, and cannot be larger than a page frame[1]. Because of the hardware constraints, a block size must also be a multiple of the sector size. Therefore, in many architectures, the authorized block sizes are 512, 1024, 2048, and 4096 bytes.

Like char devices, block devices are accessed by means of filesystem nodes in the `/dev` directory. They are also identified by a major and a minor number, unique for each block device file. The major number identifies the driver associated with the device, for example `/dev/hd`. The minor number is used only by the driver identified by the major number (other parts of the kernel do not use the minor number).

The space of a block device can be partitioned: each partition is then considered as a specific device, and is accessible by its own device file. Block size is fixed for the partition at its creation. Thus, a disk can host several block sizes, but only one per partition.

Another characteristic of block devices is that blocks of bytes stored on the device can be addressed randomly: the time needed to transfer a data block can be assumed independent of the block address inside the device and of the current device state.

An efficient way to avoid low access time consists of using a *buffer cache*, which is a disk cache storing blocks. The idea behind the buffer cache is to relieve user processes having to wait for relatively slow disks to retrieve or store data. Thus, it would be counterproductive to write a lot of data at once; instead, data should be written piecemeal at regular intervals so that I/O operations have a minimal impact on the speed of the user processes and on response time resented by human users.

[1]The available RAM is partitioned into page frames 4 or 8 KBytes in length.

Network interfaces

All network transactions are made through an interface, that is a device able to exchange data with other hosts. Usually, an interface is a hardware device, but it might also be a software device, like the loopback interface[2]. A network interface is in charge of sending and receiving data packets, without knowing how individual transactions map to the actual packets being transmitted.

Each network interface is identified by an unique name (such as `eth0`), but that name has not a corresponding entry in the filesystem. Instead of using the filesystem, a relationship between the device name and a network address must be set up. Therefore, data communication between an application program and the network interface is not based on the standard file-related system calls; it is based on other system calls, such as socket(), bind(), listen(), accept(), and connect(), which act on network addresses.

As our work concerns displays, the next section will be focused on the description of a char driver, using as an example the frame buffer driver fbmem.c (see Appendix C).

4.2 Char drivers

Char drivers are drivers that allow to transmit information from the user to the device, or vice versa, byte per byte. Because bytes are exchanged one by one, there is no need to have a buffer for the exchange of data.

User programs use several standardized system calls, such as open(), release(), read() and write(), to realize I/O operations. A device driver usually implements these different system calls to specialize them to a specific device.

In order to access to the functions behind these system calls, the kernel uses a specific structure: the `file_operations` structure. This structure contains a field for each possible system call, allowing to point towards the adequate function. Each file is associated with its own set of functions. The following list shows which operations generally appear in a char driver and what they allow to do on the device.

The setup() function

```
int __init video_setup(char *options)
```

[2]A loopback is an interface allowing to connect to himself exactly as to another host.

Sometimes, it is desirable to pass parameters to a device driver or to the Linux kernel in general. This may be necessary when automatic detection of hardware is not possible, or may result of a conflict with another hardware. This function is less used in recent versions of Linux and Linux device drivers.

The init() function

```
void __init fbmem_init(void)
```

The `init()` function is only called at boot time and is used to initialize the driver. The function has three main roles: it must test for the presence of a device, create internal device driver structures and finally register the device at the kernel level. This is done when the system is started up or when the driver modules are initialized, by using the following function:

```
int {devfs_}register_chrdev (unsigned int major, const char *name, struct
file_operations *fops);
```

This instruction associates, in a table handled by the kernel, a major number with a device file's name and its associated functions, put in argument. The major number is used as index in this table. If the `register_chrdev()` instruction is called with zero as major number, the kernel will associate dynamically a free major number to the specified device file. Otherwise, the value of the `major` argument is used as major number for the device.

The open() and release() functions

```
int fb_open(struct inode *inode, struct file *file);
int fb_release(struct inode *inode, struct file *file);
```

The `open()` function is used to realize initialization, in preparation for later operations with the device. It has to perform several tasks:

- If the driver is integrated in a module, it increments the usage count to avoid being unloaded before the closing of the device file.
- If the driver handles several devices, it selects the appropriate one by using the minor number. Possibly, it specializes the file operations table.
- The driver checks for device-specific errors (hardware problems, device not ready,...).
- Possibly the driver sends an initialization request to the hardware device.

- The driver initializes and fills any data structure (sometimes, some of these actions are realized at boot time by the init() function).

The role of the release() function is the opposite of the open() function. It has to execute the following tasks: liberate and clean the data structure used by the driver, decrement the usage count and possibly shutdown the device.

Though open() and release() are always the first and the last operations performed on the device, these functions can be missing.

open() and release() need two arguments. The first argument, inode, is a pointer to a inode structure. This structure contains data that allow to assuredly identify the device. The second argument, the file structure, is an important data structure used in device drivers and represents an open file. It contains several information like if the file is readable or writable (or both), the current reading or writing position, different flags for critical resources managing and a pointer to the file_operations structure (see below). These two methods return 0 when they succeed and an error number when they fail.

The read() and write() functions

```
ssize_t fb_read(struct file *file, char *buffer, size_t size, loff_t *offset);
ssize_t fb_write(struct file *file, const char *buffer, size_t size, loff_t *
offset);
```

The instruction read() allows to retrieve data from the device and serve them to an application. write() allows to send data to the device and works like the read() function but in the opposite sense. Because device data and user application are in different address spaces, the read() and write() functions have to move segments of data from the kernel space to the user space, or vice versa, and call copy_to_user() or copy_from_user() to do that.

The file argument is the same as in open() and release() and allows to identify device file. count contains the size of the data to transfer. buffer points to a user buffer that contains data to write to the device's I/O memory or points to an empty memory area in the user space where read data must be stored. Finally, offset points to a variable that contains the current position in the user buffer. The returned value is the number of bytes effectively transferred. For example, when an application uses the write() system call, in order to write pixels into the frame buffer, the fb_write() function in fbmem.c works like this: buffer points to the beginning of the user buffer. The user application can transfer pixels in several segments by calling several times the fb_write() function. count tells

the size of a transfer and `offset` the beginning of the transferred segment.

The ioctl() function

```
int fb_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg);
```

The ioctl system call offers a way to issue device-specific control commands, like reading or changing device parameters, reading or modifying the palette,... `ioctl()` is device specific in the sense that, unlike read, write and others methods, it allows applications to access to specific features of the hardware being driven. Most of the ioctl implementations consists of a switch instruction that selects the adapted actions according to the `cmd` argument. The last parameter, `arg`, allows to pass an argument to the selected command. The type of `arg` authorizes to pass only one argument, but it could be a pointer to a set of data, like a structure for example.

By looking at the ioctl() code, we can observe two kinds of commands: commands that "get" data, and commands that "put" data. In both case, there is exchange of data (the `arg` argument) between user space and kernel space and there is a need to use `copy_to_user()` and `copy_from_user()` to do that.

The ioctl() commands of the fbmem.c driver allows to extract or modify the main data structures that concern the display, the LCD controller and the frame buffer.

The llseek() function

```
loff_t llseek(struct file *file, loff_t *offset, int whence);
```

The llseek() method allows an user program to change the current read/write position in a file, the new position being returned as a return value. The `offset` argument indicates the length of the offset and `whence` defines from which place (beginning or end of the file, current position). llseek() is not implemented in the frame buffer driver.

The poll() function

```
unsigned int poll(struct file *file, struct poll_table_struct *table);
```

The `poll()` method is the back end of two system calls, *poll* and *select*, both used to inquire if a process can read from or write to one (or more) open file(s) without blocking. These system calls are used by applications that must use multiple input or output streams

without blocking on any one of them. The method returns a bit mask describing operations that could be immediately performed without blocking. `poll()` is not implemented in the frame buffer driver.

The mmap() function

```
int fb_mmap(struct file *file, struct vm_area_struct *vma);
```

Linux provides the mmap() system call, that allows to a part of memory device to be associated, or mapped, with the process address space. It means that whenever the program reads or writes in the assigned address space, it accesses the memory device; an operation (read or write) on a byte of the assigned address space is translated by the kernel into the same action on the corresponding byte in the memory device. For example, using mmap() on the `/dev/fb` file allows to an application a quick and easy access to the frame buffer memory: in order to modify some pixels, the application has just to write new values directly at the good place, and does not have to take up position with llseek() before writing each pixel.

The mmap() function asks two arguments: the usual `struct file`, and `vma` that contains information about virtual addresses used to access to the device. mmap() returns 0 if it succeeds, otherwise an integer representing an error.

The registration function

The kernel needs to know which devices are available. For this reason, the kernel maintains two tables, `chrdevs` for char devices and `blkdevs` for block devices, that contain a description of each device. This description indicates the major number and the name of the device, as well as the possible operations on it. If a device driver is statically included in the kernel, like the frame buffer driver, the corresponding device is registered during the system initialization. However, if a device driver is dynamically loaded as a module, the corresponding device is registered when the module is loaded and unregistered when the module is unloaded.

```
int {un}register_framebuffer(struct fb_info *fb_info)
```

The frame buffer driver registers itself by calling `register_framebuffer()`, with a pointer to a `fb_info` structure in argument. This function registers the major number and maintains an internal list of which frame buffer device is in charge of each minor number. The `fb_info` includes everything needed for specific device management. It also creates a device file in the /dev directory.

Logically, the `unregister_framebuffer()` function allows to unregister the driver by deleting corresponding entry in the table and removing device file in the filesystem.

4.3 A methodology for device driver designing

4.3.1 The ideal device driver

The ideal driver is the one that hides completely the hardware. This means first that the device driver should be the only piece of software in the entire system that reads or writes the device's control and status registers. Secondly, when the device generates an interrupt, this interrupt should be integrally processed by the device driver. Moreover, it should be not needed to change the programming of the driver when the underlying peripheral is replaced by another one of the same type; for example, the programming interface of a "flash" memory driver should work with any "flash" memory device.

Nevertheless, in practice, hiding the hardware completely is difficult, because of the broad features of the devices. But trying to design towards an ideal device driver gives the three benefits:

1. The modularization makes that the structure of overall software is easier to understand and maintain.
2. Because only one driver interacts directly with the peripheral's registers, the hardware state can be tracked more accurately;
3. Interface software changes are localized in the device driver.

The most tangible effect is the reduction of the system bugs.

4.3.2 Design methodology

The methodology of hiding hardware particularities and interactions when designing a device driver consists of five rules. To implement them as simply and incrementally as possible, these rules should be used in the following order.

1. **Defining a data structure that overlays the memory-mapped control and status registers of the device.**

   The first step in the driver development process is to create a structure that looks just like the memory-mapped registers of the device. This structure contains fields

corresponding to the control and status registers. To make the bits within the control register easier to read and write individually, it is also interesting to define bitmasks. A *bitmask* generally consists in a 32 bits word that can be combined, with help of logical operations, with a register content to select a specific (set of) bit(s). For example, the bitmask of the LEN field of the LCCR0 register is defined like this:

```
#define LCCR0_LEN    0x00000001    /* LCD ENable    */.
```

It can be used to put the bit LEN to 0 of the LCCR0 register, and that whatever the value contained in `fbi->reg_lccr0`.

```
LCCR0 = fbi->reg_lccr0 & ~LCCR0_LEN;
```

2. **Declaring a set of variables to store the current state of the hardware and device driver.**

   The second step in the driver development process is to figure out what variables will be needed to contain the state of the hardware and device driver. For example, it would be useful to keep in a variable the number of bits used to represent a pixel.

3. **Implementing a routine to initialize the hardware to a known state.**

   Once defined the way to store the state of the device, one can start writing the functions that interact with the device and control it. The best is to begin with the hardware initialization routine. This is to be made anyway, and it is a good way to get familiar with the device interaction.

4. **Implementing a set of routines that, taken together, provide an API for users to the device driver.**

   In this step, the other functions are added to the driver. This step requires a good knowledge of the different operations that the hardware can perform. It is also needed to know how to request the operations and transmit parameters to the device. All these functions must be tested in each possible condition. It is also important to keep in mind that many devices are critical resources.

5. **Implementing one or more interrupt service routines**

   This step consists in identifying the possible interrupts, locating where the driver can detect them, and implementing appropriate answers.

Chapter 5

Interrupts

An *interrupt signal* is defined as an event that alters the sequence of instructions executed by the processor. Such events correspond to electrical signals generated by hardware circuits inside or outside the processor.

Interrupt signals are subdivided into exceptions, hardware interrupts and software interrupts:

- An *exception* (or synchronous interrupt) is produced in the CPU control unit when special instructions are executed or when an error or an abnormal situation occurs. This interrupt is also called synchronous interrupt because the control unit issues it only after terminating the execution of the processed instruction.

- A *(hardware) interrupt* (or asynchronous interrupt) is generated by another hardware device, such as a timer or an I/O device, at arbitrary time with respect to the CPU clock. For instance, the reception of a complete frame by the display system causes an interrupt.

- A *software interrupt* is generated by a special software instruction. Such an interrupt is generally used to implement system calls.

This chapter is essentially focused on the hardware interrupts, because they concern the interactions between the operating system and the devices.

5.1 Polling mode and interrupt mode

The duration of an I/O operation is often unpredictable. It can depend on mechanical considerations (the current position of a disk head with respect to the block to be transferred), on random events (when a data packet arrives in the network card), or on human

factors (when a user presses a key on the keyboard). In any case, the device driver that started an I/O operation needs a monitoring mechanism signaling to the CPU either the termination of the I/O operation or a timeout. The two techniques available to monitor the end of an I/O operation are respectively called *the polling mode* and *the interrupt mode.*

In the polling mode, the processor repeatedly checks (polls) the status register of the device, until its value signals the completion or the failure of the I/O operation. Because an I/O operation can take a long time, the driver has to release the CPU after each polling operation, to allow other processes to continue their execution.

In the interrupt mode, the I/O controller of a device signals to the processor the completion or the failure of an I/O operation, by sending an electrical interrupt signal. At the reception of an interrupt, the processor transmits this one to an *interrupt handler*, which is able to interpret the interrupt and perform the needed actions.

On the whole, interrupts offers a much more efficient use of the processor than polling, because the processor is able to use a larger percentage of its waiting time to perform useful actions. However, it takes time to put aside the current process and transfer control to the interrupt handler.

5.2 Interrupts handling

This section describes how interrupts are handled and is subdivided into three parts: the first part explains the way an interrupt generated by a device is detected by the CPU. The second part describes how the kernel selects the appropriate interrupt handler, and the third part is focused on the role of an interrupt handler and how it works.

5.2.1 Interrupts detection

Most of devices are able to transmit interrupts to the processor. An interrupt is just an electric signal emitted by the device controller on an output line. All these lines are connected to the input pins of a hardware circuit called the *interrupt controller*, which warns the processor when an interrupt occurs. There are two kinds of interrupts:

- The *unmaskable interrupts*, that cannot be disabled. Only a few critical events, such as hardware failures, give rise to unmaskable interrupts.
- *The maskable interrupts*, that can be disabled. Maskable interrupts are generally used to signal non critical events, such as the completion of a data transfer. The lines transmitting maskable interrupts are called *IRQ* (Interrupt ReQuest).

When the interrupt controller receives an interrupt, it stores the line number in one of its registers, called ICIP[1], which is accessible to the processor. Afterwards, the interrupt controller sends a signal to the processor to point out that an interrupt has occurred (see figure 5.1). The processor can then start a new sequence of instructions, called the *kernel control path*.

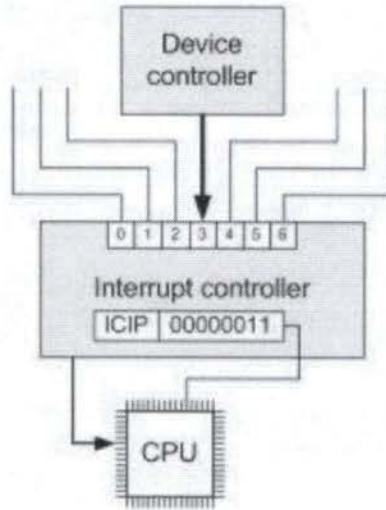

Figure 5.1: Detection of an interrupt by the processor.

5.2.2 Interrupt handler selection

When an interrupt is signaled to the processor, this latter stops what it is currently doing and switches to a new sequence of instructions, called the *kernel control path*. This step requires to save the current content of the program counter on a kernel specific stack, and to actualize the counter with the address of the first instruction of the kernel control path.

The kernel control path is executed in kernel mode. The first action performed by the kernel control path consists in saving the current content of the processor registers in the kernel stack. It is necessary to keep these values in order to restore the processor in its previous state at the end of the interrupt handling. After that, the kernel control path reads the content of the ICIP register and confirms the reception of this one to the interrupt controller.

[1]for Interrupt Controller IRQ Pending.

Afterwards, the kernel control path executes the appropriate *interrupt handler*, which is a function capable to handle the interrupt. In practice, the kernel maintains a table, called *Interrupt Descriptor Table* or IDT, that contains the address of each interrupt handler. The content of the ICIP register indicates the position, in the IDT table, of the appropriate interrupt handler address.

At the end of the interrupt handler execution, the kernel control path restores the CPU registers with the values stored on the top of the stack, and restores the program counter with the previous saved value.

An interrupt might occur when the kernel is handling another interrupt. The kernel allows to interrupt a kernel control path to execute another one, with the consequence that the kernel control paths can be nested.

Interleaving the kernel control paths allows to confirm faster to the interrupt controller the reception of an interrupt. Indeed, this one remains blocked until receiving the acknowledgment. Thanks to the interleaving, the kernel is able to send the acknowledgment even if it is handling a previous interrupt.

5.2.3 Interrupt handler role and working

An interrupt generally signals the reception of a set of data, for instance the reception of an entire frame by the LCD controller or the reception of a data packet by a network interface. The role of an interrupt handler is to give feedback to its device about interrupt reception.

The first step usually consists of clearing a bit of the device controller register; most hardware devices will not generate other interrupts until their "interrupt-pending" bit has been cleared. The other typical task of an interrupt handler is to awake sleeping processes, which are waiting for an event.

The number of IRQ lines is generally limited, so that several devices might share a same IRQ line. This means that the ICIP register does not give a sufficient information to identify the source of the interrupt. Therefore an interrupt handler must be flexible enough to serve several devices. Thus, several *interrupt service routines* (ISR) may be associated with the same interrupt handler; each of them is a function related to a single device sharing the IRQ line, and implements the task of the interrupt handler concerning the device (figure 5.2). As it is not possible to know in advance which particular device uses the IRQ, each ISR is successively executed to verify whether its device needs attention, by consult-

ing the status register of the device controller. So, the ISR performs all the operations needed to be executed when the device raises an interrupt. The interrupt service routine

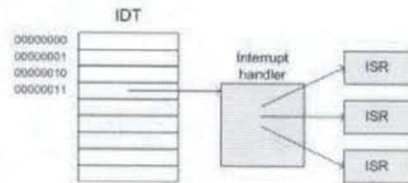

Figure 5.2: Selection of the interrupt handler and access to the ISRs.

of a device is implemented into its driver. Each ISR requires the same arguments: `irq` represents the IRQ number, `dev_id` is the device identifier and `regs` is a pointer to the kernel stack area containing the registers saved just after the interrupt has occurred. The first argument allows a single ISR to handle several IRQ lines, the second one allows the ISR to take care of several devices of the same type and the last one gives the access to the execution context of the interrupted process. Generally, these arguments are rarely used.

5.3 The initialization

Because the IRQ lines are in limited number, a device has to request to the kernel an IRQ line before using it. This is generally performed by the init() function of the device's driver. Thus, the driver proposes an IRQ number, which can be accepted or refused if this IRQ is already reserved by a device refusing to share it. More, the init() function has also to provide the ISR corresponding to the device. These two actions are performed at the same time by calling the following function:

```
int request_irq(unsigned int irq, void *isr, unsigned long flags, const
char *dev_name, void *dev_id);
```

where `irq` is the interrupt number being requested, and `isr` is a pointer to the interrupt service routine being installed. `flags` can take one of these two possible values: SA_INTERRUPT, indicating that the ISR must be executed with interrupts disabled; or SA_SHIRQ, signaling that the interrupt line can be shared between devices. `dev_name` provides the name of the device owner of the interrupt line and `dev_id` points to the data structure of the related device. `request_irq()` returns 0 to indicate success or a negative value to notify an error.

A driver has also to release the IRQ when it does not use it anymore. This is especially the case when a module is unloaded. Releasing an IRQ is simply performed by using the function:

```
void free_irq(unsigned int irq, void *dev_id).
```

5.4 Bottom and top half parts of an interrupt handler

Interrupts can occur at any time, for instance during the execution of a process. As the duration of an interrupt handling may be long, the waiting processes is sometimes blocked for a long time. Moreover, some parts of the interrupt handling are not urgent and could be run later. For instance, assume a block of data arriving on a network line. This event leads to a hardware interrupt of the kernel. As this interrupt is generally long, the current process is stopped for the same long time. When the hardware interrupts the kernel, it could be simpler to mark the presence of data, let the processor come back to its previous task, and then process the incoming data (i.e. moving the data into a buffer where its recipient process can find it).

Linux resolves this problem by dividing an interrupt handler in two parts: a *top half* part, that the kernel executes right away to respond to an interrupt, and a *bottom half* part, which is a routine required by the top half, but executed later, at safer time.

The bottom half part generally performs low priority or longtime actions such as awakening processes, starting another I/O operation, handling received data and so on. This is a low-priority function awaiting that the kernel finds a convenient moment to run it. Bottom halves that are in waiting state will be executed only when the kernel finishes handling a system call, an exception or an interrupt, or executes the schedule() function to select a new process to run. Thus, when a top half requests the execution of a bottom half, a long time interval can occur before its execution.

The kernel has to keep track of all the functions representing bottom halves in waiting state and must be able to call them at another time for processing. Thus, a bottom half is generally put into a *task queue*, a *kernel task queue* or into a *tasklet*, which are queue or list of functions being waiting for execution. The working of the task queues and the tasklets is described in the next sections.

5.5 Task queues

A *task queue* is a linked list of tasks where tasks (such as a bottom half) are stored in order to delay their execution to a determined-system safe time (see figure 5.3). Thus, tasks are

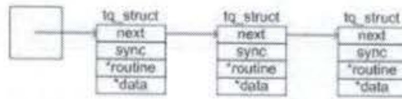

Figure 5.3: A task queue.

progressively accumulated in a queue to be executed later, when the queue execution is required. A task put in a queue several times is executed only once. Generally, a task has low priority and might take a long time to finish. Task queues are not only used to delay the execution of bottom halves: they can be used to perform repetitive actions, such as polling a device, without burdening too much the processor.

A task comprises a function and an argument. When a task is run, its function is simply executed with the argument. The argument is a pointer to a data, which can be either a single value or a data structure. In any case, the function returns `void`. A task in a queue is defined by the following structure:

```
struct tq_struct {
    struct tq_struct *next;
    int sync;
    void (*routine)(void *);
    void *data;
};
```

The most important fields in this data structure are `routine` and `data`. These two fields must be initialized before inserting the task in the queue. The initialization is made respectively with a reference to the function and a reference to the argument. `next` is used to store a pointer to the next task in the queue and `sync` is used by the kernel to prevent queueing the same task several times. The two fields must also be initialized, with respectively null and 0. Another data structure involved in task queues is `task_queue`, which is the type of a task queue. This is just a pointer to a `struct tq_struct` task. The following list presents operations that can be performed on task queues and tasks. Their effect is summarized in the figure 5.4

```
DECLARE_TASK_QUEUE(name);
```

This macro declares a task queue with `name` as variable name and initializes this pointer to null.

```
int queue_task(struct tq_struct *task, task_queue *list);
```

This function puts a task into a task queue. The return value is 0 if the task was already present on the given queue, nonzero otherwise.

```
void run_task_queue(task_queue *list);
```

This function is used to consume a queue of accumulated tasks. When all the tasks are executed, the task queue is empty.

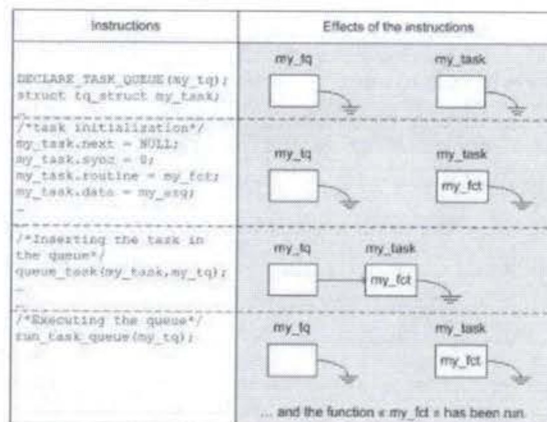

Figure 5.4: Initialization and execution of a task queue.

Task queues are generally run in interrupt mode. Thus, three constraints are imposed to the function of a task. First, no access to user space is allowed. The only way to receive data from the user space is to use the argument. Secondly, the task cannot sleep. A task sleeps when it executes a wait function. Finally, the function cannot request a process scheduling.

5.6 Kernel task queues

Classical task queue presents a disadvantage: the queue execution must be explicitly requested by calling `run_task_queue()`. For this reason, the easiest way to perform deferred execution is to use the queues that are already maintained by the kernel. Kernel task queues are automatically run by the kernel and can contain several tasks coming from many places in the kernel. There are few of these queues, but a driver generally uses one of these following three ones:

The scheduler queue

Contrary to a classical task queue, the scheduler queue allows its tasks to sleep. As a matter of fact, the kernel runs a special process, called *keventd*, whose sole job is running tasks from the scheduler queue. Thus, tasks from the scheduler queue are run in process context and not in interrupt mode. Moreover, as a task shares the scheduler queue with other tasks, also authorized to sleep, the time that elapses before a task runs could be significant. The scheduler queue is not directly accessible; rather than use `queue_task()` with the scheduler queue in argument, the code must call the following function to put a task on the queue:

```
int schedule_task(struct tq_struct *task);
```

`task`, of course, is the task to be scheduled. The return value is 0 if the task was already present on the given queue, nonzero otherwise.

The timer queue

The system has a timing hardware feature, called *timer*, used by the kernel to keep track of time intervals. The timer generates an interrupt at regular intervals, which provokes the execution of an interrupt service routine. This ISR increments a counter, that the kernel uses for instance to know the current time or to delay operation for a specified amount of time.

The timer queue is run at the time of a timer tick. It is used to store the tasks that must be done at the next timer interruption. Tasks run from the timer queue are run in interrupt mode: they cannot sleep, exchange data with user space or request a scheduling. Contrary to the scheduler queue, the timer queue is directly accessible: to put a task in the timer queue, the code must simply use the `queue_task()` instruction, with `tq_timer` as queue name.

The immediate queue

The immediate queue is the fastest queue in the system. It is run as soon as possible but at a safe time, either as soon as a process returns from a system call or when the scheduler is run. Like the timer queue, the immediate queue is run in interrupt mode, so that it cannot sleep, exchange data with user space or request a scheduling.

5.7 Tasklets

Tasklets look like task queues in a number of ways. They are a way to defer a task until a safe time and they are always run in interrupt mode. Like task queues, a tasklet will be run only once, even if it is inserted several times. But contrary to a task queue, each tasklet is associated with only one function, which is called when the tasklet is executed. This function receives an unsigned long argument and returns no value. A tasklet must be declared with one of these two functions:

```
DECLARE_TASKLET(name, function, data);
DECLARE_TASKLET_DISABLED(name, function, data);
```

These functions declare a tasklet with the given name; when the tasklet is executed, the given function is called with the (unsigned long) data value. In the second function, the initial state of the tasklet is disabled, meaning that it will not be executed until enabled at some future time. The following instruction must be used to request the execution of the tasklet:

```
tasklet_schedule(&tasklet);
```

The tasklet system provides two other functions for advanced use of tasklets:

```
void tasklet_disable(struct tasklet_struct *t);
void tasklet_enable(struct tasklet_struct *t);
```

The first function disables the given tasklet. The tasklet may still be called, with `tasklet_schedule()`, but its execution will be deferred until a time when the tasklet has been enabled again. The second function logically allows to enable a tasklet that had been previously disabled. If a disabled tasklet has already been called with `tasklet_schedule()`, it will run soon.

Chapter 6

The LCD controller frame buffer driver

This section concerns the description of the hardware dependent part of the LCD controller frame buffer[1] driver, implemented in sa1100fb.c[2]. The power reduction techniques for TFT LCD displays have been implemented in this driver.

6.1 Device dependent and independent levels

The Linux operating system supports various architectures. To allow that, the drivers implementation has to take into account many kinds of devices, having many similarities but also some differences. This is especially the case of the frame buffer driver, that can manage till fifty kinds of controller devices.

In order to support all of them without weighing down the kernel, the frame buffer driver is subdivided into two levels: the device independent level, implemented in the fbmem.c file, and the device dependent level, implemented in a different file for each kind of device. The device independent level is common for all kinds of devices; the device dependent level is specific to a device and specializes the driver for this device. When the kernel is compiled, the compiler selects the appropriate device dependent file according to the compilation's parameters and neglects the others.

In practice, a variable in fbmem.c, called `fb_drivers`, is initialized at boot time with the name of the init() or setup() function of the device dependent level. When fbmem_init() or fbmem_setup() are called, they use this variable to require the execution of the device

[1] See Chapter two.

[2] See Appendix D.

dependent init() or setup() functions.

For the StrongARM 1100 LCD controller, the device dependent level is defined in the sa1100fb.c file. Device independent function and structure names are prefixed by *fb_* or *fbmem_*, while LCD controller specific function and structure names generally begin by the *sa1100fb_* prefix.

6.2 The initialization

```
int __init sa1100fb_init(void)
```

The *sa1100fb_init()* function performs eight main operations:

1. Initialize the main device-related data structures;
2. Insert the "controller task" into the task queue;
3. Initialize the frame buffer memory;
4. Install the interrupt service routine;
5. Define the content of the LCD controller registers;
6. Register the frame buffer driver;
7. Register the device power management function;
8. Update the controller registers and enable the LCD screen and the controller.

6.2.1 Initializing the main device-related data structures

The LCD controller frame buffer driver uses seven important structures. These seven structures are divided up two files, which are fb.h[3] and sa1100fb.h[4]. The following structures are included in fb.h:

- `fb_var_screeninfo` is used to describe the display features modifiable by the user, such as the resolution, color depth or the clock frequency.

[3]See Appendix E.

[4]See Appendix F.

- `fb_fix_screeninfo` defines the properties that are created when a display mode is selected and cannot be changed otherwise. Thus, the content of `fb_fix_screeninfo` can only be modified by changing the display mode, whereas the content of `fb_var_screeninfo` can be changed anytime. An example is the start of the frame buffer memory. This can depend on what mode is set, and while using a mode, memory position does not have to change.

- `fb_cmap` is concerned with colors definition in the palette or color map.

- `fb_ops` regroups a collection of needed functions to control the device, such as `fb_open()` or `fb_ioctl()`, and also device specific functions. This structure is instanced two times: once in fbmem.c to reference the functions that are accessible everywhere in the kernel; once in sa1100fb.c to regroup the functions usable by fbmem.c.

- `fb_info`, includes the four previous structures and thus is able to contain a configuration for the display system.

The driver also uses two other structures, called `sa1100fb_info` and `sa1100fb_mach_info`, which are device dependent and defined in sa1100fb.h. `sa1100fb_info` includes the `fb_info` structure and is used to store the current configuration of the display system. The `fb_ops` structure is initialized with a variable called `sa1100fb_ops`, which contains references to five functions:

- `sa1100fb_get_fix()`

- `sa1100fb_get_var()`

- `sa1100fb_set_var()`

- `sa1100fb_get_cmap()`

- `sa1100fb_set_cmap()`

The first three functions concern the display mode handling and the user settings, and the two other functions are related to the palette.

Only the device dependent structures are initialized. Values affectation to the `sa1100fb_info` structure is performed by calling the `sa1100fb_init_fbinfo()` function. These values are affected to a variable pointed by `fbi`. The following table describes the main fields of the `sa1100fb_info` structure.

struct sa1100fb_info	struct fb_info	struct fb_var_screeninfo, struct fb_fix_screeninfo and struct fb_cmap	Description
struct fb_info fb	struct fb_var_screeninfo var	__u32 xres	horizontal resolution of picture
		__u32 yres	vertical resolution of picture
		__u32 bits_per_pixel	size of pixel data
		__u32 pixclock	pixel clock period (picosecond)
		__u32 left_margin	time from sync to picture (hor.)
		__u32 right_margin	time from picture to sync (hor.)
		__u32 upper_margin	time from sync to picture (ver.)
		__u32 lower_margin	time from picture to sync (ver.)
		__u32 hsync_len	length of horizontal sync
		__u32 vsync_len	length of vertical sync
	struct fb_fix_screeninfo fix	unsigned long smem_start	physical address of fb memory
		__u32 smem_len	length of frame buffer memory
	struct fb_cmap cmap	__u32 start	first entry of the palette
		__u32 len	number of entries
		__u32 *red	table of red entries
		__u32 *green	table of green entries
		__u32 *blue	table of green entries
		__u32 *transp	table of transparency levels
	struct fb_ops *fbops		frame buffer operations
dma_addr_t dbar1			next value for DBAR1
dma_addr_t dbar2			next value for DBAR2
u_int reg_lccr0			next value for LCCR0
u_int reg_lccr1			next value for LCCR1
u_int reg_lccr2			next value for LCCR2
u_int reg_lccr3			next value for LCCR3
volatile u_char state			state of the LCD controller

The `sa1100fb_mach_info` structure is used to contain the initial display features that will be written into the control registers (LCCR0, 1, 2 and 3) at the initialization time. These values will be also written in `fbi->fb.var` and correspond to the normal configuration of the screen.

6.2.2 Inserting the "controller task" into the task queue

After initializing the data structure, the `sa1100fb_init_fbinfo()` function inserts a task into a task queue. This task, called the *controller task*, consists essentially in writing the content of `fbi->reg_lccr0, 1, 2 and 3` into the four controller registers. To do that, the task only calls `set_ctrlr_state()`, whose the working is explained further.

The controller task will be executed only at the end of the initialization process or when the configuration is modified during the running time. The execution of the task is requested by the `sa1100fb_schedule_task()` function.

```
static inline void sa1100fb_schedule_task(struct sa1100fb_info *fbi, u_int
state)
```

6.2.3 Initializing the frame buffer memory

`static int __init sa1100fb_map_video_memory(struct sa1100fb_info *fbi)`

The driver must allocate memory to the frame buffer. This memory area will contain the palette, where colors are stored, and the pixel data. `sa1100fb_map_video_memory()` allows to make the reservation of this space.

6.2.4 Installing the interrupt service routine

This step is performed by calling `request_irq()`. This function requests an interrupt channel (IRQ) and associates it with an interrupt service routine (ISR)[5]:

`static void sa1100fb_handle_irq(int irq, void *dev_id, struct pt_regs *regs).`

This handler is called when a bit is set into the status register LCSR. The interrupt handler begins by clearing the "interrupt-pending" bit, in this case the LDM field in LCCR0, in order to reactivate interrupts. The next performed step is awaking all the processes concerning the device. The concerned process will be able to handle the interrupt.

6.2.5 Defining the content of the LCD controller registers

`static int sa1100fb_set_var(struct fb_var_screeninfo *var, int con, struct fb_info *info)`

The fifth step consists in calculating the content of the LCD controller registers according to the values stored in the argument "var".

This step, relatively complicated, is performed by calling the `sa1100fb_set_var()` function, which generates the following actions:

1. Validate the new display mode settings, stored in "var", or eventually adapt them (performed by `sa1100fb_validate_var()`);
2. Store the dimensions of the palette and configure it (performed by `sa1100fb_hw_set_var()`);
3. Define the new content of the controller registers and store it in `fbi->reg_lccr0, 1, 2 and 3` (performed by `sa1100fb_activate_var()`);

[5]See the chapter 5 "Interrupts".

4. Request the execution of the controller task (performed by `sa1100fb_schedule_task()`). The effective configuration of the controller registers will be performed as soon as possible.

`sa1100fb_set_var()` is not only called at initialization time, but can be used at any time, in order to modify the display mode at run time. The figure 6.1 illustrates the successive function calls occurring when `sa1100fb_set_var()` is called.

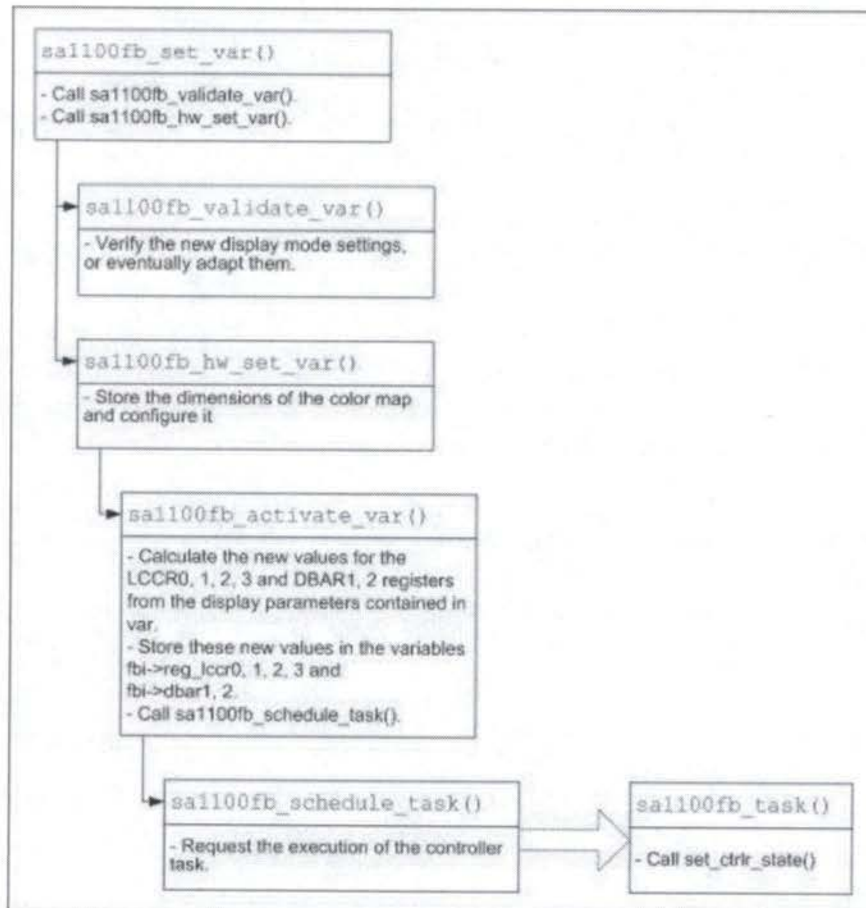

Figure 6.1: Defining the content of the LCD controller register.

The next two other functions allow to copy `fbi->fb.fix` and `fbi->fb.var` into a variable specified in argument.

```
static int sa1100fb_get_fix(struct fb_fix_screeninfo *fix, int con, struct
fb_info *info)
```

```
static int sa1100fb_get_var(struct fb_var_screeninfo *var, int con, struct
fb_info *info)
```

6.2.6 Registering the frame buffer driver

Registering the frame buffer driver at the kernel is simply performed by calling the `register_framebuffer()`[6].

6.2.7 Registering the device power management function

By default, the driver provides the possibility to set the display in "sleep" mode, which allows to reduce the power consumption. In sleep mode, the controller is disabled, the screen is power down and the backlight is turned off. The kernel must be informed that the device can be put to sleep mode. The driver signals this information by using the kernel function `pm_register()`. This function takes in argument a pointer to the power management function, which is here:

```
static int sa1100fb_pm_callback(struct pm_dev *pm_dev, pm_request_t req, void
*data).
```

The `req` argument can be PM_SUSPEND, which puts the device in the state "C_DISABLE", or PM_RESUME, which selects the state "C_ENABLE".

This function is generally used to reduce the power consumption during idle periods. But contrary to the dynamic power management techniques we have implemented, it does not allow to use the screen.

6.2.8 Updating the controller registers and enabling the LCD screen and the controller

```
static void set_ctrlr_state(struct sa1100fb_info *fbi, u_int state)
```

Now that the new controller registers content is defined in `fbi->reg_lccr0, 1, 2` and 3, the driver has to effectively copy these values into the controller registers, and finally enable the controller. This is performed by calling `set_ctrlr_state()`. As a matter of fact, this function puts the controller in the enabled or disabled state, according to the `state` argument.

[6]See the section 4.2.

state can take five possible values:

- **C_DISABLE:** The controller is disabled, the LCD screen is powered down and the backlight is turned off. In this state, the display system is in a low power consumption state.

- **C_ENABLE:** Select this state enables the controller, powers up the LCD screen and turns on the backlight. The display system is in a normal operating state.

- **C_REENABLE:** This option is selected when controller registers are re-programmed while the controller is enabled. It is used to change mode without completely shutting down the screen. The controller is simply disabled for the changes and directly enabled after.

- **C_DISABLE_CLKCHANGE:** Selecting this option disables the controller for clock change, but does not act on the LCD screen or on the backlight.

- **C_ENABLE_CLKCHANGE:** this option allows to enable the controller after clock change. It does not act on the LCD screen or on the backlight.

Enabling and disabling the controller is an easy task, which is performed by calling the `sa1100fb_enable_controller()` and `sa1100fb_disable_controller()` functions.

```
static void sa1100fb_enable_controller(struct sa1100fb_info *fbi)
static void sa1100fb_disable_controller(struct sa1100fb_info *fbi)
```

`sa1100fb_enable_controller()` writes the values contained in `fbi->reg_lccr0, 1, 2` and `3` into the LCCR registers. This is the only function that allows to update all the LCCR registers. It also writes 1 into the LEN field of the LCCR0 register, in order to enable the LCD controller.

`sa1100fb_enable_controller()` also updates the registers DBAR1 and DBAR2, by copying values stored in `fbi->dbar1` and `fbi->dbar2` into these two registers.

Powering up and down the LCD screen is performed by the `sa1100fb_power_up_lcd()` and `sa1100fb_power_up_lcd()` functions, whereas the backlight is controlled by `sa1100fb_backlight_on()` and `sa1100fb_backlight_off()`. These four functions simply call functions located somewhere else in the kernel. The figure 6.2 illustrates the working of the `set_ctrlr_state()` function.

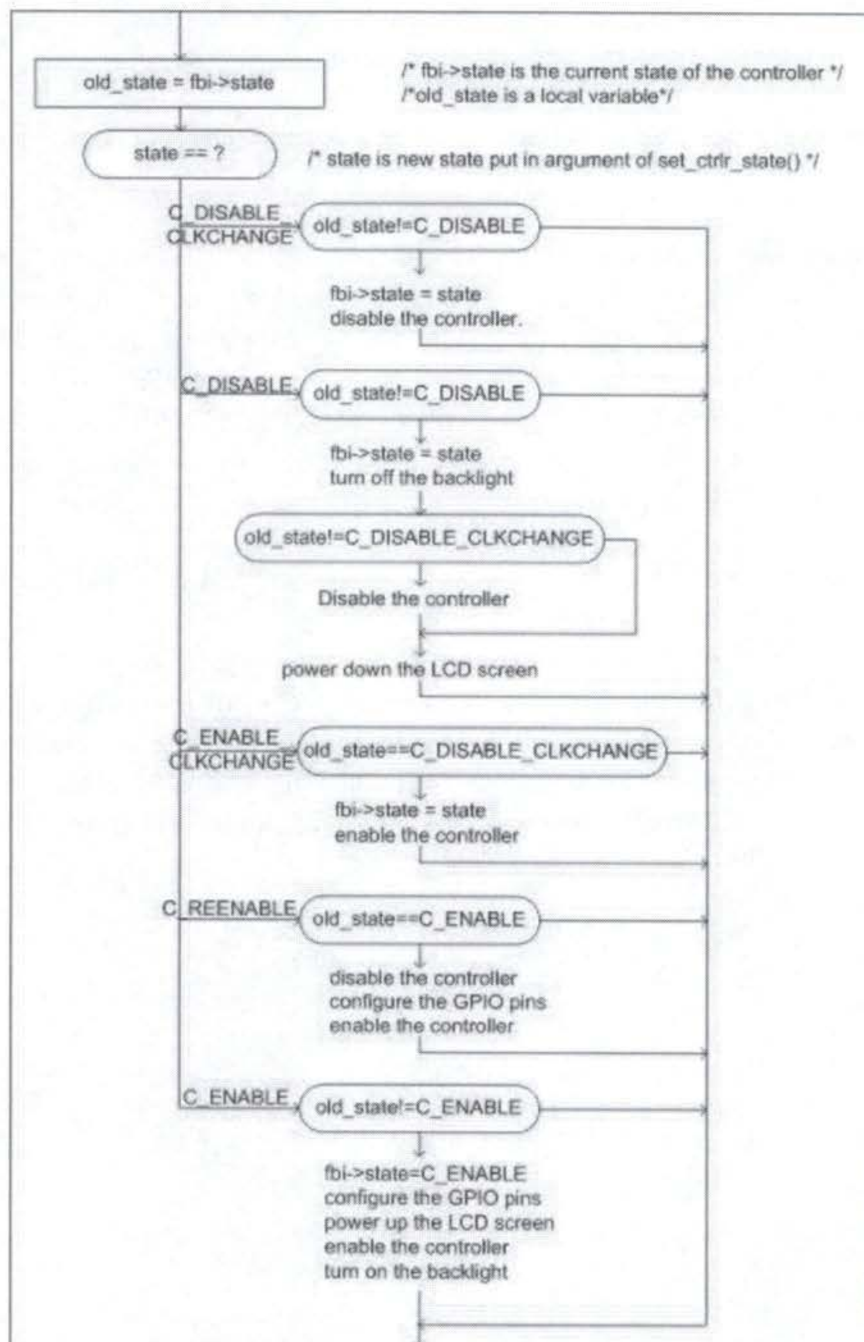

Figure 6.2: Updating the controller registers and enabling the LCD screen and the controller

6.3 Updating the palette

```
static int sa1100fb_get_cmap(struct fb_cmap *cmap, int kspc, int con, struct
fb_info *info)
sa1100fb_set_cmap(struct fb_cmap *cmap, int kspc, int con, struct fb_info *
info)
```

These two functions allow to access to the palette, one to read its content, the other one to edit it. The first function only makes a copy of the palette at the memory address "cmap", put in argument. The second function is used to replace the current palette by a new one, referenced by the argument "cmap". This function quickly calls `__do_set_cmap()`, which eventually defines a default palette and calls `fb_set_cmap()`[7] with a reference to the `sa1100fb_setcolreg()` function in argument. `fb_set_cmap()` contains a loop that applies `sa1100fb_setcolreg()` to each palette entry to update them.

```
static int sa1100fb_setcolreg(u_int regno, u_int red, u_int green, u_int blue,
u_int trans, struct fb_info *info)
```

This function is used to edit a palette entry. The `regno` argument represents the number of the updated palette entry, whereas the new color is composed of the combination of `red`, `green`, `blue` and `trans` (for transparency). The new palette entry is also composed according to the color mode stored in `info`. The update of the palette is depicted in the figure 6.3.

[7]`fb_set_cmap()` is implemented in /linux/drivers/video/fbcmap.c directory of the source code.

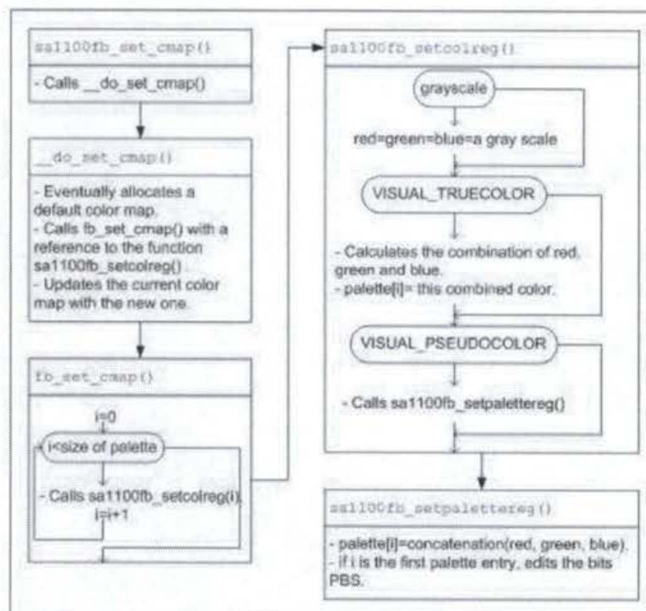

Figure 6.3: Updating the palette.

Chapter 7

Driver adaptation for power reduction of a TFT LCD display

This chapter provides the results of the work performed during my training period at the University of Bologna. The objective was to implement various techniques of power management in a frame LCD controller driver of the Linux operating system.

The first section presents the four considered software techniques to reduce the power consumption of a TFT LCD display: *Variable dot clock*, *Variable frame refresh*, *Liquid crystals orientation shift* and *Backlight shifting*. The second section describes the implementation of the "Variable dot clock" and "Variable frame refresh" techniques inside the LCD controller frame buffer driver sa1100fb.c; an outline of the implementation of the "Liquid crystals orientation shift" technique are also proposed. Experimental results for the "Variable dot clock" and "Variable frame refresh" technique are then presented and analyzed in the last section.

The above techniques were initially developed for power reduction of a LCD controller driver of the ECOS operating system and gave interesting results [1]. Our work has adapted the techniques for a TFT LCD display.

7.1 Software techniques for low-power TFT LCD displays

Defining a power management strategy on a LCD display system is a complex task, because the display does not experience explicit idle times. As a matter of fact, the display can be considered as idle when the user does not watch on it, a situation that the system cannot detect. For this reason, most policies concerning LCD displays are based on applications

requirements (for example, a text editor requires lower resolution than a video player) or on user interventions (reducing the performance when no key is pressed for a long time).

7.1.1 Variable dot clock

The first technique, called *Variable dot clock*, consists of decreasing the frequency of the display system dot clock, i.e. the pixel clock, used for the controller-screen communication. This is a typical example of clock gating. Reducing the pixel clock frequency slows down the pixels transmission between the LCD controller and the screen, and consequently decreases the refresh rate.

Thus, the pixel clock can be set to the lower possible frequency until flicker becomes excessive. This setting causes a reduction of the power consumption for every synchronized component of the display system. The pixel clock frequency can be configured according to the type of application: a simple text editor could use the lowest frequency, while higher frequencies should be used by programs where images change quickly (for instance a video player).

7.1.2 Variable frame refresh

The second technique is called *Variable frame refresh* and consists of repeatedly turning off and on the LCD controller. Turning off the LCD controller closes the communication to the display, so that no more image is transferred and the screen ceases to be refreshed. The power is saved because the whole display system is shut down. This method is viable for a LCD display because liquid crystals can maintain their orientation for a short time, so that the last displayed image before turning off the controller persists on the screen with a luminance decreasing progressively. When the controller is turned on, pixels transmission restarts and the screen is refreshed. The delay between turning off and on the controller must be regulated in order to keep a suitable refresh rate.

7.1.3 Liquid crystals orientation shift

The *Liquid crystals orientation shift* technique takes advantage of the fact that that user applications are often displayed in a window that occupies just a portion of the screen. In this case, the user does not need to see other portions with the same clarity. It is possible to modify the characteristics of useless portions of the screen, in order to reduce power consumption. The color generated by a liquid crystal depends on the charge it receives. In order to reduce power consumption, it is possible to provide a null charge to every liquid crystal located in an useless part of the screen. This is simply performed by affecting a

null value to the pixels corresponding to useless areas of the screen. The result is that each of the useless liquid crystals will adopt the default color of the screen (generally black or white).

7.1.4 Backlight shifting

The *Backlight shifting* technique is based on the observation that the display has a higher visibility in poorly illuminated environment because there is a high contrast between the luminosity of the display and the one of the environment. Decreasing the light intensity of the backlight of the screen allows to significatively reduce power consumption. It is possible to measure the luminosity of the environment with the help of a light sensor, and automatically regulate the backlight intensity to keep a sufficient contrast with respect to the environment. This method gives very significant power savings, due to the large consumption of the backlight with respect to the whole display system.

7.2 Implementation

7.2.1 Variable dot clock

The goal of the Variable dot clock technique is to reduce the pixel clock frequency. The pixel clock frequency can be dynamically set by modifying the PCD field of the LCD controller control register LCCR3[1]. We have seen in the chapter two that this field allows to configure the pixel clock frequency according to the formula $Pixel\ clock\ frequency = \frac{CCLK}{2\times(PCD+2)}$ where $CCLK$ is the frequency of a CPU clock.

Controlling the PCD field does not require to implement a new function inside the driver. Indeed, as explained in the chapter six, the driver disposes of a structure regrouping the display features modifiable by the user: the `struct fb_var_screeninfo` structure. This structure includes the field `pixclock`, which represents the current pixel clock period. The driver provides a function allowing to modify the content of the `struct fb_var_screeninfo` structure: the `sa1100fb_set_var()` function. When this function is called with a new content for the structure in argument, the driver updates the variable as well as the I/O registers content to make effective the new configuration. Thus, the pixel clock period can be easily modified, simply by calling `sa1100fb_set_var()` with a new pixel clock period. This period is then translated into a corresponding PCD value, which is afterwards written in the PCD field.

[1] See Appendix A.

The sa1100fb_set_var() function must be available for user applications. It is thus referenced in a variable whose type is fb_ops, the sa1100fb_ops variable, which references the functions exported to fbmem.c. Rather than implementing a specific system call in fbmem.c, the driver allows to use this function by means of an ioctl() system call. We have seen in the chapter four that the ioctl() system call provides a way to issue device-specific control commands, like reading or changing display parameters. User applications can thus perform one of these device-specific commands by executing an ioctl() system call with in argument the desired command and the associated argument.

The ioctl command allowing to modify the information stored in the fb_var_screeninfo structure is FBIOPUT_VSCREENINFO. The required argument is a variable whose type must be struct fb_var_screeninfo. Each field of this structure determines a feature of the display configuration. In order to modify the pixel clock period without changing other features of the current configuration, it is preferable to first copy the current content of the fb_var_screeninfo structure in a temporary variable of the user application, then replace the content of the pixclock field by the new pixel clock period, and finally recopy the temporary variable inside the fb_var_screeninfo structure by using the FBIOPUT_VSCREENINFO command. Copying the fb_var_screeninfo structure content can be performed with another ioctl command, which is FBIOGET_VSCREENINFO, with the target variable as argument. The figure 7.1 illustrates how the fb_var_screeninfo is updated.

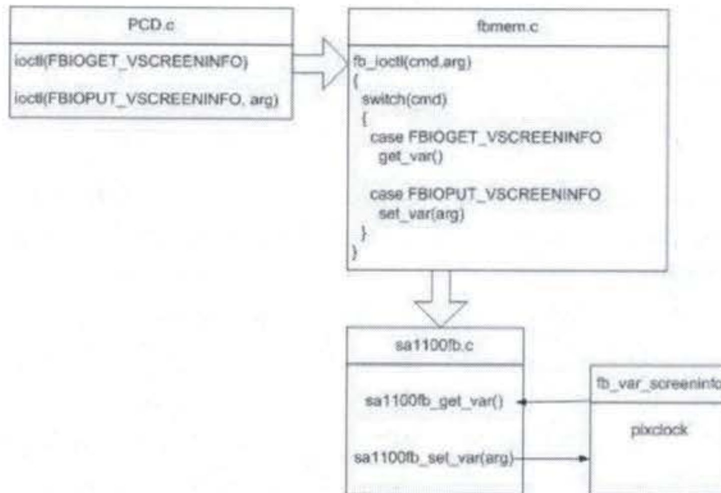

Figure 7.1: Updating the fb_var_screeninfo structure.

The program PCD.c[2] allows to replace the pixel clock period with a new value put in argument. PCD.c first opens the device file representing the frame buffer,i.e. /dev/fb, by using the open() system call, and then copies in a temporary variable (`my_var`) the content of the `fb_var_screeninfo` structure with help of the ioctl() system call with the `FBIOGET_VSCREENINFO` command. Afterwards, `my_var` is updated with the new pixel clock period and replaces the current `fb_var_screeninfo` structure content by means of the ioctl() system call with the `FBIOPUT_VSCREENINFO` command. Finally, the device file /dev/fb is closed.

7.2.2 Variable frame refresh

The Variable frame refresh technique is implemented with a loop which repeatedly switches off and on the LCD controller. Switching off and on the controller is performed by respectively writing 0 and 1 in the LEN field of LCCR0. The driver disposes of two functions to do that: `sa1100fb_disable_controller()` and `sa1100fb_enable_controller()`.

We have to take care of two problems occurring when an infinite loop is run in the kernel code. First, the loop does not share the processor with other processes, with the result that no applications can be executed. In order to solve that, a scheduling instruction has to be inserted inside the loop to liberate the processor for other processes. We propose to use the `schedule_timeout(long timeout)` instruction, which causes the process to sleep during at most `timeout` clock ticks. During this time, the scheduler is called, allowing all sleeping processes to run. This instruction has to be placed between the "disable" and the "enable" instructions in order to keep the controller off during a non negligible period. We also add this instruction after `sa1100fb_enable_controller()`.

The second problem is that the execution flow of the kernel is blocked by the loop. For this reason, executing the loop inside a task is preferable, so that the loop is executed separately from the kernel. We propose to insert the "loop task" into the scheduler queue because we do not have to launch its execution and because this queue allows a task to sleep.

Our loop task function is implemented in sa1100fb.c and is called `pm_len_function(struct pm_len_arg *my_arg)`. The task is initialized and inserted into the scheduler queue by calling the `sa1100fb_enable_pm_len(struct fb_delay *del, struct sa1100fb_info *fbi)` function and ends when `sa1100fb_disable_pm_len()` is called. These two last functions can be called by an application by means of an ioctl() system call. For that, we must declare these two functions in the `sa1100fb_ops` structure to export them to fbmem.c, and we add in fbmem.c two command names in the `fb_ioctl()` function, respectively

[2]See Appendix B.1.

FBIOPMLEN_ON and FBIOPMLEN_OFF.

LENON.c[3] is the application allowing to launch the task while LENOFF.c[4] allows to end the task. Their workings are similar to PCD.c.

7.2.3 Liquid crystals orientation shift

Shifting the liquid crystals orientation of the useless parts of the screen can be performed by implementing a function writing a null value to the corresponding pixels in the frame buffer. We first need a way to determine if a pixel is located in a useless part or not. This will be a function receiving as arguments the coordinates of the active window, i.e. the useful screen area. The active window is a rectangle defined by the XY-coordinates of its upper left corner with respect of the upper left corner of the screen, and by the length of its sides. The length unit is the pixel. As pixels are sequentially stored in the frame buffer memory, we need an algorithm to calculate if a pixel in memory owns to the active window or not. The figure 7.2 depicts the coordinates of the active window and considers four areas around this window: UP, DOWN, LEFT and RIGHT.

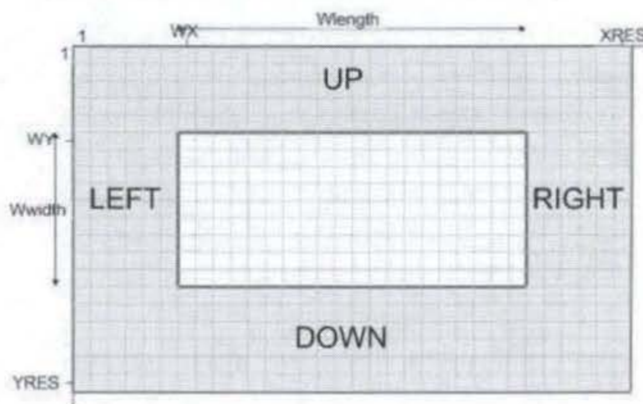

Figure 7.2: The active window coordinates.

Assume p the position of a pixel in the frame buffer memory ($1 \leq p \leq (XRES \times YRES)$). p is not located into the active window if $p \in UP$ or $p \in DOWN$ or $p \in LEFT$ or $p \in RIGHT$. p owns to one of these sets if one of the following equations is verified:

$$p \in UP \iff p \leq (WY - 1) \times XRES$$

[3]See Appendix B.2.

[4]See Appendix B.3.

$$
\begin{aligned}
p \in DOWN &\iff p > (WY + Wwidth - 1) \times XRES \\
p \in LEFT &\iff (p \; modulo \; XRES) < WX \\
p \in RIGHT &\iff (p \; modulo \; XRES) \geq (WX + Wlength)
\end{aligned}
$$

In sa1100fb.c, the address of the first pixel data is contained in the field `screen_cpu` of the `sa1100fb_info` structure. By default, a pixel data is represented with one byte. We can thus test each pixel by means of a loop and modify the value of each pixel located outside the active window.

7.3 Experimental results

In this section, we present the results obtained on the Assabet by applying the Variable dot clock and the Variable frame refresh techniques. For each technique we have measured the current consumption of the whole system. Power monitoring has been supported through an I/V converter realized with a 1-ohm resistor, allowing to measure the average value of the current and the power in a specified range of time. To do this, the voltage level on the resistor is measured with a data acquisition board controlled by a Labview program for measurement control. Measurements have been performed with 10000 samples at the frequency of 1000 samples/sec.

Flicker is observed with a small light image displayed on a black background. The flicker is more visible and disturbing when the screen becomes brighter.

7.3.1 Variable dot clock

The normal value of the pixel clock period is 171521 picoseconds. We first calculate the corresponding refresh rate (see the formulas in the section 2.3), based on the initial content of LCCR1 and LCCR2 set by the sa1100fb.c driver.

$$\text{Pixel clock frequency} = \frac{10^{12}}{171521\ ps} = 5830190 \text{ Hz}$$

$$\text{Pixel clock pulses per line} = 61 + 9 + 5 + 320 = 395$$

$$\text{Lines per frame} = 3 + 0 + 1 + 240 = 244$$

$$\text{Pixel clock pulses per frame} = 395 \times 244 = 96380$$

$$\text{Refresh rate} = \frac{5830190\ Hz}{96380} = 60.5 \text{ Hz}$$

The normal refresh rate of the display is thus 60.5 Hz. The table 7.1 presents the current consumption of the Assabet with the Variable dot clock technique. By increasing the pixel clock period step by step, we have collected the corresponding current values.

Pixel clock period (in ps)	Refresh rate (in Hz)	Medium consumption (in mA)	Power reduction (in %)	Flicker
171521	60.5	570	0%	no flicker
300000	34.6	554	2.8%	no flicker
500000	20.8	545	4.4%	no flicker
550000	18.9	542.5	4.8%	soft
700000	14.8	540.5	5.2%	visible
1000000	10.4	537.5	5.7%	visible
1500000	6.9	535.4	6.1%	high
2000000	5.1	534.1	6.3%	high
2080895	5.0	533.8	6.4%	last possible value
2080896	5.0	526	7.7%	unusable screen

Table 7.1: Assabet current consumption with Variable dot clock.

We can find out that the Variable dot clock technique allows to reduce the power consumption until 36.2 Amps, corresponding to a reduction of 6.4% of the consumption of the whole system. Moreover, the display refresh rate can be reduced to 20 Hertz without significative lost of visibility. Such a refresh rate corresponds to a reduction of 4.4% of the power consumption.

7.3.2 Variable frame refresh

Running the loop task increases the total power consumption with respect to the idle state. Measurements should be performed while an application is running to get a better view on the effectiveness of this technique. As a matter of fact, measurements were performed with an adapted version of the loop task. The reason comes from the working of the `schedule_timeout()` instruction. When `schedule_timeout()` is executed, the scheduler is called, allowing all sleeping processes to run. However, if few processes are in progress, the scheduler reallocates the processor to the process running the loop task before `timeout` clock ticks elapsed. Hence, we cannot perform measurements with various disabled or enabled delays. For this reason, we were constraint to replace `schedule_timeout()` by the `mdelay(unsigned long msecs)` instruction. The `mdelay()` instruction delays the execution during exactly `msecs` milliseconds, by means of an active loop.

```
while (my_stop==0) {
   sa1100fb_disable_controller(my_arg->info);
   mdelay(my_arg->delay_dis);
   sa1100fb_enable_controller(my_arg->info);
   mdelay(my_arg->delay_en);
}
```

Executing a loop keeps the processor busy and increases its power consumption. However, we can imagine that this active loop occupies the processor as if an application run. For this reason, the following measurements must be considered with caution. As a matter of fact, they should be done in more realistic conditions.

When the loop task only contains the `mdelay(100)` instruction inside the loop (i.e. without any instructions to disable and enable the controller), the medium consumption of the Assabet is 693 mA. We compare the measurements with this value in the table 7.2.

Tests have been subdivided in two parts. First, we have varied the delay when the LCD controller is disabled by keeping constant the delay when it is enabled. Afterwards, we vary the delay when the controller is enabled without changing the delay when it is disabled.

We observe that the Variable frame refresh technique allows to save power when the processor is fully used. In the first set of measurements, the flicker quickly becomes bad. In order to keep an usable display, the period when the controller is disabled should not be higher than 40 ms. We also observe that power consumption increases when the running duration of the `mdelay()` instructions increases. In the second set of measurements, we

Disable delay (in ms)	Enable delay (in ms)	Medium consumption (in mA)	Power reduction (in %)	Flicker
10	0	624	10.0	visible
20	0	633	8.7	visible
30	0	638	7.9	visible
40	0	641	7.5	visible
50	0	643	7.2	high
100	0	646	6.8	high
10	5	624	10.0	visible
10	10	646	6.8	soft to visible
10	20	656	5.3	soft
10	30	668	3.6	soft
10	40	674	2.7	no flicker
10	50	686	1.0	no flicker
10	100	689	0.6	no flicker

Table 7.2: Assabet current consumption with Variable frame refresh.

can reduce the power consumption up to 10%, while keeping an acceptable visibility of the displayed image.

Conclusion

The rapidly increasing demand for portable systems encourages the development of more and more sophisticated devices. Designing a portable system requires finding a trade-off between delivering high performance and maintaining a limited consumption of electrical power. Designers and researchers have developed numerous technological innovations and optimization techniques to reduce power consumption. One of the most successful approaches to power minimization is Dynamic Power Management.

In this dissertation we have presented several power reduction techniques for a TFT LCD display and shown how to integrate them in a Linux device driver. The two basic techniques at the component level are *Variable dot clock* (a type of clock gating) and *Variable frame refresh* (a type of power off), which we have developed especially for our research on a Personal Digital Assistant "Assabet" using the Linux operating system. We have also explained how power can be managed at system level by means of a power manager.

Presenting these developments has required explanation about the notions of device file and device driver in Linux, the structure of a char driver, the interrupts detection and the task queue, in order to explain how I/O devices are managed by the operating system Linux.

Our contribution is twofold. First, we have developed a description of the main data structure and functions of the frame buffer LCD controller driver sa1100fb.c and indicated how the display configuration can be modified. Secondly, we have implemented two power reduction techniques in this driver, the *Variable dot clock* and *Variable frame refresh* techniques. As shown by the observed measurements, we were able to reduce the Assabet power consumption by up to 6.4% with the *Variable dot clock* technique and up to 10% with the *Variable frame refresh* technique. Finally, we have outlined a solution for the *Liquid crystal orientation shift* technique.

The project of managing power consumption for a TFT LCD display has further issues to explore, since there are two other techniques which could be implemented in the Linux driver. We hope that this work will lead to new enhancements and developments of the

DPM techniques.

In conclusion, this dissertation demonstrates that a software adaptation in the operating system, without modifying the hardware, can lead to an appreciable reduction of the power consumption in a portable system. Therefore the DPM techniques can be applied to improve the lifetime of existing portable systems batteries.

Bibliography

[1] F. Gatti, A. Acquaviva, L. Benini, and B. Riccó. Low Power Control Techniques for TFT LCD Displays. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems CASES'02*, October 2002.

[2] Y.-H. Lu, L. Benini, and G. De Micheli. Requester-Aware Power Reduction. In *International Symposium on System Synthesis*, pages 18–23. Stanford University, September 2000.

[3] L. Benini, G. Castelli, A. Macii, E. Macii, and R. Scarsi. Battery-Driven Dynamic Power Management of Portable Systems. *International Symposium on System Synthesis*, pages 25–33, September 2000.

[4] L. Benini, G. Castelli, A. Macii, and R. Scarsi. Battery-Driven Dynamic Power Management. *IEEE Design and Test of Computers special issue on Dynamic Power Management of Electronic Systems*, pages 53–60, March/April 2001.

[5] Y.-H. Lu and G. De Micheli. Adaptive Hard Disk Power Management on Personal Computers. In *Proceedings of the IEEE Great Lakes Symposium*, pages 50–53, March 1999.

[6] D. Bertozzi, A. Raghunathan, L. Benini, and S. Ravi. Transport Protocol Optimization for Energy Efficient Wireless Embedded Systems. *Proceedings of the Design, Automation and Test in Europe Conference Exhibition (DATE'03)*, 2003.

[7] SA-1100 Microprocessor Technical Reference Manual. Intel Corp., September 1998. http://www.intel.com.

[8] L. Benini and G. De Micheli. *Dynamic Power Management: Design Techniques and CAD Tools.* Kluwer, 1997.

[9] A. Rubini and J. Corbet. *Linux Device Drivers, 2nd edition.* O'Reilly, 2002.

[10] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly, 2001.

[11] M. Barr. *Programming Embedded Systems in C and C++*. O'Reilly, 1999.

[12] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals, 2nd edition*. Addison-Wesley, 1998.

[13] I. Choi, H. Shim, and N. Chang. Low Power Color TFT LCD Display for Hand-Held Embedded Systems. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'02*, August 2002.

[14] L. Benini, A. Bogliolo, and G. De Micheli. A Survey of Design Techniques for System-Level Dynamic Power Management. *IEEE Transactions on VLSI Systems, Vol.8, No. 3*, June 2000.

[15] L. Benini, A. Bogliolo, and G. De Micheli. System-Level Dynamic Power Management. *VOLTA-99: IEEE Alessandro Volta Memorial Workshop on Low-Power Design*, pages 18–23, March 1999.

[16] IBM and Monta Vista Software. Dynamic Power Management for Embedded Systems, Version 1.1. http://www.research.ibm.com/arl/projects/papers/DPM_V1.1.pdf, 2002.

[17] T. Simunic, A. Acquaviva, L. Benini, and G. De Micheli. Dynamic Voltage Scaling and Power Management for Portable Systems. *Proceedings of the 38th Design Automation Conference DAC'01*, 2001.

[18] P. Pillai and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. *Proceedings of the 18th Symposium on Operating Systems Principles SOSP'01*, October 2001.

[19] T. Burd, T. Pering, A. Stratakos, and R. Brodersen. A Dynamic Voltage Scaled Microprocessor System. *Proceedings of the International Solid-State Circuits Conference*, 2000.

[20] T. Pering and R. Brodersen. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. *Proceedings of the International Symposium on Low-Power Electronic and Design ISLPED'98*, June 1998.

[21] L. Benini, A. Macii, E. Macii, and M. Poncino. Selective instruction compression for memory energy reduction in embedded systems. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'99*, August 1999.

[22] P. Greenawalt. Modeling Power Management for Hard Disks. *International Workshop on Modeling, Analysis and Simulation of Computer ans Telecommunication Systems*, pages 62–6, 1994.

[23] F. Matià. Writing a Linux Driver, April 1998. http://www.linuxjournal.com/.

[24] The ARM Linux Project web site: http://www.arm.linux.org.uk.

[25] The Handelds.org web site: http://www.handelds.org.

[26] The ASSABET web page: http://www-2.cs.cmu.edu/~wearable/software/assabet.html.

Table of acronyms

ACPI	Advanced Configuration and Power Interface
API	Application Programming Interface
BFW	Beginning-of-Frame line clock Wait
BLW	Beginning-of-Line pixel clock Wait
CPU	Central Processing Unit
DBAR	DMA Base Address Register
DCAR	DMA Current Address Register
DMA	Direct Memory Access
DPM	Dynamic Power Management
DRAM	Dynamic Random Access Memory
DVS	Dynamic Voltage Scaling
EFW	End-of-Frame line clock Wait
ELW	End-of-Line pixel clock Wait
EOF	End of File
FAT	File Association Table
FIFO	First In First Out
GNU	GNU is Not Unix
GPL	GNU Public Licence
HSW	Horizontal Sync pulse Width
ICIP	Interrupt Controller IRQ Pending
IDT	Interrupt Descriptor Table (interrupt)
IDT	Instruction Decompression Table (memory)
IRQ	Interrupt ReQuest
ISR	Interrupt Service Routine
I/O	Input / Output
LAN	Local Area Network
LCCR	LCD Controller Control Register
LCD	Liquid Crystal Display
LCSR	LCD Controller Status Register

LPP	Lines per Panel
MAC	Medium Access Control
MPEG	Moving Picture Experts Group
NTFS	Windows NT FileSystem
OS	Operating System
PBS	Palette Bit Size
PPL	Pixels Per Line
PC	Personal Computer
PDA	Personal Digital Assistant
PM	Power Manager
PMC	Power Manageable Component
RAM	Random Access Memory
SCSI	Small Computer System Interface
SWC	Silly Window Syndrome
TCP/IP	Transmission Control Protocol / Internet Protocol
TFT	Thin-Film Transistor
VSW	Vertical Sync pulse Width

Glossary

Advanced Configuration and Power Interface (ACPI): general and OS-independent specification defining the interface between the operating system and an ACPI-compliant hardware. This interface can be used both for hardware configuration and power management.

Block: (fixed) size of transfer from or to a block device.

Block device: Device that can host a filesystem, accessible only by block of bytes.

Bottom half part: Routine required by the top half part, but executed later.

Break-even time: Minimum idle time amortizing the state transition costs.

Char (or character) device: Device that can be accessed in the same way as a stream of bytes, where bytes are addressed sequentially.

Clock gating: Power reduction technique consisting of reducing the frequency of the component clock.

Color map: See **Palette**.

Context: The current state of the registers and flags of the processor.

Control register: Register of a controller allowing to send commands to the device.

Device driver: Part of the operating system that manages communication with a specific device.

Device file: File allowing user programs to access hardware devices of the system.

Direct Memory Access (DMA): Mechanism providing a device controller the ability to transfer data directly to or from the memory without involving the processor.

Directory: Set of files regrouped in the filesystem.

Dynamic Power Management (DPM): Design methodology that dynamically reconfigures an electronic system to provide the requested services and performance levels with a minimum number of active components or a minimum load of such components. Dynamic power management encompasses a set of techniques that achieve energy-efficient computation by selectively turning off (or reducing the performance of) system components when they are idle (or partially unexploited).

Dynamic Random Access Memory (DRAM): Memory that contains the instructions and data of a program while it is running, which allows faster access than accessing a magnetic disk.

Exception: Electrical signal produced in the CPU control unit when special instructions are executed or when an error or an abnormal situation occurs.

Filesystem: Set of rules that define how the files are organized and manipulated.

Flag: Field of a status register being set and cancelled by the hardware.

Frame: Set of pixels corresponding to an entire image on the screen.

Frame Buffer: Memory area used to contain the pixels to display on the screen.

Frame clock: Signal occurring at the end of a frame transmission.

Idle: A component or a system is in an idle state if it has no request to serve.

Input register: Register of a controller allowing to fetch data from the device.

(Hardware) interrupt: Electrical signal generated by an hardware device, such as a timer or an I/O device, at arbitrary time with respect to the CPU clock.

Interrupt controller: Hardware circuit that warns the processor when an interrupt occurs.

Interrupt Descriptor Table (IDT): Table containing the address of each interrupt handler.

Interrupt handler: Function called when an interrupt occurs and capable to handle the interrupt.

Interrupt Request (IRQ): Lines transmitting maskable interrupts.

Interrupt Service Routines (ISR): Function related to a single device sharing an IRQ line, and implements the task of the interrupt handler concerning the device.

Interrupt signal: Event that alters the sequence of instructions executed by the processor.

ioctl system call: System call allowing to issue device-specific control commands, like reading or changing device parameters, reading or modifying the palette,...

I/O port or **I/O register**: Memory location that is a part of a device and used for the communication with a device.

Kernel: Main part of the Linux operating system.

Kernel control path: Sequence of instructions executed when an interrupt occurs and that launches the execution of the appropriate interrupt handler.

LCD controller: Electronic device allowing the control a peripheral.

Line clock: Signal occurring at the end of the transmission of a pixel line.

Loopback: Interface allowing to connect to himself exactly as to another host.

Major number: Number ranking from 1 to 255 that identifies the device type.

Markov chain: Probability function of the evolution of the system state, according to the state at present time.

Minor number: Number ranking from 1 to 255 used by the driver to identify a specific device among a group of devices that share the same major number.

Module: Object file whose code can be dynamically linked to (and unlinked from) the kernel at any time during the runtime. A module regroups a set of additional functions to be realized by a filesystem, a device driver, a protocol or other features.

Network interface: Interface in charge of sending and receiving data packets, without knowing how individual transactions map to the actual packets being transmitted.

Output register: Register of a controller allowing to send data to the device.

Palette: Set of data representing the possible colors a pixel of a frame can takes.

Palette entry: Representation of a possible color in the palette.

Pipe: Special file used for interprocess communication.

Pixel: The smallest monochromatic element of an image.

Pixel clock: Regular signal determining the frequency of pixels transmission on the data pins. This signal is emitted on a specific pin simply called pixel clock pin.

Pixel data: Color of a pixel of the frame.

Policy: Algorithm selectively shutting down idle resources and based on the observation of present and past workload and operating conditions.

Power Manageable Component (PMC): Component characterized by multiple states of operation.

Power Manager (PM): System part (hardware or software) that performs DPM at system level.

Power State Machine (PSM): State machine representing the power states a component or a system accepts and the transition costs between states.

Process: Instance of a program in execution.

Program counter: A register in the processor that contains the address of the next instruction to be executed. Also known as a instruction pointer.

Regular file: Set of jointly stored data.

refresh rate: Amount of times a display's image is repainted per second.

Run: A component or a system is in a run state if it is serving a request.

Scheduler: Part of the operating system that decides which task or process to run next. This decision is based on readiness of each process, their relative priorities, and the specific scheduling algorithm implemented.

Sector: Set of 512 adjacent bytes on a disk.

Session: Time interval when requests frequently occur.

Sleep: A component or a system in a sleep state is disabled in order to consume less power.

Socket: Software mechanism allowing programs to communicate with a local or remote application.

Software interrupt: Signal is generated by a special software instruction.

Status bit: Field of a status register being set by the hardware and cancelled by the software.

Status register: Register of a controller allowing to be notified about the device state.

Symbolic link: Short file containing the access path to another file.

System call: Special instruction that transfers control from user mode to a dedicated location in kernel code space.

Task queue: Linked list of tasks where tasks are stored in order to delay their execution to a determined-system time.

Top half part: Code that the kernel executes right away to answer to an interrupt.

Transition costs: Costs associated with a state change. These costs can be a delay, a performance reduction or a power overhead.

X Window System: Graphic interface system used by Linux and other Unix operating systems.

Appendix

Appendix A

LCD Controller Registers

A.1 LCD Controller Control Register 0 (LCCR0)

bits	field name	Description
0	LEN	LCD controller enable. 0 - LCD controller disabled. 1 - LCD controller enabled. Note: all the registers must be initialized before enabling the LCD controller.
1	CMS	Color/Monochrome mode select 0 - Color operation mode enabled. 1 - Monochrome operation mode enabled.
2	SDS	Single-/Dual-panel display select 0 - Single-panel display enabled. 1 - Dual-panel display enabled.
3	LDM	LCD disable done interrupt mask. 0 - LCD disable done condition generates an interrupt. 1 - LCD disable done condition does not generate an interrupt.
4	BAM	Base address update interrupt mask. 0 - Base address update condition generates interrupt. 1 - Base address update condition does not generate an interrupt.
5	ERM	Error interrupt mask. 0 - Bus error and FIFO over/underrun errors generate an interrupt. 1 - Bus error and FIFO over/underrun errors do not generate an interrupt.
6	-	reserved
7	PAS	Passive/active display select. 0 - Passive or STN display operation enabled. 1 - Active or TFT display operation enabled.

bits	field name	Description
8	BLE	Big/little endian select. 0 - Little endian byte order is selected. 1 - Big endian byte order is selected.
9	DPD	Double-pixel data pin mode. 0 - In single-panel monochrome operation, four pixels are sent each pixel clock. 1 - In single-panel monochrome operation, eight pixels are sent each pixel clock.
11..10	-	reserved.
19..12	PDD	Palette DMA request delay. Value from 0 to 255 used to specify the number of clocks between each data transfer while the palette is charged.
31..20	-	reserved.

A.2 LCD Controller Control Register 1 (LCCR1)

bits	field name	Description
9..0	PPL	Pixel per line Value from 1 to 1024 used to specify the number of pixels contained within each line on the LCD display.
15..10	HSW	Horizontal sync pulse width. Value from 1 to 64 used to specify number of pixel clock periods to pulse the line clock at the end of each line.
23..16	ELW	End-of-line pixel clock wait count. Value from 1 to 256 used to specify number of pixel clock periods to add to the end of a line transmission before line clock is asserted.
31..24	BLW	Beginning-of-line pixel clock wait count. Value from 1 to 256 used to specify number of pixel clock periods to add to the beginning of a line transmission before the first set of pixels is output to the display.

A.3 LCD Controller Control Register 2 (LCCR2)

bits	field name	Description
9..0	LPP	Line per panel Value from 1 to 1024 used to specify number of lines per panel (i.e. per frame).
15..10	VSW	Vertical sync pulse width. Value from 1 to 64 used to specify number of extra line clock periods to insert after the end-of-frame.

bits	field name	Description
23..16	EFW	End-of-frame line clock wait count. Value from 0 to 255 used to specify number of line clock periods to add to the end of each frame.
31..24	BFW	Beginning-of-frame line clock wait count. value from 0 to 255 used to specify number of line clock periods to add to the beginning of a frame before the first set of pixels is output to the display.

A.4 LCD Controller Control Register 3 (LCCR3)

bits	field name	Description
7..0	PCD	Pixel clock divisor. Value from 0 to 255 used to specify the frequency of the pixel clock based on the CPU clock (CCLK) frequency.
15..8	ACB	AC bias pin frequency. Value from 1 to 256 used to specify the number of line clocks to count before switching the ac bias pin in passive mode.
19..16	API	AC bias pin transitions per interrupt. Value from 0 to 15 used to specify the number of ac bias pin transitions to count before setting the line count status (ABC) bit, signalling an interrupt request.
20	VSP	Vertical sync polarity. 0 L_FCLK pin is active high and inactive low. 1 L_FCLK pin is active low and inactive high.
21	HSP	Horizontal sync polarity. 0 L_LCLK pin is active high and inactive low. 1 L_LCLK pin is active low and inactive high.
22	PCP	Pixel clock polarity. 0 Data is driven on the LCDs data pins on the rising edge of L_PCLK. 1 Data is driven on the LCDs data pins on the falling edge of L_PCLK.
23	OEP	Output enable polarity. 0 L_BIAS pin is active high and inactive low in active display mode and parallel data input mode. 1 L_BIAS pin is active low and inactive high in active display mode and parallel data input mode.
31..24	-	Reserved

A.5 DMA Channel 1 Base Address Register

bits	field name	Description
31..0	DBAR1	DMA channel 1 base address pointer. Used to specify the base address of the frame buffer memory.

A.6 DMA Channel 1 Current Address Register

bits	field name	Description
31..0	DCAR1	DMA channel 1 current address pointer. Read-only register. Continuously reflects the current address that DMA channel 1 is transferring from or will use in the next transfer.

A.7 DMA Channel 2 Base Address Register

bits	field name	Description
31..0	DBAR2	DMA channel 2 Base address pointer. Used to specify the base address of the frame buffer memory for the lower half of the display in dual-panel operation.

A.8 DMA Channel 2 Current Address Register

bits	field name	Description
31..0	DCAR2	DMA channel 2 Current address pointer. Read-only register. Continuously reflects the current address that DMA channel 2 is transferring from or will use in the next transfer.

A.9 LCD Controller Status Register

bits	field name	Description
0	LDD	LCD disable done flag. 0 LCD has not been disabled and the last active frame completed.
		1 LCD has been disabled and the last active frame has just completed.
1	BAU	Base address update flag (read-only). 0 Base address has been written and has not yet been transferred to the current address register. 1 Base address has been transferred to the current address register, triggered either by enabling the LCD or when the current address pointer equals the end address value calculated by the LCD.
2	BER	Bus error status. 0 DMA has not attempted an access to reserved/nonexistent memory space. 1 DMA has attempted an access to a reserved/nonexistent location in external memory.
3	ABC	AC bias count status. 0 AC bias transition counter has not decremented to zero, or API is programmed to all zeros. 1 AC bias transition counter has decremented to zero, indicating that the L_BIAS pin has transitioned the number of times specified by the API control bit field.
4	IOL	Input FIFO overrun lower panel status. 0 Input FIFO for the lower panel display has not overrun. 1 DMA attempted to place data into the input FIFO for the lower panel after it has been filled.
5	IUL	Input FIFO underrun lower panel status. 0 Input FIFO for the lower panel display has not underrun. 1 DMA not supplying data to input FIFO for the lower panel at a sufficient rate.
6	IOU	Input FIFO overrun upper panel status. 0 - Input FIFO for the upper or whole panel display has not overrun. 1 - DMA attempted to place data into the input FIFO for the upper or whole panel after it has been filled.
7	IUU	Input FIFO underrun upper panel status. 0 Input FIFO for the upper or whole panel display has not underrun. 1 DMA not supplying data to input FIFO for the upper or whole panel at a sufficient rate.
8	OOL	Output FIFO overrun lower panel status. 0 Output FIFO for the lower panel display has not overrun. 1 Dither logic attempted to place data into the output FIFO for the lower panel after it had been filled.

bits	field name	Description
9	OUL	Output FIFO underrun lower panel status. 0 Output FIFO for the lower panel display has not underrun. 1 LCD dither logic not supplying data to output FIFO for the lower panel at a sufficient rate.
10	OOU	Output FIFO overrun upper panel status. 0 Output FIFO for the upper or whole panel display has not overrun. 1 Dither logic attempted to place data into the output FIFO for the upper or whole panel after it had been filled.
11	OUU	Output FIFO underrun upper panel status. 0 Output FIFO for the upper or whole panel display has not underrun. 1 LCD dither logic not supplying data to output FIFO for the upper or whole panel at a sufficient rate.
31..12		Reserved

Appendix B

Applications

B.1 PCD.c

```
#include <fcntl.h>
#include <asm/io.h>
#include <stdio.h>
#include <linux/fb.h>
#include <linux/ioctl.h>

struct fb_var_screeninfo *my_var;
int id;
int t1;
long new_val;

main(int argc, char *argv[]) {
    if (argc != 2)
        new_val = 171521;
    else
        new_val = atol(argv[1]);
    my_var = (struct fb_var_screeninfo *)malloc(sizeof(struct fb_var_screeninfo));
    id = open("/dev/fb",O_RDONLY);
    t1 = ioctl(id,FBIOGET_VSCREENINFO,my_var);
    close(id);

    /*new value PCD*/
    my_var->pixclock = new_val;/*normal value is 171521*/

    id = open("/dev/fb",O_RDWR);
    t1 = ioctl(id,FBIOPUT_VSCREENINFO,my_var);

    close(id);
}
```

B.2 LENON.c

```
#include <fcntl.h>
#include <unistd.h>
#include <asm/io.h>
#include <stdio.h>
#include <linux/fb.h>
#include <linux/ioctl.h>
#include <math.h>

int id;
int t1;
struct fb_delay {
    int dis;
    int en;
};
struct fb_delay delay_struct;

main(int argc,char *argv[]) {

    if (argc != 3) {
        delay_struct.dis = 0;
        delay_struct.en = 0;
    }
    else {
        delay_struct.dis = atoi(argv[1]);
        delay_struct.en = atoi(argv[2]);
    }
    id = open("/dev/fb",O_RDWR);

    t1 = ioctl(id,0x4619/*FBIOPMLEN_ON*/,&delay_struct);

    close(id);
}
```

B.3 LENOFF.c

```
#include <fcntl.h>
#include <unistd.h>
#include <asm/io.h>
#include <stdio.h>
#include <linux/fb.h>
#include <linux/ioctl.h>
#include <math.h>

int id;
int t1;

main() {

    id = open("/dev/fb",O_RDWR);

    t1 = ioctl(id,0x461A/*FBIOPMLEN_OFF*/,NULL);

    close(id);
}
```

Appendix C

fbmem.c

```
/*
 *  linux/drivers/video/fbmem.c
 *
 *  Copyright (C) 1994 Martin Schaller
 *
 *  2001 - Documented with DocBook
 *  - Brad Douglas <brad@neruo.com>
 *
 * This file is subject to the terms and conditions of the GNU General Public
 * License.  See the file COPYING in the main directory of this archive
 * for more details.
 */
#include <linux/config.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/errno.h>
#include <linux/sched.h>
#include <linux/smp_lock.h>
#include <linux/kernel.h>
#include <linux/major.h>
#include <linux/slab.h>
#include <linux/mman.h>
#include <linux/tty.h>
#include <linux/console.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#ifdef CONFIG_KMOD
#include <linux/kmod.h>
#endif
#include <linux/devfs_fs_kernel.h>
#if defined(__mc68000__) || defined(CONFIG_APUS)
#include <asm/setup.h>
```

```
#endif
#include <asm/io.h>
#include <asm/uaccess.h>
#include <asm/page.h>
#include <asm/pgtable.h>
#include <linux/fb.h>
#include <video/fbcon.h>
    /*
     *  Frame buffer device initialization and setup routines
     */
extern int acornfb_init(void);
extern int acornfb_setup(char*);
extern int amifb_init(void);
extern int amifb_setup(char*);
extern int atafb_init(void);
extern int atafb_setup(char*);
extern int macfb_init(void);
extern int macfb_setup(char*);
extern int cyberfb_init(void);
extern int cyberfb_setup(char*);
extern int pm2fb_init(void);
extern int pm2fb_setup(char*);
extern int clps711xfb_init(void);
extern int cyber2000fb_init(void);
extern int retz3fb_init(void);
extern int retz3fb_setup(char*);
extern int clgenfb_init(void);
extern int clgenfb_setup(char*);
extern int hitfb_init(void);
extern int vfb_init(void);
extern int vfb_setup(char*);
extern int offb_init(void);
extern int atyfb_init(void);
extern int atyfb_setup(char*);
extern int aty128fb_init(void);
extern int aty128fb_setup(char*);
extern int igafb_init(void);
extern int igafb_setup(char*);
extern int imsttfb_init(void);
extern int imsttfb_setup(char*);
extern int dnfb_init(void);
extern int tgafb_init(void);
extern int tgafb_setup(char*);
extern int virgefb_init(void);
extern int virgefb_setup(char*);
extern int resolver_video_setup(char*);
extern int s3triofb_init(void);
```

```
extern int vesafb_init(void);
extern int vesafb_setup(char*);
extern int vga16fb_init(void);
extern int vga16fb_setup(char*);
extern int hgafb_init(void);
extern int hgafb_setup(char*);
extern int matroxfb_init(void);
extern int matroxfb_setup(char*);
extern int hpfb_init(void);
extern int sbusfb_init(void);
extern int sbusfb_setup(char*);
extern int control_init(void);
extern int control_setup(char*);
extern int platinum_init(void);
extern int platinum_setup(char*);
extern int valkyriefb_init(void);
extern int valkyriefb_setup(char*);
extern int chips_init(void);
extern int g364fb_init(void);
extern int sa1100fb_init(void);
extern int epson1356fb_init(void);
extern int mqfb_init(void);
extern int mqfb_setup(char*);
extern int fm2fb_init(void);
extern int fm2fb_setup(char*);
extern int q40fb_init(void);
extern int sun3fb_init(void);
extern int sun3fb_setup(char *);
extern int sgivwfb_init(void);
extern int sgivwfb_setup(char*);
extern int rivafb_init(void);
extern int rivafb_setup(char*);
extern int tdfxfb_init(void);
extern int tdfxfb_setup(char*);
extern int sisfb_init(void);
extern int sisfb_setup(char*);
extern int stifb_init(void);
extern int stifb_setup(char*);
extern int radeonfb_init(void);
extern int radeonfb_setup(char*);
extern int e1355fb_init(void);
extern int e1355fb_setup(char*);
extern int dcfb_init(void);
extern int anakinfb_init(void);

static struct {
    const char *name;
```

```
    int (*init)(void);
    int (*setup)(char*);
} fb_drivers[] __initdata = {

#ifdef CONFIG_FB_SBUS
    /*
     * Sbusfb must be initialized _before_ other frame buffer devices that
     * use PCI probing
     */
    { "sbus", sbusfb_init, sbusfb_setup },
#endif
    /*
     * Chipset specific drivers that use resource management
     */
#ifdef CONFIG_FB_RETINAZ3
    { "retz3", retz3fb_init, retz3fb_setup },
#endif
#ifdef CONFIG_FB_AMIGA
    { "amifb", amifb_init, amifb_setup },
#endif
#ifdef CONFIG_FB_CLPS711X
    { "clps711xfb", clps711xfb_init, NULL },
#endif
#ifdef CONFIG_FB_CYBER
    { "cyber", cyberfb_init, cyberfb_setup },
#endif
#ifdef CONFIG_FB_CYBER2000
    { "cyber2000", cyber2000fb_init, NULL },
#endif
#ifdef CONFIG_FB_PM2
    { "pm2fb", pm2fb_init, pm2fb_setup },
#endif
#ifdef CONFIG_FB_CLGEN
    { "clgen", clgenfb_init, clgenfb_setup },
#endif
#ifdef CONFIG_FB_ATY
    { "atyfb", atyfb_init, atyfb_setup },
#endif
#ifdef CONFIG_FB_MATROX
    { "matrox", matroxfb_init, matroxfb_setup },
#endif
#ifdef CONFIG_FB_ATY128
    { "aty128fb", aty128fb_init, aty128fb_setup },
#endif
#ifdef CONFIG_FB_VIRGE
    { "virge", virgefb_init, virgefb_setup },
#endif
```

```
#ifdef CONFIG_FB_RIVA
    { "riva", rivafb_init, rivafb_setup },
#endif
#ifdef CONFIG_FB_RADEON
    { "radeon", radeonfb_init, radeonfb_setup },
#endif
#ifdef CONFIG_FB_CONTROL
    { "controlfb", control_init, control_setup },
#endif
#ifdef CONFIG_FB_PLATINUM
    { "platinumfb", platinum_init, platinum_setup },
#endif
#ifdef CONFIG_FB_VALKYRIE
    { "valkyriefb", valkyriefb_init, valkyriefb_setup },
#endif
#ifdef CONFIG_FB_CT65550
    { "chipsfb", chips_init, NULL },
#endif
#ifdef CONFIG_FB_IMSTT
    { "imsttfb", imsttfb_init, imsttfb_setup },
#endif
#ifdef CONFIG_FB_S3TRIO
    { "s3trio", s3triofb_init, NULL },
#endif
#ifdef CONFIG_FB_FM2
    { "fm2fb", fm2fb_init, fm2fb_setup },
#endif
#ifdef CONFIG_FB_SIS
    { "sisfb", sisfb_init, sisfb_setup },
#endif
    /*
     * Generic drivers that are used as fallbacks
     *
     * These depend on resource management and must be initialized
     * _after_ all other frame buffer devices that use resource
     * management!
     */
#ifdef CONFIG_FB_OF
    { "offb", offb_init, NULL },
#endif
#ifdef CONFIG_FB_VESA
    { "vesa", vesafb_init, vesafb_setup },
#endif
    /*
     * Chipset specific drivers that don't use resource management (yet)
     */
#ifdef CONFIG_FB_3DFX
```

```
    { "tdfx", tdfxfb_init, tdfxfb_setup },
#endif
#ifdef CONFIG_FB_SGIVW
    { "sgivw", sgivwfb_init, sgivwfb_setup },
#endif
#ifdef CONFIG_FB_ACORN
    { "acorn", acornfb_init, acornfb_setup },
#endif
#ifdef CONFIG_FB_ATARI
    { "atafb", atafb_init, atafb_setup },
#endif
#ifdef CONFIG_FB_MAC
    { "macfb", macfb_init, macfb_setup },
#endif
#ifdef CONFIG_FB_HGA
    { "hga", hgafb_init, hgafb_setup },
#endif
#ifdef CONFIG_FB_IGA
    { "igafb", igafb_init, igafb_setup },
#endif
#ifdef CONFIG_APOLLO
    { "apollo", dnfb_init, NULL },
#endif
#ifdef CONFIG_FB_Q40
    { "q40fb", q40fb_init, NULL },
#endif
#ifdef CONFIG_FB_TGA
    { "tga", tgafb_init, tgafb_setup },
#endif
#ifdef CONFIG_FB_HP300
    { "hpfb", hpfb_init, NULL },
#endif
#ifdef CONFIG_FB_G364
    { "g364", g364fb_init, NULL },
#endif
#ifdef CONFIG_FB_SA1100
    { "sa1100", sa1100fb_init, NULL },
#endif
#ifdef CONFIG_FB_EPSON1356
    { "epson1356", epson1356fb_init, NULL },
#endif
#ifdef CONFIG_FB_MQ200
    { "mqfb", mqfb_init,mqfb_setup },
#endif
#ifdef CONFIG_FB_SUN3
    { "sun3", sun3fb_init, sun3fb_setup },
#endif
```

```
#ifdef CONFIG_FB_HIT
    { "hitfb", hitfb_init, NULL },
#endif
#ifdef CONFIG_FB_ANAKIN
    { "anakinfb", anakinfb_init, NULL },
#endif
#ifdef CONFIG_FB_E1355
    { "e1355fb", e1355fb_init, e1355fb_setup },
#endif
#ifdef CONFIG_FB_DC
    { "dcfb", dcfb_init, NULL },
#endif
    /*
     * Generic drivers that don't use resource management (yet)
     */
#ifdef CONFIG_FB_VGA16
    { "vga16", vga16fb_init, vga16fb_setup },
#endif
#ifdef CONFIG_FB_STI
    { "stifb", stifb_init, stifb_setup },
#endif
#ifdef CONFIG_GSP_RESOLVER
    /* Not a real frame buffer device... */
    { "resolver", NULL, resolver_video_setup },
#endif
#ifdef CONFIG_FB_VIRTUAL
    /*
     * Vfb must be last to avoid that it becomes your primary display if
     * other display devices are present
     */
    { "vfb", vfb_init, vfb_setup },
#endif
};
#define NUM_FB_DRIVERS  (sizeof(fb_drivers)/sizeof(*fb_drivers))
extern const char *global_mode_option;
static initcall_t pref_init_funcs[FB_MAX];
static int num_pref_init_funcs __initdata = 0;
struct fb_info *registered_fb[FB_MAX];
int num_registered_fb;
extern int fbcon_softback_size;
static int first_fb_vc;
static int last_fb_vc = MAX_NR_CONSOLES-1;
static int fbcon_is_default = 1;
#ifdef CONFIG_FB_OF
static int ofonly __initdata = 0;
#endif
static int fbmem_read_proc(char *buf, char **start, off_t offset,
```

```
                int len, int *eof, void *private)
{
    struct fb_info **fi;
    int clen;
    clen = 0;
    for (fi = registered_fb; fi < &registered_fb[FB_MAX] && len < 4000; fi++)
        if (*fi)
            clen += sprintf(buf + clen, "%d %s\n",
                        GET_FB_IDX((*fi)->node),
                        (*fi)->modename);
    *start = buf + offset;
    if (clen > offset)
        clen -= offset;
    else
        clen = 0;
    return clen < len ? clen : len;
}

static ssize_t
fb_read(struct file *file, char *buf, size_t count, loff_t *ppos)
{
    unsigned long p = *ppos;
    struct inode *inode = file->f_dentry->d_inode;
    int fbidx = GET_FB_IDX(inode->i_rdev);
    struct fb_info *info = registered_fb[fbidx];
    struct fb_ops *fb = info->fbops;
    struct fb_fix_screeninfo fix;
    if (! fb || ! info->disp)
        return -ENODEV;
    fb->fb_get_fix(&fix,PROC_CONSOLE(info), info);
    if (p >= fix.smem_len)
        return 0;
    if (count >= fix.smem_len)
        count = fix.smem_len;
    if (count + p > fix.smem_len)
        count = fix.smem_len - p;
    if (count) {
        char *base_addr;
        base_addr = info->disp->screen_base;
        count -= copy_to_user(buf, base_addr+p, count);
        if (!count)
        return -EFAULT;
        *ppos += count;
    }
    return count;
}
```

```
static ssize_t
fb_write(struct file *file, const char *buf, size_t count, loff_t *ppos)
{
    unsigned long p = *ppos;
    struct inode *inode = file->f_dentry->d_inode;
    int fbidx = GET_FB_IDX(inode->i_rdev);
    struct fb_info *info = registered_fb[fbidx];
    struct fb_ops *fb = info->fbops;
    struct fb_fix_screeninfo fix;
    int err;
    if (! fb || ! info->disp)
        return -ENODEV;
    }
    fb->fb_get_fix(&fix, PROC_CONSOLE(info), info);
    if (p > fix.smem_len)
        return -ENOSPC;
    }
    if (count >= fix.smem_len)
        count = fix.smem_len;
    err = 0;
    if (count + p > fix.smem_len) {
        count = fix.smem_len - p;
        err = -ENOSPC;
    }
    if (count) {
        char *base_addr;
        base_addr = info->disp->screen_base;
        count -= copy_from_user(base_addr+p, buf, count);
        *ppos += count;
        err = -EFAULT;
    }
    if (count)
        return count;
    return err;
}

#ifdef CONFIG_KMOD
static void try_to_load(int fb)
{
    char modname[16];
    sprintf(modname, "fb%d", fb);
    request_module(modname);
}
#endif /* CONFIG_KMOD */

static int
fb_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
```

```
    unsigned long arg)
{
    int fbidx = GET_FB_IDX(inode->i_rdev);
    struct fb_info *info = registered_fb[fbidx];
    struct fb_ops *fb = info->fbops;
    struct fb_cmap cmap;
    struct fb_var_screeninfo var;
    struct fb_fix_screeninfo fix;
    struct fb_con2fbmap con2fb;
    struct fb_delay delay_struct;//I add this line
    int i;

    if (! fb)
        return -ENODEV;
    switch (cmd) {
    case FBIOGET_VSCREENINFO:
        if ((i = fb->fb_get_var(&var, PROC_CONSOLE(info), info)))
            return i;
        return copy_to_user((void *) arg, &var,
                    sizeof(var)) ? -EFAULT : 0;
    case FBIOPUT_VSCREENINFO:
        if (copy_from_user(&var, (void *) arg, sizeof(var)))
            return -EFAULT;
        i = var.activate & FB_ACTIVATE_ALL
                ? set_all_vcs(fbidx, fb, &var, info)
                : fb->fb_set_var(&var, PROC_CONSOLE(info), info);
        if (i)
            return i;
        if (copy_to_user((void *) arg, &var, sizeof(var)))
            return -EFAULT;
        return 0;
    case FBIOGET_FSCREENINFO:
        if ((i = fb->fb_get_fix(&fix, PROC_CONSOLE(info), info)))
            return i;
        return copy_to_user((void *) arg, &fix, sizeof(fix)) ?
            -EFAULT : 0;
    case FBIOPUTCMAP:
        if (copy_from_user(&cmap, (void *) arg, sizeof(cmap)))
            return -EFAULT;
        return (fb->fb_set_cmap(&cmap, 0, PROC_CONSOLE(info), info));
    case FBIOGETCMAP:
        if (copy_from_user(&cmap, (void *) arg, sizeof(cmap)))
            return -EFAULT;
        return (fb->fb_get_cmap(&cmap, 0, PROC_CONSOLE(info), info));
    case FBIOPAN_DISPLAY:
        if (copy_from_user(&var, (void *) arg, sizeof(var)))
            return -EFAULT;
```

```
        if (fb->fb_pan_display == NULL)
            return (var.xoffset || var.yoffset) ? -EINVAL : 0;
        if ((i=fb->fb_pan_display(&var, PROC_CONSOLE(info), info)))
            return i;
        if (copy_to_user((void *) arg, &var, sizeof(var)))
            return -EFAULT;
        return i;
    case FBIOGET_CON2FBMAP:
        if (copy_from_user(&con2fb, (void *)arg, sizeof(con2fb)))
            return -EFAULT;
        if (con2fb.console < 1 || con2fb.console > MAX_NR_CONSOLES)
            return -EINVAL;
        con2fb.framebuffer = con2fb_map[con2fb.console-1];
        return copy_to_user((void *)arg, &con2fb,
                    sizeof(con2fb)) ? -EFAULT : 0;
    case FBIOPUT_CON2FBMAP:
        if (copy_from_user(&con2fb, (void *)arg, sizeof(con2fb)))
            return - EFAULT;
        if (con2fb.console < 0 || con2fb.console > MAX_NR_CONSOLES)
            return -EINVAL;
        if (con2fb.framebuffer < 0 || con2fb.framebuffer >= FB_MAX)
            return -EINVAL;
#ifdef CONFIG_KMOD
        if (!registered_fb[con2fb.framebuffer])
            try_to_load(con2fb.framebuffer);
#endif /* CONFIG_KMOD */
        if (!registered_fb[con2fb.framebuffer])
            return -EINVAL;
        if (con2fb.console != 0)
            set_con2fb_map(con2fb.console-1, con2fb.framebuffer);
        else
            /* set them all */
            for (i = 0; i < MAX_NR_CONSOLES; i++)
            set_con2fb_map(i, con2fb.framebuffer);
        return 0;
    case FBIOBLANK:
        if (info->blank == 0)
            return -EINVAL;
        (*info->blank)(arg, info);
        return 0;
    case FBIODIS_CTRLR://I add this case
        fb->fb_disable_controller(info);
        return 0;
    case FBIOEN_CTRLR://I add this case
        fb->fb_enable_controller(info);
        return 0;
    case FBIOPMLEN_ON://I add this case
```

```
        if (copy_from_user(&delay_struct, (void *)arg, sizeof(struct fb_delay)))
            return - EFAULT;
        printk("<1> \n\n!!!Tutti va bene.\n\n");
        printk("<1>\n\n!!!delay_dis = %d",delay_struct.dis);
        printk("<1>\n\n!!!delay_en = %d",delay_struct.en);
        fb->fb_enable_pm_len(&delay_struct,info);
        return 0;
    case FBIOPMLEN_OFF://I add this case
        fb->fb_disable_pm_len();
        return 0;
    case FBIODARKENS://I add this case
        win = (struct fb_window *)arg;

        return 0;
    default:
        if (fb->fb_ioctl == NULL)
            return -EINVAL;
        return fb->fb_ioctl(inode, file, cmd, arg, PROC_CONSOLE(info),
                    info);
    }
}

static int
fb_mmap(struct file *file, struct vm_area_struct * vma)
{
    int fbidx = GET_FB_IDX(file->f_dentry->d_inode->i_rdev);
    struct fb_info *info = registered_fb[fbidx];
    struct fb_ops *fb = info->fbops;
    unsigned long off;
#if !defined(__sparc__) || defined(__sparc_v9__)
    struct fb_fix_screeninfo fix;
    struct fb_var_screeninfo var;
    unsigned long start;
    u32 len;
#endif
    if (vma->vm_pgoff > (~0UL >> PAGE_SHIFT))
        return -EINVAL;
    off = vma->vm_pgoff << PAGE_SHIFT;
    if (!fb)
        return -ENODEV;
    if (fb->fb_mmap) {
        int res;
        lock_kernel();
        res = fb->fb_mmap(info, file, vma);
        unlock_kernel();
        return res;
    }
```

```
#if defined(__sparc__) && !defined(__sparc_v9__)
    /* Should never get here, all fb drivers should have their own
       mmap routines */
    return -EINVAL;
#else
    /* !sparc32... */
    lock_kernel();
    fb->fb_get_fix(&fix, PROC_CONSOLE(info), info);
    /* frame buffer memory */
    start = fix.smem_start;
    len = PAGE_ALIGN((start & ~PAGE_MASK)+fix.smem_len);
    if (off >= len) {
        /* memory mapped io */
        off -= len;
        fb->fb_get_var(&var, PROC_CONSOLE(info), info);
        if (var.accel_flags)
            return -EINVAL;
        start = fix.mmio_start;
        len = PAGE_ALIGN((start & ~PAGE_MASK)+fix.mmio_len);
    }
    unlock_kernel();
    start &= PAGE_MASK;
    if ((vma->vm_end - vma->vm_start + off) > len)
        return -EINVAL;
    off += start;
    vma->vm_pgoff = off >> PAGE_SHIFT;
#if defined(__sparc_v9__)
    vma->vm_flags |= (VM_SHM | VM_LOCKED);
    if (io_remap_page_range(vma->vm_start, off,
                vma->vm_end - vma->vm_start, vma->vm_page_prot, 0))
        return -EAGAIN;
    vma->vm_flags |= VM_IO;
#else
#if defined(__mc68000__)
#if defined(CONFIG_SUN3)
    pgprot_val(vma->vm_page_prot) |= SUN3_PAGE_NOCACHE;
#else
    if (CPU_IS_020_OR_030)
        pgprot_val(vma->vm_page_prot) |= _PAGE_NOCACHE030;
    if (CPU_IS_040_OR_060) {
        pgprot_val(vma->vm_page_prot) &= _CACHEMASK040;
        /* Use no-cache mode, serialized */
        pgprot_val(vma->vm_page_prot) |= _PAGE_NOCACHE_S;
    }
#endif
#elif defined(__powerpc__)
    pgprot_val(vma->vm_page_prot) |= _PAGE_NO_CACHE|_PAGE_GUARDED;
```

```
#elif defined(__alpha__)
    /* Caching is off in the I/O space quadrant by design.  */
#elif defined(__i386__)
    if (boot_cpu_data.x86 > 3)
        pgprot_val(vma->vm_page_prot) |= _PAGE_PCD;
#elif defined(__mips__)
    pgprot_val(vma->vm_page_prot) &= ~_CACHE_MASK;
    pgprot_val(vma->vm_page_prot) |= _CACHE_UNCACHED;
#elif defined(__arm__)
    vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot);
    /* This is an IO map - tell maydump to skip this VMA */
    vma->vm_flags |= VM_IO;
#elif defined(__sh__)
    pgprot_val(vma->vm_page_prot) &= ~_PAGE_CACHABLE;
#else
#warning What do we have to do here??
#endif
    if (io_remap_page_range(vma->vm_start, off,
                vma->vm_end - vma->vm_start, vma->vm_page_prot))
        return -EAGAIN;
#endif /* !__sparc_v9__ */
    return 0;
#endif /* !sparc32 */
}
#if 1 /* to go away in 2.5.0 */
int GET_FB_IDX(kdev_t rdev)
{
    int fbidx = MINOR(rdev);
    if (fbidx >= 32) {
    int newfbidx = fbidx >> 5;
    static int warned;
    if (!(warned & (1<<newfbidx))) {
        warned |= 1<<newfbidx;
        printk("Warning: Remapping obsolete /dev/fb* minor %d to %d\n",
           fbidx, newfbidx);
    }
    fbidx = newfbidx;
    }
    return fbidx;
}
#endif

static int
fb_open(struct inode *inode, struct file *file)
{
    int fbidx = GET_FB_IDX(inode->i_rdev);
    struct fb_info *info;
```

```
    int res = 0;
#ifdef CONFIG_KMOD
    if (!(info = registered_fb[fbidx]))
        try_to_load(fbidx);
#endif /* CONFIG_KMOD */
    if (!(info = registered_fb[fbidx]))
        return -ENODEV;
    if (info->fbops->owner)
        __MOD_INC_USE_COUNT(info->fbops->owner);
    if (info->fbops->fb_open) {
        res = info->fbops->fb_open(info,1);
        if (res && info->fbops->owner)
            __MOD_DEC_USE_COUNT(info->fbops->owner);
    }
    return res;
}

static int
fb_release(struct inode *inode, struct file *file)
{
    int fbidx = GET_FB_IDX(inode->i_rdev);
    struct fb_info *info;
    lock_kernel();
    info = registered_fb[fbidx];
    if (info->fbops->fb_release)
        info->fbops->fb_release(info,1);
    if (info->fbops->owner)
        __MOD_DEC_USE_COUNT(info->fbops->owner);
    unlock_kernel();
    return 0;
}

static struct file_operations fb_fops = {
    owner:      THIS_MODULE,
    read:       fb_read,
    write:      fb_write,
    ioctl:      fb_ioctl,
    mmap:       fb_mmap,
    open:       fb_open,
    release:    fb_release,
#ifdef HAVE_ARCH_FB_UNMAPPED_AREA
    get_unmapped_area: get_fb_unmapped_area,
#endif
};

static devfs_handle_t devfs_handle;
/**
```

```
 * register_framebuffer - registers a frame buffer device
 * @fb_info: frame buffer info structure
 *
 * Registers a frame buffer device @fb_info.
 *
 * Returns negative errno on error, or zero for success.
 *
 */
int
register_framebuffer(struct fb_info *fb_info)
{
    int i, j;
    char name_buf[8];
    static int fb_ever_opened[FB_MAX];
    static int first = 1;

    if (num_registered_fb == FB_MAX)
        return -ENXIO;
    num_registered_fb++;
    for (i = 0 ; i < FB_MAX; i++)
        if (!registered_fb[i])
            break;
    fb_info->node = MKDEV(FB_MAJOR, i);
    registered_fb[i] = fb_info;
    if (!fb_ever_opened[i]) {
        struct module *owner = fb_info->fbops->owner;
        /*
         * We assume initial frame buffer devices can be opened this
         * many times
         */
        for (j = 0; j < MAX_NR_CONSOLES; j++)
            if (con2fb_map[j] == i) {
                if (owner)
                    __MOD_INC_USE_COUNT(owner);
                if (!fb_info->fbops->fb_open)
                    continue;
                if (!fb_info->fbops->fb_open(fb_info,0))
                    continue;
                if (owner)
                    __MOD_DEC_USE_COUNT(owner);
            }
        fb_ever_opened[i] = 1;
    }
    if (first) {
        first = 0;
        take_over_console(&fb_con, first_fb_vc, last_fb_vc, fbcon_is_default);
    }
```

```
    sprintf (name_buf, "%d", i);
    fb_info->devfs_handle =
        devfs_register (devfs_handle, name_buf, DEVFS_FL_DEFAULT,
                FB_MAJOR, i, S_IFCHR | S_IRUGO | S_IWUGO,
                &fb_fops, NULL);
    return 0;
}
/**
 * unregister_framebuffer - releases a frame buffer device
 * @fb_info: frame buffer info structure
 *
 * Unregisters a frame buffer device @fb_info.
 *
 * Returns negative errno on error, or zero for success.
 *
 */
int
unregister_framebuffer(struct fb_info *fb_info)
{
    int i, j;
    i = GET_FB_IDX(fb_info->node);
    for (j = 0; j < MAX_NR_CONSOLES; j++)
        if (con2fb_map[j] == i)
            return -EBUSY;
    if (!registered_fb[i])
        return -EINVAL;
    devfs_unregister (fb_info->devfs_handle);
    fb_info->devfs_handle = NULL;
    devfs_unregister (fb_info->devfs_lhandle);
    fb_info->devfs_lhandle = NULL;
    registered_fb[i]=NULL;
    num_registered_fb--;
    return 0;
}
/**
 * fbmem_init - init frame buffer subsystem
 *
 * Initialize the frame buffer subsystem.
 *
 * NOTE: This function is _only_ to be called by drivers/char/mem.c.
 *
 */
void __init
fbmem_init(void)
{
    int i;
    create_proc_read_entry("fb", 0, 0, fbmem_read_proc, NULL);
```

```
    devfs_handle = devfs_mk_dir (NULL, "fb", NULL);
    if (devfs_register_chrdev(FB_MAJOR,"fb",&fb_fops))
        printk("unable to get major %d for fb devs\n", FB_MAJOR);

#ifdef CONFIG_FB_OF
    if (ofonly) {
        offb_init();
        return;
    }
#endif
    /*
     * Probe for all builtin frame buffer devices
     */
    for (i = 0; i < num_pref_init_funcs; i++)
        pref_init_funcs[i]();
    for (i = 0; i < NUM_FB_DRIVERS; i++)
        if (fb_drivers[i].init)
            fb_drivers[i].init();
}
/**
 * video_setup - process command line options
 * @options: string of options
 *
 * Process command line options for frame buffer subsystem.
 *
 * NOTE: This function is a __setup and __init function.
 *
 * Returns zero.
 *
 */
int __init video_setup(char *options)
{
    int i, j;

    if (!options || !*options)
        return 0;
    if (!strncmp(options, "scrollback:", 11)) {
        options += 11;
        if (*options) {
        fbcon_softback_size = simple_strtoul(options, &options, 0);
        if (*options == 'k' || *options == 'K') {
            fbcon_softback_size *= 1024;
            options++;
        }
        if (*options != ',')
            return 0;
        options++;
```

```
        } else
            return 0;
    }
    if (!strncmp(options, "map:", 4)) {
        options += 4;
        if (*options)
            for (i = 0, j = 0; i < MAX_NR_CONSOLES; i++) {
                if (!options[j])
                    j = 0;
                con2fb_map[i] = (options[j++]-'0') % FB_MAX;
            }
        return 0;
    }
    if (!strncmp(options, "vc:", 3)) {
        options += 3;
        if (*options)
        first_fb_vc = simple_strtoul(options, &options, 10) - 1;
        if (first_fb_vc < 0)
        first_fb_vc = 0;
        if (*options++ == '-')
        last_fb_vc = simple_strtoul(options, &options, 10) - 1;
        fbcon_is_default = 0;
    }
#ifdef CONFIG_FB_OF
    if (!strcmp(options, "ofonly")) {
        ofonly = 1;
        return 0;
    }
#endif
    if (num_pref_init_funcs == FB_MAX)
        return 0;
    for (i = 0; i < NUM_FB_DRIVERS; i++) {
        j = strlen(fb_drivers[i].name);
        if (!strncmp(options, fb_drivers[i].name, j) &&
        options[j] == ':') {
            if (!strcmp(options+j+1, "off"))
                fb_drivers[i].init = NULL;
            else {
                if (fb_drivers[i].init) {
                    pref_init_funcs[num_pref_init_funcs++] =
                        fb_drivers[i].init;
                    fb_drivers[i].init = NULL;
                }
                if (fb_drivers[i].setup)
                    fb_drivers[i].setup(options+j+1);
            }
            return 0;
```

```
        }
    }
    /*
     * If we get here no fb was specified.
     * We consider the argument to be a global video mode option.
     */
    global_mode_option = options;
    return 0;
}

__setup("video=", video_setup);
    /*
     *  Visible symbols for modules
     */
EXPORT_SYMBOL(register_framebuffer);
EXPORT_SYMBOL(unregister_framebuffer);
EXPORT_SYMBOL(registered_fb);
EXPORT_SYMBOL(num_registered_fb);
#if 1 /* to go away in 2.5.0 */
EXPORT_SYMBOL(GET_FB_IDX);
#endif
```

Appendix D

sa1100fb.c

```
/*
 *  linux/drivers/video/sa1100fb.c
 *
 *  Copyright (C) 1999 Eric A. Thomas
 *   Based on acornfb.c Copyright (C) Russell King.
 *
 * This file is subject to the terms and conditions of the GNU General Public
 * License.  See the file COPYING in the main directory of this archive for
 * more details.
 *
 *          StrongARM 1100 LCD Controller Frame Buffer Driver
 */
#include <linux/config.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/errno.h>
#include <linux/string.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
#include <linux/fb.h>
#include <linux/delay.h>
#include <linux/pm.h>
#include <linux/init.h>
#include <linux/cpufreq.h>
#include <asm/hardware.h>
#include <asm/io.h>
#include <asm/irq.h>
#include <asm/mach-types.h>
#include <asm/uaccess.h>
#include <video/fbcon.h>
#include <video/fbcon-mfb.h>
```

```
#include <video/fbcon-cfb4.h>
#include <video/fbcon-cfb8.h>
#include <video/fbcon-cfb16.h>
#include <linux/tqueue.h>//I add this line
#include <linux/delay.h>//I add this line for "mdelay"

#undef CHECK_COMPAT

#define DEBUG 0

#define DEBUG_VAR 1

#undef ASSABET_PAL_VIDEO

#include "sa1100fb.h"

void (*sa1100fb_blank_helper)(int blank);
EXPORT_SYMBOL(sa1100fb_blank_helper);

#ifdef CHECK_COMPAT
static void
sa1100fb_check_shadow(struct sa1100fb_lcd_reg *new_reg,
               struct fb_var_screeninfo *var, u_int pcd)
{
    struct sa1100fb_lcd_reg shadow;
    int different = 0;
    /*
     * These machines are good machines!
     */
    if (!machine_is_assabet() && !machine_is_bitsy())
        return;
    /*
     * The following ones are bad, bad, bad.
     * Please make yours good!
     */
    if (machine_is_pangolin()) {
        DPRINTK("Configuring Pangolin LCD\n");
        shadow.lccr0 =
            LCCR0_LEN + LCCR0_Color + LCCR0_LDM +
            LCCR0_BAM + LCCR0_ERM + LCCR0_Act +
            LCCR0_LtlEnd + LCCR0_DMADel(0);
        shadow.lccr1 =
            LCCR1_DisWdth(var->xres) + LCCR1_HorSnchWdth(64) +
            LCCR1_BegLnDel(160) + LCCR1_EndLnDel(24);
        shadow.lccr2 =
            LCCR2_DisHght(var->yres) + LCCR2_VrtSnchWdth(7) +
            LCCR2_BegFrmDel(7) + LCCR2_EndFrmDel(1);
```

```
        shadow.lccr3 =
            LCCR3_PixClkDiv(pcd) + LCCR3_HorSnchH +
            LCCR3_VrtSnchH + LCCR3_PixFlEdg + LCCR3_OutEnH;
        DPRINTK("pcd = %x, PixCldDiv(pcd)=%x\n",
            pcd, LCCR3_PixClkDiv(pcd));
    }
    if (machine_is_freebird()) {
        DPRINTK("Configuring  Freebird LCD\n");
#if 1
        shadow.lccr0 = 0x00000038;
        shadow.lccr1 = 0x010108e0;
        shadow.lccr2 = 0x0000053f;
        shadow.lccr3 = 0x00000c20;
#else
        shadow.lccr0 =
            LCCR0_LEN + LCCR0_Color + LCCR0_Sngl +
            LCCR0_LDM + LCCR0_BAM + LCCR0_ERM + LCCR0_Pas +
            LCCR0_LtlEnd + LCCR0_DMADel(0);
        /* Check ,Chester */
        shadow.lccr1 =
            LCCR1_DisWdth(var->xres) + LCCR1_HorSnchWdth(5) +
            LCCR1_BegLnDel(61) + LCCR1_EndLnDel(9);
        /* Check ,Chester */
        shadow.lccr2 =
            LCCR2_DisHght(var->yres) + LCCR2_VrtSnchWdth(1) +
            LCCR2_BegFrmDel(3) + LCCR2_EndFrmDel(0);
        /* Check ,Chester */
        shadow.lccr3 =
            LCCR3_OutEnH + LCCR3_PixFlEdg + LCCR3_VrtSnchH +
            LCCR3_HorSnchH + LCCR3_ACBsCntOff +
            LCCR3_ACBsDiv(2) + LCCR3_PixClkDiv(pcd);
#endif
    }
    if (machine_is_brutus()) {
        DPRINTK("Configuring  Brutus LCD\n");
        shadow.lccr0 =
            LCCR0_LEN + LCCR0_Color + LCCR0_Sngl + LCCR0_Pas +
            LCCR0_LtlEnd + LCCR0_LDM + LCCR0_BAM + LCCR0_ERM +
            LCCR0_DMADel(0);
        shadow.lccr1 =
            LCCR1_DisWdth(var->xres) + LCCR1_HorSnchWdth(3) +
            LCCR1_BegLnDel(41) + LCCR1_EndLnDel(101);
        shadow.lccr2 =
            LCCR2_DisHght(var->yres) + LCCR2_VrtSnchWdth(1) +
            LCCR2_BegFrmDel(0) + LCCR2_EndFrmDel(0);
        shadow.lccr3 =
            LCCR3_OutEnH + LCCR3_PixRsEdg + LCCR3_VrtSnchH +
```

```
            LCCR3_HorSnchH + LCCR3_ACBsCntOff +
            LCCR3_ACBsDiv(2) + LCCR3_PixClkDiv(44);
    }
    if (machine_is_huw_webpanel()) {
        DPRINTK("Configuring  HuW LCD\n");
        shadow.lccr0 = LCCR0_LEN + LCCR0_Dual + LCCR0_LDM;
        shadow.lccr1 = LCCR1_DisWdth(var->xres) +
            LCCR1_HorSnchWdth(3) +
            LCCR1_BegLnDel(41) + LCCR1_EndLnDel(101);
        shadow.lccr2 = 239 + LCCR2_VrtSnchWdth(1);
        shadow.lccr3 = 8 + LCCR3_OutEnH +
            LCCR3_PixRsEdg + LCCR3_VrtSnchH +
            LCCR3_HorSnchH + LCCR3_ACBsCntOff + LCCR3_ACBsDiv(2);
    }
#ifdef CONFIG_SA1100_CERF
    if (machine_is_cerf()) {
        DPRINTK("Configuring Cerf LCD\n");
#if defined (CONFIG_CERF_LCD_72_A)
        shadow.lccr0 =
            LCCR0_LEN + LCCR0_Color + LCCR0_Dual +
            LCCR0_LDM + LCCR0_BAM + LCCR0_ERM + LCCR0_Pas +
            LCCR0_LtlEnd + LCCR0_DMADel(0);
        shadow.lccr1 =
            LCCR1_DisWdth(var->xres) + LCCR1_HorSnchWdth(5) +
            LCCR1_BegLnDel(61) + LCCR1_EndLnDel(9);
        shadow.lccr2 =
            LCCR2_DisHght(var->yres / 2) + LCCR2_VrtSnchWdth(1) +
            LCCR2_BegFrmDel(3) + LCCR2_EndFrmDel(0);
        shadow.lccr3 =
            LCCR3_OutEnH + LCCR3_PixRsEdg + LCCR3_VrtSnchH +
            LCCR3_HorSnchH + LCCR3_ACBsCntOff +
            LCCR3_ACBsDiv(2) + LCCR3_PixClkDiv(38);
#elif defined (CONFIG_CERF_LCD_57_A)
        shadow.lccr0 =
            LCCR0_LEN + LCCR0_Color + LCCR0_Sngl +
            LCCR0_LDM + LCCR0_BAM + LCCR0_ERM + LCCR0_Pas +
            LCCR0_LtlEnd + LCCR0_DMADel(0);
        shadow.lccr1 =
            LCCR1_DisWdth(var->xres) + LCCR1_HorSnchWdth(5) +
            LCCR1_BegLnDel(61) + LCCR1_EndLnDel(9);
        shadow.lccr2 =
            LCCR2_DisHght(var->yres) + LCCR2_VrtSnchWdth(1) +
            LCCR2_BegFrmDel(3) + LCCR2_EndFrmDel(0);
        shadow.lccr3 =
            LCCR3_OutEnH + LCCR3_PixRsEdg + LCCR3_VrtSnchH +
            LCCR3_HorSnchH + LCCR3_ACBsCntOff +
            LCCR3_ACBsDiv(2) + LCCR3_PixClkDiv(38);
```

```
#elif defined (CONFIG_CERF_LCD_38_A)
        shadow.lccr0 =
            LCCR0_LEN + LCCR0_Color + LCCR0_Sngl +
            LCCR0_LDM + LCCR0_BAM + LCCR0_ERM + LCCR0_Pas +
            LCCR0_LtlEnd + LCCR0_DMADel(0);
        shadow.lccr1 =
            LCCR1_DisWdth(var->xres) + LCCR1_HorSnchWdth(5) +
            LCCR1_BegLnDel(61) + LCCR1_EndLnDel(9);
        shadow.lccr2 =
            LCCR2_DisHght(var->yres) + LCCR2_VrtSnchWdth(1) +
            LCCR2_BegFrmDel(3) + LCCR2_EndFrmDel(0);
        shadow.lccr3 =
            LCCR3_OutEnH + LCCR3_PixRsEdg + LCCR3_VrtSnchH +
            LCCR3_HorSnchH + LCCR3_ACBsCntOff +
            LCCR3_ACBsDiv(2) + LCCR3_PixClkDiv(38);
#else
#error "Must have a CerfBoard LCD form factor selected"
#endif
    }
#endif
    if (machine_is_lart()) {
        DPRINTK("Configuring LART LCD\n");
#if defined LART_GREY_LCD
        shadow.lccr0 =
            LCCR0_LEN + LCCR0_Mono + LCCR0_Sngl + LCCR0_Pas +
            LCCR0_LtlEnd + LCCR0_LDM + LCCR0_BAM + LCCR0_ERM +
            LCCR0_DMADel(0);
        shadow.lccr1 =
            LCCR1_DisWdth(var->xres) + LCCR1_HorSnchWdth(1) +
            LCCR1_BegLnDel(4) + LCCR1_EndLnDel(2);
        shadow.lccr2 =
            LCCR2_DisHght(var->yres) + LCCR2_VrtSnchWdth(1) +
            LCCR2_BegFrmDel(0) + LCCR2_EndFrmDel(0);
        shadow.lccr3 =
            LCCR3_PixClkDiv(34) + LCCR3_ACBsDiv(512) +
            LCCR3_ACBsCntOff + LCCR3_HorSnchH + LCCR3_VrtSnchH;
#endif
#if defined LART_COLOR_LCD
        shadow.lccr0 =
            LCCR0_LEN + LCCR0_Color + LCCR0_Sngl + LCCR0_Act +
            LCCR0_LtlEnd + LCCR0_LDM + LCCR0_BAM + LCCR0_ERM +
            LCCR0_DMADel(0);
        shadow.lccr1 =
            LCCR1_DisWdth(var->xres) + LCCR1_HorSnchWdth(2) +
            LCCR1_BegLnDel(69) + LCCR1_EndLnDel(8);
        shadow.lccr2 =
            LCCR2_DisHght(var->yres) + LCCR2_VrtSnchWdth(3) +
```

```
            LCCR2_BegFrmDel(14) + LCCR2_EndFrmDel(4);
        shadow.lccr3 =
            LCCR3_PixClkDiv(34) + LCCR3_ACBsDiv(512) +
            LCCR3_ACBsCntOff + LCCR3_HorSnchL + LCCR3_VrtSnchL +
            LCCR3_PixFlEdg;
#endif
#if defined LART_VIDEO_OUT
        shadow.lccr0 =
            LCCR0_LEN + LCCR0_Color + LCCR0_Sngl + LCCR0_Act +
            LCCR0_LtlEnd + LCCR0_LDM + LCCR0_BAM + LCCR0_ERM +
            LCCR0_DMADel(0);
        shadow.lccr1 =
            LCCR1_DisWdth(640) + LCCR1_HorSnchWdth(95) +
            LCCR1_BegLnDel(40) + LCCR1_EndLnDel(24);
        shadow.lccr2 =
            LCCR2_DisHght(480) + LCCR2_VrtSnchWdth(2) +
            LCCR2_BegFrmDel(32) + LCCR2_EndFrmDel(11);
        shadow.lccr3 =
            LCCR3_PixClkDiv(8) + LCCR3_ACBsDiv(512) +
            LCCR3_ACBsCntOff + LCCR3_HorSnchH + LCCR3_VrtSnchH +
            LCCR3_PixFlEdg + LCCR3_OutEnL;
#endif
    }
    if (machine_is_graphicsclient()) {
        DPRINTK("Configuring GraphicsClient LCD\n");
        shadow.lccr0 =
            LCCR0_LEN + LCCR0_Color + LCCR0_Sngl + LCCR0_Act;
        shadow.lccr1 =
            LCCR1_DisWdth(var->xres) + LCCR1_HorSnchWdth(9) +
            LCCR1_EndLnDel(54) + LCCR1_BegLnDel(54);
        shadow.lccr2 =
            LCCR2_DisHght(var->yres) + LCCR2_VrtSnchWdth(9) +
            LCCR2_EndFrmDel(32) + LCCR2_BegFrmDel(24);
        shadow.lccr3 =
            LCCR3_PixClkDiv(10) + LCCR3_ACBsDiv(2) +
            LCCR3_ACBsCntOff + LCCR3_HorSnchL + LCCR3_VrtSnchL;
    }
    if (machine_is_omnimeter()) {
        DPRINTK("Configuring  OMNI LCD\n");
        shadow.lccr0 = LCCR0_LEN | LCCR0_CMS | LCCR0_DPD;
        shadow.lccr1 =
            LCCR1_BegLnDel(10) + LCCR1_EndLnDel(10) +
            LCCR1_HorSnchWdth(1) + LCCR1_DisWdth(var->xres);
        shadow.lccr2 = LCCR2_DisHght(var->yres);
        shadow.lccr3 =
            LCCR3_ACBsDiv(0xFF) + LCCR3_PixClkDiv(44);
//jca (GetPCD(25) << LCD3_V_PCD);
```

```
}
if (machine_is_xp860()) {
    DPRINTK("Configuring XP860 LCD\n");
    shadow.lccr0 =
        LCCR0_LEN + LCCR0_Color + LCCR0_Sngl + LCCR0_Act +
        LCCR0_LtlEnd + LCCR0_LDM + LCCR0_ERM + LCCR0_DMADel(0);
    shadow.lccr1 =
        LCCR1_DisWdth(var->xres) +
        LCCR1_HorSnchWdth(var->hsync_len) +
        LCCR1_BegLnDel(var->left_margin) +
        LCCR1_EndLnDel(var->right_margin);
    shadow.lccr2 =
        LCCR2_DisHght(var->yres) +
        LCCR2_VrtSnchWdth(var->vsync_len) +
        LCCR2_BegFrmDel(var->upper_margin) +
        LCCR2_EndFrmDel(var->lower_margin);
    shadow.lccr3 =
        LCCR3_PixClkDiv(6) + LCCR3_HorSnchL + LCCR3_VrtSnchL;
}

if (shadow.lccr0 != new_regs->lccr0) {
    printk(KERN_ERR "LCCR1 mismatch: 0x%08x != 0x%08x\n",
        shadow.lccr1, new_regs->lccr1);
    different = 1;
}
if (shadow.lccr1 != new_regs->lccr1) {
    printk(KERN_ERR "LCCR1 mismatch: 0x%08x != 0x%08x\n",
        shadow.lccr1, new_regs->lccr1);
    different = 1;
}
if (shadow.lccr2 != new_regs->lccr2) {
    printk(KERN_ERR "LCCR2 mismatch: 0x%08x != 0x%08x\n",
        shadow.lccr2, new_regs->lccr2);
    different = 1;
}
if (shadow.lccr3 != new_regs->lccr3) {
    printk(KERN_ERR "LCCR3 mismatch: 0x%08x != 0x%08x\n",
        shadow.lccr3, new_regs->lccr3);
    different = 1;
}
if (different) {
    printk(KERN_ERR "var: xres=%d hslen=%d lm=%d rm=%d\n",
        var->xres, var->hsync_len,
        var->left_margin, var->right_margin);
    printk(KERN_ERR "var: yres=%d vslen=%d um=%d bm=%d\n",
        var->yres, var->vsync_len,
        var->upper_margin, var->lower_margin);
```

```
        printk(KERN_ERR "Please report this to Russell King "
            "<rmk@arm.linux.org.uk>\n");
    }
    DPRINTK("olccr0 = 0x%08x\n", shadow.lccr0);
    DPRINTK("olccr1 = 0x%08x\n", shadow.lccr1);
    DPRINTK("olccr2 = 0x%08x\n", shadow.lccr2);
    DPRINTK("olccr3 = 0x%08x\n", shadow.lccr3);
}
#else
#define sa1100fb_check_shadow(regs,var,pcd)
#endif
/*
 * IMHO this looks wrong.  In 8BPP, length should be 8.
 */
static struct sa1100fb_rgb rgb_8 = {
    red:    { offset: 0,  length: 4, },
    green:  { offset: 0,  length: 4, },
    blue:   { offset: 0,  length: 4, },
    transp: { offset: 0,  length: 0, },
};

static struct sa1100fb_rgb def_rgb_16 = {
    red:    { offset: 11, length: 5, },
    green:  { offset: 5,  length: 6, },
    blue:   { offset: 0,  length: 5, },
    transp: { offset: 0,  length: 0, },
};

#ifdef CONFIG_SA1100_ASSABET
static struct sa1100fb_mach_info assabet_info __initdata = {
#ifdef ASSABET_PAL_VIDEO
    pixclock:   67797,      bpp:        16,
    xres:       640,        yres:       512,
    hsync_len:  64,     vsync_len:  6,
    left_margin:    125,        upper_margin:   70,
    right_margin:   115,        lower_margin:   36,
    sync:       0,
    lccr0:      LCCR0_Color | LCCR0_Sngl | LCCR0_Act,
    lccr3:      LCCR3_OutEnH | LCCR3_PixRsEdg | LCCR3_ACBsDiv(512),
#else
    pixclock:   171521,     bpp:        8,
    xres:       320,        yres:       240,
    hsync_len:  5,      vsync_len:  1,
    left_margin:    61,     upper_margin:   3,
    right_margin:   9,      lower_margin:   0,
    sync:       FB_SYNC_HOR_HIGH_ACT | FB_SYNC_VERT_HIGH_ACT,
    lccr0:      LCCR0_Color | LCCR0_Sngl | LCCR0_Act,
```

```
    lccr3:      LCCR3_OutEnH | LCCR3_PixRsEdg | LCCR3_ACBsDiv(2),
#endif
};
#endif

#ifdef CONFIG_SA1100_BITSY
static struct sa1100fb_mach_info bitsy_info __initdata = {
    pixclock:   0,      bpp:        16,
    xres:       320,        yres:       240,
    hsync_len:  3,      vsync_len:  3,
    left_margin:    12,     upper_margin:   10,
    right_margin:   17,     lower_margin:   1,
    sync:       0,
    lccr0:      LCCR0_Color | LCCR0_Sngl | LCCR0_Act,
    lccr3:      LCCR3_OutEnH | LCCR3_PixRsEdg | LCCR3_ACBsDiv(2) |
            0x10 /* PCD */,
#error FIXME
    /*
     * FIXME: please get rid of the PCD definition in favour of
     * LCCR3_PixClkDiv. --rmk
     */
};

static struct sa1100fb_rgb bitsy_rgb_16 = {
    red:    { offset: 12, length: 4, },
    green:  { offset: 7,  length: 4, },
    blue:   { offset: 1,  length: 4, },
    transp: { offset: 0,  length: 0, },
};
#endif

#ifdef CONFIG_SA1100_BRUTUS
static struct sa1100fb_mach_info brutus_info __initdata = {
    pixclock:   0,      bpp:        8,
    xres:       320,        yres:       240,
    hsync_len:  3,      vsync_len:  1,
    left_margin:    41,     upper_margin:   0,
    right_margin:   101,        lower_margin:   0,
    sync:       FB_SYNC_HOR_HIGH_ACT | FB_SYNC_VERT_HIGH_ACT,
    lccr0:      LCCR0_Color | LCCR0_Sngl | LCCR0_Pas,
    lccr3:      LCCR3_OutEnH | LCCR3_PixRsEdg | LCCR3_ACBsDiv(2) |
            LCCR3_PixClkDiv(44),
};
#endif

#ifdef CONFIG_SA1100_CERF
static struct sa1100fb_mach_info cerf_info __initdata = {
```

```
    pixclock:   171521,     bpp:        8,
#if defined(CONFIG_CERF_LCD_72_A)
    xres:       640,        yres:       480,
    lccr0:      LCCR0_Color | LCCR0_Dual | LCCR0_Pas,
    lccr3:      LCCR3_OutEnH | LCCR3_PixRsEdg | LCCR3_ACBsDiv(2) |
            LCCR3_PixClkDiv(38),
#elif defined(CONFIG_CERF_LCD_57_A)
    xres:       320,        yres:       240,
    lccr0:      LCCR0_Color | LCCR0_Sngl | LCCR0_Pas,
    lccr3:      LCCR3_OutEnH | LCCR3_PixRsEdg | LCCR3_ACBsDiv(2) |
            LCCR3_PixClkDiv(38),
#elif defined(CONFIG_CERF_LCD_38_A)
    xres:       240,        yres:       320,
    lccr0:      LCCR0_Color | LCCR0_Sngl | LCCR0_Pas,
    lccr3:      LCCR3_OutEnH | LCCR3_PixRsEdg | LCCR3_ACBsDiv(2) |
            LCCR3_PixClkDiv(38),
#else
#error "Must have a CerfBoard LCD form factor selected"
#endif
    hsync_len:  5,      vsync_len:  1,
    left_margin:    61,     upper_margin:   3,
    right_margin:   9,      lower_margin:   0,
    sync:       FB_SYNC_HOR_HIGH_ACT | FB_SYNC_VERT_HIGH_ACT,
};
#endif

#ifdef CONFIG_SA1100_FREEBIRD
#warning Please check this carefully
static struct sa1100fb_mach_info freebird_info __initdata = {
    pixclock:   171521,     bpp:        16,
    xres:       240,        yres:       320,
    hsync_len:  3,      vsync_len:  2,
    left_margin:    2,      upper_margin:   0,
    right_margin:   2,      lower_margin:   0,
    sync:       FB_SYNC_HOR_HIGH_ACT | FB_SYNC_VERT_HIGH_ACT,
    lccr0:      LCCR0_Color | LCCR0_Sngl | LCCR0_Pas,
    lccr3:      LCCR3_OutEnH | LCCR3_PixFlEdg | LCCR3_ACBsDiv(2),
};

static struct sa1100fb_rgb freebird_rgb_16 = {
    red:    { offset: 8,  length: 4, },
    green:  { offset: 4,  length: 4, },
    blue:   { offset: 0,  length: 4, },
    transp: { offset: 12, length: 4, },
};
#endif
```

```
#ifdef CONFIG_SA1100_GRAPHICSCLIENT
static struct sa1100fb_mach_info graphicsclient_info __initdata = {
    pixclock:   0,      bpp:        8,
    xres:       640,        yres:       480,
    hsync_len:  9,      vsync_len:  9,
    left_margin:    54,     upper_margin:   24,
    right_margin:   54,     lower_margin:   32,
    sync:       0,
    lccr0:      LCCR0_Color | LCCR0_Sngl | LCCR0_Act,
    lccr3:      LCCR3_OutEnH | LCCR3_PixRsEdg | LCCR3_ACBsDiv(2) |
            LCCR3_PixClkDiv(10),
};
#endif

#ifdef CONFIG_SA1100_HUW_WEBPANEL
static struct sa1100fb_mach_info huw_webpanel_info __initdata = {
    pixclock:   0,      bpp:        8,
    xres:       640,        yres:       480,
    hsync_len:  3,      vsync_len:  1,
    left_margin:    41,     upper_margin:   0,
    right_margin:   101,        lower_margin:   0,
    sync:       FB_SYNC_HOR_HIGH_ACT | FB_SYNC_VERT_HIGH_ACT,
    lccr0:      LCCR0_Color | LCCR0_Dual | LCCR0_Pas,
    lccr3:      LCCR3_OutEnH | LCCR3_PixRsEdg | LCCR3_ACBsDiv(2) | 8,
#error FIXME
    /*
     * FIXME: please get rid of the '| 8' in preference to an
     * LCCR3_PixClkDiv() version. --rmk
     */
};
#endif

#ifdef LART_GREY_LCD
static struct sa1100fb_mach_info lart_grey_info __initdata = {
    pixclock:   150000,     bpp:        4,
    xres:       320,        yres:       240,
    hsync_len:  1,      vsync_len:  1,
    left_margin:    4,      upper_margin:   0,
    right_margin:   2,      lower_margin:   0,
    cmap_greyscale: 1,
    sync:       FB_SYNC_HOR_HIGH_ACT | FB_SYNC_VERT_HIGH_ACT,
    lccr0:      LCCR0_Mono | LCCR0_Sngl | LCCR0_Pas | LCCR0_4PixMono,
    lccr3:      LCCR3_OutEnH | LCCR3_PixRsEdg | LCCR3_ACBsDiv(512),
};
#endif
#ifdef LART_COLOR_LCD
static struct sa1100fb_mach_info lart_color_info __initdata = {
```

```
    pixclock:   150000,      bpp:        16,
    xres:       320,         yres:       240,
    hsync_len:  2,       vsync_len:  3,
    left_margin:     69,      upper_margin:   14,
    right_margin:    8,       lower_margin:   4,
    sync:       0,
    lccr0:      LCCR0_Color | LCCR0_Sngl | LCCR0_Act,
    lccr3:      LCCR3_OutEnH | LCCR3_PixFlEdg | LCCR3_ACBsDiv(512),
};
#endif
#ifdef LART_VIDEO_OUT
static struct sa1100fb_mach_info lart_video_info __initdata = {
    pixclock:   39721,       bpp:        16,
    xres:       640,         yres:       480,
    hsync_len:  95,      vsync_len:  2,
    left_margin:     40,      upper_margin:   32,
    right_margin:    24,      lower_margin:   11,
    sync:       FB_SYNC_HOR_HIGH_ACT | FB_SYNC_VERT_HIGH_ACT,
    lccr0:      LCCR0_Color | LCCR0_Sngl | LCCR0_Act,
    lccr3:      LCCR3_OutEnL | LCCR3_PixFlEdg | LCCR3_ACBsDiv(512),
};
#endif

#ifdef CONFIG_SA1100_OMNIMETER
static struct sa1100fb_mach_info omnimeter_info __initdata = {
    pixclock:   0,       bpp:        4,
    xres:       480,         yres:       320,
    hsync_len:  1,       vsync_len:  1,
    left_margin:     10,      upper_margin:   0,
    right_margin:    10,      lower_margin:   0,
    cmap_greyscale: 1,
    sync:       FB_SYNC_HOR_HIGH_ACT | FB_SYNC_VERT_HIGH_ACT,
    lccr0:      LCCR0_Mono | LCCR0_Sngl | LCCR0_Pas | LCCR0_8PixMono,
    lccr3:      LCCR3_OutEnH | LCCR3_PixRsEdg | LCCR3_ACBsDiv(255) |
            LCCR3_PixClkDiv(44),
#error FIXME: fix pixclock, ACBsDiv
    /*
     * FIXME: I think ACBsDiv is wrong above - should it be 512 (disabled)?
     *   - rmk
     */
};
#endif

#ifdef CONFIG_SA1100_PANGOLIN
static struct sa1100fb_mach_info pangolin_info __initdata = {
    pixclock:   341521,      bpp:        16,
    xres:       800,         yres:       600,
```

```
    hsync_len: 64,     vsync_len: 7,
    left_margin:    160,        upper_margin:   7,
    right_margin:   24,     lower_margin:   1,
    sync:       FB_SYNC_HOR_HIGH_ACT | FB_SYNC_VERT_HIGH_ACT,
    lccr0:      LCCR0_Color | LCCR0_Sngl | LCCR0_Act,
    lccr3:      LCCR3_OutEnH | LCCR3_PixFlEdg,
};
#endif

#ifdef CONFIG_SA1100_XP860
static struct sa1100fb_mach_info xp860_info __initdata = {
    pixclock:   0,      bpp:        8,
    xres:       1024,       yres:       768,
    hsync_len:  3,      vsync_len:  3,
    left_margin:    3,      upper_margin:   2,
    right_margin:   2,      lower_margin:   1,
    sync:       0,
    lccr0:      LCCR0_Color | LCCR0_Sngl | LCCR0_Act,
    lccr3:      LCCR3_OutEnH | LCCR3_PixRsEdg | LCCR3_PixClkDiv(6),
};
#endif

static struct sa1100fb_mach_info * __init
sa1100fb_get_machine_info(struct sa1100fb_info *fbi)
{
    struct sa1100fb_mach_info *inf = NULL;
    /*
     *          R       G       B       T
     * default  {11,5}, { 5,6}, { 0,5}, { 0,0}
     * bitsy    {12,4}, { 7,4}, { 1,4}, { 0,0}
     * freebird { 8,4}, { 4,4}, { 0,4}, {12,4}
     */
#ifdef CONFIG_SA1100_ASSABET
    if (machine_is_assabet()) {
        inf = &assabet_info;
    }
#endif
#ifdef CONFIG_SA1100_BITSY
    if (machine_is_bitsy()) {
        inf = &bitsy_info;
        fbi->rgb[RGB_16] = &bitsy_rgb_16;
    }
#endif
#ifdef CONFIG_SA1100_BRUTUS
    if (machine_is_brutus()) {
        inf = &brutus_info;
    }
```

```
#endif
#ifdef CONFIG_SA1100_CERF
    if (machine_is_cerf()) {
        inf = &cerf_info;
    }
#endif
#ifdef CONFIG_SA1100_FREEBIRD
    if (machine_is_freebird()) {
        inf = &freebird_info;
        fbi->rgb[RGB_16] = &freebird_rgb16;
    }
#endif
#ifdef CONFIG_SA1100_GRAPHICSCLIENT
    if (machine_is_graphicsclient()) {
        inf = &graphicsclient_info;
    }
#endif
#ifdef CONFIG_SA1100_HUW_WEBPANEL
    if (machine_is_huw_webpanel()) {
        inf = &huw_webpanel_info;
    }
#endif
#ifdef CONFIG_SA1100_LART
    if (machine_is_lart()) {
#ifdef LART_GREY_LCD
        inf = &lart_grey_info;
#endif
#ifdef LART_COLOR_LCD
        inf = &lart_color_info;
#endif
#ifdef LART_VIDEO_OUT
        inf = &lart_video_info;
#endif
    }
#endif
#ifdef CONFIG_SA1100_OMNIMETER
    if (machine_is_omnimeter()) {
        inf = &omnimeter_info;
    }
#endif
#ifdef CONFIG_SA1100_PANGOLIN
    if (machine_is_pangolin()) {
        inf = &pangolin_info;
    }
#endif
#ifdef CONFIG_SA1100_XP860
    if (machine_is_xp860()) {
```

```
        inf = &xp860_info;
    }
#endif
    return inf;
}

static int sa1100fb_activate_var(struct fb_var_screeninfo *var, struct sa1100fb_info *);
static void set_ctrlr_state(struct sa1100fb_info *fbi, u_int state);

static inline void sa1100fb_schedule_task(struct sa1100fb_info *fbi, u_int state)
{
    unsigned long flags;
    local_irq_save(flags);
    /*
     * We need to handle two requests being made at the same time.
     * There are two important cases:
     *  1. When we are changing VT (C_REENABLE) while unblanking (C_ENABLE)
     *     We must perform the unblanking, which will do our REENABLE for us.
     *  2. When we are blanking, but immediately unblank before we have
     *     blanked.  We do the "REENABLE" thing here as well, just to be sure.
     */
    if (fbi->task_state == C_ENABLE && state == C_REENABLE)
        state = (u_int) -1;
    if (fbi->task_state == C_DISABLE && state == C_ENABLE)
        state = C_REENABLE;
    if (state != (u_int)-1) {
        fbi->task_state = state;
        schedule_task(&fbi->task);
    }
    local_irq_restore(flags);
}
/*
 * Get the VAR structure pointer for the specified console
 */
static inline struct fb_var_screeninfo *get_con_var(struct fb_info *info, int con)
{
    struct sa1100fb_info *fbi = (struct sa1100fb_info *)info;
    return (con == fbi->currcon || con == -1) ? &fbi->fb.var : &fb_display[con].var;
}
/*
 * Get the DISPLAY structure pointer for the specified console
 */
static inline struct display *get_con_display(struct fb_info *info, int con)
{
    struct sa1100fb_info *fbi = (struct sa1100fb_info *)info;
    return (con < 0) ? fbi->fb.disp : &fb_display[con];
}
```

```
/*
 * Get the CMAP pointer for the specified console
 */
static inline struct fb_cmap *get_con_cmap(struct fb_info *info, int con)
{
    struct sa1100fb_info *fbi = (struct sa1100fb_info *)info;
    return (con == fbi->currcon || con == -1) ? &fbi->fb.cmap : &fb_display[con].cmap;
}

static inline u_int
chan_to_field(u_int chan, struct fb_bitfield *bf)
{
    chan &= 0xffff;
    chan >>= 16 - bf->length;
    return chan << bf->offset;
}
/*
 * Convert bits-per-pixel to a hardware palette PBS value.
 */
static inline u_int
palette_pbs(struct fb_var_screeninfo *var)
{
    int ret = 0;
    switch (var->bits_per_pixel) {
#ifdef FBCON_HAS_CFB4
    case 4:  ret = 0 << 12; break;
#endif
#ifdef FBCON_HAS_CFB8
    case 8:  ret = 1 << 12; break;
#endif
#ifdef FBCON_HAS_CFB16
    case 12:
    case 16: ret = 2 << 12; break;
#endif
    }
    return ret;
}

static int
sa1100fb_setpalettereg(u_int regno, u_int red, u_int green, u_int blue,
               u_int trans, struct fb_info *info)
{
    struct sa1100fb_info *fbi = (struct sa1100fb_info *)info;
    u_int val, ret = 1;

    if (regno < fbi->palette_size) {
        val = ((red >> 4) & 0xf00);
```

```
        val |= ((green >> 8) & 0x0f0);
        val |= ((blue >> 12) & 0x00f);
        if (regno == 0)
            val |= palette_pbs(&fbi->fb.var);
        fbi->palette_cpu[regno] = val;
        ret = 0;
    }
    return ret;
}

static int
sa1100fb_setcolreg(u_int regno, u_int red, u_int green, u_int blue,
           u_int trans, struct fb_info *info)
{
    struct sa1100fb_info *fbi = (struct sa1100fb_info *)info;
    u_int val;
    int ret = 1;
    /*
     * If greyscale is true, then we convert the RGB value
     * to greyscale no mater what visual we are using.
     */
    if (fbi->fb.var.grayscale)
        red = green = blue = (19595 * red + 38470 * green +
                    7471 * blue) >> 16;
    switch (fbi->fb.disp->visual) {
    case FB_VISUAL_TRUECOLOR:
        /*
         * 12 or 16-bit True Colour.  We encode the RGB value
         * according to the RGB bitfield information.
         */
        if (regno < 16) {
            u16 *pal = fbi->fb.pseudo_palette;
            val  = chan_to_field(red, &fbi->fb.var.red);
            val |= chan_to_field(green, &fbi->fb.var.green);
            val |= chan_to_field(blue, &fbi->fb.var.blue);
            pal[regno] = val;
            ret = 0;
        }
        break;
    case FB_VISUAL_PSEUDOCOLOR:
        ret = sa1100fb_setpalettereg(regno, red, green, blue, trans, info);
        break;
    }
    return ret;
}

static int sa1100fb_validate_var(struct fb_var_screeninfo *var,
```

```
                  struct sa1100fb_info *fbi)
{
    int ret = -EINVAL;
    if (var->xres < MIN_XRES)
        var->xres = MIN_XRES;
    if (var->yres < MIN_YRES)
        var->yres = MIN_YRES;
    if (var->xres > fbi->max_xres)
        var->xres = fbi->max_xres;
    if (var->yres > fbi->max_yres)
        var->yres = fbi->max_yres;
    var->xres_virtual =
        var->xres_virtual < var->xres ? var->xres : var->xres_virtual;
    var->yres_virtual =
        var->yres_virtual < var->yres ? var->yres : var->yres_virtual;
    DPRINTK("var->bits_per_pixel=%d\n", var->bits_per_pixel);
    switch (var->bits_per_pixel) {
#ifdef FBCON_HAS_CFB4
    case 4:  ret = 0; break;
#endif
#ifdef FBCON_HAS_CFB8
    case 8:  ret = 0; break;
#endif
#ifdef FBCON_HAS_CFB16
    case 12:
        if ((fbi->lccr0 & LCCR0_PAS) == LCCR0_Pas)
            ret = 0;
        break;
    case 16:
        if ((fbi->lccr0 & LCCR0_PAS) == LCCR0_Act)
            ret = 0;
        break;
#endif
    default:
        break;
    }
    return ret;
}

static inline void sa1100fb_set_truecolor(u_int is_true_color)
{
    DPRINTK("true_color = %d\n", is_true_color);
#ifdef CONFIG_SA1100_ASSABET
    if (machine_is_assabet()) {
#if 1
        // phase 4 or newer Assabet's
        if (is_true_color)
```

```
            BCR_set(BCR_LCD_12RGB);
        else
            BCR_clear(BCR_LCD_12RGB);
#else
        // older Assabet's
        if (is_true_color)
            BCR_clear(BCR_LCD_12RGB);
        else
            BCR_set(BCR_LCD_12RGB);
#endif
    }
#endif
}

static void
sa1100fb_hw_set_var(struct fb_var_screeninfo *var, struct sa1100fb_info *fbi)
{
    u_long palette_mem_size;
    fbi->palette_size = var->bits_per_pixel == 8 ? 256 : 16;
    palette_mem_size = fbi->palette_size * sizeof(u16);
    DPRINTK("palette_mem_size = 0x%08lx\n", (u_long) palette_mem_size);
    fbi->palette_cpu = (u16 *)(fbi->map_cpu + PAGE_SIZE - palette_mem_size);
    fbi->palette_dma = fbi->map_dma + PAGE_SIZE - palette_mem_size;
    fb_set_cmap(&fbi->fb.cmap, 1, sa1100fb_setcolreg, &fbi->fb);
    /* Set board control register to handle new color depth */
    sa1100fb_set_truecolor(var->bits_per_pixel >= 16);
#ifdef CONFIG_SA1100_OMNIMETER
#error Do we have to do this here?  We already do it at init time.
    if (machine_is_omnimeter())
        SetLCDContrast(DefaultLCDContrast);
#endif
    sa1100fb_activate_var(var, fbi);
    fbi->palette_cpu[0] = (fbi->palette_cpu[0] &
                     0xcfff) | palette_pbs(var);
}
/*
 * sa1100fb_set_var():
 *  Set the user defined part of the display for the specified console
 */
static int
sa1100fb_set_var(struct fb_var_screeninfo *var, int con, struct fb_info *info)
{
    struct sa1100fb_info *fbi = (struct sa1100fb_info *)info;
    struct fb_var_screeninfo *dvar = get_con_var(&fbi->fb, con);
    struct display *display = get_con_display(&fbi->fb, con);
    int err, chgvar = 0, rgbidx;
    DPRINTK("set_var\n");
```

```
    /*
     * Decode var contents into a par structure, adjusting any
     * out of range values.
     */
    err = sa1100fb_validate_var(var, fbi);
    if (err)
        return err;
    if (var->activate & FB_ACTIVATE_TEST)
        return 0;
    if ((var->activate & FB_ACTIVATE_MASK) != FB_ACTIVATE_NOW)
        return -EINVAL;
    if (dvar->xres != var->xres)
        chgvar = 1;
    if (dvar->yres != var->yres)
        chgvar = 1;
    if (dvar->xres_virtual != var->xres_virtual)
        chgvar = 1;
    if (dvar->yres_virtual != var->yres_virtual)
        chgvar = 1;
    if (dvar->bits_per_pixel != var->bits_per_pixel)
        chgvar = 1;
    if (con < 0)
        chgvar = 0;
    switch (var->bits_per_pixel) {
#ifdef FBCON_HAS_CFB4
    case 4:
        if (fbi->cmap_static)
            display->visual = FB_VISUAL_STATIC_PSEUDOCOLOR;
        else
            display->visual = FB_VISUAL_PSEUDOCOLOR;
        display->line_length    = var->xres / 2;
        display->dispsw     = &fbcon_cfb4;
        rgbidx          = RGB_8;
        break;
#endif
#ifdef FBCON_HAS_CFB8
    case 8:
        if (fbi->cmap_static)
            display->visual = FB_VISUAL_STATIC_PSEUDOCOLOR;
        else
            display->visual = FB_VISUAL_PSEUDOCOLOR;
        display->line_length    = var->xres;
        display->dispsw     = &fbcon_cfb8;
        rgbidx          = RGB_8;
        break;
#endif
#ifdef FBCON_HAS_CFB16
```

```
    case 12:
    case 16:
        display->visual      = FB_VISUAL_TRUECOLOR;
        display->line_length    = var->xres * 2;
        display->dispsw      = &fbcon_cfb16;
        display->dispsw_data    = fbi->fb.pseudo_palette;
        rgbidx          = RGB_16;
        break;
#endif
    default:
        rgbidx = 0;
        display->dispsw = &fbcon_dummy;
        break;
    }
    display->screen_base    = fbi->screen_cpu;
    display->next_line = display->line_length;
    display->type       = fbi->fb.fix.type;
    display->type_aux   = fbi->fb.fix.type_aux;
    display->ypanstep   = fbi->fb.fix.ypanstep;
    display->ywrapstep = fbi->fb.fix.ywrapstep;
    display->can_soft_blank = 1;
    display->inverse    = 0;
    *dvar           = *var;
    dvar->activate      &= ~FB_ACTIVATE_ALL;
    /*
     * Copy the RGB parameters for this display
     * from the machine specific parameters.
     */
    dvar->red       = fbi->rgb[rgbidx]->red;
    dvar->green     = fbi->rgb[rgbidx]->green;
    dvar->blue      = fbi->rgb[rgbidx]->blue;
    dvar->transp        = fbi->rgb[rgbidx]->transp;
    DPRINTK("RGBT length = %d:%d:%d:%d\n",
        dvar->red.length, dvar->green.length, dvar->blue.length,
        dvar->transp.length);
    DPRINTK("RGBT offset = %d:%d:%d:%d\n",
        dvar->red.offset, dvar->green.offset, dvar->blue.offset,
        dvar->transp.offset);
    /*
     * Update the old var.  The fbcon drivers still use this.
     * Once they are using fbi->fb.var, this can be dropped.
     */
    display->var = *dvar;
    /*
     * If we are setting all the virtual consoles, also set the
     * defaults used to create new consoles.
     */
```

```
	if (var->activate & FB_ACTIVATE_ALL)
		fbi->fb.disp->var = *dvar;
	/*
	 * If the console has changed and the console has defined
	 * a changevar function, call that function.
	 */
	if (chgvar && info && fbi->fb.changevar)
		fbi->fb.changevar(con);
	/* If the current console is selected, activate the new var. */
	if (con != fbi->currcon)
		return 0;
	sa1100fb_hw_set_var(dvar, fbi);
	return 0;
}

static int
__do_set_cmap(struct fb_cmap *cmap, int kspc, int con,
	      struct fb_info *info)
{
	struct sa1100fb_info *fbi = (struct sa1100fb_info *)info;
	struct fb_cmap *dcmap = get_con_cmap(info, con);
	int err = 0;
	if (con == -1)
		con = fbi->currcon;
	/* no colormap allocated? (we always have "this" colour map allocated) */
	if (con >= 0)
		err = fb_alloc_cmap(&fb_display[con].cmap, fbi->palette_size, 0);
	if (!err && con == fbi->currcon)
		err = fb_set_cmap(cmap, kspc, sa1100fb_setcolreg, info);
	if (!err)
		fb_copy_cmap(cmap, dcmap, kspc ? 0 : 1);
	return err;
}

static int
sa1100fb_set_cmap(struct fb_cmap *cmap, int kspc, int con,
		  struct fb_info *info)
{
	struct display *disp = get_con_display(info, con);
	if (disp->visual == FB_VISUAL_TRUECOLOR)
		return -EINVAL;
	return __do_set_cmap(cmap, kspc, con, info);
}

static int
sa1100fb_get_fix(struct fb_fix_screeninfo *fix, int con, struct fb_info *info)
{
```

```
    struct display *display = get_con_display(info, con);
    *fix = info->fix;
    fix->line_length = display->line_length;
    fix->visual  = display->visual;
    return 0;
}

static int
sa1100fb_get_var(struct fb_var_screeninfo *var, int con, struct fb_info *info)
{
    *var = *get_con_var(info, con);
    return 0;
}

static int
sa1100fb_get_cmap(struct fb_cmap *cmap, int kspc, int con, struct fb_info *info)
{
    struct fb_cmap *dcmap = get_con_cmap(info, con);
    fb_copy_cmap(dcmap, cmap, kspc ? 0 : 2);
    return 0;
}
/*
 *  sa1100fb_switch():
 *  Change to the specified console.  Palette and video mode
 *      are changed to the console's stored parameters.
 *
 *  Uh oh, this can be called from a tasklet (IRQ)
 */
static int sa1100fb_switch(int con, struct fb_info *info)
{
    struct sa1100fb_info *fbi = (struct sa1100fb_info *)info;
    struct display *disp;
    struct fb_cmap *cmap;
    DPRINTK("con=%d info->modename=%s\n", con, fbi->fb.modename);
    if (con == fbi->currcon)
        return 0;
    if (fbi->currcon >= 0) {
        disp = fb_display + fbi->currcon;
        disp->var = fbi->fb.var;
        if (disp->cmap.len)
            fb_copy_cmap(&fbi->fb.cmap, &disp->cmap, 0);
    }
    fbi->currcon = con;
    disp = fb_display + con;
    fb_alloc_cmap(&fbi->fb.cmap, 256, 0);
    if (disp->cmap.len)
        cmap = &disp->cmap;
```

```
	else
		cmap = fb_default_cmap(1 << disp->var.bits_per_pixel);
	fb_copy_cmap(cmap, &fbi->fb.cmap, 0);
	fbi->fb.var = disp->var;
	fbi->fb.var.activate = FB_ACTIVATE_NOW;
	sa1100fb_set_var(&fbi->fb.var, con, info);
	return 0;
}

static void sa1100fb_blank(int blank, struct fb_info *info)
{
	struct sa1100fb_info *fbi = (struct sa1100fb_info *)info;
	int i;

	DPRINTK("sa1100fb_blank: blank=%d info->modename=%s\n", blank,
		fbi->fb.modename);
	switch (blank) {
	case VESA_POWERDOWN:
	case VESA_VSYNC_SUSPEND:
	case VESA_HSYNC_SUSPEND:
		if (fbi->fb.disp->visual == FB_VISUAL_PSEUDOCOLOR ||
			fbi->fb.disp->visual == FB_VISUAL_STATIC_PSEUDOCOLOR)
			for (i = 0; i < fbi->palette_size; i++)
				sa1100fb_setpalettereg(i, 0, 0, 0, 0, info);
		sa1100fb_schedule_task(fbi, C_DISABLE);
		if (sa1100fb_blank_helper)
			sa1100fb_blank_helper(blank);
		break;
	case VESA_NO_BLANKING:
		if (sa1100fb_blank_helper)
			sa1100fb_blank_helper(blank);
		if (fbi->fb.disp->visual == FB_VISUAL_PSEUDOCOLOR ||
			fbi->fb.disp->visual == FB_VISUAL_STATIC_PSEUDOCOLOR)
			fb_set_cmap(&fbi->fb.cmap, 1, sa1100fb_setcolreg, info);
		sa1100fb_schedule_task(fbi, C_ENABLE);
	}
}

static int sa1100fb_updatevar(int con, struct fb_info *info)
{
	DPRINTK("entered\n");
	return 0;
}
/*
 * Calculate the PCD value from the clock rate (in picoseconds).
 * We take account of the PPCR clock setting.
 */
```

```
static inline int get_pcd(unsigned int pixclock)
{
    unsigned int pcd;

    if (pixclock) {
        pcd = get_cclk_frequency() * pixclock;
        pcd /= 10000000;
        pcd += 1;    /* make up for integer math truncations */
    } else {
        printk(KERN_WARNING "Please convert me to use the PCD calculations\n");
        pcd = 0;
    }
    return pcd;
}
/*
 * sa1100fb_activate_var():
 * Configures LCD Controller based on entries in var parameter.  Settings are
 *      only written to the controller if changes were made.
 */
static int sa1100fb_activate_var(struct fb_var_screeninfo *var, struct sa1100fb_info *fbi)
{
    struct sa1100fb_lcd_reg new_regs;
    u_int pcd = get_pcd(var->pixclock);
    u_long flags;
    DPRINTK("Configuring SA1100 LCD\n");
    DPRINTK("var: xres=%d hslen=%d lm=%d rm=%d\n",
        var->xres, var->hsync_len,
        var->left_margin, var->right_margin);
    DPRINTK("var: yres=%d vslen=%d um=%d bm=%d\n",
        var->yres, var->vsync_len,
        var->upper_margin, var->lower_margin);
#if DEBUG_VAR
    if (var->xres < 16         || var->xres > 1024)
        printk(KERN_ERR "%s: invalid xres %d\n",
            fbi->fb.fix.id, var->xres);
    if (var->hsync_len < 1     || var->hsync_len > 64)
        printk(KERN_ERR "%s: invalid hsync_len %d\n",
            fbi->fb.fix.id, var->hsync_len);
    if (var->left_margin < 1   || var->left_margin > 255)
        printk(KERN_ERR "%s: invalid left_margin %d\n",
            fbi->fb.fix.id, var->left_margin);
    if (var->right_margin < 1 || var->right_margin > 255)
        printk(KERN_ERR "%s: invalid right_margin %d\n",
            fbi->fb.fix.id, var->right_margin);
    if (var->yres < 1          || var->yres > 1024)
        printk(KERN_ERR "%s: invalid yres %d\n",
            fbi->fb.fix.id, var->yres);
```

```
    if (var->vsync_len < 1    || var->vsync_len > 64)
        printk(KERN_ERR "%s: invalid vsync_len %d\n",
            fbi->fb.fix.id, var->vsync_len);
    if (var->upper_margin < 0 || var->upper_margin > 255)
        printk(KERN_ERR "%s: invalid upper_margin %d\n",
            fbi->fb.fix.id, var->upper_margin);
    if (var->lower_margin < 0 || var->lower_margin > 255)
        printk(KERN_ERR "%s: invalid lower_margin %d\n",
            fbi->fb.fix.id, var->lower_margin);
#endif
    new_regs.lccr0 = fbi->lccr0 |
        LCCR0_LEN | LCCR0_LDM | LCCR0_BAM |
        LCCR0_ERM | LCCR0_LtlEnd | LCCR0_DMADel(0);
    new_regs.lccr1 =
        LCCR1_DisWdth(var->xres) +
        LCCR1_HorSnchWdth(var->hsync_len) +
        LCCR1_BegLnDel(var->left_margin) +
        LCCR1_EndLnDel(var->right_margin);
    new_regs.lccr2 =
        LCCR2_DisHght(var->yres) +
        LCCR2_VrtSnchWdth(var->vsync_len) +
        LCCR2_BegFrmDel(var->upper_margin) +
        LCCR2_EndFrmDel(var->lower_margin);
    new_regs.lccr3 = fbi->lccr3 |
        (var->sync & FB_SYNC_HOR_HIGH_ACT ? LCCR3_HorSnchH : LCCR3_HorSnchL) |
        (var->sync & FB_SYNC_VERT_HIGH_ACT ? LCCR3_VrtSnchH : LCCR3_VrtSnchL) |
        LCCR3_ACBsCntOff;
    if (pcd)
        new_regs.lccr3 |= LCCR3_PixClkDiv(pcd);
    sa1100fb_check_shadow(&new_regs, var, pcd);
    DPRINTK("nlccr0 = 0x%08x\n", new_regs.lccr0);
    DPRINTK("nlccr1 = 0x%08x\n", new_regs.lccr1);
    DPRINTK("nlccr2 = 0x%08x\n", new_regs.lccr2);
    DPRINTK("nlccr3 = 0x%08x\n", new_regs.lccr3);
    /* Update shadow copy atomically */
    local_irq_save(flags);
    fbi->dbar1 = fbi->palette_dma;
    fbi->dbar2 = fbi->screen_dma +
          (var->xres * var->yres * var->bits_per_pixel / 8 / 2);
    fbi->reg_lccr0 = new_regs.lccr0;
    fbi->reg_lccr1 = new_regs.lccr1;
    fbi->reg_lccr2 = new_regs.lccr2;
    fbi->reg_lccr3 = new_regs.lccr3;
    local_irq_restore(flags);
    /*
     * Only update the registers if the controller is enabled
     * and something has changed.
```

```
     */
    if ((LCCR0 != fbi->reg_lccr0)       || (LCCR1 != fbi->reg_lccr1) ||
        (LCCR2 != fbi->reg_lccr2)       || (LCCR3 != fbi->reg_lccr3) ||
        (DBAR1 != (Address) fbi->dbar1) || (DBAR2 != (Address) fbi->dbar2))
        sa1100fb_schedule_task(fbi, C_REENABLE);
    return 0;
}

static void sa1100fb_backlight_on(struct sa1100fb_info *fbi)
{
    DPRINTK("backlight on\n");
#ifdef CONFIG_SA1100_FREEBIRD
#error FIXME
    if (machine_is_freebird()) {
        BCR_set(BCR_FREEBIRD_LCD_PWR | BCR_FREEBIRD_LCD_DISP);
    }
#endif
#ifdef CONFIG_SA1100_BITSY
#error FIXME
    if (machine_is_bitsy()) {
        set_bitsy_egpio(EGPIO_BITSY_LCD_ON |
                EGPIO_BITSY_LCD_PCI |
                EGPIO_BITSY_LCD_5V_ON |
                EGPIO_BITSY_LVDD_ON);
    }
#endif
#ifdef CONFIG_SA1100_FREEBIRD
    if (machine_is_freebird()) {
        /* Turn on backlight ,Chester */
        BCR_set(BCR_FREEBIRD_LCD_BACKLIGHT);
    }
#endif
#ifdef CONFIG_SA1100_HUW_WEBPANEL
#error FIXME
    if (machine_is_huw_webpanel()) {
        BCR_set(BCR_CCFL_POW + BCR_PWM_BACKLIGHT);
        set_current_state(TASK_UNINTERRUPTIBLE);
        schedule_task(200 * HZ / 1000);
        BCR_set(BCR_TFT_ENA);
    }
#endif
#ifdef CONFIG_SA1100_OMNIMETER
    if (machine_is_omnimeter())
        LEDBacklightOn();
#endif
}
```

```
static void sa1100fb_backlight_off(struct sa1100fb_info *fbi)
{
    DPRINTK("backlight off\n");
#ifdef CONFIG_SA1100_BITSY
#error FIXME
    if (machine_is_bitsy()) {
        clr_bitsy_egpio(EGPIO_BITSY_LCD_ON |
                EGPIO_BITSY_LCD_PCI |
                EGPIO_BITSY_LCD_5V_ON |
                EGPIO_BITSY_LVDD_ON);
    }
#endif
#ifdef CONFIG_SA1100_FREEBIRD
#error FIXME
    if (machine_is_freebird()) {
        BCR_clear(BCR_FREEBIRD_LCD_PWR | BCR_FREEBIRD_LCD_DISP
              /*| BCR_FREEBIRD_LCD_BACKLIGHT */ );
    }
#endif
#ifdef CONFIG_SA1100_OMNIMETER
    if (machine_is_omnimeter())
        LEDBacklightOff();
#endif
}

static void sa1100fb_power_up_lcd(struct sa1100fb_info *fbi)
{
    DPRINTK("LCD power on\n");
#if defined(CONFIG_SA1100_ASSABET) && !defined(ASSABET_PAL_VIDEO)
    if (machine_is_assabet())
        BCR_set(BCR_LCD_ON);
#endif
#ifdef CONFIG_SA1100_HUW_WEBPANEL
    if (machine_is_huw_webpanel())
        BCR_clear(BCR_TFT_NPWR);
#endif
#ifdef CONFIG_SA1100_OMNIMETER
    if (machine_is_omnimeter())
        LCDPowerOn();
#endif
}

static void sa1100fb_power_down_lcd(struct sa1100fb_info *fbi)
{
    DPRINTK("LCD power off\n");
#if defined(CONFIG_SA1100_ASSABET) && !defined(ASSABET_PAL_VIDEO)
    if (machine_is_assabet())
```

```
        BCR_clear(BCR_LCD_ON);
#endif
#ifdef CONFIG_SA1100_HUW_WEBPANEL
    // dont forget to set the control lines to zero (?)
    if (machine_is_huw_webpanel())
        BCR_set(BCR_TFT_NPWR);
#endif
}

static void sa1100fb_setup_gpio(struct sa1100fb_info *fbi)
{
    u_int mask = 0;
        if ((fbi->reg_lccr0 & LCCR0_CMS) == LCCR0_Color &&
        (fbi->reg_lccr0 & (LCCR0_Dual|LCCR0_Act)) != 0) {
        mask = GPIO_LDD11 | GPIO_LDD10 | GPIO_LDD9  | GPIO_LDD8;
        if (fbi->fb.var.bits_per_pixel > 8)
            mask |= GPIO_LDD15 | GPIO_LDD14 | GPIO_LDD13 | GPIO_LDD12;
    }

#ifdef CONFIG_SA1100_FREEBIRD
#error Please contact <rmk@arm.linux.org.uk> about this
    if (machine_is_freebird()) {
        /* Color single passive */
        mask |= GPIO_LDD15 | GPIO_LDD14 | GPIO_LDD13 | GPIO_LDD12 |
            GPIO_LDD11 | GPIO_LDD10 | GPIO_LDD9  | GPIO_LDD8;
    }
#endif
#ifdef CONFIG_SA1100_BITSY
#error Please contact <rmk@arm.linux.org.uk> about this, preferably with a patch!
    /*
     * This should be covered by the above test, so this is redundant.
     */
#if 0
    if (machine_is_bitsy()) {
        /* color single active */
        mask |= GPIO_LDD15 | GPIO_LDD14 | GPIO_LDD13 | GPIO_LDD12 |
                GPIO_LDD11 | GPIO_LDD10 | GPIO_LDD9  | GPIO_LDD8;
    }
#endif
#endif
#ifdef CONFIG_SA1100_CERF
#error Please contact <rmk@arm.linux.org.uk> about this
    if (machine_is_cerf()) {
        /* GPIO15 is used as a bypass for 3.8" displays */
        mask |= GPIO_GPIO15;
        mask |= GPIO_LDD15 | GPIO_LDD14 | GPIO_LDD13 | GPIO_LDD12 |
            GPIO_LDD11 | GPIO_LDD10 | GPIO_LDD9  | GPIO_LDD8;
```

```
    }
#endif
    if (mask) {
        GPDR |= mask;
        GAFR |= mask;
    }
}

static void sa1100fb_enable_controller(struct sa1100fb_info *fbi)
{
    DPRINTK("Enabling LCD controller\n");
    /*
     * Make sure the mode bits are present in the first palette entry
     */
    fbi->palette_cpu[0] &= 0xcfff;
    fbi->palette_cpu[0] |= palette_pbs(&fbi->fb.var);
    /* Sequence from 11.7.10 */
    LCCR3 = fbi->reg_lccr3;
    LCCR2 = fbi->reg_lccr2;
    LCCR1 = fbi->reg_lccr1;
    LCCR0 = fbi->reg_lccr0 & ~LCCR0_LEN;
    DBAR1 = (Address) fbi->dbar1;
    DBAR2 = (Address) fbi->dbar2;
    LCCR0 |= LCCR0_LEN;
#ifdef CONFIG_SA1100_GRAPHICSCLIENT
#error Where is GPIO24 set as an output?  Can we fit this in somewhere else?
    if (machine_is_graphicsclient()) {
        // From ADS doc again...same as disable
        set_current_state(TASK_UNINTERRUPTIBLE);
        schedule_timeout(20 * HZ / 1000);
        GPSR |= GPIO_GPIO24;
    }
#endif
    DPRINTK("DBAR1 = %p\n", DBAR1);
    DPRINTK("DBAR2 = %p\n", DBAR2);
    DPRINTK("LCCR0 = 0x%08x\n", LCCR0);
    DPRINTK("LCCR1 = 0x%08x\n", LCCR1);
    DPRINTK("LCCR2 = 0x%08x\n", LCCR2);
    DPRINTK("LCCR3 = 0x%08x\n", LCCR3);
}

static void sa1100fb_disable_controller(struct sa1100fb_info *fbi)
{
    DECLARE_WAITQUEUE(wait, current);
    DPRINTK("Disabling LCD controller\n");
#ifdef CONFIG_SA1100_GRAPHICSCLIENT
#error Where is GPIO24 set as an output?  Can we fit this in somewhere else?
```

```
    if (machine_is_graphicsclient()) {
        GPCR |= GPIO_GPIO24;
        set_current_state(TASK_UNINTERRUPTIBLE);
        schedule_timeout(20 * HZ / 1000);
    }
#endif
#ifdef CONFIG_SA1100_HUW_WEBPANEL
#error Move me into sa1100fb_power_up_lcd and/or sa1100fb_backlight_on
    if (machine_is_huw_webpanel()) {
        // dont forget to set the control lines to zero (?)
        DPRINTK("ShutDown HuW LCD controller\n");
        BCR_clear(BCR_TFT_ENA + BCR_CCFL_POW + BCR_PWM_BACKLIGHT);
    }
#endif
    add_wait_queue(&fbi->ctrlr_wait, &wait);
    set_current_state(TASK_UNINTERRUPTIBLE);
    LCSR = 0xffffffff;  /* Clear LCD Status Register */
    LCCR0 &= ~LCCR0_LDM;     /* Enable LCD Disable Done Interrupt */
    enable_irq(IRQ_LCD);     /* Enable LCD IRQ */
    LCCR0 &= ~LCCR0_LEN;     /* Disable LCD Controller */
    schedule_timeout(20 * HZ / 1000);
    current->state = TASK_RUNNING;
    remove_wait_queue(&fbi->ctrlr_wait, &wait);
}

/*This is my contribution. ALAIN DECOSTRE*/
static struct tq_struct pm_len_task;

long my_stop;

struct pm_len_arg {
    struct sa1100fb_info *info;
    int delay_dis;
    int delay_en;
};

static struct pm_len_arg my_pm_len_arg;

static void pm_len_function(struct pm_len_arg *my_arg) {

    while (my_stop==0) {
        sa1100fb_disable_controller(my_arg->info);
        schedule_timeout(my_arg->delay_dis);
        sa1100fb_enable_controller(my_arg->info);
        schedule_timeout(my_arg->delay_en);
    }
}
```

```
static void sa1100fb_enable_pm_len(struct fb_delay *del, struct sa1100fb_info *fbi)
{
    my_stop = 0;
    my_pm_len_arg.delay_dis = del->dis;
    my_pm_len_arg.delay_en = del->en;
    my_pm_len_arg.info = fbi;
    pm_len_task.routine = (void *)(void *) pm_len_function;
    pm_len_task.data = &my_pm_len_arg;
    schedule_task(&pm_len_task);
}

static void sa1100fb_disable_pm_len() {
    my_stop = 1;
}
/*End of my contribution. ALAIN DECOSTRE*/

static struct fb_ops sa1100fb_ops = {
    owner:      THIS_MODULE,
    fb_get_fix: sa1100fb_get_fix,
    fb_get_var: sa1100fb_get_var,
    fb_set_var: sa1100fb_set_var,
    fb_get_cmap:    sa1100fb_get_cmap,
    fb_set_cmap:    sa1100fb_set_cmap,
    fb_enable_controller:  sa1100fb_enable_controller,
    fb_disable_controller: sa1100fb_disable_controller,
    fb_enable_pm_len:   sa1100fb_enable_pm_len,
    fb_disable_pm_len:  sa1100fb_disable_pm_len
};
/*
 * sa1100fb_handle_irq: Handle 'LCD DONE' interrupts.
 */
static void sa1100fb_handle_irq(int irq, void *dev_id, struct pt_regs *regs)
{
    struct sa1100fb_info *fbi = dev_id;
    unsigned int lcsr = LCSR;
    if (lcsr & LCSR_LDD) {
        LCCR0 |= LCCR0_LDM;
        wake_up(&fbi->ctrlr_wait);
    }
    LCSR = lcsr;
}
/*
 * This function must be called from task context only, since it will
 * sleep when disabling the LCD controller, or if we get two contending
 * processes trying to alter state.
 */
```

```
static void set_ctrlr_state(struct sa1100fb_info *fbi, u_int state)
{
    u_int old_state;
    down(&fbi->ctrlr_sem);
    old_state = fbi->state;
    switch (state) {
    case C_DISABLE_CLKCHANGE:
        /*
         * Disable controller for clock change.  If the
         * controller is already disabled, then do nothing.
         */
        if (old_state != C_DISABLE) {
            fbi->state = state;
            sa1100fb_disable_controller(fbi);
        }
        break;
    case C_DISABLE:
        /*
         * Disable controller
         */
        if (old_state != C_DISABLE) {
            fbi->state = state;
            sa1100fb_backlight_off(fbi);
            if (old_state != C_DISABLE_CLKCHANGE)
                sa1100fb_disable_controller(fbi);
            sa1100fb_power_down_lcd(fbi);
        }
        break;
    case C_ENABLE_CLKCHANGE:
        /*
         * Enable the controller after clock change.  Only
         * do this if we were disabled for the clock change.
         */
        if (old_state == C_DISABLE_CLKCHANGE) {
            fbi->state = C_ENABLE;
            sa1100fb_enable_controller(fbi);
        }
        break;
case C_REENABLE:
        /*
         * Re-enable the controller only if it was already
         * enabled.  This is so we reprogram the control
         * registers.
         */
        if (old_state == C_ENABLE) {
            sa1100fb_disable_controller(fbi);
            sa1100fb_setup_gpio(fbi);
```

```
            sa1100fb_enable_controller(fbi);
        }
        break;
case C_ENABLE:
        /*
         * Power up the LCD screen, enable controller, and
         * turn on the backlight.
         */
        if (old_state != C_ENABLE) {
            fbi->state = C_ENABLE;
            sa1100fb_setup_gpio(fbi);
            sa1100fb_power_up_lcd(fbi);
            sa1100fb_enable_controller(fbi);
            sa1100fb_backlight_on(fbi);
        }
        break;
    }
    up(&fbi->ctrlr_sem);
}
/*
 * Our LCD controller task (which is called when we blank or unblank)
 * via keventd.
 */
static void sa1100fb_task(void *dummy)
{
    struct sa1100fb_info *fbi = dummy;
    u_int state = xchg(&fbi->task_state, -1);
    set_ctrlr_state(fbi, state);
}

#ifdef CONFIG_CPU_FREQ
/*
 * CPU clock speed change handler.  We need to adjust the LCD timing
 * parameters when the CPU clock is adjusted by the power management
 * subsystem.
 */
static int
sa1100fb_clkchg_notifier(struct notifier_block *nb, unsigned long val,
            void *data)
{
    struct sa1100fb_info *fbi = TO_INF(nb, clockchg);
    u_int pcd;
    switch (val) {
    case CPUFREQ_MINMAX:
        /* todo: fill in min/max values */
        break;
    case CPUFREQ_PRECHANGE:
```

```
        set_ctrlr_state(fbi, C_DISABLE_CLKCHANGE);
        break;
    case CPUFREQ_POSTCHANGE:
        pcd = get_pcd(fbi->fb.var.pixclock);
        fbi->reg_lccr3 = (fbi->reg_lccr3 & ~0xff) | LCCR3_PixClkDiv(pcd);
        set_ctrlr_state(fbi, C_ENABLE_CLKCHANGE);
        break;
    }
    return 0;
}
#endif

#ifdef CONFIG_PM

static int
sa1100fb_pm_callback(struct pm_dev *pm_dev, pm_request_t req, void *data)
{
    struct sa1100fb_info *fbi = pm_dev->data;
    DPRINTK("pm_callback: %d\n", req);
    if (req == PM_SUSPEND || req == PM_RESUME) {
        int state = (int)data;
        if (state == 0) {
            /* Enter D0. */
            set_ctrlr_state(fbi, C_ENABLE);
        } else {
            /* Enter D1-D3.  Disable the LCD controller.  */
            set_ctrlr_state(fbi, C_DISABLE);
        }
    }
    DPRINTK("done\n");
    return 0;
}
#endif

/*
 * sa1100fb_map_video_memory():
 *      Allocates the DRAM memory for the frame buffer.  This buffer is
 *  remapped into a non-cached, non-buffered, memory region to
 *      allow palette and pixel writes to occur without flushing the
 *      cache.  Once this area is remapped, all virtual memory
 *      access to the video memory should occur at the new region.
 */
static int __init sa1100fb_map_video_memory(struct sa1100fb_info *fbi)
{
    /*
     * We reserve one page for the palette, plus the size
     * of the framebuffer.
```

```
	 */
	fbi->map_size = PAGE_ALIGN(fbi->fb.fix.smem_len + PAGE_SIZE);
	fbi->map_cpu = consistent_alloc(GFP_KERNEL, fbi->map_size,
					&fbi->map_dma);
	if (fbi->map_cpu) {
		fbi->screen_cpu = fbi->map_cpu + PAGE_SIZE;
		fbi->screen_dma = fbi->map_dma + PAGE_SIZE;
		fbi->fb.fix.smem_start = fbi->screen_dma;
	}
	return fbi->map_cpu ? 0 : -ENOMEM;
}

/* Fake monspecs to fill in fbinfo structure */
static struct fb_monspecs monspecs __initdata = {
	30000, 70000, 50, 65, 0 /* Generic */
};

static struct sa1100fb_info * __init sa1100fb_init_fbinfo(void)
{
	struct sa1100fb_mach_info *inf;
	struct sa1100fb_info *fbi;
	fbi = kmalloc(sizeof(struct sa1100fb_info) + sizeof(struct display) +
		      sizeof(u16) * 16, GFP_KERNEL);
	if (!fbi)
		return NULL;
	memset(fbi, 0, sizeof(struct sa1100fb_info) + sizeof(struct display));
	fbi->currcon		= -1;
	strcpy(fbi->fb.fix.id, SA1100_NAME);
	fbi->fb.fix.type	= FB_TYPE_PACKED_PIXELS;
	fbi->fb.fix.type_aux	= 0;
	fbi->fb.fix.xpanstep	= 0;
	fbi->fb.fix.ypanstep	= 0;
	fbi->fb.fix.ywrapstep	= 0;
	fbi->fb.fix.accel	= FB_ACCEL_NONE;
	fbi->fb.var.nonstd	= 0;
	fbi->fb.var.activate	= FB_ACTIVATE_NOW;
	fbi->fb.var.height	= -1;
	fbi->fb.var.width	= -1;
	fbi->fb.var.accel_flags = 0;
	fbi->fb.var.vmode	= FB_VMODE_NONINTERLACED;
	strcpy(fbi->fb.modename, SA1100_NAME);
	strcpy(fbi->fb.fontname, "Acorn8x8");
	fbi->fb.fbops		= &sa1100fb_ops;
	fbi->fb.changevar	= NULL;
	fbi->fb.switch_con	= sa1100fb_switch;
	fbi->fb.updatevar	= sa1100fb_updatevar;
	fbi->fb.blank		= sa1100fb_blank;
```

```
    fbi->fb.flags        = FBINFO_FLAG_DEFAULT;
    fbi->fb.node         = -1;
    fbi->fb.monspecs     = monspecs;
    fbi->fb.disp         = (struct display *)(fbi + 1);
    fbi->fb.pseudo_palette  = (void *)(fbi->fb.disp + 1);
    fbi->rgb[RGB_8]      = &rgb_8;
    fbi->rgb[RGB_16]     = &def_rgb_16;
    inf = sa1100fb_get_machine_info(fbi);
    fbi->max_xres            = inf->xres;
    fbi->fb.var.xres         = inf->xres;
    fbi->fb.var.xres_virtual     = inf->xres;
    fbi->max_yres            = inf->yres;
    fbi->fb.var.yres         = inf->yres;
    fbi->fb.var.yres_virtual     = inf->yres;
    fbi->max_bpp             = inf->bpp;
    fbi->fb.var.bits_per_pixel   = inf->bpp;
    fbi->fb.var.pixclock         = inf->pixclock;
    fbi->fb.var.hsync_len        = inf->hsync_len;
    fbi->fb.var.left_margin      = inf->left_margin;
    fbi->fb.var.right_margin     = inf->right_margin;
    fbi->fb.var.vsync_len        = inf->vsync_len;
    fbi->fb.var.upper_margin     = inf->upper_margin;
    fbi->fb.var.lower_margin     = inf->lower_margin;
    fbi->fb.var.sync         = inf->sync;
    fbi->fb.var.grayscale        = inf->cmap_greyscale;
    fbi->cmap_inverse        = inf->cmap_inverse;
    fbi->cmap_static         = inf->cmap_static;
    fbi->lccr0           = inf->lccr0;
    fbi->lccr3           = inf->lccr3;
    fbi->state           = C_DISABLE;
    fbi->task_state          = (u_char)-1;
    fbi->fb.fix.smem_len         = fbi->max_xres * fbi->max_yres *
                fbi->max_bpp / 8;
    init_waitqueue_head(&fbi->ctrlr_wait);
    INIT_TQUEUE(&fbi->task, sa1100fb_task, fbi);
    init_MUTEX(&fbi->ctrlr_sem);
    return fbi;
}

int __init sa1100fb_init(void)
{
    struct sa1100fb_info *fbi;
    int ret;
    fbi = sa1100fb_init_fbinfo();
    ret = -ENOMEM;
    if (!fbi)
        goto failed;
```

```
        /* Initialize video memory */
        ret = sa1100fb_map_video_memory(fbi);
        if (ret)
            goto failed;
        ret = request_irq(IRQ_LCD, sa1100fb_handle_irq, SA_INTERRUPT,
                  fbi->fb.fix.id, fbi);
        if (ret) {
            printk(KERN_ERR "sa1100fb: failed in request_irq: %d\n", ret);
            goto failed;
        }
#if defined(CONFIG_SA1100_ASSABET) && defined(ASSABET_PAL_VIDEO)
        if (machine_is_assabet())
            BCR_clear(BCR_LCD_ON);
#endif
#ifdef CONFIG_SA1100_FREEBIRD
#error Please move this into sa1100fb_power_up_lcd.
        if (machine_is_freebird()) {
            BCR_set(BCR_FREEBIRD_LCD_DISP);
            mdelay(20);
            BCR_set(BCR_FREEBIRD_LCD_PWR);
            mdelay(20);
        }
#endif
        sa1100fb_set_var(&fbi->fb.var, -1, &fbi->fb);
        ret = register_framebuffer(&fbi->fb);
        if (ret < 0)
            goto failed;
#ifdef CONFIG_PM
        /*
         * Note that the console registers this as well, but we want to
         * power down the display prior to sleeping.
         */
        fbi->pm = pm_register(PM_SYS_DEV, PM_SYS_VGA, sa1100fb_pm_callback);
        if (fbi->pm)
            fbi->pm->data = fbi;
#endif
#ifdef CONFIG_CPU_FREQ
        fbi->clockchg.notifier_call = sa1100fb_clkchg_notifier;
        cpufreq_register_notifier(&fbi->clockchg);
#endif
        /*
         * Ok, now enable the LCD controller
         */
        set_ctrlr_state(fbi, C_ENABLE);
        /* This driver cannot be unloaded at the moment */
        MOD_INC_USE_COUNT;
        return 0;
```

```
failed:
    if (fbi)
        kfree(fbi);
    return ret;
}

int __init sa1100fb_setup(char *options)
{
#if 0
    char *this_opt;
    if (!options || !*options)
        return 0;
    for (this_opt = strtok(options, ","); this_opt;
         this_opt = strtok(NULL, ",")) {
        if (!strncmp(this_opt, "bpp:", 4))
            current_par.max_bpp =
                simple_strtoul(this_opt + 4, NULL, 0);
        if (!strncmp(this_opt, "lccr0:", 6))
            lcd_shadow.lccr0 =
                simple_strtoul(this_opt + 6, NULL, 0);
        if (!strncmp(this_opt, "lccr1:", 6)) {
            lcd_shadow.lccr1 =
                simple_strtoul(this_opt + 6, NULL, 0);
            current_par.max_xres =
                (lcd_shadow.lccr1 & 0x3ff) + 16;
        }
        if (!strncmp(this_opt, "lccr2:", 6)) {
            lcd_shadow.lccr2 =
                simple_strtoul(this_opt + 6, NULL, 0);
            current_par.max_yres =
                (lcd_shadow.
                 lccr0 & LCCR0_SDS) ? ((lcd_shadow.
                            lccr2 & 0x3ff) +
                           1) *
                2 : ((lcd_shadow.lccr2 & 0x3ff) + 1);
        }
        if (!strncmp(this_opt, "lccr3:", 6))
            lcd_shadow.lccr3 =
                simple_strtoul(this_opt + 6, NULL, 0);
    }
#endif
    return 0;
}
```

Appendix E

fb.h

```
#ifndef _LINUX_FB_H
#define _LINUX_FB_H
#include <linux/tty.h>
#include <asm/types.h>
/* Definitions of frame buffers                          */
#define FB_MAJOR          29
#define FB_MAX            32  /* sufficient for now */
/* ioctls
   0x46 is 'F'                                    */
#define FBIOGET_VSCREENINFO 0x4600
#define FBIOPUT_VSCREENINFO 0x4601
#define FBIOGET_FSCREENINFO 0x4602
#define FBIOGETCMAP       0x4604
#define FBIOPUTCMAP       0x4605
#define FBIOPAN_DISPLAY       0x4606
/* 0x4607-0x460B are defined below */
/* #define FBIOGET_MONITORSPEC  0x460C */
/* #define FBIOPUT_MONITORSPEC  0x460D */
/* #define FBIOSWITCH_MONIBIT   0x460E */
#define FBIOGET_CON2FBMAP   0x460F
#define FBIOPUT_CON2FBMAP   0x4610
#define FBIOBLANK         0x4611       /* arg: 0 or vesa level + 1 */
#define FBIOGET_VBLANK       _IOR('F', 0x12, struct fb_vblank)
#define FBIO_ALLOC              0x4613
#define FBIO_FREE               0x4614
#define FBIOGET_GLYPH           0x4615
#define FBIOGET_HWCINFO         0x4616
#define FBIOPUT_MODEINFO        0x4617
#define FBIOGET_DISPINFO        0x4618
/*I add those six lines*/
#define FBIODIS_CTRLR         0x460C
#define FBIOEN_CTRLR          0x460D
```

```
#define FBIOPMLEN_ON          0x4619
#define FBIOPMLEN_OFF         0x461A
#define FBIODARKENS     0x461B
#define FB_TYPE_PACKED_PIXELS        0   /* Packed Pixels    */
#define FB_TYPE_PLANES            1   /* Non interleaved planes */
#define FB_TYPE_INTERLEAVED_PLANES  2   /* Interleaved planes   */
#define FB_TYPE_TEXT              3   /* Text/attributes  */
#define FB_TYPE_VGA_PLANES        4   /* EGA/VGA planes   */
#define FB_AUX_TEXT_MDA       0   /* Monochrome text */
#define FB_AUX_TEXT_CGA       1   /* CGA/EGA/VGA Color text */
#define FB_AUX_TEXT_S3_MMIO 2   /* S3 MMIO fasttext */
#define FB_AUX_TEXT_MGA_STEP16  3   /* MGA Millenium I: text, attr, 14 reserved bytes */
#define FB_AUX_TEXT_MGA_STEP8   4   /* other MGAs:      text, attr,  6 reserved bytes */
#define FB_AUX_VGA_PLANES_VGA4       0   /* 16 color planes (EGA/VGA) */
#define FB_AUX_VGA_PLANES_CFB4       1   /* CFB4 in planes (VGA) */
#define FB_AUX_VGA_PLANES_CFB8       2   /* CFB8 in planes (VGA) */
#define FB_VISUAL_MONO01          0   /* Monochr. 1=Black 0=White */
#define FB_VISUAL_MONO10          1   /* Monochr. 1=White 0=Black */
#define FB_VISUAL_TRUECOLOR       2   /* True color    */
#define FB_VISUAL_PSEUDOCOLOR        3   /* Pseudo color (like atari) */
#define FB_VISUAL_DIRECTCOLOR        4   /* Direct color */
#define FB_VISUAL_STATIC_PSEUDOCOLOR    5   /* Pseudo color readonly */
#define FB_ACCEL_NONE         0   /* no hardware accelerator  */
#define FB_ACCEL_ATARIBLITT 1   /* Atari Blitter          */
#define FB_ACCEL_AMIGABLITT 2   /* Amiga Blitter               */
#define FB_ACCEL_S3_TRIO64  3   /* Cybervision64 (S3 Trio64)    */
#define FB_ACCEL_NCR_77C32BLT   4   /* RetinaZ3 (NCR 77C32BLT)      */
#define FB_ACCEL_S3_VIRGE   5   /* Cybervision64/3D (S3 ViRGE)  */
#define FB_ACCEL_ATI_MACH64GX   6   /* ATI Mach 64GX family     */
#define FB_ACCEL_DEC_TGA    7   /* DEC 21030 TGA        */
#define FB_ACCEL_ATI_MACH64CT   8   /* ATI Mach 64CT family     */
#define FB_ACCEL_ATI_MACH64VT   9   /* ATI Mach 64CT family VT class */
#define FB_ACCEL_ATI_MACH64GT   10  /* ATI Mach 64CT family GT class */
#define FB_ACCEL_SUN_CREATOR    11  /* Sun Creator/Creator3D    */
#define FB_ACCEL_SUN_CGSIX  12  /* Sun cg6          */
#define FB_ACCEL_SUN_LEO    13  /* Sun leo/zx           */
#define FB_ACCEL_IMS_TWINTURBO  14  /* IMS Twin Turbo       */
#define FB_ACCEL_3DLABS_PERMEDIA2 15    /* 3Dlabs Permedia 2        */
#define FB_ACCEL_MATROX_MGA2064W 16 /* Matrox MGA2064W (Millenium)  */
#define FB_ACCEL_MATROX_MGA1064SG 17    /* Matrox MGA1064SG (Mystique)  */
#define FB_ACCEL_MATROX_MGA2164W 18 /* Matrox MGA2164W (Millenium II) */
#define FB_ACCEL_MATROX_MGA2164W_AGP 19 /* Matrox MGA2164W (Millenium II) */
#define FB_ACCEL_MATROX_MGAG100 20  /* Matrox G100 (Productiva G100) */
#define FB_ACCEL_MATROX_MGAG200 21  /* Matrox G200 (Myst, Mill, ...) */
#define FB_ACCEL_SUN_CG14   22  /* Sun cgfourteen        */
#define FB_ACCEL_SUN_BWTWO  23  /* Sun bwtwo            */
#define FB_ACCEL_SUN_CGTHREE    24  /* Sun cgthree          */
```

```
#define FB_ACCEL_SUN_TCX      25  /* Sun tcx          */
#define FB_ACCEL_MATROX_MGAG400 26  /* Matrox G400          */
#define FB_ACCEL_NV3          27  /* nVidia RIVA 128          */
#define FB_ACCEL_NV4          28  /* nVidia RIVA TNT      */
#define FB_ACCEL_NV5          29  /* nVidia RIVA TNT2     */
#define FB_ACCEL_CT_6555x     30  /* C&T 6555x            */
#define FB_ACCEL_3DFX_BANSHEE   31  /* 3Dfx Banshee          */
#define FB_ACCEL_ATI_RAGE128    32  /* ATI Rage128 family        */
#define FB_ACCEL_IGS_CYBER2000  33  /* CyberPro 2000         */
#define FB_ACCEL_IGS_CYBER2010  34  /* CyberPro 2010         */
#define FB_ACCEL_IGS_CYBER5000  35  /* CyberPro 5000         */
#define FB_ACCEL_SIS_GLAMOUR    36  /* SiS 300/630/540              */

struct fb_fix_screeninfo {
    char id[16];            /* identification string eg "TT Builtin" */
    unsigned long smem_start;   /* Start of frame buffer mem */
                /* (physical address) */
    __u32 smem_len;         /* Length of frame buffer mem */
    __u32 type;         /* see FB_TYPE_*        */
    __u32 type_aux;         /* Interleave for interleaved Planes */
    __u32 visual;           /* see FB_VISUAL_*      */
    __u16 xpanstep;         /* zero if no hardware panning  */
    __u16 ypanstep;         /* zero if no hardware panning  */
    __u16 ywrapstep;        /* zero if no hardware ywrap    */
    __u32 line_length;      /* length of a line in bytes    */
    unsigned long mmio_start;   /* Start of Memory Mapped I/O   */
                /* (physical address) */
    __u32 mmio_len;         /* Length of Memory Mapped I/O  */
    __u32 accel;            /* Type of acceleration available */
    __u16 reserved[3];      /* Reserved for future compatibility */
};
/* Interpretation of offset for color fields: All offsets are from the right,
 * inside a "pixel" value, which is exactly 'bits_per_pixel' wide (means: you
 * can use the offset as right argument to <<). A pixel afterwards is a bit
 * stream and is written to video memory as that unmodified. This implies
 * big-endian byte order if bits_per_pixel is greater than 8.
 */
struct fb_bitfield {
    __u32 offset;           /* beginning of bitfield    */
    __u32 length;           /* length of bitfield       */
    __u32 msb_right;        /* != 0 : Most significant bit is */
                /* right */
};
#define FB_NONSTD_HAM       1   /* Hold-And-Modify (HAM)        */
#define FB_ACTIVATE_NOW     0   /* set values immediately (or vbl)*/
#define FB_ACTIVATE_NXTOPEN 1   /* activate on next open    */
#define FB_ACTIVATE_TEST    2   /* don't set, round up impossible */
```

```
#define FB_ACTIVATE_MASK        15
                /* values           */
#define FB_ACTIVATE_VBL         16  /* activate values on next vbl  */
#define FB_CHANGE_CMAP_VBL      32  /* change colormap on vbl   */
#define FB_ACTIVATE_ALL         64  /* change all VCs on this fb    */
#define FB_ACCELF_TEXT      1   /* text mode acceleration */
#define FB_SYNC_HOR_HIGH_ACT    1   /* horizontal sync high active  */
#define FB_SYNC_VERT_HIGH_ACT   2   /* vertical sync high active    */
#define FB_SYNC_EXT     4   /* external sync        */
#define FB_SYNC_COMP_HIGH_ACT   8   /* composite sync high active   */
#define FB_SYNC_BROADCAST   16  /* broadcast video timings      */
                /* vtotal = 144d/288n/576i => PAL  */
                /* vtotal = 121d/242n/484i => NTSC */
#define FB_SYNC_ON_GREEN    32  /* sync on green */
#define FB_VMODE_NONINTERLACED  0   /* non interlaced */
#define FB_VMODE_INTERLACED 1   /* interlaced   */
#define FB_VMODE_DOUBLE     2   /* double scan */
#define FB_VMODE_MASK       255
#define FB_VMODE_YWRAP      256 /* ywrap instead of panning     */
#define FB_VMODE_SMOOTH_XPAN    512 /* smooth xpan possible (internally used) */
#define FB_VMODE_CONUPDATE  512 /* don't update x/yoffset   */

struct fb_var_screeninfo {
    __u32 xres;         /* visible resolution       */
    __u32 yres;
    __u32 xres_virtual;     /* virtual resolution       */
    __u32 yres_virtual;
    __u32 xoffset;          /* offset from virtual to visible */
    __u32 yoffset;          /* resolution           */
    __u32 bits_per_pixel;       /* guess what           */
    __u32 grayscale;        /* != 0 Graylevels instead of colors */
    struct fb_bitfield red;     /* bitfield in fb mem if true color, */
    struct fb_bitfield green;   /* else only length is significant */
    struct fb_bitfield blue;
    struct fb_bitfield transp;  /* transparency         */
    __u32 nonstd;           /* != 0 Non standard pixel format */
    __u32 activate;         /* see FB_ACTIVATE_*        */
    __u32 height;           /* height of picture in mm    */
    __u32 width;            /* width of picture in mm     */
    __u32 accel_flags;      /* acceleration flags (hints)   */
    /* Timing: All values in pixclocks, except pixclock (of course) */
    __u32 pixclock;         /* pixel clock in ps (pico seconds) */
    __u32 left_margin;      /* time from sync to picture    */
    __u32 right_margin;     /* time from picture to sync    */
    __u32 upper_margin;     /* time from sync to picture    */
    __u32 lower_margin;
    __u32 hsync_len;        /* length of horizontal sync    */
```

```
    __u32 vsync_len;        /* length of vertical sync  */
    __u32 sync;         /* see FB_SYNC_*        */
    __u32 vmode;            /* see FB_VMODE_*       */
    __u32 reserved[6];      /* Reserved for future compatibility */
};

/*I add this declaration*/
struct fb_delay {
    long dis;
    long en;
};

struct fb_cmap {
    __u32 start;            /* First entry  */
    __u32 len;          /* Number of entries */
    __u16 *red;         /* Red values   */
    __u16 *green;
    __u16 *blue;
    __u16 *transp;          /* transparency, can be NULL */
};

struct fb_con2fbmap {
    __u32 console;
    __u32 framebuffer;
};
/* VESA Blanking Levels */
#define VESA_NO_BLANKING        0
#define VESA_VSYNC_SUSPEND      1
#define VESA_HSYNC_SUSPEND      2
#define VESA_POWERDOWN          3

struct fb_monspecs {
    __u32 hfmin;            /* hfreq lower limit (Hz) */
    __u32 hfmax;            /* hfreq upper limit (Hz) */
    __u16 vfmin;            /* vfreq lower limit (Hz) */
    __u16 vfmax;            /* vfreq upper limit (Hz) */
    unsigned dpms : 1;      /* supports DPMS */
};
#define FB_VBLANK_VBLANKING 0x001   /* currently in a vertical blank */
#define FB_VBLANK_HBLANKING 0x002   /* currently in a horizontal blank */
#define FB_VBLANK_HAVE_VBLANK   0x004   /* vertical blanks can be detected */
#define FB_VBLANK_HAVE_HBLANK   0x008   /* horizontal blanks can be detected */
#define FB_VBLANK_HAVE_COUNT    0x010   /* global retrace counter is available */
#define FB_VBLANK_HAVE_VCOUNT   0x020   /* the vcount field is valid */
#define FB_VBLANK_HAVE_HCOUNT   0x040   /* the hcount field is valid */
#define FB_VBLANK_VSYNCING  0x080   /* currently in a vsync */
#define FB_VBLANK_HAVE_VSYNC    0x100   /* verical syncs can be detected */
```

```
struct fb_vblank {
    __u32 flags;            /* FB_VBLANK flags */
    __u32 count;            /* counter of retraces since boot */
    __u32 vcount;           /* current scanline position */
    __u32 hcount;           /* current scandot position */
    __u32 reserved[4];      /* reserved for future compatibility */
};
#ifdef __KERNEL__
#if 1 /* to go away in 2.5.0 */
extern int GET_FB_IDX(kdev_t rdev);
#else
#define GET_FB_IDX(node)    (MINOR(node))
#endif
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/devfs_fs_kernel.h>

struct fb_info;
struct fb_info_gen;
struct vm_area_struct;
struct file;
    /*
     * Frame buffer operations
     */
struct fb_ops {
    /* open/release and usage marking */
    struct module *owner;
    int (*fb_open)(struct fb_info *info, int user);
    int (*fb_release)(struct fb_info *info, int user);
    /* get non settable parameters */
    int (*fb_get_fix)(struct fb_fix_screeninfo *fix, int con,
              struct fb_info *info);
    /* get settable parameters */
    int (*fb_get_var)(struct fb_var_screeninfo *var, int con,
              struct fb_info *info);
    /* set settable parameters */
    int (*fb_set_var)(struct fb_var_screeninfo *var, int con,
              struct fb_info *info);
    /* get colormap */
    int (*fb_get_cmap)(struct fb_cmap *cmap, int kspc, int con,
               struct fb_info *info);
    /* set colormap */
    int (*fb_set_cmap)(struct fb_cmap *cmap, int kspc, int con,
               struct fb_info *info);
    /* pan display (optional) */
    int (*fb_pan_display)(struct fb_var_screeninfo *var, int con,
              struct fb_info *info);
```

```
    /* perform fb specific ioctl (optional) */
    int (*fb_ioctl)(struct inode *inode, struct file *file, unsigned int cmd,
            unsigned long arg, int con, struct fb_info *info);
    /* perform fb specific mmap */
    int (*fb_mmap)(struct fb_info *info, struct file *file, struct vm_area_struct *vma);
    /* switch to/from raster image mode */
    int (*fb_rasterimg)(struct fb_info *info, int start);
    void (*fb_disable_controller)(struct fb_info *info);//I add this line
    void (*fb_enable_controller)(struct fb_info *info);//I add this line
    void (*fb_enable_pm_len)(struct fb_delay *del,struct fb_info *info);//I add this line
    void (*fb_disable_pm_len)();//I add this line
};

struct fb_info {
   char modename[40];           /* default video mode */
   kdev_t node;
   int flags;
   int open;                            /* Has this been open already ? */
#define FBINFO_FLAG_MODULE  1   /* Low-level driver is a module */
   struct fb_var_screeninfo var;        /* Current var */
   struct fb_fix_screeninfo fix;        /* Current fix */
   struct fb_monspecs monspecs;         /* Current Monitor specs */
   struct fb_cmap cmap;                 /* Current cmap */
   struct fb_ops *fbops;
   char *screen_base;                   /* Virtual address */
   struct display *disp;        /* initial display variable */
   struct vc_data *display_fg;      /* Console visible on this display */
   char fontname[40];           /* default font name */
   devfs_handle_t devfs_handle;         /* Devfs handle for new name         */
   devfs_handle_t devfs_lhandle;        /* Devfs handle for compat. symlink  */
   int (*changevar)(int);       /* tell console var has changed */
   int (*switch_con)(int, struct fb_info*);
                     /* tell fb to switch consoles */
   int (*updatevar)(int, struct fb_info*);
                     /* tell fb to update the vars */
   void (*blank)(int, struct fb_info*); /* tell fb to (un)blank the screen */
                     /* arg = 0: unblank */
                     /* arg > 0: VESA level (arg-1) */
   void *pseudo_palette;                /* Fake palette of 16 colors and
                         the cursor's color for non
                                           palette mode */
   /* From here on everything is device dependent */
   void *par;
};

#ifdef MODULE
#define FBINFO_FLAG_DEFAULT FBINFO_FLAG_MODULE
```

```
#else
#define FBINFO_FLAG_DEFAULT 0
#endif
    /*
     * This structure abstracts from the underlying hardware. It is not
     * mandatory but used by the 'generic' frame buffer operations.
     * Read drivers/video/skeletonfb.c for more information.
     */
struct fbgen_hwswitch {
    void (*detect)(void);
    int (*encode_fix)(struct fb_fix_screeninfo *fix, const void *par,
              struct fb_info_gen *info);
    int (*decode_var)(const struct fb_var_screeninfo *var, void *par,
              struct fb_info_gen *info);
    int (*encode_var)(struct fb_var_screeninfo *var, const void *par,
              struct fb_info_gen *info);
    void (*get_par)(void *par, struct fb_info_gen *info);
    void (*set_par)(const void *par, struct fb_info_gen *info);
    int (*getcolreg)(unsigned regno, unsigned *red, unsigned *green,
             unsigned *blue, unsigned *transp, struct fb_info *info);
    int (*setcolreg)(unsigned regno, unsigned red, unsigned green,
             unsigned blue, unsigned transp, struct fb_info *info);
    int (*pan_display)(const struct fb_var_screeninfo *var,
               struct fb_info_gen *info);
    int (*blank)(int blank_mode, struct fb_info_gen *info);
    void (*set_disp)(const void *par, struct display *disp,
             struct fb_info_gen *info);
};
struct fb_info_gen {
    struct fb_info info;
    /* Entries for a generic frame buffer device */
    /* Yes, this starts looking like C++ */
    u_int parsize;
    struct fbgen_hwswitch *fbhw;
   /* From here on everything is device dependent */
};
    /*
     * 'Generic' versions of the frame buffer device operations
     */
extern int fbgen_get_fix(struct fb_fix_screeninfo *fix, int con,
             struct fb_info *info);
extern int fbgen_get_var(struct fb_var_screeninfo *var, int con,
             struct fb_info *info);
extern int fbgen_set_var(struct fb_var_screeninfo *var, int con,
             struct fb_info *info);
extern int fbgen_get_cmap(struct fb_cmap *cmap, int kspc, int con,
              struct fb_info *info);
```

```
extern int fbgen_set_cmap(struct fb_cmap *cmap, int kspc, int con,
              struct fb_info *info);
extern int fbgen_pan_display(struct fb_var_screeninfo *var, int con,
                 struct fb_info *info);
    /*
     *  Helper functions
     */
extern int fbgen_do_set_var(struct fb_var_screeninfo *var, int isactive,
                struct fb_info_gen *info);
extern void fbgen_set_disp(int con, struct fb_info_gen *info);
extern void fbgen_install_cmap(int con, struct fb_info_gen *info);
extern int fbgen_update_var(int con, struct fb_info *info);
extern int fbgen_switch(int con, struct fb_info *info);
extern void fbgen_blank(int blank, struct fb_info *info);

/* drivers/video/fbmem.c */
extern int register_framebuffer(struct fb_info *fb_info);
extern int unregister_framebuffer(struct fb_info *fb_info);
extern int num_registered_fb;
extern struct fb_info *registered_fb[FB_MAX];
/* drivers/video/fbmon.c */
extern int fbmon_valid_timings(u_int pixclock, u_int htotal, u_int vtotal,
                   const struct fb_info *fb_info);
extern int fbmon_dpms(const struct fb_info *fb_info);
/* drivers/video/fbcmap.c */
extern int fb_alloc_cmap(struct fb_cmap *cmap, int len, int transp);
extern void fb_copy_cmap(struct fb_cmap *from, struct fb_cmap *to,
             int fsfromto);
extern int fb_get_cmap(struct fb_cmap *cmap, int kspc,
               int (*getcolreg)(u_int, u_int *, u_int *, u_int *,
                    u_int *, struct fb_info *),
               struct fb_info *fb_info);
extern int fb_set_cmap(struct fb_cmap *cmap, int kspc,
               int (*setcolreg)(u_int, u_int, u_int, u_int, u_int,
                    struct fb_info *),
               struct fb_info *fb_info);
extern struct fb_cmap *fb_default_cmap(int len);
extern void fb_invert_cmaps(void);

struct fb_videomode {
    const char *name;   /* optional */
    u32 refresh;    /* optional */
    u32 xres;
    u32 yres;
    u32 pixclock;
    u32 left_margin;
    u32 right_margin;
```

```
    u32 upper_margin;
    u32 lower_margin;
    u32 hsync_len;
    u32 vsync_len;
    u32 sync;
    u32 vmode;
};

#ifdef MODULE
static inline int fb_find_mode(struct fb_var_screeninfo *var,
                   struct fb_info *info, const char *mode_option,
                   const struct fb_videomode *db,
                   unsigned int dbsize,
                   const struct fb_videomode *default_mode,
                   unsigned int default_bpp)
{
    extern int __fb_try_mode(struct fb_var_screeninfo *var,
                    struct fb_info *info,
                  const struct fb_videomode *mode,
                  unsigned int bpp);
    /*
     *  FIXME: How to make the compiler optimize vga640x400 away if
     *         default_mode is non-NULL?
     */
    static const struct fb_videomode vga640x400 = {
    /* 640x400 @ 70 Hz, 31.5 kHz hsync */
    NULL, 70, 640, 400, 39721, 40, 24, 39, 9, 96, 2,
    0, FB_VMODE_NONINTERLACED
    };
    if (!default_mode)
    default_mode = &vga640x400;
    if (!default_bpp)
    default_bpp = 8;
    return __fb_try_mode(var, info, default_mode, default_bpp);
}
#else
extern int __init fb_find_mode(struct fb_var_screeninfo *var,
                   struct fb_info *info, const char *mode_option,
                   const struct fb_videomode *db,
                   unsigned int dbsize,
                   const struct fb_videomode *default_mode,
                   unsigned int default_bpp);
#endif

#endif /* __KERNEL__ */

#if 1
```

```
#define FBCMD_GET_CURRENTPAR     0xDEAD0005
#define FBCMD_SET_CURRENTPAR     0xDEAD8005

#endif

#if 1 /* Preliminary */
   /*
    *    Hardware Cursor
    */
#define FBIOGET_FCURSORINFO     0x4607
#define FBIOGET_VCURSORINFO     0x4608
#define FBIOPUT_VCURSORINFO     0x4609
#define FBIOGET_CURSORSTATE     0x460A
#define FBIOPUT_CURSORSTATE     0x460B

struct fb_fix_cursorinfo {
    __u16 crsr_width;        /* width and height of the cursor in */
    __u16 crsr_height;       /* pixels (zero if no cursor)    */
    __u16 crsr_xsize;        /* cursor size in display pixels */
    __u16 crsr_ysize;
    __u16 crsr_color1;       /* colormap entry for cursor color1 */
    __u16 crsr_color2;       /* colormap entry for cursor color2 */
};

struct fb_var_cursorinfo {
    __u16 width;
    __u16 height;
    __u16 xspot;
    __u16 yspot;
    __u8 data[1];            /* field with [height][width]        */
};

struct fb_cursorstate {
    __s16 xoffset;
    __s16 yoffset;
    __u16 mode;
};

#define FB_CURSOR_OFF       0
#define FB_CURSOR_ON        1
#define FB_CURSOR_FLASH     2
#endif /* Preliminary */
#endif /* _LINUX_FB_H */
```

Appendix F

sa1100fb.h

```
/*
 * linux/drivers/video/sa1100fb.h
 *    -- StrongARM 1100 LCD Controller Frame Buffer Device
 *
 *  Copyright (C) 1999 Eric A. Thomas
 *   Based on acornfb.c Copyright (C) Russell King.
 *
 * This file is subject to the terms and conditions of the GNU General Public
 * License.  See the file COPYING in the main directory of this archive
 * for more details.
 */
/*
 * These are the bitfields for each
 * display depth that we support.
 */
struct sa1100fb_rgb {
    struct fb_bitfield  red;
    struct fb_bitfield  green;
    struct fb_bitfield  blue;
    struct fb_bitfield  transp;
};
/*
 * This structure describes the machine which we are running on.
 */
struct sa1100fb_mach_info {
    u_long      pixclock;
    u_short     xres;
    u_short     yres;
    u_char      bpp;
    u_char      hsync_len;
    u_char      left_margin;
    u_char      right_margin;
```

```
    u_char      vsync_len;
    u_char      upper_margin;
    u_char      lower_margin;
    u_char      sync;
    u_int       cmap_greyscale:1,
            cmap_inverse:1,
            cmap_static:1,
            unused:29;
    u_int       lccr0;
    u_int       lccr3;
};
/* Shadows for LCD controller registers */
struct sa1100fb_lcd_reg {
    Word lccr0;
    Word lccr1;
    Word lccr2;
    Word lccr3;
};

#define RGB_8   (0)
#define RGB_16  (1)
#define NR_RGB  2

struct sa1100fb_info {
    struct fb_info      fb;
    signed int      currcon;
    struct sa1100fb_rgb *rgb[NR_RGB];
    u_int           max_bpp;
    u_int           max_xres;
    u_int           max_yres;
    /*
     * These are the addresses we mapped
     * the framebuffer memory region to.
     */
    dma_addr_t      map_dma;
    u_char *        map_cpu;
    u_int           map_size;
    u_char *        screen_cpu;
    dma_addr_t      screen_dma;
    u16 *           palette_cpu;
    dma_addr_t      palette_dma;
    u_int           palette_size;
    dma_addr_t      dbar1;
    dma_addr_t      dbar2;
    u_int           lccr0;
    u_int           lccr3;
    u_int           cmap_inverse:1,
```

```
                cmap_static:1,
                unused:30;
    u_int           reg_lccr0;
    u_int           reg_lccr1;
    u_int           reg_lccr2;
    u_int           reg_lccr3;
    volatile u_char     state;
    volatile u_char     task_state;
    struct semaphore    ctrlr_sem;
    wait_queue_head_t   ctrlr_wait;
    struct tq_struct    task;

#ifdef CONFIG_PM
    struct pm_dev       *pm;
#endif
#ifdef CONFIG_CPU_FREQ
    struct notifier_block   clockchg;
#endif
};
#define __type_entry(ptr,type,member) ((type *)((char *)(ptr)-offsetof(type,member)))
#define TO_INF(ptr,member)  __type_entry(ptr,struct sa1100fb_info,member)
#define SA1100_PALETTE_MODE_VAL(bpp)    (((bpp) & 0x018) << 9)
/*
 * These are the actions for set_ctrlr_state
 */
#define C_DISABLE       (0)
#define C_ENABLE        (1)
#define C_DISABLE_CLKCHANGE (2)
#define C_ENABLE_CLKCHANGE  (3)
#define C_REENABLE      (4)
#define SA1100_NAME "SA1100"
/*
 *  Debug macros
 */
#if DEBUG
#  define DPRINTK(fmt, args...) printk("%s: " fmt, __FUNCTION__ , ## args)
#else
#  define DPRINTK(fmt, args...)
#endif
/*
 * Minimum X and Y resolutions
 */
#define MIN_XRES    64
#define MIN_YRES    64
```