My life with the MCF5407 Colfire Board (draft)

Norman Feske

July 10, 2004

Contents

1 Introduction

In the recent time, the Atari Coldfire Project was not publicly present very much. Thus, some people are sceptical about the current state of the project. As I cannot speak for all members of the project — each one dedicated to a different field of the project — I will present my activies and views in regard to create a Coldfire-based operating system in this document. I have to disappoint those of you, who expect the presentation of a ready-to-use operating system. This goal is not reached, yet. The time frame needed to build this operating system depends mainly on my spare time.

Concretely, this document describes my experiences with the MCF5407 Coldfire board. The board was supplied by Gerald Kupris from Motorola to enable us to develop our operating system before our custom hardware will be ready. A lot of thanks for this great opportunity! I also use this document as a personal notebook about implementation-related issues. Thus, do not wonder about the level of detail in some sections.

The the goals of my work are to get MiNT to run on the Coldfire and to try out ideas about operating system architectures. The most of the stuff I did with the board so far were experiments to learn about the Coldfire and are not immediately related to Atari. So please do not wonder about some weird things, described here.

2 Where to get the source code?

All things, I describe within this document are publicly available under the terms of the **GNU General Public Licence Version 2**. I use a Subversion repository for development. Subversion is a revision control system, which is similar to CVS but offers some advanced features (for example, moving and copying of files) that CVS lacks of. Further information about Subversion is

available at `http://subversion.tigris.org`. There is a very detailed description of this tool available at `http://svnbook.red-bean.com`.

My Subversion repository is hosted at `http://os.inf.tu-dresden.de/~nf2/svn`. You can browse the repository using your web browser. You can check out the Coldfire related stuff by using the instruction:

```
svn checkout http://os.inf.tu-dresden.de/~nf2/svn/trunk/acp
```

All revision information, such as log messages, are world-readable, too.

3 Cypress PCI bridge

The MCF5407 evaluation board features a PCI connector. This is a great chance for me to learn about programming PCI devices. My concrete goal was to drive a graphics card. The following sections contain technical stuff that I learned from my experiments or retrieved from various sources (documentation, uClinux kernel, internet).

The memory area from `0xffff0000` to `0xffff3fff` is dedicated to the Cypress PCI bridge. The lower area (`0x0` - `0x1fff`) is the configuration space of the bridge, the higer area (`0x2000` - `0x3fff`) is a window to the PCI address space. The visible content of this window can be defined using the Cypress register `0x460` (see section 3.2).

3.1 Building the PCI address space

Before PCI resources can be accessed, we must map them to the PCI address space, which has nothing to do with the host's address space. We can set all bits and subsequently read the memory base address registers of the PCI device's configuration space to determine the type and size of the provided ressources (this is a standard way which is described in Markus Fichtenbauer's PCI article `http://acp.atari.org/files/pci-bios/index.html`). The lowest bit of an base address register indicates if the entry belongs to an I/O (bit0 = 1) or memory resource (bit0 = 0). The number of the other lower bits which are set to zero provide the information about the size of the resource.

I/O and memory ressources can be mapped to PCI addresses by writing the desired start addresses to their corresponding memory base registers in the PCI configuration block of the PCI device.

Additionally, the access to I/O and memory resources must be enabled by setting the bits 0 and 1 of the DEVCTRL (`0x3`) register in the configuration space.

3.2 PCI window

The Cypress PCI bridge supports three types of PCI transactions which can be performed with the help of the physical base address register (`0x460`) of the Cypress PCI bridge:

configuration transaction The physical base address register must be set to `0x40000006`.

IO transaction The physical base address register must be set to the desired base address of the PCI address space where we mapped the I/O ressource (see section 3.1). Additionally, the lowest two bits must be set to `0x2`.

memory transaction We have to do the same as for I/O transactions except that the lowest two bits of the physical base address register must be set to `0x4`.

After that, the defined configuration, I/O or memory space of the connected PCI card is available at the PCI address space window (at `0xffff2000` - `0xffff3fff`).

3.3 Endian

The Cypress PCI bridge seem to perform a conversion to big-endian. I guess this can be configured using the Local Bus Configuration Register (`0x4fc`) bits 6,8,9 - but I did not manage it to properly switch the conversion off.

4 Matrox 1064

Messing around with graphics cards took a lot of my time. My attempts of initializing such devices range from ground-up driver implementation (following the documentation) through hacking video drivers out of Linux to a video-device-driver environment that is able to use unmodified Linux video-drivers on the Coldfire evaluation board.

The positive side of this soul-destroying task is that I gained a lot of knowledge about PCI and the programming of various graphics cards (ATI Rage, ATI Radeon7500, Matrox1064, MatroxG450).

The following sections present some technical aspects that I stumbled across and found noteworthy. There might be references to source codes I am working on. Please do not get irritated by that.

4.1 Endian crap

The Matrox 1064 features limited big-endian support (activated via bit 31 of the OPTION(`0x40`) register of the PCI configuration space) which must not be be switched on with the Cypress PCI bridge because endian conversion is already be performed by the Cypress chip.

I checked that using the OPMODE and IEN registers of the Matrox card, which I set to `0xffffffff` and subsequentely read it - the zeroed-out bits allow an estimation of the correct byte order.

4.2 Attribute controller register

The Attribute controller registers are funny. They must be accessed 8bit wise. Therefore the endian conversion of the Cypress chip produces total crap. Thus, we have to access the ATTRX(`0x1fc0`) via the address `0x1fc0+3` and the ATTRD(`0x1fc1`) via the address `0xfc1-1`. Secondly, the ATTRX register changes its meaning with every write access. The first access must specify an index followed by a second write access specifying the value to write to the indexed register. Reading the register will always return the current index. The content of a register can be read via the ATTRD register. To reset the toggling of ATTRX we have to read the INSTS1(`0x1fda` - I am not 100% sure about that address) register. Guess how much time I needed to explore that!

4.3 The X-Registers

The X-Registers (`0x3c00 - 0x3c0a`) are programed in a similar way as the attribute controller register. There is an index register PALWTADD(`0x3c00+3`) respectively PALRDADD(`0x3c03-3`) (which seem to have the same meaning) to specify an index. After that, the indexed register can be accessed via X_DATAREG(`0x3c0a-1`) and PALDATA(`0x3c01+1`).

4.3.1 Palette RAM write test

The PALRDADD register is automatically incremented while reading or writing the PALDATA register. In order to enable the palette stuff, we first have to initialise the XMISCCTRL(index `0x1e`) register. The test program looks like this:

```
int i;

/* set PCI address space window */
pci_map_control(0x2000);

/* set up DAC stuff */
v_pci_w8("PALWTADD", PALWTADD, XMISCCTRL);
v_pci_w8("X_DATAREG[XMISCCTRL]", X_DATAREG, 0xff);

v_pci_w8("PALWTADD", PALWTADD, 0);
for (i=0;i<0x12;i++) pci_w8( PALDATA, i);

v_pci_w8("PALWTADD", PALWTADD, 0);
for (i=0;i<0x12;i++) v_pci_r8("PALDATA", PALDATA);
```

4.3.2 VGA sequence register test

The following code tests the VGA sequence registers. The register SEQX(`0x1fc4+3`) is used to specify the index of the desired register while SEQD(`0x1fc5+1`) is used to perform the actual access. I just set all bits of all sequence registers to one and compared the resulting bits that stood zero with the specification.

```
pci_map_control(0x0);
for (i=0;i<5;i++) {
  v_pci_w8("SEQX", SEQX, i);
  v_pci_w8("SEQD", SEQD, 0xff);
}
```

5 Video Device Driver Environment

After a half-successful attempt to hack the Matrox driver code from the Linux-2.2 kernel (after 2-3 days of intensive digging in the driver code I even got the Matrox to display a blue screen) I

decided to give the Device Driver Environment (DDE) approach a go. Instead of modifying the original driver code and inducing new bugs I decided to let the original driver code of Linux-2.4.20 run in an emulation enviroment. A small emulation code layer provides all functions needed to let the driver "think" it is at home in the Linux kernel. The following steps were needed to perform this task:

5.1 Choosing the driver sources

The needed driver source code files had to be identified. I choose the files shown in table 1. Not all of them are actually needed to drive a Mystique but including them simplifies later support for other Matrox models. As you can see, the ported driver code consists of ca. 9000 lines (without comments).

source file	lines of code	notes
g450_pll.c	390	
g450_pll.h	5	
i2c-matroxfb.c	280	the DAC as used on Mystique
matroxfb_DAC1064.c	877	
matroxfb_DAC1064.h	150	
matroxfb_Ti3026.c	675	
matroxfb_Ti3026.h	8	
matroxfb_accel.c	999	
matroxfb_accel.h	9	
matroxfb_base.c	2232	main init code
matroxfb_base.h	702	
matroxfb_crtc2.c	730	
matroxfb_crtc2.h	37	
matroxfb_g450.c	111	
matroxfb_g450.h	9	
matroxfb_maven.c	122	
matroxfb_maven.h	6	
matroxfb_misc.c	830	also core functions
matroxfb_misc.h	17	
total	8965	

Table 1: Used driver source codes from the Linux-2.4.20 kernel.

5.2 Compiling the driver

I had to determine the needed Linux-kernel header files. This was easy:

```
grep -h "^#include" *.c *.h | sort | uniq
```

The more difficult job was to decide which header files of the Linux kernel could be used in their original form and which ones had to be replaced by own implementations or fakes. Table 2 shows an overview over the unchanged and modified/faked header files. This step was - by far - the most

time intensive one. I also had to determine the right defines to choose to produce a driver that actually does something useful (the configuration can be found in `include/linux/autoconf.h`). In result all original C source files compiled happily.

original	replaced by own implementations
include/linux/config.h	include/linux/autoconf.h
include/linux/types.h	include/linux/module.h
include/linux/stddef.h	include/linux/kernel.h
include/linux/posix_types.h	include/linux/console.h
include/linux/fb.h	include/linux/delay.h
include/linux/console_struct.h	include/linux/fs.h
include/linux/list.h	include/linux/i2c.h
include/linux/timer.h	include/linux/init.h
include/linux/devfs_fs.h	include/linux/ioctl.h
include/linux/i2c-algo-bit.h	include/linux/ioport.h
include/linux/errno.h	include/linux/mm.h
include/linux/selection.h	include/linux/pci.h
include/linux/matroxfb.h	include/linux/slab.h
include/linux/pci_ids.h	include/linux/spinlock.h
include/linux/i2c-id.h	include/linux/string.h
include/asm/types.h	include/linux/tty.h
include/asm/posix_types.h	include/linux/vt_buffer.h
include/asm/errno.h	include/linux/interrupt.h
include/asm/ioctl.h	include/linux/kdev_t.h
include/video/fbcon-cfb16.h	include/linux/devfs_fs_kernel.h
include/video/fbcon-cfb2.h	include/linux/prefetch.h
include/video/fbcon-cfb24.h	include/linux/sched.h
include/video/fbcon-cfb32.h	include/asm/io.h
include/video/fbcon-cfb4.h	include/asm/uaccess.h
include/video/fbcon-cfb8.h	include/asm/unaligned.h
	include/asm/semaphore.h
	include/asm/rwsem.h
	include/asm/page.h
	include/asm/param.h
	include/video/fbcon.h

Table 2: Untouched and replaced header files of the Linux kernel

5.3 Linking and the implementation of the driver environment

In order to link an executable binary, all unresolved symbols had to be implemented (or at least faked). At first I created dummy implementations using a `sed` script with linker error messages as input. All dummy implementations can now be found in `mga_dde_dummy.c` . I had to fully implement all essential functions such as PCI device access. Table 3 shows the emulation code files.

source file	lines of code	notes
emu_lx_fbcon.c	19	fake of Linux framebuffer
emu_lx_pci.c	104	Linux PCI interface
mga_dde_dummy.c	197	misc (48) dummy implementations
mga_dde_init.c	40	PCI and Matrox init
mga_dde_init.h	1	
mga_dde_pci.c	39	core PCI for MCF5407
mga_dde_pci.h	7	
mga_dde_pci_debug.c	76	verbose PCI functions
mga_dde_pci_debug.h	6	
total	480	

Table 3: Matrox DDE source files

5.3.1 Changes of the original driver code

The original driver code had to be slightly changed to make it run on the Coldfire board. Due to the DDE approach these changes were very small (nearly non-existent):

- `matroxfb_base.c:` Set the default value of the static variable `noinit` to 1 (original value 0)
- `matroxfb_base.c:` The original code to detect the size of the framebuffer does not work correctly, yet. Thus, I commented out the `goto failVideoIO` in function `initMatrox2` to let the driver proceed with the initialisation procedure even if the memory probing fails.
- `matroxfb_base.h:` I renamed the define `__LITTLE_ENDIAN` to `__LITTLE_ENDIAN_OUTW` to make the driver use the `readb` and `writeb` functions.

5.4 Firing the bullet

After successfull compilation and linking I had to find the proper way to jump into the drivers code. To my very surprise all I had to do was to call the function `matroxfb_init` in `matroxfb_base.c`. When setting the `noinit` variable to `0` the driver even sets a default video mode (640x480 16bit). So there is a huge improvement over the Matrox driver code in Linux-2.2! So now we have screen output on the Coldfire evaluation board - a small step for the world but a huge step for Norman :-)

5.5 Experiments with the ATI Rage driver from Linux-2.4.20

I enhanced the video device driver environment to provide all functionality needed for the aty driver from Linux-2.4.20. The driver starts up and successfully probes my 3D Rage (GT) card. It also initialises a default video mode but I have not managed to display a descent picture on screen.

In the meanwhile I discovered that the power consumption of more recent graphics boards is just too high for the Coldfire evaluation board. I experimented with ATI Radeon, Matrox G450, ATI Rage LX and various other graphics cards, but the Matrox 1064 was the only one with a low-enough power consumption.

5.5.1 Modifications of the Linux driver

- `mach64_ct.c:` return at the begin of function `aty_set_pll_ct`

If the Coldfire executes the function `aty_st_pll` for setting up the PLL_GEN_CNTL register - the program freezes shortly afterwards. Even the break-button of the Coldfire evaluation board refuses to work then.

6 Interrupt-based driver for Wacom Artpad

The MCF5407 evaluation board features two serial ports — one is used for debugging and the other is available for custom use. Since I needed an input device for my further work, I decided to implement a driver for a Wacom Artpad, that lied around somewhere. The nice thing about the Wacom Artpad is its simple communication protocol. Since the driver had to work interrupt-driven, I had to discover how to program interrupts on the Coldfire, which was a fairly easy walk. The initialitation of the serial interfaces is well described by the source code of the Motorola dBUG monitor and the uCLinux source code.

7 DOpE window server

I ported the DOpE window system to the Coldfire board. DOpE is a window system that I develop for the real-time operating system DROPS. It is my current research project at the university. Information about DOpE are available on my private website:

`http://os.inf.tu-dresden.de/~nf2`

Basically, I needed the port of DOpE to the Coldfire for ego-reasons ;-)

8 MicroAPL CF68K Emulation Library

One fundamental thing for running Atari applications on the Coldfire is a working emulation of mk68k instructions that are not present in the Coldfire. MicroAPL provides a solution for this problem: The cf68klib is a virtual machine that can either execute usermode 68k instructions only, or virtualizes a whole 68k CPU including supervisor mode! The exception vector table and all interrupts are also virualized by cf68klib such that a whole operating system could be run within the virtual machine. After a weekend of studdying the documentation and experimenting, I successfully started up the virtual machine and executed instructions that are not natively implemented in the Coldfire CPU! This is a big step for getting MiNT to run on this platform.

9 Porting MiNT to the Coldfire

There are two possible ways of how to port the FreeMiNT kernel to the Coldfire:

- We can port the kernel sources to a new target system and compile it using the native gcc for the Coldfire. This needs a lot of knowledge of the MiNT kernel sources and a lot of porting work. Of course, the Coldfire-branch of the kernel must be maintained to be up-to-date with the main brach of MiNT.

- We can try to use a MiNT kernel image, compiled for the original Atari and execute it inside the cf68klib. For this approach, we need to find out about all the things, MiNT relies on — such as the used TOS system calls and timers and exception vectors — and must provide these pre-requisites by our underlying system. The drawback of this approach is possible performance-loss because some instructions of the MiNT kernel image must be emulated by the cf68klib. The big advantage is, that we were always up-to-date with the current MiNT kernel as we just need the binary kernel image (mint.prg). The performance can be improved later by using the Coldfire-gcc for compiling the C-parts of MiNT and changing the assembly routines not to use non-native Coldfire instructions anymore.

I decided to give the second approach a go.

For executing the MiNT kernel, we first must relocate the TOS executable — TOS executables can be loaded at any address in memory. Thus, when absolute references within the executable are used by the program these references must be adapted to the actual position of the executable in memory. I just deploy the relocation routine that I used in many demos and it works well on the Coldfire. Actually, even the relocation routine runs inside the cf68klib virtual machine because it uses some non-Coldfire mc68k instructions.

9.1 Providing an execution environment to MiNT

The MiNT kernel image is a normal TOS executable and thus, it relies on the environment, provided by TOS — such as a valid basepage containing all information about the segments, environment string and arguments of the program. I implemened a basepage initialization routine. The address of the basepage is then passed to MiNT. Now, with the basepage provided we can jump into the MiNT kernel.

We must let MiNT think to be at home at TOS by providing the TOS functions that are actually needed by MiNT. The first things we see from the MiNT kernel are some attempts to call GEMDOS functions via the `trap #1` instruction. Thus, we must install a handler for such `trap` calls. The MicroAPL cf68klib allows us to use user-level trap handlers that look exactly like MC68k trap handlers. The MC68k as well as the Coldfire4 feature a table of exception vectors at an address, which is defined via the `vbr` register. Initially, this register is set to zero. That means, MiNT expects the exception vectors to reside just at the beginning of the address space. I do not want to change MiNT so I move the real exception vector table of the Coldfire to another location (`0x01000000`) and use the address space from `0x0` to `0x3ff` for storing the virtual exception vector table of the virtual MC68k cpu. For serving the needed GEMDOS calls, a custom (user-level) exception handler for `trap #1` must be installed at address `0x84`.

In result we get some nice GEMDOS traps such as Mshrink, Fsfirst and Cconws. Yes! There are the first boot messages of MiNT!

9.2 Trap handler with gcc

Sadly, the way of how arguments are passed to TOS via the known TOS syscall bindings is not compatible with the function calling conventions used by gcc. TOS programs use to push 16bit and 32bit arguments onto the stack without any gaps in-between. In contrast, gcc alignes all arguments to 32bit-even addresses by leaving 2-byte gaps between two 16bit arguments. According to Peter Barada, the maintainer of the Coldfire-gcc, there is no way to prevent the 4-byte stack alignment of gcc.

Thus, we need to convert the syscall frame to a valid gcc call frame. This conversion is called marshalling (slightly inspired by the terminology used for IDL compilers). For this, I created a small tool that generates the needed conversion code for specified syscall signatures.

In the meanwhile, Frank Naumann told me a way of how to work around the 32-bit stack alignment. All one need to do is to pass the parameters not directly but using a structure. Thus, MiNT applies its alignment policy for aligning struct elements when constructing the structure in memory. The default policy for that is 16bit alignment - exactly what we need. So we could implement the trap dispatch functions simply by using

```
struct setexc_args { short vecnum, long handler; };
static long setexc(struct setexc_args args) {
  ...
}
```

instead of

```
static long setexc(short vecnum, long handler) {
  ...
}
```

Anyway, I kept using the generated stubs because I do not want the kernel to work directly on the user stack pointer. In this case any malicious user program could crash the system by setting its stack pointer to an illegal address before calling the kernel.

9.3 Telling MiNT some sweet lies

During startup, MiNT determines on which kind of hardware platform and TOS-version it runs and takes this information in account for its later operation. To make MiNT behave as we like it, we tell MiNT some (bogus) information. Later, we can build-in Coldfire-specific support into the MiNT kernel. One information source, MiNT takes into account, is the *cookie jar*. This is a central point in the system that provides so-called cookies_, which hold information for certain system properties and interfaces to software extensions of the operating system. On the Atari the cookie jar is created by TOS and filled with information about the machine type. In our execution environment we also create such a cookie jar and supply the following cookies:

COOKIE__MCH This is the machine type. We define its value to be `0x30000`L, which represents “Falcon hardware”.

COOKIE__VDO This cookie specifies the installed video hardware. We set this cookie to the value `0x30000`L, which means “Falcon video”. This way, MiNT requests the size of the frame buffer via the XBIOS function `VgetSize` during its memory initialization. Otherwise, it would use line-A stuff, which I really do not like to care about ;-)

COOKIE_PMMU This cookie holds the information about the availability of a PMMU (Programable Memory Management Unit). Since the Coldfire does not feature a PMMU, we need to make MiNT to keep its hands away from PMMU related things and thus, set this cookie to `0x0`L.

9.4 Timer initialization

One thing, MiNT relies on, is a 200Hz timer that operates on the exception vector `0x114`. On the Atari, this timer is called Timer-C. On the Coldfire, there is no such Timer-C, but two internal timers (Timer-0 and Timer-1). The initialization of these timers is not compatible with Atari-ST hardware. On Atari machines, TOS initializes all timers and MiNT just installs itself into the corresponding exception vector. Thanks to that, MiNT does not perform any hardware-specific timer initialization and gives us the chance to do this initialization in advance within our execution environment. One tricky thing is, that the Coldfire does not allow us to use the exception vector `0x114` for its internal timer (the internal timer generates an autovector interrupt, for which exists a dedicated range of exception vectors). I solved this problem by installing an exception routine at the actually used exception vector `0x74` that just jumps right to the routine, `0x114` is pointing at:

```
timer_200hz_vec:
  bsr    timer_200hz_start    | init next timer interrupt
  addq.l #1,JIFFIES_200HZ     | increment jiffies value
  move.l TIMER_C_VEC,-(%sp)   | branch to timer-c exception handler
  rts
```

The subroutine `timer_200hz_start` just schedules the next timer interrupt. `JIFFIES_200HZ` is actually the address `0x4ba`. This is a TOS system variable that counts the number of Timer-C interrupts. MiNT relies on this system variable. Thus, we need to feed it here. The `TIMER_C_VEC` is the exception vector `0x114`, which is not in use by the Coldfire but only holds the logical Timer-C exception vector. From MiNT's viewpoint, Timer-C just works as normal.

Now, with a timer provided by us, MiNT passes its "calibrate delay loop" successfully. Argh! What do we see there? 2.54 BogoMIPS? This was really not the performance, I expected! In fact, it is only half as fast as a Falcon. Instead of whining, I checked the cache of the Coldfire and found it switched off. After activating the cache we get 151.55 BogoMIPS. This was the value, I expected from the 150Mhz Coldfire processor. In fact, the cf68klib does not affect this value because the benchmarking function does not use any instructions, that must be emulated.

9.5 Character output via BIOS

At the very first stage of MiNT's startup, it uses GEMDOS for textual ouput. After it takes over the GEMDOS, BIOS and XBIOS traps, however, it switches over to BIOS-based textual output. In fact, it does not use BIOS traps for printing characters (one trap exception per character would be damn slow) but it uses the vectors `xcostat` and `xconout` as shortcuts. Thus, MiNT just calls the routines installed at these vectors via a normal branch instruction. This way quite tricky to find out ;-)

9.6 Keyboard input

Currently, MiNT does not feature a custom keyboard driver but relies on the BIOS for keyboard input. This is quite nice for us because it gives us the opportunity to easily emulate a keyboard by our custom BIOS routines that can use the serial terminal as a virtual keyboard (the Coldfire evaluation board does not feature a keyboard connector).

Keyboard events are passed from the BIOS to MiNT via a ring-buffer, called `IOREC`. MiNT requests the location of this ring-buffer by using the XBIOS function `Iorec`. Every 20 miliseconds (on each VBL interrupt), MiNT polls the ring-buffer for new keyboard events. In our execution environment we install an interrupt handler for the serial interface. The interrupt handler supplies all incoming data to the ring-buffer, shared with MiNT. Each entry of the event buffer has the size of 4 bytes. We can submit the incoming ASCII values directly to the buffer by setting the first three bytes to zero and the 4th byte to the ASCII value. This way, we even do not need to care about the conversion between ASCII and keycodes - really cool!

9.7 TOS services that are needed by MiNT

The MiNT kernel relies on a number of TOS system variables and TOS system calls during its bootup.

9.7.1 System variables

TOS uses the memory area from `0x400` to `0x600` for storing OS specific variables that are (more or less) documented. I identified the following system variables to be used by MiNT during the startup. There may be more variables needed by different subsystems of MiNT and TOS applications.

_sysbase (`0x4f2`) Pointer to a structure that describes the installed operating system (TOS). MiNT determines the installed TOS version by reading the `os_version` member of this structure. It also uses the `pkbshift` member. In our execution environment we install an os_header structure and set the `os_version` to `0x102` to let MiNT think, it is dealing with TOS version 1.02.

xcostat (`0x55e + 0x8`) The eight variables at `0x55e` are pointers to BIOS `xcostat` routines for the different BIOS devices. MiNT uses the device number 2 for its textual output of the console. Thus, we need to install a handler at `0x55e` + `0x8`.

xconout (`0x57e + 0x8`) Similar to `xcostat`, the eight variables at `0x57e` point to corresponding handlers for the character output to the different BIOS device numbers. We need to install an output handler for device 2.

getbpb (`0x472`) This is a vector to a function that returns the BIOS parameter block. MiNT executes this routine. Thus, we need to install at least a dummy routine there.

_longframe (`0x59e`) This variable describes the format of the exception stack frame. It is evaluated by the Timer-C handler of MiNT. We just have to make sure that this variable is set to zero.

_timr_ms (`0x442`) This value describes the period of the VBL timer interrupt. Normally, it is set to 20 (20 miliseconds) - so we do the same. This system variable is used by MiNT's VBL handler to schedule context switches.

_p_cookies This variable contains the pointer to the cookie jar. As described in section 9.3, we provide a cookie jar to MiNT.

_hz_200 (0x4ba) This variable is also called *jiffies counter*. Its value is incremented on each Timer-C interrupt. MiNT does not increment this value by itself but relies on TOS to increment this variable.

9.7.2 System calls

In the execution environment, we handle trap 1 (GEMDOS), trap 2 (GEM), trap 13 (BIOS) and trap 14 (XBIOS). By default, we do nothing and just return 0. Anyhow, some system calls must be dispatched by the execution environment. This section is a brief summary of the system calls that must be handled specifically to make MiNT behave nicely.

GEMDOS calls

mxalloc Mxalloc is used by MiNT to obtain its memory from TOS. MiNT takes all memory it can get by subsequentially calling this function. All received memory blocks are then handled over to MiNT's memory manager. We only return one chunk of ST-RAM memory.

cconws This function is used during the first stage of startup for printing the very first boot messages. We just forward all characters to the serial connection.

tgettime Tgettime is used during the startup for waiting for the users input. We just return a counter value that is incremented on each call to make MiNT proceed with its startup.

super This function is used by MiNT for entering the supervisor mode and thus, to take over control over the machine. Ok, so be it.

BIOS calls

bconout This function is used by MiNT for its textual output after the very first startup stage is finished. We just forward all characters to the serial console.

XBIOS calls

supexec This XBIOS call is used to execute one subroutine in supervisor mode. MiNT uses `Supexec` to perform the CPU detection.

bconmap We just return a -1 to make MiNT believe that we do not feature any `Bconmap` facility.

keytbl This function returns a pointer to a key table structure. This structure contains mapping tables from hardware keycodes to ASCII values. Right now, the tables are bogus but, at least, they do exist.

iorec `Iorec` is called by MiNT to obtain the location of the keyboard input ring buffer as described in section 9.6.

GEM calls

appl_init MiNT calls `appl_init` just to make sure, that GEM is not already present and complains in the other case.

9.8 Modifications of the MiNT kernel

Thanks to the approach of the execution environment, I only needed some very minor modifications of the MiNT kernel. Actually, I had to modify only two lines of code. ;-)

9.8.1 CPU detection

Although the cf68klib emulates an MC68030 CPU, MiNT detects an MC68040 for some reason. I did not examine this issue any further and set the `mcpu` variable manualy to the value 30 (which represents a MC68030). (function `_getmch` in file `arch/init_mach.c`) Of course, the obvious way to get this right is to implement a CPU detection routine for the Coldfire.

9.8.2 Incompatibility of the `div` instruction

The division instruction of the Coldfire is not fully compatible with the MC68K division instruction but does not make the Coldfire to trap. Thus, the cf68klib has essentially no chance to emulate this instruction properly. This problem occured only once within the function `PUTL` in `libkern/vsprintf.c`, which is called by sprintf. The function looks like this:

```
PUTL (char *p, long *cnt, ulong u,
      int base, int width, int fill_char) {
    char obuf[32];
    char *t = obuf;
    long put = 0;

    do {
       *t++ = "0123456789ABCDEF"[u % base];
       u /= base;
       width--;
    } while (u > 0);
```

With each iteration, the value `u` is divided by the `base` and then used as abort criteria for the while loop. For some reason, the abort criteria is never reached. This problem does not occur when we cast `u` to a signed long when dividing:

```
       (long)u /= base;
```

10 Various other things

During my life together with the Coldfire evaluation board I did a lot of experiments, which I did not described in detail. Among these things were the attempt to port the MiNT kernel to run natively on the Coldfire CPU, programming the hardware acceleration features of the Matrox 1064, building a toolchain for gcc-3.3.2 and creating a build system for my Coldfire developments based on GNU make.

11 The next things to do

This is roughly a list of my next steps on the path to a proper TOS- compatible OS on the Coldfire:

- Implementing a pseudo block device that works on a RAM disk image.
- Porting a VDI and AES to the system — this point is still a bit unclear to me. As for now I am not sure about the options I have (fVDI? XAAES?). Help of other developers would be greatly appreciated.

Once I have MiNT running at a usable degree, I will try out some OS architectural ideas leading to the direction of a microkernel (similar to L4). My vision is to have a microkernel-based multi-server operating system running with MiNT as single-server side-by-side. (quite similar to L4Linux running on L4).

12 Document history

changes at 2004-03-14 *fixed references, added table of contents*

changes at 2004-06-13 *added section 9*

changes at 2004-06-14 *added section 2*

changes at 2004-07-10 *complemented section 9.2 with Frank's solution, added sections 9.3, 9.4, 9.5, 9.6, 9.7 and 9.8*