Porting OpenBSD

Niall O’Higgins <niallo@openbsd.org>
Uwe Stühler <uwe@openbsd.org>

OpenCON, 2005
Outline

1. Porting OpenBSD
   - What It Takes
   - Preparation
   - Cross-Development
   - The Boot Loader
   - Building The Kernel
   - Adapting Startup Code
   - Writing Device Drivers
   - Going Native
   - Subsequent Work

2. OpenBSD/zaurus
   - History
   - What It Took
   - What Was Done
   - Tricky Parts
   - Current Status
   - Future Plans
What It Takes

- motivation, some experience, time and persistence
What It Takes

- motivation, some experience, time and persistence
- about 20 developers having the machines
What It Takes

- motivation, some experience, time and persistence
- about 20 developers having the machines
- a user community
What It Takes

- motivation, some experience, time and persistence
- about 20 developers having the machines
- a user community
- “full” support includes that:
  - release install media is known to work
  - architecture can compile itself
  - most of the basic tools exist on the architecture
  - snapshots are made available on a regular basis
  - packages exist
Preparation

- get a hold of documentation
Preparation

- get a hold of documentation
- familiarise yourself with the architecture
Preparation

- get a hold of documentation
- familiarise yourself with the architecture
- start from an existing port that is very similar
Porting OpenBSD

Preparation

- get a hold of documentation
- familiarise yourself with the architecture
- start from an existing port that is very similar
- copy and rename machine-dependent sources
  - `sys/arch/machine/...`
  - `share/man/mann/mann.machine/...`
  - `etc/etc.machine/...`
  - `distrib/...`
Preparation

- get a hold of documentation
- familiarise yourself with the architecture
- start from an existing port that is very similar
- copy and rename machine-dependent sources
  - \texttt{sys/arch/machine/...}
  - \texttt{share/man/mann/mann.machine/...}
  - \texttt{etc/etc.machine/...}
  - \texttt{distrib/...}

- poke around in interesting places

  - opportunity to learn about things
Preparation

- get a hold of documentation
- familiarise yourself with the architecture
- start from an existing port that is very similar
- copy and rename machine-dependent sources
  - `sys/arch/machine/...`
  - `share/man/mann/mann.machine/...`
  - `etc/etc.machine/...`
  - `distrib/...`
- poke around in interesting places and try to remember what you’ve changed
  - opportunity to learn about things
  - it’s easy to make mistakes and some things can’t be tested immediately
to start the port, normally you have to cross-compile
to start the port, normally you have to cross-compile

- we have to use the GNU compiler toolchain (binutils, gcc, gdb, ...)
  - makes it difficult to port OpenBSD to architectures not already supported by the toolchain
Cross-Development

- to start the port, normally you have to cross-compile
- we have to use the GNU compiler toolchain (binutils, gcc, gdb, ...)
  - makes it difficult to port OpenBSD to architectures not already supported by the toolchain
- “make cross-tools” and “make cross-distrib” are there to aid the porter
Cross-Development

- to start the port, normally you have to cross-compile
- we have to use the GNU compiler toolchain (binutils, gcc, gdb, ...)
  - makes it difficult to port OpenBSD to architectures not already supported by the toolchain
- "make cross-tools" and "make cross-distrib" are there to aid the porter
- cross-compiling is not used once the port can compile itself
Cross-Development

- to start the port, normally you have to cross-compile
- we have to use the GNU compiler toolchain (binutils, gcc, gdb, ...)
  - makes it difficult to port OpenBSD to architectures not already supported by the toolchain
- “make cross-tools” and “make cross-distrib” are there to aid the porter
- cross-compiling is not used once the port can compile itself
- native builds are a good way to test the machine and a new port
Cross-Development

- to start the port, normally you have to cross-compile
- we have to use the GNU compiler toolchain (binutils, gcc, gdb, ...)
  - makes it difficult to port OpenBSD to architectures not already supported by the toolchain
- “make cross-tools” and “make cross-distrib” are there to aid the porter
- cross-compiling is not used once the port can compile itself
- native builds are a good way to test the machine and a new port
- as a result, we switch to native builds as soon as possible
need a way to load the kernel (JTAG is nice, but not always available)
The Boot Loader

- need a way to load the kernel (JTAG is nice, but not always available)
- loader can be 50 lines of assembly or a big C program
need a way to load the kernel (JTAG is nice, but not always available)

loader can be 50 lines of assembly or a big C program

in the long run you want to port `boot(8)` - the *stand-alone* kernel

- easier to port than the BSD kernel: does not use the full build infrastructure
- harder if you have no BIOS, Open Firmware-compliant or similarly sophisticated firmware to call out to (for console and disk access, device tree traversal, etc.)
- but `boot(8)` can run on and replace another operating system in memory - e.g. Linux : )
The Boot Loader

- need a way to load the kernel (JTAG is nice, but not always available)
- loader can be 50 lines of assembly or a big C program
- in the long run you want to port `boot(8)` - the *stand-alone* kernel
  - easier to port than the BSD kernel: does not use the full build infrastructure
  - harder if you have no BIOS, Open Firmware-compliant or similarly sophisticated firmware to call out to (for console and disk access, device tree traversal, etc.)
  - but `boot(8)` can run on and replace another operating system in memory - e.g. Linux :)
- good firmware can be used to simplify things at runtime
  - like OpenBoot callouts on “sparc” to traverse the device tree or print characters on the console
get familiar with *config*(8) and *files.conf*(5)
Building The Kernel

- get familiar with `config(8)` and `files.conf(5)`
- “multi-arch” platforms (e.g. cats, macppc, sgi, solbourne, zaurus) vs. “single-arch” platforms (amd64, i386, hppa, sparc, sparc64)
Building The Kernel

- get familiar with `config(8)` and `files.conf(5)`
- “multi-arch” platforms (e.g. cats, macppc, sgi, solbourne, zaurus) vs. “single-arch” platforms (amd64, i386, hppa, sparc, sparc64)
- work on RAMDISK first, then on GENERIC
  - with bsd.rd you can interactively test and debug the kernel and drivers
Building The Kernel

- get familiar with `config(8)` and `files.conf(5)`
- “multi-arch” platforms (e.g. cats, macppc, sgi, solbourne, zaurus) vs. “single-arch” platforms (amd64, i386, hppa, sparc, sparc64)
- work on RAMDISK first, then on GENERIC
  - with bsd.rd you can interactively test and debug the kernel and drivers
- building bsd.rd is only slightly more complicated
  - install `crunch tools` from `distrib/crunch`
  - run `make` in `distrib/machine/ramdisk`
  - `rdsetroot` may give you problems during cross-development
begin with \textit{start()} (locore.S)

- disable interrupts
- bring the processor into a known state
- initialise or disable MMU and caching
- relocate the kernel image
- initialise interrupt controller
- pick up boot arguments
- initialise early console (serial)
- find memory and initialise \textit{pmap(9)} backend
  - map the kernel
  - set up stack(s)
  - trap/vector tables
- call \textit{main()}
- begin with `start()` (locore.S)
  - disable interrupts
  - bring the processor into a known state
  - initialise or disable MMU and caching
  - relocate the kernel image
  - initialise interrupt controller
  - pick up boot arguments
  - initialise early console (serial)
  - find memory and initialise `pmap(9)` backend
    - map the kernel
    - set up stack(s)
    - trap/vector tables
  - call `main()`
begin with \textit{start()} (locore.S)
- disable interrupts
- bring the processor into a known state
- initialise or disable MMU and caching
- relocate the kernel image
- initialise interrupt controller
- pick up boot arguments
- initialise early console (serial)
- find memory and initialise \textit{pmap(9)} backend
  - map the kernel
  - set up stack(s)
  - trap/vector tables
- call \textit{main()}

use reliable, \textit{unbuffered} indicators for debugging (LED)
some drivers have to be done first:

- serial port (or another console device)
- interrupt controller
- crucial machine-dependent bus drivers such as `mainbus(4)` or `pxaip(4)`
Writing Device Drivers

- some drivers have to be done first:
  - serial port (or another console device)
  - interrupt controller
  - crucial machine-dependent bus drivers such as `mainbus(4)` or `pxaip(4)`
- BSD has the `autoconf(9)` framework
  - basically, there is direct and indirect configuration
some drivers have to be done first:
- serial port (or another console device)
- interrupt controller
- crucial machine-dependent bus drivers such as `mainbus(4)` or `pxaip(4)`

BSD has the `autoconf(9)` framework
- basically, there is direct and indirect configuration
- direct configuration is used where enumeration is possible (PCI, PCMCIA, ...) - drivers can easily be matched with hardware by device class, vendor ID and product ID
some drivers have to be done first:

- serial port (or another console device)
- interrupt controller
- crucial machine-dependent bus drivers such as `mainbus(4)` or `pxaip(4)`

BSD has the `autoconf(9)` framework

- basically, there is direct and indirect configuration
- direct configuration is used where enumeration is possible (PCI, PCMCIA, ...) - drivers can easily be matched with hardware by device class, vendor ID and product ID
- indirect configuration is used where busses have to way to see what devices are attached and how (ISA, I²C, ...) - drivers have to probe for the hardware
some drivers have to be done first:
- serial port (or another console device)
- interrupt controller
- crucial machine-dependent bus drivers such as
  \textit{mainbus}(4) or \textit{pxaip}(4)

BSD has the \textit{autoconf}(9) framework
- basically, there is direct and indirect configuration
- direct configuration is used where enumeration is possible
  (PCI, PCMCIA, ...) - drivers can easily be matched with
  hardware by device class, vendor ID and product ID
- indirect configuration is used where buses have to way to
  see what devices are attached and how (ISA, I^2C, ...) -
  drivers have to probe for the hardware

layered drivers and \textit{attachment drivers}
- \textit{apm}(4), \textit{lcd}(4), \textit{ohci}(4), \textit{pcmcia}(4), more?
- because of some obscure chips (\textit{scoop}(4), backlight
  control, ...
some drivers have to be done first:
- serial port (or another console device)
- interrupt controller
- crucial machine-dependent bus drivers such as *mainbus(4)* or *pxaip(4)*

BSD has the *autoconf(9)* framework
- basically, there is direct and indirect configuration
- direct configuration is used where enumeration is possible (PCI, PCMCIA, ...) - drivers can easily be matched with hardware by device class, vendor ID and product ID
- indirect configuration is used where busses have to way to see what devices are attached and how (ISA, I²C, ...) - drivers have to probe for the hardware

layered drivers and *attachment drivers*
- *apm(4), lcd(4), ohci(4), pcmcia(4)*, more?
- because of some obscure chips (*scoop(4)*, backlight control, ...)

you can use drivers from other BSDs
you can mount an NFS root until the disk driver works
(“make cross-distrib” can build a minimal distribution)
Going Native

- you can mount an NFS root until the disk driver works
  (“make cross-distrib” can build a minimal distribution)
- you have to cheat, but it’s done only once:
  - use the natively-built distribution from another port with the
    same CPU architecture (cats for zaurus)
  - worst case: cross-compile the native compiler
Subsequent Work

- fix most annoying bugs
Subsequent Work

- fix most annoying bugs
- port `boot(8)`
Subsequent Work

- fix most annoying bugs
- port `boot(8)`
- document the boot process (`boot_zaurus(8)`, ...)
Subsequent Work

- fix most annoying bugs
- port `boot(8)`
- document the boot process (`boot_zaurus(8)`, ...)
- document already supported devices (`intro(4)`, ...)
Subsequent Work

- Fix most annoying bugs
- Port `boot(8)`
- Document the boot process (`boot_zaurus(8)`, ...)
- Document already supported devices (`intro(4)`, ...)
- Build snapshots, announce the port and make it available
  - Update web pages
  - Set up a mailing list
  - Make other people do these things :)

Porting OpenBSD

OpenBSD/zaurus
Subsequent Work

- fix most annoying bugs
- port `boot(8)`
- document the boot process (`boot_zaurus(8)`, ...)
- document already supported devices (`intro(4)`, ...)
- build snapshots, announce the port and make it available
  - update web pages
  - set up a mailing list
  - make other people do these things :)
- write and document new device drivers
Subsequent Work

- fix most annoying bugs
- port `boot(8)`
- document the boot process (`boot_zaurus(8)`, ...)
- document already supported devices (`intro(4)`, ...)
- build snapshots, announce the port and make it available
  - update web pages
  - set up a mailing list
  - make other people do these things :)
- write and document new device drivers
- fix more bugs
Subsequent Work

- fix most annoying bugs
- port `boot(8)`
- document the boot process (`boot_zaurus(8)`, ...)
- document already supported devices (`intro(4)`, ...)
- build snapshots, announce the port and make it available
  - update web pages
  - set up a mailing list
  - make other people do these things :)
- write and document new device drivers
- fix more bugs
- make the ports tree aware of the new platform, eg create plists
Outline

1. Porting OpenBSD
   - What It Takes
   - Preparation
   - Cross-Development
   - The Boot Loader
   - Building The Kernel
   - Adapting Startup Code
   - Writing Device Drivers
   - Going Native
   - Subsequent Work

2. OpenBSD/zaurus
   - History
   - What It Took
   - What Was Done
   - Tricky Parts
   - Current Status
   - Future Plans
“cats” and “zaurus” are “multi-arch” ports

NetBSD/cats ported to OpenBSD by Dale Rahn (drahn@) to support ARM processors

Dale started in December 2004 based on OpenBSD/cats (but worked on some stuff before, like \texttt{lcd(4)})

kind-of usable for Theo in January 2005

first release was 3.7 (released in May 2005)

- only a few things missing, like audio support

work is ongoing
What It Took

- ball really started rolling at OpenCON 2004
What It Took

- ball really started rolling at OpenCON 2004
- more than 10 developers who contributed Zaurus-specific code in different areas (some of them didn’t even have one)
  - eg pascoe@ worked on `zaudio(4)` without even having a zaurus, all remotely
What It Took

- ball really started rolling at OpenCON 2004
- more than 10 developers who contributed Zaurus-specific code in different areas (some of them didn’t even have one)
  - eg pascoe@ worked on `zaudio(4)` without even having a zaurus, all remotely
- good documentation
What It Took

- Ball really started rolling at OpenCON 2004
- More than 10 developers who contributed Zaurus-specific code in different areas (some of them didn’t even have one)
  - Eg pascoe@ worked on `zaudio(4)` without even having a zaurus, all remotely
- Good documentation
  - Processor documentation was extremely useful, even for writing device drivers - it is a System-on-Chip design
What It Took

- ball really started rolling at OpenCON 2004
- more than 10 developers who contributed Zaurus-specific code in different areas (some of them didn’t even have one)
  - eg pascoe@ worked on `zaudio(4)` without even having a zaurus, all remotely
- good documentation
  - processor documentation was extremely useful, even for writing device drivers - it is a System-on-Chip design
  - touch-screen controller, audio controller and the microdrive were also documented
What It Took

- ball really started rolling at OpenCON 2004
- more than 10 developers who contributed Zaurus-specific code in different areas (some of them didn’t even have one)
  - eg pascoe@ worked on `zaudio(4)` without even having a zaurus, all remotely
- good documentation
  - processor documentation was extremely useful, even for writing device drivers - it is a System-on-Chip design
  - touch-screen controller, audio controller and the microdrive were also documented
  - but some Sharp-made chips are not, and depending on Linux source code as documentation is horrifying
What It Took

- Ball really started rolling at OpenCON 2004
- More than 10 developers who contributed Zaurus-specific code in different areas (some of them didn’t even have one)
  - Eg pascoe@ worked on `zaudio(4)` without even having a zaurus, all remotely
- Good documentation
  - Processor documentation was extremely useful, even for writing device drivers - it is a System-on-Chip design
  - Touch-screen controller, audio controller and the microdrive were also documented
  - But some Sharp-made chips are not, and depending on Linux source code as documentation is horrifying
- Money for machines for developers
What It Took

- ball really started rolling at OpenCON 2004
- more than 10 developers who contributed Zaurus-specific code in different areas (some of them didn’t even have one)
  - eg pascoe@ worked on `zaudio(4)` without even having a zaurus, all remotely
- good documentation
  - processor documentation was extremely useful, even for writing device drivers - it is a System-on-Chip design
  - touch-screen controller, audio controller and the microdrive were also documented
  - but some Sharp-made chips are not, and depending on Linux source code as documentation is horrifying
- money for machines for developers
- an unknown amount of beer :D to start things
What Was Done

- processor startup code adapted for the Zaurus
  (`arm/arm/locore.S` is shared between different machines)
What Was Done

- processor startup code adapted for the Zaurus
  (arm/arm/locore.S is shared between different machines)
- \texttt{boot(8)} ported to Linux
What Was Done

- processor startup code adapted for the Zaurus (`arm/arm/locore.S` is shared between different machines)
- `boot(8)` ported to Linux
- hacked up `com(4)` driver for PXA27x
  - integrated `pcic(4)` driver for PXA27x from NetBSD
  - integrated `lcd(4)` for PXA27x
What Was Done

- processor startup code adapted for the Zaurus
  \texttt{arm/arm/locore.S} is shared between different machines
- \texttt{boot(8)} ported to Linux
- hacked up \texttt{com(4)} driver for PXA27x
  - integrated \texttt{pcic(4)} driver for PXA27x from NetBSD
  - integrated \texttt{lcd(4)} for PXA27x
- fake \texttt{apm(4)} backend to use the existing framework
What Was Done

- Processor startup code adapted for the Zaurus (arm/arm/locore.S is shared between different machines)
- `boot(8)` ported to Linux
- Hacked up `com(4)` driver for PXA27x
  - Integrated `pcic(4)` driver for PXA27x from NetBSD
  - Integrated `lcd(4)` for PXA27x
- Fake `apm(4)` backend to use the existing framework
- And many little things...
Tricky Parts

- in general, porting went on fairly quickly and straightforward
Tricky Parts

- in general, porting went on fairly quickly and straightforward
- `nlist()` on non-native objects - a cross-development (non-)issue in elfrdsetroot
Tricky Parts

- in general, porting went on fairly quickly and straightforward
- `nlist()` on non-native objects - a cross-development (non-)issue in elfrdsetroot
- undocumented chips and circuitry (`scoop(4)`, backlight controller, power circuit)
in general, porting went on fairly quickly and straightforward

\textit{nlist()} on non-native objects - a cross-development (non-)issue in elfrdsetroot

undocumented chips and circuitry (\textit{scoop(4)}, backlight controller, power circuit)

we couldn’t support C860 machines
  - no disk drive; just raw flash
  - too many differences between models
  - more developers have the C3x00
in general, porting went on fairly quickly and straightforward

*nlist()* on non-native objects - a cross-development (non-)issue in elfrdsetroot

undocumented chips and circuitry (*scoop*(4), backlight controller, power circuit)

we couldn't support C860 machines
  - no disk drive; just raw flash
  - too many differences between models
  - more developers have the C3x00

*zkbd*(4) is just a bunch of GPIO inputs
Tricky Parts

- in general, porting went on fairly quickly and straightforward
- `nlist()` on non-native objects - a cross-development (non-)issue in elfrdsetroot
- undocumented chips and circuitry (`scoop(4)`, backlight controller, power circuit)
- we couldn’t support C860 machines
  - no disk drive; just raw flash
  - too many differences between models
  - more developers have the C3x00
- `zkbd(4)` is just a bunch of GPIO inputs
- UART was not completely compatible to a standard 16550A or 16750
Tricky Parts

- in general, porting went on fairly quickly and straightforward
- `nlist()` on non-native objects - a cross-development (non-)issue in elfrdsetroot
- undocumented chips and circuitry (`scoop(4)`, backlight controller, power circuit)
- we couldn’t support C860 machines
  - no disk drive; just raw flash
  - too many differences between models
  - more developers have the C3x00
- `zkbd(4)` is just a bunch of GPIO inputs
- UART was not completely compatible to a standard 16550A or 16750
- screen rotation on framebuffer console - not in 3.7
in general, porting went on fairly quickly and straightforward

\texttt{nlist()} on non-native objects - a cross-development (non-)issue in elfrdsetroot

undocumented chips and circuitry (\texttt{scoop(4)}, backlight controller, power circuit)

we couldn’t support C860 machines
  - no disk drive; just raw flash
  - too many differences between models
  - more developers have the C3x00

\texttt{zkbd(4)} is just a bunch of GPIO inputs

UART was not completely compatible to a standard 16550A or 16750

screen rotation on framebuffer console - not in 3.7

no hardware floating-point unit - creates performance problems with some software (e.g. xmms, mplayer; on zaurus we use integer math decoders for these kinds programs where possible)
Current Status

- mostly feature-complete
Current Status

- mostly feature-complete
- we support wired and infrared serial ports
Current Status

- mostly feature-complete
- we support wired and infrared serial ports
- all existing USB and PCMCIA cards should just work
Current Status

- mostly feature-complete
- we support wired and infrared serial ports
- all existing USB and PCMCIA cards should just work
- LCD works in portrait and landscape mode (under X)
  - but you have to restart X
  - on-the-fly rotation is hard - no API in X to do that
Current Status

- mostly feature-complete
- we support wired and infrared serial ports
- all existing USB and PCMCIA cards should just work
- LCD works in portrait and landscape mode (under X)
  - but you have to restart X
  - on-the-fly rotation is hard - no API in X to do that
- audio playback works well, with occasional glitches
Current Status

- mostly feature-complete
- we support wired and infrared serial ports
- all existing USB and PCMCIA cards should just work
- LCD works in portrait and landscape mode (under X)
  - but you have to restart X
  - on-the-fly rotation is hard - no API in X to do that
- audio playback works well, with occasional glitches
- even Java works
Future Plans

- of course, continue to fix and improve stuff
  - better keyboard support (*zkbd*(4))
  - “xdm=YES” should work out of the box
  - PCMCIA bugfixes (some detection problems, voltage switching)
  - anything else?
Future Plans

- of course, continue to fix and improve stuff
  - better keyboard support (*zkbd*(*4*))
  - “xdm=YES” should work out of the box
  - PCMCIA bugfixes (some detection problems, voltage switching)
  - anything else?

- add “sysctl hw.cpuspeed” support for running at 91 Mhz, 208 Mhz or 416 Mhz
Future Plans

- of course, continue to fix and improve stuff
  - better keyboard support (**zkbd(4)**)
  - “xdm=YES” should work out of the box
  - PCMCIA bugfixes (some detection problems, voltage switching)
  - anything else?

- add “sysctl hw.cpusespeed” support for running at 91 Mhz, 208 Mhz or 416 Mhz

- **wsdisplay(4)** is improving thanks to miod@
Future Plans

- of course, continue to fix and improve stuff
  - better keyboard support (*zkbd(4)*)
  - “xdm=YES” should work out of the box
  - PCMCIA bugfixes (some detection problems, voltage switching)
  - anything else?

- add “sysctl hw.cpusespeed” support for running at 91 Mhz, 208 Mhz or 416 Mhz

- *wsdisplay(4)* is improving thanks to miod@

- support CF XGA cards (miod@ and matthieu@ are working on this)
Future Plans

- of course, continue to fix and improve stuff
  - better keyboard support (**zkbd(4)**)
  - “xdm=YES” should work out of the box
  - PCMCIA bugfixes (some detection problems, voltage switching)
  - anything else?

- add “sysctl hw.cpuspeed” support for running at 91 Mhz, 208 Mhz or 416 Mhz

- **wsdisplay(4)** is improving thanks to miod@

- support CF XGA cards (miod@ and matthieu@ are working on this)

- **apm(4)** improvements
  - turn off some more chips when suspended
  - extended power-saving measures, perhaps
Future Plans

- of course, continue to fix and improve stuff
  - better keyboard support (\texttt{zkbd(4)})
  - “xdm=YES” should work out of the box
  - PCMCIAA bugfixes (some detection problems, voltage switching)
  - anything else?
- add “sysctl hw.cpuspeed” support for running at 91 Mhz, 208 Mhz or 416 Mhz
- \texttt{wsdisplay(4)} is improving thanks to miod@
- support CF XGA cards (miod@ and matthieu@ are working on this)
- \texttt{apm(4)} improvements
  - turn off some more chips when suspended
  - extended power-saving measures, perhaps
- \texttt{gpiocntl(8)} support (LED, hinge state?)
Future Plans

- of course, continue to fix and improve stuff
  - better keyboard support (\texttt{zkbd(4)})
  - “xdm=YES” should work out of the box
  - PCMCIA bugfixes (some detection problems, voltage switching)
  - anything else?
- add “\texttt{sysctl hw.cpuspeed}” support for running at 91 Mhz, 208 Mhz or 416 Mhz
- \texttt{wsdisplay(4)} is improving thanks to miod@
- support CF XGA cards (miod@ and matthieu@ are working on this)
- \texttt{apm(4)} improvements
  - turn off some more chips when suspended
  - extended power-saving measures, perhaps
- \texttt{gpiocctl(8)} support (LED, hinge state?)
- Fix the ARM pmap issue with write-back caching
Future Plans (cont’d)

- audio recording
Future Plans (cont’d)

- audio recording

- BlueTooth support via USB dongles - grange@ already ported `ubt(4)` from FreeBSD and created net/bluetooth-tools
Future Plans (cont’d)

- audio recording
- BlueTooth support via USB dongles - grange@ already ported `ubt(4)` from FreeBSD and created `net/bluetooth-tools`
- support more Zaurus models (C860, probably even StrongARM-based SL-5500)
Future Plans (cont’d)

- audio recording
- BlueTooth support via USB dongles - grange@ already ported `ubt(4)` from FreeBSD and created `net/bluetooth-tools`
- support more Zaurus models (C860, probably even StrongARM-based SL-5500)
- SDIO support - new framework
Future Plans (cont’d)

- audio recording
- BlueTooth support via USB dongles - grange@ already ported `ubt(4)` from FreeBSD and created net/bluetooth-tools
- support more Zaurus models (C860, probably even StrongARM-based SL-5500)
- SDIO support - new framework
- USB device framework
  - client-side `cdce(4)`
  - storage class device support needs to be thought through. is it useful after all?
Future Plans (cont’d)

- Audio recording
- Bluetooth support via USB dongles - grange@ already ported `ubt(4)` from FreeBSD and created `net/bluetooth-tools`
- Support more Zaurus models (C860, probably even StrongARM-based SL-5500)
- SDIO support - new framework
- USB device framework
  - Client-side `cdce(4)`
  - Storage class device support needs to be thought through. Is it useful after all?
- What can you think of?
need donations (time, money, bugfixes, beer)

thanks