Porting NetBSD/evbarm to the Arcom Viper

Antti Kantee
<pooka@cs.hut.fi>

Helsinki University of Technology

ABSTRACT

NetBSD is best and foremost known for its portability. This paper examines that claim in the light of porting NetBSD to an ARM XScale-based single-board computer. The paper starts with a general discussion on NetBSD code organization and how the code is divided into parts dependent and independent of the underlying hardware. After this the porting effort is investigated in more detail, outlining what needs to be done to add support for new hardware to NetBSD/evbarm and describing the problems and respective solutions in the effort.

1. Introduction

The definition of *port* in NetBSD is a very loose one. Sometimes a port consists of only a single type of machine with self-contained CPU support, such as *NetBSD/pc532*, sometimes it can consists of a single machine with "outsourced" CPU support such as *NetBSD/shark*, or sometimes it is simply a collection of hardware that seems to fit nicely under a common umbrella, say *NetBSD/hpcsh*. In the good old days all machines under one port used to be able to boot a common kernel and use autodetection to figure out what kind of hardware was available, but in modern times things are different. In "consumer hardware", such as PCs or Macs the ability to have a single distributed kernel for all machines is still important, but in ports featuring embedded systems, such as *evbarm*, a special-purpose kernel for each machine is acceptable.

Adding support for a completely new CPU has been discussed in the AMD64 porting effort [1] and a similar effort of adding machine support where CPU support exists already has been documented in adding support for the JavaStation to NetBSD/sparc [2]. This paper's contribution, in addition to being full of (non-)amusing anecdotes, aims to be to explain the porting effort and outline the involved steps in terms which are hopefully understandable to an audience without experience in kernel hacking.

It should be noted that the discussion is quite specific at some parts, and can be guaranteed to hold only for the evbarm port of NetBSD, and even there sometimes only for the Viper hardware support. Other ports have other conventions as dictated by the hardware and do things differently. Some parts do apply to NetBSD in general, but no attempt is made to separate the specifics from the generics.

The hardware

The Viper is a single-board computer built around the PXA255 Intel XScale™ RISC processor. Features hardware include a CompactFLASH socket, TFT connector, audio device, serial ports, USB, 10/100BaseTX Ethernet, PC/104 expansions bus and the usual set of GPIO pins. However, at this date NetBSD supports only the necessary hardware for bringing the system up to multiuser in an NFS-root configuration. As is typical for an ARM-based system, there are no fans or other noise-generating components involved. In other words, for home use the system would make a nice MP3-player, even though the physical size is quite large by modern standards. Of course professional use is another completely different story.

2. NetBSD ARM Code

2.1. NetBSD/evbarm

The NetBSD port for ARM evaluation boards is quite simply a collection of mostly indepent hardware support for development and prototyping boards which feature some version of the ARM processor[1]. These independent pieces of support code consist of:

- low-level startup. This code is written in the assembly language and takes care of setting up an acceptable memory mapping for the rest of the bootstrapping process.
- machine-dependent C-language initialization routines. It is the responsibility of these routines to set up the console, initialize the memory management information in the CPU-specific pmap and machine-independent UVM ready for prime-time, coerce the CPU into the mode we want it to operate in and initialize machine-dependent vectors.
- device driver frontends. NetBSD has most hardware support readily available in machine-independent format. Only a small amount of glue code is required to attach the MI driver to specific bus behind which the device (or a bus itself) is sitting.

In addition to the machine-specific implementations, an important part of the port is the shared ARM code located under *sys/arch/arm*. It is this shared code that enables to add support for a new machine with relatively minor effort.

3. The Porting Effort

All efforts for writing support for a specific platform consist more or less of the following tasks:

- Locating documentation for the hardware, reading the documentation and understanding the documentation. As usual, since time is of the essence, there is a great danger of trying skimp at this stage. It will have consequences later on[2].
- Creating a kernel configuration file, and if required the necessary auxiliary files to match the set of hardware that should be supported.
- "Filling in the blanks", i.e. writing the necessary glue code where required.
- Setting up the development environment. This includes building a cross-compiling toolchain, setting up a place where the system can boot from (usually over network), cross-compiling the target system kernel and userland and configuring the system firmware to fetch and execute code from the desired location. These actions will not be discussed in this paper any further.

3.1. Documentation

The documentation for the Viper consists of documentation which specifically deals with the hardware [4], generic documentation on the ARM ISA [3] and finally documentation dealing with the XScale [5] processor family. Contrary to the usual situation with modern hardware, the above mentioned documentation is readily available from the Internet without any need for NDAs or other lawyerly trickeries.

In addition, documentation for various chips on the evaluation board can usually be found from the manufacturers of those chips. A popular trick for accomplishing this is to punch the chip number into GooˆHˆHˆHa search engine and see what happens. Usually the documentation will present itself.

Finally, it is a good idea to know how the firmware works to get a kernel loaded and the kernel execution under way. The Viper features RedBoot firmware, for which documentation [6] is available from the Internet. In light of the porting effort it is also nice to provide the relevant information for users in the NetBSD Installation Guide. This way potential users have a consistent set of sources and matching documentation available from one source and do not have to go hunting around the Internet for various bits and pieces to figure out how to get the system up and running.

[1] It can be likened to a support shelter for various homeless pieces of hardware with no other place to go ... if that is a comparison I am allowed to make.

[2] For example, if your question is answered with the words: "read ARM ARM [3] Chapter 1.1.1.", you know you should have read the documentation more carefully. It is left as an exercise to the reader to figure out if this is a purely fictional example.

Entry to NetBSD

```
/*
 * You are standing at the gate to NetBSD. --More--
 * Unspeakable cruelty and harm lurk down there. --More--
 * Are you sure you want to enter?
 */
mov     pc, r8                          /* So be it */
```

3.2. Writing the Configuration File

BSD systems decide what to include in kernels and how to probe the device-tree with the help of a configuration file [7,8]. A good idea for creating a configuration file for some specific machine is to copy a similar configuration file and modify that. In this case the definition of "similar" was "another board based on PXA255".

The only piece of information specific to the Viper in the current supported hardware configuration is the Ethernet controller:

```
# SMC91C111 Ethernet
sm0 at pxaip0 addr 0x08000300 intr 0
```

The above tells the system autoconfiguration that *sm0* can be probed as a child of the *pxaip0* bus at address *0x08000300* and that the device will interrupt at interrupt level *0*. This information will be used by the driver. Note that in this special case, since, as we soon shall see, the networking driver frontend is written specifically for the Viper, providing the address and interrupt in the config file is not strictly necessary. The information could be hardcoded into the driver as well. However, since using the configuration file to contain configuration information is the correct approach as opposed to hardcoding the information into various drivers all around the source tree, we use that approach.

In addition to the configuration file itself, the following files need to be modified:

- *conf/files.machname*
- *conf/mk.machname*
- *conf/std.machname*

The configuration file itself makes sure that versions of the above files specific for this hardware are used by including *std.viper*.

files.viper

This file specifies all the devices and source files specific to the Viper. Currently it tells the system to include the file *viper_machdep.c* in the kernel and, as presented below, informs of the possibility to attach the *sm* driver at the *pxaip* bus:

```
# SMSC LAN91C111
attach sm at pxaip with sm_pxaip
file arch/evbarm/viper/if_sm_pxaip.c \
                                sm_pxaip
```

The information about the *sm* driver is a system-level counterpart of what was written in the configuration file and keeps the config-file author from having to know details about the actual implementation of the driver.

mk.viper

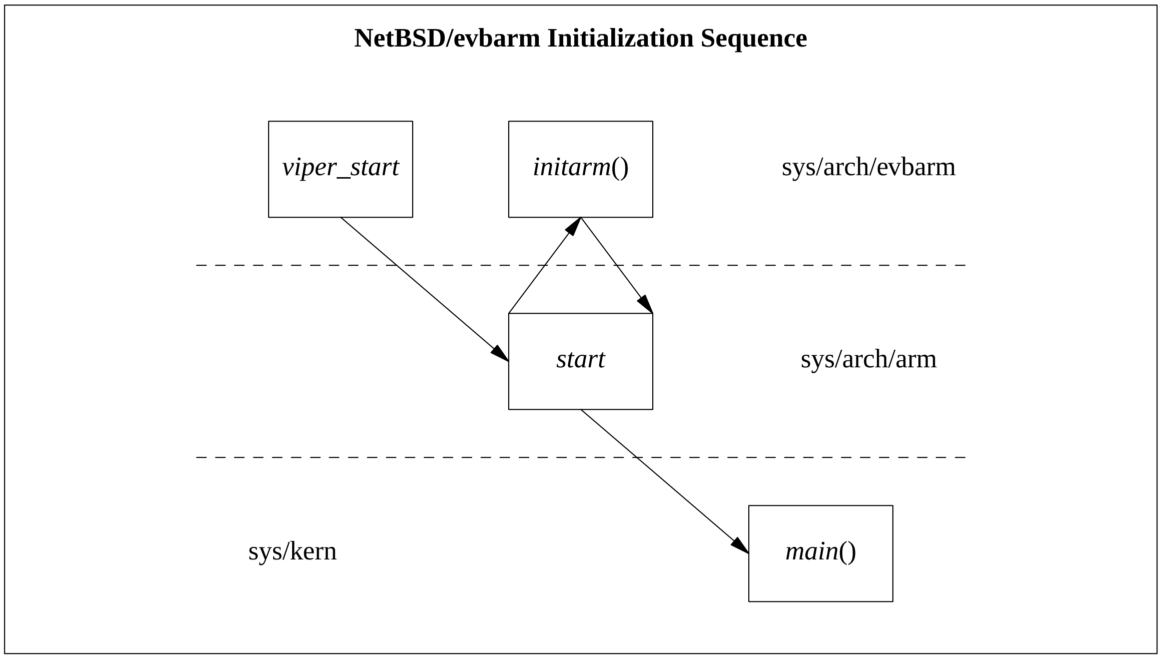
As the name suggests, this file deals with the viles of the kernel Makefile framework. The current significance is specifying the kernel base address and object format copy used in linking the kernel image. In addition, the first object file linked into the kernel image is specified here. This object file should, at the beginning of it, contain the kernel entry point.

std.viper

The file contains the standard config options you cannot live (or run) without. These include for example support for execution of *ELF* binaries and the config option for specifying that the system uses the PXA-specific interrupt handling implementation.

3.3. Low-Level Startup Code (*viper_start.S*)

For a programmer with previous experience in programming XScale CPUs, writing the low-level code is mostly a walk in the park. In case you are not familiar with ARM assembly, the MMU and such, it is more involving, but only slightly. By no means can it be likened to black magic.



The level of ARM assembly programming prowess required for writing the first stage initialization code is not very spectacular. One has to have the knowledge on how to read and write memory, do simple arithmetic-logical instructions, and how to write loops and jumps. In addition one needs to know how to talk to the the MMU. This level of skill, of course, results in non-optimal code, but a microsecond more to the boottime and a few hundred bytes more code are probably not an issue. If they are, feel free to increase your skill levels beyond the description given above.

Most evbarm machines accomplish the low-level init by first turning the MMU off, jumping to the physical memory address of the loaded kernel, and then proceeding to configure the memory map and other CPU information, and finally turning the MMU back on. This is a good way to proceed, since there is no need to worry about protection levels, cache flushing and other complex issues related to the MMU. However, this approach did not work for some reason on the Viper, no matter how careful one tried to be.

The problem, or more specifically the effect, was the system going totally dead after turning the MMU off. This might have been a simple bug in the physical address calculation routine, which made the kernel jump to hyperspace instead of jumping to the physical address of the code as it was supposed to. Or it might have been some really complex interaction between multiple variables. Nevertheless, the author was left perplexed after around 200 attempts to work around the problem. Finally, it was decided to do the initialization with the MMU on.

The decision of doing the initialization with the MMU active morphed the steps of low-level init into the following:

- Build an identity VA mapping for VA == PA. It is easier to copypaste code[3] from other sources later on if this assumption holds.
- Map the system physical memory to `0xc0000000` and up. For 64MB of memory present on the Viper, this spells mapping memory up to `0xc4000000`.
- Map devices used during the bootstrap process.
- Relocate the kernel to the location we want it at.

Building mappings

The XScale MMU deals with memory mappings on multiple levels. The first level, or L1, page table entries are 1MB, or 0x100000 bytes, in size while L2 entries describe a single page of memory. This standard multilevel approach makes it possible to map large chunks of memory easily and with little overhead (both space and coding effort) while still allowing for a

[3] Yes, so we all sin every now and then.

fine-grain per-page description. The page table is described on the XScale by a continuous set of memory, with the first four bytes in the table describing L1-sized chunk of memory at at virtual address `0x0`, the second the virtual address `0x100000`, and so forth. For L1 mappings, each entry contains the significant bits of the physical address in addition to the protection levels of that particular table entry. For L2, the physical memory address of the relevant L2 descriptors is indicated. The mappings are modified by modifying the memory contents of the correct offset in the page table. And of course it is possible to instruct the MMU to switch to use a different mapping table located at a different address.

Per the NetBSD convention, we load the kernel at `0xc02000000`. Per the same convention, physical memory is also mapped starting from `0xc0000000` onward. This leaves room to allocate bootstrap memory from the two megabytes before the kernel load address. Per a different conversion, physical memory is assumed to have an identity mapping during the C-level bootstrap process. So we map also from the virtual address `0xa0000000` onward to physical memory.

To be able to use various devices during the startup sequence, or at any point during execution for that matter, the devices need to be mapped into the memory so that accessing them is possible. During later stages of execution mapping devices in and out is fairly easy, since we have C-level convenience functions available for managing the mappings, but during the very first steps we must manually build the necessary mappings in assembly. Technically it would be enough to map a few simple bytes of memory window to operate the devices. However, since dealing with them is a fuss in assembly, we map an entire L1 entry for each device[4].

Kernel relocation

Kernel relocation sounds much more difficult than it actually is. It simply involves just size calculation and a load-store loop. The kernel image consists basically of text followed by data. We know that the kernel entry point is at the beginning of the kernel[5] and we also know that the end of the data segment is marked by the symbol `_edata`. The size of the kernel image for copying is a simple operation: end_address - start_address. This is rounded up to the next four bytes, since the load-store loop is done one word at a time.

And now for something C-pish

After having done all machine-specific initialization, *viper_start* calls *start* located under *sys/arch/arm*. This routine is responsible for setting up an initial stack for running C code and calling *initarm*(), which once again is a routine specific to the Viper.

3.4. Low-Level debugging

Another annoying part of writing the low-level init is that no console is available, and one must resort to various forms of trickery for debugging. A popular approach is to blink LEDs attached to the system to give hints on where and how the code is executing. The only downside to this method is that one must be bothered to connect the LEDs to some available ports and also to figure out a way to toggle those ports.

The RedBoot firmware is nice enough to contain a debugger, namely gdb over serial. This means that instead of taping LEDs to the back of the board, most cases can be solved by simply examining the mess at hand in gdb. Because we are not yet running C code, the mess will simply present itself in assembly language. A useful trick is to load idle machine registers with relevant information on system state and upon a crash (or explicit *bkpt* instruction) examine the system state with the gdb command `info registers`.

It should also be noted, for sake of being complete, that you need to run a version of gdb compiled for the target system, not the host system. Luckily NetBSD makes this easy, and you can build a cross-gdb simply by giving the argument `-V MKCROSSGDB=yes` to *build.sh* [9] when building the cross toolchain used for development.

3.5. *initarm*()

After a long struggle with the assembly language low-level init, the platform-dependent C initialization code is almost a piece of cake to

[4] And feel slightly guilty about wasting many megabytes of virtual address space... well, no, not really. We'd much rather feel guilty for eating creme brulee with chocolate sabayon.

[5] That's what we specified in *mk.viper*.

handle.

First of all, the device mapping we generated in *start* must be described to the C code so that the correct mapping can be built later on. This is done by building a table of *pmap_devmap* structures, each entry containing the virtual and physical address, mapping size, protection level and caching attributes; the contents are similar to what we used in the assembly code for building the device mappings.

Second, we want to initialize the console device so that we finally gain the ability to do debugging-by-printf. After having mapped the console serial port to memory, attaching the console is a job of calling the machine independent *comcnattach*() to specify where the console port is found and what its parameters are. If all goes well, the console will work after that. And lo, there was *printf*.

After this, *initarm*() performs memory management initialization. Since that code was refactored in[6], discussion on it will be skipped. The code looks straightforward, but since I am not the author of that particular code, I cannot comment on how straightforward it was to write and to get into place originally.

As its final act, *initarm*() returns the new stack pointer, which is put into use by *start*. Finally, *start* calls the machine independent kernel entry point: *main*(), which takes care of the rest of initialization tasks, such as device autoconfiguration based on information in the config file.

3.6. Networking

Although it would have been entirely possible to include a root disk image in the kernel and therefore accomplish a "full" boot, a system these days is not really usable without networking, so the development direction was adding networking support.

The networking chip in the Viper is an extremely common chip designated "SMC91C111". Support was merely a question of a frontend for the existing driver in *sys/dev/ic/smc91xx.c*.

The frontend driver is divided into two interface functions: a *match* function which tells the system autoconfiguration if the probed device is present and an *attach* function which readies the driver (but not necessarily the device itself) for operation.

Match & Attach

The easiest way to write a match function for hardware that is always present is to return success in all cases. This is also the common lazy idiom for writing *match*-functions for devices which are non-detachable and somehow non-detachably integrated into a certain system. However, this is possible of course only if *match* is called for only the device in question. Since the PXA interrupt controller probes through all the devices under *pxaip*, matching every caller as the network device is not a good idea. The interrupt controller did not work very well as the NIC, but this was fixed making the match check against the device physical address before deciding if it was the right driver for the job.

The attach routine contains three parts. First of all, *bus_space_map*() is called for the device to generate a *bus_space_handle* for it. Second, *smc91cxx_intr*() from the MI driver is established as the interrupt handler by calling *pxa2x0_gpio_intr_establish*(). Finally, the frontend is attached to the MI driver by calling *smc91cxx_attach*().

The Trouble with Tuples

As fate usually has it, even though the driver should, according to theory, have been working flawlessly after writing the frontend, NFS mounting root was still not successful. After running tcpdump the problem was revealed: `ff:45:67:64:2e:ef > 01:ff:ff:ff:ff:ff`. The query was not properly broadcast to the Ethernet broadcast address because some mysterious "01" had managed to mangle itself into the middle of the packet. But the mysteriousness of the mystery became much more fathomable once the on-wire Ethernet protocol was recalled after some hours of banging ones head against The Wall: on-wire the destination comes before the source, tcpdump just decides to print them the other way around.

This revelation lead to careful analysis of the Ethernet chip documentation. The length of an Ethernet frame is specified to the hardware by writing the frame's length in 16 bits to the chip prior to writing the packet contents. The original MI part of the driver did this write in two one-

[6] Which is just a really fancy way of saying that it was copypasted.

byte pieces. However, due to the 16bit bus on the Viper, the chip got two 16bit values instead of two 8bit values containing the lower and upper bytes. The chip proceeded to interpret the high-order byte of the length as data bound for the network and a chaos was ready to ensue. Changing from two wrong writes to one right write fixed the problem.

Buffer Space, The Final Frontier

Having root on NFS places a fair deal of stress on the networking subsystem right after mountroot. This is because NFS tries to send maximal size UDP[7] packets to transport the binaries to the client system.

Some Ethernet chips have only a tiny amount of buffer space available, such as the 8kB specimen on the Viper. If the buffer is filled before the operating system has a chance to offload frames from the Ethernet chip into operating system memory (there is no DMA), the nature of Ethernet is to lose frames. Getting a full default size 8kB UDP packet through up to the application level without dropping a single one of the Ethernet frames that make up the fragments is something closely akin to winning the lottery[8]: the timings are really critical. If a single frame is dropped, the UDP packet can never be reassembled and therefore the data does not reach its destination. NFS deals with this by requesting the same information again, but it is very likely that the resent data will not reach its destination either.

A simple workaround for the problem is to set the NFS read and write sizes to a low default using `options NFS_BOOT_RWSIZE=1024` in the kernel configuration. Since 1024 bytes is less than the Ethernet frame size (with tolerance for header overhead), dropped UDP fragments are not a problem. The real solution is immensely more complex involving a soldering iron and some really steady handywork.

4. Conclusions

For someone, namely the author, who had no previous experience in working with the evbarm port and only a limited number of encounters with the ARM CPU and no real background in writing ARM assembly, porting NetBSD/evbarm to a new platform proved to be extremely easy. Out-of-the-box cross-buildability proved its usefulness once again, since a toolchain for development was available after typing in one command. One of the really big surprises was that after fixing all the bugs in *initarm*(), the kernel managed to bootstrap itself all the way up to *mountroot*() without a single error. The battle preparations for weeding through Viper-induced bugs and glitches in the machine-independent code were completely unnecessary.

[7] Assuming we are using UDP as the transport in NFS, of course.

[8] But, if I could choose, I would rather choose to the win the lottery.

Acknowledgements

This paper and the code imported to NetBSD was reviewed by Steve Woodford. The Viper hardware for development was provided by Data Respons OY.

References

1. Frank van der Linden, *Porting NetBSD to the AMD x86-64: a case study in OS portability,* pp. 1-10, Proceedings of BSDCon ’02 (2002).
2. Valeriy Ushakov, *Porting NetBSD to JavaStation-NC,* pp. 161-165, Proceedings BSDCon Europe 2002 (2002).
3. *ARM Architecture Reference Manual,* Addison Wesley. ISBN 0-201-73719-1.
4. Arcom, *Viper Technical Manual.*
5. *Intel XScale(R) Microarchitecture for the PXA255 Processor User Manual* (March, 2003). Order number 278796.
6. eCosCentric Limited and Red Hat, Inc., *RedBoot User’s Guide.*
7. Chris Torek, *Device Configuration in 4.4BSD* (December 17, 1992).
8. *config -- the autoconfiguration framework “device definition” language.* NetBSD Kernel Developer’s Manual.
9. Matthew Green and Luke Mewburn, *build.sh: Cross-building NetBSD*, pp. 47-56, Proceedings of BSDCon ’03 (2003).

Appendix 1: Kernel bootlog

```
RedBoot> load -r -b 0x2000000 netbsd.kaesi
Using default protocol (TFTP)
Raw file loaded 0x02000000-0x0224b1eb, assumed entry at 0x02000000
RedBoot> go

NetBSD/evbarm (viper) booting ...
initarm: Configuring system ...
init subsystems: stacks vectors undefined page pmap
Loaded initial symtab at 0xc03dba58, strtab at 0xc0413018, # entries 13244
pmap_postinit: Allocated 35 static L1 descriptor tables
Copyright (c) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005
    The NetBSD Foundation, Inc.  All rights reserved.
Copyright (c) 1982, 1986, 1989, 1991, 1993
    The Regents of the University of California.  All rights reserved.

NetBSD 3.99.3 (VIPER) #255: Sun Jun  5 22:07:00 EEST 2005
        pooka@brain-damage.localhost.fi:/sys/arch/evbarm/compile/obj/VIPER
total memory = 65536 KB
avail memory = 59092 KB
mainbus0 (root)
cpu0 at mainbus0: PXA255/26x step A-0 (XScale core)
cpu0: DC enabled IC enabled WB enabled LABT branch prediction enabled
cpu0: 32KB/32B 32-way Instruction cache
cpu0: 32KB/32B 32-way write-back-locking Data cache
pxaip0 at mainbus0: PXA2x0 Onchip Peripheral Bus
pxaip0: CPU clock = 396.361 MHz
pxaintc0 at pxaip0 addr 0x40d00000-0x40d0001f: Interrupt Controller
pxagpio0 at pxaip0 addr 0x40e00000-0x40e0006f: GPIO Controller
sm0 at pxaip0 addr 0x8000300 intr 0
sm0: SMC91C111, revision 1, buffer size: 8192
sm0: MAC address xx:xx:xx:xx:xx:xx, default media MII (internal PHY)
sqphy0 at sm0 phy 0: Seeq 84220 10/100 media interface, rev. 0
sqphy0: using Seeq 84220 isolate/reset hack
sqphy0: 10baseT, 10baseT-FDX, 100baseTX, 100baseTX-FDX, auto
com0 at pxaip0 addr 0x40100000-0x4010001f intr 22: ns16550a, working fifo
com0: console
com1 at pxaip0 addr 0x40200000-0x4020001f intr 21: ns16550a, working fifo
com2 at pxaip0 addr 0x40700000-0x4070001f intr 20: ns16550a, working fifo
saost0 at pxaip0 addr 0x40a00000-0x40a0001f
saost0: SA-11x0 OS Timer
clock: hz=100 stathz = 64
```

Appendix 2: Devices in the configuration file

```
# The main bus device
mainbus0 at root

# The boot CPU
cpu0     at mainbus?

# peripherals
pxaip0        at mainbus0

# interrupt controller & gpio pins
pxaintc0 at pxaip0
pxagpio0 at pxaip0

# serial ports
options  COM_PXA2X0
options  FFUARTCONSOLE
com0 at pxaip0 addr 0x40100000 intr 22  # FFUART
com1 at pxaip0 addr 0x40200000 intr 21  # BTUART
com2 at pxaip0 addr 0x40700000 intr 20

# these two are not hanging off of pxaip, not really tested either
#com3    at pxaip0 addr 0x14300000   # COM5
#com4    at pxaip0 addr 0x14300010   # COM4

# SMC91C111 ethernet
sm0  at pxaip0 addr 0x08000300 intr 0

# MII/PHY support
sqphy*   at mii? phy ?          # Seeq 80220/80221/80223 PHYs
```