UFEEJ4-40-3 - Individual Project (CRTS)

Student:	Edwin Langley 02503280
Project supervisor:	Craig Duffy
Second reader:	Nigel Gunton
Award title:	BSc Computing for Real Time Systems
Project title:	uClinux on the Pluto 6 embedded control board
Word count:	16500

Synopsis

This project is based around a control board known as the Pluto 6, produced by a company called Heber Ltd. The aim of the project was to port an operating system to the Pluto 6 board. Following a successful port, drivers were to be written for some of the custom hardware on the board.
The design techniques used to develop the drivers for the project depended on the level of interaction required with the rest of the kernel. The frame buffer driver written was a bottom-up design. The operation of the hardware, and the behaviour expected by the rest of the kernel was known, so algorithms were then designed to bridge the gap between these layers until the video card was utilised correctly by the kernel. Conversely, the VFD and multiplexed lamp drivers were a top-down design. These drivers had no major subsystem, allowing flexibility. The interface required by user programs was first defined, and the code to drive the hardware designed accordingly.
The programming language used throughout was C, except for the kernel entry code, which was GNU 68K assembly. The equipment used in the project, aside from the board itself, was a P&E BDM debugging module, CompactFlash card reader, and serial cable. These components were used with an X86 development host running Linux.
The progress made on the project was good, the kernel was ported to the board and made to execute in place, and drivers written for the Vacuum Fluorescent Display (VFD), multiplexed lamps and graphics frame buffer. The Integrated Drive Electronics (IDE) interface was enabled by editing a driver provided in uClinux. The result is that the Pluto 6 will boot into uClinux at power on, and can then run programs from the root file system mounted on the CompactFlash card, including the Nano-X server running through the frame buffer device.
Although time didn't allow writing drivers for all of the devices suggested, such as the I2C, the three main areas desired were achieved – that is the kernel port , a frame buffer graphics driver with X and the IDE interface. The project can therefore be considered successful.

Table of Contents

1 Introduction....6
1.1 Project origin....6
1.1.1 The current software system....6
1.1.1.1 Advantages....6
1.1.1.2 Disadvantages....6
1.1.2 The problem....7
1.1.3 The solution....7
1.2 Choosing an OS....7
1.3 Requirements....7
1.4 Project life cycle....8
2 The Pluto 6 board and the current system....9
2.1 ColdFire CPU background....11
2.2 Further Pluto 6 information....13
2.3 Current software system....14
3 uClinux....15
3.1 Overview....15
3.2 History....15
3.3 Differences to the normal kernel....19
3.3.1 The role of the Memory management Unit....19
3.3.1.1 Virtual memory....20
3.3.1.2 Protection....20
3.3.2 Running an OS without a MMU....20
3.3.3 Other differences....24
3.4 Summary....25
4 Porting uClinux to the Pluto 6....27
4.1 Linux kernel boot sequence....27
4.1.1 After assembly....27
4.2 Design....28
4.2.1 Kernel version....28
4.2.2 Kernel location options....28
4.3 Implementation....29
4.3.1 Setting up a development environment....29
4.3.1.1 Testing the development environment....31
4.3.2 Creating the initial port....32
4.3.2.1 Entry assembly....32
4.3.2.2 Linker description file....33
4.3.3 Running the kernel....35
4.3.4 Getting more memory....48
4.3.5 Execute in place....50
4.4 Testing....51
5 The Frame buffer driver....53
5.1 Introduction....53
5.2 History....53
5.3 Graphics controller capabilities....54
5.4 About the Linux frame buffer subsystem....54
5.4.1 Origin....54
5.4.2 In user space....54
5.4.3 In kernel space....59
5.5 The Fujitsu Cremson graphics controller....66
5.6 Implementing a Linux frame buffer driver on the Cremson....68
5.6.1 Design....68

5.6.2 Implementation......72
5.6.2.1 16 bit mode......72
5.6.2.2 8 bit mode......76
5.6.3 Running Nano X......81
5.6.3.1 16 bit mode......82
5.6.3.2 8 bit mode......83
6 Conclusion......85
6.1 Performance of the system......85
6.2 Further work......85
6.3 What has been learned......85
7 Bibliography......87
7.1 Books......87
7.2 Articles......87
7.3 Papers......88
7.4 Manuals......88
7.5 Websites......88
7.6 Mailing lists......89
8 Glossary......90
9 Appendix A - Further ColdFire information......92
9.1 Differences to the 68000......93
9.1.1 Simplified addressing modes:......93
9.1.2 Reduced instruction set......93
9.1.3 Restricted instructions......94
9.1.4 Supervisor mode differences......94
10 Appendix B - Further Pluto 6 information......95
10.1 Accompanying Hardware on the Pluto 6......95
10.1.1 FPGA......95
10.1.2 Microchip PIC......95
10.1.3 CPLD......95
10.2 Hardware Boot sequence......96
10.3 ColdFire configuration on the Pluto 6......96
10.3.1 Pin assignments......96
10.3.2 Interrupts on the Pluto 6 board......97
10.3.3 Chip selects and DRAM......98
11 Appendix C – The Heber software architecture......99
11.1 Software architecture......99
11.2 The boot sequence......101
12 Appendix D - uClinux and Linux boot sequence comparison......103
12.1 Power on......103
12.2 X86 PC Linux kernel boot sequence......103
12.3 ColdFire uClinux kernel boot sequence......104
13 Appendix E - Variations in Graphics controller capabilities......106
14 Appendix F - Features of the Cremson MB86290A......108
14.1 Resolutions......108
14.2 Layer/frame control......109
14.3 2d primitives and anti-aliasing......109
14.4 3d polygons and perspective texture mapping......110
14.5 Display lists......110
15 Appendix G - The multiplexed lamps and VFD drivers......112
15.1 The VFD driver......112
15.1.1 Design......112
15.1.2 Implementation......112
15.2 The multiplexed lamps driver......112
15.2.1 Design......113
15.2.2 Implementation......113
16 Appendix H - The CompactFlash IDE driver......115

16.1 Design....................115
16.2 Implementation....................115
17 Appendix I - The working kernel output....................117
18 Appendix J - the BDM-CHK test output....................119
19 Appendix K - Source code listings....................124
19.1 crt0_ram.S....................124
19.2 ram.ld....................127
19.3 crt0_rom.S....................128
19.4 rom.ld....................135
19.5 gdbinit....................136
19.6 vfd.c....................140
19.7 mpxlamps.c....................145
19.8 pluto6_fpga.h....................152
19.9 uclinux.c....................152
19.10 cremsonfb.c....................163
19.11 cremsonfb.h....................177

1 Introduction

Heber Ltd provide control systems for the gambling machine (also known as fruit machine) industry. The Pluto 6 board is sold to fruit machine manufacturers who write the games to run on it. The board is then placed into a fruit machine cabinet and controls all of the peripherals present. The Hardware in a game cabinet can include up to two video screens, lamps, button switches, stepper motors controlling the reels in the cabinet, coin and note acceptors, VFDs and audio speakers.

1.1 Project origin

1.1.1 The current software system

The board as supplied by Heber uses pre-compiled custom hardware libraries written from scratch in-house, which are then linked with the game code to create a single binary. The current libraries can drive the hardware provided such as the serial ports, and even offer FAT32 support for the CompactFlash card.

1.1.1.1 Advantages

The no OS approach taken has been successful, games have been written by fruit machine manufacturers and sold well. The libraries provide a high level of control over the system, due to the code being developed in house and hence well known.

1.1.1.2 Disadvantages

There are however certain problems with the current software architecture.

- The project makefiles use relative paths to include the library files. Every time a new version of a library is released, the version number in the paths of the makefile must be altered. The hardware library path is also used in the linker script to refer to the vector table location and this must also be altered. It soon becomes easy for a game developer to get out of sync with the four separate libraries in both of these files, causing compilation or linking errors. It is up to them to remember to maintain their project files.
- Since the libraries are all compiled separately before the project is compiled and linked with them, development on the libraries can become tedious, e.g. if something is changed in the hardware library, it must be recompiled, since the other libraries have dependencies on it they must be recompiled, before a project can be compiled to actually test the changes.
- The Pluto range is responsible for the majority of Hebers technical support effort. Due to the code being closed source and completely proprietary, maintaining it and removing bugs, while other engineers are continually adding features is a big undertaking.

1.1.2 The problem

So far the hardware in the Pluto range has not presented a significant problem as far as writing drivers from scratch is concerned. However Hebers future boards will be more powerful, possibly featuring sophisticated peripheral interfaces including USB and Ethernet. The software architecture in place may restrict future development on these new products. It is not feasible to write a TCP/IP stack from scratch for Ethernet, or a USB stack along with drivers for every plausible USB device which could be used in a fruit machine.

1.1.3 The solution

The increasing complexity of hardware interfaces and peripheral buses may be handled by providing support for an Operating System (OS) on future boards. Future Pluto platforms may therefore require moving to an OS as the software base. The bulk of the driver development will have been done, and a support community can aid in solving bugs.

It was therefore decided to port an OS to the current board, the Pluto 6.

1.2 Choosing an OS

The OS chosen was uClinux – a fork of Linux with alterations to allow it to run on low cost micro controllers with no memory management unit. UClinux was used for the following reasons:

- The CPU on the board is a Motorola ColdFire, which has no MMU. UClinux is designed for exactly this level of micro controller, hence the ColdFire range is one of the architectures most strongly supported the operating system.
- UClinux is similar enough to Linux that the knowledge gained from writing drivers for the project would be useful for many other systems running a Linux port. This was the main deterrent from using one of the numerous real time executives available for the ColdFire, such as FreeRTOS or VxWorks.
- UClinux is open source, so it can be fully debugged. Access to the source code is also important in order to make sufficient alterations kernel for it to run on the board with the right memory map.
- UClinux is free.
- UClinux has a strong community support base, due to its wide use across multiple architectures.

1.3 Requirements

The Pluto 6 was not expected to perform well running an OS, but would serve as a case study for

Heber, as well as an interesting project. The specification for the project was therefore left open, however suggested drivers to write for the board included:

- An IDE driver to access a file system on the Compact Flash card.
- Communication with the PIC micro controller via the I2C bus, allowing access to a real time clock and security logs.
- Communication with the EEPROM via I2C, possibly accessible as a character based device, e.g. /dev/e2rom.
- Reading/writing of the general purpose I/O and lamps, possibly accessible as individual files in /proc.
- Frame buffer driver for the Cremson Graphical Processing Unit (GPU), possibly leading to a driver for the TinyX X server.
- Sound driver, possibly integrated with Advanced Linux Sound Architecture.

1.4 Project life cycle

The project life cycle for this project was different to that of regular software development. Because of the amount of fixed variables in the system, i.e. the kernel software architecture and the hardware of the board, the project was split into separate stages. The first stage was to port the kernel to run from RAM, using rapid prototyping methods. Each stage following that was to develop a device driver, the methods and design techniques used in these stages varied from bottom-up to top-down depending on the level of provision in the kernel for that type of device (e.g. the VFD driver was flexible, while the frame buffer and IDE has a very structured kernel subsystem). The only dependency the porting stage had on the later driver stages was for the IDE driver. In order to extend the Kernel to execute in place a disk based file system was required.

2 The Pluto 6 board and the current system

The Pluto 6 is an embedded control board intended for use in coin operated gambling machines. The board has numerous features to target this area:

- The main CPU of the board - a Motorola ColdFire, executes the game. It also provides stereo audio using DMA.
- 2Mb of DRAM is used to run the game, also present is 256Kb of battery backed SRAM for frequently accessed or non volatile data.
- IO support - a FPGA controls up to 512 multiplexed lamps, each at 8 brightness levels. It also provides registers to control 64 open drain outputs, and read 32 input switches. The outputs can be used to drive stepper motors used in the reels of fruit machine cabinets.
- Security - the FPGA allows the ColdFire code to be locked to a certain board, preventing piracy.
- A secondary PIC micro controller runs from battery at power down to provide a real time clock, as well as monitoring of security door switches, readings of which are stored in a log. The PIC also reads inputs allowing the percentage payout of a game to be set. Code running on the ColdFire can get data from the PIC over I2C.
- 6 serial ports - 2 in the ColdFire, 2 in the FPGA and 2 in a DUART IC. These can all be routed independently by the FPGA to 1 of 4 RS232 ports, 1 RS485, 1 TTL and 2 ccTalk (commonly used by coin acceptors)
- An IDE connection can interface to a hard disc/CompactFlash card (running in true IDE mode) or a CD drive. The Heber boot loader loads the game from disc in S19 format.
- 2 expansion slots allow the board to be fitted with one or two Calypso 32 graphics cards. These are fitted with 32Mb of SDRAM and a Fujitsu Cremson video controller. Standard VGA connectors are provided on the Pluto 6 board. Graphics for video based games can be loaded from an IDE device.

The Pluto 6 is shown here:

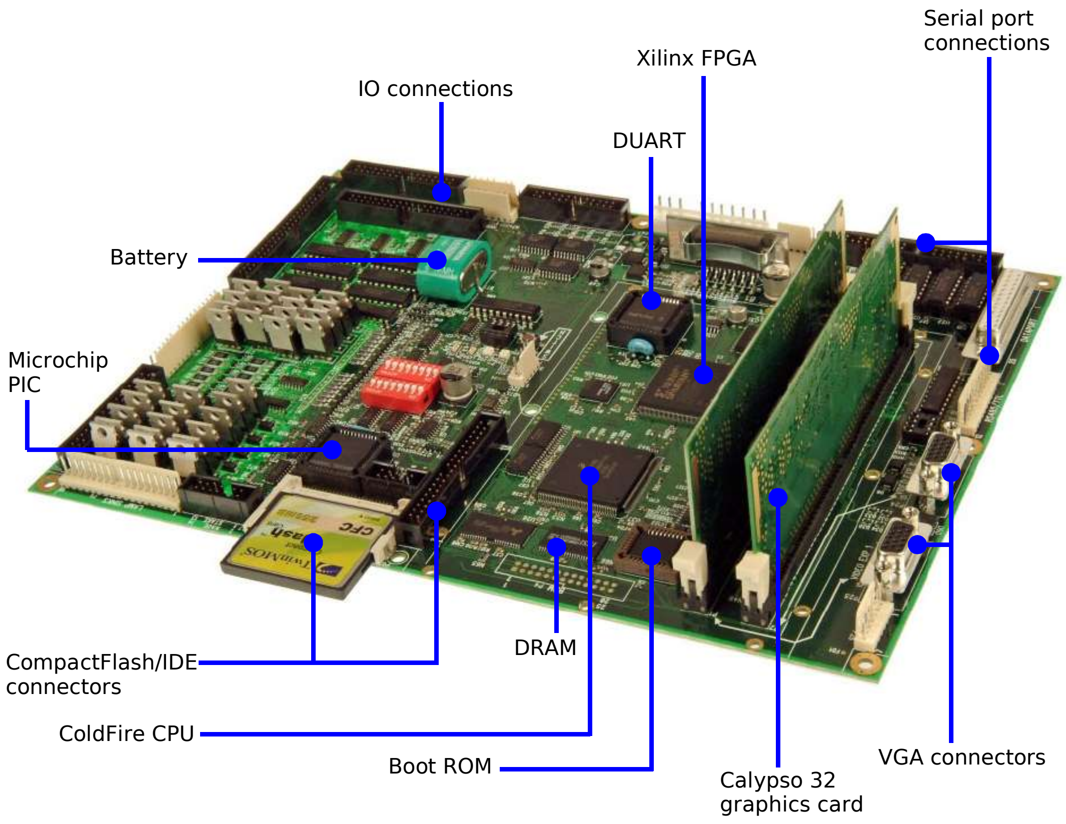


Illustration 1 The Pluto 6 gambling machine control board

The above image shows two Calypso 32 graphics card inserted. The front side of the Calypso 32 graphics card is shown here:

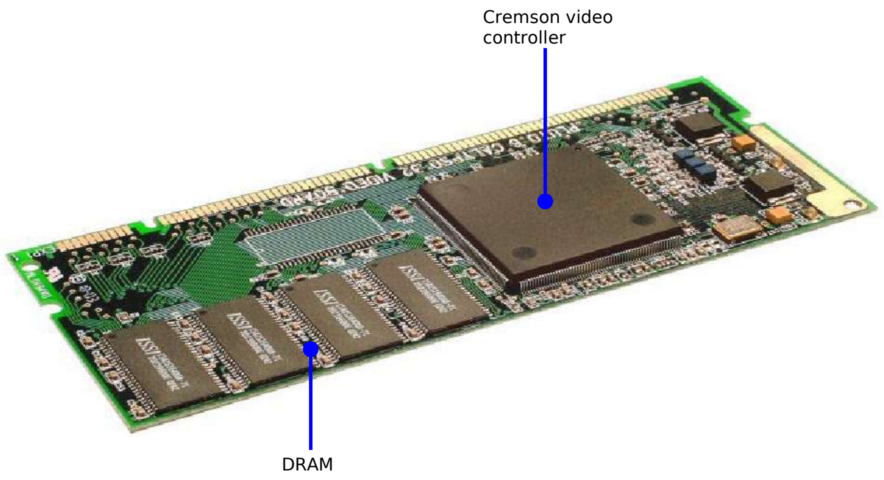


Illustration 2 The Calypso 32 graphics card

2.1 ColdFire CPU background

The CPU used on the Heber Pluto 6 is the Motorola ColdFire 5206e. At the time of writing it is not the most current ColdFire version, newer ones have other peripherals, and the more advanced V4 core. Other ColdFire variants and their features at the time of writing are shown in this table:

Device	***Dhrys. 2.1 MIPS @ Max. MHz***	Cache SRAM Embedded Flash	Serial	Timers CS GPIO	DMA	DRAM Controller	Operating Frequency (MHz)	Applications
5206e	***50***	4 KB I 8 KB -	2 UARTs, 1 I2C	2 8 8	2-ch.	FPM, EDO	40, 54	Vacuum System Controllers, Printer and LAN Interfaces
5207/08	***159***	8K I/D 16 KB -	3 UARTs, 1 opt FEC 1 I2C, 1 QSPI	8 8 Up to 50	16-ch.	DDR/SDR SDRAM	166	Imaging Systems, POS Printers, Networking, Medical Equipment
5211/12/13	***76***	- Up to 32 KB Up to 256 KB	3 UARTs, 1 I2C, 1 QSPI 1 opt CAN	Up to 16 0 Up to 55	4-ch.	-	66, 80	Low Power Entry-Level

Device	***Dhrys. 2.1 MIPS @ Max. MHz***	Cache SRAM Embedded Flash	Serial	Timers CS GPIO	DMA	DRAM Controller	Operating Frequency (MHz)	Applications
5214/16	***63***	2 KB I 64 KB Up to 512 KB	3 UARTs, 1 I2C, 1 CAN	8 + 4 DMA 7 Up to 150	4-ch.	SDR SDRAM	66	POS, Vending, Industrial Control, Security
523x	***150***	8 KB I/D 64 KB -	3 UARTs, 2 CANs, 1 FEC, QSPI, I2C Opt Encryption	Up to 32-ch. eTPU, 4 DMA 8 Up to 113	4-ch.	SDR SDRAM	80, 100, 150	Motor Control, Industrial Control
5249(L)	***125***	8 KB I 96 KB -	2 UARTs, 2 I 2 C , 4 I2 S , 1 QSPI	2 4 Up to 47	4-ch.	SDR SDRAM	140	Digital Audio, Industrial Control, Imaging
5270/71	***96***	8 KB Config. 64 KB -	3 UARTs, 1 I2C, 1 FEC, 1 QSPI	8 8 Up to 61	4-ch.	SDR SDRAM	100	POS, Security, Networking, Gaming, Medical
5272	***63***	1 KB I 4 KB -	2 UARTs, 1 USB, 1 FEC, 1 QSPI	4 8 Up to 32	2-ch.	SDR SDRAM	66	Imaging Systems, Security, Networking, Telecommunications
5274(L)/ 5275(L)	***159***	16 KB Config. 64 KB -	3 UARTs, 1 I2C, Up to 2 FECs, 1 USB, 1 QSPI	8 8 Up to 61	4-ch.	DDR SDRAM	133, 166	POS, Security, Networking, Gaming, Medical
528x	***76***	2 KB I 64 KB Up to 512 KB	3 UARTs, 1 I 2C , 1 FEC, 1 CAN	8 + 4 DMA 7 Up to 150	4-ch.	SDR SDRAM	66, 80	Networking, Industrial Control, Security
5307	***75***	8 KB U 4 KB -	2 UARTs, 1 I2 C	2 8 16	4-ch.	SDR SDRAM, FPM, EDO	66, 90	Printer Servers, Barcode Printers, DVB Boxes

Device	***Dhrys. 2.1 MIPS @ Max. MHz***	Cache SRAM Embedded Flash	Serial	Timers CS GPIO	DMA	DRAM Controller	Operating Frequency (MHz)	Applications
532x	***211***	16 KB 32 KB -	3 UARTs, 1 I2C, 1 FECs, QSPI, USB 2.0, opt. CAN, SSI	Up to 94	16-ch.	DDR/SDR SDRAM	240	POS, Security/Access, Control, Health Care, Building and Factory Automation
537x	***211***	16 KB 32 KB -	3 UARTs, 1 I2C, QSPI, SSI	Up to 62	16-ch.	DDR/SDR SDRAM	240	VoIP, Security/Access, Control, Health Care, Building and Factory Automation
5407	***316***	16 KB I, 8 KB D 4 KB -	1 UART, 1 USART, 1 I 2C	2 8 16	4-ch.	SDR SDRAM, FPM, EDO	162, 220	Mediaweb Boxes, Digital Video Recorders, Telecom Cards
547x	***410***	32 KB I, 32 KB D 32 KB -	4 PSCs, Up to 2 FECs, 1 I2C, 1 PCI, 1 DSPI, opt. USB 2.0	6 6 Up to 99	16-ch.	DDR/SDR SDRAM	200, 266	POS, Network-Attached Storage, Security/Access Control
548x	***308***	32 KB I, 32 KB D 32 KB -	4 PSCs, Up to 2 FECs, 1 I2C, 1 PCI, 1 DSPI, opt. USB 2.0, 2 CANs	6 6 Up to 99	16-ch.	DDR/SDR SDRAM	166, 200	Building and Factory Automation, Process Control Equipment

Table 1 Feature comparison of the ColdFire range

See appendix A for further information on the ColdFire and its differences to the 68000.

2.2 Further Pluto 6 information

See appendix B for details of the other hardware present on the Pluto 6 board, and how the ColdFire is configured for it.

2.3 Current software system

See appendix C for details of the current Heber software architecture.

3 uClinux

3.1 Overview

The uClinux kernel was created to support mid range micro controller units which operate at a sufficient speed can address enough memory to run an OS, but do not have a memory management unit (MMU). UClinux is distributed as patches to the main Linux kernel sources, versions are available for releases of the 2.0, 2.4 and 2.6 series kernels. Regular distribution releases are also available, comprising the following:

- the latest pre-patched versions of the 2.0, 2.4 and 2.6 kernel sources
- a much smaller standard C library called uClibc which has almost all the functionality of Glibc,
- a large list of applications modified to use the much smaller library and less capable kernel, including BusyBox, Tinylogin and MicroWindows/Nano-X.

The minimal changes required to port applications to uClinux are explained later in the chapter.
The kernel source patches add new architecture trees under the /arch directory, e.g. arm-nommu, blackfin, cris, m68k-nommu, niosnommu. As well as the extra architectures, several drivers for peripherals on more popular boards featuring the MMU-less architectures are added in the /drivers directory. The most notable change in the kernel is in the area of memory management, where the directory /mmnommu is added as a counterpart to the /mm directory in the top level of the source tree.

3.2 History

The history of uClinux is shown in the following table:

Date	***Event***
January 1998	The first system without a MMU to run a Linux kernel port was the Motorola Dragonball MC68328 using a modified 2.0.33 kernel, implemented in a SCADA controller. Kenneth Albanowski and Dr Jeff Dione undertook the work at a company called Rt-Control.
February 1998	The work on uClinux is released publicly under the GPL. The first target platform to be shown publicly running the port was a Palm Pilot PDA with a custom boot loader written especially for the project.

Date	***Event***
September 1998	Refinement of the port continued, and uClinux 2.0.38 was released which became the most widely popular version for use in embedded devices. The popularity was increased by Rt-Controls focus on using the kernel to implement actual working embedded devices, rather than other commercial embedded Linux distributions who focused on software only, e.g. porting the kernel and writing toolkits, then handing further development over to device manufacturers.
December 1998	The hardware focus at Rt-Control was re-enforced by the decision to design and manufacture a single board computer (SBC) known as the uCsimm. This board is an inch high, with a standard 30 pin SIMM form factor. It featured a Motorola DragonBall 68EZ328 similar to that used on the original port, and was equipped with 2Mb ROM, 8Mb RAM, Ethernet controller, serial port and I2C. More importantly the uCsimm was specifically designed to run the uClinux OS, and it became popular for numerous applications. The porting of RTLinux kernel patches to uClinux for this board allowed it to be used as a real time device, for example a robotic arm axis controller, which could receive instructions via Ethernet. While Rt-Control were designing their tiny SBC, a company specialising in embedded VPN and Internet appliances called Moreton Bay Ventures were designing the hardware to go into their NetTel line of network routers. For this application they chose Motorola ColdFire CPUs which were to run uClinux, a major design win for the platform.
February 1999	As a result of the design choices at Moreton Bay new ports were added for the Motorola ColdFire 5206 and 5307 architectures. Moreton Bay and Rt-Control worked closely together during this period creating saleable and useful embedded Linux devices. As interest in the ports grew from other device manufacturers and enthusiasts, Rt-Control began distributing CDs containing the patched kernel, a copy of the uClibc library, patched applications and cross compiling tool chains.

Date	***Event***
February 2000	Rt-Control was acquired by a larger embedded Linux company called Lineo. Lineo's main product was an embedded Linux software development kit known as Embeddix, which closely followed the trends of larger Linux systems. During the takeover process the issue of whether to port uClinux to the 2.2 kernel had arisen. Pressure was applied from Lineo, who faced expectations from outside market forces to be providing the latest kernel versions in their products. There was little advantage in using the 2.2 kernel in MMU-less devices, whilst requiring large changes. It was found that merging the 2.0.3x uClinux code and Lineos Embeddix Linux 2.2 SDK was overly complex because the SDK required proprietary start up code in the kernels it built. Implementing this code in the uClinux ports would add little value to the functionality of the working systems, and involve significant changes to the code. A further unresolved issue was how to include the MMU-less support in the Embeddix SDK. The decision was made to leave uClinux out of Embeddix, as a result development slowed, and 2.0.3x uClinux kernels remained in use for some time. However Lineo did recognise the potential of uClinux, and resisted several outsider attempts to take over the project. Soon after the acquisition the original Rt-Control members were allowed to continue uClinux development. The lack of progression to new kernel versions was fortunate for uClinux. The 2.2 kernel code was being developed for larger desktop systems with enhanced feature sets, resulting in kernel sizes that were unreasonably large for uClinux based devices. Kernels using 2.0 could be as small as 200Kb (with networking support), whilst 2.2 kernels with the same functionality were typically twice as big. Drivers added to the 2.2 kernel could be ported to the 2.0 kernel if required, and required reworking anyway to allow for the lack of fork() command, and the mmap() differences in an MMU-less system.
March 2000	Lineo acquired Moreton Bay Ventures. The acquisition enhanced Lineos expertise base, gave them an office in Australia and allowed Rt-Control and Moreton bay to work more closely as they were now owned by the same parent company.
September 2000	Jeff Dionne and John Drabik announce they have ported uClinux to an FPGA running the Leon SPARC open source core.

Date	***Event***
December 2000	The developments in the 2.4 Linux kernel were deemed valuable enough to warrant its use on MMU-less systems, so due to careful preparation - less than a week after the official 2.4 Linux kernel was released, it was followed by the first uClinux version. The port was done by former members of both Rt-Control and Moreton Bay at Lineo, including David McCullough and Greg Ungerer. Since then most interest around uClinux has been on 2.4, but there is still strong interest and support for 2.0 due to its small footprint.
March 2001	The uCsimm SBC was superseded by the uCdimm, with a faster Dragonball CPU, SPI, serial ports and Ethernet in a DIMM form factor. The uCdimm was sold pre-flashed with its own uClinux port, and marketed by Lineo.
June 2001	Lineo announced a new uClinux development board based around an ARM7 CPU.
December 2001	Around this time Lineo decided to divest itself of several non core businesses. The original Rt-Control group were spun-off. By February 2002 the Rt-Control group had formed a new company called Arcturus Networks and had maintained the rights to uClinux. Also spun out of Lineo at the same time was Moreton Bay, who then became Snapgear. These two new companies continued working closely together much as they had done before, hosting and managing the uClinux project, and using it as the base for saleable network devices for VPNs, gateways and control systems.
January 2002	A port of uClinux 2.0.38 was released for the Hitachi H8300.
February 2002	Bernhard Kuhns releases an RTAI port for uClinux on the Motorola ColdFire, providing a hard real time solution for deeply embedded Linux devices.
March 2002	The uCdimm line grew with the addition of a new board, running on a ColdFire 5272 CPU. The product line is now marketed by Arcturus networks.
April 2002	Arcturus Networks officially took on the hosting of the uClinux project.
July 2002	Greg Ungerer of Snapgear releases the first uClinux distribution for download. This contains patched 2.0 and 2.4 kernels, uC-Libc and uClibc libraries, and patched user programs, including Busybox and Microwindows. Prior to this release, only patches were supplied for download, which then had to be applied to the standard kernel which was obtained separately. Snapgear also announce a new website - uCdot, which they would host to announce advances in uClinux.
May 2003	David McCullough of Snapgear released a uClinux port for the Hitachi H8 architecture.
October 2003	ADI released a port of uClinux based on kernel 2.4.6 for their own Blackfin DSP processor.

Date	***Event***
November 2003	Dr John Williams of the University of Queensland in Brisbane, Australia released a uClinux port for the Microblaze soft core CPU - a 32 bit RISC processor synthesized for Xilinx FPGAs.
December 2003	The official release of the 2.6 kernel was closely followed by the uClinux version. Uptake of 2.6 was slow due to the larger footprint and need to port the start-up code for each board to the new kernel. 2.4 remained the main kernel series in use on embedded MMU-less devices.
April 2004	Arcturus Networks released a new board in the uCdimm series based on the Freescale ColdFire 5282.

Table 2 The history of uClinux

Since 2004 support has continued with releases of standard Linux kernels closely followed by uClinux counterparts. Running a Linux kernel on devices of such low specification mentioned above demonstrates the flexibility of the system. During its history uClinux has been ported to many other MMU-less architectures not mentioned above including the ETRAX, ARM7TDMI and other ARM cores, the NEC v850E and the Intel i960. uClinux has rapidly become a popular choice for embedded Linux systems running on MMU-less CPUs, and also MMU enabled systems, with the MMU turned off for extra speed.
The distribution releases now have built in support for a large range of evaluation boards and target platforms. Non-MMU kernels can be readily run on commercial SBCs, evaluation boards, network routers, soft core CPUs running in FPGAs, and are also popular with hobbyists for porting to a range of unintended targets including the Apple iPod and Nintendo Gameboy Advance.

3.3 Differences to the normal kernel

Most differences to the mainline Linux kernel encountered in uClinux stem from the lack of memory management in the hardware of the target architectures. This has effects both in the kernel and for user processes.

3.3.1 The role of the Memory management Unit

A MMU has two main purposes:

- To present a virtual address space to user space applications.
- To prevent the Kernel and applications from corrupting each other

3.3.1.1 Virtual memory

Each application is linked to run from the same virtual address (usually 0), which is done because it isn't known at link time what memory address the kernel will load the program into at run time. While running in user mode the CPU translates virtual addresses given to it by the running process to physical ones, which point to the actual memory chips in which the code or data is residing. The MMU in the CPU performs this operation transparently during the fetch execute cycle, and the normal kernel relies on this management in order to hand control of the CPU over to a process which thinks it is located in one contiguous block. In reality the process could be scattered in sections known as page frames, each containing a "page" of memory. The page frame size (typically 4Kb) is determined by the paging unit of the MMU.

For each running process on the system the kernel must set up and maintain a hierarchical page index, allowing each virtual address to be translated into the offset within a specific page. For example, a processes page index on an Intel® x86 comprises a page directory containing entries to multiple page tables, which in turn point to multiple pages. During code execution the running process will use logical addresses which are passed through an x86 specific layer known as the segmentation unit and translated into linear addresses. The linear addresses then are passed to the paging unit and split down into fields. The first field indexes the page directory to get the page table, the second field indexes the page table to get the page, and finally the offset field points into the page to the memory location required. The final result is the physical memory address. The page index does not need to be fully populated, hence making it hierarchical will save memory from being swallowed up by page directories and tables for small processes requiring only limited memory.

With the paging unit in place a page can be located anywhere in actual memory in relation to other pages. The kernel can also move pages from memory into a swap area on a hard disk, providing a full blown virtual memory system. When a process tries to access a swapped out page, an exception occurs. The kernel handles the exception by reading the required page into memory, updating the page table, and continuing the process.

3.3.1.2 Protection

The MMU also stops processes from overwriting each other or worse, the kernel. Should a process try to write outside of the virtual address space it is currently allocated an exception would be flagged, usually causing the kernel to interrupt and kill off the rogue process.

3.3.2 Running an OS without a MMU

Without a MMU, a CPU has no virtual addresses, only physical ones. Without virtual addresses, there is no need for a paging unit, and so the hardware of the CPU has no notion of pages. Therefore in order to allow processes to run without the assistance of an MMU, the uClinux kernel must take the following measures:

All user processes must be linked to absolute addresses. Every memory address given in any assembly instruction must be a valid physical address. Without paging in hardware, processes can't be split up, so each must be located in memory entirely contiguously.

A process may have others residing above and below it in memory and therefore cannot be allowed to increase the amount of memory allocated to itself at run time. On MMU systems the malloc implementation typically calls the brk system call or the sbrk library wrapper. These calls allocate more space at the end of a processes data segment, thereby increasing the processes address space. When the address space is virtual, this isn't a problem, however on uClinux this address space is physical, increasing it would result in overwriting the start of the next process in physical memory. In order to cope with this, uClinux doesn't implement brk or sbrk, but instead collects the free memory of the system into a single pool of addresses. Any applications which call sbrk must be patched. Allocations are made to processes from this pool using a special malloc implementation. The disadvantage of this technique is that rogue processes can take all the available memory in the system. The advantage of the global pool is that it is managed by the kernel. This means the allocation is transparent to any code that calls malloc, unlike other embedded solutions which require extra code in each application. Also, only the amount of memory required is allocated as opposed to a pre-allocated heap scheme, and memory is easily returned to the global pool as soon as it is freed.

Traditional executable formats for user processes (e.g. ELF) are reliant on virtual memory and therefore cannot be linked to and run from absolute physical memory addresses. To overcome this problem uClinux adds support for FLAT binary executables with the loader code in uClinux/fs/binfmt_flat.c. A FLAT binary consists of the standard text, data and bss sections, but also contains a relocation table created by the elf2flt tool when the flat format binary is created. The relocation table is required because it isn't known at link time where the data and bss sections will be loaded in memory, and all the addresses used must be actual physical addresses. There are two types of relocation table – Global Offset Table and code RELOCs. The GOT is optional, and if included is placed before the data section. The GOT contains a table of offset addresses pointing into the data section at addresses which need to be altered at run time. When loaded these offsets are added to the actual location of the loaded data section to create physical addresses. The table is then used to step through the loaded data and alter all the addresses within it to point to valid addresses. Code RELOCS use a similar principle to GOT, a table of addresses is created by elf2flt and when loaded these addresses point into values in the data section containing pointers, which are then altered to the correct address values.

A useful side effect of the need to relocate the data and bss sections is that it is easy to implement execute in place (XIP), where the code is left in ROM while the data and BSS are copied to RAM thus freeing up memory. However, the relocation tables cannot be used to alter values in the text section of the binary if XIP is to be used, for the following reasons:

1. the same copy of the code in the text section will be used by each instance of the running program
2. the text section may be in ROM.

In order for user binaries to execute in place there must be support in the elf2flt tool, the kernel FLAT binary loader code, the compiler tool chain which must create position independent code, and the crt0.o generated by the library.
The kernel itself may be XIP enabled, but this isn't dependent on, or specific to uClinux. As long as the text, data, init and bss sections of the kernel image are linked into the appropriate addresses and the kernel entry assembly copies the data section from its Virtual Memory Address (VMA) to its Load Memory Address (LMA), the kernel code can be executed from read only memory (e.g. flash). In order for this to work there must be no variables in the text section of the kernel binary, which rules out certain ports of the kernel, but not the ColdFire port used in this project.

Each process in a multitasking OS requires its own stack. In an MMU system, when a process writes over the top of the stack, an exception is thrown, causing the kernel to step in and allocate more memory, allowing the stack to “grow” as required. As mentioned, a running process on an MMU-less system cannot increase in size dynamically at run time, so a fixed stack size must be set at compile time, and allocated at load time. The FLAT format provides a fixed user process stack size, which is set using the flag passed to the compiler:

```
$ FLTFLAGS = -s <stacksize>
$ export FLTFLAGS
$ make application
```

or changed by calling the flthdr command on the compiled FLAT binary:

```
flthdr -s <stacksize> executable
```

The default stack size is 4Kb, although for larger applications such as the busybox binary, a stack size of 20Kb is more suitable. Whilst writing or porting an application for uClinux, developers must be mindful about stack usage. If the limit is exceeded other processes or even the kernel will be

corrupted.
Swapping running processes to disk is not feasible to implement, aside from no pages, most uClinux systems don't have hard disks.

uClinux provides an alternative kernel memory allocator. The standard Linux kernel uses a power of 2 memory allocator by default. This groups all the free regions by size in powers of 2 (e.g. Groups for 1, 2, 4, 8, 16 pages etc.) Newly freed regions are split into the nearest power of 2, the remainder is split in the same manner until all the free pages are put into groups. This is a fast method of finding memory regions large enough to satisfy requests. However, every allocation must be a power of 2, which may be wasteful, e.g. A request for 129Kb will allocate 256Kb, wasting 127Kb. This is fine for larger systems, but unacceptable on the memory constrained platforms uClinux is run on. The alternative allocator is still based around pages, even though MMU-less hardware has no notion of them. A large portion of the kernel source including many drivers assumes memory is available in pages, aligned on the appropriate page boundary. Removing all use of pages from the kernel would represent a huge porting effort.
The uClinux kernel memory allocator is called page_alloc2 or kmalloc2. It uses power of 2 method on a byte basis for allocations up to one page in size, then allocates blocks of memory rounded up to the page size. Page_alloc2 attempts to avoid memory fragmentation - allocations of two or less pages are made from the start of memory onwards, larger requests are taken from end of memory back. As mentioned uClinux requires large blocks of contiguous memory to load processes, so reducing memory fragmentation is important. RAM constrained systems may find that although there is enough free RAM for a request, there isn't enough in a contiguous block to be able to use it. De-fragmenting memory on the fly isn't possible because the contents of an address can't be moved if an active process is using it, the best that can be achieved is to allocate memory wisely before processes are loaded.

uClinux has no fork() system call. On an MMU system, calling fork() would create a new child instance of the process with the same state. The data and bss sections, the heap and the stack of the process are replicated in new areas of memory on the system. The process can continue to use the same addresses to access and change data because the addresses are virtual and will be decoded to unique memory locations by the MMU. If the unmodified fork() were called by a process on uClinux there would be two possibilities:

1. The child process could share its data with the parent. This will cause problems because the application may be written with the assumption that this is not how fork operates. The two processes could instantly go on to change the data somehow as they run and when they are scheduled in and out of the CPU they could find the data is different to how they left it. This would lead to errors at best and probably corruption of the whole system at worst (especially since

uClinux has no memory protection)

2. The data of the child process is replicated elsewhere, however all the pointers the process holds to the data are to physical memory addresses and will suddenly become invalid. If the process uses malloc to dynamically allocate more memory at runtime there is no way to account for this in the FLAT relocation tables created at build time by elf2flt.

The simplest solution is to just not have the fork() system call at all. Similar functionality is provided by the vfork() call, it is known by programmers that after this call the parent and child will share the same set of process data. The parent will be suspended until the child either runs a new process by calling exec() (At which point a new process and associated data is created, removing the need to share the parents), or exits. When Linux applications are ported to uClinux, any calls to fork() must be altered to vfork(), allowing programmers to check that no data is modified by the child before it calls exec(). This technique allows multi-tasking in uClinux, after a few adjustments.

> Making an application use vfork instead of fork usually falls into the absolutely simple or incredibly difficult category. Generally, if the application does not fork and then exec() almost immediately, it needs to be checked carefully before fork() can be replaced with vfork
> ().
> (McCullough 2004)

As mentioned above there is no brk() system call in the uClinux kernel. In standard C libraries (e.g. Glibc) there is a wrapper for brk() called sbrk() which was originally absent from uClibc, the library that usually accompanies uClinux, until recent later versions which implement sbrk() using the global kernel memory pool rather than expanding the process address space. It is often simpler and more portable however to adjust applications not to use the calls at all.

Certain kernel features are heavily dependent on virtual memory and aren't implemented in uClinux at all, such as tmpfs.

3.3.3 Other differences

Aside from the effects of the lack of memory management hardware, uClinux is typically ran on a variety of unusual and deeply embedded micro controllers. This requires a certain amount of porting work and consideration both in and outside the kernel:

A lot of devices which are used on larger architectures can be put on a SBC (e.g. network interfaces, IDE drives etc.) The drivers for these devices are typically written to operate through certain buses such as ISA and PCI, resulting in the use low interrupt numbers below 16, and address ranges below 0x3ff which are the consequence of legacy hardware on platforms such as the x86. Most micro controllers running uClinux have no built in support for particular buses and access devices through the local bus interface. Access routines in device drivers to be ported from an ISA platform for example must be altered to use 8/16/32 bit memory mapped IO addresses instead of 16 bit port addresses. Interrupt vector numbers must be selected from the range available on the micro controller rather than one below 16 (typically the vector number is set by the design of the board and cannot be changed, as opposed to the dynamic numbers assigned on the PC).

The mmap() system call is used to map a file into memory. Its usage on uClinux can be controlled so that no extra memory allocation is performed if the file is already in memory within a romfs file system. This efficiency is preferred on the low memory platforms on which uClinux is used, otherwise it must find and allocate a single block of memory large enough, then copy the data from the file into it. In order to avoid memory allocation using mmap certain conditions must be met:

- The romfs file system must be used, because it is the only file system which guarantees that files will be stored contiguously.
- Only read only mappings can be shared, so therefore all memory mapped files must be read only, to avoid memory allocation.
- The file must be either in ROM or RAM, or otherwise contained within the CPUs address space. A file cannot be directly memory mapped if it must first be read from a disc.

3.4 Summary

The result of running a Linux kernel on MMU-less hardware aside from the need to adjust applications which use fork(), is a lack of memory protection provided inherently by the MMU for the kernel and processes. The standard kernel uses the MMU as a generic method to protect the kernel from being overwritten by running processes, and protects each process from others. Without a MMU this protection is not provided, allowing programming errors to result in strange erratic behaviour instead of segmentation faults.

The ColdFire on the Pluto 6 provides only very basic memory protection at the device level. Each chip select mask register and DRAM control mask register contains bits in the lower word to allow data or code access for the user or supervisor mode. This could conceivably be used to implement memory protection, by running the kernel from one chip select device or DRAM bank, while allocating memory for the processes and running them from another. However this would not be portable since every board has a different memory configuration, so ensuring the access bits are set correctly for each chip select would make the porting process much more complicate and difficult to maintain. Also many boards only have one bank of RAM and one ROM device, in which case the kernel would have to store its data on the same memory device as the user processes, so the ColdFire couldn't stop code executing in user mode from overwriting the kernel data. The kernel code could be stored in ROM, but since this can't be written anyway, the protection would be obsolete.

It is perhaps simplest therefore to accept that MMU-less micro controllers such as the ColdFire cannot do adequate memory protection and concentrate on creating portable workarounds for the problem.

4 Porting uClinux to the Pluto 6

Before describing the process of porting a uClinux kernel, it is useful to understand the differences at boot between uClinux and the normal kernel.

4.1 Linux kernel boot sequence

When a computer system (anything from a large server to a single board embedded system) is turned on, the method in which the hardware is initialised, the kernel loaded into memory and then executed is totally architecture dependent. Only when the start_kernel() function is reached (often the earliest part of the kernel to be written in C, rather than assembly) do the many diverse ports of the kernel begin to show common functionality.

> The architecture-independent starting point is *start_kernel* in *init/main.c*. This function is invoked from architecture-specific code, to which it never returns. It is in charge of spinning the wheel and can thus be considered the "mother of all functions," the first breath in the computer's life. Before *start_kernel*, there was chaos.
> (Rubini and Corbet, 2001)

An analysis the boot sequence of uClinux compared to the more complex operation required on the x86 PC architecture up until C code is run may be found in appendix D.

4.1.1 After assembly

Once start_kernel() has been reached, the various ports of uClinux and the standard kernel are all running the same code in linux/init/main.c. However, many of the functions called are unique to each system, residing in files under the arch/ branch in the sources. In particular setup_arch(), init_IRQ() and mem_init(). If a boot loader is used, the method of passing command line options to the kernel varies depending on the boot loader, architecture and even the board for non standardised architectures. Usually, an ASCII string is left in a certain location, such as after the BSS section of the kernel. To deal with this the setup_arch() function takes a char pointer and points it at the string. setup_arch() is generally populated with #ifdefs for various machine types containing code which either sets the pointer to the string where it resides, or copies it from a variable created in the boot assembly.

It is during the later stages of the boot in start_kernel() that mistakes in the earlier assembly can cause problems, making them difficult to track down. On uClinux this is particularly the case with the assembly variables containing pointers to the start and end of RAM, which if wrong can cause various errors, as was found when porting uClinux to the Pluto 6, described below.

4.2 Design

4.2.1 Kernel version

The 2.4 kernel in the uClinux distribution was chosen for the port, this was because the 2.4 kernel series had support for the most uClinux systems, and was at the time the most used in the uClinux community. This meant the most help would be available if required, many questions were already answered on the uClinux mailing list, and there were example files for other boards to examine.

4.2.2 Kernel location options

There are two main choices of where to place the kernel at run time in an embedded system, RAM or ROM. Which is used is generally determined by the facilities provided by the board. For the Pluto 6 there are several methods to execute code on the ColdFire MCU, and so get an initial uClinux kernel running on the board:

- The ColdFire on the board has 16Kb of internal SRAM, and the Pluto 6 has a 256Kb external SRAM chip. These can be written to by other code, or programmed over BDM, but are both too small to contain a kernel.
- The first code executed by the ColdFire at reset is contained in a 512kb flash EEPROM. This is still a rather limited size for a uClinux kernel, which has a similar footprint to the standard one. The EEPROM is a socketed PLCC packaged device, so could in theory be removed and replaced with a larger one. There are various tools which can program flash devices in-circuit over the BDM cable, including the open source bdm-load program, so removing the chip and placing it in a programmer wouldn't be required for each new kernel build. Unfortunately in practice the catalogues of numerous IC companies didn't contain a pin compatible PLCC packaged EEPROM chip in any size larger than 512Kb, and this is too small for a kernel and a ROM file system. Although a kernel in ROM can be debugged, hardware breakpoints can't be set(the instruction at the appropriate location is replaced by a HALT), so this solution isn't viable for development purposes.
- The Pluto 6 also has a 1.5Mb bank of DRAM, where code is usually run from when the board is used in gambling machines. This is more than large enough for the games and Heber libraries which are normally run on the board, but only just large enough for a uClinux kernel. Code can be downloaded to this memory over BDM and debugged fully. Alternatively the standard Heber boot loader in the EEPROM can copy an S19 file from a FAT32 partition on the CompactFlash card to the DRAM and execute it.

Using the DRAM provides the most room for a kernel and the ROMFS file system which is attached to

it by the uClinux build system. Using the Heber boot loader wouldn't provide any of the feedback which the debugger would provide because it is very simple and has no user output, and the Compact flash would need to be removed and mounted on the development host every time a new kernel is to be tested.
So the initial approach chosen to run the kernel on the Pluto 6 was the simplest one – download the kernel to the DRAM over BDM, where it can be debugged.

4.3 Implementation

4.3.1 Setting up a development environment

A Linux development host was used because this provides the most support for the build tools and their requirements (e.g. Symbolic links are created for the Busy box target binaries, so they must be built on a file system supporting sym-links). A pre-built cross compiling GCC tool chain was obtained, built to compile for the m68k-elf target. A binary tool chain was downloaded instead of compiling one from scratch for a number of reasons:

- Building a GCC tool chain from scratch is a very complicated procedure requiring certain versions of a number of packages which must be combined to build successfully.
- A pre-built tool chain for the M68K is available which is well tested with the uClinux distribution and is known to compile successfully for all supported variations of the ColdFire micro controller.

At Heber the tool chain used was m68k-coff-gcc, which was the most widely used binary format for Linux kernels before 2.4. However the 2.4 series introduced the init calls mechanism:

> The idea of init calls was added in version 2.3.13 and is not available in older kernels; it is designed to avoid hairy `#ifdef` conditionals all over the initialization code. Every optional kernel feature (device driver or whatever) must be initialized only if configured in the system, so the call to initialization functions used to be surrounded by `#ifdef CONFIG_`*FEATURE* and `#endif`. With init calls, each optional feature declares its own initialization function; the compilation process then places a reference to the function in a special ELF section. At boot time, *do_initcalls* scans the ELF section to invoke all the relevant initialization functions. (Corbet et al, 2001)

The init sections also allowed GCC specifiers to be used on functions and variables throughout the source, marking them as used for initialisation only. Once the kernel is up and running the memory they occupy can be freed. These sections are defined in linux/include/linux/init.h:

```
#ifndef MODULE
```

```
#define __init        __attribute__ ((__section__ (".text.init")))
#define __initdata    __attribute__ ((__section__ (".data.init")))
#else
#define __init
#define __initdata
#endif
```

The result of these changes to the kernel is that all ports including uClinux versions had to adopt new tool chains based around a more flexible format than COFF. The most commonly adopted format was ELF which could handle the new init sections. Another advantage of ELF is that it can create the fixed position kernel binary and position independent code user binaries in a single set of tools. COFF required separate tool chains for the kernel and user binaries.

The uClinux distribution version 20041215 (latest at the time) was downloaded and extracted to /usr/src/uClinux-dist/. This distribution contains kernel versions 2.0.39, 2.4.27 and 2.6.9.

At Heber, source level debugging on the Pluto 6 is carried out using a BDM module created by a micro controller development tools specialist called Lauterbach, together with MS windows software called Trace32. The Lauterbach suite is expensive and not supplied with the Pluto 6 development kit. The Pluto 6 DK provides a simpler BDM module made by P&E Micro Systems, with MS Windows software called ICDCFZ. This software allows examination of all ColdFire registers and addressable memory, but isn't as functional as Lauterbachs, since it can't initialise the ColdFire to a working state in a scripted fashion (the boot ROM must be run and then halted before downloading code). ICDCFZ only supports debugging at assembly level only, a proprietary P&E symbol file format is required to debug C source, to create these symbol files requires extra software costs more than the BDM module and ICDCFZ software together.
Since the rest of the tool chain was to run in Linux, debugging in Windows was not preferred. Conveniently, there is a Linux BDM driver and patches to GDB to support debugging 68K/ColdFire targets over the P&E BDM module. Because GDB is a mature, capable debugger, it provides advanced features for the ColdFire target as a matter of course. These features include full C source level debugging as well as assembly stepping, using the well known symbol format in ELF binaries. The other important feature is support for scripting via gdbinit scripts. Once the Module Base Address Register is set, every other register in the ColdFire can be initialised to the correct value – allowing the DRAM to be accessed, and code downloaded. This can all be done in a single script and run every time GDB is started.

The m68k BDM tools package was downloaded, from this a Linux BDM driver module could be built, which operates through device nodes on the host with major number 34, the minor number depends on the target CPU (68K or ColdFire) and which parallel port the P&E module is connected to. For the ColdFire on LPT1 the device file is:

```
crw-rw-rw-  1 root root 34, 3 2006-04-09 20:15 /dev/bdmcf0
```

The device driver operates together with a library which must be installed on the development host. Also provided in the BDM tools package is a small test program called bdm-chk, when run with the device file as an argument this prints the ColdFire type detected and the values of all the CPU registers. See appendix J for the BDM-CHK output.
In practice the BDM device driver wouldn't connect to the target, however the BDM package also provides a BDM network server which when run as root can access the parallel port using the ioperm () system call for compatibility with other Unix versions. The test program and the supplied GDB patches use the library to access the BDM module, the library first tries the Driver, if that fails it will try to access the server on localhost by default. Installing the server and configuring inetd to start it allowed debugging as a normal user without the device driver.
The lack of success with the Linux driver could have been due to peculiarities of the Linux distribution used, however time constraints meant this couldn't be investigated further when a working alternative was available.
In order to use GDB, version 6.0 was downloaded from gnu.org, the patches applied, and GDB built with BDM support. GDB could then be connected to the target with the command:

```
(gdb) target bdm localhost:/dev/bdmcf0
```

The register values in the initialisation scripts used at Heber with the Lauterbach suite were referred to in order to create a gdbinit script to bring the Pluto 6 board in to a usable state when GDB is run. See appendix K for the gdbinit script listing.

This allowed the boot ROM to be removed from the board so there would be no doubt what state the hardware was in.
Having set up the tool chain, the uClinux kernels and the Heber source code was set up with the LXR cross referencing tool.

4.3.1.1 Testing the development environment

The GDB build and the script were tested by power cycling the board, initialising it using m68k-elf-gdb only, and downloading the Heber COFF format demonstration files. Although the native format for the patched GDB is ELF, it is backward compatible with older COFF binaries. The program counter was set to the beginning of the DRAM, and the code executed, confirming that the GDB script had set up the ColdFire correctly

4.3.2 Creating the initial port

There are two levels at which porting may be required for the m68knommu architecture under uClinux:

- A new platform, this level is only used for the m68knommu architecture branch because there are so many ColdFire versions, differences between each are dealt with at this level, such as presence of a floating point unit, user/supervisor programming model variations, ans on chip peripherals. The Pluto 6 uses the 5206e, which is already ported at this level.
- A new board. This layer deals with the differing memory maps on each board supported, it must ensure that the kernel is linked to the correct physical memory address on the target, and that the entry assembly correctly sets the amount of RAM present. The Pluto 6 is added at this level.

The starting point for adding the Pluto 6 board configuration is to create a new directory under:

```
uClinux-dist/linux-2.4.x/arch/m68knommu/platform/5206e/
```

Commonly the directory is named after the manufacturer of the board, this doesn't matter as long as it matches that used in other configuration files. Into this directory must be added the entry assembly for the kernel, and a linker description file:

```
uClinux-dist/linux-2.4.x/arch/m68knommu/platform/5206e/Heber/crt0_ram.S
uClinux-dist/linux-2.4.x/arch/m68knommu/platform/5206e/Heber/ram.ld
```

4.3.2.1 Entry assembly

The code in crt0_ram.S is GNU assembly for the M68k, its main responsibility is to create and fill in certain variables describing the available memory, which are then referred to during memory initialisation. These variables are:

- _rambase - the start of addressable memory, which is not necessarily the same as the start address of the kernel code.
- _ramvec - the address in which the ColdFire vector table will be written to by the kernel.
- _ramstart - the start address of memory allocatable to user programs by the kernel, usually set to the address at the end of the kernel BSS section.
- _ramend - the end address of memory.
- _current_task - this should be set to the address of the init_task_union structure, an initialised

task_struct used to set up the first task table, and specified to reside in the init section of the ELF.

This diagram illustrates how the variables should be set:

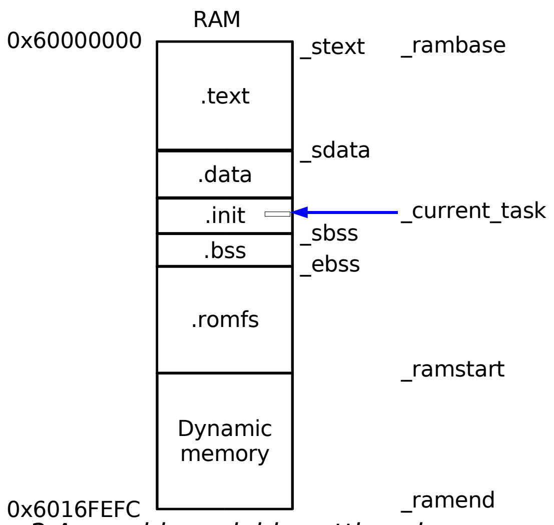


Illustration 3 Assembly variable settings in crt0_ram.S

As well as setting these variables, the ColdFire stack pointer should be set to the end of ram, ready for the CPU to run C code. The file follows a general template across all ColdFire boards, and the values placed in these variables are defined at the top of the file, the initial values used were:

```
#define     MEM_BASE    0x60000000
#define     MEM_SIZE    0x0016fefc
#define     VBR_BASE    MEM_BASE
```

4.3.2.2 Linker description file

Ram.ld is a GNU linker description file in the format described in the ld info pages. On the Pluto 6 the DRAM is configured by the DRAM Controller Address Register 0 (DCAR0) to be accessed at address 0x60000000, so this is the initial address that was placed into ram.ld:

```
MEMORY {
ram    : ORIGIN = 0x60000000, LENGTH = 0x0016fefc
}
```

In order to use these new files, the board must be added to the configuration menu. The following was added to the file uClinux-dist/linux-2.4.x/arch/m68knommu/config.in:

```
if [ "$CONFIG_M5206e" = "y" ]; then
      ...
      bool 'Heber Pluto 6 board support' CONFIG_PLUTO6
fi
```

This file uses the same Config language format as the rest of the kernel to create config menus, as described in linux/Documentation/kbuild/config-language.txt.
The other build related file to add to was uClinux-dist/linux-2.4.x/arch/m68knommu/Boards.mk, which sets the build variables PLATFORM and BOARD to ensure the correct crt0_ram.S and ram.ld pair are used, e.g.

```
uClinux-dist/linux-2.4.x/arch/m68knommu/platform/$(PLATFORM)/$(BOARD)/crt0_ram.S
```

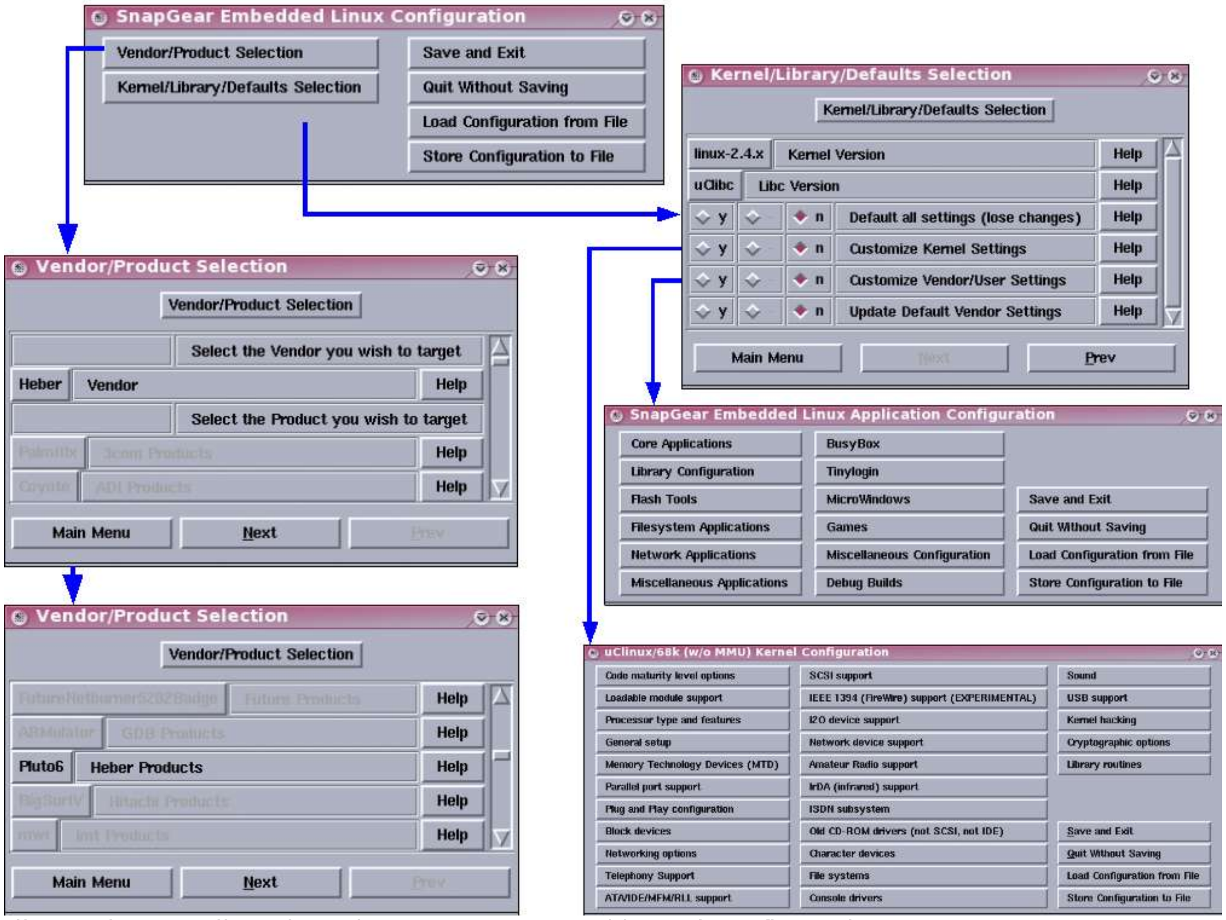
The following was added to Boards.mk:

```
ifdef CONFIG_M5206e
PLATFORM := 5206e
...
ifdef CONFIG_PLUTO6
BOARD := Heber
endif
endif
```

Default configurations were then created for the kernels and ROMFS contents by copying equivalent files from another 5206e board configuration to the following:

```
uClinux-dist/vendors/Heber/Pluto6/Makefile
uClinux-dist/vendors/Heber/Pluto6/config.arch
uClinux-dist/vendors/Heber/Pluto6/config.linux-2.4
uClinux-dist/vendors/Heber/Pluto6/config.uClibc
uClinux-dist/vendors/Heber/Pluto6/config.vendor
```

The makefile controls the type of images created (e.g. ELF and BIN) and the creation of the ROM file system, config.arch specifies the CPU flags to pass to the compiler (e.g. -m5200, to ensure only valid ColdFire instructions are used in the generated code). The others are normal .config files created after running make menuconfig.
The configuration system was then run to configure the kernel for the Pluto 6 board:



linux-2.4.x Kernel Version
uClibc Libc Version
y n Default all settings (lose changes)
y n Customize Kernel Settings
y n Customize Vendor/User Settings
y n Update Default Vendor Settings
Heber Vendor
Pluto6 Heber Products

Illustration 4 uClinux board, user program and kernel configuration

4.3.3 Running the kernel

After successful compilation the following images are created:

```
ed@Eds-PC:/usr/src/uClinux-dist/images$ ls
image.bin  image.elf*  linux.bin*  romfs.img
```

The file image.elf is the one which is downloaded to the board using m68k-elf-gdb. The ROM filesystem is attached to the end of the ELF:

```
ed@Eds-PC:/usr/src/uClinux-dist/images$ m68k-elf-objdump -h image.elf


image.elf:     file format elf32-m68k
```

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00056420  60000000  60000000  00002000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00015be0  60056420  60056420  00058420  2**4
                  CONTENTS, ALLOC, LOAD, DATA
  2 .init         00004c24  6006c000  6006c000  0006e000  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  3 .bss          00011f24  60070c24  60070c24  00072c24  2**2
                  ALLOC
  4 .stab         00175344  00000000  00000000  00072c24  2**2
                  CONTENTS, READONLY, DEBUGGING
  5 .comment      00005b6e  00211904  00211904  001e7f68  2**0
                  CONTENTS, READONLY
  6 .stabstr      0009c5c0  00175344  00175344  001edad6  2**0
                  CONTENTS, READONLY, DEBUGGING
  7 .romfs        00035000  60070c24  60070c24  0028ac24  2**0
                  CONTENTS, ALLOC, LOAD, DATA
```

The kernel was then downloaded to the board and run. There was no output on the serial port as expected, however breaking into the debugger and performing a stack back trace revealed the where code had reached, shown here in a simplified form:

```
start_kernel() : linux/init/main.c
{
      setup_arch() : linux/arch/m68knommu/kernel/setup.c
      {
            reserve_bootmem() : linux/mmnommu/bootmem.c
            {
                  reserve_bootmem_core() : linux/mmnommu/bootmem.c
                  {
                        if (!size) BUG();
```

This was good because the code had proceeded past the call to printk() at the top of start_kernel(), so the C code is running with the initialised data and the stack correctly. However the serial port hasn't been set up at this stage in the boot, so the output from printk is stored in a character array to be sent later called log_buf, which could be printed by GDB. The BUG function is called because value of size is zero. BUG() is an architecture specific macro which performs general crashing duties, on the m68knommu it is defined in linux/include/asm-m68knommu/page.h as:

```
#define BUG() do { \
       BUG_PRINT(); \
       BUG_PANIC(); \
} while (0)

#define BUG_PRINT() printk("kernel BUG at %s:%d!\n", __FILE__, __LINE__)

#define BUG_PANIC() panic("BUG!")
```

The generic panic() function is defined in linux/kernel/panic.c, and outputs the oops message with the CPU register and stack dump seen during kernel panics on all architectures.

The function names where the error was occurring were a clue, and after examining the code which calculates the size variable and stepping through the code to examine the values of other variables, it was decided to check the memory addresses defined at the top of crt0_ram.S. After analysing the assembly code in crt0_ram.S, it was realised that MEM_SIZE should be set to the address of the end of usable memory and not, as the name suggests, the amount of memory. The value was changed to:

```
#define     MEM_SIZE    0x6016fefc
```

The kernel then ran to the following warning before panicking:

```
start_kernel() : linux/init/main.c
{
      setup_arch() : linux/arch/m68knommu/kernel/setup.c
      {
            paging_init(): linux/arch/m68knommu/mm/init.c
            {
                  unsigned long zones_size[3] = {0, 0, 0};

                  zones_size[0] = 0;
                  zones_size[1] = (end_mem – PAGE_OFFSET) >> PAGE_SHIFT;
                  zones_size[2] = 0;

                  free_area_init(zones_size) : linux/mm/page_alloc.c
```

```
                {
                        free_area_init_node() : linux/mm/page_alloc.c
                        {
                                free_area_init_core()
                                {
                                        for(i = 0; i < 3; i++)
                                                total_pages += zones_size[i];
                                        size = (total_pages +1) + (sizeof struct page);

                                        __alloc_bootmemsize) : linux/mm/bootmem.c
                                        {
                                                printk(KERN_ALERT "bootmem alloc of %lu
bytes failed!\n", size);
```

The value of size in the call to printk was 17317696, about 17Mb. A simple outline of how size is calculated is included in the above back trace. The zones_size array contains the size of each memory zone. A zone is a memory range, there are three zones which deal with "hardware constraints that may limit the way page frames can be used" (Bovet and Cesati, 2002). The concept of memory ranges that can only be used for certain purposes is aimed mainly at the x86, and the three zone classifications used are:

ZONE_DMA
    Contains pages of memory below 16MB
ZONE_NORMAL
    Contains pages of memory at and above 16MB and below 896MB
ZONE_HIGHMEM
    Contains pages of memory at and above 896Mb

(Bovet and Cesati, 2002)

These memory ranges are irrelevant to uClinux systems. However the use of zones is typical of uClinux; the alterations to the normal kernel code is minimised by abstracting the rigid structures imposed by other legacy architectures onto the flexible hardware of embedded systems. The zones array is a good example of this, the DMA and HIGHMEM regions are kept for compatibility with other code, but set to a size of 0 and so effectively ignored.
It can be seen that the middle zone, ZONE_NORMAL, is set to contain the end address of memory minus PAGE_OFFSET. Therefore the initial cause of the resulting wrong value in size was discovered to be PAGE_OFFSET. This macro should be set to the value of the start of addressable memory (i.e.

The same as the value in MEM_BASE in crt0_ram.S) in order for the ZONE_NORMAL element of zones_size to contain the right memory size value. The porting process is made simpler in uClinux at this point, due to the use of only physical addresses. Therefore there is no need to find out whether addresses are virtual or physical in different areas of the kernel source.
PAGE_OFFSET is defined in linux/include/asm-m68knommu/page.h:

```
#define PAGE_OFFSET             (PAGE_OFFSET_RAW)
```

PAGE_OFFSET_RAW is defined in linux/include/asm-m68knommu/page_offset.h, the file was altered for the Pluto 6 DRAM address:

```
#ifdef CONFIG_COLDFIRE
#if defined(CONFIG_SMALL)
#define PAGE_OFFSET_RAW         0x30020000
#elif defined(CONFIG_CFV240)
#define PAGE_OFFSET_RAW         0x02000000
#elif defined(CONFIG_PLUTO6)
#define PAGE_OFFSET_RAW         0x60000000
#else
#define PAGE_OFFSET_RAW         0x00000000
#endif
#endif
```

The kernel then ran to:

```
start_kernel() : linux/init/main.c
{
      parse_options(char *line) : linux/init/main.c
      {
            if (!*line)
            <exception occurs>
            ENTRY(trap) : linux/arch/m68knommu/platform/5307/entry.S
                  trap_c(struct frame *fp) : linux/arch/m68knommu/kernel/traps.c
                  {
                        bad_super_trap() : linux/arch/m68knommu/kernel/traps.c
                        {
                              printk("*** Exception %d ***   FORMAT=%X\n",
                                     (fp->ptregs.vector) >> 2,
                                   fp->ptregs.format);
                            die_if_kernel("BAD KERNEL TRAP", &fp->ptregs, 0);
```

Here a pointer to the command line string is passed to the generic function parse_options(), since a boot loader is not being used on the Pluto 6, there isn't a kernel command line, so a default string is created in setup_arch(). When the pointer is dereferenced in parse_options, the ColdFire throws an exception and starts executing the trap assembly routine. This code resides in entry.S in the 5307 directory, even though the ColdFire in use is a 5206e, this is because the exception stack format is common across the 5xxx range. This assembly reads the stack to determine the vector number and format field, populating a frame struct and placing a pointer to that on the stack, before calling back into C code at trap_c. Normally the handler deals with the exception in the context of the current process, however because the CPU was in supervisor mode before the exception, bad_super_trap() is called.

Examining the frame structure revealed the exception was an address error, examining the line pointer before and in parse_options revealed that the stack was corrupted when entering this function. There was a concern that the DRAM wouldn't accommodate a kernel and ROMFS at one end, a stack at the other, and leave room for dynamic allocation in the middle, which appeared to be confirmed. The stack was moved to the 256Kb external SRAM accessed via chip select 1 on the Pluto 6, by altering the entry assembly in crt0_ram.S:

```
#define MEM_STACK 0x10040000

...

      move.l      #MEM_STACK, %a0         /* Set the stack ptr to the */
      move.l      %a0, %d0
      move.l      %d0, %sp                /* end of external SRAM */
```

During the research and analysis required to find the solution to this problem, it was discovered that in order for the kernel to set the integrated peripherals in the ColdFire up properly (e.g. the interrupt controller, serial ports) the correct address contained in the the module base address register (MBAR) must be defined in the kernel. All of the ColdFire system integration module (SIM) registers are accessed as offsets from the MBAR address, which is set in crt0_ram.S:

```
      move.l      #MCF_MBAR+1, %a0        /* Set I/O base addr */
      movec       %a0, %MBAR              /* Note: bit 0 is Validate */
```

The MBAR allows the ColdFire register set to be moved to the preferred address in memory, but in order to set this address an initial register must be written to. This is accomplished using the movec (Move Control Register) instruction, which using its own set of fixed addresses to access certain registers. The %MBAR label above is always 0xC0F on the ColdFire 52xx, allowing the value in MBAR to be set when it has no address. MCF_MBAR is defined in linux/include/asm-m68knommu/coldfire.h:

```
#define MCF_MBAR        0x10000000
```

On the Pluto 6 this address was also being used by chip select 1 to access the external 256Kb bank of SRAM. The chip select address register 0 is set in the gdbinit script after the MBAR, and so overriding it. Therefore whenever the kernel had been attempting to write to the built in peripherals of the ColdFire, it was actually writing to the SRAM.
The MCF_MBAR definition was altered to:

```
#if defined(CONFIG_PLUTO6)
#define     MCF_MBAR    0xF0000000
#else
#define     MCF_MBAR    0x10000000
#endif
```

Prior to this point, the kernel had been crashing early in the C code before the serial ports are configured in console_init(), so the the broken access to the ColdFire SIM had gone unnoticed.

However having moved the stack to SRAM to remove the address exception, and ensuring the ColdFire registers were being written to properly, the serial port began transmitting console messages. However the kernel messages stopped abruptly with no oops register dump, GDB revealed the code had run to:

```
start_kernel() : linux/init/main.c
{
      lock_kernel();
      printk(linux_banner);
      setup_arch(&command_line);
      printk("Kernel command line: %s\n", saved_command_line);
      parse_options(command_line);
      trap_init();
      init_IRQ();
      sched_init();
      softirq_init();
```

```
time_init();

console_init();

kmem_cache_init();
sti();
calibrate_delay();

mem_init() : linux/arch/m68knommu/mm/init.c
{
      nr_free_pages() : linux/mmnommu/page_alloc.c
      {
            for_each_zone(zone) : linux/include/linux/mmzone.h
                  <infinite loop>
```

The macro for_each_zone is defined as a for loop which steps over the three types of zone used in the system as explained above. Therefore for_each_zone should iterate three times, however when stepping through it was found to be looping infinitely. The answer turned out to be the ROMFS.

The board specific makefile adds the ROMFS binary image as a new section to the end of the ELF file which is downloaded to the board. Examination of the output from m68k-elf-objdump above reveals that it is added at the start address of the BSS section. This is correct, because the BSS section doesn't contain anything in the ELF, it just defines the memory range to be used at run time for uninitialised variables. So the ROMFS section can be added at the location of BSS in the ELF file without corrupting anything, which then saves space if a system were to place the ELF in ROM for extraction to RAM at run time.

Once the kernel image is in RAM however, the ROMFS image is then residing where the BSS section should be, as shown:

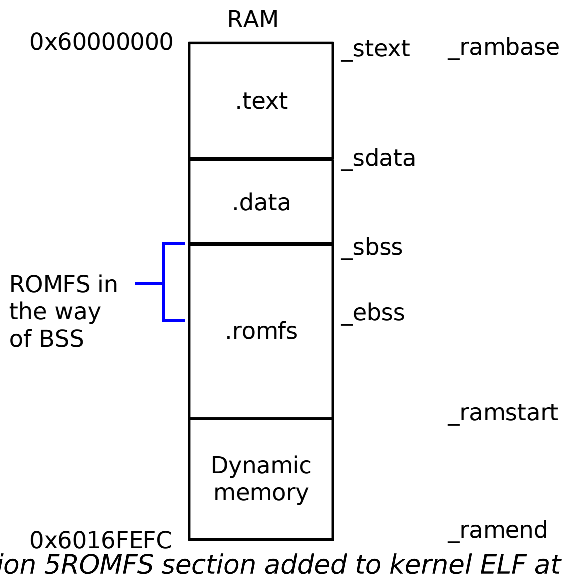


Illustration 5ROMFS section added to kernel ELF at BSS address

The entry assembly in crt0_ram.S handles this by moving the ROMFS image from the end forward to its correct location:

```
      /*
       *    Move ROM filesystem above bss :-)
       */
      lea.l       _sbss, %a0                /* Get start of bss */
      lea.l       _ebss, %a1                /* Get end of bss */
      move.l      %a0, %a2                  /* Copy of bss start */

...

      move.l      8(%a0), %d1               /* Get size of ROMFS */
      addq.l      #8, %d1                   /* Allow for rounding */
      and.l       #0xfffffffc, %d1          /* Whole words */

      add.l       %d1, %a0                  /* calc new address of romfs start */
      add.l       %d1, %a1                  /* calc new address of romfs end */
      move.l      %a1, _ramstart            /* Set start of dynamic ram */

_copy_romfs:
      move.l      -(%a0), %d0               /* Copy dword */
      move.l      %d0, -(%a1)
      cmp.l       %a0, %a2                /* Check if at end */
      bne   _copy_romfs
```

In this code it is known that the ROMFS header resides at the start of the BSS section, so reading this gives the size, then adding it to the start and end pointers in a0 and a1 gives the new area to copy to. As the data is copied, the pointers are decremented, stepping back through the BSS and copying the ROMFS from the back first.

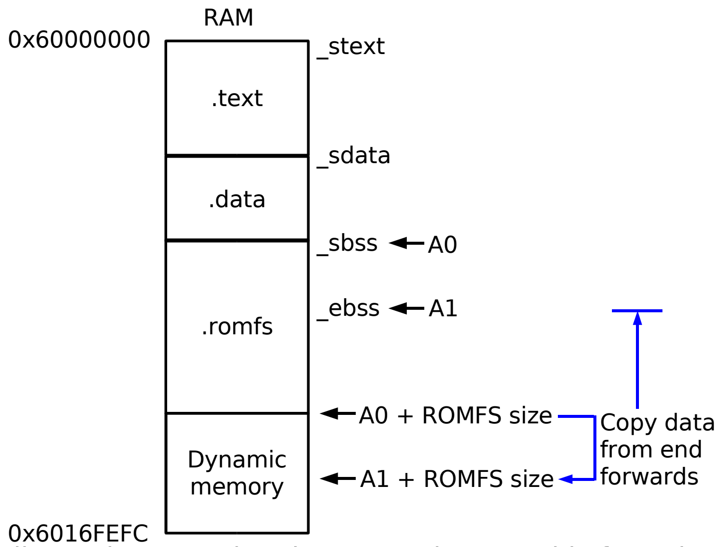


Illustration 6 Moving the ROMFS in assembly from the end forwards

The problem was that the ROMFS was configured to have too many programs in it for the DRAM size, so the calculated value placed in the _ramstart variable (the end of the new ROMFS location) had an address higher than that of _ramend set by the MEMSIZE definition. This was resulting in invalid values in the pgdat_list structure which for_each_zone uses, causing it to loop endlessly. To fix this the user/vendor configuration was set to contain the bare minimum of user programs - a shell and init.

The kernel then ran to:

```
start_kernel() : linux/init/main.c
{
      lock_kernel();
      printk(linux_banner);
      setup_arch(&command_line);
```

```
printk("Kernel command line: %s\n", saved_command_line);
parse_options(command_line);
trap_init();
init_IRQ();
sched_init();
softirq_init();
time_init();

console_init();

kmem_cache_init();
sti();
calibrate_delay();
mem_init()
kmem_cache_sizes_init();
pgtable_cache_init();
fork_init(num_mappedpages);
proc_caches_init();

vfs_caches_init(num_physpages) : linux/fs/dcache.c
{
      mnt_init() : linux/fs/namespace.c
      {
            init_mnt_tree() : linux/fs/namespace.c
            {
                  set_fs_pwd(current->fs) : linux/include/linux/fs_struct.h
                  {
                        struct dentry *old_pwd;
                        ...
                        old_pwd = fs->pwd;
                        <exception occurs>
```

The exception was another address error, it was suspected that the current pointer wasn't set properly to the current process context. After the previous errors, crt0_ram.S was compared with the same file from other boards, and the following line was found missing from the Pluto 6 version:

```
movel   %a0, _current_task
```

The kernel then output the message:

```
Spurious interrupt 1
```

while booting, followed by a kernel panic with oops output. At first a device on the board such as the FPGA was suspected, however inspection of the ColdFire interrupt mask register revealed that only the on-chip timer 0 and serial port 0 peripherals were unmasked. Both of these devices are managed by the kernel, so the interrupt was likely due to an exception. This was supported by the crash after the interrupt.
Stepping through the code narrowed the problem to:

```
start_kernel() : linux/init/main.c
{
      ...

      vfs_caches_init(num_physpages)

      buffer_init(num_physpages);
      page_cache_init(num_physpages);
      signals_init();
      proc_root_init();

      check_bugs();
      {
            <exception occurs>
```

In order to find the cause of the exception, the code was debugged at assembly level using the GDB command:

```
(gdb) display /i $pc
```

to show the assembly instruction currently pointed to by the program counter. The assembly could then be stepped using:

```
(gdb) stepi
```

The first instruction in the check_bugs() function was an orib (OR bytes together). The 5206e manual doesn't list this version of the or in its instruction set, it was one of the 68000 instructions removed from the ColdFire. The tool chain was suspected initially of generating incorrect instructions, however the -m5200 compiler flag was present in all of the build output.

The image.elf file was examined using m68k-elf-objdump, and there were no orib instructions in check_bugs at all. The memory around check_bugs on the board was examined after the crash, and found to be full of the same repeated orib instruction. The answer was then obvious - the kernel was generating a vector table at run time, which was overwriting the start of memory. To confirm this the memory containing the check_bugs code was examined between stepping over each function in start_kernel(), and the oribs appeared after init_IRQ(). The ColdFire vector table has 256 entries, each 4 bytes, so the memory up to 0x60000400 was being overwritten. All of the functions called after init_IRQ() up until check_bugs() were residing after the vector table.

To fix the kernel could be linked to start at 0x60000400, or the vector base address register could be pointed elsewhere. Since the system was already short of RAM, it was decided to place the vector table into the 8Kb of built-in SRAM on the ColdFire. In order to do this the following definition was added to crt0_ram.S:

```
#define     MEM_BUILTIN 0x50000000      /* Built in SRAM location */
```

Followed by code to set the address in _ramvec.

```
    move.l      #VBR_BASE, %a7               /* Note VBR can't be read */
    movec       %a7, %VBR
    move.l      %a7, _ramvec                 /* Set up vector addr */
    move.l      %a7, _rambase                /* Set up base RAM addr */
```

The kernel then created its vector table out of the way in SRAM, and booted all the way through to the init process, but failed to start a shell. At first it was thought the sytem had actually ran out of memory, but the mistake above was then noticed - _rambase now pointed to the SRAM, not the DRAM containing the kernel. The code was replaced with:

```
    move.l      #VBR_BASE, %a7               /* Note VBR can't be read */
    movec       %a7, %VBR
    move.l      %a7, _ramvec                 /* Set up vector addr */

    move.l      #MEM_BASE, %a7               /* set mem base */
    move.l      %a7, _rambase                /* correctly */
```

Then, finally, the kernel booted through to SASH (the standalone shell). Sash has various commands built in to it, such as cd, ls, cat and vi, so files in /proc could be examined, but after a small amount of activity the memory would become too fragmented to run anything. So although the kernel was now booting, it wasn't very useful, especially if it was to be extended to run a graphical system.

4.3.4 Getting more memory

A brief attempt was made to download and run the kernel from the video memory on the Calypso 32 graphics expansion card. The kernel was re-linked to the address 0x30000000, and the value of MEM_BASE and PAGE_OFFSET altered accordingly. Before downloading the kernel, the Calypso card required initialisation, this was achieved by running the board with the Heber boot loader to load the graphical demonstration code from the CompactFlash. The ColdFire was halted with GDB, then the image downloaded, which could be viewed on the monitor connected to the Pluto 6 filling up the video memory:

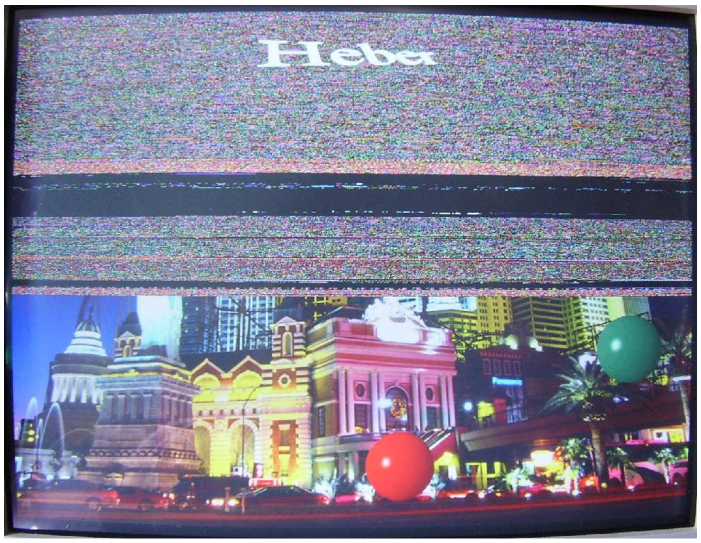


Illustration 7 The Heber Video demo halted with code downloaded; notice the black section in the middle of the code, that is the empty bss section. Also the Heber logo can be seen, this is displayed on a different layer and so isn't overwritten in video memory by the uClinux kernel

The program counter was then set to 0x30000000, however when the first assembly instructions were stepped through, the program counter was found to increment incorrectly for the known sizes of the instructions, causing them to be misinterpreted by the ColdFire. Since the board hadn't been reset (because the graphics card initialisation would have been lost) it was suspected that the CPU cache may be interfering.

Attempts to solve this halted when a better solution was found. During a visit to Heber a DRAM expansion card was obtained. This was designed for a specific customer request and only a small number were made, the customer uses an EEPROM card connected to the 3x32 P3 socket on the board, and they wanted a device which functioned on CS0 as the EEPROM did, but was volatile and could be programmed on target through the BDM. The result was this card:

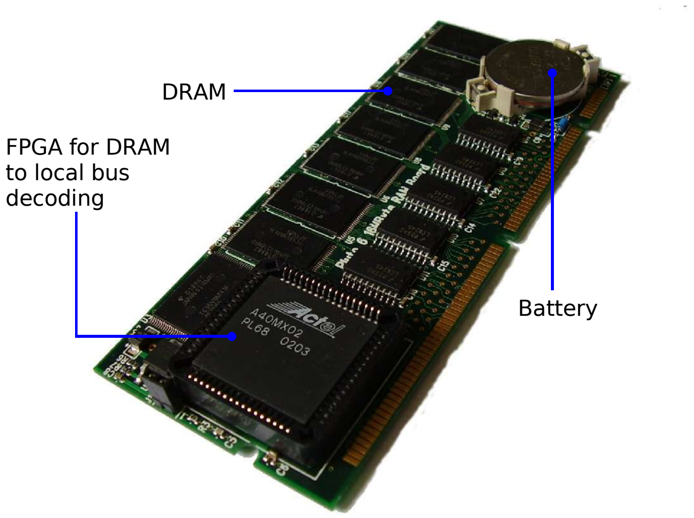


Illustration 8 The 16Mb non-volatile DRAM prototype card acquired from Heber

The card has 16Mb of DRAM on it which is decoded and refreshed by an FPGA, and maintained while power is off by a battery. The card then appears to the ColdFire as a non-volatile static memory device on CS0 in place of the boot ROM (or the EEPROM card). Given this extra memory, it was decided to upgrade to the latest 2.4 kernel version. Linux-2.4.31.tar.gz was downloaded from kernel.org, and extracted to the uClinux-dist directory. The patch uClinux-2.4.31-uc0.diff was applied. Following advice on the uclinux mailing list to keep the ColdFire MBAR set to 0x10000000, the Chip select addresses were altered in the gdbinit script as follows:

- The ColdFire register set was moved to 0x10000000 by altering the MBAR value.
- The CPLD on CS1 which controls access to the external SRAM and IDE registers was moved to 0xF0000000.
- The CS0 control register was set to use a 16bit data port for the new RAM card (the boot ROM used 8bit)

MEM_SIZE in crt0_ram.S was set to 0x01000000, MEM_BASE was now left at 0x0. Rom.ld was then set to link the kernel to 0x0, and the other files in the kernel could be left unaltered.

The new kernel was configured, compiled and booted. Execution halted half way through with no messages. By this stage an instinct for the solutions was growing, and the problem quickly identified

using the set and examine commands in GDB - the RAM card memory couldn't be written to above 0x100000, and higher addresses only read as 0xFF. The chip select address mask register was suspected, and sure enough, it was set in the gdbinit script to 0x000F0000. The upper word is the mask for the upper word of addresses matching this chip select, the lower contains access control bits. To access the full 16MB, the mask register was changed to 0x01FF0000.
The kernel then booted correctly, and had enough memory to run without memory allocation errors.

4.3.5 Execute in place

The RAM card is battery backed - making it feasible to execute a kernel in place. In order to do this, two new counterparts to the linker script and boot assembly files are required - crt0_rom.S and rom.ld.
The rom.ld linker script must split the RAM, and treat some as ROM:

```
MEMORY {
      flash  : ORIGIN = 0x00000000, LENGTH = 0x00200000
      ram    : ORIGIN = 0x00200000, LENGTH = 0x00E00000
}
```

The size of flash was chosen as 2Mb to contain even a large kernel. The flash region is where the kernel text section is executed from, and the data, init and BSS are stored before run time, by linking their LMA addresses to it using the AT directive:

```
      .data : AT (ADDR(.text) + SIZEOF(.text)) {
```

The rest of the RAM is treated as RAM, and the VMA of the data, init and BSS sections are linked here, so at run time C code will look in the RAM for them.
The entry assembly in crt0_rom.S therefore must copy the data section to the correct address in RAM before jumping to C code. It does this using _etext as the address where the data is, _sdata as where to copy it to, and _init_end as the the length to copy.
The other important difference in crt0_rom.S is that it must contain all of the hardware setup that was performed in the gdbinit script up to this point, as well as a vector table at the start of the text section:

```
.text

_vectors:
```

```
.long 0x00000000, _start
.long _fault, _fault, ....
```

At power on the ColdFire will then read the second entry (_start, the entry point to the code), place that value in the program counter, and start running.

After a number of attempts the kernel booted at power on, however it would fail to mount the ROMFS. It was relised this had been completely forgotten in the new boot code. The crt0_rom.S files for other boards which provided them contained no references to moving the ROMFS section. Also the makefile was adding it at the VMA of the BSS section, so it was not being stored in the ROM area when downloaded to the board.

A simpler solution was to use the CompactFlash drive for the root file system. The XIP kernel was left while the IDE driver developed.
Once the IDE was working, The contents of the ROMFS image were mounted and copied to an ext2 partition on a CompactFlash card. The file uClinux-dist/vendors/Heber/Pluto6Makefile was edited so no ROMFS section was added to the ELF. The compiled in kernel command line string was then altered to:

```
      root=/dev/hda1 console=tty0 console=ttyS0,9600n8
```

The kernel then booted and mounted the compact flash partition at boot. The uClinux FLAT binary loading code can detect when the user binaries must be entirely loaded into RAM, so they didn't require modification with the flthdr tool to run from disc.

The result is a board that boots at power on, with a file system that can contain as many programs as the disk size permits. There are no more long downloading times for the file system, but the kernel can be replaced over the BDM at any time, because the RAM card is still writeable.

4.4 Testing

The kernel on the Pluto 6 has been tested not by formal means, but by repeated use running as much of the software ported to uClinux as can be compiled successfully, with various CompactFlash cards.

5 The Frame buffer driver

This chapter describes the process of implementing a Linux frame buffer driver on the Calypso 32 graphics card used on the Pluto 6.

5.1 Introduction

The concept of a frame buffer device is simple - it is an area of memory in a computer which the host CPU can write values to, and which the graphics controller will then convert to electrical signals suitable for outputting to a video monitor. A graphics card will typically hold a video controller chip together with banks of memory known as the video memory, which can often be written to faster than the standard RAM in a system. The video controller on the card then implements the frame buffer system by constantly reading the video memory every time the screen is refreshed. The Linux frame buffer subsystem is the software interface the OS uses to handle setting up the graphics controller hardware for frame buffer operation, and providing accelerated features where possible.

5.2 History

The theory and advantages of a frame buffer device were discussed long before the memory became available to actually build one. In 1969, Joan Miller of Bell Labs had created the first known working frame buffer device, which drew an image with 3 bit colour depth[1].
Richard Shoup developed the SuperPaint system at Xerox PARC in 1972, this took advantage of the rapidly progressing advances in memory technology, and so was the first device which could display and hold on screen a standard video image. It had 640x480 resolution with 8 bit colour. The frame buffer memory was implemented as a 307,200 byte shift register - using 16 separate memory boards each loaded with a 2 Kilobit Shift register. The problem with this was each pixel value could only be loaded when the appropriate memory position rolled around. Shoup also experimented using this system with lookup tables, creating the first palette based frame buffer.

The first commercially available system was created in 1974 by the computer firm Evans & Sutherland based at the University of Utah. The device could output 512x512 at 8 bit grey scale, and was sold for $15,000. Three of the systems were later combined at the New York Institute of Technology to create the first 24 bit colour frame buffer, each 8 bit frame buffer device supplied one of red, green or blue, in an RGB colour output, using a micro controller to synchronise the signals.

As integrated circuit technology improved and memory costs reduced, frame buffer devices found their way into many of the home computers of the late 1970s, although they were often very poor quality with low resolutions and high latency. High end workstations built by companies such as IBM, Sun Microsystems and DEC throughout the 1980s used frame buffers of a higher quality, and since then nearly all computers with graphical capabilities use a frame buffer device to output video signals.

5.3 Graphics controller capabilities

The Linux kernel provides a device abstraction of any frame buffers in the system. The reason it does this rather than simply supplying a pointer to a block of graphics memory is due to the the wide variation of capabilities provided by any one graphics controller, which could then be different to others. Examples of features which may vary are given in appendix E.

The frame buffer subsystem in the Linux kernel is made complicated by the need to handle these variations in graphics hardware capabilities. Many drivers will implement certain frame buffer functions (e.g. panning) using the specific capabilities of the video controller.

5.4 About the Linux frame buffer subsystem

5.4.1 Origin

Initially Linux ran on PCs only, and relied on the built in text console and font sets found in the VGA graphics cards used on the platform. When the OS began to spread to other platforms, a solution was needed to display a console without a default text mode display where there was none, e.g. on the Apple Macintosh.
Since then the frame buffer subsystem has become popular on many platforms with video output for its ability to display Unicode fonts, rather than the 512 characters that VGA fonts provide. It also provides smooth text at much higher resolutions allowing many more characters on screen at once.
There are various applications which use the frame buffer, most predominantly the KDrive Tiny-X server, but also Microwindows/Nano-X, Mplayer, as well as the libraries SDL, GTK+, and DirectFB, which provides a framework for hardware acceleration. There is also a kernel resident windowing system called FBUI, allowing multiple applications to use the frame buffer simultaneously.

5.4.2 In user space

The frame buffer is abstracted and presented to user space applications using the device files /dev/fb0../dev/fb255 which all have major number 29, the minor number is used to identify each frame buffer registered in the kernel. Most computers only have one frame buffer, although systems with multiple graphics cards or multi-head cards will have more.
The device file can be read or written to in order to get or set the pixel values on the screen. E.G. the

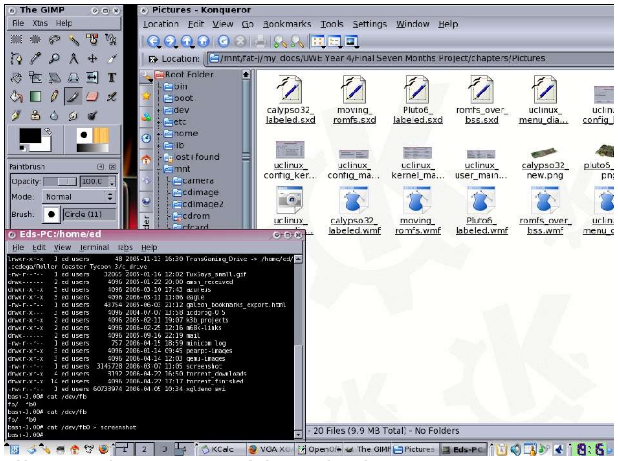
command:

```
~/# cp /dev/fb0 screenshot
```

will copy the contents of the frame buffer as raw pixel data into the file screenshot. This file can't be opened in any image editor because it is raw data, but can be written back to the frame buffer:

```
~/# cat screenshot > /dev/fb0
```

Here is an example - first a 1024x768 32 bpp X desktop is dumped to a file:



```
lrwxr-xr-x   1 ed users       48 2005-11-13 16:30 TransGaming_Drive -> /home/ed/
.cedega/Roller Coaster Tycoon 3/c_drive
-rw-r--r--   1 ed users    32065 2005-01-16 12:02 TuxSays_small.gif
drwx------   2 ed users     4096 2005-01-22 20:00 amsn_received
drwxr-xr-x   3 ed users     4096 2006-03-10 17:43 azureus
drwxr-xr-x   3 ed users     4096 2006-03-11 11:06 eagle
-rw-r--r--   1 ed users    43754 2005-06-03 21:12 galeon_bookmarks_export.html
drwxr-x---   3 ed users     4096 2004-07-07 13:58 icdprog-0.5
drwxr-xr-x   2 ed users     4096 2005-02-11 19:07 k3b_projects
drwxr-xr-x   2 ed users     4096 2006-02-25 12:16 m68k-links
drwx------   2 ed users     4096 2005-09-16 22:19 mail
-rw-r--r--   1 ed users      757 2006-04-15 18:59 minicom.log
drwxr-xr-x   3 ed users     4096 2006-01-14 09:45 pearpc-images
drwxr-xr-x   3 ed users     4096 2006-04-14 12:03 qemu-images
-rw-r--r--   1 ed users  3145728 2006-03-07 11:05 screenshot
drwxr-xr-x   4 ed users     8192 2006-04-22 16:50 torrent_downloads
drwxr-xr-x  14 ed users     4096 2006-04-22 17:17 torrent_finished
-rw-r--r--   1 ed users 60733974 2006-04-05 10:34 xgldemo.avi
bash-3.00# cat /dev/fb
fb/  fb0
bash-3.00# cat /dev/fb
fb/  fb0
bash-3.00# cat /dev/fb0 > screenshot
bash-3.00#
```

Illustration 9 A 32 bit X desktop

The display is then switched to a virtual console running the vesafb driver at the same resolution, but with 16 bpp colour depth, and the frame buffer dump file is copied back to the frame buffer device:

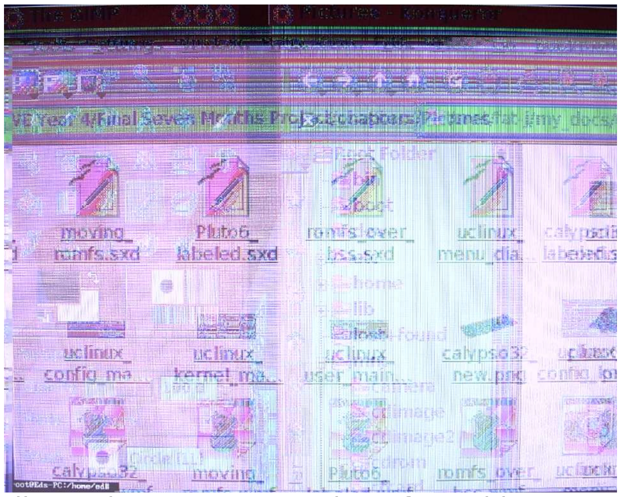


Illustration 10 A screen shot of a 32 bit desktop copied back to /dev/fb0 whilst in a 16 bit virtual console

The display is all warped, because the file created was 32bpp, but during the switch from X to a virtual console, the video mode of the graphics card was changed to 16bpp.

In order to control the video mode variables - such as the resolution and accompanying timing variables, the frame buffer subsystem in the kernel uses a set of structures. Some of these structures are made visible to user space programs which use the frame buffer device directly, so that they know how the pixel values are encoded, and how big the screen is. Examples of such software are the fbset utility, the purpose of which is to change frame buffer video modes, and the XFBdev X server. The structures available in user space are:

```
struct fb_fix_screeninfo {
      char id[16];                  /* identification string */
      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 */
};
```

This structure contains device independent information which does not change unless the video mode of the graphics controller changes. This includes the pointer to the frame buffer video memory, and its length. The structure can be obtained in user space by calling the FBIOGET_FSCREENINFO ioctl command on the frame buffer device file.

```
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 */
};
```

This structure is obtained from the kernel by the FBIOGET_VSCREENINFO ioctl command. It contains information about the current video mode which can vary at any time. This includes the physical and virtual resolutions, and the x and y offset if the virtual resolution is bigger than the physical one. The offset values can be changed by the FBIOPAN_DISPLAY ioctl, thereby panning the display. The structure also contains the variable bits_per_pixel and bit fields for red, green, blue and transparency which tell user applications the size of each pixel value in the video memory and how the colours are mapped into them. The other main group of variables are the timing values, which are discussed in implementation of the Cremson driver below.

As well as get information about the current video mode via the fb_var_screeninfo structure, user space programs can populate their own and pass it in to the kernel via the FBIOPUT_VSCREENINFO ioctl command. The X server and fbset utility mentioned above operate in this fashion to change the video mode of the frame buffer.

As well as reading and writing to the frame buffer device files, some software will simplify the operation by calling the mmap system call on the frame buffer device. This maps the frame buffer to an area of memory so that software can write to it directly from user space. Not all devices use a large block of memory and hence are not suitable for memory mapping, a frame buffer device is one example where implementing the system call is logical.

The third frame buffer structure accessible in kernel space is the fb_cmap structure:

```
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 */
};
```

This represents a colour lookup table - also known as a palette, which is used in frame buffers with indirect colour as mentioned above. The palette table of a frame buffer can be obtained using the FBIOGETCMAP ioctl, and an optimum palette can be set with the FBIOPUTCMAP ioctl. The structure contains a pointer to an array for each colour, each element in each array is 16 bits, regardless of the current video mode of the hardware. When the colour map is requested or set by an application, the values in the colour map must be truncated or enlarged as necessary.

5.4.3 In kernel space

The structures available in user space above are used in the same form within the kernel, as part of a larger framework. The kernel sources provide two distinct example frame buffer driver frameworks, the first is the virtual frame buffer - a build-able example which requires no actual graphics hardware:

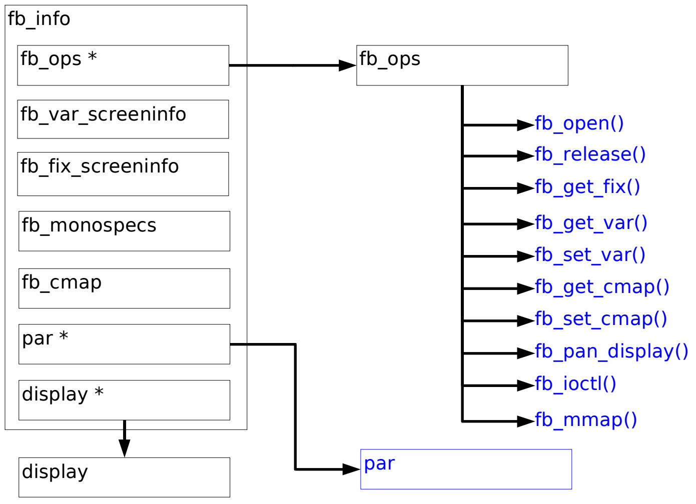


Illustration 11 Diagram of the virtual fame buffer framework. The black elements are generic kernel structures populated by the driver, blue elements are specific to the driver

The structure fb_info represents a single frame buffer device instance. As well as the structures fb_var_screeninfo, fb_fix_info and fb_cmap which are the same as those accessible in user space, it contains numerous other structures and variables:

```
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 */
   u32 mapped_vram;                 /* ioremap()'ed VRAM */
   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;
};
```

fb_monspecs contains maximum and minimum limits of the video timing values for a monitor. This prevents damage being caused to a screen by setting the vsync or hsync signals too high. The only problem with this safety net is that the values in the fb_monspecs struct contained in fb_info must be set by the driver, then the function fbmon_valid_timings in the file fbmon.c must be called to check that value about to be set in the graphics controller are acceptable. Very few of the drivers in the kernel actually do this, and for a number of releases of the 2.4 kernel, the checking code within the fbmon_valid_timings function has been left out of the build.

The fb_ops structure contains pointers to the functions which the driver uses to set data in the various structures contained within fb_info:

```
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);
    /* for kernels with no mmu */
    unsigned long (*get_fb_unmapped_area)(struct file *file,
                   unsigned long addr,
                   unsigned long len,
                   unsigned long pgoff,
                   unsigned long flags);
};
```

E.G. calling the FBIOGET_VSCREENINFO ioctl command from user space will result in the driver function pointed to by fb_get_var being run. Drivers using this framework must implement their own functions here, particularly fb_get_var and fb_set_var, which typically read and set values in the par structure explained below. It is through this set of functions that the graphics controller is set up by the driver and applications discover the pixel mapping and video mode.

The display structure is the interface between the frame buffer driver and the console driver. It could be seen as the only "in kernel" system to use the frame buffer in a similar manner to user applications. In order to function the console driver defines the display structure in fbcon.h to keep track of the frame buffer settings:

```
struct display {
    /* Filled in by the frame buffer device */

    struct fb_var_screeninfo var;   /* variable infos. yoffset and vmode */
                                    /* are updated by fbcon.c */
    struct fb_cmap cmap;            /* colormap */
    char *screen_base;              /* pointer to top of virtual screen */
                                    /* (virtual address) */
    int visual;
    int type;                       /* see FB_TYPE_* */
    int type_aux;                   /* Interleave for interleaved Planes */
    u_short ypanstep;               /* zero if no hardware ypan */
    u_short ywrapstep;              /* zero if no hardware ywrap */
    u_long line_length;             /* length of a line in bytes */
    u_short can_soft_blank;         /* zero if no hardware blanking */
    u_short inverse;                /* != 0 text black on white as default */
    struct display_switch *dispsw;  /* low level operations */
    void *dispsw_data;              /* optional dispsw helper data */

    /* Filled in by the low-level console driver */
    struct vc_data *conp;           /* pointer to console data */
    struct fb_info *fb_info;        /* frame buffer for this console */
    int vrows;                      /* number of virtual rows */
    unsigned short cursor_x;        /* current cursor position */
    unsigned short cursor_y;
    int fgcol;                      /* text colors */
    int bgcol;
    u_long next_line;               /* offset to one line below */
    u_long next_plane;              /* offset to next plane */
    u_char *fontdata;               /* Font associated to this display */
    unsigned short _fontheightlog;
    unsigned short _fontwidthlog;
    unsigned short _fontheight;
    unsigned short _fontwidth;
    int userfont;                   /* != 0 if fontdata kmalloc()ed */
    u_short scrollmode;             /* Scroll Method */
    short yscroll;                  /* Hardware scrolling */
    unsigned char fgshift, bgshift;
    unsigned short charmask;        /* 0xff or 0x1ff */
};
```

The frame buffer driver must set the display_switch pointer to an appropriate set of console operations in order for console text to appear on the screen, for example:

```
      switch(var->bits_per_pixel)
      {
      case 8:
            info->disp->dispsw = &fbcon_cfb8;
            break;
      case 16:
            info->disp->dispsw = &fbcon_cfb16;
            break;
      }
```

fbcon_cfb8 and fbcon_cfb16 are sets of generic console operations for 8 bit and 16 bit colour modes, others exist for most colour depths, to ensure the characters are drawn correctly. The set of console operations are as follows:

```
struct display_switch {
    void (*setup)(struct display *p);
    void (*bmove)(struct display *p, int sy, int sx, int dy, int dx,
                  int height, int width);
    /* for clear, conp may be NULL, which means use a
        blanking (black) color */
    void (*clear)(struct vc_data *conp, struct display *p, int sy, int sx,
                  int height, int width);
    void (*putc)(struct vc_data *conp, struct display *p, int c, int yy,
                 int xx);
    void (*putcs)(struct vc_data *conp, struct display *p,
                  const unsigned short *s, int count, int yy, int xx);
    void (*revc)(struct display *p, int xx, int yy);
    void (*cursor)(struct display *p, int mode, int xx, int yy);
    int  (*set_font)(struct display *p, int width, int height);
    void (*clear_margins)(struct vc_data *conp, struct display *p,
                           int bottom_only);
    unsigned int fontwidthmask;      /* 1 at (1 << (width - 1)) if width
                                        is supported */
};
```

The console driver must also be told how the colours in the pixel values are mapped by filling in display.fb_cmap appropriately. The above assignment must be done by the frame buffer driver in the fb_set_var function, so that whenever a console switch occurs, the frame buffer driver will set the console operations properly. The console functions are then called by the generic console driver in drivers/char/console.c.

The final significant item in fb_info is the par structure – referring to a particular graphics controller. It is up to each frame buffer driver to define par and put whatever variables are required into it - it should define a unique video mode for the graphics hardware. Therefore par is completely different in each driver. Par can then be used to set information in the fb_var_info, or copy values into hardware.

The second framework for a frame buffer device is the file skeletonfb.c, which is a template based on the following layout:

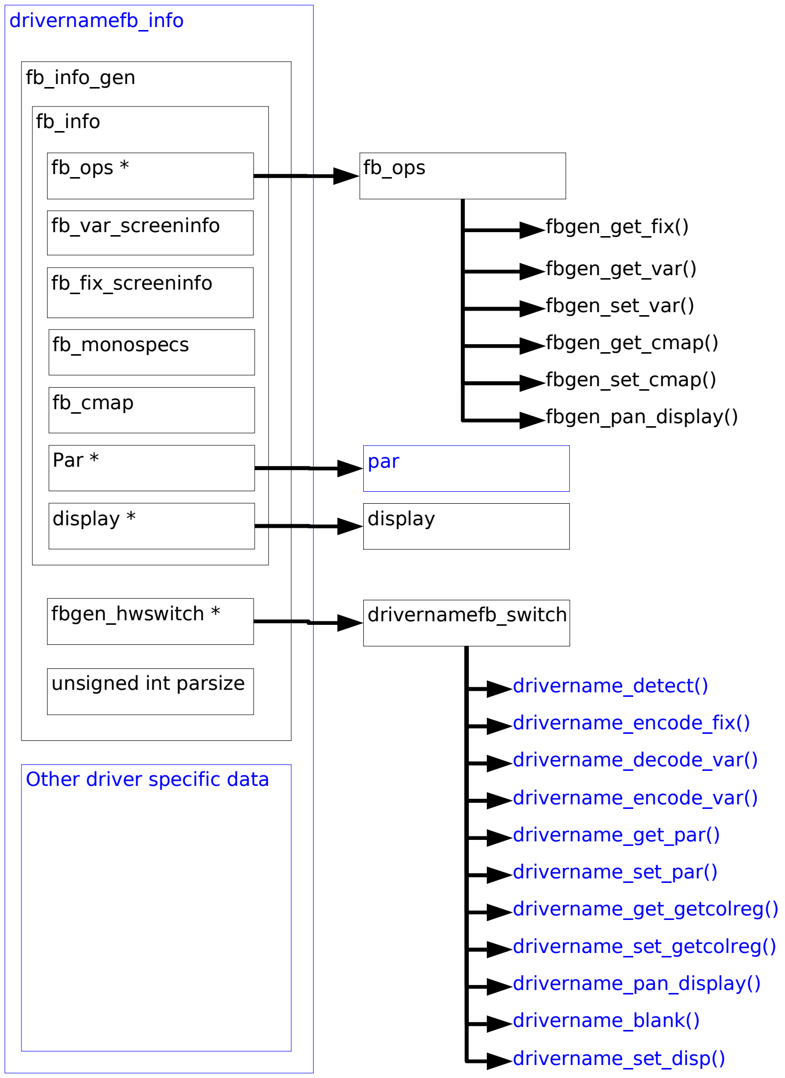


Illustration 12 Diagram of the skeleton fame buffer framework. As above the black elements are generic kernel structures populated by the driver, blue elements are specific to the driver

This framework expands upon the previous one, however this time the top level structure is one defined by each driver. This usually contains the fb_info_gen structure and any other important variables that don't fit anywhere else. Inside fb_info_gen is another set of function pointers in fbgen_hwswitch:

```
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);
};
```

The fb_ops structure is filled with generic functions, these traverse back up the tree of structures using the container_of macro and call the functions in fbgen_hwswitch. These functions deal directly with the driver specific par structure, reading and writing the values in par to and from the hardware and the other generic structures. Each function in fbgen_hwsitch therefore has a simpler, well defined purpose, making the task of fitting the hardware requirements into the generic framework easier.

The provision of two methods of implementing a frame buffer driver is a source of confusion, particularly since each of the drivers present in the kernel appears to be based on one or the other at random.

5.5 The Fujitsu Cremson graphics controller

The Calypso 32 graphics card which can be inserted into the Pluto 6 board has 32MB of on board DRAM, and a Fujitsu MB86290A Cremson video controller. The Fujitsu Cremson is one of a line of video controllers targeted at embedded systems, specifically automotive displays (GPS navigation, in car entertainment), palmtop/HPC video output, and consumer devices such as digital set top boxes.

Device	**Name**	**Chip freq**	**2D**	**3D**	**Texture Mapping**	**No of Layers**	**Alpha blending**	**RGB Digital**	**RGB Analog**	**Video Memory**	**CPU Interface**	**Pack-age**
MB87J2120	Lavender	64MHz	**x**			**4**		**x**	**x**	external (up to 8MByte)	FR	BGA256
MB87P2020	Jasmine	64MHz	**x**			**4**		**x**	**x**	embedded (1MByte)	FR	QFP208
MB86290A	Cremson	100MHz	**x**	**x**	256x256	**4**	4 bit / layer		**x**	external (up to 32MByte)	FR,SH3/4, V850	QFP240
MB86291A	Scarlet	100MHz	**x**	**x**	256x256	**4**	4 bit / layer	**x**	**x**	embedded (2MByte)	FR,SH3/4, V850	QFP208
MB86292	Orchid	100MHz	**x**	**x**	256x256	**4**	4 bit / layer	**x**		external (up to 32MByte)	FR,SH3/4, V850	QFP256
MB86293	Coral Q	166MHz	**x**	**x**	4096x4096	**6**	8 bit alpha-plane	**x**		external (upto 64MByte)	FR,SH3/4, V850	QFP256
MB86294	Coral B	166MHz	**x**	**x**	4096x4096	**6**	8 bit alpha-plane	**x**	**x**	external (up to 64MByte)	FR,SH3/4, V850	QFP or BGA256
MB86295	Coral P	166MHz	**x**	**x**	4096x4096	**6**	8 bit alpha-plane	**x**	**x**	external (up to 64MByte)	PCI	BGA256
MB86296	Coral PA	166MHz	**x**	**x**	4096x4096	**6**	8 bit alpha-plane	**x**	**x**	external (up to 64MByte)	PCI (improved bw)	BGA256
MB862xx	Lime	100MHz	**x**		4096x4096	**6**	8 bit alpha-plane	**x**		external (up to 32MByte)	FR,SH3/4, V850 + serial I/F	BGA256

Table 3 Fujitsu graphics controller comparison

The Cremson was chosen to power the Calypso 16 and 32 cards which accompany the Pluto 5/6 boards respectively. The Cremson can be integrated with certain CPUs without glue logic (no the ColdFire) and supports multiple memory configurations:

Type	Data bus width	# of devices	Total capacity
SDRAM 64 Mbit (x32 bit)	32 bit	1	8 MB
SDRAM 64 Mbit (x32 bit)	64 bit	2	16 MB
SDRAM 64 Mbit (x32 bit)	64 bit	3	32 MB

Table 4 Cremson memory configurations

On the Calypso 32 graphics card the Cremson is coupled with 4 8bit DRAM chips accessed across its 64 bit data path using an address arbiter. This graphics ram is also addressable by the ColdFire host using chip select 3 and accessed over its 32 bit data bus.

See appendix C for further details on the Cremson graphics controller and its capabilities.

5.6 Implementing a Linux frame buffer driver on the Cremson

5.6.1 Design

The provision of a comprehensive framework in the kernel for frame buffer drivers meant the design of the Cremson driver was focused on interfacing what the hardware does onto what the kernel does. After some thought it was decided that the starting point was to decide what should go into the par structure. After looking at other frame buffer drivers in the kernel it was found that par commonly contains a mirror set of the registers in the graphics controller which must be written in order to set the desired video mode. This then influenced the decision to use the driver framework from skeletonfb, since most of the functions it uses are concerned with getting and setting values in par to and from hardware, and translating them to values in the other structures. The following initial decisions were made:

- To begin with only 16 bit direct colour mode will be used (8 bit can be added after the driver is running correctly)
- Only the base layer will be enabled and used for frame buffer memory.
- A known set of timing register values will be used and set in the Cremson to set the resolution, at least until the on-the-fly calculations are validated, to avoid breaking monitors.
- The initial resolution will be 1024x768.
- The generic frame buffer driver frame work from skeletonfb.c will be used.

The basis for the Cremson frame buffer driver therefore appears as follows:

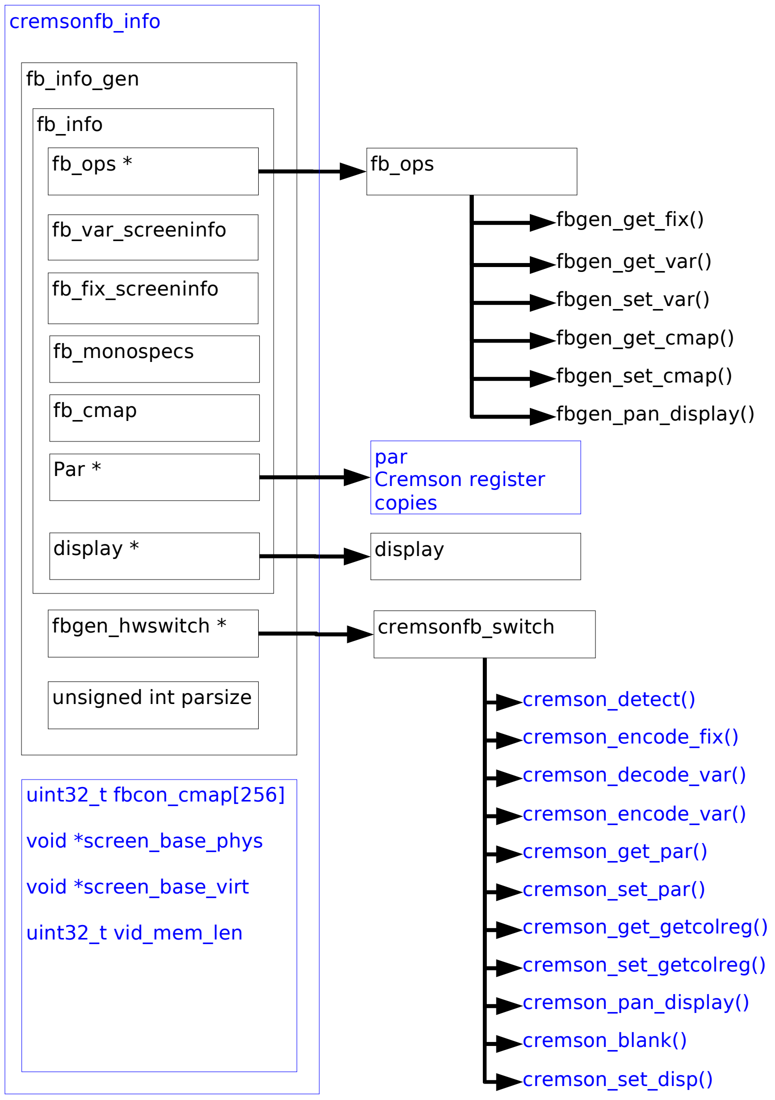


Illustration 13 Cremson frame buffer driver framework

In order to determine the Cremson registers required in par, the Heber source code was consulted and the Cremson driver followed through from power on to graphics on the screen. This was supplemented with information from the Cremson data sheet and resulted in the following steps:

Always apply these settings:

1) Set Cremson memory timing register to suit the ISSI IS42S16400B-7T DRAM chips used on the Calypso32 graphics card.

Resolution specific settings:

2) Set the following video mode registers:

- Display Control Mode Register - non interlaced, external sync off, SYNC signals active low, dot clock scalar, dot clock source
- Horizontal Total Pixels Register - X resolution + borders + sync time
- Horizontal Display Period Register - actual X resolution
- Horizontal Display Boundary Register - same as HDP (no layer splitting)
- HSync Position Register - X resolution + right border
- HSync Width Register
- Vsync Width Register
- Vert Total Rasters Register - Y resolution + borders + sync time
- Vsync Position Register - Y resolution + bottom border
- Vertical Display Period Register - Y resolution

Layer specific settings:
3) Set the following layer registers:

- Base Left layer mode - 8bit indirect/16bit direct colour mode, double buffering on/off, width and height of layer in pixels
- Base Left layer origin address 0 - address of start of layer in video memory for layer frame 0
- Base Left layer display address 0 - address in video memory to display in layer frame 0 (used for panning)
- Base Left layer origin address 1 - address of start of layer in video memory for layer frame 1
- Base Left layer display address 1 - address in video memory to display in layer frame 1 (used for panning)
- Base Left layer display position X - Position of layer on screen
- Base Left layer display position Y - Position of layer on screen
-
- Display control enable - turn on the display and required layer

The par structure created therefore includes all the registers set in the above steps:

```
struct cremsonfb_par {
  __u32 xres;           /* Hor Resolution px */
  __u32 yres;           /* Vert Resolution px */
  __u8  bpp;            /* bits per pixel */
  /* vid mode register shadows */
  __u16 dcm;            /* Display Control Mode Register  */
  __u16 htp;            /* Hor Total Pixels Register  */
  __u16 hdp;            /* Hor Display Period Register  */
  __u16 hdb;           /* Hor Display Boundary Register  */
  __u16 hsp;           /* HSync Position Register  */
  __u8  hsw;           /* HSync Width Register  */
  __u8  vsw;           /* Vsync Width Register  */
  __u16 vtr;           /* Vert Total Rasters Register  */
  __u16 vsp;           /* Vsync Position Register  */
  __u16 vdp;           /* Vert Display Period Register  */

  /* layer information */
  __u8  layer_num;     /* BL | BR | ML | MR | W | C */
  __u32 layer_width;   /* px */
  __u32 layer_height;  /* px */
  __u32 layer_mode;    /* SINGLE | DOUBLEBUFFER, INDIRECT | DIRECT_COLOUR,
                          layer frame width in 64 byte units, height in px */
  __u16 transparent_color;
  __u16 xoffset;  /* offset in px of layer frame from top left */
  __u16 yoffset;  /* offset in px of layer frame from top left */
  __u32 frame0_addr;         /* start address double buffer frame 0 */
  __u32 frame1_addr;         /* start address double buffer frame 1 */
  __u32 display0_addr;  /* display address double buffer frame 0 */
  __u32 display1_addr;  /* display address double buffer frame 1 */

  __u8  palette_size;        /* 16 for fake 16 bit console palette,
                                256 for real 8 bit one */
};
```

The flow diagram accompanied by the par structure could then be translated directly to the cremson_set_par() function, which simply writes all the values from step 2 and 3 in the diagram from par structure to the hardware. The final cremson_get_par() function would then be a reversal of this procedure, however to begin with a pre-defined par struct for 1024x768x16 was used to ensure known values are written to the controller.

The functions cremson_encode_var(), cremson_decode_var() and cremson_encode_fix() are for the most part a simple matter of copying over variables to/from par. The exception to this is the timing data in the fb_var_screeninfo structure, which introduces some uncertainty. The timing data variables in the structure are:

```
      /* 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_* */
```

The pixclock is the time in pico seconds that each pixel is displayed for:

```
Cremson clock rate
      = (Cremson PLL CLK source) * (clock scalar in Cremson Display Control Mode
register)
      = 200454520 * (1/3) //taken from Cremson data sheet,
                             //1/3 used in Heber code
      = 66000000
So var->pixclock = 1/66000000
      = 1.5x10^-8 Seconds
```

The rest of the timing data is in multiples of the pixclock, hsync_len and vsync_len are the number of pixels worth of time required for the monitor beams to return to their start positions after a scan. The margin values are required for the delay between the HSYNC signal turning on or off and the electron beams starting or stopping a scan. In order to accommodate this extra pixel time, the graphics controller is often told to display a larger resolution than that visible on the screen - as is the case in the pre-defined par structure for 1024x768, where the Cremson is given total pixel values of 1388x805.

5.6.2 Implementation

There was initial uncertainty as to how the frame buffer would be used on the Pluto 6 board, which has no keyboard input directly. The frame buffer and console subsystems are closely tied, and the console driver expects keyboard input.

5.6.2.1 16 bit mode

The driver was added to the configuration by editing the following files:

```
/usr/src/uClinux-dist/linux-2.4.x/drivers/video/Config.in:

   if [ "$CONFIG_PLUTO6" = "y" ]; then
      bool '  Pluto6 Cremson framebuffer' CONFIG_FB_P6CREMSON
   fi
```

/usr/src/uClinux-dist/linux-2.4.x/drivers/video/Makefile:

```
obj-$(CONFIG_FB_P6CREMSON)        += cremsonfb.o fbgen.o
```

Initially the fbgen.o was left out, causing linker errors because the generic frame buffer functions weren't being included in the build. The kernel configuration was then altered as follows:

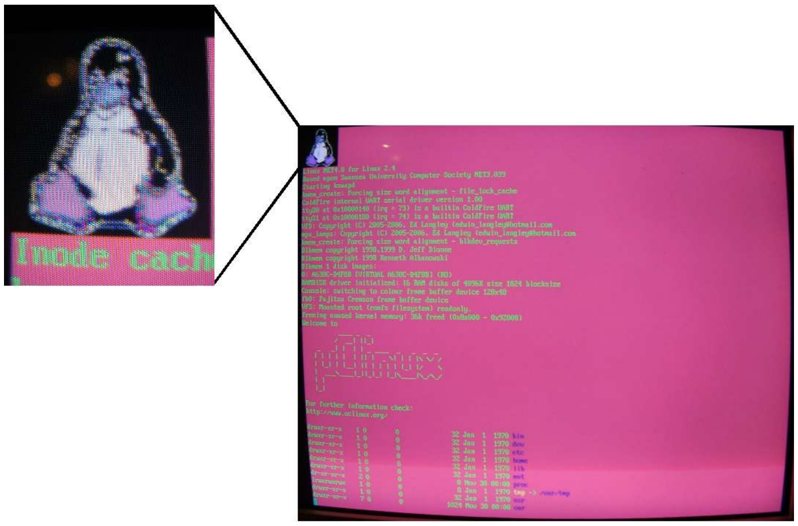
```
Kernel configuration
      Char drivers
            Virtual terminal = y //required to enable console drivers menu
            Support for console on virtual terminal = y //required to make frame
buffer do something interesting

      Console drivers
            Frame-buffer support
                  Support for frame buffer devices = y
                  Pluto 6 Cremson Framebuffer = y
                  Advanced low level driver options = y
                  16 bpp packed pixels support = y //required to include console
operations (putc, putcs etc) for this colour mode
                  Select compiled in fonts = y
                  VGA 8x16 font = y //required for many cards which don't have
built in font sets

      Kernel hacking
            Kernel boot parameter = "console=tty0 console=ttyS0,9600n8" //show some
messages on the screen as well as the serial port
```

Initial builds of the driver didn't yield any results on screen, due to the following problems, in chronological order:

- It was neglected to include <asm/io.h> in the driver, use of readb/w/h and writeb/w/h macros still compiled correctly, but didn't actually do anything.

- The Cremson is little endian, the ColdFire is big endian, so had to use cpu_to_le16/32 in writes to all the Cremson registers.
- In the header file cremsonfb.h, the #define VID_MEM_BASE was entered one 0 short, resulting in all writes to the Cremson (control registers and video memory) going to entirely the wrong address space.
- In the cremsonfb_init function, functions were called with the parameter disp.var, then other functions would check variables inside fb_info.gen.info.var.
- The variable fb_info.gen.parsize wasn't set, causing stack corruption in generic functions which create a local struct par variable.
- The bitmask CREMSON_BL_ENABLE was set the same as CREMSON_DISPLAY_ENABLE, so the display was coming on but no layer was shown.

Once these problems were fixed, the display would come on and show a pink strip of noise down the left of the screen. After close inspection of the code it was found that par->hsw and par->vsw were being written to the Cremson using writew, the registers are actually 8 bits wide, so neighbouring registers were being corrupted. Also par->hdp wasn't being written at all. With these last issues fixed, finally the following was displayed:



```
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Starting kswapd
kmem_create: Forcing size word alignment - file_lock_cache
ColdFire internal UART serial driver version 1.00
ttyS0 at 0x10000140 (irq = 73) is a builtin ColdFire UART
ttyS1 at 0x10000180 (irq = 74) is a builtin ColdFire UART
UFS: Copyright (C) 2005-2006, Ed Langley (edwin_langley@hotmail.com
mpx_lamps: Copyright (C) 2005-2006, Ed Langley (edwin_langley@hotmail.com
kmem_create: Forcing size word alignment - blkdev_requests
Blkmem copyright 1998,1999 D. Jeff Dionne
Blkmem copyright 1998 Kenneth Albanowski
Blkmem 1 disk images:
0: A630C-D4F88 [VIRTUAL A630C-D4F88] (RO)
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
Console: switching to colour frame buffer device 128x48
fb0: Fujitsu Cremson frame buffer device
VFS: Mounted root (romfs filesystem) readonly.
Freeing unused kernel memory: 36k freed (0x0a000 - 0x32000)
Welcome to

For further information check:
http://www.uclinux.org/

drwxr-xr-x   1 0        0                  32 Jan  1  1970 bin
drwxr-xr-x   1 0        0                  32 Jan  1  1970 dev
drwxr-xr-x   1 0        0                  32 Jan  1  1970 etc
drwxr-xr-x   1 0        0                  32 Jan  1  1970 home
drwxr-xr-x   1 0        0                  32 Jan  1  1970 lib
dr-xr-xr-x   2 0        0                  32 Jan  1  1970 mnt
[illegible]  1 0        0                   0 Nov 30 00:00 proc
[illegible]  1 0        0                   8 Jan  1  1970 tmp -> /var/tmp
drwxr-xr-x   7 0        0                  32 Jan  1  1970 usr
                                         1024 Nov 30 00:00 var
```

Illustration 14 A 16 bit frame buffer console with an incorrectly coloured penguin, background and text

Suddenly it became apparent that of course the video memory is treated as little endian by the Cremson and written to as big endian by the kernel. This is the reason why the penguin and the console text are discernible but the colours are all wrong. Swapping the bytes as they are written to the video memory would remedy the problem, however there is more than one point at which the frame buffer memory is read and written. The read and write functions for the device file are implemented in the file fbmem.c, these could be replaced by changing the function pointers in the fb_fops structure to point to customised versions in the driver. The console driver however is kernel resident and hence doesn't use these functions through the device node. The fix for the console was to add byte swapping in the macros fb_writew and fb_writel in the file fbcon.h:

```
#define fb_writew(b,addr) (*(volatile u16 *) (addr) = ( cpu_to_le16(b) ))

#define fb_writel(b,addr) (*(volatile u32 *) (addr) = ( cpu_to_le32(b) ))
```

This fixed the penguin logo, but not the rest of the console output:

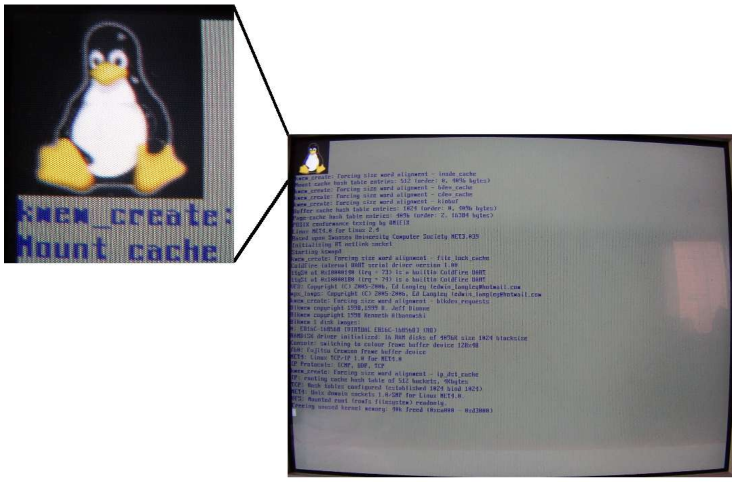


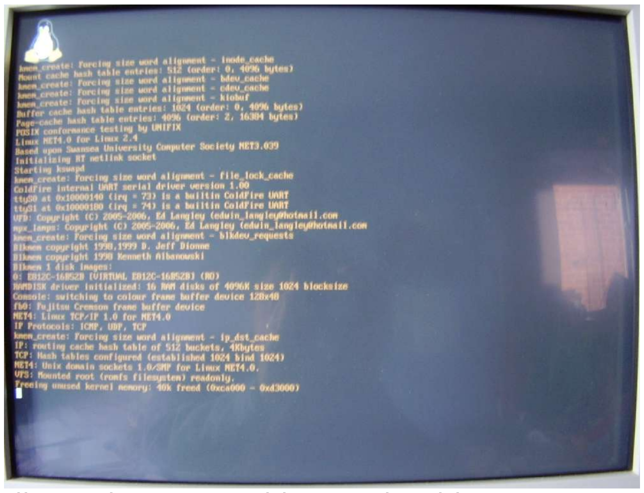
Illustration 15 A correct 16 bit penguin with still incorrectly coloured console text

Stepping through the code in GDB with breakpoints on functions using the fb_writeb/w/l macros showed that code to display the penguin uses fb_writew, the rest of the console is displayed using fb_writel. It was realised that there is not just an endian issue - a black pixel is made up of all zeroes in video memory, swapping the bytes has no effect, as shown by the background of the penguin logo before and after fixing it.

The frame buffer console uses a palette table even in direct colour mode (although the fake palette only has 16 entries), colours in the palette are set and retrieved using the functions cremson_setcolreg and cremson_getcolreg, these ensure the colour values passed in are rounded to the correct number of bits each. The reason the console background was pink originally was because the disp structure which the console driver uses to interact with the frame buffer contains a pointer to the colour palette array. This must be set in the function cremson_set_disp along with the pointer to the correct console functions for the colour depth. The pointer assignment to the palette was missing, causing wrong values to be read from a random address. The following assignment was added to cremson_set_disp:

```
disp->dispsw_data = &info->fbcon_cmap;
```

The frame buffer console then appeared as shown:



```
kmem_create: Forcing size word alignment - inode_cache
Mount cache hash table entries: 512 (order: 0, 4096 bytes)
kmem_create: Forcing size word alignment - bdev_cache
kmem_create: Forcing size word alignment - cdev_cache
kmem_create: Forcing size word alignment - kiobuf
Buffer cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 4096 (order: 2, 16384 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Initializing RT netlink socket
Starting kswapd
kmem_create: Forcing size word alignment - file_lock_cache
ColdFire internal UART serial driver version 1.00
ttyS0 at 0x10000140 (irq = 73) is a builtin ColdFire UART
ttyS1 at 0x10000180 (irq = 74) is a builtin ColdFire UART
UFD: Copyright (C) 2005-2006, Ed Langley (edwin_langley@hotmail.com
mpx_lamps: Copyright (C) 2005-2006, Ed Langley (edwin_langley@hotmail.com
kmem_create: Forcing size word alignment - blkdev_requests
Blkmem copyright 1998,1999 D. Jeff Dionne
Blkmem copyright 1998 Kenneth Albanowski
Blkmem 1 disk images:
0: E812C-16B52B [VIRTUAL E812C-16B52B] (RO)
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
Console: switching to colour frame buffer device 128x48
fb0: Fujitsu Cremson frame buffer device
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
kmem_create: Forcing size word alignment - ip_dst_cache
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 1024 bind 1024)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
VFS: Mounted root (romfs filesystem) readonly.
Freeing unused kernel memory: 40k freed (0xca000 - 0xd3000)
```

Illustration 16 A 16 bit console with correct penguin and background but orange text

The console was still not correct because the text was orange, it should be grey, as is the frame buffer console text on a normal PC. It was suspected that one of the colour components was not being written to the video memory correctly, possibly due to an alignment issue. Examining the memory with GDB confirmed that this was the case. Due to time constraints it was decided to leave this and implement the 8 bit mode.

5.6.2.2 8 bit mode

In order to test the 8 bit frame buffer console mode for the driver, a new par instance called indirect_1024_768 was created to hold the 8 bit mode settings, with the same resolution as 16 bit mode. The bpp value in the new par instance was altered from 16 to 8. This initial attempt produced the following video output:

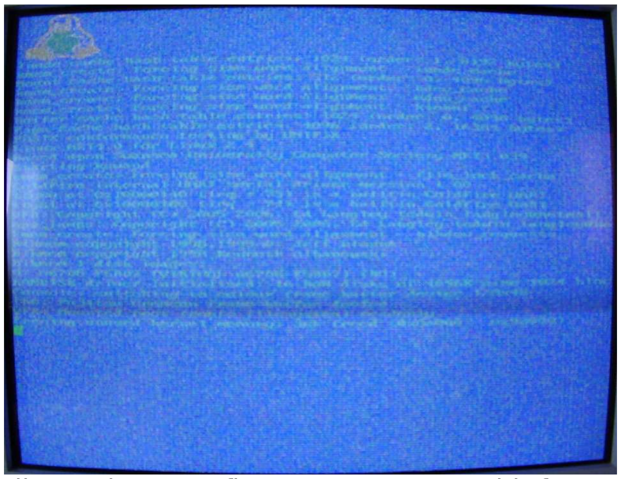


Illustration 17 A first attempt at an 8 bit frame buffer console

The penguin can be made out in the corner, twice as wide as normal. The reason for this is that the Cremson was set by the new par instance to display the contents of the video memory using 8 bits per pixel, but the kernel was still using 16 BPP, so every pixel intended to be shown by the kernel results in two pixels on the screen.

In order to set the correct BPP in the kernel, the length value of each colour bit field structure in fb_var_screeninfo had to be set correctly in the function cremson_encode_var:

```
switch(par->bpp)
{
case 8:
      var->bits_per_pixel = 8;
      /* only length in these bitfields are important for indirect colour */
      var->red.length = 8;
      var->green.length = 8;
      var->blue.length = 8;
      break;
case 16:
      var->bits_per_pixel = 16;
      var->red.offset = 10;
      var->green.offset = 5;
      var->blue.offset = 0;
      var->red.length = 5;
      var->green.length = 5;
      var->blue.length = 5;
      break;
```

```
        default:
              /* cremson doesn't do any other colour depths */
        }
```

The kernel then produced the following frame buffer console:

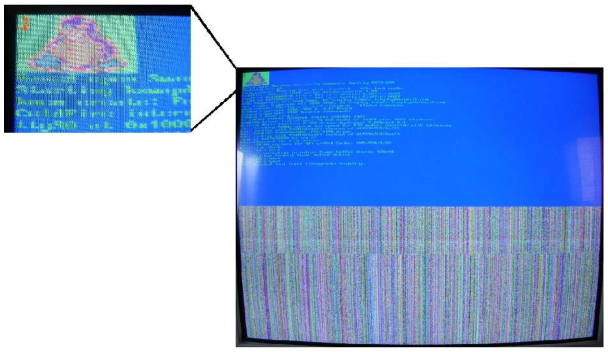
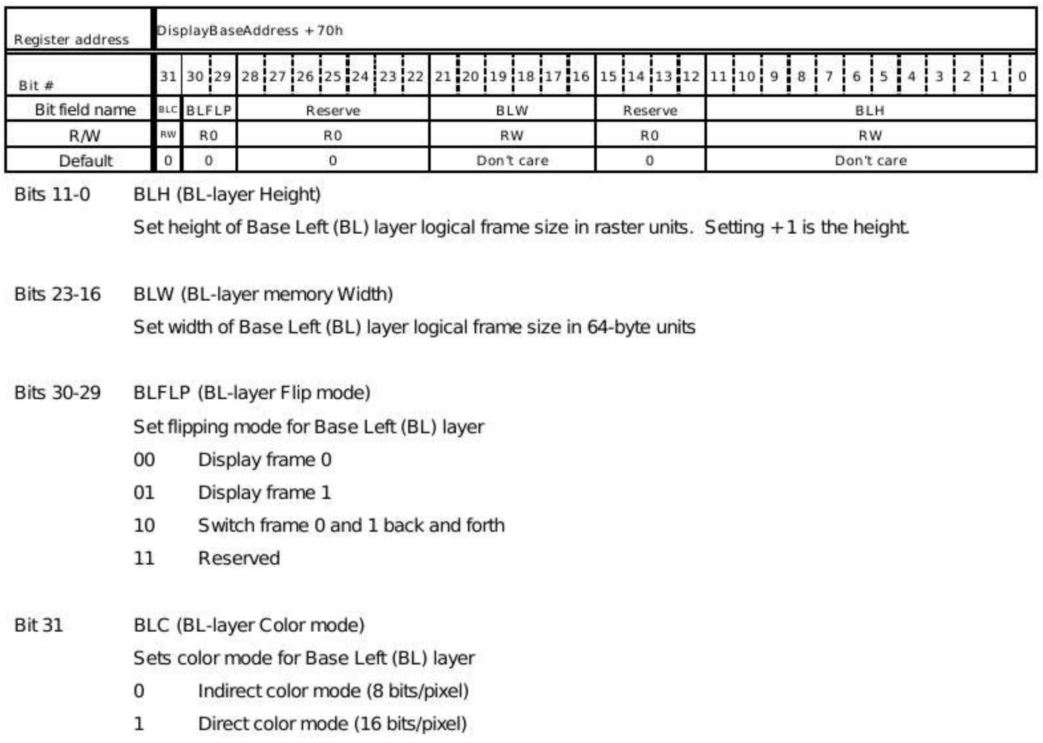


Illustration 18 An 8 bit frame buffer console with full width, but only half height

The width of the penguin and text was the same as 16 bit mode at 1024x768. An incorrect setting in the Cremson registers was suspected at this point. The main display registers control the resolution and so don't need changing. The colour depth dependent settings are in the layer registers, so the Base Left layer Mode (BLM) register setting was examined in the Heber code. Curiously, it is the width field in this register which must be changed, not the height. The layout of the register is shown here:



Register address	DisplayBaseAddress + 70h					
Bit #	31	30 29	28 27 26 25 24 23 22	21 20 19 18 17 16	15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
Bit field name	BLC	BLFLP	Reserve	BLW	Reserve	BLH
R/W	RW	R0	R0	RW	R0	RW
Default	0	0	0	Don't care	0	Don't care

Bits 11-0 BLH (BL-layer Height)

Set height of Base Left (BL) layer logical frame size in raster units. Setting +1 is the height.

Bits 23-16 BLW (BL-layer memory Width)

Set width of Base Left (BL) layer logical frame size in 64-byte units

Bits 30-29 BLFLP (BL-layer Flip mode)

Set flipping mode for Base Left (BL) layer

00	Display frame 0
01	Display frame 1
10	Switch frame 0 and 1 back and forth
11	Reserved

Bit 31 BLC (BL-layer Color mode)

Sets color mode for Base Left (BL) layer

0	Indirect color mode (8 bits/pixel)
1	Direct color mode (16 bits/pixel)

Illustration 19 The layout of the Cremson BLM register

The 16 bit setting for the BLM register was 0x80200300, so the value in the width field was 32, 32*64 is 2048 bytes. For 16 BPP this results in 1024 pixels. However for 8 BPP the same layer width setting makes the layer 2048 pixels wide, so the whole layer was displayed in only the top half of the screen.

The kernel execution was halted and the width field in the BLM register altered with GDB. The value in the register was mistakenly reduced 8*64 bytes, creating the following output:

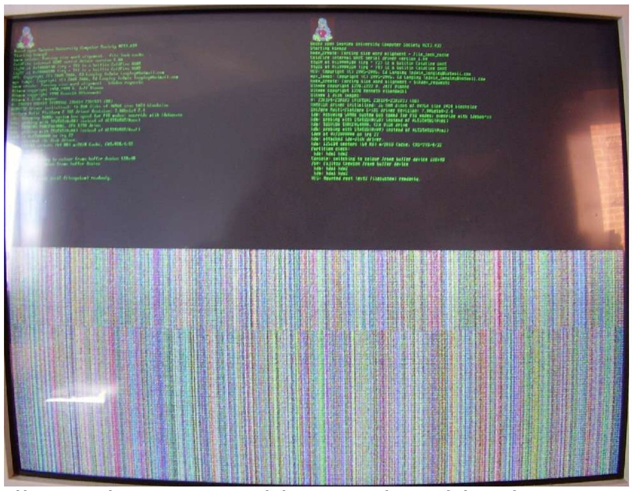
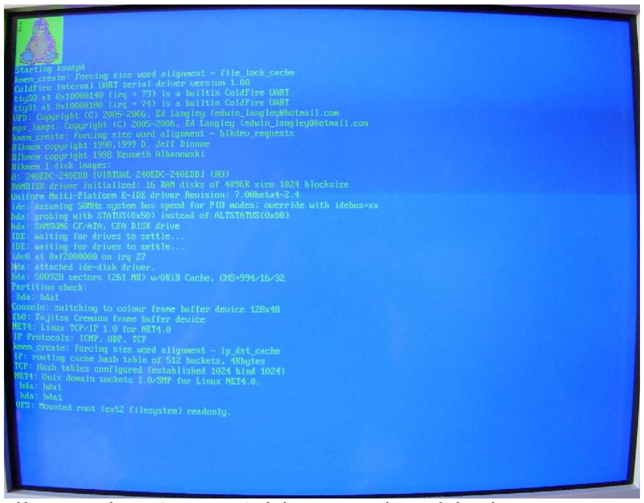


Illustration 20 An 8 bit console, with a layer width of half the value required, resulting in tiling of the layer

With the layer width now only using half of the screen resolution, the layer is tiled across, a technique that may be used to create scrolling tiled backgrounds in the games of Hebers customers. The correct layer width was applied; 16*64 bytes creates a 1024 byte wide layer, which in 8 bit mode results in 1024 pixels across. The console was then output as shown:



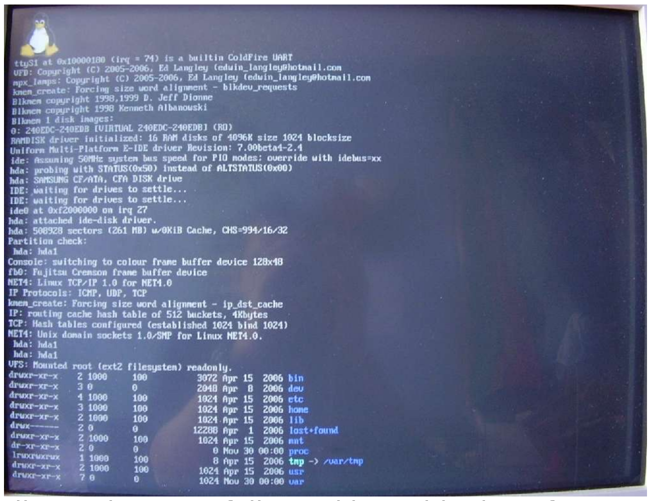
```
Starting kswapd
kmem_create: Forcing size word alignment - file_lock_cache
ColdFire internal UART serial driver version 1.00
ttyS0 at 0x10000140 (irq = 73) is a builtin ColdFire UART
ttyS1 at 0x10000180 (irq = 74) is a builtin ColdFire UART
VFD: Copyright (C) 2005-2006, Ed Langley (edwin_langley@hotmail.com
mpx_lamps: Copyright (C) 2005-2006, Ed Langley (edwin_langley@hotmail.com
kmem_create: Forcing size word alignment - blkdev_requests
Blkmem copyright 1998,1999 D. Jeff Dionne
Blkmem copyright 1998 Kenneth Albanowski
Blkmem 1 disk images:
0: 240EDC-240EDB [VIRTUAL 240EDC-240EDB] (RO)
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
Uniform Multi-Platform E-IDE driver Revision: 7.00beta4-2.4
ide: Assuming 50MHz system bus speed for PIO modes: override with idebus=xx
hda: probing with STATUS(0x50) instead of ALTSTATUS(0x00)
hda: SAMSUNG CF/ATA, CFA DISK drive
IDE: waiting for drives to settle...
IDE: waiting for drives to settle...
ide0 at 0xf2000000 on irq 27
hda: attached ide-disk driver.
hda: 508928 sectors (261 MB) w/0KiB Cache, CHS=994/16/32
Partition check:
 hda: hda1
Console: switching to colour frame buffer device 128x48
fb0: Fujitsu Cremson frame buffer device
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
kmem_create: Forcing size word alignment - ip_dst_cache
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 1024 bind 1024)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
 hda: hda1
 hda: hda1
VFS: Mounted root (ext2 filesystem) readonly.
```

Illustration 21 An 8 bit console with the correct size, but the wrong palette

The size of the console was at least correct. The final step was to ensure the palette table in the video memory was being written with the correct colour values, in order to turn the green and blue console into grey and black. The kernel sets up the palette by calling the cremson_setcolreg and cremson_getcolreg functions in the frame buffer driver. In the 16 bit mode, these functions merely maintained the pretend 16 entry console palette used for direct colour modes. For 8 bit mode it must actually set the real palette table. The following code was added to cremson_setcolreg:

```
/* set the palette in the card */
*( (u8 *)(cremson_pal_ptr + 4*regno + 3)) = 0; /* no alpha blending*/
*( (u8 *)(cremson_pal_ptr + 4*regno + 2)) = ((u8)(red >> 8));
*( (u8 *)(cremson_pal_ptr + 4*regno + 1)) = ((u8)(green >> 8));
*( (u8 *)(cremson_pal_ptr + 4*regno)) = ((u8)(blue >> 8));
```

This change produced the console output shown below:



```
ttyS1 at 0x10000180 (irq = 74) is a builtin ColdFire UART
UFD: Copyright (C) 2005-2006, Ed Langley (edwin_langley@hotmail.com
mpx_lamps: Copyright (C) 2005-2006, Ed Langley (edwin_langley@hotmail.com
kmem_create: Forcing size word alignment - blkdev_requests
Blkmem copyright 1998,1999 D. Jeff Dionne
Blkmem copyright 1998 Kenneth Albanowski
Blkmem 1 disk images:
0: 240EDC-240EDB [VIRTUAL 240EDC-240EDB] (RO)
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
Uniform Multi-Platform E-IDE driver Revision: 7.00beta4-2.4
ide: Assuming 50MHz system bus speed for PIO modes: override with idebus=xx
hda: probing with STATUS(0x50) instead of ALTSTATUS(0x00)
hda: SAMSUNG CF/ATA, CFA DISK drive
IDE: waiting for drives to settle...
IDE: waiting for drives to settle...
ide0 at 0xf2000000 on irq 27
hda: attached ide-disk driver.
hda: 508928 sectors (261 MB) w/0KiB Cache, CHS=994/16/32
Partition check:
 hda: hda1
Console: switching to colour frame buffer device 128x48
fb0: Fujitsu Cremson frame buffer device
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
kmem_create: Forcing size word alignment - ip_dst_cache
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 1024 bind 1024)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
 hda: hda1
 hda: hda1
VFS: Mounted root (ext2 filesystem) readonly.
drwxr-xr-x    2 1000     100          3072 Apr 15  2006 bin
drwxr-xr-x    3 0        0            2048 Apr  8  2006 dev
drwxr-xr-x    4 1000     100          1024 Apr 15  2006 etc
drwxr-xr-x    3 1000     100          1024 Apr 15  2006 home
drwxr-xr-x    2 1000     100          1024 Apr 15  2006 lib
drwx------    2 0        0           12288 Apr  1  2006 lost+found
drwxr-xr-x    2 1000     100          1024 Apr 15  2006 mnt
dr-xr-xr-x    2 0        0               0 Nov 30 00:00 proc
lrwxrwxrwx    1 1000     100             8 Apr 15  2006 tmp -> /var/tmp
drwxr-xr-x    2 1000     100          1024 Apr 15  2006 usr
drwxr-xr-x    7 0        0            1024 Nov 30 00:00 var
```

Illustration 22 A fully working 8 bit Linux frame buffer console on the Cremson

Finally the correct colours were shown for both the penguin and the console text and background.

5.6.3 Running Nano X

Nano-X is a very small X server which can be configured to operate using the Linux frame buffer device interface. It is part of the Microwindows project as opposed to Xfree86 or Xorg, but shares similar characteristics with the standard X window system, for example using the network layer to communicate between the server and clients. It is API compatible with the standard X interface - graphical applications should be able to run on it without substantial code modification.

After successfully compiling Nano X and adding it to the ROMFS, it gave the following output:

```
# nano-X
Cannot bind to named socket
```

This required that networking support be added to the kernel configuration. Having done this the kernel then crashed on start up, and it was found that a number of network drivers are enabled by default, and their probing routines were causing exceptions. Having turned these off and rebuilt the kernel the next error message said:

```
# nano-X
Cannot mmap /dev/fb0
```

This was because the Cremson frame buffer driver didn't implement the mmap() system call. In order to do this the get_fb_unmapped_area function pointer had to be added to the fb_ops structure of the

Cremson frame buffer driver:

```
static struct fb_ops cremsonfb_ops = {
      owner:            THIS_MODULE,
      fb_get_fix: fbgen_get_fix,
      fb_get_var: fbgen_get_var,
      fb_set_var: fbgen_set_var,
      fb_get_cmap:      fbgen_get_cmap,
      fb_set_cmap:      fbgen_set_cmap,
      fb_pan_display:   fbgen_pan_display,
      get_fb_unmapped_area: cremson_get_unmapped_area,
};
```

The cremson_get_unmapped_area function is very simple, it just returns an address to which the calling application can map the memory of the device. For devices which aren't directly addressable by the CPU, memory would have to be allocated, and support code run to ensure values placed there are put into the device. The video memory is directly addressable, so a pointer to it can be returned allowing the X server to write directly into the video memory itself:

```
static unsigned long cremson_get_unmapped_area(struct file *file, unsigned long
addr, unsigned long len, unsigned long pgoff, unsigned long flags)
{
      return (unsigned long)fb_info.screen_base_phys;
}
```

5.6.3.1 16 bit mode

Running nano-X on the board then produced the following:

Illustration 23 A bare Nano-X display; black with a white cursor

and running nanowm showed this:

Illustration 24 The Nano Window Manager; the desktop is displayed

At this stage there wasn't enough detail to know whether this was displaying correctly or not. Having memory mapped the device in order to write to it directly, it is up to Nano-X to ensure the correct endianess of the video data.

5.6.3.2 8 bit mode

Running Nano-X while the Cremson frame buffer was in 8 bit mode resulted in the same black screen with white cursor, however running nanowm presented the following screen:

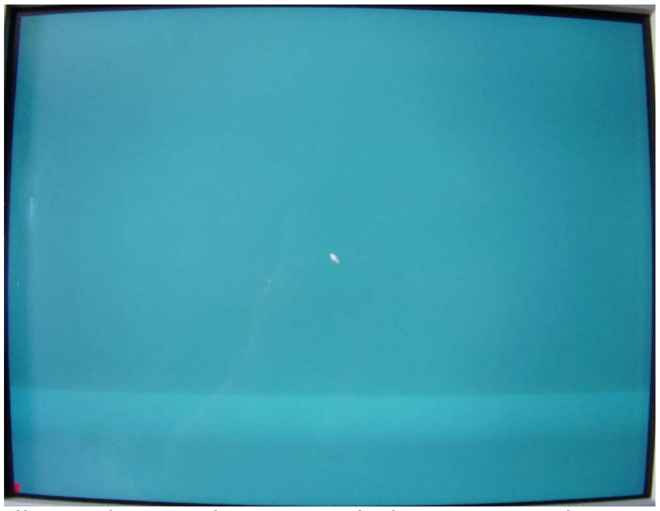


Illustration 25 The Nano Window Manager in 8 bit mode; the green desktop is the correct colour as opposed to purple

This green desktop is what is expected from Nano X. By this stage, several demonstration programs had been added to the file system, some of which are shown running here:

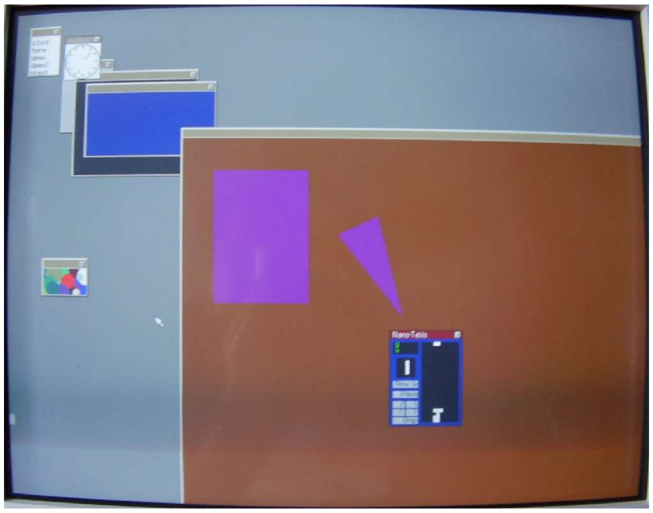


Illustration 26 A full session in Nano X; shown running are Npanel, Nclock, some demsonatrations, and a game of Ntetris

The cursor was controlled by connecting a serial mouse to the second ColdFire serial port.

6 Conclusion

The project was successful; a sufficient proportion of the Pluto 6 boards functionality has been enabled under uClinux to make it a useful example system, which was the original aim. The knowledge of implementing everything achieved will help Heber run an OS on their future boards.

6.1 Performance of the system

One purpose of the project was to establish how well the Pluto 6 performed with an OS. The resulting performance of uClinux on the board is rather poor, however this is due more to the low specification of the 5206e than the OS or the drivers. Noticeable pauses occur particularly when running commands which make extensive use of the file system.

6.2 Further work

One area where performance is lacking is the absence of double buffering in the frame buffer driver. When dragging windows or really doing anything in Nano-X, the screen refresh is very slow. In order to implement the double buffering, an interrupt routine must be implemented for the vertical synchronisation interrupt of the Cremson video controller. As described in the IDE implementation, the FPGA outputs the same interrupt level for both the Cremson and the IDE. This could be dealt with by using shared interrupts in the kernel, however the IDE call to request_irq() is made deep within the IDE layer rather than in the particular device driver. Sharing the interrupts also has the side effect that the devices would have the same priority, whilst completing a transaction in an IDE interrupt, the double buffering would be frozen. A more elegant solution would be to alter the VHDL of the FPGA to use separate interrupt levels for the devices, however the software required for this simple change means this is a job for Heber.

Another area which uClinux provides support is for the MBus module on the ColdFire. This would then open up access to the PIC MCU and the features programmed in it, as well as the EEPROM. This extra functionality is provided by the kernel for minimal effort, and would be another good example of the benefits of running an OS on the board.

6.3 What has been learned

The project has clarified the interaction between an operating system and the hardware that it runs on. The process of porting an OS to fresh hardware is a valuable exercise, allowing an instinct for solutions to grow. Towards the end of the project, the reliance on the debugger reduced, while answers to problems were predicted with increasing accuracy.

Despite this, the debugging and downloading facilities provided by the BDM were vital. Without them, porting the kernel would have taken much longer. The power to break into kernel code also helped when developing the drivers. In the future, a strong recommendation for debug facilities will be made to those in charge of picking hardware for a project.

A lot has been learnt about the inner workings of the Linux kernel. Despite using uClinux, the process of writing device drivers for Linux is largely the same. The changes required for different architectures often come down to those mentioned in chapter 3.

A good deal of experience has been gained of the issues encountered when working with embedded systems. A log book full of problems and solutions is testament to this.

7 Bibliography

7.1 Books

Bovet D., Cesati M., 2002. *Understanding the Linux kernel*. 2nd ed. Sebastopol, CA: O'Reilly.

Corbet J., Rubini A., 2001. *Linux Device Drivers*. 2nd ed. Sebastopol, CA: O'Reilly.

Corbet J., *et al*, 2005. *Linux Device Drivers*. 3rd ed. Sebastopol, CA: O'Reilly.

Hollabaugh C., 2002. *Embedded Linux: Hardware, Software and Interfacing*. Reading, MA: Addison-Wesley.

Kernighan B., Ritchie D., 1988. *The C Programming Language*. 2nd ed. Upper Saddle River, NJ: Prentice Hall.

Mitchell M., *et al*, 2001. *Advanced Linux Programming*. Indianapolis, IN: New Riders.

Pate S., 2003. *UNIX Filesystems - Evolution, Design and Implementation*. Indianapolis, IN: Wiley.

Sutter E., 2002. *Embedded Systems Firmware Demystified*. Gilroy, CA: CMP books.

Williams R., 2001. *Computer Systems Architecture A Networking Approach*. Essex: Addison-Wesley.

Yaghmour K., 2003. *Building Embedded Linux Systems*. Sebastopol, CA: O'Reilly.

7.2 Articles

Anon, 2006. Framebuffer. *Wikipedia* [online] (28 April 2006). Available from: http://en.wikipedia.org/wiki/Frame_buffer [Accessed 1 May 2006].

Anon, 2006. Planar. *Wikipedia* [online] (23 January 2006). Available from: http://en.wikipedia.org/wiki/Planar [Accessed 2 February 2006].

Braendler D., 2005. Debugging uClinux on ColdFire [online]. Available from: http://www.luv.asn.au/overheads/embedded/Debugging_uClinux_on_ColdFire.pdf [Accessed 26 October 2005].

Buell A., 2000. Framebuffer HOWTO. *The Linux documentation project* [online] (27 February 2000). Available from: http://www.tldp.org/HOWTO/Framebuffer-HOWTO.html [Accessed 4 February 2006].

Burian M., et al, 2005. The Linux Kernel Module Programming Guide. *The Linux Documentation Project* [online] (December 2005). Available from: http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf [Accessed 20 January 2005].

Cybertec, 2005. Crossfire Debugging Guide. [online]. Available from: http://www.cybertec.com.au/documents/CrossFire/examples/Debugging.html [Accessed 6 October 2005].

Erlich S., 2004. Custom Linux: A Porting Guide - Porting LinuxPPC to a custom SBC. *The Linux Documentation Project* [online] (March 2004). Available from: http://www.tldp.org/LDP/cpg/Custom-Porting-Guide.pdf [Accessed 20 October 2005].

McCullough D., 2004. uClinux for Linux Programmers. *Linux Journal* [online] (1 July 2004). Available from: http://www.linuxjournal.com/article/7221 [Accessed 2 January 2005].

McCullough D., 2002. Why does uClinux still use pages?. *uCdot* [online]. Available from: http://www.ucdot.org/article.pl?sid=02/09/16/0156259&mode=thread [Accessed 3 march 2005].

McCullough D., 2002. Why is malloc different under uClinux?. *Cyberguard* [online], Technical bulletin #11. Available from: http://www.cyberguard.info/snapgear/tb20020530.html [Accessed 13 April 2005].

Peacock C., 2005. Gcc-2.95.3 m68k-elf for uClinux. *Beyond Logic* [online]. Available from: http://www.beyondlogic.org/uClinux/gcc-2.95.3.pdf [Accessed 8 October 2005].

Peacock C., 2005. uClinux - BFLT Binary Flat Format. *Beyond Logic* [online] (15 June 2005). Available from: http://www.beyondlogic.org/uClinux/bflt.htm [Accessed 15 January 2006].

Simmons J., 2001. Linux Framebuffer Driver Writing HOWTO. *The Linux documentation project* [online] (November 2001). Available from: http://linuxconsole.sourceforge.net/fbdev/HOWTO/index.html [Accessed 6 February 2006].

Wilshire P., 2002. eXecute In Place (XIP) Overview. *System Design and Consulting Services* [online] (22 October 2002). Available from: http://www.ucdot.org/article.pl?sid=02/08/28/0434210&mode=thread [Accessed 2 January 2005].

7.3 Papers

Ben-Yehuda M., 2003. Linux Kernel Oopsing. *In*: *LKDSG 2003* [online]. Available from: http://www.mulix.org/lectures/kernel_oopsing/kernel_oopsing.pdf [accessed 26 October 2005].

COMPACTFLASH ASSOCIATION. 2003. *CF+ and CompactFlash Specification Revision 3.0*. Palo Alto, CA: CompactFlash Association.

Drabik J., 2002. World's First Embedded Linux Distribution Keeps on Growing [online]. Toronto: Arcturus Networks. Available from: http://www.arcturusnetworks.com/Docs/AboutuClinux.pdf [Accessed 29 January 2006].

Nikkanen K., 2003. UCLINUX AS AN EMBEDDED SOLUTION. Bachelor's Thesis, Turku Polytechnic.

Ungerer G., 2004. uClinux Micro Controller Linux. In: linux.conf.au, Adelaide 12-17 January 2004. [online] Available from: http://www.linux.org.au/conf/2004/eventrecord/LCA2004-cd/papers/16-greg-ungerer-uClinux.doc [Accessed 26 October 2005].

7.4 Manuals

FUJITSU. 2000. *MB86290A Graphics Controller Hardware Specifications Revision 2.0b*. Fujitsu Limited.

MOTOROLA. 1998. *MCF5206e ColdFire® Integrated Microprocessor User's Manual*. Motorola, Inc.

MOTOROLA. 1998. *MCF5307 ColdFire® Integrated Microprocessor User's Manual*. Motorola, Inc.

7.5 Websites

BDM TOOLS, 2006. *Sourceforge.net BDM tools* [online]. Available from: http://sourceforge.net/projects/bdm/ [Accessed 15 September 2005].

FREESCALE SEMICONDUCTOR, 2006. *68k/ColdFire* [online]. Available from: http://www.freescale.com/coldfire [Accessed 3 April 2006].

FUJITSU, 2006. Graphics Display Controller Products [online]. Available from: http://www.fujitsu.com/ca/en/services/edevices/microelectronics/otherassps/graphicdisplay/products/ [Accessed 4 February 2006].

LAUTERBACH, 2006. *Lauterbach GmbH - TRACE32 Microprocessor Development Tools* [online]. Available from: http://www.lauterbach.com/frames.html [Accessed 12 September 2005].

LINUX CROSS-REFERENCE, 2005. *Cross-Referencing Linux kernel sources* [online]. Available from: http://lxr.linux.no/ [Accessed 20 September 2005].

LINUXBIOS, 2006. *The linuxBIOS homepage* [online]. Available from: http://www.linuxbios.org/index.php/Main_Page [Accessed 15 December 2005].

P&E, 2005. *P&E Microcomputer Systems* [online]. Available from: http://www.pemicro.com/ [Accessed 12 September 2005].

THE UCLINUX DIRECTORY, 2006. *The uClinux Directory* [online]. Available from: http://uclinux.home.at/ [Accessed 1 October 2005].

THE NANO-X WINDOW SYSTEM, 2005. *The Nano-X Window System* [online]. Available from: http://www.microwindows.org/ [Accessed 27 February 2005].

UCDOT, 2006. *Embedded Linux and uClinux Developer Forum* [online]. Available from: http://www.ucdot.org/ [Accessed 8 October 2005].

UCLIBC, 2006. *uClibc Embedded C Library* [online]. Available from: http://www.uclibc.org/ [Accessed 8 October 2005].

UCLINUX EMBEDDED LINUX MICRO CONTROLLER PROJECT, 2006. *uClinux – Embedded Linux Micro Controller Project* [online]. Available from: http://www.uclinux.org/ [Accessed 6 October 2005].

7.6 Mailing lists

COLD FIRE MAILING LIST, 2006. ColdFire Mailing List [online]. Available from: http://www.wildrice.com/ColdFire/Subscribe.html [Accessed 14 November 2005].

LKML, 2006. The Linux Mailing List Archive [online]. Available from: http://lkml.org/ [Accessed 14 November 2005].

UCLINUX MAILING LIST SITE SEARCH, 2006. uClinux-dev mailing list site search [online]. Available from: http://mailman.uclinux.org/htdig/ [Accessed 14 November 2005].

8 Glossary

ASCII	American Standard Code for Information Interchange
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
CTS	Clear To Send
DMA	Dynamic Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
DUART	Dual Asynchronous Receiver/Transmitter
EEPROM	Electrically Erasable Programmable Read Only Memory
FPGA	Field Programmable Gate Array
GNU	GNU's Not Unix
GPL	GNU Public License
I2C	Inter Integrated Circuit serial communications bus
IC	Integrated Circuit
IDE	Integrated Drive Electronics
IO	Input/Output
ISA	Industry Standard Architecture
LMA	Load Memory Address
MBUS	Motorola Bus (I2C compatible)
MMU	Memory Management Unit
NiMH	Nickel Metal-Hydride
OS	Operating System
PC	Personal Computer
PIC	Peripheral Interrupt Controller (Adopted by Microchip as a micro controller name)
PCI	Peripheral Component Interconnect
PDA	Personal Digital Assistant
RAM	Random Access Memory
RISC	Reduced Instruction Set
RTS	Ready To Send
Rx	Receiver
SBC	Single Board Computer
SCADA	Supervisory Control And Data Acquisition
SRAM	Static Random Access Memory

ASCII	American Standard Code for Information Interchange
Tx	Transmitter
VMA	Virtual Memory Address
VPN	Virtual Private Network
XIP	eXecute In Place

9 Appendix A - Further ColdFire information

The ColdFire 5206e is a 32 bit micro controller derived from the 68000 series, however the core is simplified and provided with on chip peripherals for embedded markets. The rationale behind the introduction of the ColdFire is that many of the more obscure instructions from the 68000 are not used by high level language constructs. Motorola looked to the RISC market and saw that reduced instruction set processors provide high performance for a lower degree of chip complexity (and are therefore cheaper). However the fixed length instructions used in RISC don't create dense code. Code size doesn't matter in desktop computers but in embedded control applications it can become very important. This led to Motorola basing the ColdFire on the concept of variable length RISC - the instructions are varied in length creating smaller code, but the rest of the architecture shares characteristics with other RISC architectures. The smaller code allows system builders to reach the same target performance with slower, cheaper memory.

The 5206e is based on a version 2 ColdFire core with the following features:
- 40/54Mhz clock speeds
- Multiply and accumulate (MAC) unit supporting 16x16 and 32x32 bit multiplication and 32 bit accumulate
- Hardware divide module supporting 32/32 and 32/16 operations
- User/supervisor modes - access to supervisor registers e.g. upper word of condition control register (CCR) is disabled in user mode.
- 32 bit internal and external data bus
- 32 bit internal address bus, 28 bit external address bus, with the top four pins A[27:24] multiplexed with the top four chip select lines CS[7:4] and the four write enable lines WE[3:1]
- 8 data registers D[7:0]
- 7 address registers A[6:0]
- 1 stack pointer register A7
- 1 vector base register allowing the exception vector table to be located anywhere in addressable memory.
- On chip instruction cache with one control register and two access registers allowing portions of the memory space to be non-cache-able.
- 81 instruction set.

The ColdFire range is designed for embedded control applications, the 5206e provides the following on chip peripherals:
- 8kb on chip SRAM providing one clock cycle access to data
- DRAM controller supporting up to to 2 banks
- Dual UARTs
- MBUS (I2C compatible)
- Dual 16 bit timers,
- 8 bit parallel port,
- DMA controller with two channels
- Chip select module supporting up to 8 chip select lines with individual control and address registers allowing differing data port sizes and wait states for varying external peripherals
- System integration module (SIM) incorporating interrupt controller, watchdog timer and pin assignment control for handling multiplexed pins on the chip, plus control registers for the other the on chip peripherals

The 5206e is the enhanced version of the 5206, with the same peripheral set, as well as DMA, MAC, Hardware Divide, a larger cache, and larger built in SRAM. It is pin compatible with the MCF5206, with the DMA pins muxed with Timer 0 pins.

The ColdFire has 4 states of operation:
- Normal processing - in either user or supervisor mode
- Exception processing - a trap or interrupt has occurred, the processor resumes normal processing

when the fetch of the first instruction of the handler is initiated

- Stopped - this is a low power mode state entered by executing the STOP instruction. The ColdFire wakes up when a non-masked interrupt or reset occurs
- Halted - if an exception (e.g. An access error) occurs when in the exception processing state, the processor cannot save the registers for a context switch, nor return to normal processing, the the ColdFire will halt. Only a reset can return the processor from this state.

9.1 Differences to the 68000

The ColdFire is a derivative of the M68000 core, with the following differences to features introduced in 68xxx cores after the 68020:

9.1.1 Simplified addressing modes:

Fully supported in ColdFire V2:

- Data Register Direct
- Address Register Direct
- Address Register Indirect
- Post-increment
- Pre-decrement
- Displacement (16-bit displacement)
- PC Displacement (16-bit displacement)
- Absolute Short
- Absolute Long
- Immediate

Partially supported in ColdFire V2 :

- Indexed
- PC Indexed

The restrictions on these two modes are:

- The displacement constant is 8-bit only;
- "Zero-suppressed" registers are not supported;
- The Index register can only be handled as a Long. Word-length index registers are not supported.
- The scale factor must be 1, 2, or 4. Scale factors of 8 are not supported.

Not implemented at all in ColdFire V2:

- Memory-indirect post-indexed
- Memory-indirect pre-indexed
- PC-indirect post-indexed
- PC-indirect pre-indexed

Further restrictions may be imposed on the addressing modes supported by particular instructions, even if a particular addressing mode is itself available on ColdFire.

9.1.2 Reduced instruction set

Not available in ColdFire V2:
DBcc, EXG, RTR, RTD, CMPM,
ROL, ROR, ROXL, ROXR, MOVE16
ABCD, SBCD, NBCD
BFCHG, BFCLR, BFEXTS, BFEXTU
BFFFO, BFINS, BFSET, BFTST
CALLM, RTM, PACK, UNPK

CHK, CHK2, CMP2, CAS, CAS2, TAS *(restored in V4 core)*,
BKPT, BGND, LPSTOP, TBLU, TBLS, TBLUN, TBLSN
TRAPV, TRAPcc, MOVEP, MOVES, RESET
ORI to CCR, EORI to CCR, ANDI to CCR

9.1.3 Restricted instructions

MULU (multiply unsigned) and MULS (multiply signed) producing a 64-bit result are not implemented, but 16 x 16 producing 32-bit, and 32 x 32 producing (truncated) 32-bit, are available.

Most arithmetic and logical instructions can act on Long words only. This applies to:
ADD, ADDA, ADDI, ADDQ, ADDX, AND, ANDI, ASL, ASR
CMP, CMPI *(word/byte forms re-introduced in version 4 core)*
CMPA, EOR, EORI, LSL, LSR,
NEG, NEGX, NOT, OR, ORI,
SUB, SUBA, SUBI, SUBQ, SUBX
MOVEM.W has also been removed from the instruction set.
The only instructions which do act on the full set of byte, word and long operands are CLR, MOVE and TST. EXT.W, EXTB.L and EXT.L survive, as do MULx.W and MULx.L

Some arithmetic instructions cannot act directly on memory - the destination must be a register. This applies to:
ADDI, ADDX, ANDI, CMPI, ASL, ASR, LSL, LSR,
NEG, NEGX, NOT, EORI, ORI, SUBI, SUBX, Scc
Note that ADDQ and SUBQ **can** act directly on memory

In most cases the ColdFire instructions operate in the same exactly the same manner as the 68xxx counterparts, however there are exceptions, for example MULU and MULS do not set the overflow bit.

Note: Some of these differences do not apply to later ColdFire versions using the V4 core - instructions have been put back, as well as new variations of existing ones.

9.1.4 Supervisor mode differences

There is only one stack shared between user and supervisor modes (i.e. A single stack pointer) so interrupt handlers must not write below their stack frame, otherwise the user stack beneath it will be corrupted.

10 Appendix B - Further Pluto 6 information

10.1 Accompanying Hardware on the Pluto 6

The ColdFire is in overall control of the Pluto 6 board, however it is assisted by a number of peripheral hardware items.

10.1.1 FPGA

A Xilinx Spartan FPGA as configured by Heber controls the routing between each of the 6 serial port devices and the physical connection. Each serial device has its Tx/Rx and CTS/RTS lines connected to the FPGA. The ColdFire can set the FPGA to route the signals to any of the physical port connections.
As well as the serial ports, the FPGA also handles the strobing of the multiplexed lamp arrays.
The FPGA also has security functionality. Areas of binaries to be loaded onto the board can be encrypted with a key specific to each customer. The FPGA configuration is specific to the customer and will decrypt the code once it is loaded into RAM on the board. The address map of the registers inside the FPGA is different for each customer configuration, requiring custom header files for the software.
A secured game will only run on a single customers board, preventing others from stealing the code from a fruit machine and running it on theirs, profiting from other peoples game development.

10.1.2 Microchip PIC

A PIC 18F452 micro controller made by Microchip is programmed by Heber to contain a real time clock, security logs and the FPGA configuration to load into the Spartan FPGA at power on. While the Pluto 6 board is powered off the PIC continues to run in low power mode, powered by a 3.6V Ni-MH rechargeable battery. The PIC scales up its timers to create a clock which records the time of day, this and the other data can be fetched and set by the ColdFire via I2C. While the board is powered down the PIC will monitor four switches, these are intended as door switches or similar to detect tampering with the fruit machine. Any switch activity is recorded in up to 10 time stamped logs. It is up to the writer of the game to fetch these logs from the PIC and issue warnings if necessary. The FPGA configuration is stored with the rest of the code for the PIC in its flash ROM. This is done by converting the FPGA configuration file into C code containing the raw binary data in an array. The configuration can then be programmed into the PIC as data which the PIC code can access and feed out to the FPGA at power on. This allows the FPGA configuration to be altered by removing the socketed PIC chip and reprogramming it.

10.1.3 CPLD

The programming of the FPGA by the PIC is managed at power on by a CPLD. This is also customised to match the FPGA for each customer. It is non-volatile and programmed at Heber using a "bed of nails" rig, which presses contacts against certain pins on the underside of the board. This is necessary because the CPLD is contained in a BGA package to avoid hackers monitoring the signals on the pins and cracking the security. As the main section of the PCB is 4 layered they would not be able to trace the tracks from the underside of the BGA package to accessible pins. There is a code in the CPLD which must match that in the FPGA once it is programmed before the board is allowed to boot any further. This prevents pirates from removing PICs from Pluto 6 boards configured for other companies and using them (or clones) in their own boards to run stolen game code. If the CPLD and FPGA codes match, ColdFire will be allowed to come out of reset.
The CPLD also has the chip select lines routed to it and outputs them to the relevant devices if "glue" logic is needed. For example chip select 0 can activate the boot flash ROM which requires its output enable and write enable lines to be asserted appropriately or the flash can be removed and a SRAM

expansion card fitted, which is addressed using a single chip select line. Also the on-board SRAM on CS1 uses two chip select lines for the upper and lower bytes. These are split and asserted by the CPLD which takes the single CS1 input from the ColdFire.
The CPLD also provides access to the IDE registers on the CompactFlash or hard disc.

10.2 Hardware Boot sequence

To summarise the sequence of operations at power on in the device descriptions above:
The PIC MCU is periodically woken from its low power "sleep" mode by the watchdog timer interrupt and checks if the rest of the board is powered up. If not it runs a small amount of code to maintain the real time clock, reads the security switches and logs any changes on them. It then goes back into low power mode until the next watchdog timer interrupt. When the rest of the board is powered on and the PIC wakes up it branches to other code which programs the FPGA. This is done via a state machine so the PIC can maintain the real time clock at the same time. After sending the data to the FPGA the PIC polls an input which the FPGA will assert when it is running. Once the FPGA is programmed the PIC will poll the percentage, stake and DIL switch banks and flash an LED on the board on and off every second to show it is running. The PIC will continue these operations in a loop until it detects the board has powered down and will branch back to the low power loop. The switches are polled ready for requests via I2C from the ColdFire, which are interrupt driven in the PIC.
Once programmed the FPGA will continue independently of the PIC and resume the boot. Handshaking will occur between the FPGA and the CPLD, to check they both have the same code. Once the FPGA configuration is confirmed to match the CPLD, the FPGA will allow the ColdFire to come out of reset.
Code to be run by the ColdFire must be secured before it is loaded onto the board in S19 format. This is done using a small program created at the same time as the custom FPGA configuration. When running the program two address ranges are specified together with the boot loader and game S19 files to secure. The address ranges in the game S19 are filled with specific data. These ranges must be set in a boot loader header file. The custom boot loader is then loaded into the flash ROM on the board to match the secured FPGA and CPLD.
When the boot loader executes, it reads the game S19 file from Compact flash into the DRAM. It then steps through the two secured address ranges, using the values to index a pre-defined data array. The values indexed in the array are sent to the FPGA, once both address ranges have been processed the FPGA should have been sent a correct sequence of data and the security cleared. If the FPGA security is not cleared 27 seconds after board power up, the FPGA will shut down key features of the board and reset the ColdFire. After sending the security data to the FPGA the boot loader will then start execution of the loaded game regardless of whether security has been cleared or not.

10.3 ColdFire configuration on the Pluto 6

10.3.1 Pin assignments

The multiplexed pins on the MCF5206e chip are configured for their chosen function by setting bits in the pin assignment register in the system integration module (SIM). The SIM is comprised of a set of registers, accessed as offsets from the Module Base Address Register, the address of which must be set in the MBAR CPU register during initialisation of the ColdFire, this allows the SIM to be re-located anywhere on the memory map as necessary.
The Pin Assignment Register controls the function of the following multiplexed pins:

TOUT[0]/DREQ[1]	Timer 0 output (TOUT[0]) or DMA channel 1 request input
TIN[0]/DREQ[0]	Timer 0 input (TIN[0]) or DMA channel 0 request input

The Pluto 6 uses these two pins as DMA request inputs. The DMA channels are used to send audio data to the DAC and then out to the speakers.

RTS[2]/RSTO	Reset out Output (RSTO]) or UART 2 request to send

The Pluto 6 uses this pin as the request to send line for the second ColdFire UART

IRQ7/IPL2, IRQ4/IPL1, IRQ1/IPL0 Individual interrupt inputs (IRQ1, 4, 7) or encoded interrupt priority levels:

IPL2	IPL1	IPL0	
0	0	0	interrupt vector level 1
...			
1	1	1	interrupt vector level 7

The Pluto 6 uses these three interrupt lines as encoded interrupt levels. See below for details.

PP[7:4]/PST[3:0] general purpose I/O signals PP7 - PP4 or background debug mode signals PST3 - PST0
PP[3:0]/DDATA[3:0] Parallel Port output signals PP3 - PP0 or background debug mode signals DDATA3 - DDATA0
The Pluto 6 uses these pins as background debug mode signals.

CS[7:4]/A[27:24]/WE[3:0] Three bits can be set in the pin assignment register to choose from 12 combinations of write enable, upper four chip select or address lines for these four pins.
The upper four chip select lines are not required on the Pluto 6, but the external address lines A [27:24] are required to access the 32MB of graphics RAM on the Calypso 32 graphics card, so these pins are used as address bus lines.

10.3.2 Interrupts on the Pluto 6 board

The peripherals external to the ColdFire which require servicing via interrupts (e.g. The DUART chip, the UARTS in the FPGA) are routed via the FPGA. When a peripheral raises an interrupt the FPGA sets the three interrupt lines routed to the ColdFire to the correct value. Once this value has been held for two ColdFire clock cycles the interrupt mask register in the SIM is checked. If the interrupt level is unmasked then the interrupt is passed to the ColdFire core.
Each interrupt source - up to seven external devices, plus the eight internal ColdFire peripherals (SWT, timer 1 and 2, MBUS, UART 1 and 2, DMA channel 1 and 2) has its own interrupt control register is the SIM. Each register has three bits to set the interrupt to one of the seven levels (for external interrupts these levels are fixed and so the bits in their registers are reserved) and two bits to set interrupts at the same levels to one of four priorities.

Interrupt Control Register:

7	6	5	4	3	2	1	0
AVEC	-	-	IL2	IL1	IL0	IP1	IP0

This is useful because the internal peripherals may need to share one of the same 7 levels with an external interrupt (if they are all used), the priority level then allows one interrupt to override the servicing of the other. If the auto-vector bit is set in an ICR for an interrupt source then the processor can straight away call the handler in the vector table, the interrupt vectors start at vector 25, so the pointer to the handler function is at the interrupt level + 24. If the AVEC bit is not set the ColdFire core carries out an interrupt acknowledge cycle during which the vector number is returned by the peripheral, this vector number is then used directly to index the vector table (for internal ColdFire devices the vector number can be set in their control registers in the SIM)

On the Pluto 6 the external interrupt sources are set to auto-vector, this removes the need for the devices external to the ColdFire to provide a vector number during an interrupt acknowledge cycle, simplifying their operation. These interrupts will vector to the auto vector range of the interrupt table 25-31. Interrupt acknowledge cycles are easier to implement with the peripherals integrated onto the ColdFire, so they can use vectors in the user definable range of the vector table (64-255). The interrupt control registers on the Pluto 6 are set up as follows:

	ICR value	Purpose
External Interrupt Priority Level 1	0x80	FPGA UART interrupt (IL1 IP0)
External Interrupt Priority Level 2	0x80	External DUART interrupt (IL2 IP0)
External Interrupt Priority Level 3	0x83	Graphics card sync (IL3 IP3)
External Interrupt Priority Level 4	0x00	Unused
External Interrupt Priority Level 5	0x00	Unused
External Interrupt Priority Level 6	0x00	Unused
External Interrupt Priority Level 7	0x00	Unused
Software Watchdog Timer	0x00	Unused
Timer 1	0x96	Timer 1 (1MS) - (IL5 IP2)
Timer 2	0x88	Timer 2 (10MS) - (IL2 IP0)
MBUS (I2C)	0x00	Unused
UART 1	0x12	UART 1 - (IL4 IP2)
UART 2	0x11	UART 2 - (IL4 IP1)
DMA 0	0x00	DMA0 (disabled)
DMA 1	0x1B	DMA1 - (IL6 IP3)

As can be seen the timers are set to auto vector. Timer 2 is set up to tick at 10 ms, this shares the same interrupt level with the external interrupt level 2 which is the DUART chip. The priorities are also the same, even though using the same level and priority for two interrupt sources is warned against in the ColdFire manual. The handler in the vector table checks the interrupt pending register to see which source caused the interrupt and takes the appropriate action. Originally timer 2 had its own vector, the decision was taken to move this to share with the DUART as this timer needed a higher priority. Timer 2 drives the multiplexed lamps and they were flickering when the video (which had a higher priority interrupt) was enabled.
Also notice that the interrupt for DMA channel 0 are disabled. Channel 0 is not used because a silicon error on the 5206e means it does not work correctly.

10.3.3 Chip selects and DRAM

The Pluto 6 board is configured with the following memory map, the devices are addressed via either the four chip select lines (addresses starting 0 to 3), the DRAM controller (the DRAM), or the ColdFire internally (internal SRAM and peripheral registers). Note that the top four address lines are internal only.

```
Device                  Size       address range
Boot ROM                512Kb      0x00000000 - 0x0003FEFC
External SRAM           256Kb      0x10000000 - 0x10040000
ATA controller          16b        0x12000000 - 0x12000010
FPGA                    64Mb       0x20000000 - 0x22FFFFFF
Graphics RAM            32Mb       0x30000000 - 0x31FC0000
Graphics controller     200Kb      0x31FC0000 - 0x31FF0400
CF Internal SRAM        8Kb        0x50000000 - 0x50002000
DRAM                    1.5Mb      0x60000000 - 0x6016FEFC
CF peripheral regs      500Kb      0xF0000000 - 0xF00001F0
```

11 Appendix C – The Heber software architecture

Heber develop code for the Pluto 6 board in C using an old version of GCC (1.95) built to compile for the M68k architecture.

The Pluto 6 as sold to fruit machine developers currently follows a closed source model. Software engineers at Heber developed their own custom libraries from scratch using only the data sheets for components on the board and the schematics produced by the hardware engineers for reference. The code which handles the hardware is split into four libraries:

- Hardware – entry code, ATA, MBus, DMA, serial ports within the ColdFire, FPGA and DUART chip.
- Interface – FAT32, Heber Image Format and Windows bitmap file handling.
- Peripherals – E2ROM, communication with PIC MCU, multiplexed I/O.
- Customer peripherals – code for an example reel (stepper motor) driver. Customers may use this as a template to write their own drivers.

When new versions of these libraries have been tested and released by Heber, the modules from each are compiled into object files and then one main .a library file. These are distributed with the header files to the customer.
Heber also provides demonstration projects with source code. These come with linker scripts, the customer is then able to compile the game code they write and link it with the library files from Heber. The makefile used to build the game code by the customer has the paths to the libraries embedded in them, the code can then refer to header files of the libraries and hence use the functions provided by Heber to handle the hardware.

11.1 Software architecture

The hardware is handled by the code using instances of a Device struct for each device. When a driver is written a global Device struct for it is declared. The device struct is defined in the hardware library in the header file for the device manager module:

```
typedef     struct device
{
      BYTE        type;
      BYTE        id;
      BYTE        lines;

      BYTE        (*install)(const struct device* dev,WORD cfg,WORD base);
      void        (*shutdown)(const struct device* dev);
      void        (*restore)(const struct device* dev);

      void*       vars;
      void*       zvars;
      ERRVAR*           errvar;

      const void* config;
      const void* services;
      const void* errordata;
} DEVICE;
```

A device structure is then created for every driver which is to be used in the system. Here is an example declaration for the E2ROM driver in the e2rom module of the peripherals library:

```
const DEVICE      E2RomDevice=
{
      E2RDEVICE,
      STDE2R,
      -1,

      InstallE2Rom,
      ShutdownE2Rom,
      RestoreE2Rom,

      &E2RomVars,
      0,
      0,
      E2RomCfg,
      0,
      0,
};
```

The device struct contains pointers to functions and other structures including config. The structure config is defined in the library code, but can be populated by the game developer in the game code using an extern declaration in the file config.c of the project:

```
/* EEPROM */
const   E2RCFG    E2RomCfg[]=
{
      {
            &MbusDevice,      /* I2C MBUS Device Driver */
            NM24C04,          /* I2C Device address */
            16                /* page size in bytes */
            256               /* bank size in bytes */
            2                 /* number of banks */
            5,          /* Retry time after failed read/write*/
            1,          /* Retries */
    }
};
```

This allows the game author to configure certain parameters of the devices on the board (E.G. the number and size of pages and banks of the E2ROM they are going to put on the board) The config struct is then pointed to by the config pointer in the main Device struct.
The Device struct also contains a pointer to another struct called Services. The Services struct may contain pointers to functions which a driver can allow higher level drivers to use. For example the Mbus driver config struct contains a valid pointer to a services struct:

```
const DEVICE      MbusDevice=
{
      MBUSDEVICE,
      STDMBUS,
      -1,
      InstallMbus,
      ShutdownMbus,
      RestoreMbus,
      0,
      0,
      0,
      0,
      &MbusServices,    /* services struct */
      0,
};
```

```
static const      SERVICES MbusServices=
{
      MbusRead,
      MbusWrite,
};
```

MbusRead and MbusWrite are functions to read and write to the I2C bus connected to the ColdFire. The E2ROM driver relies on the MBus driver for its I2C communication from the ColdFire, this is why the config structure in the e2rom device contains a pointer to the MbusDEVICE structure, the e2rom driver can then call the functions in the Mbus device services struct:

```
E2RomDevice.E2RomCfg.mbusdev->MbusRead(...);
```

This is also useful for the FAT32 and ISO drivers in the interface library, which can both use the ATA driver in the hardware library to operate drives connected to the IDE bus on the board.

The driver modules contain the functions to call from game code. These are separate from the device structure instances, these must be passed in as parameters. This allows functions to be re-used for slightly varying devices, for example there is a version of Pluto 6 which can accommodate two calypso 32 graphics cards and produce dual video output. The two graphics cards have separate device structures, Calypso32Dual0Device and Calypso32Dual1Device. Both of these have separate config structures. Instead of requiring a separate set of functions to drive each one, they can both use the same functions, the device parameter tells the function which card to use e.g.:

```
/* enable double buffering on first card only */
void CalypsoEnableDoubleBuffer(Calypso32Dual0Device)
```

The device structures are directly associated with three functions however - each device struct contains pointers its install(), suspend(), and shutdown() functions. These pointers are used during the boot sequence.

11.2 The boot sequence

Execution of code on the ColdFire begins in an assembly file in the hardware library called entry.h. This code does the following:

1. sets the module and vector base addresses in the appropriate ColdFire registers
2. resets and enables the cache
3. configures the chip selects and DRAM controller so that the addresses of RAM and peripherals are at the addresses expected by the code in the libraries (this is configurable using the extern config structs in the game code config.h as above for extra flexibility)
4. sets the system integration module and interrupt control registers to respond to interrupts correctly.
5. Sets the pin assignment register, and the system watchdog vector register.
6. The code then tests where it is executing from by comparing the ENTRY: assembly label to literal hex addresses. Possible execution locations are DRAM, EPROM, external SRAM or a DRAM expansion card. If a DRAM expansion card is found, the relevant chip selects are altered and the code jumps to the start address of that memory.
7. The code checks the program identity (set in the game code) against the ID stored in the battery backed SRAM by the COFF section id_ram in the linker script. If the Ids are not equal the rest of the SRAM is cleared. Otherwise this allows data to persist in the SRAM between power cycles.
8. The EPROM checksum is also compared with a copy stored in SRAM. If these differ then the SRAM

is cleared.
9. Finally the assembly code branches to C code by calling main().

The main() function calls the function installdevices(DeviceList) from the devmgr module in the hardware library. Pointers to each device driver to load are placed in a DEVICE array in the file devices.h in the game code:

```
const INSTALLDEVICE     DeviceList[]=
{
      {&ATADevice             ,0    ,0,0},
      {&DUART0Device          ,0    ,0,0},
      {&SerialDevice          ,0    ,0,0},
      {&LampDevice            ,0    ,0,0},
      {&OutputDevice          ,0    ,0,0},
      {&StepperDevice         ,0    ,0,0},
      {&MXIDevice             ,0    ,0,0},
      {&FPUART0Device         ,0    ,0,0},
      {&FPUART1Device         ,0    ,0,0},
      {&TimerDevice           ,0    ,0,0},
      {&MbusDevice            ,0    ,0,0},
      {&PICDevice             ,0    ,0,0},
      {&E2RomDevice           ,0    ,0,0},
//    {&Calypso32Device       ,0    ,0,0},
      {0,0,0}
};
```

This array contains pointers to the device structures described above. The devmgr module can use this array from an extern declaration in it header file. It then steps through this array and calls the install() function for each device. This mechanism allows the game developers to leave out support for devices not required in their game, reducing the number of interrupt routines, and so CPU load. (E.G. If the game developer does not require graphics, she may remove the pointer to the Calypso32Device struct) The install function for a driver may install interrupt service routines into an interrupt list. These are handled by the except module in the hardware library. The following interrupt lists are defined:

```
void (*Int1msLst[])()={0,0,0,0,0,0,0,0,0,EOF_FUNCTIONS};
void (*Int10msLst[])()={0,0,0,0,0,0,0,0,0,EOF_FUNCTIONS};
void (*Int250msLst[])()={0,0,0,0,0,0,0,0,0,EOF_FUNCTIONS};
void (*IntDuartLst[])()={0,0,0,0,EOF_FUNCTIONS};

void (*IntVideoLst[])()={0,0,0,0,0,0,0,0,0,EOF_FUNCTIONS};
void (*IntFpuartLst[])()={0,0,0,0,0,0,EOF_FUNCTIONS};
void (*IntDMALst[])()={0,0,0,0,EOF_FUNCTIONS};

void (*IntDMALst[])()={0,0,0,0,EOF_FUNCTIONS};
```

When an interrupt occurs the relevant ISR is called (the vector table is set up using a list of function pointers in vectors.c) this ISR will then call the function InterruptSheduler() from the except module with the appropriate interrupt list. Any functions which are installed in that list by driver install() functions are then called in the order they were installed.

Once main() has installed all the device drivers, it then calls Game() in the game code, from which point it is up to the game developers to decide what happens.

12 Appendix D - uClinux and Linux boot sequence comparison

12.1 Power on

When a board is powered on, all of the circuitry, especially that within the CPU and volatile memory, is in a random state. In order to initialise the hardware, software has to be run to set it up, in order for that to happen, the reset signal which every CPU has must be asserted and then negated. This could be performed by a small circuit involving a few resistors and capacitors, or it may follow a complex pre-boot operation involving configuring an FPGA, such as on the Pluto 6.
Once the CPU comes out of reset, the most common approach to begin booting is to assert a global chip select signal, then jump to a physical address set in the hardware and execute the code found there. It is up to the designers of a board to ensure that the chip select line asserted at boot is connected to a memory device which can contain boot code, and that the starting address will access the correct memory. Often the initial boot address is 0x0, as is the case for the ColdFire which asserts CS0 after reset, on the Pluto 6 this is connected to a 512kb EEPROM. On the x86 however, the initial address jumped to is 0xFFFFFFF0. This maps to the ROM containing what is known as the BIOS. The concept of a BIOS is the first point at which the boot sequences of a small micro controller and the PC diverge.

12.2 X86 PC Linux kernel boot sequence

The initial code in the PC BIOS carries out a Power-On Self Test and displays a banner with information about the RAM and PCI devices found (or more recently a graphical splash screen), then identifies a bootable drive in the priority set by the user, this setting along with others and the time is stored in battery backed CMOS. The first sector (512 bytes) is copied from the drive to RAM and executed. 512 bytes isn't enough code to drive all the possible IDE hard disks and floppy drives that could contain an operating system, so the running software can call back into routines in the BIOS via interrupts. These routines can get data from a disk, or information about the hardware present. Older operating systems (e.g. MSDOS) rely on BIOS calls all the time to use the hardware. Modern OS's can't rely on the BIOS during normal operation because the calls are only available in the legacy Real Mode entered after reset, during which the CPU can only address 640Kb of memory. In order to use all of the physical memory present the CPU must switch to Protected Mode, while the code in the BIOS only runs in Real Mode.
The lack of BIOS availability in Protected Mode doesn't matter for Linux, because the decision was made not to use the BIOS calls. This was partly for portability, but also because the kernel code can often identify the hardware present more accurately itself. During boot and before entering protected mode however, the Linux Loader (LILO) and the kernel are forced to use the BIOS, in order to get the image from disk and discover available memory. Here is an excerpt from the boot code in linux/arch/i386/boot/setup.S:

```
movb  $0x88, %ah
int   $0x15
movw  %ax, (2)
```

This BIOS call is the simplest and last tried of three methods of discovering the RAM in the system. The BIOS procedure identifier (0x88) is placed in the ah register, the interrupt triggered, and after the code executed in the BIOS ROM returns, the result is moved to the ax register.

Because of the infliction of the 640Kb memory restriction in Real Mode, a multi-stage boot loader must be used to load enough code into the available RAM to manage the rest of the boot once protected mode is entered and the BIOS calls are lost. The following diagram illustrates the procedure the X86 port of Linux uses to boot itself.

The first sector loaded from disk is LILO or GRUB. This then uses the BIOS to load the rest of itself and presents a menu of OSs. Assuming Linux is chosen, the compressed image is loaded at 0x00001000, and the initial boot code at 0x00090200. The initial boot code enters protected mode, then executes the code preceding the compressed image. Now that enough memory is addressable, this code decompresses the image to 0x00100000. The entry assembly at the start of the decompressed image is executed until start_kernel() is reached.

12.3 ColdFire uClinux kernel boot sequence

Most embedded systems differ in that they don't have a true BIOS. The term BIOS is usually used in this area to refer to a boot loader in ROM, but while the PC has a set of routines to call back to in its BIOS from other code executing in RAM, smaller embedded systems such as the Pluto 6 have no standardised set of routines in ROM. When the ColdFire starts booting uClinux, there is nothing below this level but wires and solder, so all the routines required to finish booting must be in this code. This is often made simpler by placing the kernel image in addressable memory rather than a disk, an approach now spreading back to the PC with the LinuxBIOS project (the ROM containing the BIOS is overwritten with an XIP kernel). Also micro controllers such as the ColdFire have no legacy modes or memory restrictions, removing the need for multi-stage boot loaders.
The ColdFire treats the beginning contents of its boot ROM as an initial vector table, the first entry contains the initial value to place in the stack pointer register, the second is the initial program counter value. This will point to the start of the code after the vector table.

On an embedded system booting uClinux, the initial code executed in the ROM will do one of two things:

- The kernel runs directly from ROM without a boot loader, known as execute in place (XIP, as described in the chapter on differences between the normal Linux kernel and uClinux). Although this is the simplest boot sequence, the assembly preceding the call to start_kernel() must do the work of initialising all of the required hardware, including if present the DRAM controller and CPU cache. The data section of the kernel must then be copied into RAM, and space after that cleared for the BSS section. The amount of RAM must be either detected or pre-defined, and pointers to the start and end of available RAM must be placed into assembly variables for later use in C code. At entry to start_kernel() the CPU must be left in the highest privilege level (supervisor mode), with interrupts disabled. Without a boot loader, there is no way to pass command lines to the kernel, so they must be compiled in using the configuration option.
- The hardware is initialised by separate boot code, then the kernel copied from an area of ROM into RAM at the address it was linked to and executed. In this case the entry assembly of the kernel can assume the CPU has been initialised to access the required memory correctly and omit the hardware set up. The boot code could range from a small piece of assembly located in ROM preceding the kernel, to a larger standard boot loader set to automatically decompress the kernel from ROM into RAM and execute it. Examples of standard boot loaders include Uboot, Colilo and Dbug, which run on multiple systems, and some even multiple architectures. Often the more architectures a boot loader has been ported to, the more functionality it has, ranging from drivers for Ethernet drivers and a TCP/IP stack to loadable modules. Usually a prompt is presented to the user via a serial port, allowing commands to be typed. Development boards often use this as a method for developers to download programs over the serial port or Ethernet to the board and run them, including Linux kernels.

Regardless of how the uClinux kernel is executed, the main difference early in the boot sequence between it and the standard kernel is the lack of a decompresser. When the uClinux kernel code is entered, either from ROM or RAM, it assumes that it is residing in the location it will run from. This contrasts with the standard Linux kernel, which even on embedded systems such as ARM boards, can contain code to decompress itself and execute from another area of memory. UCLinux stays where it is in memory, and needs only execute the assembly in a single file (crt0_rom.S or crt0_ram.S) before jumping to start_kernel().

13 Appendix E - Variations in Graphics controller capabilities

Some of the capabilities which can vary between different garaphics controllers are:

- Supported display resolution - Some controllers support a certain set of resolutions (e.g. the VESA modes in modern VGA chips) while others support any resolution by allowing the output timings to be explicitly specified.
- Current display resolution - the size of the current display resolution output by the graphics hardware could vary, this will affect the size of the frame buffer memory. In the simplest case this is (width in pixels) * (height in pixels) * (bytes per pixel). Display resolution has become less standardised in embedded systems, where LCD panels are commonly used which may have a native resolution of only 320x240 or less.
- Virtual resolutions - Some graphics controllers can set up memory for a large virtual resolution, then display a portion of it through a window at a lower actual resolution. E.G. viewing a portion of a 1024*768 desktop at 800*600. The controller may then also support hardware panning using values placed into offset registers.
- Colour modes - a pixel value in the frame buffer memory may correspond to a particular colour directly, this is known as "Direct colour" and is commonly used for 16bit colour and above. For 8 bit colour "Indirect colour" is usually used - the pixel value in frame buffer memory is used as an index into a 256 (2^8) colour palette. The palette data is usually held in off screen video memory, because the video controller must be told where it is and perform the lookup for each pixel each time it redraws the display. Graphics controllers which use indirect colour can often display more colours than the palette can hold, the software running on the host CPU must choose an optimum set of colour values and write them into the palette location in video memory. This was the technique used by the old 256 colour VGA graphics cards on the PC, - badly written graphical MS-DOS software would write its own colour palette to the video card and not replace it afterwards, resulting in warped colours.
- Colour mapping - each pixel may be represented by a different number of bits or bytes in the frame buffer memory. Monochrome displays only required 1 bit per pixel - black or white/green. Standards then progressed as shown in the following table:

Bits Per Pixel	Number of Colours Available	Common Name(s)
1	2	Monochrome
2	4	CGA
4	16	EGA
8	256	VGA
16	65536	XGA
24	16777216	SVGA
32	4294967296	SXGA

Table 5 Colour depths at certain graphics modes

It should be noted that the names in the above table refer to specific resolutions as well as colour depth, although the amount of video RAM on current graphics cards allow a choice of resolution without compromising on colour depth. The bits for each pixel are split into 3 groups - red, green and blue. E.G. one common grouping for 16 bit colour is 555, - 5 bits for blue, 5 for green etc. The leftover bit may be ignored or perhaps used for transparency/alpha blend enable. This colour mapping style is known as "Chunky", and is the common technique used in modern hardware. Older hardware in the 80s often used "Planar"[2] graphics, a technique whereby the video memory was separated into bit planes (separate regions of video memory), and each bit in a bit plane represented a separate pixel. The bit in each bit plane was combined with the bits at the same address in the other bit planes to create a pixel value. This technique saved memory where the bit depth was not a multiple of 8, because the chunky technique always aligned the pixel values on byte boundaries (E.G. If using 4 bits per pixel, in a chunky display the other 4 bits would be wasted, in a planar display, 4 bit planes could be used in half the space). Planar displays became less popular when the bit depth went over 8 bits because of the growing number of bit planes required and the inherent complexity in the software required to drive them.

- Single or double buffering - if large amounts of the screen need to be updated at once, E.G. text scrolling on a console, or a window being dragged around on a desktop, then the data in the frame buffer memory can change whilst the monitor is refreshing. The result of this is jagged scrolling of text, and windows appearing to be "sliced" horizontally. Many graphics controllers handle this by allocating to regions of memory, one is displayed on screen whilst the other is drawn to, when a vertical synchronisation occurs the regions are swapped. This is known as double buffering, and requires the graphics controller to generate an interrupt on every vertical sync period, so that the software/OS running on the system can switch the graphics memory pointers accordingly.
- Layers - graphics controllers designed for specific purposes or markets support hardware layering. The graphics memory is partitioned into separate "layers", the contents of each are then drawn on top of each other. A certain colour code (usually the one representing black) can then be set as the transparent colour, so the contents of the layer behind can be seen, and this allows for other features in hardware, such as alpha blending layers together for a transparency effect.
- Acceleration - The concept of a frame buffer is based on bit mapped operation. Certain graphics controllers offer acceleration using vector techniques. E.G. 2D operations such as drawing a line between two specified screen co-ordinates, or a circle with a specified centre and radius. The most common 3D operation is to draw a triangular polygon with 3 dimensional co-ordinates, x, y and z. This allows the construction of 3d meshes, the z value becoming especially necessary if the graphics controller supports texture mapping, where a bitmap is read from off-screen video memory and applied to the 3D model. This area is less relevant to frame buffers, however some graphics hardware has built in text handling, or embedded fonts (E.G. VGA chips).

14 Appendix F - Features of the Cremson MB86290A

The Cremson supports 8 bit indirect and 16 bit direct colour. The 8 bit mode follows a standard approach using a 256 entry palette table – each entry contains an 18 bit colour value, 6 bits each for red, green and blue. This mode therefore allows 256 colours on screen from a selection of 262,144. The 16 bit direct mode uses the standard RGB 555 colour mapping – bit 15 toggles alpha blending on/off for the pixel, then 5 bits each are used for red, green and blue from MSB down to LSB.

14.1 Resolutions

The Cremson has a set of display parameter registers which allow direct control over the timing of the video signal. These are all set independently allowing any conceivable resolution to be set from 320x234 up to 1024x768. The registers which must be set are shown in this diagram:

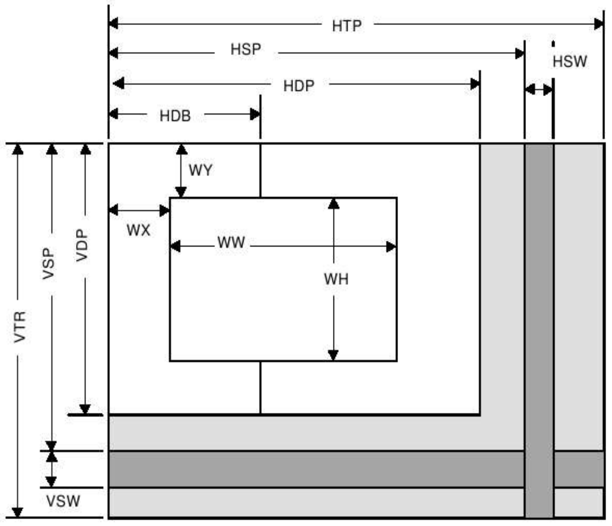


Illustration 27 Cremson display controller parameter usage (Fujitsu, 2000)

HTP	Horizontal Total Pixels
HSP	Horizontal Synchronize pulse Position
HSW	Horizontal Synchronize pulse Width
HDP	Horizontal Display Period
HDB	Horizontal Display Boundary
VTR	Vertical Total Raster
VSP	Vertical Synchronize pulse Position
VSW	Vertical Synchronize pulse Width
VDP	Vertical Display Period
WX	Window position X
WY	Window position Y
WW	Window Width
WH	Window Height

Illustration 28 Cremson display parameters (Fujitsu, 2000)

Each of the parameters in the above diagram are in pixels, but can also be thought of as multiples of the pixclock/dotclock value. The pixclock is the time 1 pixel is displayed for – this is more logical for values such as the Hsync width, where no pixels are actually on the screen. The pixclock value is set by placing values in the Display control mode register to select a clock source and a scalar value. The Pluto 6 uses Cremsons built in 200MHz clock, either as is or divided from 2-32 as the pixclock. Once the pixel clock is establiahed the rest of the values for a resolution are calculated.

14.2 Layer/frame control

The cremson supports splitting the video memory into four seperate layers, and displaying them on top of each other:

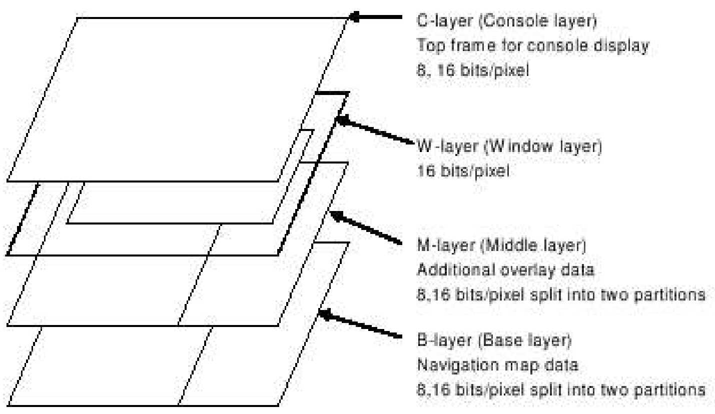


Illustration 29 Cremson display controller layer configuration (Fujitsu, 2000)

Each layer can have a different colour depth, actual size and visible window size and transparent colour (therefore showing the layer beneath). The layers can then be panned and enabled/disabled independently, and the console layer can be alpha blended with those below. Each layer has its own set of registers to set its start and end address in video memory, width and height, colour mode and whether the layer is single or double buffered.
To aid the handling of double buffering by the ColdFire (switching the video memory address to write to and flipping the frame displayed) the Cremson can assert interrupts for multiple events:

- External synchronization error
- Vertical synchronization timing detect
- Field synchronization timing detect
- Command error
- Command complete

These are enabled separately in the interrupt mask register.

14.3 2d primitives and anti-aliasing

The Cremson can draw primitive 2D shapes using vector operations, the following types are supported:

- Point
- Line
- Triangle
- Fast2DLine
- Fast2Dtriangle
- Polygon

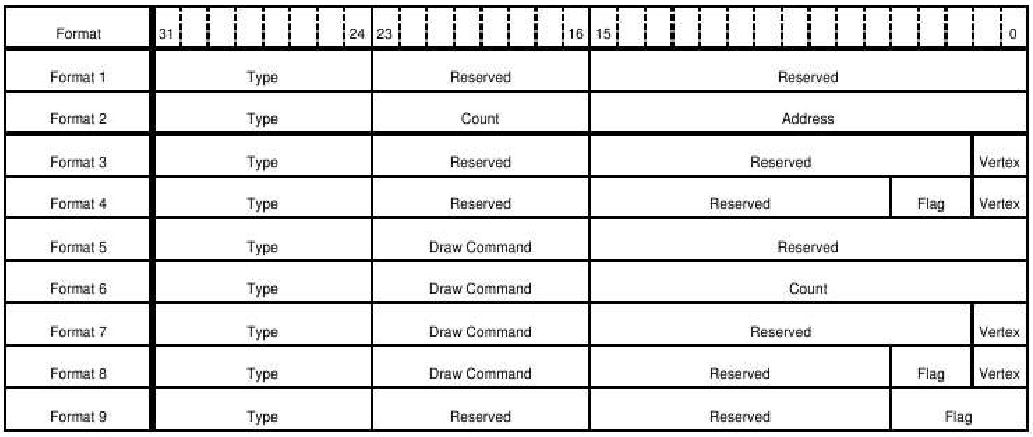
These shapes can be rendered with width (1-32 pixels) and anti-aliasing attributes.

14.4 3d polygons and perspective texture mapping

The Cremson supports texture mapping onto the primitives it can render, perspective correction is provided using z co-ordinates on the polygon points, giving primitive 3D functionality. The textures can either be stored in an on chip 8kb texture buffer which can hold a 64x64 pixel texture, or in general purpose video memory, which is slower.

14.5 Display lists

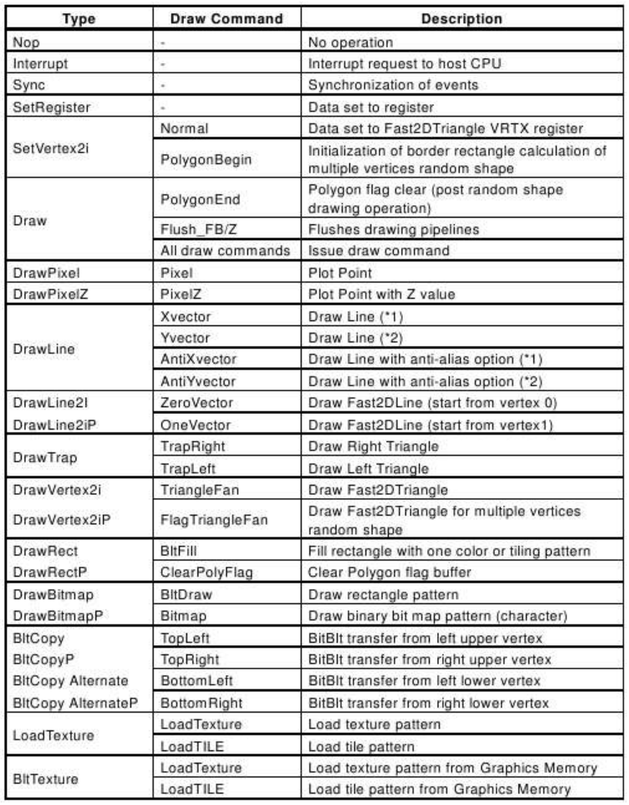
The Cremson has a FIFO buffer in which to place specially formatted commands. A collection of these commands are referred to as a display list, which the Cremson can process independently of the host CPU. The commands are executed concurrently once placed into the FIFO, which can be written to directly by the CPU, or via DMA transfer by the Cremson controller from the video memory. There are nine different formats of display list commands depending on their type:



Format	31 … 24	23 … 16	15 …		… 0
Format 1	Type	Reserved	Reserved		
Format 2	Type	Count	Address		
Format 3	Type	Reserved	Reserved		Vertex
Format 4	Type	Reserved	Reserved	Flag	Vertex
Format 5	Type	Draw Command	Reserved		
Format 6	Type	Draw Command	Count		
Format 7	Type	Draw Command	Reserved		Vertex
Format 8	Type	Draw Command	Reserved	Flag	Vertex
Format 9	Type	Reserved	Reserved	Flag	

Illustration 30 Cremson display list formats (Fujitsu, 2000)

Which format is used varies between each command, making programming display lists for the Cremson a complicated procedure. A display command isn't necessarily a draw command - the commands are:



Type	Draw Command	Description
Nop	-	No operation
Interrupt	-	Interrupt request to host CPU
Sync	-	Synchronization of events
SetRegister	-	Data set to register
SetVertex2i	Normal	Data set to Fast2DTriangle VRTX register
	PolygonBegin	Initialization of border rectangle calculation of multiple vertices random shape
Draw	PolygonEnd	Polygon flag clear (post random shape drawing operation)
	Flush_FB/Z	Flushes drawing pipelines
	All draw commands	Issue draw command
DrawPixel	Pixel	Plot Point
DrawPixelZ	PixelZ	Plot Point with Z value
DrawLine	Xvector	Draw Line (*1)
	Yvector	Draw Line (*2)
	AntiXvector	Draw Line with anti-alias option (*1)
	AntiYvector	Draw Line with anti-alias option (*2)
DrawLine2I	ZeroVector	Draw Fast2DLine (start from vertex 0)
DrawLine2iP	OneVector	Draw Fast2DLine (start from vertex1)
DrawTrap	TrapRight	Draw Right Triangle
	TrapLeft	Draw Left Triangle
DrawVertex2i	TriangleFan	Draw Fast2DTriangle
DrawVertex2iP	FlagTriangleFan	Draw Fast2DTriangle for multiple vertices random shape
DrawRect	BltFill	Fill rectangle with one color or tiling pattern
DrawRectP	ClearPolyFlag	Clear Polygon flag buffer
DrawBitmap	BltDraw	Draw rectangle pattern
DrawBitmapP	Bitmap	Draw binary bit map pattern (character)
BltCopy	TopLeft	BitBlt transfer from left upper vertex
BltCopyP	TopRight	BitBlt transfer from right upper vertex
BltCopy Alternate	BottomLeft	BitBlt transfer from left lower vertex
BltCopy AlternateP	BottomRight	BitBlt transfer from right lower vertex
LoadTexture	LoadTexture	Load texture pattern
	LoadTILE	Load tile pattern
BltTexture	LoadTexture	Load texture pattern from Graphics Memory
	LoadTILE	Load tile pattern from Graphics Memory

Illustration 31 Cremson display list comands (Fujitsu 2001)

This functionality can be used to create a set of commands to draw different frames of a sprite to a layer, setting the display list to be repeatedly loaded into the command FIFO would result in an animation without intervention by the host CPU. This technique is used in later versions of the Heber Pluto 6 API.

15 Appendix G - The multiplexed lamps and VFD drivers

15.1 The VFD driver

The VFD is controlled by three auxillary outputs from the FPGA, which are directly accessible by writing the least significant bits of three registers in the FPGA. The driver was a practice in order to get used to writing Linux drivers.

15.1.1 Design

The VFD is such a simple device that the design barely needed consideration. It is a simple character device, supporting the write function only, on the basis that if a program writes to the device, it should know what it has written and not need to read it back. When a character is written to the device file, it appears on the display. When the display is full, it wraps around.
The display is initialised in the open function. When writing characters, lower case letters are found and converted to upper case by checking the ASCII values and subtracting 32. This avoids the need for to_lower() which is not present in the kernel.
There is a connector to support another display on the board, this is supported by the driver by providing two device files with major number 120, which is in the private/development range, and the minor number 0 or 1 to choose the VFD.

15.1.2 Implementation

With the driver written and compiled into the kernel, it was used by writing to the device file:

```
/ # echo hello\ world! > /dev/vfd0
```

And the text is displayed:

Illustration 32 The VFD in action

15.2 The multiplexed lamps driver

The FPGA supports strobing through 16 rows of lamps, each row containing 32 outputs. The lamps are strobed every 1ms. The 1ms period of each row is broken down by the FPGA into 8 further periods, each lamp can be on or off for each of these sub-periods to control its brightness level.
In order for software to control this, the FPGA presents a bank of 32 long (32 bit) registers. Each

register represents an eighth of a millisecond for the strobe, each bit in each register represents a lamp.
When the 1ms strobe period is up, the software must replace the values in the 32 registers with the values for the next row of lamps, then signal to the FPGA the start of the next strobe period by writing to a control register.

15.2.1 Design

The challenge with this device was to decide how user space would send the 32 4 byte registers into the kernel for the 16 rows of lamps. Three possibilities were found:

- Use 1 device file, write a character for each of the 256 lamps containing a brightness level.
- Use 16 device files, write 32 characters into each to control the brightness values of each row separately.
- Use 256 device files, write a single character to each representing the brightness.

The first option would present a huge over head just to change the brightness of one lamp, requiring a 256 byte transfer into kernel space. The third option would involve opening too many files individually at run time, every time a file is opened, written and closed requires 3 system calls, each involving 2 context switches, meaning setting the whole block of lamps would require 256*3*2 which is 1536 context switches. This would be far to great a performance penalty on this level of system.
The second option was chosen as a compromise between the performance of a single device file and the conveniance of individual lamp control. This approach also allows an array of lamp values to be held and altered in the kernel more easily. A kernel timer was used to step through the array every millisecond and write to the FPGA registers.

15.2.2 Implementation

With the driver written, a row of lamps could be set by echoing a string of ASCII '0'-'7's ('0'=off to '7'=brightest) into the device file for the row desired.
At the first attempt the lamps only flashed slowly, this was found to be because the kernel timer function was written based on the jiffies value, which is updated in every timer interrupt. The frequency of timer interrupts is determined by the setting of HZ defined in linux/include/asm-m68knommu/param.h. This is set to 100 by default, unfortunately 100 timer interrupts per second is 10 times too slow for the FPGA.
HZ was set to 1000, and the lamps then came on steadily. However on a 40MHz system, a HZ setting of 1000 can slow the system as a whole.
The lamps driver can be tested on the command line as follows:

```
/ # echo 0123456776543210 > /dev/lamps10
/ # echo 0707070707070707 > /dev/lamps15
```

Which results in the following:

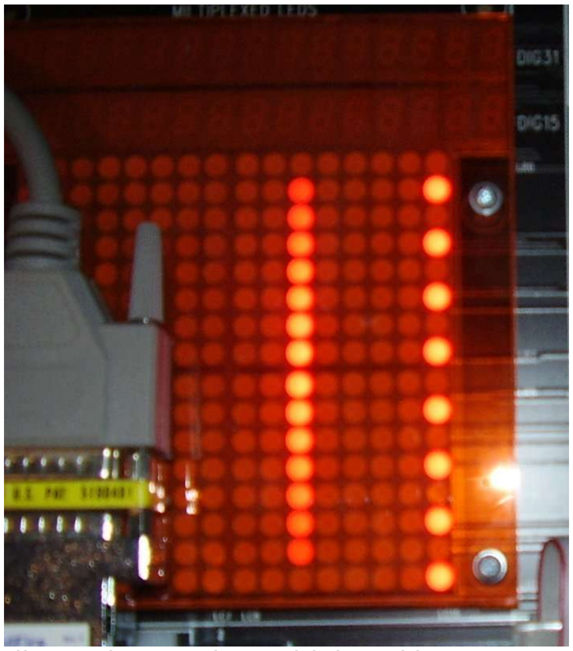


Illustration 33 The Multiplexed lamps, partially obscured by the BDM module

16 Appendix H - The CompactFlash IDE driver

16.1 Design

The uClinux distribution provides an easily adaptable IDE driver in linux/drivers/ide/legacy/uclinux.c, so a design for this driver was not required. In order to use the driver the interrupt vector number and the address of the IDE registers must be set in an array of ide_defaults structures. The IDE registers on the Pluto 6 board are accessed through the CPLD at 0x12000000. It could be seen by examining the board schematics that the interrupt line for the CompactFlash card was connected to the FPGA, but the IDE socket was not. The VHDL used to configure the standard FPGA was examined and the interrupt level value output on the 3 ColdFire interrupt lines was found to be the same as the Cremson video controller, so the vector number was 27.

16.2 Implementation

The interrupt vector and IDE register address was added to the driver, but the first run didn't work, although it printed a banner for the "Default IDE driver". Eventually it was discovered that Although IDE support had been configured in the kernel configuration, the normal IDE driver had not been chosen, so a large portion of the IDE subsystem was missing from the kernel.
The correct IDE driver would then fail to recognise the CompactFlash card present. Eventually it was realised that the chip select 0 address had been changed to make way for the MBAR, so the correct address of the IDE registers was 0xF2000000.
At the next run through, the driver recognised the card, but hang when attempting to access the /dev/hda device. A "lost interrupt" message was printed. This was found to be due to the ColdFire masking out the IDE interrupt. In order to unmask the interrupt, the driver calls the function mcf_autovector() which also sets the external interrupt to automatically run the function allocated in the vector table, rather than executing an interrupt cycle to fetch the vector number. However the driver passes the vector parameter before it is set, so the interrupt remains masked.
Having amended the driver, the kernel then printed "spurious interrupt x" when the driver was being initialised at boot (the value of x steadily incremented). It was realised this was the Cremson vertical synchronisation interrupt which was now unmasked. In order to stop the VSYNC interrupt it must be masked in the Cremson interrupt mask register. The IDE driver presents a function to do this called uclinux_irq_fixup() which is called within the IDE layer when the interrupt is installed.
Once the Cremson interrupt was blocked and the IDE signal could get through, the message "Unknown partition table" was printed every time the kernel attempted to access the CompactFlash device file. This included at boot when the partition table is read, and then when running Fdisk during numerous attempts to create a valid partition table.
It was suspected that perhaps the data bus width was incorrect, other boards using the driver had macros in #ifdef blocks to vary the data bus width between 16 and 8 bit. The macros ENABLE8/16/32 and DISABLE16/32 were created for the Pluto 6 board, these switch between an 8/16 bit data bus by toggling the setting in the Chip Select Control Register 0 (CSCR0). These macros are then called INB/W/L and OUTB/W/L macros called by the IDE subsystem.

Having created these macros, the partition table error remained. The answer was eventually found on the uClinux mailing list, the following kernel configuration setting had to be enabled:

```
Kernel config
      Filesystems
            Partition types
                  PC BIOS (MSDOS partition tables) support
```

The name of the option is a misnomer, it actually allows the kernel to recognise discs that have been low level formatted on a PC using the MBR/primary/extended scheme in place before any partitions

are added.
Once this option was enabled, the card worked flawlessly. As mentioned in the XIP section of the porting chapter, the root file system was then mounted on the card without a problem.

17 Appendix I - The working kernel output

This is the output over the serial console from the working kernel:

```
Linux version 2.4.31-uc0 (ed@.Linux version 2.4.31-uc0 (ed@) (gcc version 2.95.3 26
uClinux/COLDFIRE(m5206e)
COLDFIRE port done by Greg Ungerer, gerg@snapgear.com
Flat model support (C) 1998,1999 Kenneth Albanowski, D. Jeff Dionne
On node 0 totalpages: 4096
zone(0): 0 pages.
zone(1): 4096 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/hda1 console=tty0 console=ttyS0,9600n8
Bad boy: ColdFire Timer (at 0x000033ac) called request_irq without a dev_id!
Console: colour dummy device 80x25
Calibrating delay loop... 16.57 BogoMIPS
Memory available: 13880k/16384k RAM, 0k/0k ROM (745k kernel code, 259k data)
kmem_create: Forcing size word alignment - mm_struct
kmem_create: Forcing size word alignment - filp
Dentry cache hash table entries: 2048 (order: 2, 16384 bytes)
Inode cache hash table entries: 1024 (order: 1, 8192 bytes)
kmem_create: Forcing size word alignment - inode_cache
Mount cache hash table entries: 512 (order: 0, 4096 bytes)
kmem_create: Forcing size word alignment - bdev_cache
kmem_create: Forcing size word alignment - cdev_cache
kmem_create: Forcing size word alignment - kiobuf
Buffer cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 4096 (order: 2, 16384 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Initializing RT netlink socket
Starting kswapd
kmem_create: Forcing size word alignment - file_lock_cache
ColdFire internal UART serial driver version 1.00
ttyS0 at 0x10000140 (irq = 73) is a builtin ColdFire UART
ttyS1 at 0x10000180 (irq = 74) is a builtin ColdFire UART
VFD: Copyright (C) 2005-2006, Ed Langley (edwin_langley@hotmail.com
mpx_lamps: Copyright (C) 2005-2006, Ed Langley (edwin_langley@hotmail.com
kmem_create: Forcing size word alignment - blkdev_requests
Blkmem copyright 1998,1999 D. Jeff Dionne
Blkmem copyright 1998 Kenneth Albanowski
Blkmem 1 disk images:
0: 240EDC-240EDB [VIRTUAL 240EDC-240EDB] (RO)
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
Uniform Multi-Platform E-IDE driver Revision: 7.00beta4-2.4
ide: Assuming 50MHz system bus speed for PIO modes; override with idebus=xx
hda: probing with STATUS(0x50) instead of ALTSTATUS(0x00)
hda: SAMSUNG CF/ATA, CFA DISK drive
IDE: waiting for drives to settle...
IDE: waiting for drives to settle...
ide0 at 0xf2000000 on irq 27
hda: attached ide-disk driver.
hda: 508928 sectors (261 MB) w/0KiB Cache, CHS=994/16/32
Partition check:
 hda: hda1
Console: switching to colour frame buffer device 128x48
```

```
fb0: Fujitsu Cremson frame buffer device
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
kmem_create: Forcing size word alignment - ip_dst_cache
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 1024 bind 1024)
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
 hda: hda1
 hda: hda1
VFS: Mounted root (ext2 filesystem) readonly.
Welcome to
          ____ _  _
         /  __| ||_|
    _   _| |  | | _ ____  _   _  _  _
   | | | | |  | || |  _ \| | | |\ \/ /
   | |_| | |__| || | | | | |_| |/    \
   |  ___\____|_||_|_| |_|\____|\_/\_/
   | |
   |_|

For further information check:
http://www.uclinux.org/

/ #
```

18 Appendix J - the BDM-CHK test output

Here is the first communication with the board using the BDM tools package in Linux:

```
ed@Eds-PC:/tmp/m68k-bdm-build/test$ bdm-chk /dev/bdmcf0
BDM Check for Coldfire processors.
Device: /dev/bdmcf0
trying kernel driver: /dev/bdmcf0
trying bdm server: localhost:/dev/bdmcf0
Driver Ver : 2.11
Processor  : Coldfire
Interface  : P&E Coldfire
CSR break set, target stopped.
Debug module version is 0, (5206(e))
Target status: 0x0 -- NOT RESET, NOT HALTED, NOT STOPPED, POWER ON, CONNECTED.
Register test,    1 of    1 :
   D00 : ........................................
   D01 : ........................................
   D02 : ........................................
   D03 : ........................................
   D04 : ........................................
   D05 : ........................................
   D06 : ........................................
   D07 : ........................................
   A00 : ........................................
   A01 : ........................................
   A02 : ........................................
   A03 : ........................................
   A04 : ........................................
   A05 : ........................................
   A06 : ........................................
   A07 : ........................................
Read/Write SRAM Test, 1 loops
     1 : ........................................
Alignment SRAM Test, 1 loops
Byte Write alignment write,    1 of    1 :
................................
................................
................................
................................
................................
................................
................................
................................
 reading bytes :
................................
................................
................................
................................
................................
................................
................................
................................
 reading words :
................
................
................
```

```
................
................
................
................
................
 reading long words :
........
........
........
........
........
........
........
........
Word (16bits) Write alignment verify,    1 of    1 :
................
................
................
................
................
................
................
................
 reading bytes :
................................
................................
................................
................................
................................
................................
................................
................................
 reading words :
................
................
................
................
................
................
................
................
 reading long words :
........
........
........
........
........
........
........
........
Long Word (32bits) Write alignment verify,    1 of    1 :
........
........
........
........
........
........
```

```
........
........
 reading bytes :
................................
................................
................................
................................
................................
................................
................................
................................
 reading words :
................
................
................
................
................
................
................
................
 reading long words :
........
........
........
........
........
........
........
........
Block Write alignment verify,    1 of    1 :
 reading bytes :
................................
................................
................................
................................
................................
................................
................................
................................
 reading words :
................
................
................
................
................
................
................
................
 reading long words :
........
........
........
........
........
........
........
........
```

```
Coldfire execution test, loading code to SRAM.
Stepping code.
Stepping, pc is 0x20000000, csr = 0x01000000
Stepping, pc is 0x20000004, csr = 0x01000030
Stepping, pc is 0x2000000a, csr = 0x01000030
Stepping, pc is 0x20000010, csr = 0x01000030
Stepping, pc is 0x20000016, csr = 0x01000030
Stepping, pc is 0x2000001c, csr = 0x01000030
Stepping, pc is 0x20000022, csr = 0x01000030
Stepping, pc is 0x20000028, csr = 0x01000030
Stepping, pc is 0x2000002e, csr = 0x01000030
Stepping, pc is 0x20000034, csr = 0x01000030
Stepping, pc is 0x2000003a, csr = 0x01000030
Stepping, pc is 0x20000040, csr = 0x01000030
Stepping, pc is 0x20000046, csr = 0x01000030
Stepping, pc is 0x2000004c, csr = 0x01000030
Stepping, pc is 0x20000052, csr = 0x01000030
Stepping, pc is 0x20000058, csr = 0x01000030
Stepping, pc is 0x2000005e, csr = 0x01000030
Stepping, pc is 0x20000064, csr = 0x01000030
A0: A0A0A0A0   D0: D0D0D0D0
A1: A1A1A1A1   D1: D1D1D1D1
A2: A2A2A2A2   D2: D2D2D2D2
A3: A3A3A3A3   D3: D3D3D3D3
A4: A4A4A4A4   D4: D4D4D4D4
A5: A5A5A5A5   D5: D5D5D5D5
A6: A6A6A6A6   D6: D6D6D6D6
A7: A7A7A7A7   D7: D7D7D7D7
        RPC:2000006A
         SR:00002708
        VBR:00000074
       CACR:00000000
       ACR0:FFFB60E4
       ACR1:FFFE4064
     RAMBAR:20000001
       MBAR:10000000
        CSR:01000030
       AATR:00000005
        TDR:00000000
        PBR:00000000
       PBMR:00000000
       ABHR:00000000
       ABLR:00000000
        DBR:00000000
       DBMR:00000000
CSR halt set, target halted.
Target status: 0x0 -- NOT RESET, NOT HALTED, NOT STOPPED, POWER ON, CONNECTED.
A0: A0A0A0A0   D0: D0D0D0D0
A1: A1A1A1A1   D1: D1D1D1D1
A2: A2A2A2A2   D2: D2D2D2D2
A3: A3A3A3A3   D3: D3D3D3D3
A4: A4A4A4A4   D4: D4D4D4D4
A5: A5A5A5A5   D5: D5D5D5D5
A6: A6A6A6A6   D6: D6D6D6D6
A7: A7A7A7A7   D7: D7D7D7D7
        RPC:20000088
         SR:00002708
        VBR:00000074
       CACR:00000000
       ACR0:FFFB60E4
       ACR1:FFFE4064
     RAMBAR:20000001
       MBAR:10000000
        CSR:02000000
       AATR:00000005
        TDR:00000000
        PBR:00000000
       PBMR:00000000
       ABHR:00000000
       ABLR:00000000
        DBR:00000000
       DBMR:00000000
0FC00000: 15151515
01000000: 00000000
01001000: 0022D5F0
01001050: 0022D5F0
01001054: 63202079
```

19 Appendix K - Source code listings

19.1 crt0_ram.S

This is the kernel entry assembly to run from RAM on the Pluto 6:

```
/*****************************************************************************/

/*
 *    crt0_ram.S -- startup code for MCF5206e based Heber Pluto 6 board.
 *
 *    (C) Copyright 1999-2002, Greg Ungerer (gerg@snapgear.com)
 *
 *    1999/02/24 Modified for the 5307 processor David W. Miller
 */

/*****************************************************************************/

#include "linux/autoconf.h"
#include "asm/coldfire.h"
#include "asm/mcfsim.h"

/*****************************************************************************/

/*
 *    Heber Pluto 6 Board (MCF5206e), chip select and memory setup.
 */

/* Boot flash replaced with battery backed DRAM */
#define MEM_BASE  0x00000000
#define     MEM_SIZE    0x01000000      /* 16 MB of DRAM on expansion card */
#define     MEM_END           MEM_BASE+MEM_SIZE

#define     MEM_BUILTIN 0x50000000      /* Built in SRAM location */
/* move external SRAM (CS1) all the way up to 0xF0000000 to make way for MBAR */
#define MEM_STACK 0xF0040000  /* Use external SRAM for stack */

#define VBR_BASE  MEM_BUILTIN /* Use built in SRAM for vectors */


/*****************************************************************************/

.global     _start
.global _rambase
.global _ramvec
.global     _ramstart
.global     _ramend

/*****************************************************************************/

.data

/*
 *    Set up the usable of RAM stuff. Size of RAM is determined then
 *    an initial stack set up at the end.
 */
_rambase:
```

```
.long 0
_ramvec:
.long 0
_ramstart:
.long 0
_ramend:
.long 0

#if CONFIG_BLK_DEV_INITRD
/*
 *    Setup initial RAM disk limits of using INITRD.
 */
.extern     initrd_start
.extern initrd_end
#endif

/*****************************************************************************/

.text

/*
 *    This is the codes first entry point. This is where it all
 *    begins...
 */

_start:
      nop                          /* Filler */
      move.w      #0x2700, %sr                 /* No interrupts */



      /* set the vector table location */

      move.l      #VBR_BASE, %a7               /* Note VBR can't be read */
      movec   %a7, %VBR
      move.l      %a7, _ramvec                 /* Set up vector addr */

      move.l      #MEM_BASE, %a7               /* Ed did this - set mem base */
      move.l      %a7, _rambase                /* correctly */

      move.l      #MEM_END, %a0

      move.l      %a0, %d0               /* Mem end addr is in a0 */
      move.l      %d0, %sp               /* Set up initial stack ptr */
      move.l      %d0, _ramend                 /* Set end ram addr */

/* Set the stack ptr to the end of builtin SRAM */
/*    move.l      #MEM_STACK, %a0
      move.l      %a0, %d0
      move.l      %d0, %sp
*/

      /*
       *    Enable CPU internal cache.
       */
      move.l      #0x01000000, %d0       /* Invalidate cache cmd */
      movec %d0, %CACR             /* Invalidate cache */
      move.l      #0x80000100, %d0        /* Setup cache mask */
      movec %d0, %CACR              /* Enable cache */


#ifdef CONFIG_ROMFS_FS
      /*
       *    Move ROM filesystem above bss :-)
       */
      lea.l _sbss, %a0              /* Get start of bss */
      lea.l _ebss, %a1              /* Set up destination  */
      move.l      %a0, %a2                /* Copy of bss start */

#if CONFIG_BLK_DEV_INITRD
      move.l      %a1, %d2
      add.l #0xfff, %d2             /* Round ROMfs start to page */
      and.l #0xfffff000, %d2
      move.l      %d2, %a1                /* Save result for later */
#endif

      move.l      8(%a0), %d1             /* Get size of ROMFS */
      addq.l      #8, %d1                       /* Allow for rounding */
      and.l #0xfffffffc, %d1        /* Whole words */

      add.l %d1, %a0                /* Copy from end */
      add.l %d1, %a1                /* Copy from end */
      move.l      %a1, _ramstart                /* Set start of ram */

_copy_romfs:
      move.l      -(%a0), %d0             /* Copy dword */
      move.l      %d0, -(%a1)
      cmp.l %a0, %a2                /* Check if at end */
      bne   _copy_romfs

#else /* !CONFIG_ROMFS_FS */

      lea.l _ebss, %a1              /* Get end of bss segment */
      move.l      %a1, _ramstart                /* Set start of ram */

#endif /* !CONFIG_ROMFS_FS */

      /*
       *    Zero out the bss region.
       */
      lea.l _sbss, %a0              /* Get start of bss */
      lea.l _ebss, %a1              /* Get end of bss */
      clr.l %d0                     /* Set value */
_clear_bss:
      move.l      %d0, (%a0)+             /* Clear each word */
      cmp.l %a0, %a1                /* Check if at end */
      bne   _clear_bss

#if CONFIG_BLK_DEV_INITRD
# if CONFIG_ROMFS_FS
      /*
       *    Setup up RAMdisk info if using it.
       *    (Must do this after clearing the bss :-)
       */
      move.l      %d2, initrd_start       /* Set up start of initrd */
      add.l %d1, %d2                /* Calculate end of initrd */
      move.l      %d2, initrd_end

# else /* !CONFIG_ROMFS_FS */

      /*
       * initrd support for Coldfire currently requires romfs.
       * Generic initrd support requires a bootloader protocol to
       * tell the kernel where the image is located.
       */
      moveq.l     #0, %d2
      move.l      %d2, initrd_start
      move.l      %d2, initrd_end

# endif /* !CONFIG_ROMFS_FS */
#endif /* CONFIG_BLK_DEV_INITRD */

      /*
       *      Load the current task pointer and stack.
       */
      lea     init_task_union, %a0
      movel   %a0, _current_task
      lea     0x2000(%a0), %sp

      /*
       *    Assember start up done, start code proper.
       */
      jsr   start_kernel                 /* Start Linux kernel */

_exit:
      jmp   _exit                   /* Should never get here */

/*****************************************************************************/
```

19.2 ram.ld

This is the linker script to link the kernel to RAM on the Pluto 6:

```
MEMORY {
      ram    : ORIGIN = 0x00000000, LENGTH = 0x01000000
}

SECTIONS {

      .text : {
            _stext = . ;
            *(.text)
                  *(.text.exit)
                  *(.text.lock)
                  *(.exitcall.exit)
                  *(.rodata)
                  *(.rodata.str1.1)
            . = ALIGN(0x4) ;
                  *(.kstrtab)
            . = ALIGN(16);          /* Exception table              */
            __start___ex_table = .;
                  *(__ex_table)
            __stop___ex_table = .;

            __start___ksymtab = .;  /* Kernel symbol table          */
                  *(__ksymtab)
            __stop___ksymtab = .;
            . = ALIGN(4) ;
            _etext = . ;
      } > ram

      .data BLOCK(0x4) : {
            _sdata = . ;
            __data_start = . ;
            *(.data)
                  *(.data.exit)
            . = ALIGN(0x2000) ;
            *(.data.init_task)
            . = ALIGN(0x2000) ;
            _edata = . ;
      } > ram

      .init BLOCK(4096) : {
            __init_begin = .;
                  *(.text.init)
                  *(.data.init)
            . = ALIGN(16);
            __setup_start = .;
                  *(.setup.init)
            __setup_end = .;
            __initcall_start = .;
                  *(.initcall.init)
            . = ALIGN(4) ;
            __initcall_end = .;
            __init_end = .;
      } > ram

      .bss BLOCK(0x4) : {
            _sbss = . ;
                  *(.bss)
                  *(COMMON)
            . = ALIGN(4) ;
            _ebss = . ;
            _end = . ;
      } > ram
}
```

19.3 crt0_rom.S

This is the kernel entry assembly to run from ROM on the Pluto 6:

```
/****************************************************************************/

/*
```

```
 *    crt0_rom.S -- startup code for MCF5206e based Heber Pluto 6 board.
 *    This version is to boot and execute the kernel in place from the
 *    16 Mb RAM expansion card.
 *
 *    (C) Copyright 1999-2002, Greg Ungerer (gerg@snapgear.com)
 *
 *    1999/02/24 Modified for the 5307 processor David W. Miller
 */

/*****************************************************************************/

#include "linux/autoconf.h"
#include "asm/coldfire.h"
#include "asm/mcfsim.h"

/*****************************************************************************/

/*
 *    Heber Pluto 6 Board (MCF5206e), chip select and memory setup.
 */

/* Boot flash replaced with battery backed DRAM */
#define MEM_BASE  0x00000000
#define     MEM_SIZE    0x01000000      /* 16 MB of DRAM on expansion card */
#define     MEM_END           MEM_BASE+MEM_SIZE

#define     MEM_BUILTIN 0x50000000      /* Built in SRAM location */
/* move external SRAM (CS1) all the way up to 0xF0000000 to make way for MBAR */
#define MEM_STACK 0xF0040000  /* Use external SRAM for stack */

#define VBR_BASE  MEM_BUILTIN /* Use built in SRAM for vectors */

/* Chip select and DRAM settings */

#define     CS0_ADDR    0x00000000  /* CS0 is the RAM expansion card */
#define     CS0_MASK    0x01ff0000  /* 0x0 to 0x01000000 */
#define     CS0_CTRL    0x0000299f  /* Auto acknowledge 10 wait states,
                                16 bit data port */
#define     CS1_ADDR    0x0000f000  /* CS1 is the external SRAM */
#define     CS1_MASK    0x0fff0000  /* 0xF0000000 to 0xFFFFFFFF */
#define     CS1_CTRL    0x000000df  /* 16 bit data port */
#define     CS2_ADDR    0x00002000  /* CS2 is the FPGA */
#define     CS2_MASK    0x0fff0000  /* 0x20000000 to 0x2FFFFFFF */
#define     CS2_CTRL    0x0000005f  /* 8 bit data port */
#define     CS3_ADDR    0x00003000  /* CS3 is the video RAM and Cremson */
#define     CS3_MASK    0x0fff0000  /* 0x30000000 to 0x3FFFFFFF */
#define     CS3_CTRL    0x0000001f  /* 32 bit data port */
#define     CS4_ADDR    0x00000000  /* CS4 not connected */
#define     CS4_MASK    0x00000000
#define     CS4_CTRL    0x00000000
#define     CS5_ADDR    0x00000000  /* CS5 not connected */
#define     CS5_MASK    0x00000000
#define     CS5_CTRL    0x00000000
#define     CS6_ADDR    0x00000000  /* CS6 not connected */
#define     CS6_MASK    0x00000000
#define     CS6_CTRL    0x00000000
#define     CS7_ADDR    0x00000000  /* CS7 not connected */
```

```
#define     CS7_MASK    0x00000000
#define     CS7_CTRL    0x00000000
#define     DMC_CTRL    0x00002d4c  /* default memory control */

#define     DCRR        0x000000ff  /* Refresh period  */
/* DCTR definition:
     <15>: DAEM, 1 = Drive Multiplexed Address During External Master DRAM xfer
     <14>: EDO,  1 = EDO, 0 = Normal
     <12>: RCD,  1 = 2 clk RAS-to-CAS, 0 = 1.0 clk RAS-to-CAS
  <10:09>: RSH,  10 = 3.5 clk RAS low, 01 = 2.5 clk, 00 = 1.5 clk
  <06:05>: RP,   10 = 3.5 clk RAS Precharge, 01 = 2.5 clk, 00 = 1.5 clk
     <03>: CAS,  1 = 2.5 clk CAS assertion, 0 = 1.5 clk
     <01>: CP,   1 = 1.5 CAS clk precharge, 0 = .5 clk
     <00>: CSR,  1 = 2.0 clk CAS before RAS setup, 0 = 1.0 clk
*/
#define     DCTR        0x0000566B  /* DRAM timing, use EDO CAS timing,
                                 2 CLKs before CAS assert,
                                 RAS negate 3.5 CLK after CAS assert,
                                 RAS precharge 3.5 CLKs,
                                 CAS asserted for 2 CLKs,
                                 CAS negated for 2 CLKs,
                                 CAS asserts 2 CLKs before assertion of RAS */
#define     DCAR0       0x00006000  /* DRAM0 address */
#define     DCMR0       0x001E0000  /* 0x60000000 tp 0x601EFFFF */
#define     DCCR0       0x00000093  /* 16 bit data port, 1Kb page size,
                                 normal page mode, RD and WR enable */
#define     DCAR1       0x00007000  /* DRAM1 address (not present) */
#define     DCMR1       0x001E0000  /* 0x70000000 to 0x701EFFFF */
#define     DCCR1       0x00000013  /* 32 bit data port, 1Kb page size,
                                 normal page mode, RD and WR enable */

#define PAR             0x0000FFFC      /* Pin assignment register
                                 A[27:24] as address lines
                                 BDM signals on,
                                 use the 3 IRQ lines as encoded level,
                                 UART2 RTS and DMA signals enabled */
/******************************************************************************/

.global     _start
.global _rambase
.global _ramvec
.global     _ramstart
.global     _ramend

/******************************************************************************/

.data

/*
 *    Set up the usable of RAM stuff. Size of RAM is determined then
 *    an initial stack set up at the end.
 */
_rambase:
.long 0
_ramvec:
.long 0
_ramstart:
```

```
.long 0
_ramend:
.long 0

/*********************************************************************************/

.text

_vectors:

.long 0x00000000, _start
.long _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault

.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault

.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault

.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault
.long _fault, _fault, _fault, _fault, _fault, _fault, _fault, _fault

/*********************************************************************************/


/*
 *    This is the codes first entry point. This is where it all
 *    begins...
 */

_start:
      nop                           /* Filler */
```

```
move.w      #0x2700, %sr                  /* No interrupts */

move.l      #MCF_MBAR+1, %a0        /* Set I/O base addr */
movec %a0, %MBAR             /* Note: bit 0 is Validate */
move.l      #MEM_BUILTIN+1,%a0            /* Set SRAM base addr */
movec %a0, %RAMBAR0                 /* Note: bit 0 is Validate */

move.l      #MCF_MBAR, %a0                /* Get I/O base addr */

/* ----------------------- CS1 ----------------------- */
move.w      #CS1_ADDR, %d0                /* CS1 address */
move.w      %d0, MCFSIM_CSAR1(%a0)        /* CS1 address */
move.l      #CS1_MASK, %d0                /* CS1 mask */
move.l      %d0, MCFSIM_CSMR1(%a0)        /* CS1 mask */
move.w      #CS1_CTRL, %d0                /* CS1 control */
move.w      %d0, MCFSIM_CSCR1(%a0)        /* CS1 control */

/* ----------------------- CS2 ----------------------- */
move.w      #CS2_ADDR, %d0                /* CS2 address */
move.w      %d0, MCFSIM_CSAR2(%a0)        /* CS2 address */
move.l      #CS2_MASK, %d0                /* CS2 mask */
move.l      %d0, MCFSIM_CSMR2(%a0)        /* CS2 mask */
move.w      #CS2_CTRL, %d0                /* CS2 control */
move.w      %d0, MCFSIM_CSCR2(%a0)        /* CS2 control */

/* ----------------------- CS3 ----------------------- */
move.w      #CS3_ADDR, %d0                /* CS3 address */
move.w      %d0, MCFSIM_CSAR3(%a0)        /* CS3 address */
move.l      #CS3_MASK, %d0                /* CS3 mask */
move.l      %d0, MCFSIM_CSMR3(%a0)        /* CS3 mask */
move.w      #CS3_CTRL, %d0                /* CS3 control */
move.w      %d0, MCFSIM_CSCR3(%a0)        /* CS3 control */

/* ----------------------- CS4 ----------------------- */
move.w      #CS4_ADDR, %d0                /* CS4 address */
move.w      %d0, MCFSIM_CSAR4(%a0)        /* CS4 address */
move.l      #CS4_MASK, %d0                /* CS4 mask */
move.l      %d0, MCFSIM_CSMR4(%a0)        /* CS4 mask */
move.w      #CS4_CTRL, %d0                /* CS4 control */
move.w      %d0, MCFSIM_CSCR4(%a0)        /* CS4 control */

/* ----------------------- CS5 ----------------------- */
move.w      #CS5_ADDR, %d0                /* CS5 address */
move.w      %d0, MCFSIM_CSAR5(%a0)        /* CS5 address */
move.l      #CS5_MASK, %d0                /* CS5 mask */
move.l      %d0, MCFSIM_CSMR5(%a0)        /* CS5 mask */
move.w      #CS5_CTRL, %d0                /* CS5 control */
move.w      %d0, MCFSIM_CSCR5(%a0)        /* CS5 control */

/* ----------------------- CS6 ----------------------- */
move.w      #CS6_ADDR, %d0                /* CS6 address */
move.w      %d0, MCFSIM_CSAR6(%a0)        /* CS6 address */
move.l      #CS6_MASK, %d0                /* CS6 mask */
move.l      %d0, MCFSIM_CSMR6(%a0)        /* CS6 mask */
move.w      #CS6_CTRL, %d0                /* CS6 control */
move.w      %d0, MCFSIM_CSCR6(%a0)        /* CS6 control */
```

```
/* ---------------------- CS7 ---------------------- */
move.w      #CS7_ADDR, %d0                /* CS7 address */
move.w      %d0, MCFSIM_CSAR7(%a0)        /* CS7 address */
move.l      #CS7_MASK, %d0                /* CS7 mask */
move.l      %d0, MCFSIM_CSMR7(%a0)        /* CS7 mask */
move.w      #CS7_CTRL, %d0                /* CS7 control */
move.w      %d0, MCFSIM_CSCR7(%a0)        /* CS7 control */

/* -------------------- Default -------------------- */
move.w      #DMC_CTRL, %d0                /* Default control */
move.w      %d0, MCFSIM_DMCR(%a0)         /* Default control */

/* ---------------------- DRAM ----------------------- */
move.w      #DCRR, %d0              /* Refresh period */
move.w      %d0, MCFSIM_DCRR(%a0)         /* Refresh period */
move.w      #DCTR, %d0              /* Timing address */
move.w      %d0, MCFSIM_DCTR(%a0)         /* Timing address */
move.w      #DCAR0, %d0             /* DRAM0 base address */
move.w      %d0, MCFSIM_DCAR0(%a0)        /* DRAM0 base address */
move.l      #DCMR0, %d0             /* DRAM0 mask */
move.l      %d0, MCFSIM_DCMR0(%a0)        /* DRAM0 mask */
move.b      #DCCR0, %d0             /* DRAM0 control */
move.b      %d0, MCFSIM_DCCR0(%a0)        /* DRAM0 control */
move.w      #DCAR1, %d0             /* DRAM1 base address */
move.w      %d0, MCFSIM_DCAR1(%a0)        /* DRAM1 base address */
move.l      #DCMR1, %d0             /* DRAM1 mask */
move.l      %d0, MCFSIM_DCMR1(%a0)        /* DRAM1 mask */
move.b      #DCCR1, %d0             /* DRAM1 control */
move.b      %d0, MCFSIM_DCCR1(%a0)        /* DRAM1 control */

/* ------------- Pin Assignment Register ------------- */
move.w      #PAR, %d0               /* Pin assignment */
move.w      %d0, MCFSIM_PAR(%a0)          /* Pin assignment */

/*
 * ChipSelect 0 - ROM cs
 *
 * ChipSelect 0 is the global chip select coming out of system reset.
 * CS0 is asserted for every access until CSMR0 is written.  Therefore,
 * the entire ChipSelect must be properly set prior to asserting
 * CSCR0_V.
 */
move.w      #CS0_ADDR, %d0                /* CS0 address */
move.w      %d0, MCFSIM_CSAR0(%a0)        /* CS0 address */
move.l      #CS0_MASK, %d0                /* CS0 mask */
move.l      %d0, MCFSIM_CSMR0(%a0)        /* CS0 mask */
move.w      #CS0_CTRL, %d0                /* CS0 control */
move.w      %d0, MCFSIM_CSCR0(%a0)        /* CS0 control */


/* set the vector table location */

move.l      #VBR_BASE, %a7                /* Note VBR can't be read */
movec   %a7, %VBR


/*
```

```
	 *	Enable CPU internal cache.
	 */
	move.l	#0x01000000, %d0	/* Invalidate cache cmd */
	movec	%d0, %CACR		/* Invalidate cache */
	move.l	#0x80000100, %d0	/* Setup cache mask */
	movec	%d0, %CACR		/* Enable cache */

	/*
	 *	Copy initialized data region to RAM.
	 */
	lea.l	_sdata, %a0		/* Get start of data */
	lea.l	__init_end, %a1		/* Get end of data+init */
	lea.l	_etext, %a2		/* Get end of text */
_data_copy:
	move.l	(%a2)+, %d0		/* Get next word */
	move.l	%d0, (%a0)+		/* Copy to RAM */
	cmp.l	%a0, %a1		/* Check if at end */
	bne	_data_copy

	/*
	 *	Zero out the bss region.
	 */
	lea.l	_sbss, %a0		/* Get start of bss */
	lea.l	_ebss, %a1		/* Get end of bss */
	clr.l	%d0			/* Set value */
_clear_bss:
	move.l	%d0, (%a0)+		/* Clear each word */
	cmp.l	%a0, %a1		/* Check if at end */
	bne	_clear_bss


	move.l	#VBR_BASE, %a7
	move.l	%a7, _ramvec			/* Set up vector addr */

	move.l	#MEM_BASE, %a7			/* Ed did this - set mem base */
	move.l	%a7, _rambase			/* correctly */

	lea.l	_ebss, %a1		/* Get end of bss segment */
	move.l	%a1, _ramstart			/* Set start of ram */

	move.l	#MEM_END, %a0
	move.l	%a0, %d0		/* Mem end addr is in a0 */
	move.l	%d0, %sp		/* Set up initial stack ptr */
	move.l	%d0, _ramend			/* Set end ram addr */



	/*
	 *	Load the current task pointer and stack.
	 */
	lea	init_task_union, %a0
	movel	%a0, _current_task
	lea	0x2000(%a0), %sp

	/*
	 *	Assember start up done, start code proper.
	 */
```

```
//    halt
      jsr   start_kernel                  /* Start Linux kernel */

_fault:
_exit:
      jmp   _exit                   /* Should never get here */

/******************************************************************************/
```

19.4 rom.ld

This is the linker script to link the kernel to ROM on the Pluto 6:

```
MEMORY {
      flash  : ORIGIN = 0x00000000, LENGTH = 0x00200000
      ram    : ORIGIN = 0x00200000, LENGTH = 0x00E00000
}

SECTIONS {

      .text : {
            _stext = . ;
            *(.text)
            *(.text.exit)
            *(.text.lock)
            *(.exitcall.exit)
            *(.rodata)
            *(.rodata.str1.1)
            . = ALIGN(0x4) ;
            *(.kstrtab)
            . = ALIGN(16);          /* Exception table              */
            __start___ex_table = .;
            *(__ex_table)
            __stop___ex_table = .;

            __start___ksymtab = .;  /* Kernel symbol table          */
            *(__ksymtab)
            __stop___ksymtab = .;
            . = ALIGN(4) ;
            _etext = . ;
      } > flash

      .data : AT (ADDR(.text) + SIZEOF(.text)) {
            _sdata = . ;
            __data_start = . ;
            *(.data)
            *(.data.exit)
            . = ALIGN(0x2000) ;
            *(.data.init_task)
            . = ALIGN(0x2000) ;
            _edata = . ;
      } > ram

      .init BLOCK(4096) : AT (ADDR(.text) + SIZEOF(.text) + SIZEOF(.data)) {
            __init_begin = .;
            *(.text.init)
            *(.data.init)
            . = ALIGN(16);
            __setup_start = .;
            *(.setup.init)
            __setup_end = .;
            __initcall_start = .;
            *(.initcall.init)
            . = ALIGN(4) ;
            __initcall_end = .;
            __init_end = .;
      } > ram

      .bss : AT (ADDR(.text) + SIZEOF(.text) + SIZEOF(.data) + SIZEOF(.init)) {
            _sbss = . ;
            *(.bss)
            *(COMMON)
            . = ALIGN(4) ;
            _ebss = . ;
            _end = . ;
      } > ram
}
```

19.5 gdbinit

This is the script to initialise the Pluto 6 in GDB:

```
#
# GDB Init script for the Coldfire 5206e processor on the Pluto 6.
# The MBAR has been moved to 0x10000000 and the external SRAM (CS1)
# moved to 0xF0000000, to fit the uCLinux defaults more easily.
#
# Except above this version has memory map set as per Heber code.
# However the CS0 control register is set to 16 bit port size
# to handle the 16Mb RAM expansion card.

#
# Commands to increase readability of this script.
# These command write 8, 16 and 32 values.
#
define set8
  set *((unsigned char*) $arg0) = $arg1
end

document set8
Set 8bit memory: set8 ADDRESS VALUE
end

define set16
  set *((unsigned short*) $arg0) = $arg1
end

document set16
Set 16bit memory: set16 ADDRESS VALUE
end
```

```
define set32
  set *((unsigned long*) $arg0) = $arg1
end

document set32
Set 32bit memory: set32 ADDRESS VALUE
end


#
# Create pointers to the addresses of registers to initialise,
# this is to ease understanding of the rest of this script.
#
define addresses
  printf "Setting Register Addresses ... "

# Memory Base Adress
#  bit 0 must be set for valid flag

  set $mbar  = 0x10000001
#  set $mbar  = 0xF0000001

# SIM and Interrupt control registers

  set $simr  = $mbar - 1 + 0x003
  set $icr1  = $mbar - 1 + 0x014
  set $icr2  = $mbar - 1 + 0x015
  set $icr3  = $mbar - 1 + 0x016
  set $icr4  = $mbar - 1 + 0x017
  set $icr5  = $mbar - 1 + 0x018
  set $icr6  = $mbar - 1 + 0x019
  set $icr7  = $mbar - 1 + 0x01A
  set $icr8  = $mbar - 1 + 0x01B
  set $icr9  = $mbar - 1 + 0x01C
  set $icr10 = $mbar - 1 + 0x01D
  set $icr11 = $mbar - 1 + 0x01E
  set $icr12 = $mbar - 1 + 0x01F
  set $icr13 = $mbar - 1 + 0x020
  set $imr   = $mbar - 1 + 0x036
  set $ipr   = $mbar - 1 + 0x03A
  set $rsr   = $mbar - 1 + 0x040
  set $sypcr = $mbar - 1 + 0x041
  set $swivr = $mbar - 1 + 0x042
  set $swsr  = $mbar - 1 + 0x043
  set $par   = $mbar - 1 + 0x0CA

  # bus master arbitration control (MARB)
  set $marb  = $mbar - 1 + 0x007

# Chip Select registers

  set $csar0 = $mbar - 1 + 0x064
  set $csmr0 = $mbar - 1 + 0x068
  set $cscr0 = $mbar - 1 + 0x06E
  set $csar1 = $mbar - 1 + 0x070
  set $csmr1 = $mbar - 1 + 0x074
  set $cscr1 = $mbar - 1 + 0x07A
  set $csar2 = $mbar - 1 + 0x07C
  set $csmr2 = $mbar - 1 + 0x080
  set $cscr2 = $mbar - 1 + 0x086
  set $csar3 = $mbar - 1 + 0x088
  set $csmr3 = $mbar - 1 + 0x08C
  set $cscr3 = $mbar - 1 + 0x092

  # Default Memory Control register
  set $dmcr  = $mbar - 1 + 0x0C6

# SDRAM Control registers

  set $dcrr  = $mbar - 1 + 0x046
  set $dctr  = $mbar - 1 + 0x04A
  set $dcar0 = $mbar - 1 + 0x04C
  set $dcmr0 = $mbar - 1 + 0x050
  set $dccr0 = $mbar - 1 + 0x057
  set $dcar1 = $mbar - 1 + 0x058
  set $dcmr1 = $mbar - 1 + 0x05C
  set $dccr1 = $mbar - 1 + 0x063

# Timer Control resgisters?

  set $tmr1  = $mbar - 1 + 0x100
  set $trr1  = $mbar - 1 + 0x104
  set $tcr1  = $mbar - 1 + 0x108
  set $tcn1  = $mbar - 1 + 0x10C
  set $ter1  = $mbar - 1 + 0x111
  set $tmr2  = $mbar - 1 + 0x120
  set $trr2  = $mbar - 1 + 0x124
  set $tcr2  = $mbar - 1 + 0x128
  set $tcn2  = $mbar - 1 + 0x12C
  set $ter2  = $mbar - 1 + 0x131

# 5206e Internal RAM

  set $rambar = 0x50000000

  printf "done\n"
end

#
# Setup the Chip selects on the board
#
define setup-cs
  printf "Chip Select setup ... "
  set16 $csar3 0x3000
  set32 $csmr3 0x0FFF0000
  set16 $cscr3 0x001F
  set16 $csar2 0x2000
  set32 $csmr2 0x0FFF0000
  set16 $cscr2 0x005F
  set16 $csar1 0xF000
  set32 $csmr1 0x0FFF0000
  set16 $cscr1 0x00DF
  set16 $csar0 0x0000
```

```
# set32 $csmr0 0x000F0000
  set32 $csmr0 0x01FF0000
# 16 bit port on CS0
# set16 $cscr0 0x295F
  set16 $cscr0 0x299F
  printf "done\n"
end

#
# Setup the SDRAM
#
define setup-sdram
  printf "SDRAM setup ... "

  set16 $dmcr  0x2D4C
  set16 $par   0xFFFC

  set16 $dcrr  0x00FF
  set16 $dctr  0x566B
  set16 $dcar0 0x6000
  set32 $dcmr0 0x001E0000
  set8  $dccr0 0x93
  set16 $dcar1 0x7000
  set32 $dcmr1 0x001E0000
  set8  $dccr1 0x13

  printf "done\n"
end

#
# Setup the System Integration Module
#
define setup-sim
  printf "SIM setup ... "

  printf "done\n"
end

#
# Load the kernel into target RAM and symbol table into GDB
#
define load-image
  printf "Loading Image ... "

  load image.elf
  symbol-file image.elf
  set $pc = _start

  printf "done\n"
end


#
# Run the above commands
#
#target bdm /dev/bdmcf0
# altered for GDB 5 compatibility
```

```
target bdm localhost:/dev/bdmcf0
addresses
setup-cs
setup-sdram
load-image
```

19.6 vfd.c

This is the VFD character driver:

```
/*****************************************************************************/

/*
 *    vfd.c -- simple driver for vacuum flourescent displays
 *
 *    (C) Copyright 2005-2006, Ed Langley (edwin_langley@hotmail.com)
 */

/*****************************************************************************/

#include <linux/config.h>
#include <linux/version.h>
#include <linux/types.h>
#include <linux/sched.h>
#include <linux/timer.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/delay.h>
#include <linux/slab.h>
#include <asm/param.h>
#include <asm/uaccess.h>

/*****************************************************************************/

/*
 *    Define driver major number.
 */
#define     VFD_MAJOR   120

#if LINUX_VERSION_CODE < 0x020100
#define GET_USER(a,b)   a = get_user(b)
#else
#define GET_USER(a,b)   get_user(a,b)
#endif

/*****************************************************************************/

/*
 *     Define VFD addresses
 */

/*****************************************************************************/
#if defined(CONFIG_PLUTO6)
```

```
/******************************************************************************/

#include "pluto6_fpga.h"

#define VFD_BYTE_DELAY     50 /* microseconds */
#define VFD_RESET_DELAY    500

#define VFD_OP_LOW         1
#define VFD_OP_HIGH        0

#define VFD_NUM_CHARS      16

/*
 *      VFD access functions
 */

int vfd_putchar(unsigned char vfd_num, unsigned char c)
{
  volatile unsigned char *clock, *data;
  int count;

  switch(vfd_num)
    {
    case 0:
      clock = (char*)VFD0_CLOCK;
      data = (char*)VFD0_DATA;
      break;
    case 1:
      clock = (char*)VFD1_CLOCK;
      data = (char*)VFD1_DATA;
      break;
    default:
      return -1;
    }
  /*
  *clock = 1;
  *data = 1;
  */
  /* clock in the data MSB down to LSB */
  for(count = 0; count < 8; count++)
    {
      if(c & 0x80)
      {
        *data = VFD_OP_HIGH;
      }
      else
      {
        *data = VFD_OP_LOW;
      }
      *clock = VFD_OP_HIGH;
      *clock = VFD_OP_LOW;
      c <<= 1;
    }
  udelay(VFD_BYTE_DELAY);
  return(0);
}
```

```
int vfd_reset(unsigned char vfd_num)
{
  volatile unsigned char *clock, *data, *reset;
  int i;

  switch(vfd_num)
    {
    case 0:
      clock = (char*)VFD0_CLOCK;
      data = (char*)VFD0_DATA;
      reset = (char*)VFD0_RESET;
      break;
    case 1:
      clock = (char*)VFD1_CLOCK;
      data = (char*)VFD1_DATA;
      reset = (char*)VFD1_RESET;
      break;
    default:
      return -1;
    }
  *reset = VFD_OP_LOW;
  *clock = VFD_OP_LOW;
  udelay(VFD_RESET_DELAY);

  *reset = VFD_OP_HIGH;
  udelay(VFD_RESET_DELAY);

  /* need to write the following before the
     display will print anything */
  vfd_putchar(vfd_num, 0xC0);
  udelay(VFD_BYTE_DELAY);
  udelay(VFD_BYTE_DELAY);
  vfd_putchar(vfd_num, 0xAF);
  udelay(VFD_BYTE_DELAY);
  udelay(VFD_BYTE_DELAY);
  vfd_putchar(vfd_num, 0xFF);
  udelay(VFD_BYTE_DELAY);
  udelay(VFD_BYTE_DELAY);

  /* clear the display */
  for(i = 0; i < VFD_NUM_CHARS; i++)
    {
      vfd_putchar(vfd_num, ' ');
    }
  return 0;
}

/******************************************************************************/
#else /* UNKNOWN HARDWARE */
/******************************************************************************/

#error "VFD: I don't know what hardware addresses to use?"

#endif
/******************************************************************************/

int vfd_open(struct inode *inode, struct file *filp)
```

```
{
#if VFD_DEBUG
  printk("vfd_open()\n");
#endif
  if(vfd_reset(MINOR(inode->i_rdev)))
    {
#if VFD_DEBUG
      printk("Invalid VFD number\n");
#endif
      return(-EFAULT);
    }
  return(0);
}

/*******************************************************************************/

int vfd_release(struct inode *inode, struct file *filp)
{
#if VFD_DEBUG
  printk("vfd_close()\n");
#endif
  return 0;
}

/*******************************************************************************/

ssize_t vfd_write(struct file *filp, const char *buf, size_t count, loff_t *ppos)
{
  int num_written;
  char *kern_buf;
  ssize_t retval;

#if VFD_DEBUG
  printk("vfd_write(buf=%x,count=%d)\n", (int) buf, count);
#endif

  kern_buf = (char*)kmalloc(count, GFP_KERNEL);
  if(copy_from_user(kern_buf, buf, count))
    {
      retval = -EFAULT;
      goto out;
    }


  for(num_written = 0; num_written <= count; num_written++)
    {

      /* VFD controller displays printable ASCII characters including
       upper case letters only, others come up as garbage */
      if((*kern_buf >= 32) && (*kern_buf <= 126))
      {
        /* change lower case to upper case */
        if((*kern_buf >= 97) && (*kern_buf <= 122)) (*kern_buf) -= 32;

        if(vfd_putchar(MINOR(filp->f_dentry->d_inode->i_rdev), *kern_buf))
          {
#if VFD_DEBUG
```

```
            printk("Invalid VFD number\n");
#endif
            retval = -EFAULT;
            goto out;
          }
        kern_buf++;
      }
    }
  /* no feedback from vfd so either fault or fully written */
  retval = count;
 out:
  kfree(kern_buf);
  return(retval);
}

/*****************************************************************************/

ssize_t vfd_read(struct file *filp, const char *buf, size_t count, loff_t *ppos)
{
  return 0;
}

int vfd_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned
long arg)
{
  int retval = 0;

  switch (cmd)
    {
    case 1:
#if VFD_DEBUG
      printk(KERN_INFO "vfd:  resetting display\n");
#endif
      vfd_reset(MINOR(inode->i_rdev));
      break;
    default:
      retval = -EINVAL;
      break;
    }

  return(retval);
}

/*****************************************************************************/

/*
 *    Exported file operations structure for driver...
 */

struct file_operations  vfd_fops = {
  read:         vfd_read,       /* dummy read */
  write:    vfd_write,  /* write */
  ioctl:    vfd_ioctl,  /* ioctl */
  open:           vfd_open,   /* open */
  release:  vfd_release,      /* release */
};
```

```
/****************************************************************************/

static int __init vfd_init(void)
{
  /* Register vfd as character device */
  if ((register_chrdev(VFD_MAJOR, "vfd", &vfd_fops)) < 0)
    {
      printk(KERN_WARNING "VFD: can't get major %d\n",
           VFD_MAJOR);
      return (-EBUSY);
    }

  printk("VFD: Copyright (C) 2005-2006, Ed Langley "
       "(edwin_langley@hotmail.com\n");

  return 0;
}

module_init(vfd_init);

/****************************************************************************/
```

19.7 mpxlamps.c

This is the multiplexed lamps driver:

```
/****************************************************************************/

/*
 *    mpxlamps.c -- Driver for multiplexed lamps on the Pluto 6 Board
 *                    No locking required because only writing data.
 *    (C) Copyright 2005-2006, Ed Langley (edwin_langley@hotmail.com)
 */

/****************************************************************************/

#include <linux/config.h>
#include <linux/version.h>
#include <linux/types.h>
#include <linux/sched.h>
#include <linux/timer.h>
#include <linux/kernel.h>
#include <linux/init.h>
//#include <linux/fs.h>
//#include <linux/mm.h>
#include <linux/delay.h>
#include <linux/slab.h>
#include <asm/param.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/ioport.h>
#include <linux/module.h>
/****************************************************************************/

/*
 *    Define driver major number.
 */
#define     LAMP_MAJOR  121

/****************************************************************************/

/*
 *     Define MPX addresses
 */

/****************************************************************************/
#if defined(CONFIG_PLUTO6)
/****************************************************************************/

#include "pluto6_fpga.h"

#define NUM_LAMP_STROBES   16
#define NUM_LAMP_PERIODS   8
#define NUM_LAMPS_PER_STROBE 32
//#define MPX_DATA_NBYTES 60

/****************************************************************************/
#else /* UNKNOWN HARDWARE */
/****************************************************************************/

#error "mpx_lamps: I don't know what hardware addresses to use?"

#endif
/****************************************************************************/

#define ASCII_MODE 0
#define RAW_MODE   1

struct mpx_lamp_dev
{
  /* pointers to I/O set up with ioremap for portability */
  unsigned long *control_ptr;
  int *data_ptr;

  char shutting_down;

  /* write ascii chars or values 0-8? */
  //  char mode;

  /* 2D array to hold data ready to dump to FPGA */
  int data[NUM_LAMP_STROBES][NUM_LAMP_PERIODS];

  struct timer_list strobe_timer;
};

struct mpx_lamp_dev *lamp_dev_ptr;



/* write data for 1 lamp strobe to FPGA every 1ms */

void output_strobe(unsigned long fpga_strobe_idx)
{
  //int brightness;
  //int *data_ptr = lamp_dev_ptr->data_ptr;

  /*  for(brightness = 0; brightness < NUM_LAMP_PERIODS; brightness++)
    {
      writel(lamp_dev_ptr->data[fpga_strobe_idx][brightness], data_ptr++);
    }
  */

  memcpy_toio(lamp_dev_ptr->data_ptr, lamp_dev_ptr->data[fpga_strobe_idx],
            NUM_LAMP_PERIODS * 4);

  writeb(fpga_strobe_idx, lamp_dev_ptr->control_ptr);

  if(++fpga_strobe_idx >= NUM_LAMP_STROBES)
    {
      fpga_strobe_idx = 0;
    }

  if(!lamp_dev_ptr->shutting_down)
    {
      lamp_dev_ptr->strobe_timer.data = fpga_strobe_idx;
      //      lamp_dev_ptr->strobe_timer.function = output_strobe;
      lamp_dev_ptr->strobe_timer.expires = jiffies + HZ/1000;
      add_timer(&lamp_dev_ptr->strobe_timer);
    }

}


/*****************************************************************************/

ssize_t lamp_write(struct file *filp, const char *buf, size_t count, loff_t *ppos)
{
  char *kern_buf;
  ssize_t retval;
  int lamp_bitmask;
  int lamp_idx, strobe_idx;
  char brightness, offset;

#if LAMP_DEBUG
  printk("lamp_write(buf=%x,count=%d)\n", (int) buf, count);
#endif

  if( (kern_buf = (char*)kmalloc(count, GFP_KERNEL)) == NULL)
    {
      printk(KERN_WARNING "mpx_lamps : lamp_write : can't alloc kern_buf\n");
      retval = -ENOMEM;
      goto out;
    }
  if(copy_from_user(kern_buf, buf, count))
    {
      retval = -EFAULT;
      goto out;
    }

  strobe_idx = MINOR(filp->f_dentry->d_inode->i_rdev);
  if( (strobe_idx < 0) || (strobe_idx >= NUM_LAMP_STROBES) )
    {
      retval = -EFAULT;
      goto out;
    }

  /* clear the strobe of previous values */
  for(brightness = 0; brightness < NUM_LAMP_PERIODS; brightness++)
    {
      lamp_dev_ptr->data[strobe_idx][brightness] = 0;
    }

  /* set offset depending on whether accepting ascii chars or real
     values */
  offset = (filp->private_data == ASCII_MODE) ? '0' : 0;

  /* fill the mpx data array based on brightness values passed */
  for(brightness = 0; brightness < NUM_LAMP_PERIODS; brightness++)
    {
      lamp_bitmask = 1;
      for(lamp_idx = 0; (lamp_idx < count) && (lamp_idx < NUM_LAMPS_PER_STROBE);
        lamp_idx++)
      {
        /* check value supplied is in correct range and larger
           than current brightness */
        if( (kern_buf[lamp_idx] >= offset) &&
            (kern_buf[lamp_idx] < offset+NUM_LAMP_PERIODS) &&
            (kern_buf[lamp_idx] > brightness+offset))
          {
            lamp_dev_ptr->data[strobe_idx][brightness] |= lamp_bitmask;
          }
        else
          {
            lamp_dev_ptr->data[strobe_idx][brightness] &= ~lamp_bitmask;
          }
        lamp_bitmask <<= 1;
      }
    }

  /* no feedback from lamps so either fault or fully written */
  retval = count;
 out:
  kfree(kern_buf);
  return(retval);
}

/******************************************************************************/

int lamp_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
             unsigned long arg)
{
  int retval = 0;
  switch(cmd)
    {


    default:
      retval = -ENOTTY;
      break;
    }
  return retval;
}

/*****************************************************************************/

int lamp_open(struct inode *inode, struct file *filp)
{
  filp->private_data = ASCII_MODE;
  return 0;
}

/*****************************************************************************/

int lamp_release(struct inode *inode, struct file *filp)
{
  /* do nothing I suppose */
  return 0;
}

/*****************************************************************************/
/*
 *    Exported file operations structure for driver...
 */

struct file_operations  lamp_fops = {
  owner:        THIS_MODULE,
  write:    lamp_write, /* write */
  ioctl:    lamp_ioctl, /* ioctl */
  open:           lamp_open,  /* open */
  release:  lamp_release,     /* release */
};

/*****************************************************************************/



void __exit lamp_cleanup(void)
{
  unregister_chrdev(LAMP_MAJOR, "mpx_lamps");

  lamp_dev_ptr->shutting_down++;
  del_timer_sync(&lamp_dev_ptr->strobe_timer);

  if(lamp_dev_ptr)
    {
      if(lamp_dev_ptr->control_ptr)
      {
        iounmap(lamp_dev_ptr->control_ptr);
      }
      if(lamp_dev_ptr->data_ptr)
      {
        iounmap(lamp_dev_ptr->data_ptr);
      }
    }

  release_mem_region(MPX_CTRL, 1);
  release_mem_region(MPX_LAMP_DATA, NUM_LAMP_PERIODS * 4);

  /*
  if(mpx_lamp_data)
    {
      for(i = 0; i < NUM_LAMP_STROBES; i++)
      {
        if(mpx_lamp_data[i])
          {
            kfree(mpx_lamp_data[i]);
          }
      }
      kfree(mpx_lamp_data);
    }
  */

  if(lamp_dev_ptr)
    {
      kfree(lamp_dev_ptr);
    }
}

/******************************************************************************/

static int __init lamp_init(void)
{
  int i, j;
  int err = 0;

  printk("mpx_lamps: Copyright (C) 2005-2006, Ed Langley "
       "(edwin_langley@hotmail.com\n");

  /* allocate the 2d array to hold lamp output data */
  /*  if( (mpx_lamp_data = kmalloc(sizeof(int*) * NUM_LAMP_STROBES, GFP_KERNEL)) ==
NULL)
    {
      printk(KERN_WARNING "mpx_lamps: can't alloc first dimension array\n");
      err = -ENOMEM;
      goto fail;
    }
  for(i = 0; i < NUM_LAMP_STROBES; i++)
    {
      if( (mpx_lamp_data[i] = kmalloc(sizeof(int*) * NUM_LAMP_PERIODS)) == NULL)
      {
        printk(KERN_WARNING
             "mpx_lamps: can't alloc second dimension array:%d\n", i);
        err = -ENOMEM;
      }
    }
  if(err)
    goto fail;
  */

  if( (lamp_dev_ptr = kmalloc(sizeof(struct mpx_lamp_dev), GFP_KERNEL)) == NULL)
```

```
    {
      printk(KERN_WARNING
           "mpx_lamps: can't alloc mem for lamp dev struct\n");
      err = -ENOMEM;
      goto out;
    }

  /* clear lamp data */
  for(i = 0; i < NUM_LAMP_STROBES; i++)
    {
      for(j = 0; j < NUM_LAMP_PERIODS; j++)
      {
        lamp_dev_ptr->data[i][j] = 0;
      }
    }

  /* register mem mapped I/O region */
  if(request_mem_region(MPX_CTRL, 1, "mpx lamps control register") == NULL)
    {
         printk(KERN_WARNING  "mpx_lamps:  can't  alloc  mem  I/O  region  for  control
reg\n");
      err = -EBUSY;
      goto out;
    }
    if(request_mem_region(MPX_LAMP_DATA,  NUM_LAMP_PERIODS  *  4,  "mpx  lamps  data
registers") == NULL)
    {
      printk(KERN_WARNING "mpx_lamps: can't alloc mem I/O region for data reg\n");
      err = -EBUSY;
      goto out;
    }

  /* unnecessary on intended architecture, but included for portability */
  lamp_dev_ptr->control_ptr = ioremap(MPX_CTRL, 1);
  lamp_dev_ptr->data_ptr = ioremap(MPX_LAMP_DATA, NUM_LAMP_PERIODS * 4);

  //  lamp_dev_ptr->mode = ASCII_MODE;

  lamp_dev_ptr->shutting_down = 0;
  init_timer(&lamp_dev_ptr->strobe_timer);
  lamp_dev_ptr->strobe_timer.data = 0;
  lamp_dev_ptr->strobe_timer.function = output_strobe;
  lamp_dev_ptr->strobe_timer.expires = jiffies + HZ/1000;
  add_timer(&lamp_dev_ptr->strobe_timer);

  /* Register lamps as character device */
  if ((register_chrdev(LAMP_MAJOR, "mpx_lamps", &lamp_fops)) < 0)
    {
      printk(KERN_WARNING "mpx_lamps: can't get major %d\n",
           LAMP_MAJOR);
      err = -EBUSY;
      goto out;
    }

  return 0;

 out:
  lamp_cleanup();
  return err;
}

module_init(lamp_init);
module_exit(lamp_cleanup);


MODULE_AUTHOR("Ed Langley");
MODULE_DESCRIPTION("Driver for multiplexed lamps on the Pluto 6 Board");
```

19.8 pluto6_fpga.h

This is a header file for the VFD and multiplexed lamp drivers describing the register addresses in the FPGA:

```
/******************************************************************************/

/*
 *    pluto6_fpga.h -- Header file for Pluto 6 drivers which use
 *                       the standard FPGA
 *
 *    (C) Copyright 2005-2006, Ed Langley (edwin_langley@hotmail.com)
 */

/******************************************************************************/


#define FPGA_BASE     0x20000000

#define MPX_CTRL      (FPGA_BASE+0x0001)
#define MPX_LAMP_DATA (FPGA_BASE+0x0800)

#define FPGA_AUX_BASE (FPGA_BASE+0x1900)

#define VFD0_CLOCK    (FPGA_AUX_BASE)
#define VFD0_DATA     (FPGA_AUX_BASE+0x01)
#define VFD0_RESET    (FPGA_AUX_BASE+0x02)

#define VFD1_CLOCK    (FPGA_AUX_BASE+0x03)
#define VFD1_DATA     (FPGA_AUX_BASE+0x04)
#define VFD1_RESET    (FPGA_AUX_BASE+0x05)
```

19.9 uclinux.c

This is the uClinux IDE driver with modifications for the Pluto 6 board:

```
/*****************************************************************************/
/*
 *  linux/drivers/ide/legacy/uclinux.c
 *
 *  A simple driver that is easy to adjust for a large number of
 *  embedded targets,  not just coldfire.
 *
 *      Copyright (C) 2001-2004 by David McCullough <davidm@snapgear.com>
 *      Copyright (C) 2002 by Greg Ungerer <gerg@snapgear.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.
 *
 *  ex: set ts=4 sw=4
 */
/****************************************************************************/

#include <linux/config.h>
#include <linux/types.h>
#include <linux/mm.h>
#include <linux/interrupt.h>
#include <linux/blkdev.h>
#include <linux/hdreg.h>
#include <linux/delay.h>
#include <linux/ide.h>

/****************************************************************************/

#ifdef CONFIG_COLDFIRE
#include <asm/coldfire.h>
#include <asm/mcfsim.h>
#endif

/****************************************************************************/
/*
 *    Our list of ports/irq's for different boards.
 */

static struct ide_defaults {
      ide_ioreg_t base;
      int              irq;
} ide_defaults[] = {
#if defined(CONFIG_SECUREEDGEMP3)
      { ((ide_ioreg_t) 0x30800000), 29 },
#elif defined(CONFIG_eLIA) || defined(CONFIG_DISKtel)
      { ((ide_ioreg_t) 0x30c00000), 29 },
#elif defined(CONFIG_M5249C3)
      { ((ide_ioreg_t) 0x50000020), 165 },
#elif defined(CONFIG_SIGNAL_MCP751)
      { ((ide_ioreg_t) 0x42000000), 65 }, // INT1
#elif defined(CONFIG_PLUTO6)
      { ((ide_ioreg_t) 0xF2000000), 27 },
#else
      #error "No IDE setup defined"
#endif
      { ((ide_ioreg_t) -1), -1 }
};

/****************************************************************************/
/*
 * the register offsets from the base address
 */
```

```
#if defined(CONFIG_PLUTO6)
 #define IDE_DATA 0x00
 #define IDE_ERROR      0x02
 #define IDE_NSECTOR    0x04
 #define IDE_SECTOR     0x06
 #define IDE_LCYL 0x08
 #define IDE_HCYL 0x0A
 #define IDE_SELECT     0x0C
 #define IDE_STATUS     0x0E

 #define IDE_CONTROL    0x10

 #define IDE_IRQ  -1 /* not used */

#else

#define IDE_DATA  0x00
#define IDE_ERROR 0x01
#define IDE_NSECTOR     0x02
#define IDE_SECTOR      0x03
#define IDE_LCYL  0x04
#define IDE_HCYL  0x05
#define IDE_SELECT      0x06
#define IDE_STATUS      0x07

/* The control register often changes */
#define IDE_CONTROL     0x0e

#define IDE_IRQ         -1 /* not used */

#endif

static int ide_offsets[IDE_NR_PORTS] = {
    IDE_DATA, IDE_ERROR,  IDE_NSECTOR, IDE_SECTOR, IDE_LCYL,
    IDE_HCYL, IDE_SELECT, IDE_STATUS,  IDE_CONTROL, IDE_IRQ
};

#define     DBGIDE(fmt,a...)
//#define   DBGIDE(fmt,a...) printk(fmt, a)

/*****************************************************************************/

static void OUTB(u8 addr, unsigned long port);
static void OUTBSYNC(ide_drive_t *drive, u8 addr, unsigned long port);
static void OUTW(u16 addr, unsigned long port);
static void OUTL(u32 addr, unsigned long port);
static void OUTSW(unsigned long port, void *addr, u32 count);
static void OUTSL(unsigned long port, void *addr, u32 count);
static u8   INB(unsigned long port);
static u16  INW(unsigned long port);
static u32  INL(unsigned long port);
static void INSW(unsigned long port, void *addr, u32 count);
static void INSL(unsigned long port, void *addr, u32 count);

static void uclinux_irq_remove();
/*****************************************************************************/
/*
```

```
 * if needed, this function returns 1 if an interrupt was acked
 */

#if defined(CONFIG_M5249C3)
static int ack_intr(ide_hwif_t* hwif)
{
      /* Clear interrupts for GPIO5 */
      *((volatile unsigned long *) 0x800000c0) = 0x00002020;
      return 1;
}
#elif defined(CONFIG_SIGNAL_MCP751)

static void mcf_autovector(unsigned int vec)
{
      /* Everything is auto-vectored on the 5272 */
}

static int ack_intr(ide_hwif_t* hwif)
{
      volatile unsigned long  *icrp;
      /* Acknowledge interrupt */
      icrp = (volatile unsigned long *) (MCF_MBAR + MCFSIM_ICR1);
      *icrp = (*icrp & 0x77777777) | 0x80000000; // INT1
}

#elif defined(CONFIG_PLUTO6)

static int ack_intr(ide_hwif_t* hwif)
{
  /* read the ATA status register in order to clear the pending interrupt */
  u8 status;
  status = (*(volatile u8 *)(0xF2000000 + IDE_STATUS));
  return 1;
}

#else
#define ack_intr NULL
#endif

/*****************************************************************************/

void uclinux_ide_init(void)
{
      hw_regs_t hw;
      ide_ioreg_t base;
      int index, i;
      ide_hwif_t* hwif;

      for (i = 0; ide_defaults[i].base != (ide_ioreg_t) -1; i++) {
            base = ide_defaults[i].base;
            memset(&hw, 0, sizeof(hw));

#ifdef CONFIG_COLDFIRE
            mcf_autovector(hw.irq);
#endif
            hw.irq = ide_defaults[i].irq;
            ide_setup_ports(&hw, base, ide_offsets, 0, 0, ack_intr,
                        /* ide_iops, */
                        ide_defaults[i].irq);
            index = ide_register_hw(&hw, &hwif);
            if (index != -1) {
                  hwif->mmio  = 1; /* not io(0) or mmio(1) */
#ifdef CONFIG_COLDFIRE
                  hwif->OUTB  = OUTB;
                  hwif->OUTBSYNC = OUTBSYNC;
                  hwif->OUTW  = OUTW;
                  hwif->OUTL  = OUTL;
                  hwif->OUTSW = OUTSW;
                  hwif->OUTSL = OUTSL;
                  hwif->INB   = INB;
                  hwif->INW   = INW;
                  hwif->INL   = INL;
                  hwif->INSW  = INSW;
                  hwif->INSL  = INSL;

                  DRIVER(&hwif->drives[0])->cleanup = uclinux_irq_remove;

#endif
            }
            disable_irq(ide_defaults[i].irq);
      }
}

/*****************************************************************************/
/*
 * this is called after the irq handler has been installed,  so if you need
 * to fix something do it here
 */

void
uclinux_irq_fixup()
{
#ifdef CONFIG_M5249C3
      /* Enable interrupts for GPIO5 */
      *((volatile unsigned long *) 0x800000c4) |= 0x00000020;

      /* Enable interrupt level for GPIO5 - VEC37 */
      *((volatile unsigned long *) 0x80000150) |= 0x00200000;
#elif defined(CONFIG_SIGNAL_MCP751)
      // ICR1 enable INT1
      {
      volatile unsigned long  *icrp;

      icrp = (volatile unsigned long *) (MCF_MBAR + MCFSIM_ICR1);
      *icrp = (*icrp & 0x07777777) | 0xd0000000; // INT1
      }
      DBGIDE("IDE enable irq %s %s\n", __DATE__, __TIME__);

      // PITR trig L->H
      *((volatile unsigned long *) (MCF_MBAR+MCFSIM_PITR)) |= 0x80000000; // INT1

#elif defined(CONFIG_PLUTO6)
      /*  turn off/mask interrupts in the cremson graphics controller
       *  as it shares the interrupt level with the IDE
```

```
	 */
	*(volatile unsigned long *) 0x31fc0024 |= 0x1F000000;

	/* then clear EINT3 bit in the IMR of the Coldfire
	   to unmask the IDE interrupt */
	      //*((volatile unsigned long *)(MCF_MBAR+MCFSIM_IMR)) &= 0xFFF7FFFF;
	// unmasking done by mcf_autovector()

	mcf_autovector(27);
#endif
}

static void
uclinux_irq_remove()
{
#ifdef CONFIG_M5249C3
#elif defined(CONFIG_SIGNAL_MCP751)
	DBGIDE("IDE free irq %s %s\n", __DATE__, __TIME__);
	// ICR1 disable INT1
	{
	volatile unsigned long  *icrp;

	icrp = (volatile unsigned long *) (MCF_MBAR + MCFSIM_ICR1);
	*icrp = (*icrp & 0x07777777) | 0x80000000;
	}
#elif defined(CONFIG_PLUTO6)

	/* set the EINT3 bit in the IMR of the Coldfire to mask out the
	   IDE interrupt */
	      *((volatile unsigned long *)(MCF_MBAR+MCFSIM_IMR)) |= 0x00080000;

#endif
}

/****************************************************************************/
/*
 * the custom IDE IO access for platforms that need it
 */

#ifdef CONFIG_COLDFIRE

#if defined(CONFIG_eLIA) || defined(CONFIG_DISKtel)
 /* 8/16 bit acesses are controlled by flicking bits in the CS register */
 #define    MODE_16BIT()      \
	*((volatile unsigned short *) (MCF_MBAR + MCFSIM_CSCR6)) = 0x0080
 #define    MODE_8BIT() \
	*((volatile unsigned short *) (MCF_MBAR + MCFSIM_CSCR6)) = 0x0040

 #define    ENABLE16()        MODE_16BIT()
 #define    ENABLE32()        MODE_16BIT()
 #define    DISABLE16()       MODE_8BIT()
 #define    DISABLE32()       MODE_8BIT()
#endif /* defined(CONFIG_eLIA) || defined(CONFIG_DISKtel) */

#ifdef CONFIG_SECUREEDGEMP3
 #define    ADDR8(addr)       (((addr) & 0x1) ? (0x8000+(addr)-1) : (addr))
 #define    ADDR16(addr)      (addr)
 #define    ADDR32(addr)      (addr)
 #define    SWAP8(w)          ((((w) & 0xffff) << 8) | (((w) & 0xffff) >> 8))
 #define    SWAP16(w)         (w)
 #define    SWAP32(w)         (w)

 #define    io8_t unsigned short
#endif CONFIG_SECUREEDGEMP3

#if defined(CONFIG_M5249C3)
 #define    ADDR(a)                 (((a) & 0xfffffff0) + (((a) & 0xf) << 1))
 #define    ADDR8(addr)       ADDR(addr)
 #define    ADDR16(addr)      ADDR(addr)
 #define    ADDR32(addr)      ADDR(addr)

 #define    SWAP8(w)          (w)
 #define    SWAP16(w)         ((((w)&0xffff) << 8)   | (((w)&0xffff) >> 8))
 #define    SWAP32(w)         ((((w)&0xff) << 24)    | (((w)&0xff00) << 8) | \
                                             (((w)&0xff0000)   >>   8)   |   (((w)
&0xff000000) >> 24))

 #define io8_t    unsigned short
#endif

#if defined(CONFIG_SIGNAL_MCP751)
 #define    ADDR(a)                 (((a) & 0xfffffff0) + (((a) & 0xf) << 1))
 #define    ADDR8(addr)       ADDR(addr)
 #define    ADDR16(addr)      ADDR(addr)
 #define    ADDR32(addr)      ADDR(addr)

 #define    SWAP8(w)          (w&0xff)
 #define    SWAP16(w)         ((((w)&0xffff) << 8)   | (((w)&0xffff) >> 8))
 #define    SWAP32(w)         ((((w)&0xff) << 24)    | (((w)&0xff00) << 8) | \
                                             (((w)&0xff0000)   >>   8)   |   (((w)
&0xff000000) >> 24))

 #define io8_t    unsigned short
#endif

#if defined(CONFIG_PLUTO6)
 #define        ADDR(a)         (a)

// #define         SWAP16(w)        ( (((w) & 0x0ff) << 8) | (((w) & 0x0ff00) >>
8 ) )
/* try changing between 8/16 bit data path by changing PS bits in CS1 reg: */
 #define        MODE_16BIT()\
 ({\
  *((volatile unsigned short *) (MCF_MBAR + MCFSIM_CSCR1)) |= 0x0080;\
  *((volatile unsigned short *) (MCF_MBAR + MCFSIM_CSCR1)) &= ~0x0040;\
 })

 #define        MODE_8BIT()\
 ({\
  *((volatile unsigned short *) (MCF_MBAR + MCFSIM_CSCR1)) &= ~0x0080;\
  *((volatile unsigned short *) (MCF_MBAR + MCFSIM_CSCR1)) |= 0x0040;\
 })

 #define        ENABLE8()               MODE_8BIT()
 #define    ENABLE16()        MODE_16BIT()
 #define    ENABLE32()        MODE_16BIT()
 #define    DISABLE16()       MODE_8BIT()
 #define    DISABLE32()       MODE_8BIT()

#endif


/****************************************************************************/
/*
 * default anything that wasn't set
 */

#ifndef io8_t
#define io8_t     unsigned char
#endif

#ifndef io16_t
#define io16_t    unsigned short
#endif

#ifndef io32_t
#define io32_t    unsigned long
#endif

#ifndef ENABLE8
#define ENABLE8()
#endif

#ifndef ENABLE16
#define ENABLE16()
#endif

#ifndef ENABLE32
#define ENABLE32()
#endif

#ifndef DISABLE8
#define DISABLE8()
#endif

#ifndef DISABLE16
#define DISABLE16()
#endif

#ifndef DISABLE32
#define DISABLE32()
#endif

#ifndef SWAP8
#define SWAP8(x) (x)
#endif

#ifndef SWAP16
#define SWAP16(x) (x)
#endif
```

```
#ifndef SWAP16
#define SWAP16(x) (x)
#endif

#ifndef ADDR8
#define ADDR8(x) (x)
#endif

#ifndef ADDR16
#define ADDR16(x) (x)
#endif

#ifndef ADDR32
#define ADDR32(x) (x)
#endif

#ifndef SWAP8
#define SWAP8(x) (x)
#endif

#ifndef SWAP16
#define SWAP16(x) (x)
#endif

#ifndef SWAP32
#define SWAP32(x) (x)
#endif

/****************************************************************************/

static void
OUTB(u8 val, unsigned long addr)
{
      volatile io8_t    *rp;

      DBGIDE("%s(val=%x,addr=%lx)\n", __FUNCTION__, val, addr);
      rp = (volatile unsigned short *) ADDR(addr);
      ENABLE8();
      *rp = SWAP8(val);
      DISABLE8();
}

static void
OUTBSYNC(ide_drive_t *drive, u8 val, unsigned long addr)
{
      volatile io8_t    *rp;

      DBGIDE("%s(val=%x,addr=%lx)\n", __FUNCTION__, val, addr);
      rp = (volatile unsigned short *) ADDR(addr);
      ENABLE8();
      *rp = SWAP8(val);
      DISABLE8();
}

static u8
INB(unsigned long addr)
{
	volatile io8_t    *rp, val;

	rp = (volatile unsigned short *) ADDR(addr);
	ENABLE8();
	val = *rp;
	DISABLE8();
	DBGIDE("%s(addr=%lx) = 0x%x\n", __FUNCTION__, addr, SWAP8(val));
	return(SWAP8(val));
}

static void
OUTW(u16 val, unsigned long addr)
{
	volatile io16_t   *rp;

	DBGIDE("%s(val=%x,addr=%lx)\n", __FUNCTION__, val, addr);
	rp = (volatile io16_t *) ADDR(addr);
	ENABLE16();
	*rp = SWAP16(val);
	DISABLE16();
}

static void
OUTSW(unsigned long addr, void *vbuf, u32 len)
{
	volatile io16_t   *rp, val;
	unsigned short    *buf;

	DBGIDE("%s(addr=%lx,vbuf=%p,len=%lx)\n", __FUNCTION__, addr, vbuf, len);
	buf = (unsigned short *) vbuf;
	rp = (volatile io16_t *) ADDR(addr);
	ENABLE16();
	for (; (len > 0); len--) {
		val = *buf++;
		*rp = SWAP16(val);
	}
	DISABLE16();
}

static u16
INW(unsigned long addr)
{
	volatile io16_t *rp, val;

	rp = (volatile io16_t *) ADDR(addr);
	ENABLE16();
	val = *rp;
	DISABLE16();
	DBGIDE("%s(addr=%lx) = 0x%x\n", __FUNCTION__, addr, SWAP16(val));
	return(SWAP16(val));
}

static void
INSW(unsigned long addr, void *vbuf, u32 len)
{
	volatile io16_t *rp;
	unsigned short          w, *buf;
	DBGIDE("%s(addr=%lx,vbuf=%p,len=%lx)\n", __FUNCTION__, addr, vbuf, len);
	buf = (unsigned short *) vbuf;
	rp = (volatile io16_t *) ADDR(addr);
	ENABLE16();
	for (; (len > 0); len--) {
		w = *rp;
		*buf++ = SWAP16(w);
	}
	DISABLE16();
}

static u32
INL(unsigned long addr)
{
	volatile io32_t *rp, val;

	rp = (volatile io32_t *) ADDR(addr);
	ENABLE32();
	val = *rp;
	DISABLE32();
	DBGIDE("%s(addr=%lx) = 0x%lx\n", __FUNCTION__, addr, SWAP32(val));
	return(SWAP32(val));
}

static void
INSL(unsigned long addr, void *vbuf, u32 len)
{
	volatile io32_t *rp;
	unsigned long		w, *buf;

	DBGIDE("%s(addr=%lx,vbuf=%p,len=%lx)\n", __FUNCTION__, addr, vbuf, len);
	buf = (unsigned long *) vbuf;
	rp = (volatile io32_t *) ADDR(addr);
	ENABLE32();
	for (; (len > 0); len--) {
		w = *rp;
		*buf++ = SWAP32(w);
	}
	DISABLE32();
}

static void
OUTL(u32 val, unsigned long addr)
{
	volatile io32_t   *rp;

	DBGIDE("%s(val=%lx,addr=%lx)\n", __FUNCTION__, val, addr);
	rp = (volatile io32_t *) ADDR(addr);
	ENABLE32();
	*rp = SWAP32(val);
	DISABLE32();
}

static void
OUTSL(unsigned long addr, void *vbuf, u32 len)
{
      volatile io32_t   *rp, val;
      unsigned long     *buf;

      DBGIDE("%s(addr=%lx,vbuf=%p,len=%lx)\n", __FUNCTION__, addr, vbuf, len);
      buf = (unsigned long *) vbuf;
      rp = (volatile io32_t *) ADDR(addr);
      ENABLE32();
      for (; (len > 0); len--) {
            val = *buf++;
            *rp = SWAP32(val);
      }
      DISABLE32();
}

#endif /* CONFIG_COLDFIRE */

/*****************************************************************************/
```

19.10 cremsonfb.c

This is the Cremson frame buffer driver:

```
/*
 *  cremsonfb.c -- Framebuffer driver for Fujitsu MB86290A Cremson graphics
 *                 chip on the Heber Pluto 6 board
 *
 *  - Derived from skeletonfb.c, Created 28 Dec 1997 by Geert Uytterhoeven
 *
 *  (c) 2006 Ed Langley edwin_langley@hotmail.com
 *
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/tty.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/fb.h>
#include <linux/init.h>

#include <video/fbcon.h>
#include <video/fbcon-cfb8.h>
#include <video/fbcon-cfb16.h>


#include <asm/io.h>
#include <asm/byteorder.h>

#include "cremsonfb.h"

struct cremsonfb_info {
    /*
     *  Choose _one_ of the two alternatives:
```

```
     *
     *    1. Use the generic frame buffer operations (fbgen_*).
     */
    struct fb_info_gen gen;
    /*
     *    2. Provide your own frame buffer operations.
     */
  //    struct fb_info info;

    /* Here starts the frame buffer device dependent part */
    /* You can use this to store e.g. the board number if you support */
    /* multiple boards */

  uint32_t fbcon_cmap[CREMSON_PALETTE_SIZE];   /* basic palette */

  void *screen_base_phys;     /* physical address of video memory */
  void *screen_base_virt;     /* virtual address of video memory */
  unsigned long vid_mem_len;    /* length of whole video memory */
};


struct cremsonfb_par {
    /*
     *  The hardware specific data in this structure uniquely defines a video
     *  mode.
     *
     *  If your hardware supports only one video mode, you can leave it empty.
     */

  __u32 xres;           /* Hor Resolution px */
  __u32 yres;           /* Vert Resolution px */
  __u8  bpp;             /* bits per pixel */
  /* vid mode register shadows */
  __u16 dcm;            /* Display Control Mode Register  */
  __u16 htp;            /* Hor Total Pixels Register  */
  __u16 hdp;            /* Hor Display Period Register  */
  __u16 hdb;            /* Hor Display Boundary Register  */
  __u16 hsp;            /* HSync Position Register  */
  __u8  hsw;            /* HSync Width Register  */
  __u8  vsw;            /* Vsync Width Register  */
  __u16 vtr;            /* Vert Total Rasters Register  */
  __u16 vsp;            /* Vsync Position Register  */
  __u16 vdp;            /* Vert Display Period Register  */

  /* layer information */
  __u8  layer_num;              /* BL | BR | ML | MR | W | C */
  __u32 layer_width;            /* px */
  __u32 layer_height;           /* px */
  __u32 layer_mode;           /* SINGLE | DOUBLEBUFFER, INDIRECT | DIRECT_COLOUR,
                         layer frame width in 64 byte units, height in px */
  __u16 transparent_color;
  __u16 xoffset;          /* offset in px of layer frame from top left */
  __u16 yoffset;        /* offset in px of layer frame from top left */
  __u32 frame0_addr;          /* start address double buffer frame 0 */
  __u32 frame1_addr;          /* start address double buffer frame 1 */
  __u32 display0_addr;            /* display address double buffer frame 0 */
  __u32 display1_addr;            /* display address double buffer frame 1 */
  __u8  palette_size;            /* 16 for fake 16 bit console palette,
                            256 for real 8 bit one */
};

/* declare some vid modes */
static struct cremsonfb_par direct_1024_768 = {
  1024,           /* Hor Resolution */
  768,            /* Vert Resolution */
  16,            /* bits per pixel */
  0x0200,   /* Display Control Mode Register  */
  0x056c,   /* Hor Total Pixels Register  */
  0x03ff,   /* Hor Display Period Register  */
  0x03ff,   /* Hor Display Boundary Register  */
  0x0419,   /* HSync Position Register  */
  0x95,           /* HSync Width Register  */
  0x02,           /* Vsync Width Register  */
  0x0325,   /* Vert Total Rasters Register  */
  0x030f,   /* Vsync Position Register  */
  0x02ff,   /* Vert Display Period Register  */

  BL,
  1024,
  768,
  0x80200300,
  BLACK,
  0,
  0,
  0,
  0,
  0,
  0,
  16             /* palette size */
};

static struct cremsonfb_par indirect_1024_768 = {
  1024,           /* Hor Resolution */
  768,            /* Vert Resolution */
  8,             /* bits per pixel */
  0x0200,   /* Display Control Mode Register  */
  0x056c,   /* Hor Total Pixels Register  */
  0x03ff,   /* Hor Display Period Register  */
  0x03ff,   /* Hor Display Boundary Register  */
  0x0419,   /* HSync Position Register  */
  0x95,           /* HSync Width Register  */
  0x02,           /* Vsync Width Register  */
  0x0325,   /* Vert Total Rasters Register  */
  0x030f,   /* Vsync Position Register  */
  0x02ff,   /* Vert Display Period Register  */

  BL,
  1024,
  768,
  0x00100300,
  BLACK,
  0,
  0,
  0,
  0,
  0,
  0,
  255           /* palette size */
};


    /*
     *  If your driver supports multiple boards, you should make these arrays,
     *  or allocate them dynamically (using kmalloc()).
     */

static struct cremsonfb_info fb_info;
static struct cremsonfb_par current_par;
static int current_par_valid = 0;
static struct display disp;

static struct fb_var_screeninfo default_var;

//static int currcon = 0;
//static int inverse = 0;


/* ------------------- chipset specific functions -------------------------- */

static void cremson_detect(void);
static int cremson_encode_fix(struct fb_fix_screeninfo *fix,
                        struct cremsonfb_par *par,
                        const struct fb_info_gen *_info);
static int cremson_decode_var(struct fb_var_screeninfo *var,
                        struct cremsonfb_par *_par,
                        const struct fb_info *info);
static int cremson_encode_var(struct fb_var_screeninfo *var,
                        struct cremsonfb_par *par,
                        const struct fb_info *info);
static void cremson_get_par(struct cremsonfb_par *_par,
                      const struct fb_info *info);
static void cremson_set_par(struct cremsonfb_par *par,
                      const struct fb_info *info);
static int cremson_getcolreg(unsigned regno, unsigned *red,
                       unsigned *green, unsigned *blue,
                       unsigned *transp, const struct fb_info *info);
static int cremson_setcolreg(unsigned regno, unsigned red,
                       unsigned green, unsigned blue,
                       unsigned transp, const struct fb_info *info);
static int cremson_pan_display(struct fb_var_screeninfo *var,
                         struct cremsonfb_par *par,
                         const struct fb_info *info);
static int cremson_blank(int blank_mode, const struct fb_info *info);
static void cremson_set_disp(const void *par, struct display *disp,
                       struct fb_info_gen *info);


static void cremson_detect(void)
{
    /*
```

```
     *  This function should detect the current video mode settings and store
     *  it as the default video mode
     */

    struct cremsonfb_par par;

    /* ... */
    cremson_get_par(&par, &fb_info.gen.info);
    cremson_encode_var(&default_var, &par, &fb_info.gen.info);
}

static int cremson_encode_fix(struct fb_fix_screeninfo *fix,
                        struct cremsonfb_par *par,
                        const struct fb_info_gen *_info)
{
    /*
     *  This function should fill in the 'fix' structure based on the values
     *  in the `par' structure.
     */

    /* ... */
  //  __u8 colmode;

  struct cremsonfb_info *info = container_of(_info, struct cremsonfb_info, gen);

  memset(fix, 0, sizeof(struct fb_fix_screeninfo));
  strcpy(fix->id, "Fujitsu Cremson");

  fix->smem_start = (unsigned long) info->screen_base_phys;
  fix->mmio_start = (unsigned long) info->screen_base_virt;

  //  colmode = (par->layer_mode >> CREMSON_COLMODE_REG_OFFSET);

  switch(par->bpp)
    {
    case 8:
      fix->mmio_len = par->xres * par->yres;
      fix->smem_len = par->xres * par->yres;
      fix->line_length = par->xres;
      fix->visual = FB_VISUAL_PSEUDOCOLOR;
      break;
    case 16:
      /* 2 bytes per pixel */
      fix->mmio_len = 2 * par->xres * par->yres;
      fix->smem_len = 2 * par->xres * par->yres;
      fix->line_length = 2 * par->xres;
      fix->visual = FB_VISUAL_TRUECOLOR;
      break;
    default:
      /* cremson doesn't do any other colur depths */
    }

  fix->type = FB_TYPE_PACKED_PIXELS;
  fix->type_aux = 0;
  fix->xpanstep = 0;
  fix->ypanstep = 0;
  fix->ywrapstep = 0;
  fix->accel = 0;

  return 0;
}

static int cremson_decode_var(struct fb_var_screeninfo *var, struct cremsonfb_par
*_par,
                        const struct fb_info *info)
{
    /*
     *  Fill the `par' structure based on the values in `var'.
     *  If a value doesn't fit, round it up, if it's too big, return -EINVAL.
     *
     *  Suggestion: Round up in the following order: bits_per_pixel, xres,
     *  yres, xres_virtual, yres_virtual, xoffset, yoffset, grayscale,
     *  bitfields, horizontal timing, vertical timing.
     */


    /* ... */
  //  struct cremsonfb_par *chosen_par;
  struct cremsonfb_par *par = _par;

  /* choose one of the preset pars for simplicity */

  //  *par = direct_1024_768;
  *par = indirect_1024_768;

  /* alter this at some point to read var properly,
     remember to set display_addr0/1 according to
     offset fields correctly
  */

    /* pixclock in picos, htotal in pixels, vtotal in scanlines */
  //    if (!fbmon_valid_timings(pixclock, htotal, vtotal, info))
  //      return -EINVAL;

    return 0;
}

static int cremson_encode_var(struct fb_var_screeninfo *var, struct cremsonfb_par
*par,
                        const struct fb_info *info)
{
    /*
     *  Fill the 'var' structure based on the values in 'par' and maybe other
     *  values read out of the hardware.
     */

    /* ... */
  /* do this bit properly as practice for previous function */

  __u16 pixclk_scalar;

  memset(var, 0, sizeof(*var));

  var->xres = par->xres;
```

```
var->yres = par->yres;
var->xres_virtual = par->layer_width;
var->yres_virtual = par->layer_height;
var->xoffset = par->xoffset;
var->yoffset = par->yoffset;

//  if( (par->layer_mode >> CREMSON_COLMODE_REG_OFFSET)
//      == CREMSON_INDIRECT_COLOUR)

switch(par->bpp)
  {
  case 8:
    var->bits_per_pixel = 8;
    /* only length in these bitfields are important for indirect colour */
    var->red.length = 8;
    var->green.length = 8;
    var->blue.length = 8;
    //      var->red.length = 16;
    //      var->green.length = 16;
    //      var->blue.length = 16;
    break;
  case 16:
    var->bits_per_pixel = 16;
    var->red.offset = CREMSON_DIRECT_OFFSET_R;
    var->green.offset = CREMSON_DIRECT_OFFSET_G;
    var->blue.offset = CREMSON_DIRECT_OFFSET_B;
    var->red.length = CREMSON_DIRECT_COLOUR_LENGTH;
    var->green.length = CREMSON_DIRECT_COLOUR_LENGTH;
    var->blue.length = CREMSON_DIRECT_COLOUR_LENGTH;
    break;
  default:
    /* cremson doesn't do any other colour depths */
  }

var->activate = FB_ACTIVATE_NOW;

var->height = -1;
var->width = -1;
var->accel_flags = 0;

/* timing data - see fb docs in kernel sources and
   cremson data sheet */
pixclk_scalar = ((par->dcm & CREMSON_DCM_SCALAR_MASK)
             >> CREMSON_DCM_SCALAR_OFFSET);
pixclk_scalar += 1;

var->pixclock = (CREMSON_PLL_CLK_FREQ / pixclk_scalar);


/* 1/6 blank period for left margin,
   1/3 for right, and other 1/2 for
   hsync - just a guess at the mo
*/

/*
var->left_margin = (par->htp - par->hsp) / 6;
var->right_margin = 2 * var->left_margin;
```

```
  var->hsync_len = par->hsw / 2;

  var->upper_margin = (par->vtr - par->vsp) / 3;
  var->lower_margin = 2 * var->upper_margin;
  var->vsync_len = par->vsw;
  */

  var->left_margin = (par->hsp - par->hdp) + 1;
  var->right_margin = (par->htp - par->hsp - par->hsw) + 1;
  var->hsync_len = par->hsw +1;

  var->upper_margin = (par->vsp - par->vdp);
  var->lower_margin = (par->vtr - par->vsp - par->vsw) + 1;
  var->vsync_len = par->vsw + 1;

  var->sync = 0;
  var->vmode = FB_VMODE_NONINTERLACED;

  return 0;
}

static void cremson_get_par(struct cremsonfb_par *_par, const struct fb_info *info)
{
    /*
     *  Fill the hardware's 'par' structure.
     */

  struct cremsonfb_par *par= _par;

    if (current_par_valid)
      *par = current_par;
    else {
      /* ... */
      /* lazy way to do it for the moment ... */
      //      *par = direct_1024_768;
      *par = indirect_1024_768;
    }
}

static void cremson_set_par(struct cremsonfb_par *par, const struct fb_info *_info)
{
    /*
     *  Set the hardware according to 'par'.
     */

    current_par = *par;
    current_par_valid = 1;
    /* ... */

    /* cremson MMR set in init */

    /* set the video mode - cremson is little endian */
    writew(cpu_to_le16(par->dcm), CREMSONDCM);
    writew(cpu_to_le16(par->htp), CREMSONHTP);
    writew(cpu_to_le16(par->hdp), CREMSONHDP);
    writew(cpu_to_le16(par->hdb), CREMSONHDB);
    writew(cpu_to_le16(par->hsp), CREMSONHSP);
    writeb(par->hsw, CREMSONHSW);
    writeb(par->vsw, CREMSONVSW);
    writew(cpu_to_le16(par->vtr), CREMSONVTR);
    writew(cpu_to_le16(par->vsp), CREMSONVSP);
    writew(cpu_to_le16(par->vdp), CREMSONVDP);

    /* the the video mem */
    //memset_io(CREMSON_VIDMEM_BASE, 0, CREMSONHOSTBASE);

    /* initialise the layer */
    writel(cpu_to_le32(par->layer_mode), CREMSONBLM);
    writew(cpu_to_le16(par->xoffset), CREMSONBLDX);
    writew(cpu_to_le16(par->yoffset), CREMSONBLDY);
    writel(cpu_to_le32(par->frame0_addr), CREMSONBLOA0);
    writel(cpu_to_le32(par->frame1_addr), CREMSONBLOA1);
    writel(cpu_to_le32(par->display0_addr), CREMSONBLDA0);
    writel(cpu_to_le32(par->display1_addr), CREMSONBLDA1);

    /* turn on display and the base left layer */
    writew(cpu_to_le16(CREMSON_DISPLAY_ENABLE |
                   CREMSON_BL_ENABLE), CREMSONDCE);

}

static int cremson_getcolreg(unsigned regno, unsigned *red, unsigned *green,
                       unsigned *blue, unsigned *transp,
                       const struct fb_info *_info)
{
    /*
     *  Read a single color register and split it into colors/transparent.
     *  The return values must have a 16 bit magnitude.
     *  Return != 0 for invalid regno.
     */

    /* ... */
  struct cremsonfb_info *info = (struct cremsonfb_info *) _info;
  uint32_t colour;
  //u16 colour;
  u8 *cremson_pal_ptr = CREMSONMBPAL;

  if (regno > current_par.palette_size)
    return 1;

  /* convert colour data read back from colour map entry to little endian */
  //  colour = cpu_to_le16(info->fbcon_cmap[regno]);
  colour = info->fbcon_cmap[regno];

  switch(current_par.bpp)
    {
#ifdef FBCON_HAS_CFB8
    case 8:
      *red = *( (u8 *)(cremson_pal_ptr + 4*regno + 2));
      *green = *( (u8 *)(cremson_pal_ptr + 4*regno + 1));
      *blue = *( (u8 *)(cremson_pal_ptr + 4*regno));
      *transp = 0;
      break;
#endif
```

```
#if defined(FBCON_HAS_CFB15) || defined(FBCON_HAS_CFB16)
    case 16:
      *red = ((colour << 1) & 0xf800);
      *green = ((colour << 6) & 0xf800);
      *blue = ((colour << 11) & 0xf800);
      *transp = 0;
      break;
#endif
    default:
      return 1;
    }

    return 0;
}

static int cremson_setcolreg(unsigned regno, unsigned red, unsigned green,
                       unsigned blue, unsigned transp,
                       const struct fb_info *_info)
{
    /*
     *  Set a single color register. The values supplied have a 16 bit
     *  magnitude.
     *  Return != 0 for invalid regno.
     */

  struct cremsonfb_info *info = (struct cremsonfb_info *) _info;
  //  uint32_t colour;
  u8 *cremson_pal_ptr = CREMSONMBPAL;

  if (regno > current_par.palette_size)
    return 1;

  //colour = info->fbcon_cmap[regno];

  switch(current_par.bpp)
    {

#ifdef FBCON_HAS_CFB8
      case 8:
      //    info->fbcon_cmap[regno] = ( ((red & 0xf800) >> 1) |
      //                        ((green & 0xf800) >> 6) |
      //                        ((blue & 0xf800) >> 11) );
      /* set the palette in the card */
        *( (u8 *)(cremson_pal_ptr + 4*regno + 3)) = 0; /* ignore alpha for now */
            *( (u8 *)(cremson_pal_ptr + 4*regno + 2)) = ((u8)(red >> 8));
      *( (u8 *)(cremson_pal_ptr + 4*regno + 1)) = ((u8)(green >> 8));
      *( (u8 *)(cremson_pal_ptr + 4*regno)) = ((u8)(blue >> 8));

      break;
#endif
      /*
       *  Make the first 16 colors of the palette available to fbcon
       */
#if defined(FBCON_HAS_CFB15) || defined(FBCON_HAS_CFB16)
    case 16:
      /* RGB 555 */
      info->fbcon_cmap[regno] = ( ((red & 0xf800) >> 1) |
                          ((green & 0xf800) >> 6) |
                          ((blue & 0xf800) >> 11) );
      break;
#endif
    default:
      return 1;
    }
    /* ... */
    return 0;
}

static int cremson_pan_display(struct fb_var_screeninfo *var,
                         struct cremsonfb_par *par, const struct fb_info *info)
{
    /*
     *  Pan (or wrap, depending on the `vmode' field) the display using the
     *  `xoffset' and `yoffset' fields of the `var' structure.
     *  If the values don't fit, return -EINVAL.
     */

    /* ... */


  return 0;
}

static int cremson_blank(int blank_mode, const struct fb_info *info)
{
    /*
     *  Blank the screen if blank_mode != 0, else unblank. If blank == NULL
     *  then the caller blanks by setting the CLUT (Color Look Up Table) to all
     *  black. Return 0 if blanking succeeded, != 0 if un-/blanking failed due
     *  to e.g. a video mode which doesn't support it. Implements VESA suspend
     *  and powerdown modes on hardware that supports disabling hsync/vsync:
     *    blank_mode == 2: suspend vsync
     *    blank_mode == 3: suspend hsync
     *    blank_mode == 4: powerdown
     */

    /* ... */
    return 0;
}

static void cremson_set_disp(const void *par, struct display *disp,
                       struct fb_info_gen *_info)
{
    /*
     *  Fill in a pointer with the virtual address of the mapped frame buffer.
     *  Fill in a pointer to appropriate low level text console operations (and
     *  optionally a pointer to help data) for the video mode `par' of your
     *  video hardware. These can be generic software routines, or hardware
     *  accelerated routines specifically tailored for your hardware.
     *  If you don't have any appropriate operations, you must fill in a
     *  pointer to dummy operations, and there will be no text output.
     */

  struct cremsonfb_info *info = (struct cremsonfb_info *) _info;
  disp->screen_base = info->gen.info.screen_base;

  switch(current_par.bpp)
    {
#ifdef FBCON_HAS_CFB8
    case 8:
      disp->dispsw = &fbcon_cfb8;
      printk(KERN_DEBUG "FBCON_HAS_CFB8\n");
      break;
#endif
#ifdef FBCON_HAS_CFB16
    case 16:
      disp->dispsw = &fbcon_cfb16;
      disp->dispsw_data = &info->fbcon_cmap; /* console palette */
      printk(KERN_DEBUG "FBCON_HAS_CFB16\n");
      break;
#endif
    default:
      disp->dispsw = &fbcon_dummy;
      printk(KERN_DEBUG "Assigning dummy console ops\n");
      break;
    }

}


/* ------------ Interfaces to hardware functions ------------ */


struct fbgen_hwswitch cremsonfb_switch = {
    cremson_detect, cremson_encode_fix, cremson_decode_var, cremson_encode_var,
    cremson_get_par, cremson_set_par, cremson_getcolreg, cremson_setcolreg,
    cremson_pan_display, cremson_blank, cremson_set_disp
};



/* ------------------------------------------------------------------------- */


    /*
     *  Frame buffer operations
     */

/* If all you need is that - just don't define ->fb_open */
static int xxxfb_open(const struct fb_info *info, int user)
{
    return 0;
}

/* If all you need is that - just don't define ->fb_release */
static int xxxfb_release(const struct fb_info *info, int user)
{
    return 0;
}
```

```
static  unsigned  long  cremson_get_unmapped_area(struct  file  *file,  unsigned  long
addr,
                                     unsigned long len, unsigned long pgoff,
                                     unsigned long flags)
{
  return (unsigned long)fb_info.screen_base_phys;

}

    /*
     *  In most cases the `generic' routines (fbgen_*) should be satisfactory.
     *  However, you're free to fill in your own replacements.
     */

static struct fb_ops cremsonfb_ops = {
      owner:           THIS_MODULE,
      //    fb_open:    xxxfb_open,    /* only if you need it to do something */
      //    fb_release: xxxfb_release, /* only if you need it to do something */
      fb_get_fix: fbgen_get_fix,
      fb_get_var: fbgen_get_var,
      fb_set_var: fbgen_set_var,
      fb_get_cmap:     fbgen_get_cmap,
      fb_set_cmap:     fbgen_set_cmap,
      fb_pan_display:  fbgen_pan_display,
      //    fb_ioctl:   xxxfb_ioctl,   /* optional */
      get_fb_unmapped_area: cremson_get_unmapped_area,
};


/* -------------------- Hardware Independent Functions --------------------- */


    /*
     *  Initialization
     */

int __init cremsonfb_init(void)
{

  fb_info.screen_base_phys = (void*)CREMSON_VIDMEM_BASE;
  fb_info.vid_mem_len = CREMSON_VIDMEM_LENGTH;
  fb_info.screen_base_virt = ioremap(CREMSON_VIDMEM_BASE,
                             CREMSON_VIDMEM_LENGTH);

  writel(cpu_to_le32(CREMSON_MMR_VALUE), CREMSONMMR);

    fb_info.gen.fbhw = &cremsonfb_switch;
    fb_info.gen.fbhw->detect();
    fb_info.gen.parsize = sizeof(struct cremsonfb_par);
    strcpy(fb_info.gen.info.modename, "Fujitsu Cremson");
    fb_info.gen.info.changevar = NULL;
    fb_info.gen.info.node = -1;
    fb_info.gen.info.fbops = &cremsonfb_ops;
    fb_info.gen.info.disp = &disp;
    fb_info.gen.info.switch_con = &fbgen_switch;
    fb_info.gen.info.updatevar = &fbgen_update_var;
    fb_info.gen.info.blank = &fbgen_blank;
    fb_info.gen.info.flags = FBINFO_FLAG_DEFAULT;
    fb_info.gen.info.screen_base = fb_info.screen_base_virt;
    //    fb_info.gen.info.screen_size = fb_info.vid_mem_len;

    /* This should give a reasonable default video mode */
    fbgen_get_var(&disp.var, -1, &fb_info.gen.info);
    fbgen_do_set_var(&disp.var, 1, &fb_info.gen);

    /* so far have only set up var in disp struct */
    fb_info.gen.info.var = disp.var;

    fbgen_set_disp(-1, &fb_info.gen);
    fbgen_install_cmap(0, &fb_info.gen);
    if (register_framebuffer(&fb_info.gen.info) < 0)
      return -EINVAL;
    printk(KERN_INFO "fb%d: %s frame buffer device\n",
         GET_FB_IDX(fb_info.gen.info.node),
         fb_info.gen.info.modename);

    /* uncomment this if your driver cannot be unloaded */
    /* MOD_INC_USE_COUNT; */
    return 0;
}


    /*
     *  Cleanup
     */

static void __exit cremsonfb_cleanup(struct fb_info *info)
{
    /*
     *  If your driver supports multiple boards, you should unregister and
     *  clean up all instances.
     */

    unregister_framebuffer(info);

    current_par_valid = 0;
    /* ... */
}

static void __exit cremsonfb_exit(void)
{
  printk(KERN_DEBUG "Entering %s\n", __FUNCTION__);

  cremsonfb_cleanup(&fb_info.gen.info);

  printk(KERN_DEBUG "Leaving %s\n", __FUNCTION__);
}
    /*
     *  Setup
     */

int __init cremsonfb_setup(char *options)
{
    /* Parse user speficied options (`video=xxxfb:') */
  return 0;
}


/* ------------------------------------------------------------------------ */

    /*
     *  Modularization
     */


module_init(cremsonfb_init);
module_exit(cremsonfb_exit);
```

19.11 cremsonfb.h

This is the header for the Cremson frame buffer driver:

```
/*****************************************************************************/
/*
 *  cremsonfb.h -- Register address definitions for the Fujitsu Cremson
 *  framebuffer
 *                 driver on the Heber Pluto 6 board
 *
 *
 *  (c) 2006 Ed Langley edwin_langley@hotmail.com
 *
 */
/*****************************************************************************/

/*****************************************************************************/
#if defined(CONFIG_PLUTO6)
/*****************************************************************************/

#define CREMSON_VIDMEM_BASE             0x30000000
#define CREMSON_VIDMEM_LENGTH           0x01fbffff

/* this is blatantly stolen from Heber Pluto 6 code */
/* MMR memory mode register:
   Memory timing optimised for ISSI IS42S16400B-7T

   Minimum timing requirements for above device in brackets
   from ISSI data sheet for IS42S16400 Rev B March 2003 page 13

   CAS latency= 3 clk
   SDRAM bus width = 32 bits
   Row address width =14 bits
   Refresh time setting = 1024
   LOWD= 2 clk -> 20ns
   TRCD= 2 clk -> 20ns       (15ns)
   TRAS= 5 clk -> 50ns       (37ns)
   TRP= 2 clk -> 20ns        (15ns)
   TRC = 7 clk -> 70ns       (63ns)
   TRRD=2 clk -> 20ns        (14ns)
*/
```

```
#define CREMSON_MMR_VALUE               0x013d6a8b


/*****************************************************************************/
#else /* UNKNOWN HARDWARE */
/*****************************************************************************/

#error "cremsonfb: I don't know what hardware addresses to use?"

#endif
/*****************************************************************************/

/* Cremson address */
#define CREMSONHOSTBASE            CREMSON_VIDMEM_BASE + 0x1fc0000
#define CREMSONDISPLAYBASE         CREMSON_VIDMEM_BASE + 0x1fd0000
#define CREMSONDRAWBASE            CREMSON_VIDMEM_BASE + 0x1ff0000

/* Cremson Registers */

/* Host Interface Registers */
#define CREMSONDTC                 CREMSONHOSTBASE + 0x00  /*  DMA  Transfer  Count
Register  */
#define CREMSONDSU                 CREMSONHOSTBASE + 0x04  /*  DMA  Setup  Register
*/
#define CREMSONDRM                 CREMSONHOSTBASE + 0x05  /*   DMA   Request   Mask
Register  */
#define CREMSONDST                 CREMSONHOSTBASE + 0x06  /*  DMA  Status  Register
*/
#define CREMSONDTS                 CREMSONHOSTBASE + 0x08  /*   DMA   Transfer   Stop
Register  */
#define CREMSONLSTA                CREMSONHOSTBASE + 0x10  /* Display List Transfer
Status Register  */
#define CREMSONDRQ                 CREMSONHOSTBASE + 0x18  /* DMA Request Register
*/
#define CREMSONIST                 CREMSONHOSTBASE + 0x20  /*    Interrupt    Status
Register  */
#define CREMSONIMASK               CREMSONHOSTBASE + 0x24  /*     Interrupt     Mask
Register  */
#define CREMSONSRST                CREMSONHOSTBASE + 0x2c  /*     Software     Reset
Register  */
#define CREMSONLSA                 CREMSONHOSTBASE + 0x40  /* Display List  Source
Address Register  */
#define CREMSONLCO                 CREMSONHOSTBASE + 0x44  /*  Display  List  Count
Register  */
#define CREMSONLREQ                CREMSONHOSTBASE + 0x48  /* Display List Transfer
Request Register  */

/* Graphics Memory Interface Registers */
#define CREMSONMMR                 CREMSONHOSTBASE + 0xfffc      /*  Memory  IF  Mode
Register  */

/* Display Control Registers */
#define CREMSONDCM                 CREMSONDISPLAYBASE + 0x00     /* Display Control
Mode Register  */
#define CREMSONDCE                 CREMSONDISPLAYBASE + 0x02     /*         Display
Controller Enable Register  */
```

```
#define CREMSONHTP                CREMSONDISPLAYBASE + 0x06    /*    Hor    Total
Pixels Register  */
#define CREMSONHDP                CREMSONDISPLAYBASE + 0x08    /*   Hor   Display
Period Register  */
#define CREMSONHDB                CREMSONDISPLAYBASE + 0x0a    /*   Hor   Display
Boundary Register  */
#define CREMSONHSP                CREMSONDISPLAYBASE + 0x0c    /*  HSync  Position
Register  */
#define CREMSONHSW                CREMSONDISPLAYBASE + 0x0e    /*   HSync   Width
Register  */
#define CREMSONVSW                CREMSONDISPLAYBASE + 0x0f    /*   Vsync   Width
Register  */
#define CREMSONVTR                CREMSONDISPLAYBASE + 0x12    /*    Vert    Total
Rasters Register  */
#define CREMSONVSP                CREMSONDISPLAYBASE + 0x14    /*  Vsync  Position
Register  */
#define CREMSONVDP                CREMSONDISPLAYBASE + 0x16    /*   Vert   Display
Period Register  */
#define CREMSONWX           CREMSONDISPLAYBASE + 0x18    /*  Window  Position  X
Register  */
#define CREMSONWY           CREMSONDISPLAYBASE + 0x1a    /*  Window  Position  Y
Register  */
#define CREMSONWW           CREMSONDISPLAYBASE + 0x1c    /* Window Width Register
*/
#define CREMSONWH           CREMSONDISPLAYBASE + 0x1e    /*     Window     Height
Register  */
#define CREMSONCM           CREMSONDISPLAYBASE + 0x20    /* C Layer Mode */
#define CREMSONCOA                CREMSONDISPLAYBASE + 0x24    /*  C  Layer  Origin
Address */
#define CREMSONCDA                CREMSONDISPLAYBASE + 0x28    /* C Layer Display
Address  */
#define CREMSONCDX                CREMSONDISPLAYBASE + 0x2c    /* C Layer Display
Position X */
#define CREMSONCDY                CREMSONDISPLAYBASE + 0x2e    /* C Layer Display
Position Y */
#define CREMSONWM           CREMSONDISPLAYBASE + 0x30    /* W Layer Mode */
#define CREMSONWOA                CREMSONDISPLAYBASE + 0x34    /*  W  Layer  Origin
Address */
#define CREMSONWDA                CREMSONDISPLAYBASE + 0x38    /* W Layer Display
Address */
#define CREMSONMLM                CREMSONDISPLAYBASE + 0x40    /* ML Layer Mode
                  */
#define CREMSONMLOA0              CREMSONDISPLAYBASE + 0x44    /* ML Layer Origin
Address 0   */
#define CREMSONMLDA0              CREMSONDISPLAYBASE + 0x48    /*     ML     Layer
Display Address 0 */
#define CREMSONMLOA1              CREMSONDISPLAYBASE + 0x4c    /* ML Layer Origin
Address 1   */
#define CREMSONMLDA1              CREMSONDISPLAYBASE + 0x50    /*     ML     Layer
Display Address 1 */
#define CREMSONMLDX               CREMSONDISPLAYBASE + 0x54    /*     ML     Layer
Display Position X      */
#define CREMSONMLDY               CREMSONDISPLAYBASE + 0x56    /*     ML     Layer
Display Position Y      */
#define CREMSONMRM                CREMSONDISPLAYBASE + 0x58    /* MR Layer Mode
                  */
#define CREMSONMROA0              CREMSONDISPLAYBASE + 0x5c    /* MR Layer Origin
```

```
Address 0   */
#define CREMSONMRDA0                CREMSONDISPLAYBASE + 0x60     /*     MR     Layer
Display Address 0 */
#define CREMSONMROA1                CREMSONDISPLAYBASE + 0x64     /* MR Layer Origin
Address 1   */
#define CREMSONMRDA1                CREMSONDISPLAYBASE + 0x68     /*     MR     Layer
Display Address 1 */
#define CREMSONMRDX                 CREMSONDISPLAYBASE + 0x6c     /*     MR     Layer
Display Position X      */
#define CREMSONMRDY                 CREMSONDISPLAYBASE + 0x6e     /*     MR     Layer
Display Position Y      */
#define CREMSONBLM                  CREMSONDISPLAYBASE + 0x70     /* BL Layer Mode
                  */
#define CREMSONBLOA0                CREMSONDISPLAYBASE + 0x74     /* BL Layer Origin
Address 0   */
#define CREMSONBLDA0                CREMSONDISPLAYBASE + 0x78     /*     BL     Layer
Display Address 0 */
#define CREMSONBLOA1                CREMSONDISPLAYBASE + 0x7c     /* BL Layer Origin
Address 1   */
#define CREMSONBLDA1                CREMSONDISPLAYBASE + 0x80     /*     BL     Layer
Display Address 1 */
#define CREMSONBLDX                 CREMSONDISPLAYBASE + 0x84     /*     BL     Layer
Display Position X      */
#define CREMSONBLDY                 CREMSONDISPLAYBASE + 0x86     /*     BL     Layer
Display Position Y      */
#define CREMSONBRM                  CREMSONDISPLAYBASE + 0x88     /* BR Layer Mode
      */
#define CREMSONBROA0                CREMSONDISPLAYBASE + 0x8c     /* BR Layer Origin
Address 0   */
#define CREMSONBRDA0                CREMSONDISPLAYBASE + 0x90     /*     BR     Layer
Display Address 0 */
#define CREMSONBROA1                CREMSONDISPLAYBASE + 0x94     /* BR Layer Origin
Address 1   */
#define CREMSONBRDA1                CREMSONDISPLAYBASE + 0x98     /*     BR     Layer
Display Address 1 */
#define CREMSONBRDX                 CREMSONDISPLAYBASE + 0x9c     /*     BR     Layer
Display Position X      */
#define CREMSONBRDY                 CREMSONDISPLAYBASE + 0x9e     /*     BR     Layer
Display Position Y      */
#define CREMSONBRATIO               CREMSONDISPLAYBASE + 0xb4     /*  C  Layer  Blend
Ratio Control */
#define CREMSONBMODE                CREMSONDISPLAYBASE + 0xb6     /*  C  Layer  Blend
Mode Control */
#define CREMSONCTC                  CREMSONDISPLAYBASE + 0xbc     /*      C     Layer
Transparent Control */
#define CREMSONMRTC                 CREMSONDISPLAYBASE + 0xc0     /*     MR     Layer
Transparent Control */
#define CREMSONMLTC                 CREMSONDISPLAYBASE + 0xc2     /*     ML     Layer
Transparent Control */


#define CREMSONCPAL                 CREMSONDISPLAYBASE + 0x400    /* C Layer Palette
Base */
#define CREMSONMBPAL                CREMSONDISPLAYBASE + 0x800    /*    M/B    Layer
Palette Base */

/* Draw Control Registers */
```

```
#define CREMSONCTR                  CREMSONDRAWBASE + 0x400      /* Control */




/* other definitions */
#define BL                                0
#define BR                                1
#define ML                                2
#define MR                                3
#define W                                 4
#define C                                 5

#define CREMSON_PALETTE_SIZE              256

#define BLACK                             0

#define CREMSON_INDIRECT_COLOUR           0
#define CREMSON_DIRECT_COLOUR             1

#define CREMSON_COLMODE_REG_OFFSET        31

#define CREMSON_DIRECT_OFFSET_R           10
#define CREMSON_DIRECT_OFFSET_G           5
#define CREMSON_DIRECT_OFFSET_B           0

#define CREMSON_DIRECT_COLOUR_LENGTH      5

#define CREMSON_PLL_CLK_FREQ              200454520 /* Hz */
#define CREMSON_DCM_SCALAR_MASK           0x1F00
#define CREMSON_DCM_SCALAR_OFFSET         8

#define CREMSON_DISPLAY_ENABLE            0x8000
#define CREMSON_BL_ENABLE                 0x0008
```