Bringup is Hard

by

Tim Rightnour
The NetBSD Project
garbled@netbsd.org

Abstract

Bringup, is the initial stage of a new port of an Operating System to a new hardware platform. Often, bringup is one of the most difficult things to learn how to do in OS programming. This article cannot hope to teach the reader how to bringup a new machine, as most of the actual work requires a reasonably sound understanding of both the target hardware, and the target OS. However, what I do hope to convey, are some of the hints, that together with the proper knowledge, will make the task ahead much simpler. This article will concentrate on bringup of NetBSD on a target platform, but presumably most of this information could be applied to other Operating Systems. This article is written with the assumption that the reader has a general knowledge of the kernel, and C coding practices. Since bringup is a completely machine-dependant excersize, specific details particular to whatever hardware you may be working on cannot reasonably be covered.

1. What NOT to do

While this is primarily an article on how to bringup a machine, there are a number of things you should avoid at all costs.

- Don't setup a web page, or mailing list, or announce your intentions to the world before writing any actual code. You will get innundated with people who wish to help out. These people fall into two categories, people who pester you weekly to ask if it's done yet, and people who offer to help but don't have the faintest clue how. Community bringup efforts almost always fail, almost every successful bringup was done by one person, by himself.
- Don't attempt to port to a new machine as your first C program. If you can't write "hello world" then you can't bit-bang it out a serial port either. It would be ideal if you had at least some experience writing kernel code, or in the very least, are familiar with compiling your own kernels. It is also highly likely that at some point you are going to need to write some amount of assembly code for your platform. Often this is very small amounts, and is similar to existing code, which you can copy, but you may also need to write large chunks from scratch, depending on the machine you are bringing up.
- If you find someone who is helpful and willing to answer questions you might have, don't pester them night and day with a million questions. They will stop answering eventually.
- Don't pick a machine that is incredibly rare, undocumented, and some kind of esoteric set-top-box as the first machine you attempt to bringup. You will fail. Pick something easy for your first attempt, such as a different model of a machine already supported by NetBSD.
- Don't feel bad if you do fail. Bringup is really hard. If you get completely stuck, or keep banging your head against the wall and make no progress, take a break, and come back to it in a few months. When you come back, you may find that you can solve the problem much quicker.
- Don't forget to backup your code multiple times, so when you do come back in a month to it, it wasn't lost in a disk crash.
- Don't try to follow -current during bringup. Grab a CVS tree, make sure it compiles, wait around a few days if you have to for any bugs that might be in it to get fixed, and then never update the tree again until just before you are ready to commit your code.
- Don't wait until every little detail is perfect before committing your code. As soon as the machine can reach single user mode, get it in the tree. Current is diverging away from you as fast as you are writing code, and if you wait too long, you will end up having to rewrite large portions of it. Additionally, someone else might be porting to the same machine, and might beat you to the punch because you were waiting for perfection.

Sometimes you might even want to commit before single user mode is achievable, this is especially true for ports that have a high interest level, or that are extremely complex. If you get hit by a bus before you commit, nobody else will commit your code for you.

- Don't get dragged down by user requests during the early stages of bringup. Just keep working on your schedule, and your plan, do not attempt to make anyone but yourself happy. If they want it done, let them do it themselves.

2. Get to know the machine

When targeting a new machine for NetBSD, there are a few important things you need to know about the machine:

- Does it have an MMU?
- What CPU does it have?
- Does NetBSD already support that CPU?
- Does gcc support that CPU?
- Is there another port that is similar hardware wise to this one?
- Has another OS been ported to this machine already?
- Is there a native OS for this machine?
- Does the machine actually work?

2.1. Does it have an MMU?

This is a very important first question to ask. Machines without an MMU are not currently supported by NetBSD, and are rather unlikely to be supported any time soon. Furthermore, even if they were supported, they would be very difficult to work with, and would be rather un-UNIX-like in nature. Attempting to write support for an MMU-less machine is not an easy task, and definitely not something to try on your first attempt at bringup.

2.2. What CPU does it have?

You are definitely going to need to know this at some point in your attempt. It may seem like an obvious thing to know, but it isn't always that simple. Some machines may have a CPU that is obscured by the packaging, or hidden from prying eyes by stickers. Additionally, even if you think you know what CPU should be in a machine, you should always double-check by physically looking at the CPU. One machine in particular, confounded me for weeks, until I discovered that it had been cobbled together from parts, and the model number didn't match the interior components, meaning I had been writing code for the wrong CPU type.

2.3. Does NetBSD already support that CPU?

While it certainly is possible to write new CPU support for NetBSD, it is a much more difficult task. Instead of just having to deal with the machine and hardware itself, you also now have to write the entire low level back end pmap and locore functionality. Such a task is far more difficult than a simple bringup, and is outside the scope of this article.

2.4. Does gcc support that CPU?

Again, you certainly can write gcc support for your new CPU, but now you've got to write that first, before you can even begin writing any code at all for the kernel. This is probably the worst possible situation, as it means you have to write a compiler back end, followed by CPU code, and finally, actual machine-bringup.

2.5. Is there another port that is similar hardware wise to this one?

Look at other ports of NetBSD to various hardware. Look for similarities in shared components, such as PCI buses, similar chipsets, similar firmware, etc etc. Similar ports might not even be in the same CPU family. Look at all of the ports in the NetBSD tree to see if any of them look like they might deal with some aspect of the machine that you intend to port. The worst case is, you might spend a few days reading code that will likely be similar to what you will be writing yourself soon.

2.6. Has another OS been ported to this machine already?

While it isn't advised to just start copying code from some other project, it might be useful for some aspects of your project. If the other project is license-compatible with NetBSD, then it's certainly possible to re-use routines or files from that project, but be sure to give proper credit. In most cases, it's simply useful to have the code for the other project handy, so when you get stuck, you can reference that code to see how they got around the problem that is stalling you out. Sometimes, the code will have magic numbers, which cannot be derived from any existing documentation, which trigger certain hardware features. Make note of these, and remember to revisit them, and attempt to understand what they are doing.

2.7. Is there a native OS for this machine?

You may need to do some dissasembly of various routines of the original OS, or simply use it as a platform for compiling test objects from. Having the original OS available to you can be invaluable during the early stages. Additionally, there is often a great deal of useful information in the header files for the original OS, which might come in handy later.

2.8. Does the machine actually work?

Hardware gets old, flaky, and dies sometimes. If you purchased your machine from a surplus lot, or off Ebay, its possible that its dead on arrival. You should fully test out the machine as best you can before even attempting to port to it. If there is a native OS for the machine, get ahold of a copy of that OS, install it, and run tests on all the vital components, such as serial ports, video, disk drives, controller cards, etc etc. It is best to find out any issues ahead of time, rather than discovering later that the reason you can't get the serial console code working, is that the serial port is broken.

3. Get to know the machine, physically

Open the case of the machine. Start by looking at all the option cards in the machine. Are they standard cards, like MCA, ISA, PCI? Attempt to identify all of these cards, if they are standard cards, try putting them into other machines that currently run NetBSD and see if they work or not.

Look at all of the chips on the mainboard of the machine. Try to identify each chip on the machine. You can often look up chips by punching the codes on the chips into Google, or by looking at catalogs. You should attempt to identify as many chips on the mainboard as you possibly can, as each one of these chips may require a driver at some point. Of specific interest to you at this time, are the serial, network, and disk controller chips. You may also want to be on the lookout for any SuperIO type chips, which are all-in-one chips that provide a large number of IO functions on one chip. Look for clock oscillators and write down all of the frequencies of these devices. It is often helpful to have a photo or diagram of the mainboard, so later, you can attempt to guess which oscillator is tied to which chips. This data can be extremely useful when trying to talk to things like serial ports, which need to be programmed with a clock frequency in order to produce output at a specific speed. High resolution photos of both sides of the mainboard will keep you from having to re-open the case at a later time.

4. Get to know the machine, intimately

For every chip you found in the machine, find a datasheet for it. If there are datasheets or documentation for the machine itself, get ahold of that. If there is documentation on the CPU itself, books or datasheets, get as much as you can. Buy every book they wrote about the machine or chip, you will need it all.

Most datasheets can be found by searching google with the part numbers. There are a number of datasheet archives on the net which contain this information and will allow you to download the sheets in PDF format. If you cannot find information on your specific chip, look for information on other chips with similar part numbers. It is possible that your chip is simply an updated, or outdated form of another chip that is documented. Grab documentation on every chip that you think might be similar to the one you are working with, it will likely be close enough that you can do something with it.

Your last resort for finding documentation, is to try to contact the manufacturer of the chips or the machine. This is generally the last resort, because it is a highly frustrating exercise most of the time, and rarely produces anything usable. Most companies are not likely to give you documentation they haven't already released on the web. Furthermore, it is often unlikely that you will get routed from whoever you speak to initially, to someone who actually knows what documentation you are looking for, or has the authority to give it to you. However, if you do find someone willing to help, the benefits can be quite worthwhile. Often, such a person will know things about the chip that wasn't documented, or be able to answer specific questions about the chip that you might have. Do not pester them with simple questions early on. Try to explore and solve as many of the problems as you can by yourself, before asking them for help. The person you are asking is probably helping you in his or her spare time, and you can easily wear out your welcome by asking a bunch of simple questions you could have answered yourself.

Unfortunately, some things might not be documented, or might only be documented under an NDA. If this is the case, you might have to try to figure it out on your own, or look at other code that works with the chip in question. Sometimes, the chip will truly be indecipherable, these are the hardest machines of all to deal with. Perhaps you can skip support for that part of the machine, or work around the lack of support, or perhaps it is vital to the operation of the box. It is best to find this out now, before you get too far into the port.

5. Get to know the machine, exploration

Most machines have some form of boot ROM, or firmware layer that provides you with a set of facilities for booting the native OS. The very first thing you need to do is understand how this firmware layer functions. It could be a very high-level system, such as a fully featured Open Firmware, or it could be a very simple stripped-down system such as the BIOS on an old PC. Whatever the system is, you need to play around with it, and discover what features it has, and what things it will allow you to do. For example:

- Can you use the firmware to netboot an image?
- Can it handle both a physical or serial console?
- Does it provide any sort of debugging interface?
- How does it load the bootimage, is there a specific format?

Each of those questions needs to be answered before you begin any sort of work on the machine. If you don't have a console of any kind, working on the machine is going to be extremely difficult, but not impossible. Play around with booting whatever OS came with the machine in different ways, such as via netboot, or disk, or CDROM. Then attempt to boot your own images. These do not need to be working images of any kind, for example, when porting to one machine, I repeatedly tried to netboot my password file, and later /bin/ls. Each of these experiments gives you an idea of how the machine deals with images, and how it attempts to load them.

Many machines have very specific requirements for what they consider a valid bootable file, and where and how they will look for such files. In the initial stages, it is very important that you discover what these requirements are, for at least one method of booting. For undocumented machines, you can sometimes reverse engineer the boot images from the original OS. If the machine has a hard drive, or boot floppy, grab an image of that disk and inspect it with a tool such as "hexdump" or "od". You might also attempt to run "file" on the image, to see if it is some sort of standard file type. Often strange boot images are simply a magical header, sometimes with a checksum embedded, followed by a standard ELF binary. You may also find things such as x86 boot blocks followed immediately by ELF or COFF images. Sometimes these images will be a combination of different methods, such as a magical header, followed by a compressed ELF binary. If you can identify where the end of the header is, you can use a tool like "dd" to strip away the header, and write the embedded binary to your disk. This will allow you to use other tools such as "readelf" or "gunzip" to play with that binary file, and learn about it's contents.

Finally, you need to discover what the load address of the machine is, and if that address can be changed or not. A machine which is booting a file, must load the code into RAM at a specific location, and then jump to that location to begin executing the code. This location is called the load address. If you inspect a bootable image from the stock OS for a machine with a tool such as "readelf" you will find a LOAD header for the program. When you create your bootable image, this will be the address you set with the "-Ttext" ld flag. You may also wish to investigate if it is at all possible to change the load address for the machine, as you may find that you need more room later on to boot a large kernel.

6. What is the ideal machine to port to?

In a perfect world, there are a number of traits every machine would have. Lack of any of these traits does not make a porting effort impossible, just more difficult. The more factors you lack, the harder it gets.

- Machine has a working serial console that is part of the firmware, and requires no special setup on your part.
- Machine has the ability to netboot via DHCP.
- Machine has a reboot switch.
- Machine has a full featured boot ROM with built-in debugging facilities.
- Machine has a button or serial port command sequence which interrupts the running code and drops you back to the ROM or debugger.
- Machine has an easily accessible LED or LCD bank that you can write messages to.
- Machine uses standard off the shelf components and option cards.
- Machine is fully documented.

7. The perfect work environment

Ideally, there are a number of things you should have on your primary development system, and optional features that will make work on your port much easier.

- A NetBSD system with plenty of disk space, preferably the latest stable release.
- A functional DHCP, NFS, DNS and tftp server.
- A terminal server allowing you to telnet to the serial console of the device. Barring that, a good serial cable and decent terminal program will do.
- A remotely managed power switch, which will allow you to powercycle the box without having to walk over to it.
- A NetBSD machine running the same CPU architecture as your target machine, to allow you to build native binaries without having to cross compile. It can also be highly useful to use with specialized tools such as "readelf".
- Barring the above, another machine running NetBSD of the same endianness as the one you are porting to. Having a machine of the same endianess makes disassembly and reading hexdumps much simpler.
- A second target machine, on your network, running the original OS the machine came with. This can be handy to allow you to login and check things quickly, without having to revert your test setup on the target machine.

8. Setting up your cross compile environment

There has been allot of documentation written previously about how to set up a cross compile environment for NetBSD. In a nutshell, you should check out a full source tree from anonymous CVS, and then issue a "build.sh" for the closest architecture to the one you are porting to. For example, if porting to a PowerPC CPU, build the tools and GENERIC kernel for macppc or prep. This will create a full set of cross compiling tools in your TOOLDIR which can be used to build your bootimages, and later, your kernels.

An example of setting up a crosscompiling environment to build an ofppc bootloader:

```
./build.sh -m ofppc -D /usr/src/current/destdir/ofppc \
    -O /usr/src/current/objdir/ofppc -T /usr/src/current/tooldir \
    -R /usr/src/current/releasedir -U tools
cd /usr/src/current/src/sys/arch/ofppc/stand/
/usr/src/current/tooldir/bin/nbmake-ofppc
```

You may also have to run different make targets, such as "clean", "objdir", or "dependall" by hand.

9. Hello world in 1000 lines of assembler

One of the very first things you need to do when porting to a new machine, is to write a bootloader. In some cases, the machine may be so similar to another NetBSD port, that you can simply use that port's bootloader, or perhaps you already have one. If either of the above is the case, you should skip ahead to the kernel section, if not, read on.

There are two major types of bootloader that can be written for your new port. The first, is the single stage bootloader, and the second, is the multi stage bootloader.

9.1. Single stage bootloaders

A single stage bootloader is generally a single file, which is composed of an optional header, a bootloader binary, a magic number, and then the kernel image. Generally speaking, these four components are simply catted together into one large file.

The single stage bootloader works by loading the entire image into RAM, and then executing the code at the front of the image, which is the actual bootloader program. The bootloader program then scans through it's own image in RAM, to find the magic number that was placed into the image. This magic number tells the bootloader where the bootloader ends, and the kernel begins. It then copies the kernel to a start address, and jumps to that address to begin executing the kernel.

A single stage bootloader is usually the easiest type of bootloader to write, and is generally used temporarily until a multi-stage loader can be written. However, there are certain machines which require the use of a single stage loader. Such machines generally have very simplistic boot ROMs which do not provide any services whatsoever to programs that they execute. Such services are referred to as callbacks, because you call back to the boot ROM to perform some function on your behalf.

A machine without callbacks has no way to talk to the disks, controllers, NICs or other hardware on the machine, unless you specifically write drivers for these devices into the bootloader. This means that you cannot rely upon the ROM for things like serial console access, or block device IO for finding a kernel on disk.

A single stage bootloader consists of basic code to get a console working, and code as described above to launch the kernel. Everything else is left up to the kernel to set up. Machines which require single stage bootloaders often do not bother to program PCI or other option cards properly, meaning that you may have to take care of these things either in the bootloader, or very early in the kernel.

9.2. Multi-Stage bootloaders

A mutli-stage bootloader is the ideal bootloader for running NetBSD. A multi-stage bootloader is executed by the boot ROM, and then makes callbacks into that ROM, to ask it to provide services on it's behalf. For example, your bootloader would contain a simple ROM IO block device, using the native callbacks, to access the disks on the machine. Using this block device, you would then be able to read the disks looking for an FFS file-system, and load the kernel off that, and then execute it.

Some multi-stage bootloaders can be written in a single binary, while others need to be written as multiple actual stages. For example, i386 machines execute a small 512-byte chunk of code that is located in the Master Boot Record. This code has to be very simple, and only looks for a second bootloader at a specific point on the disk. Once it finds that loader, it executes it, which allows the system to run a much more complex binary, allowing for filesystem access and other features needed to find the actual kernel on your root filesystem.

Machines which do not have a bootloader size limit, or which have a fairly generous limit, can do the same thing in a single binary.

9.3. Writing your first bootloader

Before even attempting anything as complex as loading an actual kernel, you first have to write a hello world program, which can be executed by the boot ROM, and spit the words "hello world" out to your console of choice. This can range from being very simple, to incredibly complex, depending on the callbacks your ROM provides. Without callbacks, you will have to write a device driver into your program, and then use a console driver to output data to the screen. There are a few basic components to your hello world program:

9.3.1. The locore, or srt0.s file

This is the starting point of your bootloader. When you compile a C program, there is a chunk of libc code called srt0 which is prepended to your binary. This code is the very first instructions that get executed by your program. Often this is labeled as “_start:” in the assembly code. In this routine, you will save off any arguments that may have been passed to you as registers by the boot ROM, and set up your stack area and pointers to the stack area. Once this is done, you will need to issue any instructions that are necessary for the setup of your CPU type prior to running any code. Such instructions might include turning off interrupts, or disabling the MMU. Once you have accomplished those basic tasks, you will jump to main() in your actual bootloader code. If you are having trouble figuring out how to do these things, it might be worthwhile to look at how other platforms do this, or to look at the srt0 code in libc for your architecture.

9.3.2. Blinking Lights

If you are attempting to write a console driver within your bootloader, then you will need some sort of method to tell you when things work, and when things do not work. If your console driver is very simple, you can likely skip this step, however, it is very likely that you will need some sort of debugging aid in the future, so if you do run into problems later on, come back here and implement some kind of blinking light code.

Blinking light code is simply some kind of action that the machine takes to let you know that it is still alive, and is processing the code you have written. Because you don’t have a console yet, you cannot simply issue a printf to see where things are failing. Oftentimes, a machine will simply hang if you do something it doesn’t like. In order to debug hangs that occur before you have any sort of output, you need to blink a light, or somehow give physical evidence that something is occurring. There are a number of ways to provide such evidence visibly, listed in order of preference if available on your machine:

- A numeric or alphanumeric LCD panel on the machine which allows you to write arbitrary numbers or strings to it.
- A simple LED on the machine which you can blink in various patterns.
- Some kind of exit routine that returns your program back to the boot ROM.
- A power or reset register that you can use to cause the machine to power down or reboot.
- A chunk of bad assembly code that will cause the machine to reset, or machine check.
- Something that activates some kind of device, such as spinning up a floppy drive, or ejecting a cdrom.

One thing to remember is that often this early in the boot, you have absolutely no clock functionality whatsoever. This means that when blinking a light, you cannot use a sleep or delay function of any kind. Instead, you will have to write a quick for loop that counts to some large number. Remember that the faster the CPU on your machine, the larger this number will have to be. Additionally, you should probably write this function in assembler, as many C compilers will optimize away useless counting.

These blinking light functions often have to be very simple, and are generally written completely in assembler. You likely do not have access to any higher MMU functionality, or any kind of bus back-end to allow you to access a device on a bus. In order to get around this, you may need to hardcode addresses of devices, and write small inb/outb routines that bitbang values into specific registers. For example, on the RS/6000, it was necessary to write a tiny assembler routine that wired up a segment register to point to a specific region of IO space, allowing access to that region. Once the region was accessible, it was possible to write values directly into the NVRAM to have them show up on the 3-digit LCD display.

However, even writing such simple code often requires debugging. There are two types of responses a machine will generally have during this early stage of booting.

(1) The machine crashes, or machine checks.

(2) The machine hangs.

You need to write small routines that can cause both of these conditions. When your code fails, it will fail in one of the two above ways. Hopefully, the machine acts differently depending on which of the two ways it failed, if not, your work will be that much harder. If you should encounter a hang while executing your code, insert your crash routine into the code, and move it up and down the code until you have narrowed down the location of the hang. The opposite is true for the crash, simply move your hang routine around to locate the bad instruction. All of the above of course, assumes that you have no sort of boot ROM level debugger that can tell you where it all went wrong.

When you think you have enough of a routine that can blink a light of some kind you should create a very simple main() for your bootloader that simply blinks the light, and then goes into an infinite loop. Once you can get this program to load, execute, and blink the light, it is safe to move on to the next step.

9.3.3. libsa is your friend

NetBSD has an extremely helpful library called libsa, which provides a number of useful low-end utilities and functions that are designed to make writing a bootloader simpler. For example, there is basic filesystem read code for a number of common filesystems, such as msdosfs and ffs. Each of the various options to libsa can be conditionally compiled in to your finished bootloader by setting the appropriate Makefile CFLAGS. The Makefile and Makefile.inc files in “sys/arch/lib/libsa” provide some documentation on which libsa flags will provide what functionality. At a minimum, you will most likely find the stdio routines such as printf the most useful, however you should look through the various routines provided in libsa to see which things might be useful to you. Most of the higher level functionality you will likely need in a bootloader will already be provided by libsa for you.

In addition to libsa, you will also need to make use of libkern, and libz. These libraries are both located in “sys/arch/lib”. Libz provides you with the ability to decompress a compressed kernel image, which is handy for keeping your images smaller. Libkern provides the most basic functionality for your bootloader. This includes such routines as bswap, bcopy, str*, and mem* routines. Again, the purpose of these two libraries is to keep you from having to write your own versions of these essential functions, and allow different architectures to all share the same code.

9.3.4. Writing a serial console driver

There are three basic components of a bootloader serial console driver.

(1) A simple bus back-end of some kind to allow you to access the serial device behind whatever bus it might be on. This can often be a set of hardcoded addresses called through an inb/outb routine.

(2) A getc/putc pair of routines, which write and read individual characters from the serial port.

(3) An init routine which sets up the basic values of the serial port, such as the frequency it runs at, the speed, initialization of the chip, etc etc.

Almost all serial chips are one of a few basic types. Most are either zs based, or NS16550 based. You can find example drivers for both of these chips throughout the tree, and some of them are even used in simple bootloaders. If you have one of these chips, you simply have to wire up routines to talk to the bus, and then guess the right frequency for the serial clock.

Finding the clock frequency for the serial port is not always simple however. There are usually a few sets of standard values, which can be found in the datasheets, or as defines in the NetBSD drivers for these chips. However, if those do not work, you will have to guess the frequency on your own. There are two rather simple ways to do this:

(1) If your machine outputs anything readable to the serial port before loading your binary, it has likely already initialized the serial chip. You can probably get away with not writing an init routine, and then reading the clock values out of the serial device’s registers. You can then write these values out to the port with your putc routine, and then write a proper init routine. You can

only get away with this if you are very careful not to reset the port in any way during your program load. It is also dependent on the assumption that the boot ROM didn't do something like reset the port for you just before executing your code.

(2) The other method, is to write a for loop, and try every value you can. You will want to write an initialization function that loops over different frequency values, incrementing by something like 100 or so for each attempt. For each attempt, you should write out the frequency attempted, and a rather long stream of alphanumeric characters. Often when you get closer to the correct frequency, more and more characters will print correctly, but others will not. Once you get reasonably close to what you think the correct frequency is, try a more limited range of frequencies in your loop, but with a smaller increment, like 1 per loop. You may also need to play around with changing the rate and parity options of the serial port on your desktop, to try and match whatever output comes out.

You can also attempt to base your guesses on the speeds of the various clock oscillator chips that were present on the mainboard. Look for any crystals that might be near the serial ports, or serial chips. Another thing to base your guesses on is the frequency of the bus that the serial chip sits on. ISA, PCI and MCA all have standard bus speeds, and you can use these speeds to base your clock rates on, if you are lucky.

9.3.5. Writing a console driver

Having a serial driver doesn't give you immediate access to the console. Before you can use things like printf, you need to write a small console driver. A console driver consists of the following routines:

(1) A struct consdev which contains a list of functions which your driver provides.

(2) A probe function, which probes for the existence of the device, and returns a true/false.

(3) An init function, which sets up the physical device.

(4) A putchar/getchar pair of routines which put and get individual characters to/from the console.

(5) A scan routine which checks if the console device is valid and functional.

If you had to write your own serial driver, you already have most of these functions at hand, and can simply reuse them. However, if you are writing a console driver for a platform that has boot ROM callbacks, you will need to write a function for each of the above, which utilizes those callbacks to perform the requested task. There are many examples throughout the tree of console drivers in bootloaders that use both raw serial or VGA devices, as well as drivers that use Open Firmware callbacks to talk to the console.

9.3.6. Writing a disk or network driver

Assuming your boot ROM provides callbacks of some kind that allow you to access the disk or network devices, you should be able to write a fairly simple network driver for your machine. The actual writing of a network or disk driver in a bootloader is beyond the scope of this article, however, many examples of simple bootloader drivers for these devices exist throughout the tree. The most common of these are the simple Open Firmware based network and disk drivers that the sparc and macppc ports use.

For a disk driver, or an emulated disk driver, you need to provide an open, a close, init, and a strategy routine. The strategy routine is the routine that reads the actual data from the device. When building a single-stage bootloader that loads the kernel from the RAM-resident boot image, your init routine will look for the magic number which you embedded into the image and set some kind of a read-start pointer. Your strategy routine would then copy the data starting at that pointer, one block at a time into the final memory location for the kernel.

9.3.7. Pulling it all together

At this point, you should have enough routines written to create a fully functional bootloader. Your bootloader's main() should perform the following functions, in order:

(1) Initialize the console.

(2) Copy any special arguments that were passed to the bootloader, such as boot flags, or magical memory locations provided by the boot ROM into a set of bootinfo structures. These structures can later be passed to the kernel via a pointer, allowing the kernel access to this information.

(3) Initialize any devices such as network or disk.

(4) Call loadfile() from libsa to read the kernel from the selected device into memory at the proper location.

(5) Jump to that location and begin executing the kernel.

There may be other tasks that need to be performed by the bootloader depending on the machine or CPU architecture that you are booting on, such tasks include:

(1) Clearing the BSS region, or memzeroing other ranges of RAM in preparation for kernel loading or execution.

(2) Copying data provided by the boot ROM from it's original location to a different location that won't get overwritten by a large kernel image during load.

(3) Relocating the bootloader itself in RAM to avoid being overwritten by the copy of the kernel to RAM.

(4) Flushing various caches or TLBs.

(5) Shutting off watchdog timers.

(6) Preparing machine state for the load of the kernel. This can include setting special registers, mapping areas of RAM/IO, or programming certain devices that are easier to access from the bootloader than from the kernel. Anything that makes the kernel easier to write is fair game.

9.3.8. Other things to do while playing with the bootloader

Just because you can actually launch a kernel doesn't mean you should jump right to that step. Having a functional bootloader gives you the opportunity to play around with the hardware on the machine, while having a very simple interface available. This is a good time to inspect any boot arguments that might have been passed to you from the boot ROM, such as magic memory locations, or device paths.

Many machines will often pass a pointer to a location in RAM which contains data about the machine. This information is often vital during later stages of kernel bringup, as it may contain device location information, or maps of the machine. It would be well worth your while now to write a few routines to decode this data and print it to your screen, so you can use this as a reference when later attempting to utilize it in the kernel.

Having a console available, you may also wish to inspect the initialization registers of various hardware chipsets, to see what values exist in those chips before the kernel changes them on you. This is also a good place to test any low-level kernel code that you might need to write in the kernel later prior to having a console available under the kernel.

9.3.9. Some final thoughts on writing a bootloader

While it may seem tempting to write disk IO drivers into the bootloader immediately, it might not be worth your while to bother yet. Loading a kernel off disk is your eventual goal for the machine, but in order to load the kernel off the disk, you first have to put it there, which is rather difficult without a working kernel. At the same time, a floppy or CDROM driver might be nice, but again, remember that you will have to make a new floppy or CDROM every time you wish to test a new kernel. You are best served by writing only a network driver and using that to load your kernel from an NFS or tftp server of some kind. This will allow you to easily replace the image and boot hundreds of times without burning floppies all day long.

Even if you cannot netboot your actual bootloader, it is still worth writing a network driver into the bootloader. While you may have to place the bootloader on a hard drive or floppy, you can at least leave that image alone while writing the kernel code and still netboot the actual kernel images repeatedly. If you think this is not a big deal, be aware that you will likely end up booting between 300 and 500 kernel attempts before you get something workable.

Also, do not get discouraged if your boot loader doesn't work the first time. Don't get discouraged if it doesn't work the first hundred times either, it can often take hundreds of boot attempts before you have a reasonably working bootloader that can load and attempt to execute a kernel.

Finally, before you even attempt to write actual kernel code, you will need to test your ability to load and execute a kernel. There are a few easy tricks you can use to test a kernel before actually writing kernel code. One such trick, is to try booting a kernel from another port of NetBSD of the same CPU architecture. While it is highly unlikely to actually work, it will show you if the loader is actually capable of booting the file. If such an image isn't available, or doesn't provide any useful debugging information, you can also sometimes load your own bootloader as a kernel image. You can recompile your bootloader with a different start address, which would be the same start address you would use in the kernel. Then load this modified boot loader with your primary boot loader and see if it works.

When trying to write your bootloader, it is often difficult to set up all the Makefiles and machine targets to build your binary with. One easy trick to get around this, is to use another port's directory structure. For example, if you were going to write a bootloader for a new port to a PowerPC architecture, look at the bootloaders of the various PowerPC ports, and try to pick the one that is closest to what you will be doing. You can then delete all the code from that directory, and simply re-use the Makefile and directory structure under stand to build your bootloader. This will also allow you to use the build.sh tool to build parts of your image.

10. The kernel side of things

Now that you've got a bootloader working, it's time to start porting the kernel over. The kernel work is a bit harder than the bootloader work, because there is alot more to worry about. On the one side, you do have more services available to you, such as some of the machine independant bus back ends, but on the other side, there is more that you have to wire up to prepare for them. There are a few major steps in getting a kernel working on your machine:

- Getting the kernel to load, and execute.
- Get basic services running.
- Calling the basic initialization.
- Device autoconfiguration.
- Get interrupts working.
- Boot/root device detection.
- Single user mode.

10.1. Getting the kernel to load, and execute.

The very first thing you have to do, is get an actual kernel to load and run on the machine, and make some kind of sign to you that it actually did so.

10.1.1. Strip another kernel down

It is suggested that you start by finding another port of NetBSD to the same CPU architecture, and stripping that kernel down.

First, edit the config file, and remove all the devices from the machine. Then you need to edit the basic files that are required to build your kernel. These files may differ from port to port, but basically you need the following:

- **Locore.S**: Basic startup code that initializes the lowest levels of your machine, and calls your primary init routine. Written completely in assembler.
- **machdep.c**: Primary initialization routine, and other routines that are used during early setup of the machine.
- **consinit.c**: Console setup and initialization.
- **autoconf.c**: Autconfiguration callbacks and root/boot device detection routines.
- **interrupts.c**: Routines to deal with the interrupt controller.
- **mainbus.c**: The mainbus is the root bus for most architectures. It is not required to be named mainbus, but there must be a base-level bus for other devices and busses to attach to.

There may be other required files, depending on your architecture, you will have to look at how other ports do things to determine this for yourself.

Once you have the right set of files in place, you should strip as much code out of those files as possible before attempting to compile. Most of the code in these files is extremely machine-dependant, and will likely not be correct for your machine. It is possible however, that some of it is correct, or is very close to what you will end up needing to do. Therefore, the best thing to do for most of this code is to insert “#if 0” statements around the code inside each function, keeping a return intact for non-void functions. This way you can easily reference the code and bring it back a little bit at a time. Before you do any actual work on the code, try to compile the kernel you have just stripped down, and fix all the compiler errors you have caused. You may have to add back in some stub functions you removed, or edit the files in the conf directory to rip out more drivers or files. Once you have it compiling, proceed onwards.

10.1.2. Hello World in a million lines of C

Just as in the bootloader, the very first thing you need to do is somehow signal youself that the kernel has correctly booted. Generally, the easiest way to do this, is to use the same early-debugging techniques you used in the bootloader, but instead to put them in Locore.S. Place these debugging routines as high in the Locore file as you possibly can, and try to boot the kernel and get the expected response from the machine.

Don’t be afraid to mangle the file and throw code into Locore until you get things working. Import as much code as you need from your bootloader to get some kind of confirmation that loading and executing took place. You can rip it back out later if you need to.

It is also highly likely that your Locore.S file is not quite correct for this machine yet. It should generally be similar to the srt0.s file you created for the bootloader, however, there are a few subtle differences. Generally the Locore.S file either sets up the MMU, or references other code that does so. It is also expected that you setup the cpu structures that contain information about each cpu on the machine, such as all the register contents, and pointers to lwp’s and processes. Each architecture is different, but if you are working with an architecture that allready has other ports to it, it is very likely that most of this code already exists in either a shared location, or in the code of the other port. Take a close look at how the other port or ports do things, and copy it as best as you can. This is probably the hardest part of your porting effort, and there is very little advice I can offer here, other than to read the code for as many other ports as you possibly can, and see how they do things.

Once you have what you think is a properly functioning Locore.S file, you should again make use of your debugging tricks, and boot the machine and confirm that it has survived to call your primary initialization routine. The primary initialization routine at this point should simply be some kind of debugging statement that lets you know the machine is running.

you should again make use of your debugging tricks, and boot the machine and confirm that it has survived to call your primary initialization routine. The primary initialization routine at this point should simply be some kind of debugging statement that lets you know the machine is running.

10.2. Get Basic Services Running

Once you can successfully reach your initialization routine, it's time to get basic kernel services up and running. There are two services that you absolutely need running to do any kind of serious work whatsoever, and they should be done immediately. As for which to do first, I suggest that you do whichever one is easier to implement on your machine first, as they will both aid in fixing the other. The two things are:

(1) A boot console routine, to allow you to printf.

(2) A reboot function.

10.2.1. Reboot function

I cannot stress enough how important it is to have a reboot function available as soon as humanly possible on your machine. You will likely be booting at least 300 kernels on this machine before you have it up and running, and depending on how evil the machine is, maybe as many as a thousand. Power-cycling a machine a thousand times, or running across the room to hit the reboot switch gets old really fast. Write a tiny reboot function that you can just call from anywhere in the code, and place this in either Locore.S or machdep.c, as appropriate.

10.2.2. The boot console

The boot console is usually harder to write, but is absolutely vital. Hopefully, you have enough code from your bootloader, that you can re-use to get a basic boot console routine up and running. There are a few basic things you need to get a console running inside the kernel:

- An inb/outb routine that will talk to whatever bus you are using to access the console. This could be as complex as an ISA bus behind a PCI bridge, or as simple as some kind of firmware callback. Generally if a firmware callback is used, you won't need inb/outb at all.
- A getc/putc pair of routines that will read/write a single byte from/to the console device.
- Optionally, some kind of init routine that intializes the basic console.
- A "struct consdev" structure, that defines which routines to call for the various console services.

The consdev structure will require a number of functions, such as your getc/putc functions, but also expects other functions such as poll, init, probe, etc. There are empty stub functions for most of these, and other ones that are completely optional can simply be set to NULL. Look at how other arches initialize the console early. An excellent example of an extremely simple ns16650-based early boot console is the netwinder port. Look at the kcomcons code in that port to see just how simple this code can actually be.

Once you have the console working, put a printf in your primary intialization routine, and then boot your kernel and debug it until you see the printf fire off. Congratulations, you just wrote hello world in a million lines of kernel code.

10.3. Calling the basic initialization

The basic initialization sets up a few different things, depending on the architecture you are working on. Many of these things are completely machine dependant, or in the very least cpu dependant. Again, you should look at how other ports do things, and look for common shared code that you can simply call.

- Copy down any kind of boot arguments or information passed to us from the bootloader into a location that can be used by the kernel.
- Initialize the MMU.
- Initialize bus_space (if needed this early).
- Initialize the boot console, or full console if possible at this stage.
- Locate and mark the beginning and end of physical memory.
- Setup the interrupt handler (if possible this early).
- Configure UVM.
- Initialize the pmap.

For any given architecture, the order of these may be different, and some may not be possible to do right away. For each one, do the best you can to get it wired up properly, and if it is not yet possible to set it up, leave it as an empty stub for now and come back to it.

10.4. Device autoconfiguration

Device autoconfiguration is the process that NetBSD uses to locate hardware on the machine, and prepare it for use by the OS. Generally upon exit from your primary intialization routine, your Locore routine should call main(). Main will begin the autoconfiguration process based on your kernel config file. The very first device you should have is a mainbus of some kind.

10.4.1. mainbus.c

The mainbus is one of the simpler devices on your machine, however, it will help to introduce you to the basic driver mechanics on NetBSD. At a minimum, any driver needs the following things:

- A CFATTACH_DECL() which defines the basic entry points to the driver.
- A match routine, which probes for the existance of the hardware and returns a graded truth value.
- An attach routine, which actualy performs basic initialization of the device and sets it up for actual use.

Additionally, a bus needs a structure called attach arguments. This is used to pass information from the higher level bus, down to the individual drivers, or busses that attach to it. For example, on a PCI bus, you would pass down the bus number and bridge information to the driver for one of the option cards, so it knows it's own location on the machine.

The match routine for mainbus is generally very simple. Since a mainbus is not an actual device, just an attachment node, we don't need to actually look for it. Instead, we should simply set a global variable somewhere that says we have found it, and return 1 to indicate that it has been found. Additionally, we should check to see that the global found variable has not already been set, and if it has, return a zero.

Autoconfiguration works by calling all of the different match routines for each device the machine finds, and then seeing which one returned the highest value. Therefore if your mainbus routine allways returns 1, autoconfiguration will find many instances of mainbus throughout your machine. This can also be used on other devices to return a confidence level. A simple generic driver for a given card can return a 1 in the match routine, stating that it will operate the card. However, a more specific driver written for the specific chipset on the card might return a 2, saying that it is a better driver. Autoconfiguration will pick the highest return from all of the matches, and call the attach routine for that one. If nothing matches, it will move on to the next device.

Following the match routine, you have an attach routine. The attach routine is used to setup and initialize the actual hardware the machine has found for use. In the case of the mainbus however, what you actually want to do is look for other busses on the machine. For example, if you have a machine with a single PCI bus, and an ISA bus as a child of that bus, you would create a mainbus_attach routine that set up enough of the machine that the PCI bus could be found, and then called the config_found routine.

The config_found routine hands your attach arguments structure to all of the match routines in the kernel, and sees if any of them return truth. In the above example, you would need to write a PCI bus attach routine that correctly identifies the PCI bus on your machine, and returns truth.

You can also use the mainbus_attach routine to do other early machine startup initialization work, such as basic bus_space mappings. It is also likely that your mainbus_attach should call config_found with the appropriate arguments to find all of the cpu's on your machine.

As a very simple test, you can write an attach routine that simply prints a hello out, and returns. Once you have that working, you can write the dection code for the cpu's on your machine, and call the appropriate autoconfiguration routines to find those, and test that you have that working. Finally, start probing and identifying the actual busses and hardware devices on your machine. If you do not have interupt routines written for your machine, you may need to write stub routines for some of the interrupt related work, or come back to autoconfiguration later.

10.5. Get interrupts working

Interrupts are probably the next hardest thing to get working on a machine. Interrupts can be very picky about timing, and exactly how they are accessed and handled. Additionally, you need to actually know the map of which interrupt corresponds to which device. Without this map, you will be completly lost and stabbing in the dark at the machine.

While interrupt code is usually completely machine dependant, there are a few hints I can offer to help you debug interrupts.

First, when writing your handler routine, you can add small printf functions to the handler that tells you which interrupt fired and what it was. However, you have to be very careful when doing this, as it is very likely that once an interrupt fires, you might get a flood of them, especially if you aren't acknowledging or handling them properly.

Second, if your machine just stalls or hangs following autoconfiguration, it is probably an interrupt problem. Most likely, your machine either got an interrupt, and hung trying to service that interrupt. Its also highly likely that your machine is spewing spurious interrupts, or the same interupt over and over again, and is in a tight loop trying to handle them. Adding printfs to the handler routines, and enabling debug options in the various drivers will help you understand what is going on here.

It is very important to note that these printfs are both helpful and harmful when developing basic interrupt code. You should write the printfs using some kind of CPP macro that will allow you to easly switch them all on or off in the code at compile time. Many printfs in critical interrupt sections will cause the interrupt handlers to not function correctly at all. While the printfs helped debug the code, you may have very well fixed it, but the kernel still will not run, because the printfs are causing the interrupt controller to freak out. If you think you might have found something and fixed it, pull the printfs and try again.

It is also important to note, that if you are using a serial console, and have attached the actual device driver that operates that serial port, that you are now operating the serial device in interrupt mode, as opposed to the polled mode that most console serial ports run in. This means, that your printfs will generate more interrupts, which generate more printfs, ad-infinitum. An easy way to deal with this is to comment out the serial driver from your kernel config file while debugging interrupts. When this is done, the kernel will continue to use the basic bootconsole until the end of autoconfiguration is hit. Once the end of autoconfiguration is hit however, it will likely panic because you do not have a valid console driver initialized.

10.6. Boot/root device detection

It is reccomended that you write some kind of boot device detection routine, that determines what device or method was used to boot the kernel. Without this, the kernel will have to ask you what your root and swap devices are every time you boot the machine.

A boot device detection routine relies on some kind of information regarding the boot device having been passed from the bootloader to the kernel. This can be some kind of a string that identifies which device was used to boot the device, or any other representation.

Your autoconf.c routine should contain a device_register function. This function is called after the match routine for every device that is found on the machine. Using this routine, you can register data, or properties of the device into the device structure, that can later be used to match up against the data the bootloader passed you.

For example, an OpenFirmware based machine might give you a boot string of "/pci@80000000/eth@d". On your machine, there is an ethernet card whose entry in the OpenFirmware device tree happens to match this. When you find that device in the system, your device_register routine should copy that string in some way to the driver structure, so that it can later be matched against the boot string. It is reccomended that such device properties use the proplib functions provided by the kernel.

Finally, it may also be neccesary to dig certain peices out of the device tree while setting up devices. Your device_register routine may also be used to do things like find the global MAC address of the machine and write it to a location that can later be used by the ethernet driver.

10.7. Single user mode

Once you have your machine surviving autoconfiguration, finding all the devices that are needed for basic operation, and correctly detecting that it was booted over the network, you should be ready to finally attempt single user mode. As a very basic setup, you can usually grab the base.tgz and etc.tgz from another port of the same CPU type, and install those onto a directory on your NFS server somewhere. Then set up an NFS export to the machine, and configure dhcpd to hand out that export as the root directory for your machine. Once all that is done, it should allow you to pick root via nfs over your ethernet card. With any luck it will drop you at a shell prompt. Congratulations!

Now you should either commit your code, or contact one of the NetBSD developers to let them know you have a new port that is ready to go in. Now is also the time where you can finally brag that you have accomplished porting NetBSD to a new machine. In general, a good method of doing this is to post a dmesg of your machine reaching single user mode to netbsd-ports, or the cpu-specific port list.