FreeBSD Developers’ Handbook

The FreeBSD Documentation Project

FreeBSD Developers’ Handbook

by The FreeBSD Documentation Project
Published August 2000
Copyright © 2000, 2001 by The FreeBSD Documentation Project

Welcome to the Developers’ Handbook. This manual is a *work in progress* and is the work of many individuals. Many sections do not yet exist and some of those that do exist need to be updated. If you are interested in helping with this project, send email to the FreeBSD documentation project mailing list <`freebsd-doc@FreeBSD.org`>. The latest version of this document is always available from the FreeBSD World Wide Web server (http://www.FreeBSD.org/). It may also be downloaded in a variety of formats and compression options from the FreeBSD FTP server (ftp://ftp.FreeBSD.org/pub/FreeBSD/doc) or one of the numerous mirror sites (http://www.freebsd.org/handbook/mirrors-ftp.html).

Table of Contents

List of Tables

List of Examples

I. Introduction

Chapter 1. Developing on FreeBSD

This will need to discuss FreeBSD as a development platform, the vision of BSD, architectural overview, layout of /usr/src, history, etc.

Thank you for considering FreeBSD as your development platform! We hope it will not let you down.

Chapter 2. The BSD Vision

Chapter 3. Architectural Overview

Chapter 4. The Layout of /usr/src

The complete source code to FreeBSD is available from our public CVS repository. The source code is normally installed in `/usr/src` which contains the following subdirectories.

Directory	Description
`bin/`	Source for files in `/bin`
`contrib/`	Source for files from contributed software.
`crypto/`	DES source
`etc/`	Source for files in `/etc`
`games/`	Source for files in `/usr/games`
`gnu/`	Utilities covered by the GNU Public License
`include/`	Source for files in `/usr/include`
`kerberosIV/`	Source for Kerbereros version IV
`kerberos5/`	Source for Kerbereros version 5
`lib/`	Source for files in `/usr/lib`
`libexec/`	Source for files in `/usr/libexec`
`release/`	Files required to produce a FreeBSD release
`sbin/`	Source for files in `/sbin`
`secure/`	FreeSec sources
`share/`	Source for files in `/sbin`
`sys/`	Kernel source files
`tools/`	Tools used for maintenance and testing of FreeBSD
`usr.bin/`	Source for files in `/usr/bin`
`usr.sbin/`	Source for files in `/usr/sbin`

II. Basics

Chapter 5. Programming Tools

This chapter was written by James Raynard <`jraynard@FreeBSD.org`>. Modifications for the Developers' Handbook by Murray Stokely <`murray@FreeBSD.org`>.

5.1. Synopsis

This document is an introduction to using some of the programming tools supplied with FreeBSD, although much of it will be applicable to many other versions of Unix. It does *not* attempt to describe coding in any detail. Most of the document assumes little or no previous programming knowledge, although it is hoped that most programmers will find something of value in it

5.2. Introduction

FreeBSD offers an excellent development environment. Compilers for C, C++, and Fortran and an assembler come with the basic system, not to mention a Perl interpreter and classic Unix tools such as `sed` and `awk`. If that is not enough, there are many more compilers and interpreters in the Ports collection. FreeBSD is very compatible with standards such as POSIX and ANSI C, as well with its own BSD heritage, so it is possible to write applications that will compile and run with little or no modification on a wide range of platforms.

However, all this power can be rather overwhelming at first if you've never written programs on a Unix platform before. This document aims to help you get up and running, without getting too deeply into more advanced topics. The intention is that this document should give you enough of the basics to be able to make some sense of the documentation.

Most of the document requires little or no knowledge of programming, although it does assume a basic competence with using Unix and a willingness to learn!

5.3. Introduction to Programming

A program is a set of instructions that tell the computer to do various things; sometimes the instruction it has to perform depends on what happened when it performed a previous instruction. This section gives an overview of the two main ways in which you can give these instructions, or "commands" as they are usually called. One way uses an *interpreter*, the other a *compiler*. As human languages are too difficult for a computer to understand in an unambiguous way, commands are usually written in one or other languages specially designed for the purpose.

5.3.1. Interpreters

With an interpreter, the language comes as an environment, where you type in commands at a prompt and the environment executes them for you. For more complicated programs, you can type the commands into a file and get the interpreter to load the file and execute the commands in it. If anything goes wrong, many interpreters will drop you into a debugger to help you track down the problem.

The advantage of this is that you can see the results of your commands immediately, and mistakes can be corrected readily. The biggest disadvantage comes when you want to share your programs with someone. They must have the same interpreter, or you must have some way of giving it to them, and they need to understand how to use it. Also users may not appreciate being thrown into a debugger if they press the wrong key! From a performance point of view, interpreters can use up a lot of memory, and generally do not generate code as efficiently as compilers.

In my opinion, interpreted languages are the best way to start if you have not done any programming before. This kind of environment is typically found with languages like Lisp, Smalltalk, Perl and Basic. It could also be argued that the Unix shell (`sh`, `csh`) is itself an interpreter, and many people do in fact write shell “scripts” to help with various “housekeeping” tasks on their machine. Indeed, part of the original Unix philosophy was to provide lots of small utility programs that could be linked together in shell scripts to perform useful tasks.

5.3.2. Interpreters available with FreeBSD

Here is a list of interpreters that are available as FreeBSD packages (ftp://ftp.FreeBSD.org:pub/FreeBSD/packages/), with a brief discussion of some of the more popular interpreted languages.

To get one of these packages, all you need to do is to click on the hotlink for the package, then run

```
# pkg_add package name
```

as root. Obviously, you will need to have a fully functional FreeBSD 2.1.0 or later system for the package to work!

BASIC

Short for Beginner’s All-purpose Symbolic Instruction Code. Developed in the 1950s for teaching University students to program and provided with every self-respecting personal computer in the 1980s, BASIC has been the first programming language for many programmers. It’s also the foundation for Visual Basic.

The Bywater Basic Interpreter (ftp://ftp.FreeBSD.org:pub/FreeBSD/packages/lang/bwbasic-2.10.tgz) and the Phil Cockroft’s Basic Interpreter (ftp://ftp.FreeBSD.org:pub/FreeBSD/packages/lang/pbasic-2.0.tgz) (formerly Rabbit Basic) are available as FreeBSD FreeBSD packages (ftp://ftp.FreeBSD.org:pub/FreeBSD/packages/)

Lisp

A language that was developed in the late 1950s as an alternative to the “number-crunching” languages that were popular at the time. Instead of being based on numbers, Lisp is based on lists; in fact the name is short for “List Processing”. Very popular in AI (Artificial Intelligence) circles.

Lisp is an extremely powerful and sophisticated language, but can be rather large and unwieldy.

FreeBSD has GNU Common Lisp (ftp://ftp.FreeBSD.org:pub/FreeBSD/packages/gcl-2.0.tgz) available as a package.

Perl

Very popular with system administrators for writing scripts; also often used on World Wide Web servers for writing CGI scripts.

The latest version (version 5) comes with FreeBSD.

Scheme

A dialect of Lisp that is rather more compact and cleaner than Common Lisp. Popular in Universities as it is simple enough to teach to undergraduates as a first language, while it has a high enough level of abstraction to be used in research work.

FreeBSD has packages of the Elk Scheme Interpreter (ftp://ftp.FreeBSD.org:pub/FreeBSD/packages/lang/elk-3.0.tgz), the MIT Scheme Interpreter (ftp://ftp.FreeBSD.org:pub/FreeBSD/packages/lang/mit-scheme-7.3.tgz) and the SCM Scheme Interpreter (ftp://ftp.FreeBSD.org:pub/FreeBSD/packages/lang/scm-4e1.tgz).

Icon

The Icon Programming Language (ftp://ftp.FreeBSD.org:pub/FreeBSD/packages/lang/icon-9.0.tgz).

Logo

Brian Harvey’s LOGO Interpreter (ftp://ftp.FreeBSD.org:pub/FreeBSD/packages/lang/ucblogo-3.3.tgz).

Python

The Python Object-Oriented Programming Language (ftp://ftp.FreeBSD.org:pub/FreeBSD/packages/lang/python-1.2)

5.3.3. Compilers

Compilers are rather different. First of all, you write your code in a file (or files) using an editor. You then run the compiler and see if it accepts your program. If it did not compile, grit your teeth and go back to the editor; if it did compile and gave you a program, you can run it either at a shell command prompt or in a debugger to see if it works properly. [1]

Obviously, this is not quite as direct as using an interpreter. However it allows you to do a lot of things which are very difficult or even impossible with an interpreter, such as writing code which interacts closely with the operating system—or even writing your own operating system! It's also useful if you need to write very efficient code, as the compiler can take its time and optimise the code, which would not be acceptable in an interpreter. And distributing a program written for a compiler is usually more straightforward than one written for an interpreter—you can just give them a copy of the executable, assuming they have the same operating system as you.

Compiled languages include Pascal, C and C++. C and C++ are rather unforgiving languages, and best suited to more experienced programmers; Pascal, on the other hand, was designed as an educational language, and is quite a good language to start with. Unfortunately, FreeBSD doesn't have any Pascal support, except for a Pascal-to-C converter in the ports.

As the edit-compile-run-debug cycle is rather tedious when using separate programs, many commercial compiler makers have produced Integrated Development Environments (IDEs for short). FreeBSD does not have an IDE as such; however it is possible to use Emacs for this purpose. This is discussed in Section 5.7.

5.4. Compiling with `cc`

This section deals only with the GNU compiler for C and C++, since that comes with the base FreeBSD system. It can be invoked by either `cc` or `gcc`. The details of producing a program with an interpreter vary considerably between interpreters, and are usually well covered in the documentation and on-line help for the interpreter.

Once you've written your masterpiece, the next step is to convert it into something that will (hopefully!) run on FreeBSD. This usually involves several steps, each of which is done by a separate program.

1. Pre-process your source code to remove comments and do other tricks like expanding macros in C.
2. Check the syntax of your code to see if you have obeyed the rules of the language. If you have not, it will complain!
3. Convert the source code into assembly language—this is very close to machine code, but still understandable by humans. Allegedly. [2]
4. Convert the assembly language into machine code—yep, we are talking bits and bytes, ones and zeros here.

5. Check that you have used things like functions and global variables in a consistent way. For example, if you have called a non-existent function, it will complain.
6. If you are trying to produce an executable from several source code files, work out how to fit them all together.
7. Work out how to produce something that the system's run-time loader will be able to load into memory and run.
8. Finally, write the executable on the file system.

The word *compiling* is often used to refer to just steps 1 to 4—the others are referred to as *linking*. Sometimes step 1 is referred to as *pre-processing* and steps 3-4 as *assembling*.

Fortunately, almost all this detail is hidden from you, as `cc` is a front end that manages calling all these programs with the right arguments for you; simply typing

```
% cc foobar.c
```

will cause `foobar.c` to be compiled by all the steps above. If you have more than one file to compile, just do something like

```
% cc foo.c bar.c
```

Note that the syntax checking is just that—checking the syntax. It will not check for any logical mistakes you may have made, like putting the program into an infinite loop, or using a bubble sort when you meant to use a binary sort. [3]

There are lots and lots of options for `cc`, which are all in the man page. Here are a few of the most important ones, with examples of how to use them.

`-o` *`filename`*

The output name of the file. If you do not use this option, `cc` will produce an executable called `a.out`. [4]

```
% cc foobar.c               executable is a.out
% cc -o foobar foobar.c     executable is foobar
```

`-c`

Just compile the file, do not link it. Useful for toy programs where you just want to check the syntax, or if you are using a `Makefile`.

```
% cc -c foobar.c
```

This will produce an *object file* (not an executable) called `foobar.o`. This can be linked together with other object files into an executable.

`-g`

Create a debug version of the executable. This makes the compiler put information into the executable about which line of which source file corresponds to which function call. A debugger can use this information to show the source code as you step through the program, which is *very* useful; the disadvantage is that all this extra information makes the program much bigger. Normally, you compile with `-g` while you are developing a program and then compile a “release version” without `-g` when you’re satisfied it works properly.

```
% cc -g foobar.c
```

This will produce a debug version of the program. [5]

`-O`

Create an optimised version of the executable. The compiler performs various clever tricks to try and produce an executable that runs faster than normal. You can add a number after the `-O` to specify a higher level of optimisation, but this often exposes bugs in the compiler’s optimiser. For instance, the version of `cc` that comes with the 2.1.0 release of FreeBSD is known to produce bad code with the `-O2` option in some circumstances.

Optimisation is usually only turned on when compiling a release version.

```
% cc -O -o foobar foobar.c
```

This will produce an optimised version of `foobar`.

The following three flags will force `cc` to check that your code complies to the relevant international standard, often referred to as the ANSI standard, though strictly speaking it is an ISO standard.

`-Wall`

Enable all the warnings which the authors of `cc` believe are worthwhile. Despite the name, it will not enable all the warnings `cc` is capable of.

`-ansi`

Turn off most, but not all, of the non-ANSI C features provided by `cc`. Despite the name, it does not guarantee strictly that your code will comply to the standard.

`-pedantic`

Turn off *all* `cc`'s non-ANSI C features.

Without these flags, `cc` will allow you to use some of its non-standard extensions to the standard. Some of these are very useful, but will not work with other compilers—in fact, one of the main aims of the standard is to allow people to write code that will work with any compiler on any system. This is known as *portable code*.

Generally, you should try to make your code as portable as possible, as otherwise you may have to completely re-write the program later to get it to work somewhere else—and who knows what you may be using in a few years time?

```
% cc -Wall -ansi -pedantic -o foobar foobar.c
```

This will produce an executable `foobar` after checking `foobar.c` for standard compliance.

`-l`*`library`*

Specify a function library to be used during when linking.

The most common example of this is when compiling a program that uses some of the mathematical functions in C. Unlike most other platforms, these are in a separate library from the standard C one and you have to tell the compiler to add it.

The rule is that if the library is called `lib`*`something`*`.a`, you give `cc` the argument `-l`*`something`*. For example, the math library is `libm.a`, so you give `cc` the argument `-lm`. A common “gotcha” with the math library is that it has to be the last library on the command line.

```
% cc -o foobar foobar.c -lm
```

This will link the math library functions into `foobar`.

If you are compiling C++ code, you need to add `-lg++`, or `-lstdc++` if you are using FreeBSD 2.2 or later, to the command line argument to link the C++ library functions. Alternatively, you can run `c++` instead of `cc`, which does this for you. `c++` can also be invoked as `g++` on FreeBSD.

```
% cc -o foobar foobar.cc -lg++      For FreeBSD 2.1.6 and earlier
% cc -o foobar foobar.cc -lstdc++   For FreeBSD 2.2 and later
% c++ -o foobar foobar.cc
```

Each of these will both produce an executable `foobar` from the C++ source file `foobar.cc`. Note that, on Unix systems, C++ source files traditionally end in `.C`, `.cxx` or `.cc`, rather than the MS-DOS style `.cpp` (which was already used for something else). `gcc` used to rely on this to work out what kind of compiler to use on the source file; however, this restriction no longer applies, so you may now call your C++ files `.cpp` with impunity!

5.4.1. Common `cc` Queries and Problems

1. I am trying to write a program which uses the `sin()` function and I get an error like this. What does it mean?

```
/var/tmp/cc0143941.o: Undefined symbol ‘_sin’ referenced from text segment
```

When using mathematical functions like `sin()`, you have to tell `cc` to link in the math library, like so:

```
% cc -o foobar foobar.c -lm
```

2. All right, I wrote this simple program to practice using `-lm`. All it does is raise 2.1 to the power of 6.

```
#include <stdio.h>

int main() {
 float f;

 f = pow(2.1, 6);
 printf("2.1 ^ 6 = %f\n", f);
 return 0;
}
```

and I compiled it as:

```
% cc temp.c -lm
```

like you said I should, but I get this when I run it:

```
% ./a.out
2.1 ^ 6 = 1023.000000
```

This is *not* the right answer! What is going on?

When the compiler sees you call a function, it checks if it has already seen a prototype for it. If it has not, it assumes the function returns an int, which is definitely not what you want here.

3. So how do I fix this?

The prototypes for the mathematical functions are in `math.h`. If you include this file, the compiler will be able to find the prototype and it will stop doing strange things to your calculation!

```
#include <math.h>
#include <stdio.h>

int main() {
...
```

After recompiling it as you did before, run it:

```
% ./a.out
2.1 ^ 6 = 85.766121
```

If you are using any of the mathematical functions, *always* include `math.h` and remember to link in the math library.

4. I compiled a file called `foobar.c` and I cannot find an executable called `foobar`. Where's it gone?

Remember, `cc` will call the executable `a.out` unless you tell it differently. Use the `-o filename` option:

```
% cc -o foobar foobar.c
```

5. OK, I have an executable called `foobar`, I can see it when I run `ls`, but when I type in `foobar` at the command prompt it tells me there is no such file. Why can it not find it?

Unlike MS-DOS, Unix does not look in the current directory when it is trying to find out which executable you want it to run, unless you tell it to. Either type `./foobar`, which means "run the file called `foobar` in the current directory", or change your PATH environment variable so that it looks something like

```
bin:/usr/bin:/usr/local/bin:.
```

The dot at the end means “look in the current directory if it is not in any of the others”.

6. I called my executable `test`, but nothing happens when I run it. What is going on?

Most Unix systems have a program called `test` in `/usr/bin` and the shell is picking that one up before it gets to checking the current directory. Either type:

```
% ./test
```

or choose a better name for your program!

7. I compiled my program and it seemed to run all right at first, then there was an error and it said something about `core dumped`. What does that mean?

The name *core dump* dates back to the very early days of Unix, when the machines used core memory for storing data. Basically, if the program failed under certain conditions, the system would write the contents of core memory to disk in a file called `core`, which the programmer could then pore over to find out what went wrong.

8. Fascinating stuff, but what I am supposed to do now?

Use `gdb` to analyse the core (see Section 5.6).

9. When my program dumped core, it said something about a `segmentation fault`. What’s that?

This basically means that your program tried to perform some sort of illegal operation on memory; Unix is designed to protect the operating system and other programs from rogue programs.

Common causes for this are:

- Trying to write to a NULL pointer, eg

  ```
  char *foo = NULL;
  strcpy(foo, "bang!");
  ```

- Using a pointer that hasn’t been initialised, eg

  ```
  char *foo;
  strcpy(foo, "bang!");
  ```

The pointer will have some random value that, with luck, will point into an area of memory that isn't available to your program and the kernel will kill your program before it can do any damage. If you're unlucky, it'll point somewhere inside your own program and corrupt one of your data structures, causing the program to fail mysteriously.

- Trying to access past the end of an array, eg

  ```
  int bar[20];
  bar[27] = 6;
  ```

- Trying to store something in read-only memory, eg

  ```
  char *foo = "My string";
  strcpy(foo, "bang!");
  ```

  Unix compilers often put string literals like `"My string"` into read-only areas of memory.

- Doing naughty things with `malloc()` and `free()`, eg

  ```
  char bar[80];
  free(bar);
  ```

  or

  ```
  char *foo = malloc(27);
  free(foo);
  free(foo);
  ```

Making one of these mistakes will not always lead to an error, but they are always bad practice. Some systems and compilers are more tolerant than others, which is why programs that ran well on one system can crash when you try them on an another.

10. Sometimes when I get a core dump it says `bus error`. It says in my Unix book that this means a hardware problem, but the computer still seems to be working. Is this true?

No, fortunately not (unless of course you really do have a hardware problem...). This is usually another way of saying that you accessed memory in a way you shouldn't have.

11. This dumping core business sounds as though it could be quite useful, if I can make it happen when I want to. Can I do this, or do I have to wait until there's an error?

Yes, just go to another console or xterm, do

```
% ps
```

to find out the process ID of your program, and do

```
% kill -ABRT pid
```

where *`pid`* is the process ID you looked up.

This is useful if your program has got stuck in an infinite loop, for instance. If your program happens to trap SIGABRT, there are several other signals which have a similar effect.

Alternatively, you can create a core dump from inside your program, by calling the `abort()` function. See the man page of abort(3) to learn more.

If you want to create a core dump from outside your program, but don't want the process to terminate, you can use the `gcore` program. See the man page of gcore(1) for more information.

5.5. Make

5.5.1. What is `make`?

When you're working on a simple program with only one or two source files, typing in

```
% cc file1.c file2.c
```

is not too bad, but it quickly becomes very tedious when there are several files—and it can take a while to compile, too.

One way to get around this is to use object files and only recompile the source file if the source code has changed. So we could have something like:

```
% cc file1.o file2.o ... file37.c ...
```

if we'd changed `file37.c`, but not any of the others, since the last time we compiled. This may speed up the compilation quite a bit, but doesn't solve the typing problem.

Or we could write a shell script to solve the typing problem, but it would have to re-compile everything, making it very inefficient on a large project.

What happens if we have hundreds of source files lying about? What if we're working in a team with other people who forget to tell us when they've changed one of their source files that we use?

Perhaps we could put the two solutions together and write something like a shell script that would contain some kind of magic rule saying when a source file needs compiling. Now all we need now is a program that can understand these rules, as it's a bit too complicated for the shell.

This program is called `make`. It reads in a file, called a *makefile*, that tells it how different files depend on each other, and works out which files need to be re-compiled and which ones don't. For example, a rule could say something like "if `fromboz.o` is older than `fromboz.c`, that means someone must have changed `fromboz.c`, so it needs to be re-compiled." The makefile also has rules telling make *how* to re-compile the source file, making it a much more powerful tool.

Makefiles are typically kept in the same directory as the source they apply to, and can be called `makefile`, `Makefile` or `MAKEFILE`. Most programmers use the name `Makefile`, as this puts it near the top of a directory listing, where it can easily be seen. [6]

5.5.2. Example of using `make`

Here's a very simple make file:

```
foo: foo.c
 cc -o foo foo.c
```

It consists of two lines, a dependency line and a creation line.

The dependency line here consists of the name of the program (known as the *target*), followed by a colon, then whitespace, then the name of the source file. When `make` reads this line, it looks to see if `foo` exists; if it exists, it compares the time `foo` was last modified to the time `foo.c` was last modified. If `foo` does not exist, or is older than `foo.c`, it then looks at the creation line to find out what to do. In other words, this is the rule for working out when `foo.c` needs to be re-compiled.

The creation line starts with a tab (press the **tab** key) and then the command you would type to create `foo` if you were doing it at a command prompt. If `foo` is out of date, or does not exist, `make` then executes this command to create it. In other words, this is the rule which tells make how to re-compile `foo.c`.

So, when you type **`make`**, it will make sure that `foo` is up to date with respect to your latest changes to `foo.c`. This principle can be extended to `Makefile`s with hundreds of targets—in fact, on FreeBSD, it is possible to compile the entire operating system just by typing **`make world`** in the appropriate directory!

Another useful property of makefiles is that the targets don't have to be programs. For instance, we could have a make file that looks like this:

```
foo: foo.c
 cc -o foo foo.c

install:
 cp foo /home/me
```

We can tell make which target we want to make by typing:

```
% make target
```

`make` will then only look at that target and ignore any others. For example, if we type **`make foo`** with the makefile above, make will ignore the install target.

If we just type **`make`** on its own, make will always look at the first target and then stop without looking at any others. So if we typed **`make`** here, it will just go to the foo target, re-compile `foo` if necessary, and then stop without going on to the install target.

Notice that the install target doesn't actually depend on anything! This means that the command on the following line is always executed when we try to make that target by typing **`make install`**. In this case, it will copy `foo` into the user's home directory. This is often used by application makefiles, so that the application can be installed in the correct directory when it has been correctly compiled.

This is a slightly confusing subject to try and explain. If you don't quite understand how `make` works, the best thing to do is to write a simple program like "hello world" and a make file like the one above and experiment. Then progress to using more than one source file, or having the source file include a header file. The `touch` command is very useful here—it changes the date on a file without you having to edit it.

5.5.3. Make and include-files

C code often starts with a list of files to include, for example stdio.h. Some of these files are system-include files, some of them are from the project you're now working on:

```
#include <stdio.h>
#include "foo.h"

int main(....
```

To make sure that this file is recompiled the moment `foo.h` is changed, you have to add it in your `Makefile`:

```
foo: foo.c foo.h
```

The moment your project is getting bigger and you have more and more own include-files to maintain, it will be a pain to keep track of all include files and the files which are depending on it. If you change an include-file but forget to recompile all the files which are depending on it, the results will be devastating. `gcc` has an option to analyze your files and to produce a list of include-files and their dependencies: `-MM`.

If you add this to your Makefile:

```
depend:
 gcc -E -MM *.c > .depend
```

and run **`make depend`**, the file `.depend` will appear with a list of object-files, C-files and the include-files:

```
foo.o: foo.c foo.h
```

If you change `foo.h`, next time you run `make` all files depending on `foo.h` will be recompiled.

Don’t forget to run `make depend` each time you add an include-file to one of your files.

5.5.4. FreeBSD Makefiles

Makefiles can be rather complicated to write. Fortunately, BSD-based systems like FreeBSD come with some very powerful ones as part of the system. One very good example of this is the FreeBSD ports system. Here’s the essential part of a typical ports `Makefile`:

```
MASTER_SITES=   ftp://freefall.cdrom.com/pub/FreeBSD/LOCAL_PORTS/
DISTFILES=      scheme-microcode+dist-7.3-freebsd.tgz

.include <bsd.port.mk>
```

Now, if we go to the directory for this port and type **`make`**, the following happens:

1. A check is made to see if the source code for this port is already on the system.
2. If it isn’t, an FTP connection to the URL in MASTER_SITES is set up to download the source.

3. The checksum for the source is calculated and compared it with one for a known, good, copy of the source. This is to make sure that the source was not corrupted while in transit.
4. Any changes required to make the source work on FreeBSD are applied—this is known as *patching*.
5. Any special configuration needed for the source is done. (Many Unix program distributions try to work out which version of Unix they are being compiled on and which optional Unix features are present—this is where they are given the information in the FreeBSD ports scenario).
6. The source code for the program is compiled. In effect, we change to the directory where the source was unpacked and do `make`—the program's own make file has the necessary information to build the program.
7. We now have a compiled version of the program. If we wish, we can test it now; when we feel confident about the program, we can type **`make install`**. This will cause the program and any supporting files it needs to be copied into the correct location; an entry is also made into a package database, so that the port can easily be uninstalled later if we change our mind about it.

Now I think you'll agree that's rather impressive for a four line script!

The secret lies in the last line, which tells `make` to look in the system makefile called `bsd.port.mk`. It's easy to overlook this line, but this is where all the clever stuff comes from—someone has written a makefile that tells `make` to do all the things above (plus a couple of other things I didn't mention, including handling any errors that may occur) and anyone can get access to that just by putting a single line in their own make file!

If you want to have a look at these system makefiles, they're in `/usr/share/mk`, but it's probably best to wait until you've had a bit of practice with makefiles, as they are very complicated (and if you do look at them, make sure you have a flask of strong coffee handy!)

5.5.5. More advanced uses of `make`

`Make` is a very powerful tool, and can do much more than the simple example above shows. Unfortunately, there are several different versions of `make`, and they all differ considerably. The best way to learn what they can do is probably to read the documentation—hopefully this introduction will have given you a base from which you can do this.

The version of make that comes with FreeBSD is the **Berkeley make**; there is a tutorial for it in `/usr/share/doc/psd/12.make`. To view it, do

```
% zmore paper.ascii.gz
```

in that directory.

Many applications in the ports use **GNU make**, which has a very good set of "info" pages. If you have installed any of these ports, **GNU make** will automatically have been installed as `gmake`. It's also available as a port and package in its own right.

To view the info pages for **GNU make**, you will have to edit the `dir` file in the `/usr/local/info` directory to add an entry for it. This involves adding a line like

```
* Make: (make).                 The GNU Make utility.
```

to the file. Once you have done this, you can type **`info`** and then select `make` from the menu (or in **Emacs**, do **`C-h i`**).

5.6. Debugging

5.6.1. The Debugger

The debugger that comes with FreeBSD is called `gdb` (**GNU debugger**). You start it up by typing

```
% gdb progname
```

although most people prefer to run it inside **Emacs**. You can do this by:

```
M-x gdb RET progname RET
```

Using a debugger allows you to run the program under more controlled circumstances. Typically, you can step through the program a line at a time, inspect the value of variables, change them, tell the debugger to run up to a certain point and then stop, and so on. You can even attach to a program that's already running, or load a core file to investigate why the program crashed. It's even possible to debug the kernel, though that's a little trickier than the user applications we'll be discussing in this section.

`gdb` has quite good on-line help, as well as a set of info pages, so this section will concentrate on a few of the basic commands.

Finally, if you find its text-based command-prompt style off-putting, there's a graphical front-end for it xxgdb (../../ports/devel.html) in the ports collection.

This section is intended to be an introduction to using `gdb` and does not cover specialised topics such as debugging the kernel.

5.6.2. Running a program in the debugger

You'll need to have compiled the program with the `-g` option to get the most out of using `gdb`. It will work without, but you'll only see the name of the function you're in, instead of the source code. If you see a line like:

```
... (no debugging symbols found) ...
```

when `gdb` starts up, you'll know that the program wasn't compiled with the `-g` option.

At the `gdb` prompt, type **`break main`**. This will tell the debugger to skip over the preliminary set-up code in the program and start at the beginning of your code. Now type **`run`** to start the program—it will start at the beginning of the set-up code and then get stopped by the debugger when it calls `main()`. (If you've ever wondered where `main()` gets called from, now you know!).

You can now step through the program, a line at a time, by pressing `n`. If you get to a function call, you can step into it by pressing `s`. Once you're in a function call, you can return from stepping into a function call by pressing `f`. You can also use `up` and `down` to take a quick look at the caller.

Here's a simple example of how to spot a mistake in a program with `gdb`. This is our program (with a deliberate mistake):

```
#include <stdio.h>

int bazz(int anint);

main() {
 int i;

 printf("This is my program\n");
 bazz(i);
 return 0;
}

int bazz(int anint) {
 printf("You gave me %d\n", anint);
 return anint;
}
```

This program sets i to be `5` and passes it to a function `bazz()` which prints out the number we gave it.

When we compile and run the program we get

```
% cc -g -o temp temp.c
% ./temp
This is my program
anint = 4231
```

That wasn’t what we expected! Time to see what’s going on!

```
% gdb temp
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (i386-unknown-freebsd), Copyright 1994 Free Software Foundation, Inc.
(gdb) break main Skip the set-up code
Breakpoint 1 at 0x160f: file temp.c, line 9. gdb puts breakpoint at main()
(gdb) run Run as far as main()
Starting program: /home/james/tmp/temp Program starts running

Breakpoint 1, main () at temp.c:9 gdb stops at main()
(gdb) n Go to next line
This is my program Program prints out
(gdb) s step into bazz()
bazz (anint=4231) at temp.c:17 gdb displays stack frame
(gdb)
```

Hang on a minute! How did anint get to be `4231`? Didn’t we set it to be `5` in `main()`? Let’s move up to `main()` and have a look.

```
(gdb) up Move up call stack
#1  0x1625 in main () at temp.c:11 gdb displays stack frame
(gdb) p i Show us the value of i
$1 = 4231 gdb displays 4231
```

Oh dear! Looking at the code, we forgot to initialise i. We meant to put

```
…
main() {
 int i;

 i = 5;
 printf("This is my program\n");
…
```

but we left the `i=5;` line out. As we didn’t initialise i, it had whatever number happened to be in that area of memory when the program ran, which in this case happened to be `4231`.

> **Note:** `gdb` displays the stack frame every time we go into or out of a function, even if we're using `up` and `down` to move around the call stack. This shows the name of the function and the values of its arguments, which helps us keep track of where we are and what's going on. (The stack is a storage area where the program stores information about the arguments passed to functions and where to go when it returns from a function call).

5.6.3. Examining a core file

A core file is basically a file which contains the complete state of the process when it crashed. In "the good old days", programmers had to print out hex listings of core files and sweat over machine code manuals, but now life is a bit easier. Incidentally, under FreeBSD and other 4.4BSD systems, a core file is called *`progname`*`.core` instead of just `core`, to make it clearer which program a core file belongs to.

To examine a core file, start up `gdb` in the usual way. Instead of typing `break` or `run`, type

```
(gdb) core progname.core
```

If you're not in the same directory as the core file, you'll have to do **`dir /path/to/core/file`** first.

You should see something like this:

```
% gdb a.out
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.13 (i386-unknown-freebsd), Copyright 1994 Free Software Foundation, Inc.
(gdb) core a.out.core
Core was generated by 'a.out'.
Program terminated with signal 11, Segmentation fault.
Cannot access memory at address 0x7020796d.
#0  0x164a in bazz (anint=0x5) at temp.c:17
(gdb)
```

In this case, the program was called `a.out`, so the core file is called `a.out.core`. We can see that the program crashed due to trying to access an area in memory that was not available to it in a function called `bazz`.

Sometimes it's useful to be able to see how a function was called, as the problem could have occurred a long way up the call stack in a complex program. The `bt` command causes `gdb` to print out a back-trace of the call stack:

```
(gdb) bt
#0  0x164a in bazz (anint=0x5) at temp.c:17
#1  0xefbfd888 in end ()
```

```
#2  0x162c in main () at temp.c:11
(gdb)
```

The `end()` function is called when a program crashes; in this case, the `bazz()` function was called from `main()`.

5.6.4. Attaching to a running program

One of the neatest features about `gdb` is that it can attach to a program that's already running. Of course, that assumes you have sufficient permissions to do so. A common problem is when you are stepping through a program that forks, and you want to trace the child, but the debugger will only let you trace the parent.

What you do is start up another `gdb`, use `ps` to find the process ID for the child, and do

```
(gdb) attach pid
```

in `gdb`, and then debug as usual.

"That's all very well," you're probably thinking, "but by the time I've done that, the child process will be over the hill and far away". Fear not, gentle reader, here's how to do it (courtesy of the `gdb` info pages):

```
...
if ((pid = fork()) < 0) /* _Always_ check this */
 error();
else if (pid == 0) { /* child */
 int PauseMode = 1;

 while (PauseMode)
 sleep(10); /* Wait until someone attaches to us */
 ...
} else { /* parent */
 ...
```

Now all you have to do is attach to the child, set PauseMode to `0`, and wait for the `sleep()` call to return!

5.7. Using Emacs as a Development Environment

5.7.1. Emacs

Unfortunately, Unix systems don’t come with the kind of everything-you-ever-wanted-and-lots-more-you-didn’t-in-one-gigantic-package integrated development environments that other systems have. [7] However, it is possible to set up your own environment. It may not be as pretty, and it may not be quite as integrated, but you can set it up the way you want it. And it’s free. And you have the source to it.

The key to it all is Emacs. Now there are some people who loathe it, but many who love it. If you’re one of the former, I’m afraid this section will hold little of interest to you. Also, you’ll need a fair amount of memory to run it—I’d recommend 8MB in text mode and 16MB in X as the bare minimum to get reasonable performance.

Emacs is basically a highly customisable editor—indeed, it has been customised to the point where it’s more like an operating system than an editor! Many developers and sysadmins do in fact spend practically all their time working inside Emacs, leaving it only to log out.

It’s impossible even to summarise everything Emacs can do here, but here are some of the features of interest to developers:

- Very powerful editor, allowing search-and-replace on both strings and regular expressions (patterns), jumping to start/end of block expression, etc, etc.
- Pull-down menus and online help.
- Language-dependent syntax highlighting and indentation.
- Completely customisable.
- You can compile and debug programs within Emacs.
- On a compilation error, you can jump to the offending line of source code.
- Friendly-ish front-end to the `info` program used for reading GNU hypertext documentation, including the documentation on Emacs itself.
- Friendly front-end to `gdb`, allowing you to look at the source code as you step through your program.
- You can read Usenet news and mail while your program is compiling.

And doubtless many more that I’ve overlooked.

Emacs can be installed on FreeBSD using the Emacs port (../../ports/editors.html).

Once it’s installed, start it up and do **`C-h t`** to read an Emacs tutorial—that means hold down the **control** key, press **h**, let go of the **control** key, and then press **t**. (Alternatively, you can you use the mouse to select Emacs Tutorial from the Help menu).

Although Emacs does have menus, it's well worth learning the key bindings, as it's much quicker when you're editing something to press a couple of keys than to try and find the mouse and then click on the right place. And, when you're talking to seasoned Emacs users, you'll find they often casually throw around expressions like "`M-x replace-s RET foo RET bar RET`" so it's useful to know what they mean. And in any case, Emacs has far too many useful functions for them to all fit on the menu bars.

Fortunately, it's quite easy to pick up the key-bindings, as they're displayed next to the menu item. My advice is to use the menu item for, say, opening a file until you understand how it works and feel confident with it, then try doing C-x C-f. When you're happy with that, move on to another menu command.

If you can't remember what a particular combination of keys does, select Describe Key from the Help menu and type it in—Emacs will tell you what it does. You can also use the Command Apropos menu item to find out all the commands which contain a particular word in them, with the key binding next to it.

By the way, the expression above means hold down the Meta key, press x, release the Meta key, type **`replace-s`** (short for `replace-string`—another feature of Emacs is that you can abbreviate commands), press the return key, type **`foo`** (the string you want replaced), press the return key, type bar (the string you want to replace `foo` with) and press return again. Emacs will then do the search-and-replace operation you've just requested.

If you're wondering what on earth the Meta key is, it's a special key that many Unix workstations have. Unfortunately, PC's don't have one, so it's usually the **alt** key (or if you're unlucky, the escape key).

Oh, and to get out of Emacs, do `C-x C-c` (that means hold down the control key, press x, press c and release the control key). If you have any unsaved files open, Emacs will ask you if you want to save them. (Ignore the bit in the documentation where it says `C-z` is the usual way to leave Emacs—that leaves Emacs hanging around in the background, and is only really useful if you're on a system which doesn't have virtual terminals).

5.7.2. Configuring Emacs

Emacs does many wonderful things; some of them are built in, some of them need to be configured.

Instead of using a proprietary macro language for configuration, Emacs uses a version of Lisp specially adapted for editors, known as Emacs Lisp. This can be quite useful if you want to go on and learn something like Common Lisp, as it's considerably smaller than Common Lisp (although still quite big!).

The best way to learn Emacs Lisp is to download the Emacs Tutorial (ftp://prep.ai.mit.edu:pub/gnu/elisp-manual-19-2.4.tar.gz)

However, there's no need to actually know any Lisp to get started with configuring Emacs, as I've included a sample `.emacs` file, which should be enough to get you started. Just copy it into your home directory and restart Emacs if it's already running; it will read the commands from the file and (hopefully) give you a useful basic setup.

5.7.3. A sample `.emacs` file

Unfortunately, there's far too much here to explain it in detail; however there are one or two points worth mentioning.

- Everything beginning with a `;` is a comment and is ignored by Emacs.
- In the first line, the `-*- Emacs-Lisp -*-` is so that we can edit the `.emacs` file itself within Emacs and get all the fancy features for editing Emacs Lisp. Emacs usually tries to guess this based on the filename, and may not get it right for `.emacs`.
- The tab key is bound to an indentation function in some modes, so when you press the tab key, it will indent the current line of code. If you want to put a tab character in whatever you're writing, hold the control key down while you're pressing the tab key.
- This file supports syntax highlighting for C, C++, Perl, Lisp and Scheme, by guessing the language from the filename.
- Emacs already has a pre-defined function called `next-error`. In a compilation output window, this allows you to move from one compilation error to the next by doing `M-n`; we define a complementary function, `previous-error`, that allows you to go to a previous error by doing `M-p`. The nicest feature of all is that `C-c C-c` will open up the source file in which the error occurred and jump to the appropriate line.
- We enable Emacs's ability to act as a server, so that if you're doing something outside Emacs and you want to edit a file, you can just type in

  ```
  % emacsclient filename
  ```

  and then you can edit the file in your Emacs! [8]

Example 5-1. A sample `.emacs` file

```
;; -*-Emacs-Lisp-*-

;; This file is designed to be re-evaled; use the variable first-time
;; to avoid any problems with this.
(defvar first-time t
  "Flag signifying this is the first time that .emacs has been evaled")

;; Meta
(global-set-key "\M- " ’set-mark-command)
(global-set-key "\M-\C-h" ’backward-kill-word)
(global-set-key "\M-\C-r" ’query-replace)
(global-set-key "\M-r" ’replace-string)
(global-set-key "\M-g" ’goto-line)
```

```
(global-set-key "\M-h" ’help-command)

;; Function keys
(global-set-key [f1] ’manual-entry)
(global-set-key [f2] ’info)
(global-set-key [f3] ’repeat-complex-command)
(global-set-key [f4] ’advertised-undo)
(global-set-key [f5] ’eval-current-buffer)
(global-set-key [f6] ’buffer-menu)
(global-set-key [f7] ’other-window)
(global-set-key [f8] ’find-file)
(global-set-key [f9] ’save-buffer)
(global-set-key [f10] ’next-error)
(global-set-key [f11] ’compile)
(global-set-key [f12] ’grep)
(global-set-key [C-f1] ’compile)
(global-set-key [C-f2] ’grep)
(global-set-key [C-f3] ’next-error)
(global-set-key [C-f4] ’previous-error)
(global-set-key [C-f5] ’display-faces)
(global-set-key [C-f8] ’dired)
(global-set-key [C-f10] ’kill-compilation)

;; Keypad bindings
(global-set-key [up] "\C-p")
(global-set-key [down] "\C-n")
(global-set-key [left] "\C-b")
(global-set-key [right] "\C-f")
(global-set-key [home] "\C-a")
(global-set-key [end] "\C-e")
(global-set-key [prior] "\M-v")
(global-set-key [next] "\C-v")
(global-set-key [C-up] "\M-\C-b")
(global-set-key [C-down] "\M-\C-f")
(global-set-key [C-left] "\M-b")
(global-set-key [C-right] "\M-f")
(global-set-key [C-home] "\M-<")
(global-set-key [C-end] "\M->")
(global-set-key [C-prior] "\M-<")
(global-set-key [C-next] "\M->")

;; Mouse
(global-set-key [mouse-3] ’imenu)

;; Misc
```

```
(global-set-key [C-tab] "\C-q\t") ; Control tab quotes a tab.
(setq backup-by-copying-when-mismatch t)

;; Treat ’y’ or <CR> as yes, ’n’ as no.
(fset ’yes-or-no-p ’y-or-n-p)
    (define-key query-replace-map [return] ’act)
    (define-key query-replace-map [?\C-m] ’act)

;; Load packages
(require ’desktop)
(require ’tar-mode)

;; Pretty diff mode
(autoload ’ediff-buffers "ediff" "Intelligent Emacs interface to diff" t)
(autoload ’ediff-files "ediff" "Intelligent Emacs interface to diff" t)
(autoload ’ediff-files-remote "ediff"
  "Intelligent Emacs interface to diff")

(if first-time
    (setq auto-mode-alist
   (append ’(("\\.cpp$" . c++-mode)
     ("\\.hpp$" . c++-mode)
                    ("\\.lsp$" . lisp-mode)
     ("\\.scm$" . scheme-mode)
     ("\\.pl$" . perl-mode)
     ) auto-mode-alist)))

;; Auto font lock mode
(defvar font-lock-auto-mode-list
  (list ’c-mode ’c++-mode ’c++-c-mode ’emacs-lisp-mode ’lisp-mode ’perl-mode ’scheme-
mode)
  "List of modes to always start in font-lock-mode")

(defvar font-lock-mode-keyword-alist
  ’((c++-c-mode . c-font-lock-keywords)
    (perl-mode . perl-font-lock-keywords))
  "Associations between modes and keywords")

(defun font-lock-auto-mode-select ()
  "Automatically select font-lock-mode if the current major mode is
in font-lock-auto-mode-list"
  (if (memq major-mode font-lock-auto-mode-list)
      (progn
 (font-lock-mode t))
    )
  )

(global-set-key [M-f1] ’font-lock-fontify-buffer)

;; New dabbrev stuff
;(require ’new-dabbrev)
(setq dabbrev-always-check-other-buffers t)
(setq dabbrev-abbrev-char-regexp "\\sw\\|\\s_")
(add-hook ’emacs-lisp-mode-hook
   ’(lambda ()
      (set (make-local-variable ’dabbrev-case-fold-search) nil)
      (set (make-local-variable ’dabbrev-case-replace) nil)))
(add-hook ’c-mode-hook
   ’(lambda ()
      (set (make-local-variable ’dabbrev-case-fold-search) nil)
      (set (make-local-variable ’dabbrev-case-replace) nil)))
(add-hook ’text-mode-hook
   ’(lambda ()
      (set (make-local-variable ’dabbrev-case-fold-search) t)
      (set (make-local-variable ’dabbrev-case-replace) t)))

;; C++ and C mode...
(defun my-c++-mode-hook ()
  (setq tab-width 4)
  (define-key c++-mode-map "\C-m" ’reindent-then-newline-and-indent)
  (define-key c++-mode-map "\C-ce" ’c-comment-edit)
  (setq c++-auto-hungry-initial-state ’none)
  (setq c++-delete-function ’backward-delete-char)
  (setq c++-tab-always-indent t)
  (setq c-indent-level 4)
  (setq c-continued-statement-offset 4)
  (setq c++-empty-arglist-indent 4))

(defun my-c-mode-hook ()
  (setq tab-width 4)
  (define-key c-mode-map "\C-m" ’reindent-then-newline-and-indent)
  (define-key c-mode-map "\C-ce" ’c-comment-edit)
  (setq c-auto-hungry-initial-state ’none)
  (setq c-delete-function ’backward-delete-char)
  (setq c-tab-always-indent t)
;; BSD-ish indentation style
  (setq c-indent-level 4)
  (setq c-continued-statement-offset 4)
  (setq c-brace-offset -4)
  (setq c-argdecl-indent 0)
  (setq c-label-offset -4))

;; Perl mode
(defun my-perl-mode-hook ()
  (setq tab-width 4)
  (define-key c++-mode-map "\C-m" ’reindent-then-newline-and-indent)
  (setq perl-indent-level 4)
  (setq perl-continued-statement-offset 4))

;; Scheme mode...
(defun my-scheme-mode-hook ()
  (define-key scheme-mode-map "\C-m" ’reindent-then-newline-and-indent))

;; Emacs-Lisp mode...
(defun my-lisp-mode-hook ()
  (define-key lisp-mode-map "\C-m" ’reindent-then-newline-and-indent)
  (define-key lisp-mode-map "\C-i" ’lisp-indent-line)
  (define-key lisp-mode-map "\C-j" ’eval-print-last-sexp))

;; Add all of the hooks...
(add-hook ’c++-mode-hook ’my-c++-mode-hook)
(add-hook ’c-mode-hook ’my-c-mode-hook)
(add-hook ’scheme-mode-hook ’my-scheme-mode-hook)
(add-hook ’emacs-lisp-mode-hook ’my-lisp-mode-hook)
(add-hook ’lisp-mode-hook ’my-lisp-mode-hook)
(add-hook ’perl-mode-hook ’my-perl-mode-hook)

;; Complement to next-error
(defun previous-error (n)
  "Visit previous compilation error message and corresponding source code."
  (interactive "p")
  (next-error (- n)))

;; Misc...
(transient-mark-mode 1)
(setq mark-even-if-inactive t)
(setq visible-bell nil)
(setq next-line-add-newlines nil)
(setq compile-command "make")
(setq suggest-key-bindings nil)
(put ’eval-expression ’disabled nil)
(put ’narrow-to-region ’disabled nil)
(put ’set-goal-column ’disabled nil)

;; Elisp archive searching
```

```
(autoload ’format-lisp-code-directory "lispdir" nil t)
(autoload ’lisp-dir-apropos "lispdir" nil t)
(autoload ’lisp-dir-retrieve "lispdir" nil t)
(autoload ’lisp-dir-verify "lispdir" nil t)

;; Font lock mode
(defun my-make-face (face colour &optional bold)
  "Create a face from a colour and optionally make it bold"
  (make-face face)
  (copy-face ’default face)
  (set-face-foreground face colour)
  (if bold (make-face-bold face))
  )

(if (eq window-system ’x)
    (progn
      (my-make-face ’blue "blue")
      (my-make-face ’red "red")
      (my-make-face ’green "dark green")
      (setq font-lock-comment-face ’blue)
      (setq font-lock-string-face ’bold)
      (setq font-lock-type-face ’bold)
      (setq font-lock-keyword-face ’bold)
      (setq font-lock-function-name-face ’red)
      (setq font-lock-doc-string-face ’green)
      (add-hook ’find-file-hooks ’font-lock-auto-mode-select)

      (setq baud-rate 1000000)
      (global-set-key "\C-cmm" ’menu-bar-mode)
      (global-set-key "\C-cms" ’scroll-bar-mode)
      (global-set-key [backspace] ’backward-delete-char)
 ;      (global-set-key [delete] ’delete-char)
      (standard-display-european t)
      (load-library "iso-transl")))

;; X11 or PC using direct screen writes
(if window-system
    (progn
      ;;      (global-set-key [M-f1] ’hilit-repaint-command)
      ;;      (global-set-key [M-f2] [?\C-u M-f1])
      (setq hilit-mode-enable-list
     ’(not text-mode c-mode c++-mode emacs-lisp-mode lisp-mode
   scheme-mode)
     hilit-auto-highlight nil
     hilit-auto-rehighlight ’visible
     hilit-inhibit-hooks nil
     hilit-inhibit-rebinding t)
      (require ’hilit19)
      (require ’paren))
  (setq baud-rate 2400) ; For slow serial connections
  )

;; TTY type terminal
(if (and (not window-system)
  (not (equal system-type ’ms-dos)))
    (progn
      (if first-time
   (progn
     (keyboard-translate ?\C-h ?\C-?)
     (keyboard-translate ?\C-? ?\C-h)))))

;; Under UNIX
(if (not (equal system-type ’ms-dos))
    (progn
      (if first-time
   (server-start))))

;; Add any face changes here
(add-hook ’term-setup-hook ’my-term-setup-hook)
(defun my-term-setup-hook ()
  (if (eq window-system ’pc)
      (progn
;; (set-face-background ’default "red")
 )))

;; Restore the "desktop" - do this as late as possible
(if first-time
    (progn
      (desktop-load-default)
      (desktop-read)))

;; Indicate that this file has been read at least once
(setq first-time nil)

;; No need to debug anything now

(setq debug-on-error nil)

;; All done
(message "All done, %s%s" (user-login-name) ".")
```

5.7.4. Extending the Range of Languages Emacs Understands

Now, this is all very well if you only want to program in the languages already catered for in the `.emacs` file (C, C++, Perl, Lisp and Scheme), but what happens if a new language called “whizbang” comes out, full of exciting features?

The first thing to do is find out if whizbang comes with any files that tell Emacs about the language. These usually end in `.el`, short for “Emacs Lisp”. For example, if whizbang is a FreeBSD port, we can locate these files by doing

```
% find /usr/ports/lang/whizbang -name "*.el" -print
```

and install them by copying them into the Emacs site Lisp directory. On FreeBSD 2.1.0-RELEASE, this is `/usr/local/share/emacs/site-lisp`.

So for example, if the output from the find command was

```
/usr/ports/lang/whizbang/work/misc/whizbang.el
```

we would do

```
# cp /usr/ports/lang/whizbang/work/misc/whizbang.el /usr/local/share/emacs/site-lisp
```

Next, we need to decide what extension whizbang source files have. Let’s say for the sake of argument that they all end in `.wiz`. We need to add an entry to our `.emacs` file to make sure Emacs will be able to use the information in `whizbang.el`.

Find the auto-mode-alist entry in `.emacs` and add a line for whizbang, such as:

```
...
("\\.lsp$" . lisp-mode)
("\\.wiz$" . whizbang-mode)
("\\.scm$" . scheme-mode)
...
```

This means that Emacs will automatically go into `whizbang-mode` when you edit a file ending in `.wiz`.

Just below this, you’ll find the font-lock-auto-mode-list entry. Add `whizbang-mode` to it like so:

```
;; Auto font lock mode
(defvar font-lock-auto-mode-list
  (list ’c-mode ’c++-mode ’c++-c-mode ’emacs-lisp-mode ’whizbang-mode ’lisp-mode ’perl-
mode ’scheme-mode)
  "List of modes to always start in font-lock-mode")
```

This means that Emacs will always enable `font-lock-mode` (ie syntax highlighting) when editing a `.wiz` file.

And that’s all that’s needed. If there’s anything else you want done automatically when you open up a `.wiz` file, you can add a `whizbang-mode hook` (see `my-scheme-mode-hook` for a simple example that adds `auto-indent`).

5.8. Further Reading

- Brian Harvey and Matthew Wright *Simply Scheme* MIT 1994. ISBN 0-262-08226-8
- Randall Schwartz *Learning Perl* O’Reilly 1993 ISBN 1-56592-042-2
- Patrick Henry Winston and Berthold Klaus Paul Horn *Lisp (3rd Edition)* Addison-Wesley 1989 ISBN 0-201-08319-1
- Brian W. Kernighan and Rob Pike *The Unix Programming Environment* Prentice-Hall 1984 ISBN 0-13-937681-X
- Brian W. Kernighan and Dennis M. Ritchie *The C Programming Language (2nd Edition)* Prentice-Hall 1988 ISBN 0-13-110362-8
- Bjarne Stroustrup *The C++ Programming Language* Addison-Wesley 1991 ISBN 0-201-53992-6
- W. Richard Stevens *Advanced Programming in the Unix Environment* Addison-Wesley 1992 ISBN 0-201-56317-7
- W. Richard Stevens *Unix Network Programming* Prentice-Hall 1990 ISBN 0-13-949876-1

Notes

1. If you run it in the shell, you may get a core dump.
2. To be strictly accurate, `cc` converts the source code into its own, machine-independent *p-code* instead of assembly language at this stage.
3. In case you didn’t know, a binary sort is an efficient way of sorting things into order and a bubble sort isn’t.
4. The reasons for this are buried in the mists of history.
5. Note, we didn’t use the `-o` flag to specify the executable name, so we will get an executable called `a.out`. Producing a debug version called `foobar` is left as an exercise for the reader!
6. They don’t use the `MAKEFILE` form as block capitals are often used for documentation files like `README`.

7. At least, not unless you pay out very large sums of money.
8. Many Emacs users set their EDITOR environment to `emacsclient` so this happens every time they need to edit a file.

Chapter 6. Secure Programming

This chapter was written by Murray Stokely <`murray@FreeBSD.org`>.

6.1. Synopsis

This chapter describes some of the security issues that have plagued Unix programmers for decades and some of the new tools available to help programmers avoid writing exploitable code.

6.2. Secure Design Methodology

Writing secure applications takes a very scrutinous and pessimistic outlook on life. Applications should be run with the principle of “least privilege” so that no process is ever running with more than the bare minimum access that it needs to accomplish its function. Previously tested code should be reused whenever possible to avoid common mistakes that others may have already fixed.

One of the pitfalls of the Unix environment is how easy it is to make assumptions about the sanity of the environment. Applications should never trust user input (in all its forms), system resources, inter-process communication, or the timing of events. Unix processes do not execute synchronously so logical operations are rarely atomic.

6.3. Buffer Overflows

Buffer Overflows have been around since the very beginnings of the Von-Neuman 1 architecture. They first gained widespread notoriety in 1988 with the Morris Internet worm. Unfortunately, the same basic attack remains effective today. Of the 17 CERT security advisories of 1999, 10 of them were directly caused by buffer-overflow software bugs. By far the most common type of buffer overflow attack is based on corrupting the stack.

Most modern computer systems use a stack to pass arguments to procedures and to store local variables. A stack is a last in first out (LIFO) buffer in the high memory area of a process image. When a program invokes a function a new "stack frame" is created. This stack frame consists of the arguments passed to the function as well as a dynamic amount of local variable space. The "stack pointer" is a register that holds the current location of the top of the stack. Since this value is constantly changing as new values are pushed onto the top of the stack, many implementations also provide a "frame pointer" that is located near the beginning of a stack frame so that local variables can more easily be addressed relative to this value. 1 The return address for function calls is also stored on the stack, and this is the cause of stack-overflow exploits since overflowing a local variable in a function can overwrite the return address of that function, potentially allowing a malicious user to execute any code he or she wants.

Although stack-based attacks are by far the most common, it would also be possible to overrun the stack with a heap-based (malloc/free) attack.

The C programming language does not perform automatic bounds checking on arrays or pointers as many other languages do. In addition, the standard C library is filled with a handful of very dangerous functions.

`strcpy`(char *dest, const char *src)	May overflow the dest buffer
`strcat`(char *dest, const char *src)	May overflow the dest buffer
`getwd`(char *buf)	May overflow the buf buffer
`gets`(char *s)	May overflow the s buffer
`[vf]scanf`(const char *format, ...)	May overflow its arguments.
`realpath`(char *path, char resolved_path[])	May overflow the path buffer
`[v]sprintf`(char *str, const char *format, ...)	May overflow the str buffer.

6.3.1. Example Buffer Overflow

The following example code contains a buffer overflow designed to overwrite the return address and skip the instruction immediately following the function call. (Inspired by 4)

```
#include <stdio.h>

void manipulate(char *buffer) {
  char newbuffer[80];
  strcpy(newbuffer,buffer);
}

int main() {
  char ch,buffer[4096];
  int i=0;

  while ((buffer[i++] = getchar()) != ’\n’) {};

  i=1;
  manipulate(buffer);
  i=2;
  printf("The value of i is : %d\n",i);
  return 0;
}
```

Let us examine what the memory image of this process would look like if we were to input 160 spaces into our little program before hitting return.

[XXX figure here!]

Obviously more malicious input can be devised to execute actual compiled instructions (such as exec(/bin/sh)).

6.3.2. Avoiding Buffer Overflows

The most straightforward solution to the problem of stack-overflows is to always use length restricted memory and string copy functions. `strncpy` and `strncat` are part of the standard C library. These functions accept a length value as a parameter which should be no larger than the size of the destination buffer. These functions will then copy up to ‘length’ bytes from the source to the destination. However there are a number of problems with these functions. Neither function guarantees NUL termination if the size of the input buffer is as large as the destination. The length parameter is also used inconsistently between strncpy and strncat so it is easy for programmers to get confused as to their proper usage. There is also a significant performance loss compared to `strcpy` when copying a short string into a large buffer since `strncpy` NUL fills up the the size specified.

In OpenBSD, another memory copy implementation has been created to get around these problem. The `strlcpy` and `strlcat` functions guarantee that they will always null terminate the destination string when given a non-zero length argument. For more information about these functions see 6. The OpenBSD `strlcpy` and `strlcat` instructions have been in FreeBSD since 3.3.

6.3.2.1. Compiler based run-time bounds checking

Unfortunately there is still a very large assortment of code in public use which blindly copies memory around without using any of the bounded copy routines we just discussed. Fortunately, there is another solution. Several compiler add-ons and libraries exist to do Run-time bounds checking in C/C++.

StackGuard is one such add-on that is implemented as a small patch to the gcc code generator. From the StackGuard website, http://immunix.org/stackguard.html :

"StackGuard detects and defeats stack smashing attacks by protecting the return address on the stack from being altered. StackGuard places a "canary" word next to the return address when a function is called. If the canary word has been altered when the function returns, then a stack smashing attack has been attempted, and the program responds by emitting an intruder alert into syslog, and then halts."

"StackGuard is implemented as a small patch to the gcc code generator, specifically the function_prolog() and function_epilog() routines. function_prolog() has been enhanced to lay down canaries on the stack when functions start, and function_epilog() checks canary integrity when the function exits. Any attempt at corrupting the return address is thus detected before the function returns."

Recompiling your application with StackGuard is an effective means of stopping most buffer-overflow attacks, but it can still be compromised.

6.3.2.2. Library based run-time bounds checking

Compiler-based mechanisms are completely useless for binary-only software for which you cannot recompile. For these situations there are a number of libraries which re-implement the unsafe functions of the C-library (`strcpy`, `fscanf`, `getwd`, etc..) and ensure that these functions can never write past the stack pointer.

- libsafe
- libverify
- libparnoia

Unfortunately these library-based defenses have a number of shortcomings. These libraries only protect against a very small set of security related issues and they neglect to fix the actual problem. These defenses may fail if the application was compiled with -fomit-frame-pointer. Also, the LD_PRELOAD and LD_LIBRARY_PATH environment variables can be overwritten/unset by the user.

6.4. SetUID issues

There are at least 6 different IDs associated with any given process. Because of this you have to be very careful with the access that your process has at any given time. In particular, all seteuid applications should give up their privileges as soon as it is no longer required.

The real user ID can only be changed by a superuser process. The **login** program sets this when a user initially logs in and it is seldom changed.

The effective user ID is set by the `exec()` functions if a program has its seteuid bit set. An application can call `seteuid()` at any time to set the effective user ID to either the real user ID or the saved set-user-ID. When the effective user ID is set by `exec()` functions, the previous value is saved in the saved set-user-ID.

6.5. Limiting your program’s environment

The traditional method of restricting a process is with the `chroot()` system call. This system call changes the root directory from which all other paths are referenced for a process and any child processes. For this call to succeed the process must have execute (search) permission on the directory being referenced. The new environment does not actually take effect until you `chdir()` into your new environment. It should also be noted that a process can easily break out of a chroot environment if it has root privilege. This could be accomplished by creating device nodes to read kernel memory, attaching a debugger to a process outside of the jail, or in many other creative ways.

The behavior of the `chroot()` system call can be controlled somewhat with the kern.chroot_allow_open_directories `sysctl` variable. When this value is set to 0, `chroot()` will fail with EPERM if there are any directories open. If set to the default value of 1, then `chroot()` will fail with EPERM if there are any directories open and the process is already subject to a `chroot()` call. For any other value, the check for open directories will be bypassed completely.

6.5.1. FreeBSD's jail functionality

The concept of a Jail extends upon the `chroot()` by limiting the powers of the superuser to create a true 'virtual server'. Once a prison is setup all network communication must take place through the specified IP address, and the power of "root privilege" in this jail is severely constrained.

While in a prison, any tests of superuser power within the kernel using the `suser()` call will fail. However, some calls to `suser()` have been changed to a new interface `suser_xxx()`. This function is responsible for recognizing or denying access to superuser power for imprisoned processes.

A superuser process within a jailed environment has the power to :

- Manipulate credential with `setuid`, `seteuid`, `setgid`, `setegid`, `setgroups`, `setreuid`, `setregid`, `setlogin`
- Set resource limits with `setrlimit`
- Modify some sysctl nodes (kern.hostname)
- `chroot()`
- Set flags on a vnode: `chflags`, `fchflags`
- Set attributes of a vnode such as file permission, owner, group, size, access time, and modification time.
- Bind to privileged ports in the Internet domain (ports < 1024)

`Jail` is a very useful tool for running applications in a secure environment but it does have some shortcomings. Currently, the IPC mechanisms have not been converted to the `suser_xxx` so applications such as MySQL can not be run within a jail. Superuser access may have a very limited meaning within a jail, but there is no way to specify exactly what "very limited" means.

6.5.2. POSIX.1e Process Capabilities

Posix has released a working draft that adds event auditing, access control lists, fine grained privileges, information labeling, and mandatory access control.

This is a work in progress and is the focus of the TrustedBSD (http://www.trustedbsd.org) project. Some of the initial work has been committed to FreeBSD-current (cap_set_proc(3)).

6.6. Trust

An application should never assume that anything about the users environment is sane. This includes (but is certainly not limited to) : user input, signals, environment variables, resources, IPC, mmaps, the file system working directory, file descriptors, the # of open files, etc.

You should never assume that you can catch all forms of invalid input that a user might supply. Instead, your application should use positive filtering to only allow a specific subset of inputs that you deem safe. Improper data validation has been the cause of many exploits, especially with CGI scripts on the world wide web. For filenames you need to be extra careful about paths ("../", "/"), symbolic links, and shell escape characters.

Perl has a really cool feature called "Taint" mode which can be used to prevent scripts for using data derived outside the program in an unsafe way. This mode will check command line arguments, environment variables, locale information, the results of certain syscalls (`readdir()`, `readlink()`, `getpwxxx()`, and all file input.

6.7. Race Conditions

A race condition is anomalous behavior caused by the unexpected dependence on the relative timing of events. In other words, a programmer incorrectly assumed that a particular event would always happen before another.

Some of the common causes of race conditions are signals, access checks, and file opens. Signals are asynchronous events by nature so special care must be taken in dealing with them. Checking access with `access(2)` then `open(2)` is clearly non-atomic. Users can move files in between the two calls. Instead, privileged applications should `seteuid()` and then call `open()` directly. Along the same lines, an application should always set a proper umask before `open()` to obviate the need for spurious `chmod()` calls.

III. Kernel

Chapter 7. History of the Unix Kernel

Some history of the Unix/BSD kernel, system calls, how do processes work, blocking, scheduling, threads (kernel), context switching, signals, interrupts, modules, etc.

Chapter 8. Locking Notes

This chapter is maintained by the FreeBSD SMP Next Generation Project <`freebsd-smp@FreeBSD.org`>.

This document outlines the locking used in the FreeBSD kernel to permit effective multi-processing within the kernel. Locking can be achieved via several means. Data structures can be protected by mutexes or lockmgr(9) locks. A few variables are protected simply by always using atomic operations to access them.

8.1. Mutexes

A mutex is simply a lock used to guarantee mutual exclusion. Specifically, a mutex may only be owned by one entity at a time. If another entity wishes to obtain a mutex that is already owned, it must wait until the mutex is released. In the FreeBSD kernel, mutexes are owned by processes.

Mutexes may be recursively acquired, but they are intended to be held for a short period of time. Specifically, one may not sleep while holding a mutex. If you need to hold a lock across a sleep, use a lockmgr(9) lock.

Each mutex has several properties of interest:

Variable Name

The name of the struct mtx variable in the kernel source.

Logical Name

The name of the mutex assigned to it by `mtx_init`. This name is displayed in KTR trace messages and witness errors and warnings and is used to distinguish mutexes in the witness code.

Type

The type of the mutex in terms of the `MTX_*` flags. The meaning for each flag is related to its meaning as documented in mutex(9).

`MTX_DEF`

A sleep mutex

`MTX_SPIN`

A spin mutex

`MTX_COLD`

This mutex is initialized very early. Thus, it must be declared via `MUTEX_DECLARE`, and the `MTX_COLD` flag must be passed to `mtx_init`.

`MTX_TOPHALF`

This spin mutex does not disable interrupts.

`MTX_NORECURSE`

This mutex is not allowed to recurse.

Protectees

A list of data structures or data structure members that this entry protects. For data structure members, the name will be in the form of structure name.*`member name`*.

Dependent Functions

Functions that can only be called if this mutex is held.

Table 8-1. Mutex List

Variable Name	Logical Name	Type	Protectees	Dependent Functions

Variable Name	Logical Name	Type	Protectees	Dependent Functions
sched_lock	“sched lock”	MTX_SPIN | MTX_COLD	_gmonparam, cnt.v_swtch, cp_time, curpriority, mtx.*mtx_blocked*, mtx.*mtx_contested*, proc.*p_contested*, proc.*p_blocked*, proc.*p_flag* (P_PROFIL XXX, P_INMEM, P_SINTR, P_TIMEOUT, P_SWAPINREQ XXX, P_INMEN XXX), proc.*p_nice*, proc.*p_procq*, proc.*p_blocked*, proc.*p_estcpu*, proc.*p_nativepri*, proc.*p_priority*, proc.*p_usrpri*, proc.*p_rtprio*, proc.*p_rqindex*, proc.*p_stats->p_prof*, proc.*p_stats->p_ru*, proc.*p_stat*, proc.*p_cpticks* proc.*p_iticks*, proc.*p_uticks*, proc.*p_sticks*, proc.*p_swtime*, proc.*p_slptime*, proc.*p_runtime*, proc.*p_pctcpu*, proc.*p_oncpu*, proc.*p_asleep*, proc.*p_wchan*, proc.*p_wmesg*, proc.*p_slpq*, proc.*p_vmspace* (XXX - in statclock), pscnt, slpque, itqueuebits, itqueues, rtqueuebits, rtqueues,	setrunqueue, remrunqueue, mi_switch, chooseproc, schedclock, resetpriority, updatepri, maybe_resched, cpu_switch, cpu_throw

Variable Name	**Logical Name**	**Type**	**Protectees**	**Dependent Functions**
vm86pcb_lock	“vm86pcb lock”	`MTX_DEF` | `MTX_COLD`	`vm86pcb`	`vm86_bioscall`
Giant	“Giant”	`MTX_DEF` | `MTX_COLD`	nearly everything	lots
callout_lock	“callout lock”	`MTX_SPIN`	`callfree`, `callwheel`, `nextsoftcheck`, proc.*`p_itcallout`*, proc.*`p_slpcallout`*, `softticks`, `ticks`	

8.2. Lock Manager Locks

Locks that are provided via the lockmgr(9) interface are lock manager locks. These locks are reader-writer locks and may be held by a sleeping process.

Table 8-2. lockmgr(9) Lock List

Variable Name	**Protectees**	
`allproc_lock`	`allproc` `zombproc` `pidhashtbl` proc.*`p_list`* proc.*`p_hash`* `nextpid`	`proctree_lock`

8.3. Atomically Protected Variables

An atomically protected variable is a special variable that is not protected by an explicit lock. Instead, all data accesses to the variables use special atomic operations as described in atomic(9). Very few variables are treated this way, although other synchronization primitives such as mutexes are implemented with atomically protected variables.

- `astpending`
- mtx.*`mtx_lock`*

IV. Memory Management

Chapter 9. Virtual Memory System

9.1. The FreeBSD VM System

Contributed by Matthew Dillon <`dillon@FreeBSD.org`>. 6 Feb 1999

9.1.1. Management of physical memory—`vm_page_t`

Physical memory is managed on a page-by-page basis through the `vm_page_t` structure. Pages of physical memory are categorized through the placement of their respective `vm_page_t` structures on one of several paging queues.

A page can be in a wired, active, inactive, cache, or free state. Except for the wired state, the page is typically placed in a doubly link list queue representing the state that it is in. Wired pages are not placed on any queue.

FreeBSD implements a more involved paging queue for cached and free pages in order to implement page coloring. Each of these states involves multiple queues arranged according to the size of the processor's L1 and L2 caches. When a new page needs to be allocated, FreeBSD attempts to obtain one that is reasonably well aligned from the point of view of the L1 and L2 caches relative to the VM object the page is being allocated for.

Additionally, a page may be held with a reference count or locked with a busy count. The VM system also implements an "ultimate locked" state for a page using the PG_BUSY bit in the page's flags.

In general terms, each of the paging queues operates in a LRU fashion. A page is typically placed in a wired or active state initially. When wired, the page is usually associated with a page table somewhere. The VM system ages the page by scanning pages in a more active paging queue (LRU) in order to move them to a less-active paging queue. Pages that get moved into the cache are still associated with a VM object but are candidates for immediate reuse. Pages in the free queue are truly free. FreeBSD attempts to minimize the number of pages in the free queue, but a certain minimum number of truly free pages must be maintained in order to accommodate page allocation at interrupt time.

If a process attempts to access a page that does not exist in its page table but does exist in one of the paging queues (such as the inactive or cache queues), a relatively inexpensive page reactivation fault occurs which causes the page to be reactivated. If the page does not exist in system memory at all, the process must block while the page is brought in from disk.

FreeBSD dynamically tunes its paging queues and attempts to maintain reasonable ratios of pages in the various queues as well as attempts to maintain a reasonable breakdown of clean v.s. dirty pages. The amount of rebalancing that occurs depends on the system's memory load. This rebalancing is implemented by the pageout daemon and involves laundering dirty pages (syncing them with their backing store), noticing when pages are activity referenced (resetting their position in the LRU queues or moving them between queues), migrating pages between queues when the queues are out of balance, and so forth. FreeBSD's VM system is willing to take a reasonable number of reactivation page faults to determine how active or how idle a page actually is. This leads to better decisions being made as to when to launder or swap-out a page.

9.1.2. The unified buffer cache—`vm_object_t`

FreeBSD implements the idea of a generic “VM object”. VM objects can be associated with backing store of various types—unbacked, swap-backed, physical device-backed, or file-backed storage. Since the filesystem uses the same VM objects to manage in-core data relating to files, the result is a unified buffer cache.

VM objects can be *shadowed*. That is, they can be stacked on top of each other. For example, you might have a swap-backed VM object stacked on top of a file-backed VM object in order to implement a MAP_PRIVATE mmap()ing. This stacking is also used to implement various sharing properties, including, copy-on-write, for forked address spaces.

It should be noted that a `vm_page_t` can only be associated with one VM object at a time. The VM object shadowing implements the perceived sharing of the same page across multiple instances.

9.1.3. Filesystem I/O—`struct buf`

vnode-backed VM objects, such as file-backed objects, generally need to maintain their own clean/dirty info independent from the VM system’s idea of clean/dirty. For example, when the VM system decides to synchronize a physical page to its backing store, the VM system needs to mark the page clean before the page is actually written to its backing s tore. Additionally, filesystems need to be able to map portions of a file or file metadata into KVM in order to operate on it.

The entities used to manage this are known as filesystem buffers, `struct buf`’s, and also known as `bp`’s. When a filesystem needs to operate on a portion of a VM object, it typically maps part of the object into a struct buf and the maps the pages in the struct buf into KVM. In the same manner, disk I/O is typically issued by mapping portions of objects into buffer structures and then issuing the I/O on the buffer structures. The underlying vm_page_t’s are typically busied for the duration of the I/O. Filesystem buffers also have their own notion of being busy, which is useful to filesystem driver code which would rather operate on filesystem buffers instead of hard VM pages.

FreeBSD reserves a limited amount of KVM to hold mappings from struct bufs, but it should be made clear that this KVM is used solely to hold mappings and does not limit the ability to cache data. Physical data caching is strictly a function of `vm_page_t`’s, not filesystem buffers. However, since filesystem buffers are used placehold I/O, they do inherently limit the amount of concurrent I/O possible. As there are usually a few thousand filesystem buffers available, this is not usually a problem.

9.1.4. Mapping Page Tables - vm_map_t, vm_entry_t

FreeBSD separates the physical page table topology from the VM system. All hard per-process page tables can be reconstructed on the fly and are usually considered throwaway. Special page tables such as those managing KVM are typically permanently preallocated. These page tables are not throwaway.

FreeBSD associates portions of vm_objects with address ranges in virtual memory through `vm_map_t` and `vm_entry_t` structures. Page tables are directly synthesized from the `vm_map_t`/`vm_entry_t`/ `vm_object_t` hierarchy. Remember when I mentioned that physical pages are only directly associated with a `vm_object`. Well, that isn't quite true. `vm_page_t`'s are also linked into page tables that they are actively associated with. One `vm_page_t` can be linked into several *pmaps*, as page tables are called. However, the hierarchical association holds so all references to the same page in the same object reference the same `vm_page_t` and thus give us buffer cache unification across the board.

9.1.5. KVM Memory Mapping

FreeBSD uses KVM to hold various kernel structures. The single largest entity held in KVM is the filesystem buffer cache. That is, mappings relating to `struct buf` entities.

Unlike Linux, FreeBSD does NOT map all of physical memory into KVM. This means that FreeBSD can handle memory configurations up to 4G on 32 bit platforms. In fact, if the mmu were capable of it, FreeBSD could theoretically handle memory configurations up to 8TB on a 32 bit platform. However, since most 32 bit platforms are only capable of mapping 4GB of ram, this is a moot point.

KVM is managed through several mechanisms. The main mechanism used to manage KVM is the *zone allocator*. The zone allocator takes a chunk of KVM and splits it up into constant-sized blocks of memory in order to allocate a specific type of structure. You can use `vmstat -m` to get an overview of current KVM utilization broken down by zone.

9.1.6. Tuning the FreeBSD VM system

A concerted effort has been made to make the FreeBSD kernel dynamically tune itself. Typically you do not need to mess with anything beyond the `maxusers` and `NMBCLUSTERS` kernel config options. That is, kernel compilation options specified in (typically) `/usr/src/sys/i386/conf/CONFIG_FILE`. A description of all available kernel configuration options can be found in `/usr/src/sys/i386/conf/LINT`.

In a large system configuration you may wish to increase `maxusers`. Values typically range from 10 to 128. Note that raising `maxusers` too high can cause the system to overflow available KVM resulting in unpredictable operation. It is better to leave maxusers at some reasonable number and add other options, such as `NMBCLUSTERS`, to increase specific resources.

If your system is going to use the network heavily, you may want to increase `NMBCLUSTERS`. Typical values range from 1024 to 4096.

The `NBUF` parameter is also traditionally used to scale the system. This parameter determines the amount of KVA the system can use to map filesystem buffers for I/O. Note that this parameter has nothing whatsoever to do with the unified buffer cache! This parameter is dynamically tuned in 3.0-CURRENT and later kernels and should generally not be adjusted manually. We recommend that you *not* try to specify an `NBUF` parameter. Let the system pick it. Too small a value can result in extremely inefficient filesystem operation while too large a value can starve the page queues by causing too many pages to become wired down.

By default, FreeBSD kernels are not optimized. You can set debugging and optimization flags with the `makeoptions` directive in the kernel configuration. Note that you should not use `-g` unless you can accommodate the large (typically 7 MB+) kernels that result.

```
makeoptions      DEBUG="-g"
makeoptions      COPTFLAGS="-O -pipe"
```

Sysctl provides a way to tune kernel parameters at run-time. You typically do not need to mess with any of the sysctl variables, especially the VM related ones.

Run time VM and system tuning is relatively straightforward. First, use softupdates on your UFS/FFS filesystems whenever possible. `/usr/src/contrib/sys/softupdates/README` contains instructions (and restrictions) on how to configure it up.

Second, configure sufficient swap. You should have a swap partition configured on each physical disk, up to four, even on your “work” disks. You should have at least 2x the swap space as you have main memory, and possibly even more if you do not have a lot of memory. You should also size your swap partition based on the maximum memory configuration you ever intend to put on the machine so you do not have to repartition your disks later on. If you want to be able to accommodate a crash dump, your first swap partition must be at least as large as main memory and `/var/crash` must have sufficient free space to hold the dump.

NFS-based swap is perfectly acceptable on -4.x or later systems, but you must be aware that the NFS server will take the brunt of the paging load.

Chapter 10. DMA

10.1. DMA: What it is and How it Works

Copyright © 1995,1997 Frank Durda IV <`uhclem@FreeBSD.org`>, All Rights Reserved. 10 December 1996. Last Update 8 October 1997.

Direct Memory Access (DMA) is a method of allowing data to be moved from one location to another in a computer without intervention from the central processor (CPU).

The way that the DMA function is implemented varies between computer architectures, so this discussion will limit itself to the implementation and workings of the DMA subsystem on the IBM Personal Computer (PC), the IBM PC/AT and all of its successors and clones.

The PC DMA subsystem is based on the Intel 8237 DMA controller. The 8237 contains four DMA channels that can be programmed independently and any one of the channels may be active at any moment. These channels are numbered 0, 1, 2 and 3. Starting with the PC/AT, IBM added a second 8237 chip, and numbered those channels 4, 5, 6 and 7.

The original DMA controller (0, 1, 2 and 3) moves one byte in each transfer. The second DMA controller (4, 5, 6, and 7) moves 16-bits from two adjacent memory locations in each transfer, with the first byte always coming from an even-numbered address. The two controllers are identical components and the difference in transfer size is caused by the way the second controller is wired into the system.

The 8237 has two electrical signals for each channel, named DRQ and -DACK. There are additional signals with the names HRQ (Hold Request), HLDA (Hold Acknowledge), -EOP (End of Process), and the bus control signals -MEMR (Memory Read), -MEMW (Memory Write), -IOR (I/O Read), and -IOW (I/O Write).

The 8237 DMA is known as a “fly-by” DMA controller. This means that the data being moved from one location to another does not pass through the DMA chip and is not stored in the DMA chip. Subsequently, the DMA can only transfer data between an I/O port and a memory address, but not between two I/O ports or two memory locations.

Note: The 8237 does allow two channels to be connected together to allow memory-to-memory DMA operations in a non-“fly-by” mode, but nobody in the PC industry uses this scarce resource this way since it is faster to move data between memory locations using the CPU.

In the PC architecture, each DMA channel is normally activated only when the hardware that uses a given DMA channel requests a transfer by asserting the DRQ line for that channel.

10.1.1. A Sample DMA transfer

Here is an example of the steps that occur to cause and perform a DMA transfer. In this example, the floppy disk controller (FDC) has just read a byte from a diskette and wants the DMA to place it in memory at location 0x00123456. The process begins by the FDC asserting the DRQ2 signal (the DRQ line for DMA channel 2) to alert the DMA controller.

The DMA controller will note that the DRQ2 signal is asserted. The DMA controller will then make sure that DMA channel 2 has been programmed and is unmasked (enabled). The DMA controller also makes sure that none of the other DMA channels are active or want to be active and have a higher priority. Once these checks are complete, the DMA asks the CPU to release the bus so that the DMA may use the bus. The DMA requests the bus by asserting the HRQ signal which goes to the CPU.

The CPU detects the HRQ signal, and will complete executing the current instruction. Once the processor has reached a state where it can release the bus, it will. Now all of the signals normally generated by the CPU (-MEMR, -MEMW, -IOR, -IOW and a few others) are placed in a tri-stated condition (neither high or low) and then the CPU asserts the HLDA signal which tells the DMA controller that it is now in charge of the bus.

Depending on the processor, the CPU may be able to execute a few additional instructions now that it no longer has the bus, but the CPU will eventually have to wait when it reaches an instruction that must read something from memory that is not in the internal processor cache or pipeline.

Now that the DMA “is in charge”, the DMA activates its -MEMR, -MEMW, -IOR, -IOW output signals, and the address outputs from the DMA are set to 0x3456, which will be used to direct the byte that is about to transferred to a specific memory location.

The DMA will then let the device that requested the DMA transfer know that the transfer is commencing. This is done by asserting the -DACK signal, or in the case of the floppy disk controller, -DACK2 is asserted.

The floppy disk controller is now responsible for placing the byte to be transferred on the bus Data lines. Unless the floppy controller needs more time to get the data byte on the bus (and if the peripheral does need more time it alerts the DMA via the READY signal), the DMA will wait one DMA clock, and then de-assert the -MEMW and -IOR signals so that the memory will latch and store the byte that was on the bus, and the FDC will know that the byte has been transferred.

Since the DMA cycle only transfers a single byte at a time, the FDC now drops the DRQ2 signal, so the DMA knows that it is no longer needed. The DMA will de-assert the -DACK2 signal, so that the FDC knows it must stop placing data on the bus.

The DMA will now check to see if any of the other DMA channels have any work to do. If none of the channels have their DRQ lines asserted, the DMA controller has completed its work and will now tri-state the -MEMR, -MEMW, -IOR, -IOW and address signals.

Finally, the DMA will de-assert the HRQ signal. The CPU sees this, and de-asserts the HOLDA signal. Now the CPU activates its -MEMR, -MEMW, -IOR, -IOW and address lines, and it resumes executing instructions and accessing main memory and the peripherals.

For a typical floppy disk sector, the above process is repeated 512 times, once for each byte. Each time a byte is transferred, the address register in the DMA is incremented and the counter in the DMA that shows how many bytes are to be transferred is decremented.

When the counter reaches zero, the DMA asserts the EOP signal, which indicates that the counter has reached zero and no more data will be transferred until the DMA controller is reprogrammed by the CPU. This event is also called the Terminal Count (TC). There is only one EOP signal, and since only DMA channel can be active at any instant, the DMA channel that is currently active must be the DMA channel that just completed its task.

If a peripheral wants to generate an interrupt when the transfer of a buffer is complete, it can test for its -DACKn signal and the EOP signal both being asserted at the same time. When that happens, it means the DMA will not transfer any more information for that peripheral without intervention by the CPU. The peripheral can then assert one of the interrupt signals to get the processors' attention. In the PC architecture, the DMA chip itself is not capable of generating an interrupt. The peripheral and its associated hardware is responsible for generating any interrupt that occurs. Subsequently, it is possible to have a peripheral that uses DMA but does not use interrupts.

It is important to understand that although the CPU always releases the bus to the DMA when the DMA makes the request, this action is invisible to both applications and the operating systems, except for slight changes in the amount of time the processor takes to execute instructions when the DMA is active. Subsequently, the processor must poll the peripheral, poll the registers in the DMA chip, or receive an interrupt from the peripheral to know for certain when a DMA transfer has completed.

10.1.2. DMA Page Registers and 16Meg address space limitations

You may have noticed earlier that instead of the DMA setting the address lines to 0x00123456 as we said earlier, the DMA only set 0x3456. The reason for this takes a bit of explaining.

When the original IBM PC was designed, IBM elected to use both DMA and interrupt controller chips that were designed for use with the 8085, an 8-bit processor with an address space of 16 bits (64K). Since the IBM PC supported more than 64K of memory, something had to be done to allow the DMA to read or write memory locations above the 64K mark. What IBM did to solve this problem was to add an external data latch for each DMA channel that holds the upper bits of the address to be read to or written from. Whenever a DMA channel is active, the contents of that latch are written to the address bus and kept there until the DMA operation for the channel ends. IBM called these latches "Page Registers".

So for our example above, the DMA would put the 0x3456 part of the address on the bus, and the Page Register for DMA channel 2 would put 0x0012xxxx on the bus. Together, these two values form the complete address in memory that is to be accessed.

Because the Page Register latch is independent of the DMA chip, the area of memory to be read or written must not span a 64K physical boundary. For example, if the DMA accesses memory location 0xffff, after that transfer the DMA will then increment the address register and the DMA will access the next byte at location 0x0000, not 0x10000. The results of letting this happen are probably not intended.

> **Note:** “Physical” 64K boundaries should not be confused with 8086-mode 64K “Segments”, which are created by mathematically adding a segment register with an offset register. Page Registers have no address overlap and are mathematically OR-ed together.

To further complicate matters, the external DMA address latches on the PC/AT hold only eight bits, so that gives us 8+16=24 bits, which means that the DMA can only point at memory locations between 0 and 16Meg. For newer computers that allow more than 16Meg of memory, the standard PC-compatible DMA cannot access memory locations above 16Meg.

To get around this restriction, operating systems will reserve a RAM buffer in an area below 16Meg that also does not span a physical 64K boundary. Then the DMA will be programmed to transfer data from the peripheral and into that buffer. Once the DMA has moved the data into this buffer, the operating system will then copy the data from the buffer to the address where the data is really supposed to be stored.

When writing data from an address above 16Meg to a DMA-based peripheral, the data must be first copied from where it resides into a buffer located below 16Meg, and then the DMA can copy the data from the buffer to the hardware. In FreeBSD, these reserved buffers are called “Bounce Buffers”. In the MS-DOS world, they are sometimes called “Smart Buffers”.

> **Note:** A new implementation of the 8237, called the 82374, allows 16 bits of page register to be specified, allows access to the entire 32 bit address space, without the use of bounce buffers.

10.1.3. DMA Operational Modes and Settings

The 8237 DMA can be operated in several modes. The main ones are:

Single

A single byte (or word) is transferred. The DMA must release and re-acquire the bus for each additional byte. This is commonly-used by devices that cannot transfer the entire block of data immediately. The peripheral will request the DMA each time it is ready for another transfer.

The standard PC-compatible floppy disk controller (NEC 765) only has a one-byte buffer, so it uses this mode.

Block/Demand

Once the DMA acquires the system bus, an entire block of data is transferred, up to a maximum of 64K. If the peripheral needs additional time, it can assert the READY signal to suspend the transfer briefly. READY should not be used excessively, and for slow peripheral transfers, the Single Transfer Mode should be used instead.

The difference between Block and Demand is that once a Block transfer is started, it runs until the transfer count reaches zero. DRQ only needs to be asserted until -DACK is asserted. Demand Mode will transfer one more bytes until DRQ is de-asserted, at which point the DMA suspends the transfer and releases the bus back to the CPU. When DRQ is asserted later, the transfer resumes where it was suspended.

Older hard disk controllers used Demand Mode until CPU speeds increased to the point that it was more efficient to transfer the data using the CPU, particularly if the memory locations used in the transfer were above the 16Meg mark.

Cascade

This mechanism allows a DMA channel to request the bus, but then the attached peripheral device is responsible for placing the addressing information on the bus instead of the DMA. This is also used to implement a technique known as “Bus Mastering”.

When a DMA channel in Cascade Mode receives control of the bus, the DMA does not place addresses and I/O control signals on the bus like the DMA normally does when it is active. Instead, the DMA only asserts the -DACK signal for the active DMA channel.

At this point it is up to the peripheral connected to that DMA channel to provide address and bus control signals. The peripheral has complete control over the system bus, and can do reads and/or writes to any address below 16Meg. When the peripheral is finished with the bus, it de-asserts the DRQ line, and the DMA controller can then return control to the CPU or to some other DMA channel.

Cascade Mode can be used to chain multiple DMA controllers together, and this is exactly what DMA Channel 4 is used for in the PC architecture. When a peripheral requests the bus on DMA channels 0, 1, 2 or 3, the slave DMA controller asserts HLDREQ, but this wire is actually connected to DRQ4 on the primary DMA controller instead of to the CPU. The primary DMA controller, thinking it has work to do on Channel 4, requests the bus from the CPU using HLDREQ signal. Once the CPU grants the bus to the primary DMA controller, -DACK4 is asserted, and that wire is actually connected to the HLDA signal on the slave DMA controller. The slave DMA controller then transfers data for the DMA channel that requested it (0, 1, 2 or 3), or the slave DMA may grant the bus to a peripheral that wants to perform its own bus-mastering, such as a SCSI controller.

Because of this wiring arrangement, only DMA channels 0, 1, 2, 3, 5, 6 and 7 are usable with peripherals on PC/AT systems.

Note: DMA channel 0 was reserved for refresh operations in early IBM PC computers, but is generally available for use by peripherals in modern systems.

When a peripheral is performing Bus Mastering, it is important that the peripheral transmit data to or from memory constantly while it holds the system bus. If the peripheral cannot do this, it must release the bus frequently so that the system can perform refresh operations on main memory.

The Dynamic RAM used in all PCs for main memory must be accessed frequently to keep the bits stored in the components “charged”. Dynamic RAM essentially consists of millions of capacitors with each one holding one bit of data. These capacitors are charged with power to represent a `1` or drained to represent a `0`. Because all capacitors leak, power must be added at regular intervals to keep the `1` values intact. The RAM chips actually handle the task of pumping power back into all of the appropriate locations in RAM, but they must be told when to do it by the rest of the computer so that the refresh activity won’t interfere with the computer wanting to access RAM normally. If the computer is unable to refresh memory, the contents of memory will become corrupted in just a few milliseconds.

Since memory read and write cycles “count” as refresh cycles (a dynamic RAM refresh cycle is actually an incomplete memory read cycle), as long as the peripheral controller continues reading or writing data to sequential memory locations, that action will refresh all of memory.

Bus-mastering is found in some SCSI host interfaces and other high-performance peripheral controllers.

Autoinitialize

This mode causes the DMA to perform Byte, Block or Demand transfers, but when the DMA transfer counter reaches zero, the counter and address are set back to where they were when the DMA channel was originally programmed. This means that as long as the peripheral requests transfers, they will be granted. It is up to the CPU to move new data into the fixed buffer ahead of where the DMA is about to transfer it when doing output operations, and read new data out of the buffer behind where the DMA is writing when doing input operations.

This technique is frequently used on audio devices that have small or no hardware “sample” buffers. There is additional CPU overhead to manage this “circular” buffer, but in some cases this may be the only way to eliminate the latency that occurs when the DMA counter reaches zero and the DMA stops transfers until it is reprogrammed.

10.1.4. Programming the DMA

The DMA channel that is to be programmed should always be “masked” before loading any settings. This is because the hardware might unexpectedly assert the DRQ for that channel, and the DMA might respond, even though not all of the parameters have been loaded or updated.

Once masked, the host must specify the direction of the transfer (memory-to-I/O or I/O-to-memory), what mode of DMA operation is to be used for the transfer (Single, Block, Demand, Cascade, etc), and finally the address and length of the transfer are loaded. The length that is loaded is one less than the amount you expect the DMA to transfer. The LSB and MSB of the address and length are written to the same 8-bit I/O port, so another port must be written to first to guarantee that the DMA accepts the first byte as the LSB and the second byte as the MSB of the length and address.

Then, be sure to update the Page Register, which is external to the DMA and is accessed through a different set of I/O ports.

Once all the settings are ready, the DMA channel can be un-masked. That DMA channel is now considered to be “armed”, and will respond when the DRQ line for that channel is asserted.

Refer to a hardware data book for precise programming details for the 8237. You will also need to refer to the I/O port map for the PC system, which describes where the DMA and Page Register ports are located. A complete port map table is located below.

10.1.5. DMA Port Map

All systems based on the IBM-PC and PC/AT have the DMA hardware located at the same I/O ports. The complete list is provided below. Ports assigned to DMA Controller #2 are undefined on non-AT designs.

10.1.5.1. 0x00–0x1f DMA Controller #1 (Channels 0, 1, 2 and 3)

DMA Address and Count Registers

0x00	write	Channel 0 starting address
0x00	read	Channel 0 current address
0x01	write	Channel 0 starting word count
0x01	read	Channel 0 remaining word count
0x02	write	Channel 1 starting address
0x02	read	Channel 1 current address
0x03	write	Channel 1 starting word count
0x03	read	Channel 1 remaining word count
0x04	write	Channel 2 starting address
0x04	read	Channel 2 current address
0x05	write	Channel 2 starting word count
0x05	read	Channel 2 remaining word count
0x06	write	Channel 3 starting address
0x06	read	Channel 3 current address
0x07	write	Channel 3 starting word count
0x07	read	Channel 3 remaining word count

DMA Command Registers

0x08	write	Command Register
0x08	read	Status Register
0x09	write	Request Register
0x09	read	-
0x0a	write	Single Mask Register Bit
0x0a	read	-
0x0b	write	Mode Register
0x0b	read	-
0x0c	write	Clear LSB/MSB Flip-Flop
0x0c	read	-
0x0d	write	Master Clear/Reset
0x0d	read	Temporary Register (not available on newer versions)
0x0e	write	Clear Mask Register
0x0e	read	-
0x0f	write	Write All Mask Register Bits
0x0f	read	Read All Mask Register Bits (only in Intel 82374)

10.1.5.2. 0xc0–0xdf DMA Controller #2 (Channels 4, 5, 6 and 7)

DMA Address and Count Registers

0xc0	write	Channel 4 starting address
0xc0	read	Channel 4 current address
0xc2	write	Channel 4 starting word count
0xc2	read	Channel 4 remaining word count
0xc4	write	Channel 5 starting address
0xc4	read	Channel 5 current address
0xc6	write	Channel 5 starting word count
0xc6	read	Channel 5 remaining word count
0xc8	write	Channel 6 starting address
0xc8	read	Channel 6 current address

0xca	write	Channel 6 starting word count
0xca	read	Channel 6 remaining word count
0xcc	write	Channel 7 starting address
0xcc	read	Channel 7 current address
0xce	write	Channel 7 starting word count
0xce	read	Channel 7 remaining word count

DMA Command Registers

0xd0	write	Command Register
0xd0	read	Status Register
0xd2	write	Request Register
0xd2	read	-
0xd4	write	Single Mask Register Bit
0xd4	read	-
0xd6	write	Mode Register
0xd6	read	-
0xd8	write	Clear LSB/MSB Flip-Flop
0xd8	read	-
0xda	write	Master Clear/Reset
0xda	read	Temporary Register (not present in Intel 82374)
0xdc	write	Clear Mask Register
0xdc	read	-
0xde	write	Write All Mask Register Bits
0xdf	read	Read All Mask Register Bits (only in Intel 82374)

10.1.5.3. 0x80–0x9f DMA Page Registers

0x87	r/w	Channel 0 Low byte (23-16) page Register
0x83	r/w	Channel 1 Low byte (23-16) page Register

0x81	r/w	Channel 2 Low byte (23-16) page Register
0x82	r/w	Channel 3 Low byte (23-16) page Register
0x8b	r/w	Channel 5 Low byte (23-16) page Register
0x89	r/w	Channel 6 Low byte (23-16) page Register
0x8a	r/w	Channel 7 Low byte (23-16) page Register
0x8f	r/w	Low byte page Refresh

10.1.5.4. 0x400–0x4ff 82374 Enhanced DMA Registers

The Intel 82374 EISA System Component (ESC) was introduced in early 1996 and includes a DMA controller that provides a superset of 8237 functionality as well as other PC-compatible core peripheral components in a single package. This chip is targeted at both EISA and PCI platforms, and provides modern DMA features like scatter-gather, ring buffers as well as direct access by the system DMA to all 32 bits of address space.

If these features are used, code should also be included to provide similar functionality in the previous 16 years worth of PC-compatible computers. For compatibility reasons, some of the 82374 registers must be programmed *after* programming the traditional 8237 registers for each transfer. Writing to a traditional 8237 register forces the contents of some of the 82374 enhanced registers to zero to provide backward software compatibility.

0x401	r/w	Channel 0 High byte (bits 23-16) word count
0x403	r/w	Channel 1 High byte (bits 23-16) word count
0x405	r/w	Channel 2 High byte (bits 23-16) word count
0x407	r/w	Channel 3 High byte (bits 23-16) word count
0x4c6	r/w	Channel 5 High byte (bits 23-16) word count
0x4ca	r/w	Channel 6 High byte (bits 23-16) word count

0x4ce	r/w	Channel 7 High byte (bits 23-16) word count
0x487	r/w	Channel 0 High byte (bits 31-24) page Register
0x483	r/w	Channel 1 High byte (bits 31-24) page Register
0x481	r/w	Channel 2 High byte (bits 31-24) page Register
0x482	r/w	Channel 3 High byte (bits 31-24) page Register
0x48b	r/w	Channel 5 High byte (bits 31-24) page Register
0x489	r/w	Channel 6 High byte (bits 31-24) page Register
0x48a	r/w	Channel 6 High byte (bits 31-24) page Register
0x48f	r/w	High byte page Refresh
0x4e0	r/w	Channel 0 Stop Register (bits 7-2)
0x4e1	r/w	Channel 0 Stop Register (bits 15-8)
0x4e2	r/w	Channel 0 Stop Register (bits 23-16)
0x4e4	r/w	Channel 1 Stop Register (bits 7-2)
0x4e5	r/w	Channel 1 Stop Register (bits 15-8)
0x4e6	r/w	Channel 1 Stop Register (bits 23-16)
0x4e8	r/w	Channel 2 Stop Register (bits 7-2)
0x4e9	r/w	Channel 2 Stop Register (bits 15-8)
0x4ea	r/w	Channel 2 Stop Register (bits 23-16)
0x4ec	r/w	Channel 3 Stop Register (bits 7-2)
0x4ed	r/w	Channel 3 Stop Register (bits 15-8)
0x4ee	r/w	Channel 3 Stop Register (bits 23-16)
0x4f4	r/w	Channel 5 Stop Register (bits 7-2)
0x4f5	r/w	Channel 5 Stop Register (bits 15-8)
0x4f6	r/w	Channel 5 Stop Register (bits 23-16)
0x4f8	r/w	Channel 6 Stop Register (bits 7-2)
0x4f9	r/w	Channel 6 Stop Register (bits 15-8)
0x4fa	r/w	Channel 6 Stop Register (bits 23-16)

0x4fc	r/w	Channel 7 Stop Register (bits 7-2)
0x4fd	r/w	Channel 7 Stop Register (bits 15-8)
0x4fe	r/w	Channel 7 Stop Register (bits 23-16)
0x40a	write	Channels 0-3 Chaining Mode Register
0x40a	read	Channel Interrupt Status Register
0x4d4	write	Channels 4-7 Chaining Mode Register
0x4d4	read	Chaining Mode Status
0x40c	read	Chain Buffer Expiration Control Register
0x410	write	Channel 0 Scatter-Gather Command Register
0x411	write	Channel 1 Scatter-Gather Command Register
0x412	write	Channel 2 Scatter-Gather Command Register
0x413	write	Channel 3 Scatter-Gather Command Register
0x415	write	Channel 5 Scatter-Gather Command Register
0x416	write	Channel 6 Scatter-Gather Command Register
0x417	write	Channel 7 Scatter-Gather Command Register
0x418	read	Channel 0 Scatter-Gather Status Register
0x419	read	Channel 1 Scatter-Gather Status Register
0x41a	read	Channel 2 Scatter-Gather Status Register
0x41b	read	Channel 3 Scatter-Gather Status Register
0x41d	read	Channel 5 Scatter-Gather Status Register
0x41e	read	Channel 5 Scatter-Gather Status Register

0x41f	read	Channel 7 Scatter-Gather Status Register
0x420-0x423	r/w	Channel 0 Scatter-Gather Descriptor Table Pointer Register
0x424-0x427	r/w	Channel 1 Scatter-Gather Descriptor Table Pointer Register
0x428-0x42b	r/w	Channel 2 Scatter-Gather Descriptor Table Pointer Register
0x42c-0x42f	r/w	Channel 3 Scatter-Gather Descriptor Table Pointer Register
0x434-0x437	r/w	Channel 5 Scatter-Gather Descriptor Table Pointer Register
0x438-0x43b	r/w	Channel 6 Scatter-Gather Descriptor Table Pointer Register
0x43c-0x43f	r/w	Channel 7 Scatter-Gather Descriptor Table Pointer Register

V. I/O System

Chapter 11. UFS

UFS, FFS, Ext2FS, JFS, inodes, buffer cache, labeling, locking, metadata, soft-updates, LFS, portalfs, procfs, vnodes, memory sharing, memory objects, TLBs, caching

VI. Interprocess Communication

Chapter 12. Signals

Signals, pipes, semaphores, message queues, shared memory, ports, sockets, doors

VII. Networking

Chapter 13. Sockets

Sockets, bpf, IP, TCP, UDP, ICMP, OSI, bridging, firewalling, NAT, switching, etc

Chapter 14. IPv6 Internals

14.1. IPv6/IPsec Implementation

Contributed by Yoshinobu Inoue <`shin@FreeBSD.org`>, 5 March 2000.

This section should explain IPv6 and IPsec related implementation internals. These functionalities are derived from KAME project (http://www.kame.net)

14.1.1. IPv6

14.1.1.1. Conformance

The IPv6 related functions conforms, or tries to conform to the latest set of IPv6 specifications. For future reference we list some of the relevant documents below (*NOTE*: this is not a complete list - this is too hard to maintain...).

For details please refer to specific chapter in the document, RFCs, manpages, or comments in the source code.

Conformance tests have been performed on the KAME STABLE kit at TAHI project. Results can be viewed at http://www.tahi.org/report/KAME/ (http://www.tahi.org/report/KAME/). We also attended Univ. of New Hampshire IOL tests (http://www.iol.unh.edu/) in the past, with our past snapshots.

- RFC1639: FTP Operation Over Big Address Records (FOOBAR)
  - RFC2428 is preferred over RFC1639. FTP clients will first try RFC2428, then RFC1639 if failed.
- RFC1886: DNS Extensions to support IPv6
- RFC1933: Transition Mechanisms for IPv6 Hosts and Routers
  - IPv4 compatible address is not supported.
  - automatic tunneling (described in 4.3 of this RFC) is not supported.
  - gif(4) interface implements IPv[46]-over-IPv[46] tunnel in a generic way, and it covers "configured tunnel" described in the spec. See 23.5.1.5 in this document for details.
- RFC1981: Path MTU Discovery for IPv6
- RFC2080: RIPng for IPv6
  - usr.sbin/route6d support this.

- RFC2292: Advanced Sockets API for IPv6
  - For supported library functions/kernel APIs, see `sys/netinet6/ADVAPI`.

- RFC2362: Protocol Independent Multicast-Sparse Mode (PIM-SM)
  - RFC2362 defines packet formats for PIM-SM. `draft-ietf-pim-ipv6-01.txt` is written based on this.

- RFC2373: IPv6 Addressing Architecture
  - supports node required addresses, and conforms to the scope requirement.

- RFC2374: An IPv6 Aggregatable Global Unicast Address Format
  - supports 64-bit length of Interface ID.

- RFC2375: IPv6 Multicast Address Assignments
  - Userland applications use the well-known addresses assigned in the RFC.

- RFC2428: FTP Extensions for IPv6 and NATs
  - RFC2428 is preferred over RFC1639. FTP clients will first try RFC2428, then RFC1639 if failed.

- RFC2460: IPv6 specification
- RFC2461: Neighbor discovery for IPv6
  - See 23.5.1.2 in this document for details.

- RFC2462: IPv6 Stateless Address Autoconfiguration
  - See 23.5.1.4 in this document for details.

- RFC2463: ICMPv6 for IPv6 specification
  - See 23.5.1.9 in this document for details.

- RFC2464: Transmission of IPv6 Packets over Ethernet Networks
- RFC2465: MIB for IPv6: Textual Conventions and General Group
  - Necessary statistics are gathered by the kernel. Actual IPv6 MIB support is provided as a patchkit for ucd-snmp.
- RFC2466: MIB for IPv6: ICMPv6 group
  - Necessary statistics are gathered by the kernel. Actual IPv6 MIB support is provided as patchkit for ucd-snmp.
- RFC2467: Transmission of IPv6 Packets over FDDI Networks
- RFC2497: Transmission of IPv6 packet over ARCnet Networks
- RFC2553: Basic Socket Interface Extensions for IPv6
  - IPv4 mapped address (3.7) and special behavior of IPv6 wildcard bind socket (3.8) are supported. See 23.5.1.12 in this document for details.
- RFC2675: IPv6 Jumbograms
  - See 23.5.1.7 in this document for details.
- RFC2710: Multicast Listener Discovery for IPv6
- RFC2711: IPv6 router alert option
- `draft-ietf-ipngwg-router-renum-08`: Router renumbering for IPv6
- `draft-ietf-ipngwg-icmp-namelookups-02`: IPv6 Name Lookups Through ICMP
- `draft-ietf-ipngwg-icmp-name-lookups-03`: IPv6 Name Lookups Through ICMP
- `draft-ietf-pim-ipv6-01.txt`: PIM for IPv6
  - pim6dd(8) implements dense mode. pim6sd(8) implements sparse mode.
- `draft-itojun-ipv6-tcp-to-anycast-00`: Disconnecting TCP connection toward IPv6 anycast address
- `draft-yamamoto-wideipv6-comm-model-00`
  - See 23.5.1.6 in this document for details.
- `draft-ietf-ipngwg-scopedaddr-format-00.txt` : An Extension of Format for IPv6 Scoped Addresses

14.1.1.2. Neighbor Discovery

Neighbor Discovery is fairly stable. Currently Address Resolution, Duplicated Address Detection, and Neighbor Unreachability Detection are supported. In the near future we will be adding Proxy Neighbor Advertisement support in the kernel and Unsolicited Neighbor Advertisement transmission command as admin tool.

If DAD fails, the address will be marked "duplicated" and message will be generated to syslog (and usually to console). The "duplicated" mark can be checked with ifconfig(8). It is administrators' responsibility to check for and recover from DAD failures. The behavior should be improved in the near future.

Some of the network driver loops multicast packets back to itself, even if instructed not to do so (especially in promiscuous mode). In such cases DAD may fail, because DAD engine sees inbound NS packet (actually from the node itself) and considers it as a sign of duplicate. You may want to look at #if condition marked "heuristics" in sys/netinet6/nd6_nbr.c:nd6_dad_timer() as workaround (note that the code fragment in "heuristics" section is not spec conformant).

Neighbor Discovery specification (RFC2461) does not talk about neighbor cache handling in the following cases:

1. when there was no neighbor cache entry, node received unsolicited RS/NS/NA/redirect packet without link-layer address
2. neighbor cache handling on medium without link-layer address (we need a neighbor cache entry for IsRouter bit)

For first case, we implemented workaround based on discussions on IETF ipngwg mailing list. For more details, see the comments in the source code and email thread started from (IPng 7155), dated Feb 6 1999.

IPv6 on-link determination rule (RFC2461) is quite different from assumptions in BSD network code. At this moment, no on-link determination rule is supported where default router list is empty (RFC2461, section 5.2, last sentence in 2nd paragraph - note that the spec misuse the word "host" and "node" in several places in the section).

To avoid possible DoS attacks and infinite loops, only 10 options on ND packet is accepted now. Therefore, if you have 20 prefix options attached to RA, only the first 10 prefixes will be recognized. If this troubles you, please ask it on FREEBSD-CURRENT mailing list and/or modify nd6_maxndopt in `sys/netinet6/nd6.c`. If there are high demands we may provide sysctl knob for the variable.

14.1.1.3. Scope Index

IPv6 uses scoped addresses. Therefore, it is very important to specify scope index (interface index for link-local address, or site index for site-local address) with an IPv6 address. Without scope index, scoped IPv6 address is ambiguous to the kernel, and kernel will not be able to determine the outbound interface for a packet.

Ordinary userland applications should use advanced API (RFC2292) to specify scope index, or interface index. For similar purpose, sin6_scope_id member in sockaddr_in6 structure is defined in RFC2553. However, the semantics for sin6_scope_id is rather vague. If you care about portability of your application, we suggest you to use advanced API rather than sin6_scope_id.

In the kernel, an interface index for link-local scoped address is embedded into 2nd 16bit-word (3rd and 4th byte) in IPv6 address. For example, you may see something like:

```
fe80:1::200:f8ff:fe01:6317
```

in the routing table and interface address structure (struct in6_ifaddr). The address above is a link-local unicast address which belongs to a network interface whose interface identifier is 1. The embedded index enables us to identify IPv6 link local addresses over multiple interfaces effectively and with only a little code change.

Routing daemons and configuration programs, like route6d(8) and ifconfig(8), will need to manipulate the "embedded" scope index. These programs use routing sockets and ioctls (like SIOCGIFADDR_IN6) and the kernel API will return IPv6 addresses with 2nd 16bit-word filled in. The APIs are for manipulating kernel internal structure. Programs that use these APIs have to be prepared about differences in kernels anyway.

When you specify scoped address to the command line, NEVER write the embedded form (such as ff02:1::1 or fe80:2::fedc). This is not supposed to work. Always use standard form, like ff02::1 or fe80::fedc, with command line option for specifying interface (like `ping6 -I ne0 ff02::1`). In general, if a command does not have command line option to specify outgoing interface, that command is not ready to accept scoped address. This may seem to be opposite from IPv6's premise to support "dentist office" situation. We believe that specifications need some improvements for this.

Some of the userland tools support extended numeric IPv6 syntax, as documented in `draft-ietf-ipngwg-scopedaddr-format-00.txt`. You can specify outgoing link, by using name of the outgoing interface like "fe80::1%ne0". This way you will be able to specify link-local scoped address without much trouble.

To use this extension in your program, you'll need to use getaddrinfo(3), and getnameinfo(3) with NI_WITHSCOPEID. The implementation currently assumes 1-to-1 relationship between a link and an interface, which is stronger than what specs say.

14.1.1.4. Plug and Play

Most of the IPv6 stateless address autoconfiguration is implemented in the kernel. Neighbor Discovery functions are implemented in the kernel as a whole. Router Advertisement (RA) input for hosts is implemented in the kernel. Router Solicitation (RS) output for endhosts, RS input for routers, and RA output for routers are implemented in the userland.

14.1.1.4.1. Assignment of link-local, and special addresses

IPv6 link-local address is generated from IEEE802 address (ethernet MAC address). Each of interface is assigned an IPv6 link-local address automatically, when the interface becomes up (IFF_UP). Also, direct route for the link-local address is added to routing table.

Here is an output of netstat command:

```
Internet6:
Destination                  Gateway                  Flags      Netif Expire
fe80:1::%ed0/64              link#1                   UC          ed0
fe80:2::%ep0/64              link#2                   UC          ep0
```

Interfaces that has no IEEE802 address (pseudo interfaces like tunnel interfaces, or ppp interfaces) will borrow IEEE802 address from other interfaces, such as ethernet interfaces, whenever possible. If there is no IEEE802 hardware attached, last-resort pseudorandom value, which is from MD5(hostname), will be used as source of link-local address. If it is not suitable for your usage, you will need to configure the link-local address manually.

If an interface is not capable of handling IPv6 (such as lack of multicast support), link-local address will not be assigned to that interface. See section 2 for details.

Each interface joins the solicited multicast address and the link-local all-nodes multicast addresses (e.g. fe80::1:ff01:6317 and ff02::1, respectively, on the link the interface is attached). In addition to a link-local address, the loopback address (::1) will be assigned to the loopback interface. Also, ::1/128 and ff01::/32 are automatically added to routing table, and loopback interface joins node-local multicast group ff01::1.

14.1.1.4.2. Stateless address autoconfiguration on hosts

In IPv6 specification, nodes are separated into two categories: *routers* and *hosts*. Routers forward packets addressed to others, hosts does not forward the packets. net.inet6.ip6.forwarding defines whether this node is router or host (router if it is 1, host if it is 0).

When a host hears Router Advertisement from the router, a host may autoconfigure itself by stateless address autoconfiguration. This behavior can be controlled by net.inet6.ip6.accept_rtadv (host autoconfigures itself if it is set to 1). By autoconfiguration, network address prefix for the receiving interface (usually global address prefix) is added. Default route is also configured. Routers periodically generate Router Advertisement packets. To request an adjacent router to generate RA packet, a host can transmit Router Solicitation. To generate a RS packet at any time, use the *rtsol* command. rtsold(8) daemon is also available. rtsold(8) generates Router Solicitation whenever necessary, and it works great for nomadic usage (notebooks/laptops). If one wishes to ignore Router Advertisements, use sysctl to set net.inet6.ip6.accept_rtadv to 0.

To generate Router Advertisement from a router, use the rtadvd(8) daemon.

Note that, IPv6 specification assumes the following items, and nonconforming cases are left unspecified:

- Only hosts will listen to router advertisements
- Hosts have single network interface (except loopback)

Therefore, this is unwise to enable net.inet6.ip6.accept_rtadv on routers, or multi-interface host. A misconfigured node can behave strange (nonconforming configuration allowed for those who would like to do some experiments).

To summarize the sysctl knob:

```
accept_rtadv forwarding role of the node
--- --- ---
0 0 host (to be manually configured)
0 1 router
1 0 autoconfigured host
(spec assumes that host has single
interface only, autoconfigured host
with multiple interface is
out-of-scope)
1 1 invalid, or experimental
(out-of-scope of spec)
```

RFC2462 has validation rule against incoming RA prefix information option, in 5.5.3 (e). This is to protect hosts from malicious (or misconfigured) routers that advertise very short prefix lifetime. There was an update from Jim Bound to ipngwg mailing list (look for "(ipng 6712)" in the archive) and it is implemented Jim's update.

See 23.5.1.2 in the document for relationship between DAD and autoconfiguration.

14.1.1.5. Generic tunnel interface

GIF (Generic InterFace) is a pseudo interface for configured tunnel. Details are described in gif(4). Currently

- v6 in v6
- v6 in v4
- v4 in v6
- v4 in v4

are available. Use gifconfig(8) to assign physical (outer) source and destination address to gif interfaces. Configuration that uses same address family for inner and outer IP header (v4 in v4, or v6 in v6) is dangerous. It is very easy to configure interfaces and routing tables to perform infinite level of tunneling. *Please be warned*.

gif can be configured to be ECN-friendly. See 23.5.4.5 for ECN-friendliness of tunnels, and gif(4) for how to configure.

If you would like to configure an IPv4-in-IPv6 tunnel with gif interface, read gif(4) carefully. You will need to remove IPv6 link-local address automatically assigned to the gif interface.

14.1.1.6. Source Address Selection

Current source selection rule is scope oriented (there are some exceptions - see below). For a given destination, a source IPv6 address is selected by the following rule:

1. If the source address is explicitly specified by the user (e.g. via the advanced API), the specified address is used.
2. If there is an address assigned to the outgoing interface (which is usually determined by looking up the routing table) that has the same scope as the destination address, the address is used.

   This is the most typical case.

3. If there is no address that satisfies the above condition, choose a global address assigned to one of the interfaces on the sending node.
4. If there is no address that satisfies the above condition, and destination address is site local scope, choose a site local address assigned to one of the interfaces on the sending node.
5. If there is no address that satisfies the above condition, choose the address associated with the routing table entry for the destination. This is the last resort, which may cause scope violation.

For instance, ::1 is selected for ff01::1, fe80:1::200:f8ff:fe01:6317 for fe80:1::2a0:24ff:feab:839b (note that embedded interface index - described in 23.5.1.3 - helps us choose the right source address. Those embedded indices will not be on the wire). If the outgoing interface has multiple address for the scope, a source is selected longest match basis (rule 3). Suppose 3ffe:501:808:1:200:f8ff:fe01:6317 and 3ffe:2001:9:124:200:f8ff:fe01:6317 are given to the outgoing interface. 3ffe:501:808:1:200:f8ff:fe01:6317 is chosen as the source for the destination 3ffe:501:800::1.

Note that the above rule is not documented in the IPv6 spec. It is considered "up to implementation" item. There are some cases where we do not use the above rule. One example is connected TCP session, and we use the address kept in tcb as the source. Another example is source address for Neighbor Advertisement. Under the spec (RFC2461 7.2.2) NA's source should be the target address of the corresponding NS's target. In this case we follow the spec rather than the above longest-match rule.

For new connections (when rule 1 does not apply), deprecated addresses (addresses with preferred lifetime = 0) will not be chosen as source address if other choices are available. If no other choices are available, deprecated address will be used as a last resort. If there are multiple choice of deprecated addresses, the above scope rule will be used to choose from those deprecated addresses. If you would like to prohibit the use of deprecated address for some reason, configure net.inet6.ip6.use_deprecated to 0. The issue related to deprecated address is described in RFC2462 5.5.4 (NOTE: there is some debate underway in IETF ipngwg on how to use "deprecated" address).

14.1.1.7. Jumbo Payload

The Jumbo Payload hop-by-hop option is implemented and can be used to send IPv6 packets with payloads longer than 65,535 octets. But currently no physical interface whose MTU is more than 65,535 is supported, so such payloads can be seen only on the loopback interface (i.e. lo0).

If you want to try jumbo payloads, you first have to reconfigure the kernel so that the MTU of the loopback interface is more than 65,535 bytes; add the following to the kernel configuration file:

```
options "LARGE_LOMTU" #To test jumbo payload
```

and recompile the new kernel.

Then you can test jumbo payloads by the ping6(8) command with -b and -s options. The -b option must be specified to enlarge the size of the socket buffer and the -s option specifies the length of the packet, which should be more than 65,535. For example, type as follows:

```
% ping6 -b 70000 -s 68000 ::1
```

The IPv6 specification requires that the Jumbo Payload option must not be used in a packet that carries a fragment header. If this condition is broken, an ICMPv6 Parameter Problem message must be sent to the sender. specification is followed, but you cannot usually see an ICMPv6 error caused by this requirement.

When an IPv6 packet is received, the frame length is checked and compared to the length specified in the payload length field of the IPv6 header or in the value of the Jumbo Payload option, if any. If the former is shorter than the latter, the packet is discarded and statistics are incremented. You can see the statistics as output of netstat(8) command with ‘-s -p ip6’ option:

```
      % netstat -s -p ip6
      ip6:
    (snip)
    1 with data size < data length
```

So, kernel does not send an ICMPv6 error unless the erroneous packet is an actual Jumbo Payload, that is, its packet size is more than 65,535 bytes. As described above, currently no physical interface with such a huge MTU is supported, so it rarely returns an ICMPv6 error.

TCP/UDP over jumbogram is not supported at this moment. This is because we have no medium (other than loopback) to test this. Contact us if you need this.

IPsec does not work on jumbograms. This is due to some specification twists in supporting AH with jumbograms (AH header size influences payload length, and this makes it real hard to authenticate inbound packet with jumbo payload option as well as AH).

There are fundamental issues in *BSD support for jumbograms. We would like to address those, but we need more time to finalize these. To name a few:

- mbuf pkthdr.len field is typed as "int" in 4.4BSD, so it will not hold jumbogram with len > 2G on 32bit architecture CPUs. If we would like to support jumbogram properly, the field must be expanded to hold 4G + IPv6 header + link-layer header. Therefore, it must be expanded to at least int64_t (u_int32_t is NOT enough).

- We mistakingly use "int" to hold packet length in many places. We need to convert them into larger integral type. It needs a great care, as we may experience overflow during packet length computation.
- We mistakingly check for ip6_plen field of IPv6 header for packet payload length in various places. We should be checking mbuf pkthdr.len instead. ip6_input() will perform sanity check on jumbo payload option on input, and we can safely use mbuf pkthdr.len afterwards.
- TCP code needs a careful update in bunch of places, of course.

14.1.1.8. Loop prevention in header processing

IPv6 specification allows arbitrary number of extension headers to be placed onto packets. If we implement IPv6 packet processing code in the way BSD IPv4 code is implemented, kernel stack may overflow due to long function call chain. sys/netinet6 code is carefully designed to avoid kernel stack overflow. Because of this, sys/netinet6 code defines its own protocol switch structure, as "struct ip6protosw" (see `netinet6/ip6protosw.h`). There is no such update to IPv4 part (sys/netinet) for compatibility, but small change is added to its pr_input() prototype. So "struct ipprotosw" is also defined. Because of this, if you receive IPsec-over-IPv4 packet with massive number of IPsec headers, kernel stack may blow up. IPsec-over-IPv6 is okay. (Off-course, for those all IPsec headers to be processed, each such IPsec header must pass each IPsec check. So an anonymous attacker won't be able to do such an attack.)

14.1.1.9. ICMPv6

After RFC2463 was published, IETF ipngwg has decided to disallow ICMPv6 error packet against ICMPv6 redirect, to prevent ICMPv6 storm on a network medium. This is already implemented into the kernel.

14.1.1.10. Applications

For userland programming, we support IPv6 socket API as specified in RFC2553, RFC2292 and upcoming internet drafts.

TCP/UDP over IPv6 is available and quite stable. You can enjoy telnet(1), ftp(1), rlogin(1), rsh(1), ssh(1), etc. These applications are protocol independent. That is, they automatically chooses IPv4 or IPv6 according to DNS.

14.1.1.11. Kernel Internals

While ip_forward() calls ip_output(), ip6_forward() directly calls if_output() since routers must not divide IPv6 packets into fragments.

ICMPv6 should contain the original packet as long as possible up to 1280. UDP6/IP6 port unreach, for instance, should contain all extension headers and the *unchanged* UDP6 and IP6 headers. So, all IP6 functions except TCP never convert network byte order into host byte order, to save the original packet.

tcp_input(), udp6_input() and icmp6_input() can't assume that IP6 header is preceding the transport headers due to extension headers. So, in6_cksum() was implemented to handle packets whose IP6 header and transport header is not continuous. TCP/IP6 nor UDP6/IP6 header structure don't exist for checksum calculation.

To process IP6 header, extension headers and transport headers easily, network drivers are now required to store packets in one internal mbuf or one or more external mbufs. A typical old driver prepares two internal mbufs for 96 - 204 bytes data, however, now such packet data is stored in one external mbuf.

`netstat -s -p ip6` tells you whether or not your driver conforms such requirement. In the following example, "cce0" violates the requirement. (For more information, refer to Section 2.)

```
            Mbuf statistics:
                    317 one mbuf
                    two or more mbuf::
                            lo0 = 8
     cce0 = 10
                    3282 one ext mbuf
                    0 two or more ext mbuf
```

Each input function calls IP6_EXTHDR_CHECK in the beginning to check if the region between IP6 and its header is continuous. IP6_EXTHDR_CHECK calls m_pullup() only if the mbuf has M_LOOP flag, that is, the packet comes from the loopback interface. m_pullup() is never called for packets coming from physical network interfaces.

Both IP and IP6 reassemble functions never call m_pullup().

14.1.1.12. IPv4 mapped address and IPv6 wildcard socket

RFC2553 describes IPv4 mapped address (3.7) and special behavior of IPv6 wildcard bind socket (3.8). The spec allows you to:

- Accept IPv4 connections by AF_INET6 wildcard bind socket.
- Transmit IPv4 packet over AF_INET6 socket by using special form of the address like ::ffff:10.1.1.1.

but the spec itself is very complicated and does not specify how the socket layer should behave. Here we call the former one "listening side" and the latter one "initiating side", for reference purposes.

You can perform wildcard bind on both of the address families, on the same port.

The following table show the behavior of FreeBSD 4.x.

```
                  listening side          initiating side
                  (AF_INET6 wildcard      (connection to ::ffff:10.1.1.1)
                  socket gets IPv4 conn.)
                  ---                     ---
FreeBSD 4.x       configurable            supported
                  default: enabled
```

The following sections will give you more details, and how you can configure the behavior.

Comments on listening side:

It looks that RFC2553 talks too little on wildcard bind issue, especially on the port space issue, failure mode and relationship between AF_INET/INET6 wildcard bind. There can be several separate interpretation for this RFC which conform to it but behaves differently. So, to implement portable application you should assume nothing about the behavior in the kernel. Using getaddrinfo(3) is the safest way. Port number space and wildcard bind issues were discussed in detail on ipv6imp mailing list, in mid March 1999 and it looks that there's no concrete consensus (means, up to implementers). You may want to check the mailing list archives.

If a server application would like to accept IPv4 and IPv6 connections, there will be two alternatives.

One is using AF_INET and AF_INET6 socket (you'll need two sockets). Use getaddrinfo(3) with AI_PASSIVE into ai_flags, and socket(2) and bind(2) to all the addresses returned. By opening multiple sockets, you can accept connections onto the socket with proper address family. IPv4 connections will be accepted by AF_INET socket, and IPv6 connections will be accepted by AF_INET6 socket.

Another way is using one AF_INET6 wildcard bind socket. Use getaddrinfo(3) with AI_PASSIVE into ai_flags and with AF_INET6 into ai_family, and set the 1st argument hostname to NULL. And socket(2) and bind(2) to the address returned. (should be IPv6 unspecified addr). You can accept either of IPv4 and IPv6 packet via this one socket.

To support only IPv6 traffic on AF_INET6 wildcard binded socket portably, always check the peer address when a connection is made toward AF_INET6 listening socket. If the address is IPv4 mapped address, you may want to reject the connection. You can check the condition by using IN6_IS_ADDR_V4MAPPED() macro.

To resolve this issue more easily, there is system dependent setsockopt(2) option, IPV6_BINDV6ONLY, used like below.

```
int on;

setsockopt(s, IPPROTO_IPV6, IPV6_BINDV6ONLY,
   (char *)&on, sizeof (on)) < 0));
```

When this call succeed, then this socket only receive IPv6 packets.

Comments on initiating side:

Advise to application implementers: to implement a portable IPv6 application (which works on multiple IPv6 kernels), we believe that the following is the key to the success:

- NEVER hardcode AF_INET nor AF_INET6.
- Use getaddrinfo(3) and getnameinfo(3) throughout the system. Never use gethostby*(), getaddrby*(), inet_*() or getipnodeby*(). (To update existing applications to be IPv6 aware easily, sometime getipnodeby*() will be useful. But if possible, try to rewrite the code to use getaddrinfo(3) and getnameinfo(3).)
- If you would like to connect to destination, use getaddrinfo(3) and try all the destination returned, like telnet(1) does.
- Some of the IPv6 stack is shipped with buggy getaddrinfo(3). Ship a minimal working version with your application and use that as last resort.

If you would like to use AF_INET6 socket for both IPv4 and IPv6 outgoing connection, you will need to use getipnodebyname(3). When you would like to update your existing application to be IPv6 aware with minimal effort, this approach might be chosen. But please note that it is a temporal solution, because getipnodebyname(3) itself is not recommended as it does not handle scoped IPv6 addresses at all. For IPv6 name resolution, getaddrinfo(3) is the preferred API. So you should rewrite your application to use getaddrinfo(3), when you get the time to do it.

When writing applications that make outgoing connections, story goes much simpler if you treat AF_INET and AF_INET6 as totally separate address family. {set,get}sockopt issue goes simpler, DNS issue will be made simpler. We do not recommend you to rely upon IPv4 mapped address.

14.1.1.12.1. unified tcp and inpcb code

FreeBSD 4.x uses shared tcp code between IPv4 and IPv6 (from sys/netinet/tcp*) and separate udp4/6 code. It uses unified inpcb structure.

The platform can be configured to support IPv4 mapped address. Kernel configuration is summarized as follows:

- By default, AF_INET6 socket will grab IPv4 connections in certain condition, and can initiate connection to IPv4 destination embedded in IPv4 mapped IPv6 address.
- You can disable it on entire system with sysctl like below.

  ```
  sysctl -w net.inet6.ip6.mapped_addr=0
  ```

14.1.1.12.1.1. listening side

Each socket can be configured to support special AF_INET6 wildcard bind (enabled by default). You can disable it on each socket basis with setsockopt(2) like below.

```
int on;

setsockopt(s, IPPROTO_IPV6, IPV6_BINDV6ONLY,
   (char *)&on, sizeof (on)) < 0));
```

Wildcard AF_INET6 socket grabs IPv4 connection if and only if the following conditions are satisfied:

- there's no AF_INET socket that matches the IPv4 connection
- the AF_INET6 socket is configured to accept IPv4 traffic, i.e. getsockopt(IPV6_BINDV6ONLY) returns 0.

There's no problem with open/close ordering.

14.1.1.12.1.2. initiating side

FreeBSD 4.x supports outgoing connection to IPv4 mapped address (::ffff:10.1.1.1), if the node is configured to support IPv4 mapped address.

14.1.1.13. sockaddr_storage

When RFC2553 was about to be finalized, there was discussion on how struct sockaddr_storage members are named. One proposal is to prepend "__" to the members (like "__ss_len") as they should not be touched. The other proposal was that don't prepend it (like "ss_len") as we need to touch those members directly. There was no clear consensus on it.

As a result, RFC2553 defines struct sockaddr_storage as follows:

```
struct sockaddr_storage {
u_char __ss_len; /* address length */
u_char __ss_family; /* address family */
/* and bunch of padding */
};
```

On the contrary, XNET draft defines as follows:

```
struct sockaddr_storage {
u_char ss_len; /* address length */
u_char ss_family; /* address family */
/* and bunch of padding */
};
```

In December 1999, it was agreed that RFC2553bis should pick the latter (XNET) definition.

Current implementation conforms to XNET definition, based on RFC2553bis discussion.

If you look at multiple IPv6 implementations, you will be able to see both definitions. As an userland programmer, the most portable way of dealing with it is to:

1. ensure ss_family and/or ss_len are available on the platform, by using GNU autoconf,
2. have -Dss_family=__ss_family to unify all occurrences (including header file) into __ss_family, or
3. never touch __ss_family. cast to sockaddr * and use sa_family like:

   ```
   struct sockaddr_storage ss;
   family = ((struct sockaddr *)&ss)->sa_family
   ```

14.1.2. Network Drivers

Now following two items are required to be supported by standard drivers:

1. mbuf clustering requirement. In this stable release, we changed MINCLSIZE into MHLEN+1 for all the operating systems in order to make all the drivers behave as we expect.
2. multicast. If ifmcstat(8) yields no multicast group for a interface, that interface has to be patched.

If any of the driver don't support the requirements, then the driver can't be used for IPv6 and/or IPsec communication. If you find any problem with your card using IPv6/IPsec, then, please report it to <`freebsd-bugs@FreeBSD.org`>.

(NOTE: In the past we required all PCMCIA drivers to have a call to in6_ifattach(). We have no such requirement any more)

14.1.3. Translator

We categorize IPv4/IPv6 translator into 4 types:

- *Translator A* --- It is used in the early stage of transition to make it possible to establish a connection from an IPv6 host in an IPv6 island to an IPv4 host in the IPv4 ocean.

- *Translator B* --- It is used in the early stage of transition to make it possible to establish a connection from an IPv4 host in the IPv4 ocean to an IPv6 host in an IPv6 island.
- *Translator C* --- It is used in the late stage of transition to make it possible to establish a connection from an IPv4 host in an IPv4 island to an IPv6 host in the IPv6 ocean.
- *Translator D* --- It is used in the late stage of transition to make it possible to establish a connection from an IPv6 host in the IPv6 ocean to an IPv4 host in an IPv4 island.

TCP relay translator for category A is supported. This is called "FAITH". We also provide IP header translator for category A. (The latter is not yet put into FreeBSD 4.x yet.)

14.1.3.1. FAITH TCP relay translator

FAITH system uses TCP relay daemon called faithd(8) helped by the kernel. FAITH will reserve an IPv6 address prefix, and relay TCP connection toward that prefix to IPv4 destination.

For example, if the reserved IPv6 prefix is 3ffe:0501:0200:ffff::, and the IPv6 destination for TCP connection is 3ffe:0501:0200:ffff::163.221.202.12, the connection will be relayed toward IPv4 destination 163.221.202.12.

```
destination IPv4 node (163.221.202.12)
  ^
  | IPv4 tcp toward 163.221.202.12
FAITH-relay dual stack node
  ^
  | IPv6 TCP toward 3ffe:0501:0200:ffff::163.221.202.12
source IPv6 node
```

faithd(8) must be invoked on FAITH-relay dual stack node.

For more details, consult `src/usr.sbin/faithd/README`

14.1.4. IPsec

IPsec is mainly organized by three components.

1. Policy Management
2. Key Management
3. AH and ESP handling

14.1.4.1. Policy Management

The kernel implements experimental policy management code. There are two way to manage security policy. One is to configure per-socket policy using setsockopt(2). In this cases, policy configuration is described in ipsec_set_policy(3). The other is to configure kernel packet filter-based policy using PF_KEY interface, via setkey(8).

The policy entry is not re-ordered with its indexes, so the order of entry when you add is very significant.

14.1.4.2. Key Management

The key management code implemented in this kit (sys/netkey) is a home-brew PFKEY v2 implementation. This conforms to RFC2367.

The home-brew IKE daemon, "racoon" is included in the kit (kame/kame/racoon). Basically you'll need to run racoon as daemon, then setup a policy to require keys (like `ping -P 'out ipsec esp/transport//use'`). The kernel will contact racoon daemon as necessary to exchange keys.

14.1.4.3. AH and ESP handling

IPsec module is implemented as "hooks" to the standard IPv4/IPv6 processing. When sending a packet, ip{,6}_output() checks if ESP/AH processing is required by checking if a matching SPD (Security Policy Database) is found. If ESP/AH is needed, {esp,ah}{4,6}_output() will be called and mbuf will be updated accordingly. When a packet is received, {esp,ah}4_input() will be called based on protocol number, i.e. (*inetsw[proto])(). {esp,ah}4_input() will decrypt/check authenticity of the packet, and strips off daisy-chained header and padding for ESP/AH. It is safe to strip off the ESP/AH header on packet reception, since we will never use the received packet in "as is" form.

By using ESP/AH, TCP4/6 effective data segment size will be affected by extra daisy-chained headers inserted by ESP/AH. Our code takes care of the case.

Basic crypto functions can be found in directory "sys/crypto". ESP/AH transform are listed in {esp,ah}_core.c with wrapper functions. If you wish to add some algorithm, add wrapper function in {esp,ah}_core.c, and add your crypto algorithm code into sys/crypto.

Tunnel mode is partially supported in this release, with the following restrictions:

- IPsec tunnel is not combined with GIF generic tunneling interface. It needs a great care because we may create an infinite loop between ip_output() and tunnelifp->if_output(). Opinion varies if it is better to unify them, or not.
- MTU and Don't Fragment bit (IPv4) considerations need more checking, but basically works fine.
- Authentication model for AH tunnel must be revisited. We'll need to improve the policy management engine, eventually.

14.1.4.4. Conformance to RFCs and IDs

The IPsec code in the kernel conforms (or, tries to conform) to the following standards:

"old IPsec" specification documented in `rfc182[5-9].txt`

"new IPsec" specification documented in `rfc240[1-6].txt`, `rfc241[01].txt`, `rfc2451.txt` and `draft-mcdonald-simple-ipsec-api-01.txt` (draft expired, but you can take from ftp://ftp.kame.net/pub/internet-drafts/ (ftp://ftp.kame.net/pub/internet-drafts/)). (NOTE: IKE specifications, `rfc241[7-9].txt` are implemented in userland, as "racoon" IKE daemon)

Currently supported algorithms are:

- old IPsec AH
  - null crypto checksum (no document, just for debugging)
  - keyed MD5 with 128bit crypto checksum (`rfc1828.txt`)
  - keyed SHA1 with 128bit crypto checksum (no document)
  - HMAC MD5 with 128bit crypto checksum (`rfc2085.txt`)
  - HMAC SHA1 with 128bit crypto checksum (no document)
- old IPsec ESP
  - null encryption (no document, similar to `rfc2410.txt`)
  - DES-CBC mode (`rfc1829.txt`)
- new IPsec AH
  - null crypto checksum (no document, just for debugging)
  - keyed MD5 with 96bit crypto checksum (no document)
  - keyed SHA1 with 96bit crypto checksum (no document)
  - HMAC MD5 with 96bit crypto checksum (`rfc2403.txt`)
  - HMAC SHA1 with 96bit crypto checksum (`rfc2404.txt`)
- new IPsec ESP
  - null encryption (`rfc2410.txt`)
  - DES-CBC with derived IV (`draft-ietf-ipsec-ciph-des-derived-01.txt`, draft expired)
  - DES-CBC with explicit IV (`rfc2405.txt`)

- 3DES-CBC with explicit IV (`rfc2451.txt`)
- BLOWFISH CBC (`rfc2451.txt`)
- CAST128 CBC (`rfc2451.txt`)
- RC5 CBC (`rfc2451.txt`)
- each of the above can be combined with:
  - ESP authentication with HMAC-MD5(96bit)
  - ESP authentication with HMAC-SHA1(96bit)

The following algorithms are NOT supported:

- old IPsec AH
  - HMAC MD5 with 128bit crypto checksum + 64bit replay prevention (`rfc2085.txt`)
  - keyed SHA1 with 160bit crypto checksum + 32bit padding (`rfc1852.txt`)

IPsec (in kernel) and IKE (in userland as "racoon") has been tested at several interoperability test events, and it is known to interoperate with many other implementations well. Also, current IPsec implementation as quite wide coverage for IPsec crypto algorithms documented in RFC (we cover algorithms without intellectual property issues only).

14.1.4.5. ECN consideration on IPsec tunnels

ECN-friendly IPsec tunnel is supported as described in `draft-ipsec-ecn-00.txt`.

Normal IPsec tunnel is described in RFC2401. On encapsulation, IPv4 TOS field (or, IPv6 traffic class field) will be copied from inner IP header to outer IP header. On decapsulation outer IP header will be simply dropped. The decapsulation rule is not compatible with ECN, since ECN bit on the outer IP TOS/traffic class field will be lost.

To make IPsec tunnel ECN-friendly, we should modify encapsulation and decapsulation procedure. This is described in http://www.aciri.org/floyd/papers/draft-ipsec-ecn-00.txt (http://www.aciri.org/floyd/papers/draft-ipsec-ecn-00.txt), chapter 3.

IPsec tunnel implementation can give you three behaviors, by setting net.inet.ipsec.ecn (or net.inet6.ipsec6.ecn) to some value:

- RFC2401: no consideration for ECN (sysctl value -1)

- ECN forbidden (sysctl value 0)
- ECN allowed (sysctl value 1)

Note that the behavior is configurable in per-node manner, not per-SA manner (draft-ipsec-ecn-00 wants per-SA configuration, but it looks too much for me).

The behavior is summarized as follows (see source code for more detail):

```
                encapsulate                     decapsulate
                ---                             ---
RFC2401         copy all TOS bits               drop TOS bits on outer
                from inner to outer.            (use inner TOS bits as is)

ECN forbidden   copy TOS bits except for ECN    drop TOS bits on outer
                (masked with 0xfc) from inner   (use inner TOS bits as is)
                to outer.  set ECN bits to 0.

ECN allowed     copy TOS bits except for ECN    use inner TOS bits with some
                CE (masked with 0xfe) from      change.  if outer ECN CE bit
                inner to outer.                 is 1, enable ECN CE bit on
                set ECN CE bit to 0.            the inner.
```

General strategy for configuration is as follows:

- if both IPsec tunnel endpoint are capable of ECN-friendly behavior, you’d better configure both end to "ECN allowed" (sysctl value 1).
- if the other end is very strict about TOS bit, use "RFC2401" (sysctl value -1).
- in other cases, use "ECN forbidden" (sysctl value 0).

The default behavior is "ECN forbidden" (sysctl value 0).

For more information, please refer to:

http://www.aciri.org/floyd/papers/draft-ipsec-ecn-00.txt (http://www.aciri.org/floyd/papers/draft-ipsec-ecn-00.txt), RFC2481 (Explicit Congestion Notification), src/sys/netinet6/{ah,esp}_input.c

(Thanks goes to Kenjiro Cho <`kjc@csl.sony.co.jp`> for detailed analysis)

14.1.4.6. Interoperability

Here are (some of) platforms that KAME code have tested IPsec/IKE interoperability in the past. Note that both ends may have modified their implementation, so use the following list just for reference purposes.

Altiga, Ashley-laurent (vpcom.com), Data Fellows (F-Secure), Ericsson ACC, FreeS/WAN, HITACHI, IBM AIX, IIJ, Intel, Microsoft WinNT, NIST (linux IPsec + plutoplus), Netscreen, OpenBSD, RedCreek, Routerware, SSH, Secure Computing, Soliton, Toshiba, VPNet, Yamaha RT100i

VIII. Network Filesystems

Chapter 15. AFS

AFS, NFS, SANs etc]

IX. Terminal Handling

Chapter 16. Syscons

Syscons, tty, PCVT, serial console, screen savers, etc

X. Sound

Chapter 17. OSS

OSS, waveforms, etc

XI. Device Drivers

Chapter 18. Writing FreeBSD Device Drivers

This chapter was written by Murray Stokely <`murray@FreeBSD.org`> with selections from a variety of sources including the intro(4) man page by Jörg Wunsch <`joerg@FreeBSD.org`>.

18.1. Introduction

This chapter provides a brief introduction to writing device drivers for FreeBSD. A device in this context is a term used mostly for hardware-related stuff that belongs to the system, like disks, printers, or a graphics display with its keyboard. A device driver is the software component of the operating system that controls a specific device. There are also so-called pseudo-devices where a device driver emulates the behaviour of a device in software without any particular underlying hardware. Device drivers can be compiled into the system statically or loaded on demand through the dynamic kernel linker facility 'kld'.

Most devices in a Unix-like operating system are accessed through device-nodes, sometimes also called special files. These files are usually located under the directory `/dev` in the file system hierarchy. Until devfs is fully integrated into FreeBSD, each device node must be created statically and independent of the existence of the associated device driver. Most device nodes on the system are created by running `MAKEDEV`.

Device drivers can roughly be broken down into two categories; character and network device drivers.

18.2. Dynamic Kernel Linker Facility - KLD

The kld interface allows system administrators to dynamically add and remove functionality from a running system. This allows device driver writers to load their new changes into a running kernel without constantly rebooting to test changes.

The kld interface is used through the following administrator commands :

- `kldload` - loads a new kernel module
- `kldunload` - unloads a kernel module
- `kldstat` - lists the currently loadded modules

Skeleton Layout of a kernel module

```
/*
 * KLD Skeleton
 * Inspired by Andrew Reiter’s Daemonnews article
 */

#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h>  /* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  /* defines used in kernel.h */
#include <sys/kernel.h> /* types used in module initialization */

/*
 * Load handler that deals with the loading and unloading of a KLD.
 */

static int
skel_loader(struct module *m, int what, void *arg)
{
  int err = 0;

  switch (what) {
  case MOD_LOAD:                /* kldload */
    uprintf("Skeleton KLD loaded.\n");
    break;
  case MOD_UNLOAD:
    uprintf("Skeleton KLD unloaded.\n");
    break;
  default:
    err = EINVAL;
    break;
  }
  return(err);
}

/* Declare this module to the rest of the kernel */

DECLARE_MODULE(skeleton, skel_loader, SI_SUB_KLD, SI_ORDER_ANY);
```

18.2.1. Makefile

FreeBSD provides a makefile include that you can use to quickly compile your kernel addition.

```
SRCS=skeleton.c
KMOD=skeleton

.include <bsd.kmod.mk>
```

Simply running `make` with this makefile will create a file `skeleton.ko` that can be loaded into your system by typing :

```
#   kldload -v ./skeleton.ko
```

18.3. Accessing a device driver

Unix provides a common set of system calls for user applications to use. The upper layers of the kernel dispatch these calls to the corresponding device driver when a user accesses a device node. The `/dev/MAKEDEV` script makes most of the device nodes for your system but if you are doing your own driver development it may be necessary to create your own device nodes with `mknod`

18.3.1. Creating static device nodes

The `mknod` command requires four arguments to create a device node. You must specify the name of this device node, the type of device, the major number of the device, and the minor number of the device.

18.3.2. Dynamic device nodes

The device filesystem, or devfs, provides access to the kernel's device namespace in the global filesystem namespace. This eliminates the problems of potentially having a device driver without a static device node, or a device node without an installed device driver. Devfs is still a work in progress, but it is already working quite nice.

18.4. Character Devices

A character device driver is one that transfers data directly to and from a user process. This is the most common type of device driver and there are plenty of simple examples in the source tree.

This simple example pseudo-device remembers whatever values you write to it and can then supply them back to you when you read from it.

```
/*
 * Simple ‘echo’ pseudo-device KLD
 *
 * Murray Stokely
 */

#define MIN(a,b) (((a) < (b)) ? (a) : (b))

#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h> /* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  /* defines used in kernel.h */
#include <sys/kernel.h> /* types used in module initialization */
#include <sys/conf.h>   /* cdevsw struct */
#include <sys/uio.h>    /* uio struct */
#include <sys/malloc.h>

#define BUFFERSIZE 256

/* Function prototypes */
d_open_t      echo_open;
d_close_t     echo_close;
d_read_t      echo_read;
d_write_t     echo_write;

/* Character device entry points */
static struct cdevsw echo_cdevsw = {
  echo_open,
  echo_close,
  echo_read,
  echo_write,
  noioctl,
  nopoll,
  nommap,
  nostrategy,
  "echo",
  33,                     /* reserved for lkms - /usr/src/sys/conf/majors */
  nodump,
  nopsize,
  D_TTY,
  -1
};

typedef struct s_echo {
  char msg[BUFFERSIZE];
  int len;
} t_echo;
```

```
/* vars */
static dev_t sdev;
static int len;
static int count;
static t_echo *echomsg;

MALLOC_DECLARE(M_ECHOBUF);
MALLOC_DEFINE(M_ECHOBUF, "echobuffer", "buffer for echo module");

/*
 * This function acts is called by the kld[un]load(2) system calls to
 * determine what actions to take when a module is loaded or unloaded.
 */

static int
echo_loader(struct module *m, int what, void *arg)
{
  int err = 0;

  switch (what) {
  case MOD_LOAD:                /* kldload */
    sdev = make_dev(&echo_cdevsw,
     0,
     UID_ROOT,
     GID_WHEEL,
     0600,
     "echo");
    /* kmalloc memory for use by this driver */
    /*    malloc(256,M_ECHOBUF,M_WAITOK); */
    MALLOC(echomsg, t_echo *, sizeof(t_echo), M_ECHOBUF, M_WAITOK);
    printf("Echo device loaded.\n");
    break;
  case MOD_UNLOAD:
    destroy_dev(sdev);
    FREE(echomsg,M_ECHOBUF);
    printf("Echo device unloaded.\n");
    break;
  default:
    err = EINVAL;
    break;
  }
  return(err);
}
```

```
    int
    echo_open(dev_t dev, int oflags, int devtype, struct proc *p)
    {
      int err = 0;

      uprintf("Opened device \"echo\" successfully.\n");
      return(err);
    }

    int
    echo_close(dev_t dev, int fflag, int devtype, struct proc *p)
    {
      uprintf("Closing device \"echo.\"\n");
      return(0);
    }

    /*
     * The read function just takes the buf that was saved via
     * echo_write() and returns it to userland for accessing.
     * uio(9)
     */

    int
    echo_read(dev_t dev, struct uio *uio, int ioflag)
    {
      int err = 0;
      int amt;

      /* How big is this read operation?  Either as big as the user wants,
         or as big as the remaining data */
      amt = MIN(uio->uio_resid, (echomsg->len - uio->uio_offset > 0) ? echomsg->len - uio-
>uio_offset : 0);
      if ((err = uiomove(echomsg->msg + uio->uio_offset,amt,uio)) != 0) {
        uprintf("uiomove failed!\n");
      }

      return err;
    }

    /*
     * echo_write takes in a character string and saves it
     * to buf for later accessing.
     */

    int
```

```
echo_write(dev_t dev, struct uio *uio, int ioflag)
{
  int err = 0;

  /* Copy the string in from user memory to kernel memory */
  err = copyin(uio->uio_iov->iov_base, echomsg->msg, MIN(uio->uio_iov->iov_len,BUFFERSIZE));

  /* Now we need to null terminate */
  *(echomsg->msg + MIN(uio->uio_iov->iov_len,BUFFERSIZE)) = 0;
  /* Record the length */
  echomsg->len = MIN(uio->uio_iov->iov_len,BUFFERSIZE);

  if (err != 0) {
    uprintf("Write failed: bad address!\n");
  }

  count++;
  return(err);
}

DEV_MODULE(echo,echo_loader,NULL);
```

To install this driver you will first need to make a node on your filesystem with a command such as :

```
 # mknod /dev/echo c 33 0
```

With this driver loaded you should now be able to type something like :

```
        # echo -n "Test Data" > /dev/echo
        # cat /dev/echo
Test Data
```

Real hardware devices in the next chapter..

Additional Resources

- Dynamic Kernel Linker (KLD) Facility Programming Tutorial (http://www.daemonnews.org/200010/blueprints.html) - Daemonnews (http://www.daemonnews.org) October 2000
- How to Write Kernel Drivers with NEWBUS (http://www.daemonnews.org/200007/newbus-intro.html) - Daemonnews (http://www.daemonnews.org) July 2000

18.5. Network Drivers

Drivers for network devices do not use device nodes in order to be accessed. Their selection is based on other decisions made inside the kernel and instead of calling open(), use of a network device is generally introduced by using the system call socket(2).

man ifnet(), loopback device, Bill Paul's drivers, etc..

Chapter 19. ISA device drivers

This chapter was written by Sergey Babkin <`babkin@FreeBSD.org`> Modifications for the handbook made by Murray Stokely <`murray@FreeBSD.org`>, Wylie Stilwell <`wylie@osd.bsdi.com`>, and Valentino Vaschetto <`logo@blackened.com`>.

19.1. Synopsis

This chapter introduces the issues relevant to writing a driver for an ISA device. The pseudo-code presented here is rather detailed and reminiscent of the real code but is still only pseudo-code. It avoids the details irrelevant to the subject of the discussion. The real-life examples can be found in the source code of real drivers. In particular the drivers "ep" and "aha" are good sources of information.

19.2. Basic information

A typical ISA driver would need the following include files:

```
#include <sys/module.h>
#include <sys/bus.h>
#include <machine/bus.h>
#include <machine/resource.h>
#include <sys/rman.h>

#include <isa/isavar.h>
#include <isa/pnpvar.h>
```

They describe the things specific to the ISA and generic bus subsystem.

The bus subsystem is implemented in an object-oriented fashion, its main structures are accessed by associated method functions.

The list of bus methods implemented by an ISA driver is like one for any other bus. For a hypothetical driver named "xxx" they would be:

- `static void xxx_isa_identify (driver_t *, device_t);` Normally used for bus drivers, not device drivers. But for ISA devices this method may have special use: if the device provides some device-specific (non-PnP) way to auto-detect devices this routine may implement it.
- `static int xxx_isa_probe (device_t dev);` Probe for a device at a known (or PnP) location. This routine can also accommodate device-specific auto-detection of parameters for partially configured devices.

- `static int xxx_isa_attach (device_t dev);` Attach and initialize device.
- `static int xxx_isa_detach (device_t dev);` Detach device before unloading the driver module.
- `static int xxx_isa_shutdown (device_t dev);` Execute shutdown of the device before system shutdown.
- `static int xxx_isa_suspend (device_t dev);` Suspend the device before the system goes to the power-save state. May also abort transition to the power-save state.
- `static int xxx_isa_resume (device_t dev);` Resume the device activity after return from power-save state.

`xxx_isa_probe()` and `xxx_isa_attach()` are mandatory, the rest of the routines are optional, depending on the device's needs.

The driver is linked to the system with the following set of descriptions.

```
        /* table of supported bus methods */
        static device_method_t xxx_isa_methods[] = {
            /* list all the bus method functions supported by the driver */
            /* omit the unsupported methods */
            DEVMETHOD(device_identify,  xxx_isa_identify),
            DEVMETHOD(device_probe,     xxx_isa_probe),
            DEVMETHOD(device_attach,    xxx_isa_attach),
            DEVMETHOD(device_detach,    xxx_isa_detach),
            DEVMETHOD(device_shutdown,  xxx_isa_shutdown),
            DEVMETHOD(device_suspend,   xxx_isa_suspend),
            DEVMETHOD(device_resume,    xxx_isa_resume),

     { 0, 0 }
        };

        static driver_t xxx_isa_driver = {
            "xxx",
            xxx_isa_methods,
            sizeof(struct xxx_softc),
        };


        static devclass_t xxx_devclass;

        DRIVER_MODULE(xxx, isa, xxx_isa_driver, xxx_devclass,
            load_function, load_argument);
```

Here struct xxx_softc is a device-specific structure that contains private driver data and descriptors for the driver's resources. The bus code automatically allocates one softc descriptor per device as needed.

If the driver is implemented as a loadable module then `load_function()` is called to do driver-specific initialization or clean-up when the driver is loaded or unloaded and load_argument is passed as one of its arguments. If the driver does not support dynamic loading (in other words it must always be linked into kernel) then these values should be set to 0 and the last definition would look like:

```
DRIVER_MODULE(xxx, isa, xxx_isa_driver,
      xxx_devclass, 0, 0);
```

If the driver is for a device which supports PnP then a table of supported PnP IDs must be defined. The table consists of a list of PnP IDs supported by this driver and human-readable descriptions of the hardware types and models having these IDs. It looks like:

```
    static struct isa_pnp_id xxx_pnp_ids[] = {
        /* a line for each supported PnP ID */
        { 0x12345678,   "Our device model 1234A" },
        { 0x12345679,   "Our device model 1234B" },
        { 0,        NULL }, /* end of table */
    };
```

If the driver does not support PnP devices it still needs an empty PnP ID table, like:

```
    static struct isa_pnp_id xxx_pnp_ids[] = {
        { 0,        NULL }, /* end of table */
    };
```

19.3. Device_t pointer

Device_t is the pointer type for the device structure. Here we consider only the methods interesting from the device driver writer's standpoint. The methods to manipulate values in the device structure are:

- `device_t device_get_parent(dev)` Get the parent bus of a device.
- `driver_t device_get_driver(dev)` Get pointer to its driver structure.
- `char *device_get_name(dev)` Get the driver name, such as "xxx" for our example.
- `int device_get_unit(dev)` Get the unit number (units are numbered from 0 for the devices associated with each driver).
- `char *device_get_nameunit(dev)` Get the device name including the unit number, such as "xxx0" , "xxx1" and so on.
- `char *device_get_desc(dev)` Get the device description. Normally it describes the exact model of device in human-readable form.

- `device_set_desc(dev, desc)` Set the description. This makes the device description point to the string desc which may not be deallocated or changed after that.
- `device_set_desc_copy(dev, desc)` Set the description. The description is copied into an internal dynamically allocated buffer, so the string desc may be changed afterwards without adverse effects.
- `void *device_get_softc(dev)` Get pointer to the device descriptor (struct xxx_softc) associated with this device.
- `u_int32_t device_get_flags(dev)` Get the flags specified for the device in the configuration file.

A convenience function `device_printf(dev, fmt, ...)` may be used to print the messages from the device driver. It automatically prepends the unitname and colon to the message.

The device_t methods are implemented in the file kern/bus_subr.c.

19.4. Config file and the order of identifying and probing during auto-configuration

The ISA devices are described in the kernel config file like:

```
device xxx0 at isa? port 0x300 irq 10 drq 5
       iomem 0xd0000 flags 0x1 sensitive
```

The values of port, IRQ and so on are converted to the resource values associated with the device. They are optional, depending on the device needs and abilities for auto-configuration. For example, some devices don’t need DRQ at all and some allow the driver to read the IRQ setting from the device configuration ports. If a machine has multiple ISA buses the exact bus may be specified in the configuration line, like "isa0" or "isa1", otherwise the device would be searched for on all the ISA buses.

"sensitive" is a resource requesting that this device must be probed before all non-sensitive devices. It is supported but does not seem to be used in any current driver.

For legacy ISA devices in many cases the drivers are still able to detect the configuration parameters. But each device to be configured in the system must have a config line. If two devices of some type are installed in the system but there is only one configuration line for the corresponding driver, ie:

```
device xxx0 at isa?
```

then only one device will be configured.

But for the devices supporting automatic identification by the means of Plug-n-Play or some proprietary protocol one configuration line is enough to configure all the devices in the system, like the one above or just simply:

```
device xxx at isa?
```

If a driver supports both auto-identified and legacy devices and both kinds are installed at once in one machine then it's enough to describe in the config file the legacy devices only. The auto-identified devices will be added automatically.

When an ISA bus is auto-configured the events happen as follows:

All the drivers' identify routines (including the PnP identify routine which identifies all the PnP devices) are called in random order. As they identify the devices they add them to the list on the ISA bus. Normally the drivers' identify routines associate their drivers with the new devices. The PnP identify routine does not know about the other drivers yet so it does not associate any with the new devices it adds.

The PnP devices are put to sleep using the PnP protocol to prevent them from being probed as legacy devices.

The probe routines of non-PnP devices marked as "sensitive" are called. If probe for a device went successfully, the attach routine is called for it.

The probe and attach routines of all non-PNP devices are called likewise.

The PnP devices are brought back from the sleep state and assigned the resources they request: I/O and memory address ranges, IRQs and DRQs, all of them not conflicting with the attached legacy devices.

Then for each PnP device the probe routines of all the present ISA drivers are called. The first one that claims the device gets attached. It is possible that multiple drivers would claim the device with different priority, the highest-priority driver wins. The probe routines must call `ISA_PNP_PROBE()` to compare the actual PnP ID with the list of the IDs supported by the driver and if the ID is not in the table return failure. That means that absolutely every driver, even the ones not supporting any PnP devices must call `ISA_PNP_PROBE()`, at least with an empty PnP ID table to return failure on unknown PnP devices.

The probe routine returns a positive value (the error code) on error, zero or negative value on success.

The negative return values are used when a PnP device supports multiple interfaces. For example, an older compatibility interface and a newer advanced interface which are supported by different drivers. Then both drivers would detect the device. The driver which returns a higher value in the probe routine takes precedence (in other words, the driver returning 0 has highest precedence, returning -1 is next, returning -2 is after it and so on). In result the devices which support only the old interface will be handled by the old driver (which should return -1 from the probe routine) while the devices supporting the new interface as well will be handled by the new driver (which should return 0 from the probe routine). If multiple drivers return the same value then the one called first wins. So if a driver returns value 0 it may be sure that it won the priority arbitration.

The device-specific identify routines can also assign not a driver but a class of drivers to the device. Then all the drivers in the class are probed for this device, like the case with PnP. This feature is not implemented in any existing driver and is not considered further in this document.

Because the PnP devices are disabled when probing the legacy devices they will not be attached twice (once as legacy and once as PnP). But in case of device-dependent identify routines it's the responsibility of the driver to make sure that the same device won't be attached by the driver twice: once as legacy user-configured and once as auto-identified.

Another practical consequence for the auto-identified devices (both PnP and device-specific) is that the flags can not be passed to them from the kernel configuration file. So they must either not use the flags at all or use the flags from the device unit 0 for all the auto-identified devices or use the sysctl interface instead of flags.

Other unusual configurations may be accommodated by accessing the configuration resources directly with functions of families `resource_query_*()` and `resource_*_value()`. Their implementations are located in kern/subr_bus.h. The old IDE disk driver i386/isa/wd.c contains examples of such use. But the standard means of configuration must always be preferred. Leave parsing the configuration resources to the bus configuration code.

19.5. Resources

The information that a user enters into the kernel configuration file is processed and passed to the kernel as configuration resources. This information is parsed by the bus configuration code and transformed into a value of structure device_t and the bus resources associated with it. The drivers may access the configuration resources directly using functions resource_* for more complex cases of configuration. But generally it's not needed nor recommended, so this issue is not discussed further.

The bus resources are associated with each device. They are identified by type and number within the type. For the ISA bus the following types are defined:

- *SYS_RES_IRQ* - interrupt number
- *SYS_RES_DRQ* - ISA DMA channel number
- *SYS_RES_MEMORY* - range of device memory mapped into the system memory space
- *SYS_RES_IOPORT* - range of device I/O registers

The enumeration within types starts from 0, so if a device has two memory regions if would have resources of type SYS_RES_MEMORY numbered 0 and 1. The resource type has nothing to do with the C language type, all the resource values have the C language type "unsigned long" and must be cast as necessary. The resource numbers don't have to be contiguous although for ISA they normally would be. The permitted resource numbers for ISA devices are:

```
IRQ: 0-1
DRQ: 0-1
MEMORY: 0-3
IOPORT: 0-7
```

All the resources are represented as ranges, with a start value and count. For IRQ and DRQ resources the count would be normally equal to 1. The values for memory refer to the physical addresses.

Three types of activities can be performed on resources:

- set/get
- allocate/release
- activate/deactivate

Setting sets the range used by the resource. Allocation reserves the requested range that no other driver would be able to reserve it (and checking that no other driver reserved this range already). Activation makes the resource accessible to the driver doing whatever is necessary for that (for example, for memory it would be mapping into the kernel virtual address space).

The functions to manipulate resources are:

- `int bus_set_resource(device_t dev, int type, int rid, u_long start, u_long count)`

  Set a range for a resource. Returns 0 if successful, error code otherwise. Normally the only reason this function would return an error is value of type, rid, start or count out of permitted range.

  - dev - driver's device
  - type - type of resource, SYS_RES_*
  - rid - resource number (ID) within type
  - start, count - resource range

- `int bus_get_resource(device_t dev, int type, int rid, u_long *startp, u_long *countp)`

  Get the range of resource. Returns 0 if successful, error code if the resource is not defined yet.

- `u_long bus_get_resource_start(device_t dev, int type, int rid) u_long bus_get_resource_count (device_t dev, int type, int rid)`

  Convenience functions to get only the start or count. Return 0 in case of error, so if the resource start has 0 among the legitimate values it would be impossible to tell if the value is 0 or an error occurred. Luckily, no ISA resources for add-on drivers may have a start value equal 0.

- `void bus_delete_resource(device_t dev, int type, int rid)`

  Delete a resource, make it undefined.

- `struct resource * bus_alloc_resource(device_t dev, int type, int *rid, u_long start, u_long end, u_long count, u_int flags)`

Allocate a resource as a range of count values not allocated by anyone else, somewhere between start and end. Alas, alignment is not supported. If the resource was not set yet it's automatically created. The special values of start 0 and end ~0 (all ones) means that the fixed values previously set by `bus_set_resource()` must be used instead: start and count as themselves and end=(start+count), in this case if the resource was not defined before then an error is returned. Although rid is passed by reference it's not set anywhere by the resource allocation code of the ISA bus. (The other buses may use a different approach and modify it).

Flags are a bitmap, the flags interesting for the caller are:

- *RF_ACTIVE* - causes the resource to be automatically activated after allocation.
- *RF_SHAREABLE* - resource may be shared at the same time by multiple drivers.
- *RF_TIMESHARE* - resource may be time-shared by multiple drivers, i.e. allocated at the same time by many but activated only by one at any given moment of time.
- Returns 0 on error. The allocated values may be obtained from the returned handle using methods `rhand_*()`.
- `int bus_release_resource(device_t dev, int type, int rid, struct resource *r)`
- Release the resource, r is the handle returned by `bus_alloc_resource()`. Returns 0 on success, error code otherwise.
- `int bus_activate_resource(device_t dev, int type, int rid, struct resource *r)` `int bus_deactivate_resource(device_t dev, int type, int rid, struct resource *r)`
- Activate or deactivate resource. Return 0 on success, error code otherwise. If the resource is time-shared and currently activated by another driver then EBUSY is returned.
- `int bus_setup_intr(device_t dev, struct resource *r, int flags, driver_intr_t *handler, void *arg, void **cookiep)` `int bus_teardown_intr(device_t dev, struct resource *r, void *cookie)`
- Associate or de-associate the interrupt handler with a device. Return 0 on success, error code otherwise.
- r - the activated resource handler describing the IRQ

  flags - the interrupt priority level, one of:

  - `INTR_TYPE_TTY` - terminals and other likewise character-type devices. To mask them use `spltty()`.
  - `(INTR_TYPE_TTY | INTR_TYPE_FAST)` - terminal type devices with small input buffer, critical to the data loss on input (such as the old-fashioned serial ports). To mask them use `spltty()`.
  - `INTR_TYPE_BIO` - block-type devices, except those on the CAM controllers. To mask them use `splbio()`.
  - `INTR_TYPE_CAM` - CAM (Common Access Method) bus controllers. To mask them use `splcam()`.
  - `INTR_TYPE_NET` - network interface controllers. To mask them use `splimp()`.

- `INTR_TYPE_MISC` - miscellaneous devices. There is no other way to mask them than by `splhigh()` which masks all interrupts.

When an interrupt handler executes all the other interrupts matching its priority level will be masked. The only exception is the MISC level for which no other interrupts are masked and which is not masked by any other interrupt.

- *handler* - pointer to the handler function, the type driver_intr_t is defined as "void driver_intr_t(void *)"
- *arg* - the argument passed to the handler to identify this particular device. It is cast from void* to any real type by the handler. The old convention for the ISA interrupt handlers was to use the unit number as argument, the new (recommended) convention is using a pointer to the device softc structure.
- *cookie[p]* - the value received from `setup()` is used to identify the handler when passed to `teardown()`

A number of methods is defined to operate on the resource handlers (struct resource *). Those of interest to the device driver writers are:

- `u_long rman_get_start(r) u_long rman_get_end(r)` Get the start and end of allocated resource range.
- `void *rman_get_virtual(r)` Get the virtual address of activated memory resource.

19.6. Bus memory mapping

In many cases data is exchanged between the driver and the device through the memory. Two variants are possible:

(a) memory is located on the device card

(b) memory is the main memory of computer

In the case (a) the driver always copies the data back and forth between the on-card memory and the main memory as necessary. To map the on-card memory into the kernel virtual address space the physical address and length of the on-card memory must be defined as a SYS_RES_MEMORY resource. That resource can then be allocated and activated, and its virtual address obtained using `rman_get_virtual()`. The older drivers used the function `pmap_mapdev()` for this purpose, which should not be used directly any more. Now it’s one of the internal steps of resource activation.

Most of the ISA cards will have their memory configured for physical location somewhere in range 640KB-1MB. Some of the ISA cards require larger memory ranges which should be placed somewhere under 16MB (because of the 24-bit address limitation on the ISA bus). In that case if the machine has more memory than the start address of the device memory (in other words, they overlap) a memory hole must be configured at the address range used by devices. Many BIOSes allow to configure a memory hole of 1MB starting at 14MB or 15MB. FreeBSD can handle the memory holes properly if the BIOS reports them properly (old BIOSes may have this feature broken).

In the case (b) just the address of the data is sent to the device, and the device uses DMA to actually access the data in the main memory. Two limitations are present: First, ISA cards can only access memory below 16MB. Second, the contiguous pages in virtual address space may not be contiguous in physical address space, so the device may have to do scatter/gather operations. The bus subsystem provides ready solutions for some of these problems, the rest has to be done by the drivers themselves.

Two structures are used for DMA memory allocation, bus_dma_tag_t and bus_dmamap_t. Tag describes the properties required for the DMA memory. Map represents a memory block allocated according to these properties. Multiple maps may be associated with the same tag.

Tags are organized into a tree-like hierarchy with inheritance of the properties. A child tag inherits all the requirements of its parent tag or may make them more strict but never more loose.

Normally one top-level tag (with no parent) is created for each device unit. If multiple memory areas with different requirements are needed for each device then a tag for each of them may be created as a child of the parent tag.

The tags can be used to create a map in two ways.

First, a chunk of contiguous memory conformant with the tag requirements may be allocated (and later may be freed). This is normally used to allocate relatively long-living areas of memory for communication with the device. Loading of such memory into a map is trivial: it’s always considered as one chunk in the appropriate physical memory range.

Second, an arbitrary area of virtual memory may be loaded into a map. Each page of this memory will be checked for conformance to the map requirement. If it conforms then it’s left at it’s original location. If it is not then a fresh conformant "bounce page" is allocated and used as intermediate storage. When writing the data from the non-conformant original pages they will be copied to their bounce pages first and then transferred from the bounce pages to the device. When reading the data would go from the device to the bounce pages and then copied to their non-conformant original pages. The process of copying between the original and bounce pages is called synchronization. This is normally used on per-transfer basis: buffer for each transfer would be loaded, transfer done and buffer unloaded.

The functions working on the DMA memory are:

- `int bus_dma_tag_create(bus_dma_tag_t parent, bus_size_t alignment, bus_size_t boundary, bus_addr_t lowaddr, bus_addr_t highaddr, bus_dma_filter_t *filter, void *filterarg, bus_size_t maxsize, int nsegments, bus_size_t maxsegsz, int flags, bus_dma_tag_t *dmat)`

  Create a new tag. Returns 0 on success, the error code otherwise.

  - *parent* - parent tag, or NULL to create a top-level tag *alignment* - required physical alignment of the memory area to be allocated for this tag. Use value 1 for "no specific alignment". Applies only to the future `bus_dmamem_alloc()` but not `bus_dmamap_create()` calls. *boundary* - physical address boundary that must not be crossed when allocating the memory. Use value 0 for "no boundary". Applies only to the future `bus_dmamem_alloc()` but not `bus_dmamap_create()` calls. Must be power of 2. If the memory is planned to be used in non-cascaded DMA mode (i.e. the DMA addresses will be supplied not by the device itself but by the ISA DMA controller) then the boundary must be no larger than 64KB (64*1024) due to the limitations of the DMA hardware.

- *lowaddr, highaddr* - the names are slighlty misleading; these values are used to limit the permitted range of physical addresses used to allocate the memory. The exact meaning varies depending on the planned future use:
  - For `bus_dmamem_alloc()` all the addresses from 0 to lowaddr-1 are considered permitted, the higher ones are forbidden.
  - For `bus_dmamap_create()` all the addresses outside the inclusive range [lowaddr; highaddr] are considered accessible. The addresses of pages inside the range are passed to the filter function which decides if they are accessible. If no filter function is supplied then all the range is considered unaccessible.
  - For the ISA devices the normal values (with no filter function) are:

    lowaddr = BUS_SPACE_MAXADDR_24BIT

    highaddr = BUS_SPACE_MAXADDR
- *filter, filterarg* - the filter function and its argument. If NULL is passed for filter then the whole range [lowaddr, highaddr] is considered unaccessible when doing `bus_dmamap_create()`. Otherwise the physical address of each attempted page in range [lowaddr; highaddr] is passed to the filter function which decides if it is accessible. The prototype of the filter function is: `int filterfunc(void *arg, bus_addr_t paddr)` It must return 0 if the page is accessible, non-zero otherwise.
- *maxsize* - the maximal size of memory (in bytes) that may be allocated through this tag. In case it's difficult to estimate or could be arbitrarily big, the value for ISA devices would be BUS_SPACE_MAXSIZE_24BIT.
- *nsegments* - maximal number of scatter-gather segments supported by the device. If unrestricted then the value BUS_SPACE_UNRESTRICTED should be used. This value is recommended for the parent tags, the actual restrictions would then be specified for the descendant tags. Tags with nsegments equal to BUS_SPACE_UNRESTRICTED may not be used to actually load maps, they may be used only as parent tags. The practical limit for nsegments seems to be about 250-300, higher values will cause kernel stack overflow. But anyway the hardware normally can't support that many scatter-gather buffers.
- *maxsegsz* - maximal size of a scatter-gather segment supported by the device. The maximal value for ISA device would be BUS_SPACE_MAXSIZE_24BIT.
- *flags* - a bitmap of flags. The only interesting flags are:
  - *BUS_DMA_ALLOCNOW* - requests to allocate all the potentially needed bounce pages when creating the tag

    - *BUS_DMA_ISA* - mysterious flag used only on Alpha machines. It is not defined for the i386 machines. Probably it should be used by all the ISA drivers for Alpha machines but it looks like there are no such drivers yet.

  - *dmat* - pointer to the storage for the new tag to be returned

- `int bus_dma_tag_destroy(bus_dma_tag_t dmat)`

  Destroy a tag. Returns 0 on success, the error code otherwise.

  dmat - the tag to be destroyed

- `int bus_dmamem_alloc(bus_dma_tag_t dmat, void** vaddr, int flags, bus_dmamap_t *mapp)`

  Allocate an area of contiguous memory described by the tag. The size of memory to be allocated is tag's maxsize. Returns 0 on success, the error code otherwise. The result still has to be loaded by `bus_dmamap_load()` before used to get the physical address of the memory.

  - *dmat* - the tag
  - *vaddr* - pointer to the storage for the kernel virtual address of the allocated area to be returned.
  - flags - a bitmap of flags. The only interesting flag is:
    - *BUS_DMA_NOWAIT* - if the memory is not immediately available return the error. If this flag is not set then the routine is allowed to sleep waiting until the memory will become available.

  - *mapp* - pointer to the storage for the new map to be returned

- `void bus_dmamem_free(bus_dma_tag_t dmat, void *vaddr, bus_dmamap_t map)`

  Free the memory allocated by `bus_dmamem_alloc()`. As of now freeing of the memory allocated with ISA restrictions is not implemented. Because of this the recommended model of use is to keep and re-use the allocated areas for as long as possible. Do not lightly free some area and then shortly allocate it again. That does not mean that `bus_dmamem_free()` should not be used at all: hopefully it will be properly implemented soon.

  - *dmat* - the tag
  - *vaddr* - the kernel virtual address of the memory
  - *map* - the map of the memory (as returned from `bus_dmamem_alloc()`)

- `int bus_dmamap_create(bus_dma_tag_t dmat, int flags, bus_dmamap_t *mapp)`

  Create a map for the tag, to be used in `bus_dmamap_load()` later. Returns 0 on success, the error code otherwise.

  - *dmat* - the tag
  - *flags* - theoretically, a bit map of flags. But no flags are defined yet, so as of now it will be always 0.
  - *mapp* - pointer to the storage for the new map to be returned

- `int bus_dmamap_destroy(bus_dma_tag_t dmat, bus_dmamap_t map)`

  Destroy a map. Returns 0 on success, the error code otherwise.

  - dmat - the tag to which the map is associated
  - map - the map to be destroyed

- `int bus_dmamap_load(bus_dma_tag_t dmat, bus_dmamap_t map, void *buf, bus_size_t buflen, bus_dmamap_callback_t *callback, void *callback_arg, int flags)`

  Load a buffer into the map (the map must be previously created by `bus_dmamap_create()` or `bus_dmamem_alloc()`). All the pages of the buffer are checked for conformance to the tag requirements and for those not conformant the bounce pages are allocated. An array of physical segment descriptors is built and passed to the callback routine. This callback routine is then expected to handle it in some way. The number of bounce buffers in the system is limited, so if the bounce buffers are needed but not immediately available the request will be queued and the callback will be called when the bounce buffers will become available. Returns 0 if the callback was executed immediately or EINPROGRESS if the request was queued for future execution. In the latter case the synchronization with queued callback routine is the responsibility of the driver.

  - *dmat* - the tag
  - *map* - the map
  - *buf* - kernel virtual address of the buffer
  - *buflen* - length of the buffer
  - *callback*, `callback_arg` - the callback function and its argument

  The prototype of callback function is:

  `void callback(void *arg, bus_dma_segment_t *seg, int nseg, int error)`

- *arg* - the same as callback_arg passed to `bus_dmamap_load()`
- *seg* - array of the segment descriptors
- *nseg* - number of descriptors in array
- *error* - indication of the segment number overflow: if it's set to EFBIG then the buffer did not fit into the maximal number of segments permitted by the tag. In this case only the permitted number of descriptors will be in the array. Handling of this situation is up to the driver: depending on the desired semantics it can either consider this an error or split the buffer in two and handle the second part separately

Each entry in the segments array contains the fields:

- *ds_addr* - physical bus address of the segment
- *ds_len* - length of the segment

- `void bus_dmamap_unload(bus_dma_tag_t dmat, bus_dmamap_t map)`

  unload the map.

  - *dmat* - tag
  - *map* - loaded map

- `void bus_dmamap_sync (bus_dma_tag_t dmat, bus_dmamap_t map, bus_dmasync_op_t op)`

  Synchronise a loaded buffer with its bounce pages before and after physical transfer to or from device. This is the function that does all the necessary copying of data between the original buffer and its mapped version. The buffers must be synchronized both before and after doing the transfer.

  - *dmat* - tag
  - *map* - loaded map
  - *op* - type of synchronization operation to perform:

  - `BUS_DMASYNC_PREREAD` - before reading from device into buffer
  - `BUS_DMASYNC_POSTREAD` - after reading from device into buffer
  - `BUS_DMASYNC_PREWRITE` - before writing the buffer to device
  - `BUS_DMASYNC_POSTWRITE` - after writing the buffer to device

As of now PREREAD and POSTWRITE are null operations but that may change in the future, so they must not be ignored in the driver. Synchronization is not needed for the memory obtained from `bus_dmamem_alloc()`.

Before calling the callback function from `bus_dmamap_load()` the segment array is stored in the stack. And it gets pre-allocated for the maximal number of segments allowed by the tag. Because of this the practical limit for the number of segments on i386 architecture is about 250-300 (the kernel stack is 4KB minus the size of the user structure, size of a segment array entry is 8 bytes, and some space must be left). Because the array is allocated based on the maximal number this value must not be set higher than really needed. Fortunately, for most of hardware the maximal supported number of segments is much lower. But if the driver wants to handle buffers with a very large number of scatter-gather segments it should do that in portions: load part of the buffer, transfer it to the device, load next part of the buffer, and so on.

Another practical consequence is that the number of segments may limit the size of the buffer. If all the pages in the buffer happen to be physically non-contiguous then the maximal supported buffer size for that fragmented case would be (nsegments * page_size). For example, if a maximal number of 10 segments is supported then on i386 maximal guaranteed supported buffer size would be 40K. If a higher size is desired then special tricks should be used in the driver.

If the hardware does not support scatter-gather at all or the driver wants to support some buffer size even if it's heavily fragmented then the solution is to allocate a contiguous buffer in the driver and use it as intermediate storage if the original buffer does not fit.

Below are the typical call sequences when using a map depend on the use of the map. The characters -> are used to show the flow of time.

For a buffer which stays practically fixed during all the time between attachment and detachment of a device:

bus_dmamem_alloc -> bus_dmamap_load -> ...use buffer... -> -> bus_dmamap_unload -> bus_dmamem_free

For a buffer that changes frequently and is passed from outside the driver:

```
bus_dmamap_create ->
-> bus_dmamap_load -> bus_dmamap_sync(PRE...) -> do transfer ->
-> bus_dmamap_sync(POST...) -> bus_dmamap_unload ->
...
-> bus_dmamap_load -> bus_dmamap_sync(PRE...) -> do transfer ->
-> bus_dmamap_sync(POST...) -> bus_dmamap_unload ->
-> bus_dmamap_destroy
```

When loading a map created by `bus_dmamem_alloc()` the passed address and size of the buffer must be the same as used in `bus_dmamem_alloc()`. In this case it is guaranteed that the whole buffer will be mapped as one segment (so the callback may be based on this assumption) and the request will be executed immediately (EINPROGRESS will never be returned). All the callback needs to do in this case is to save the physical address.

A typical example would be:

```
  static void
alloc_callback(void *arg, bus_dma_segment_t *seg, int nseg, int error)
{
  *(bus_addr_t *)arg = seg[0].ds_addr;
}

  ...
  int error;
  struct somedata {
    ....
  };
  struct somedata *vsomedata; /* virtual address */
  bus_addr_t psomedata; /* physical bus-relative address */
  bus_dma_tag_t tag_somedata;
  bus_dmamap_t map_somedata;
  ...

  error=bus_dma_tag_create(parent_tag, alignment,
   boundary, lowaddr, highaddr, /*filter*/ NULL, /*filterarg*/ NULL,
   /*maxsize*/ sizeof(struct somedata), /*nsegments*/ 1,
   /*maxsegsz*/ sizeof(struct somedata), /*flags*/ 0,
   &tag_somedata);
  if(error)
  return error;

  error = bus_dmamem_alloc(tag_somedata, &vsomedata, /* flags*/ 0,
     &map_somedata);
  if(error)
     return error;

  bus_dmamap_load(tag_somedata, map_somedata, (void *)vsomedata,
     sizeof (struct somedata), alloc_callback,
     (void *) &psomedata, /*flags*/0);
```

Looks a bit long and complicated but that’s the way to do it. The practical consequence is: if multiple memory areas are allocated always together it would be a really good idea to combine them all into one structure and allocate as one (if the alignment and boundary limitations permit).

When loading an arbitrary buffer into the map created by `bus_dmamap_create()` special measures must be taken to synchronize with the callback in case it would be delayed. The code would look like:

```
  {
   int s;
   int error;
```

```
         s = splsoftvm();
         error = bus_dmamap_load(
             dmat,
             dmamap,
             buffer_ptr,
             buffer_len,
             callback,
             /*callback_arg*/ buffer_descriptor,
             /*flags*/0);
         if (error == EINPROGRESS) {
             /*
              * Do whatever is needed to ensure synchronization
              * with callback. Callback is guaranteed not to be started
              * until we do splx() or tsleep().
              */
            }
         splx(s);
        }
```

Two possible approaches for the processing of requests are:

1. If requests are completed by marking them explicitly as done (such as the CAM requests) then it would be simpler to put all the further processing into the callback driver which would mark the request when it's done. Then not much extra synchronization is needed. For the flow control reasons it may be a good idea to freeze the request queue until this request gets completed.

2. If requests are completed when the function returns (such as classic read or write requests on character devices) then a synchronization flag should be set in the buffer descriptor and `tsleep()` called. Later when the callback gets called it will do it's processing and check this synchronization flag. If it's set then the callback should issue a wakeup. In this approach the callback function could either do all the needed processing (just like the previous case) or simply save the segments array in the buffer descriptor. Then after callback completes the calling function could use this saved segments array and do all the processing.

19.7. DMA

The Direct Memory Access (DMA) is implemented in the ISA bus through the DMA controller (actually, two of them but that's an irrelevant detail). To make the early ISA devices simple and cheap the logic of the bus control and address generation was concentrated in the DMA controller. Fortunately, FreeBSD provides a set of functions that mostly hide the annoying details of the DMA controller from the device drivers.

The simplest case is for the fairly intelligent devices. Like the bus master devices on PCI they can generate the bus cycles and memory addresses all by themselves. The only thing they really need from the DMA controller is bus arbitration. So for this purpose they pretend to be cascaded slave DMA controllers. And the only thing needed from the system DMA controller is to enable the cascaded mode on a DMA channel by calling the following function when attaching the driver:

```
void isa_dmacascade(int channel_number)
```

All the further activity is done by programming the device. When detaching the driver no DMA-related functions need to be called.

For the simpler devices things get more complicated. The functions used are:

- `int isa_dma_acquire(int chanel_number)`

  Reserve a DMA channel. Returns 0 on success or EBUSY if the channel was already reserved by this or a different driver. Most of the ISA devices are not able to share DMA channels anyway, so normally this function is called when attaching a device. This reservation was made redundant by the modern interface of bus resources but still must be used in addition to the latter. If not used then later, other DMA routines will panic.

- `int isa_dma_release(int chanel_number)`

  Release a previously reserved DMA channel. No transfers must be in progress when the channel is released (as well as the device must not try to initiate transfer after the channel is released).

- `void isa_dmainit(int chan, u_int bouncebufsize)`

  Allocate a bounce buffer for use with the specified channel. The requested size of the buffer can't exceed 64KB. This bounce buffer will be automatically used later if a transfer buffer happens to be not physically contiguous or outside of the memory accessible by the ISA bus or crossing the 64KB boundary. If the transfers will be always done from buffers which conform to these conditions (such as those allocated by `bus_dmamem_alloc()` with proper limitations) then `isa_dmainit()` does not have to be called. But it's quite convenient to transfer arbitrary data using the DMA controller. The bounce buffer will automatically care of the scatter-gather issues.

  - *chan* - channel number
  - *bouncebufsize* - size of the bounce buffer in bytes

- `void isa_dmastart(int flags, caddr_t addr, u_int nbytes, int chan)`

  Prepare to start a DMA transfer. This function must be called to set up the DMA controller before actually starting transfer on the device. It checks that the buffer is contiguous and falls into the ISA memory range, if not then the bounce buffer is automatically used. If bounce buffer is required but not set up by `isa_dmainit()` or too small for the requested transfer size then the system will panic. In case of a write request with bounce buffer the data will be automatically copied to the bounce buffer.

- flags - a bitmask determining the type of operation to be done. The direction bits B_READ and B_WRITE are mutually exclusive.
  - B_READ - read from the ISA bus into memory
  - B_WRITE - write from the memory to the ISA bus
  - B_RAW - if set then the DMA controller will remember the buffer and after the end of transfer will automatically re-initialize itself to repeat transfer of the same buffer again (of course, the driver may change the data in the buffer before initiating another transfer in the device). If not set then the parameters will work only for one transfer, and `isa_dmastart()` will have to be called again before initiating the next transfer. Using B_RAW makes sense only if the bounce buffer is not used.
- addr - virtual address of the buffer
- nbytes - length of the buffer. Must be less or equal to 64KB. Length of 0 is not allowed: the DMA controller will understand it as 64KB while the kernel code will understand it as 0 and that would cause unpredictable effects. For channels number 4 and higher the length must be even because these channels transfer 2 bytes at a time. In case of an odd length the last byte will not be transferred.
- chan - channel number
- `void isa_dmadone(int flags, caddr_t addr, int nbytes, int chan)`

  Synchronize the memory after device reports that transfer is done. If that was a read operation with a bounce buffer then the data will be copied from the bounce buffer to the original buffer. Arguments are the same as for `isa_dmastart()`. Flag B_RAW is permitted but it does not affect `isa_dmadone()` in any way.
- `int isa_dmastatus(int channel_number)`

  Returns the number of bytes left in the current transfer to be transferred. In case the flag B_READ was set in `isa_dmastart()` the number returned will never be equal to zero. At the end of transfer it will be automatically reset back to the length of buffer. The normal use is to check the number of bytes left after the device signals that the transfer is completed. If the number of bytes is not 0 then probably something went wrong with that transfer.
- `int isa_dmastop(int channel_number)`

  Aborts the current transfer and returns the number of bytes left untransferred.

19.8. xxx_isa_probe

This function probes if a device is present. If the driver supports auto-detection of some part of device configuration (such as interrupt vector or memory address) this auto-detection must be done in this routine.

As for any other bus, if the device can not be detected or is detected but failed the self-test or some other problem happened then it returns a positive value of error. The value ENXIO must be returned if the device is not present. Other error values may mean other conditions. Zero or negative values mean success. Most of the drivers return zero as success.

The negative return values are used when a PnP device supports multiple interfaces. For example, an older compatibility interface and a newer advanced interface which are supported by different drivers. Then both drivers would detect the device. The driver which returns a higher value in the probe routine takes precedence (in other words, the driver returning 0 has highest precedence, one returning -1 is next, one returning -2 is after it and so on). In result the devices which support only the old interface will be handled by the old driver (which should return -1 from the probe routine) while the devices supporting the new interface as well will be handled by the new driver (which should return 0 from the probe routine).

The device descriptor struct xxx_softc is allocated by the system before calling the probe routine. If the probe routine returns an error the descriptor will be automatically deallocated by the system. So if a probing error occurs the driver must make sure that all the resources it used during probe are deallocated and that nothing keeps the descriptor from being safely deallocated. If the probe completes successfully the descriptor will be preserved by the system and later passed to the routine `xxx_isa_attach()`. If a driver returns a negative value it can't be sure that it will have the highest priority and its attach routine will be called. So in this case it also must release all the resources before returning and if necessary allocate them again in the attach routine. When `xxx_isa_probe()` returns 0 releasing the resources before returning is also a good idea, a well-behaved driver should do so. But in case if there is some problem with releasing the resources the driver is allowed to keep resources between returning 0 from the probe routine and execution of the attach routine.

A typical probe routine starts with getting the device descriptor and unit:

```
struct xxx_softc *sc = device_get_softc(dev);
 int unit = device_get_unit(dev);
 int pnperror;
 int error = 0;

 sc->dev = dev; /* link it back */
 sc->unit = unit;
```

Then check for the PnP devices. The check is carried out by a table containing the list of PnP IDs supported by this driver and human-readable descriptions of the device models corresponding to these IDs.

```
pnperror=ISA_PNP_PROBE(device_get_parent(dev), dev,
xxx_pnp_ids); if(pnperror == ENXIO) return ENXIO;
```

The logic of ISA_PNP_PROBE is the following: If this card (device unit) was not detected as PnP then ENOENT will be returned. If it was detected as PnP but its detected ID does not match any of the IDs in the table then ENXIO is returned. Finally, if it has PnP support and it matches on of the IDs in the table, 0 is returned and the appropriate description from the table is set by `device_set_desc()`.

If a driver supports only PnP devices then the condition would look like:

```
if(pnperror != 0)
    return pnperror;
```

No special treatment is required for the drivers which don't support PnP because they pass an empty PnP ID table and will always get ENXIO if called on a PnP card.

The probe routine normally needs at least some minimal set of resources, such as I/O port number to find the card and probe it. Depending on the hardware the driver may be able to discover the other necessary resources automatically. The PnP devices have all the resources pre-set by the PnP subsystem, so the driver does not need to discover them by itself.

Typically the minimal information required to get access to the device is the I/O port number. Then some devices allow to get the rest of information from the device configuration registers (though not all devices do that). So first we try to get the port start value:

```
sc->port0 = bus_get_resource_start(dev,
       SYS_RES_IOPORT, 0 /*rid*/); if(sc->port0 == 0) return ENXIO;
```

The base port address is saved in the structure softc for future use. If it will be used very often then calling the resource function each time would be prohibitively slow. If we don't get a port we just return an error. Some device drivers can instead be clever and try to probe all the possible ports, like this:

```
/* table of all possible base I/O port addresses for this device */
static struct xxx_allports {
    u_short port; /* port address */
    short used; /* flag: if this port is already used by some unit */
} xxx_allports = {
    { 0x300, 0 },
    { 0x320, 0 },
    { 0x340, 0 },
    { 0, 0 } /* end of table */
};

...
int port, i;
...
```

```
    port =  bus_get_resource_start(dev, SYS_RES_IOPORT, 0 /*rid*/);
    if(port !=0 ) {
        for(i=0; xxx_allports[i].port!=0; i++) {
            if(xxx_allports[i].used || xxx_allports[i].port != port)
                continue;

            /* found it */
            xxx_allports[i].used = 1;
            /* do probe on a known port */
            return xxx_really_probe(dev, port);
        }
        return ENXIO; /* port is unknown or already used */
    }

    /* we get here only if we need to guess the port */
    for(i=0; xxx_allports[i].port!=0; i++) {
        if(xxx_allports[i].used)
            continue;

        /* mark as used - even if we find nothing at this port
         * at least we won't probe it in future
         */
         xxx_allports[i].used = 1;

        error = xxx_really_probe(dev, xxx_allports[i].port);
        if(error == 0) /* found a device at that port */
            return 0;
    }
    /* probed all possible addresses, none worked */
    return ENXIO;
```

Of course, normally the driver's `identify()` routine should be used for such things. But there may be one valid reason why it may be better to be done in `probe()`: if this probe would drive some other sensitive device crazy. The probe routines are ordered with consideration of the "sensitive" flag: the sensitive devices get probed first and the rest of devices later. But the `identify()` routines are called before any probes, so they show no respect to the sensitive devices and may upset them.

Now, after we got the starting port we need to set the port count (except for PnP devices) because the kernel does not have this information in the configuration file.

```
if(pnperror /* only for non-PnP devices */
&& bus_set_resource(dev, SYS_RES_IOPORT, 0, sc->port0,
XXX_PORT_COUNT)<0)
```

```
            return ENXIO;
```

Finally allocate and activate a piece of port address space (special values of start and end mean "use those we set by `bus_set_resource()`"):

```
        sc->port0_rid = 0;
        sc->port0_r = bus_alloc_resource(dev, SYS_RES_IOPORT,
        &sc->port0_rid,
            /*start*/ 0, /*end*/ ~0, /*count*/ 0, RF_ACTIVE);

        if(sc->port0_r == NULL)
            return ENXIO;
```

Now having access to the port-mapped registers we can poke the device in some way and check if it reacts like it is expected to. If it does not then there is probably some other device or no device at all at this address.

Normally drivers don't set up the interrupt handlers until the attach routine. Instead they do probes in the polling mode using the `DELAY()` function for timeout. The probe routine must never hang forever, all the waits for the device must be done with timeouts. If the device does not respond within the time it's probably broken or misconfigured and the driver must return error. When determining the timeout interval give the device some extra time to be on the safe side: although `DELAY()` is supposed to delay for the same amount of time on any machine it has some margin of error, depending on the exact CPU.

If the probe routine really wants to check that the interrupts really work it may configure and probe the interrupts too. But that's not recommended.

```
        /* implemented in some very device-specific way */
        if(error = xxx_probe_ports(sc))
            goto bad; /* will deallocate the resources before returning */
```

The fucntion `xxx_probe_ports()` may also set the device description depending on the exact model of device it discovers. But if there is only one supported device model this can be as well done in a hardcoded way. Of course, for the PnP devices the PnP support sets the description from the table automatically.

```
        if(pnperror)
            device_set_desc(dev, "Our device model 1234");
```

Then the probe routine should either discover the ranges of all the resources by reading the device configuration registers or make sure that they were set explicitly by the user. We will consider it with an example of on-board memory. The probe routine should be as non-intrusive as possible, so allocation and check of functionality of the rest of resources (besides the ports) would be better left to the attach routine.

The memory address may be specified in the kernel configuration file or on some devices it may be pre-configured in non-volatile configuration registers. If both sources are available and different, which one should be used? Probably if the user bothered to set the address explicitly in the kernel configuration file they know what they’re doing and this one should take precedence. An example of implementation could be:

```
/* try to find out the config address first */
sc->mem0_p = bus_get_resource_start(dev, SYS_RES_MEMORY, 0 /*rid*/);
if(sc->mem0_p == 0) { /* nope, not specified by user */
    sc->mem0_p = xxx_read_mem0_from_device_config(sc);


if(sc->mem0_p == 0)
        /* can’t get it from device config registers either */
        goto bad;
} else {
    if(xxx_set_mem0_address_on_device(sc) < 0)
        goto bad; /* device does not support that address */
}

/* just like the port, set the memory size,
 * for some devices the memory size would not be constant
 * but should be read from the device configuration registers instead
 * to accommodate different models of devices. Another option would
 * be to let the user set the memory size as "msize" configuration
 * resource which will be automatically handled by the ISA bus.
 */
 if(pnperror) { /* only for non-PnP devices */
    sc->mem0_size = bus_get_resource_count(dev, SYS_RES_MEMORY, 0 /*rid*/);
    if(sc->mem0_size == 0) /* not specified by user */
        sc->mem0_size = xxx_read_mem0_size_from_device_config(sc);

    if(sc->mem0_size == 0) {
        /* suppose this is a very old model of device without
         * auto-configuration features and the user gave no preference,
         * so assume the minimalistic case
         * (of course, the real value will vary with the driver)
         */
        sc->mem0_size = 8*1024;
    }

    if(xxx_set_mem0_size_on_device(sc) < 0)
        goto bad; /* device does not support that size */

    if(bus_set_resource(dev, SYS_RES_MEMORY, /*rid*/0,
            sc->mem0_p, sc->mem0_size)<0)
        goto bad;
} else {
    sc->mem0_size = bus_get_resource_count(dev, SYS_RES_MEMORY, 0 /*rid*/);
}
```

Resources for IRQ and DRQ are easy to check by analogy.

If all went well then release all the resources and return success.

```
        xxx_free_resources(sc);
        return 0;
```

Finally, handle the troublesome situations. All the resources should be deallocated before returning. We make use of the fact that before the structure softc is passed to us it gets zeroed out, so we can find out if some resource was allocated: then its descriptor is non-zero.

```
        bad:

        xxx_free_resources(sc);
        if(error)
              return error;
        else /* exact error is unknown */
            return ENXIO;
```

That would be all for the probe routine. Freeing of resources is done from multiple places, so it’s moved to a function which may look like:

```
static void
           xxx_free_resources(sc)
              struct xxx_softc *sc;
          {
              /* check every resource and free if not zero */

              /* interrupt handler */
              if(sc->intr_r) {
                  bus_teardown_intr(sc->dev, sc->intr_r, sc->intr_cookie);
                  bus_release_resource(sc->dev, SYS_RES_IRQ, sc->intr_rid,
                      sc->intr_r);
                  sc->intr_r = 0;
              }

              /* all kinds of memory maps we could have allocated */
              if(sc->data_p) {
                  bus_dmamap_unload(sc->data_tag, sc->data_map);
```

```
            sc->data_p = 0;
        }
         if(sc->data) { /* sc->data_map may be legitimately equal to 0 */
            /* the map will also be freed */
            bus_dmamem_free(sc->data_tag, sc->data, sc->data_map);
            sc->data = 0;
        }
        if(sc->data_tag) {
            bus_dma_tag_destroy(sc->data_tag);
            sc->data_tag = 0;
        }

        ... free other maps and tags if we have them ...

        if(sc->parent_tag) {
            bus_dma_tag_destroy(sc->parent_tag);
            sc->parent_tag = 0;
        }

        /* release all the bus resources */
        if(sc->mem0_r) {
            bus_release_resource(sc->dev, SYS_RES_MEMORY, sc->mem0_rid,
                sc->mem0_r);
            sc->mem0_r = 0;
        }
        ...
        if(sc->port0_r) {
            bus_release_resource(sc->dev, SYS_RES_IOPORT, sc->port0_rid,
                sc->port0_r);
            sc->port0_r = 0;
        }
    }
```

19.9. xxx_isa_attach

The attach routine actually connects the driver to the system if the probe routine returned success and the system had chosen to attach that driver. If the probe routine returned 0 then the attach routine may expect to receive the device structure softc intact, as it was set by the probe routine. Also if the probe routine returns 0 it may expect that the attach routine for this device shall be called at some point in the future. If the probe routine returns a negative value then the driver may make none of these assumptions.

The attach routine returns 0 if it completed successfully or error code otherwise.

The attach routine starts just like the probe routine, with getting some frequently used data into more accessible variables.

```
              struct xxx_softc *sc = device_get_softc(dev);
              int unit = device_get_unit(dev);
              int error = 0;
```

Then allocate and activate all the necessary resources. Because normally the port range will be released before returning from probe, it has to be allocated again. We expect that the probe routine had properly set all the resource ranges, as well as saved them in the structure softc. If the probe routine had left some resource allocated then it does not need to be allocated again (which would be considered an error).

```
              sc->port0_rid = 0;
              sc->port0_r = bus_alloc_resource(dev, SYS_RES_IOPORT,  &sc->port0_rid,
                  /*start*/ 0, /*end*/ ~0, /*count*/ 0, RF_ACTIVE);

              if(sc->port0_r == NULL)
                   return ENXIO;

              /* on-board memory */
              sc->mem0_rid = 0;
              sc->mem0_r = bus_alloc_resource(dev, SYS_RES_MEMORY,  &sc->mem0_rid,
                  /*start*/ 0, /*end*/ ~0, /*count*/ 0, RF_ACTIVE);

              if(sc->mem0_r == NULL)
                    goto bad;

              /* get its virtual address */
              sc->mem0_v = rman_get_virtual(sc->mem0_r);
```

The DMA request channel (DRQ) is allocated likewise. To initialize it use functions of the `isa_dma*()` family. For example:

`isa_dmacascade(sc->drq0);`

The interrupt request line (IRQ) is a bit special. Besides allocation the driver's interrupt handler should be associated with it. Historically in the old ISA drivers the argument passed by the system to the interrupt handler was the device unit number. But in modern drivers the convention suggests passing the pointer to structure softc. The important reason is that when the structures softc are allocated dynamically then getting the unit number from softc is easy while getting softc from unit number is difficult. Also this convention makes the drivers for different buses look more uniform and allows them to share the code: each bus gets its own probe, attach, detach and other bus-specific routines while the bulk of the driver code may be shared among them.

```
              sc->intr_rid = 0;
              sc->intr_r = bus_alloc_resource(dev, SYS_RES_MEMORY,  &sc->intr_rid,
                    /*start*/ 0, /*end*/ ~0, /*count*/ 0, RF_ACTIVE);

              if(sc->intr_r == NULL)
                  goto bad;

              /*
               * XXX_INTR_TYPE is supposed to be defined depending on the type of
               * the driver, for example as INTR_TYPE_CAM for a CAM driver
               */
              error = bus_setup_intr(dev, sc->intr_r, XXX_INTR_TYPE,
                  (driver_intr_t *) xxx_intr, (void *) sc, &sc->intr_cookie);
              if(error)
                  goto bad;
```

If the device needs to make DMA to the main memory then this memory should be allocated like described before:

```
error=bus_dma_tag_create(NULL, /*alignment*/ 4,
    /*boundary*/ 0, /*lowaddr*/ BUS_SPACE_MAXADDR_24BIT,
    /*highaddr*/ BUS_SPACE_MAXADDR, /*filter*/ NULL, /*filterarg*/ NULL,
    /*maxsize*/ BUS_SPACE_MAXSIZE_24BIT,
    /*nsegments*/ BUS_SPACE_UNRESTRICTED,
    /*maxsegsz*/ BUS_SPACE_MAXSIZE_24BIT, /*flags*/ 0,
    &sc->parent_tag);
if(error)
    goto bad;

/* many things get inherited from the parent tag
 * sc->data is supposed to point to the structure with the shared data,
 * for example for a ring buffer it could be:
 * struct {
 *   u_short rd_pos;
 *   u_short wr_pos;
 *   char    bf[XXX_RING_BUFFER_SIZE]
 * } *data;
 */
error=bus_dma_tag_create(sc->parent_tag, 1,
    0, BUS_SPACE_MAXADDR, 0, /*filter*/ NULL, /*filterarg*/ NULL,
    /*maxsize*/ sizeof(* sc->data), /*nsegments*/ 1,
    /*maxsegsz*/ sizeof(* sc->data), /*flags*/ 0,
    &sc->data_tag);
if(error)
    goto bad;
error = bus_dmamem_alloc(sc->data_tag, &sc->data, /* flags*/ 0,
    &sc->data_map);
if(error)
     goto bad;

/* xxx_alloc_callback() just saves the physical address at
 * the pointer passed as its argument, in this case &sc->data_p.
 * See details in the section on bus memory mapping.
 * It can be implemented like:
 *
 * static void
 * xxx_alloc_callback(void *arg, bus_dma_segment_t *seg,
 *     int nseg, int error)
 * {
 *    *(bus_addr_t *)arg = seg[0].ds_addr;
 * }
 */
bus_dmamap_load(sc->data_tag, sc->data_map, (void *)sc->data,
    sizeof (* sc->data), xxx_alloc_callback, (void *) &sc->data_p,
    /*flags*/0);
```

After all the necessary resources are allocated the device should be initialized. The initialization may include testing that all the expected features are functional.

```
    if(xxx_initialize(sc) < 0)
         goto bad;
```

The bus subsystem will automatically print on the console the device description set by probe. But if the driver wants to print some extra information about the device it may do so, for example:

```
  device_printf(dev, "has on-card FIFO buffer of %d bytes\n", sc->fifosize);
```

If the initialization routine experiences any problems then printing messages about them before returning error is also recommended.

The final step of the attach routine is attaching the device to its functional subsystem in the kernel. The exact way to do it depends on the type of the driver: a character device, a block device, a network device, a CAM SCSI bus device and so on.

If all went well then return success.

```
    error = xxx_attach_subsystem(sc);
    if(error)
        goto bad;
```

```
return 0;
```

Finally, handle the troublesome situations. All the resources should be deallocated before returning an error. We make use of the fact that before the structure softc is passed to us it gets zeroed out, so we can find out if some resource was allocated: then its descriptor is non-zero.

```
bad:

xxx_free_resources(sc);
if(error)
    return error;
else /* exact error is unknown */
    return ENXIO;
```

That would be all for the attach routine.

19.10. xxx_isa_detach

If this function is present in the driver and the driver is compiled as a loadable module then the driver gets the ability to be unloaded. This is an important feature if the hardware supports hot plug. But the ISA bus does not support hot plug, so this feature is not particularly important for the ISA devices. The ability to unload a driver may be useful when debugging it, but in many cases installation of the new version of the driver would be required only after the old version somehow wedges the system and reboot will be needed anyway, so the efforts spent on writing the detach routine may not be worth it. Another argument is that unloading would allow upgrading the drivers on a production machine seems to be mostly theoretical. Installing a new version of a driver is a dangerous operation which should never be performed on a production machine (and which is not permitted when the system is running in secure mode). Still the detach routine may be provided for the sake of completeness.

The detach routine returns 0 if the driver was successfully detached or the error code otherwise.

The logic of detach is a mirror of the attach. The first thing to do is to detach the driver from its kernel subsystem. If the device is currently open then the driver has two choices: refuse to be detached or forcibly close and proceed with detach. The choice used depends on the ability of the particular kernel subsystem to do a forced close and on the preferences of the driver’s author. Generally the forced close seems to be the preferred alternative.

```
struct xxx_softc *sc = device_get_softc(dev);
int error;

error = xxx_detach_subsystem(sc);
if(error)
    return error;
```

Next the driver may want to reset the hardware to some consistent state. That includes stopping any ongoing transfers, disabling the DMA channels and interrupts to avoid memory corruption by the device. For most of the drivers this is exactly what the shutdown routine does, so if it is included in the driver we can as well just call it.

```
xxx_isa_shutdown(dev);
```

And finally release all the resources and return success.

```
xxx_free_resources(sc);
return 0;
```

19.11. xxx_isa_shutdown

This routine is called when the system is about to be shut down. It is expected to bring the hardware to some consistent state. For most of the ISA devices no special action is required, so the function is not really necessary because the device will be re-initialized on reboot anyway. But some devices have to be shut down with a special procedure, to make sure that they will be properly detected after soft reboot (this is especially true for many devices with proprietary identification protocols). In any case disabling DMA and interrupts in the device registers and stopping any ongoing transfers is a good idea. The exact action depends on the hardware, so we don't consider it here in any details.

xxx_intr

The interrupt handler is called when an interrupt is received which may be from this particular device. The ISA bus does not support interrupt sharing (except some special cases) so in practice if the interrupt handler is called then the interrupt almost for sure came from its device. Still the interrupt handler must poll the device registers and make sure that the interrupt was generated by its device. If not it should just return.

The old convention for the ISA drivers was getting the device unit number as an argument. It is obsolete, and the new drivers receive whatever argument was specified for them in the attach routine when calling `bus_setup_intr()`. By the new convention it should be the pointer to the structure softc. So the interrupt handler commonly starts as:

```
static void
xxx_intr(struct xxx_softc *sc)
{
```

It runs at the interrupt priority level specified by the interrupt type parameter of `bus_setup_intr()`. That means that all the other interrupts of the same type as well as all the software interrupts are disabled.

To avoid races it is commonly written as a loop:

```
while(xxx_interrupt_pending(sc)) {
    xxx_process_interrupt(sc);
    xxx_acknowledge_interrupt(sc);
}
```

The interrupt handler has to acknowledge interrupt to the device only but not to the interrupt controller, the system takes care of the latter.

Chapter 20. PCI Devices

This chapter will talk about the FreeBSD mechanisms for writing a device driver for a device on a PCI bus.

20.1. Probe and Attach

Information here about how the PCI bus code iterates through the unattached devices and see if a newly loaded kld will attach to any of them.

```
/*
 * Simple KLD to play with the PCI functions.
 *
 * Murray Stokely
 */

#define MIN(a,b) (((a) < (b)) ? (a) : (b))

#include <sys/types.h>
#include <sys/module.h>
#include <sys/systm.h>  /* uprintf */
#include <sys/errno.h>
#include <sys/param.h>  /* defines used in kernel.h */
#include <sys/kernel.h> /* types used in module initialization */
#include <sys/conf.h>   /* cdevsw struct */
#include <sys/uio.h>    /* uio struct */
#include <sys/malloc.h>
#include <sys/bus.h> /* structs, prototypes for pci bus stuff */

#include <pci/pcivar.h> /* For get_pci macros! */

/* Function prototypes */
d_open_t      mypci_open;
d_close_t     mypci_close;
d_read_t      mypci_read;
d_write_t     mypci_write;

/* Character device entry points */

static struct cdevsw mypci_cdevsw = {
  mypci_open,
  mypci_close,
  mypci_read,
  mypci_write,
  noioctl,
  nopoll,
  nommap,
  nostrategy,
  "mypci",
  36,                   /* reserved for lkms - /usr/src/sys/conf/majors */
  nodump,
  nopsize,
  D_TTY,
  -1
};

/* vars */
static dev_t sdev;

/* We’re more interested in probe/attach than with
   open/close/read/write at this point */

int
mypci_open(dev_t dev, int oflags, int devtype, struct proc *p)
{
  int err = 0;

  uprintf("Opened device \"mypci\" successfully.\n");
  return(err);
}

int
mypci_close(dev_t dev, int fflag, int devtype, struct proc *p)
{
  int err=0;

  uprintf("Closing device \"mypci.\"\n");
  return(err);
}

int
mypci_read(dev_t dev, struct uio *uio, int ioflag)
{
  int err = 0;

  uprintf("mypci read!\n");
  return err;
}
```

```
int
mypci_write(dev_t dev, struct uio *uio, int ioflag)
{
  int err = 0;

  uprintf("mypci write!\n");
  return(err);
}

/* PCI Support Functions */

/*
 * Return identification string if this is device is ours.
 */
static int
mypci_probe(device_t dev)
{
  uprintf("MyPCI Probe\n"
   "Vendor ID : 0x%x\n"
   "Device ID : 0x%x\n",pci_get_vendor(dev),pci_get_device(dev));

  if (pci_get_vendor(dev) == 0x11c1) {
    uprintf("We’ve got the Winmodem, probe successful!\n");
    return 0;
  }

  return ENXIO;
}

/* Attach function is only called if the probe is successful */

static int
mypci_attach(device_t dev)
{
  uprintf("MyPCI Attach for : deviceID : 0x%x\n",pci_get_vendor(dev));
  sdev = make_dev(&mypci_cdevsw,
   0,
   UID_ROOT,
   GID_WHEEL,
   0600,
   "mypci");
  uprintf("Mypci device loaded.\n");
  return ENXIO;
}
```

```
/* Detach device. */

static int
mypci_detach(device_t dev)
{
  uprintf("Mypci detach!\n");
  return 0;
}

/* Called during system shutdown after sync. */

static int
mypci_shutdown(device_t dev)
{
  uprintf("Mypci shutdown!\n");
  return 0;
}

/*
 * Device suspend routine.
 */
static int
mypci_suspend(device_t dev)
{
  uprintf("Mypci suspend!\n");
  return 0;
}

/*
 * Device resume routine.
 */

static int
mypci_resume(device_t dev)
{
  uprintf("Mypci resume!\n");
  return 0;
}

static device_method_t mypci_methods[] = {
 /* Device interface */
 DEVMETHOD(device_probe, mypci_probe),
 DEVMETHOD(device_attach, mypci_attach),
 DEVMETHOD(device_detach, mypci_detach),
 DEVMETHOD(device_shutdown, mypci_shutdown),
 DEVMETHOD(device_suspend, mypci_suspend),
 DEVMETHOD(device_resume, mypci_resume),

 { 0, 0 }
};

static driver_t mypci_driver = {
 "mypci",
 mypci_methods,
 0,
 /* sizeof(struct mypci_softc), */
};

static devclass_t mypci_devclass;

DRIVER_MODULE(mypci, pci, mypci_driver, mypci_devclass, 0, 0);
```

Additional Resources

- PCI Special Interest Group (http://www.pcisig.org)
- PCI System Architecture, Fourth Edition by Tom Shanley, et al.

20.2. Bus Resources

FreeBSD provides an object-oriented mechanism for requesting resources from a parent bus. Almost all devices will be a child member of some sort of bus (PCI, ISA, USB, SCSI, etc) and these devices need to acquire resources from their parent bus (such as memory segments, interrupt lines, or DMA channels).

20.2.1. Base Address Registers

To do anything particularly useful with a PCI device you will need to obtain the *Base Address Registers* (BARs) from the PCI Configuration space. The PCI-specific details of obtaining the BAR is abstracted in the `bus_alloc_resource()` function.

For example, a typical driver might have something similar to this in the `attach()` function. :

```
        sc->bar0id = 0x10;
```

```
     sc->bar0res = bus_alloc_resource(dev, SYS_RES_MEMORY, &(sc->bar0id),
    0, ~0, 1, RF_ACTIVE);
     if (sc->bar0res == NULL) {
         uprintf("Memory allocation of PCI base register 0 failed!\n");
         error = ENXIO;
         goto fail1;
     }

     sc->bar1id = 0x14;
     sc->bar1res = bus_alloc_resource(dev, SYS_RES_MEMORY, &(sc->bar1id),
    0, ~0, 1, RF_ACTIVE);
     if (sc->bar1res == NULL) {
         uprintf("Memory allocation of PCI base register 1 failed!\n");
         error =  ENXIO;
         goto fail2;
     }
     sc->bar0_bt = rman_get_bustag(sc->bar0res);
     sc->bar0_bh = rman_get_bushandle(sc->bar0res);
     sc->bar1_bt = rman_get_bustag(sc->bar1res);
     sc->bar1_bh = rman_get_bushandle(sc->bar1res);
```

Handles for each base address register are kept in the softc structure so that they can be used to write to the device later.

These handles can then be used to read or write from the device registers with the `bus_space_*` functions. For example, a driver might contain a shorthand function to read from a board specific register like this :

```
uint16_t
board_read(struct ni_softc *sc, uint16_t address) {
    return bus_space_read_2(sc->bar1_bt, sc->bar1_bh, address);
}
```

Similarly, one could write to the registers with :

```
void
board_write(struct ni_softc *sc, uint16_t address, uint16_t value) {
    bus_space_write_2(sc->bar1_bt, sc->bar1_bh, address, value);
}
```

These functions exist in 8bit, 16bit, and 32bit versions and you should use `bus_space_{read|write}_{1|2|4}` accordingly.

20.2.2. Interrupts

Interrupts are allocated from the object-oriented bus code in a way similar to the memory resources. First an IRQ resource must be allocated from the parent bus, and then the interrupt handler must be setup to deal with this IRQ.

Again, a sample from a device `attach()` function says more than words.

```
/* Get the IRQ resource */

    sc->irqid = 0x0;
    sc->irqres = bus_alloc_resource(dev, SYS_RES_IRQ, &(sc->irqid),
   0, ~0, 1, RF_SHAREABLE | RF_ACTIVE);
    if (sc->irqres == NULL) {
 uprintf("IRQ allocation failed!\n");
 error = ENXIO;
 goto fail3;
    }

    /* Now we should setup the interrupt handler */

    error = bus_setup_intr(dev, sc->irqres, INTR_TYPE_MISC,
    my_handler, sc, &(sc->handler));
    if (error) {
 printf("Couldn't set up irq\n");
 goto fail4;
    }

    sc->irq_bt = rman_get_bustag(sc->irqres);
    sc->irq_bh = rman_get_bushandle(sc->irqres);
```

20.2.3. DMA

On the PC, peripherals that want to do bus-mastering DMA must deal with physical addresses. This is a problem since FreeBSD uses virtual memory and deals almost exclusively with virtual addresses. Fortunately, there is a function, `vtophys()` to help.

```
#include <vm/vm.h>
#include <vm/pmap.h>

#define vtophys(virtual_address) (...)
```

The solution is a bit different on the alpha however, and what we really want is a function called `vtobus()`.

```
#if defined(__alpha__)
```

```
    #define vtobus(va)      alpha_XXX_dmamap((vm_offset_t)va)
    #else
    #define vtobus(va)      vtophys(va)
    #endif
```

20.2.4. Deallocating Resources

It’s very important to deallocate all of the resources that were allocated during `attach()`. Care must be taken to deallocate the correct stuff even on a failure condition so that the system will remain useable while your driver dies.

Chapter 21. Common Access Method SCSI Controllers

This chapter was written by Sergey Babkin <`babkin@FreeBSD.org`> Modifications for the handbook made by Murray Stokely <`murray@FreeBSD.org`>.

21.1. Synopsis

This document assumes that the reader has a general understanding of device drivers in FreeBSD and of the SCSI protocol. Much of the information in this document was extracted from the drivers :

- ncr (`/sys/pci/ncr.c`) by Wolfgang Stanglmeier and Stefan Esser
- sym (`/sys/pci/sym.c`) by Gerard Roudier
- aic7xxx (`/sys/dev/aic7xxx/aic7xxx.c`) by Justin T. Gibbs

and from the CAM code itself (by Justing T. Gibbs, see `/sys/cam/*`). When some solution looked the most logical and was essentially verbatim extracted from the code by Justin Gibbs, I marked it as "recommended".

The document is illustrated with examples in pseudo-code. Although sometimes the examples have many details and look like real code, it's still pseudo-code. It was written to demonstrate the concepts in an understandable way. For a real driver other approaches may be more modular and efficient. It also abstracts from the hardware details, as well as issues that would cloud the demonstrated concepts or that are supposed to be described in the other chapters of the developers handbook. Such details are commonly shown as calls to functions with descriptive names, comments or pseudo-statements. Fortunately real life full-size examples with all the details can be found in the real drivers.

21.2. General architecture

CAM stands for Common Access Method. It's a generic way to address the I/O buses in a SCSI-like way. This allows a separation of the generic device drivers from the drivers controlling the I/O bus: for example the disk driver becomes able to control disks on both SCSI, IDE, and/or any other bus so the disk driver portion does not have to be rewritten (or copied and modified) for every new I/O bus. Thus the two most important active entities are:

- *Peripheral Modules* - a driver for peripheral devices (disk, tape, CDROM, etc.)
- *SCSI Interface Modules* (SIM) - a Host Bus Adapter drivers for connecting to an I/O bus such as SCSI or IDE.

A peripheral driver receives requests from the OS, converts them to a sequence of SCSI commands and passes these SCSI commands to a SCSI Interface Module. The SCSI Interface Module is responsible for passing these commands to the actual hardware (or if the actual hardware is not SCSI but, for example, IDE then also converting the SCSI commands to the native commands of the hardware).

Because we are interested in writing a SCSI adapter driver here, from this point on we will consider everything from the SIM standpoint.

A typical SIM driver needs to include the following CAM-related header files:

```
#include <cam/cam.h>
#include <cam/cam_ccb.h>
#include <cam/cam_sim.h>
#include <cam/cam_xpt_sim.h>
#include <cam/cam_debug.h>
#include <cam/scsi/scsi_all.h>
```

The first thing each SIM driver must do is register itself with the CAM subsystem. This is done during the driver's `xxx_attach()` function (here and further xxx_ is used to denote the unique driver name prefix). The `xxx_attach()` function itself is called by the system bus auto-configuration code which we don't describe here.

This is achieved in multiple steps: first it's necessary to allocate the queue of requests associated with this SIM:

```
    struct cam_devq *devq;

    if(( devq = cam_simq_alloc(SIZE) )==NULL) {
        error; /* some code to handle the error */
    }
```

Here SIZE is the size of the queue to be allocated, maximal number of requests it could contain. It's the number of requests that the SIM driver can handle in parallel on one SCSI card. Commonly it can be calculated as:

```
SIZE = NUMBER_OF_SUPPORTED_TARGETS * MAX_SIMULTANEOUS_COMMANDS_PER_TARGET
```

Next we create a descriptor of our SIM:

```
    struct cam_sim *sim;

    if(( sim = cam_sim_alloc(action_func, poll_func, driver_name,
            softc, unit, max_dev_transactions,
            max_tagged_dev_transactions, devq) )==NULL) {
        cam_simq_free(devq);
        error; /* some code to handle the error */
    }
```

Note that if we are not able to create a SIM descriptor we free the devq also because we can do nothing else with it and we want to conserve memory.

If a SCSI card has multiple SCSI buses on it then each bus requires its own cam_sim structure.

An interesting question is what to do if a SCSI card has more than one SCSI bus, do we need one devq structure per card or per SCSI bus? The answer given in the comments to the CAM code is: either way, as the driver's author prefers.

The arguments are :

- `action_func` - pointer to the driver's `xxx_action` function.

```
static void xxx_action ( struct cam_sim *sim, union ccb *ccb );
```

- `poll_func` - pointer to the driver's `xxx_poll()`

```
static void xxx_poll ( struct cam_sim *sim );
```

- driver_name - the name of the actual driver, such as "ncr" or "wds"
- softc - pointer to the driver's internal descriptor for this SCSI card. This pointer will be used by the driver in future to get private data.
- unit - the controller unit number, for example for controller "wds0" this number will be 0
- max_dev_transactions - maximal number of simultaneous transactions per SCSI target in the non-tagged mode. This value will be almost universally equal to 1, with possible exceptions only for the non-SCSI cards. Also the drivers that hope to take advantage by preparing one transaction while another one is executed may set it to 2 but this does not seem to be worth the complexity.
- max_tagged_dev_transactions - the same thing, but in the tagged mode. Tags are the SCSI way to initiate multiple transactions on a device: each transaction is assigned a unique tag and the transaction is sent to the device. When the device completes some transaction it sends back the result together with the tag so that the SCSI adapter (and the driver) can tell which transaction was completed. This argument is also known as the maximal tag depth. It depends on the abilities of the SCSI adapter.

Finally we register the SCSI buses associated with our SCSI adapter:

```
        if(xpt_bus_register(sim, bus_number) != CAM_SUCCESS) {
            cam_sim_free(sim, /*free_devq*/ TRUE);
            error; /* some code to handle the error */
        }
```

If there is one devq structure per SCSI bus (i.e. we consider a card with multiple buses as multiple cards with one bus each) then the bus number will always be 0, otherwise each bus on the SCSI card should be get a distinct number. Each bus needs its own separate structure cam_sim.

After that our controller is completely hooked to the CAM system. The value of devq can be discarded now: sim will be passed as an argument in all further calls from CAM and devq can be derived from it.

CAM provides the framework for such asynchronous events. Some events originate from the lower levels (the SIM drivers), some events originate from the peripheral drivers, some events originate from the CAM subsystem itself. Any driver can register callbacks for some types of the asynchronous events, so that it would be notified if these events occur.

A typical example of such an event is a device reset. Each transaction and event identifies the devices to which it applies by the means of "path". The target-specific events normally occur during a transaction with this device. So the path from that transaction may be re-used to report this event (this is safe because the event path is copied in the event reporting routine but not deallocated nor passed anywhere further). Also it’s safe to allocate paths dynamically at any time including the interrupt routines, although that incurs certain overhead, and a possible problem with this approach is that there may be no free memory at that time. For a bus reset event we need to define a wildcard path including all devices on the bus. So we can create the path for the future bus reset events in advance and avoid problems with the future memory shortage:

```
        struct cam_path *path;

        if(xpt_create_path(&path, /*periph*/NULL,
                    cam_sim_path(sim), CAM_TARGET_WILDCARD,
                    CAM_LUN_WILDCARD) != CAM_REQ_CMP) {
            xpt_bus_deregister(cam_sim_path(sim));
            cam_sim_free(sim, /*free_devq*/TRUE);
            error; /* some code to handle the error */
        }

        softc->wpath = path;
        softc->sim = sim;
```

As you can see the path includes:

- ID of the peripheral driver (NULL here because we have none)
- ID of the SIM driver (`cam_sim_path(sim)`)
- SCSI target number of the device (CAM_TARGET_WILDCARD means "all devices")
- SCSI LUN number of the subdevice (CAM_LUN_WILDCARD means "all LUNs")

If the driver can't allocate this path it won't be able to work normally, so in that case we dismantle that SCSI bus.

And we save the path pointer in the softc structure for future use. After that we save the value of sim (or we can also discard it on the exit from `xxx_probe()` if we wish).

That's all for a minimalistic initialization. To do things right there is one more issue left.

For a SIM driver there is one particularly interesting event: when a target device is considered lost. In this case resetting the SCSI negotiations with this device may be a good idea. So we register a callback for this event with CAM. The request is passed to CAM by requesting CAM action on a CAM control block for this type of request:

```
        struct ccb_setasync csa;

        xpt_setup_ccb(&csa.ccb_h, path, /*priority*/5);
        csa.ccb_h.func_code = XPT_SASYNC_CB;
        csa.event_enable = AC_LOST_DEVICE;
        csa.callback = xxx_async;
        csa.callback_arg = sim;
        xpt_action((union ccb *)&csa);
```

Now we take a look at the `xxx_action()` and `xxx_poll()` driver entry points.

`static void` **`xxx_action`** `(` *`struct cam_sim *sim`*`,` *`union ccb *ccb`* `);`

Do some action on request of the CAM subsystem. Sim describes the SIM for the request, CCB is the request itself. CCB stands for "CAM Control Block". It is a union of many specific instances, each describing arguments for some type of transactions. All of these instances share the CCB header where the common part of arguments is stored.

CAM supports the SCSI controllers working in both initiator ("normal") mode and target (simulating a SCSI device) mode. Here we only consider the part relevant to the initiator mode.

There are a few function and macros (in other words, methods) defined to access the public data in the struct sim:

- `cam_sim_path(sim)` - the path ID (see above)
- `cam_sim_name(sim)` - the name of the sim
- `cam_sim_softc(sim)` - the pointer to the softc (driver private data) structure
- `cam_sim_unit(sim)` - the unit number
- `cam_sim_bus(sim)` - the bus ID

To identify the device, `xxx_action()` can get the unit number and pointer to its structure softc using these functions.

The type of request is stored in *ccb->ccb_h.func_code*. So generally `xxx_action()` consists of a big switch:

```
        struct xxx_softc *softc = (struct xxx_softc *) cam_sim_softc(sim);
        struct ccb_hdr *ccb_h = &ccb->ccb_h;
        int unit = cam_sim_unit(sim);
        int bus = cam_sim_bus(sim);

        switch(ccb_h->func_code) {
        case ...:
            ...
        default:
            ccb_h->status = CAM_REQ_INVALID;
            xpt_done(ccb);
            break;
        }
```

As can be seen from the default case (if an unknown command was received) the return code of the command is set into *ccb->ccb_h.status* and the completed CCB is returned back to CAM by calling `xpt_done(ccb)`.

`xpt_done()` does not have to be called from `xxx_action()`: For example an I/O request may be enqueued inside the SIM driver and/or its SCSI controller. Then when the device would post an interrupt signaling that the processing of this request is complete `xpt_done()` may be called from the interrupt handling routine.

Actually, the CCB status is not only assigned as a return code but a CCB has some status all the time. Before CCB is passed to the `xxx_action()` routine it gets the status CCB_REQ_INPROG meaning that it's in progress. There are a surprising number of status values defined in `/sys/cam/cam.h` which should be able to represent the status of a request in great detail. More interesting yet, the status is in fact a "bitwise or" of an enumerated status value (the lower 6 bits) and possible additional flag-like bits (the upper bits). The enumerated values will be discussed later in more detail. The summary of them can be found in the Errors Summary section. The possible status flags are:

- *CAM_DEV_QFRZN* - if the SIM driver gets a serious error (for example, the device does not respond to the selection or breaks the SCSI protocol) when processing a CCB it should freeze the request queue by calling `xpt_freeze_simq()`, return the other enqueued but not processed yet CCBs for this device back to the CAM queue, then set this flag for the troublesome CCB and call `xpt_done()`. This flag causes the CAM subsystem to unfreeze the queue after it handles the error.
- *CAM_AUTOSNS_VALID* - if the device returned an error condition and the flag CAM_DIS_AUTOSENSE is not set in CCB the SIM driver must execute the REQUEST SENSE command automatically to extract the sense (extended error information) data from the device. If this attempt was successful the sense data should be saved in the CCB and this flag set.
- *CAM_RELEASE_SIMQ* - like CAM_DEV_QFRZN but used in case there is some problem (or resource shortage) with the SCSI controller itself. Then all the future requests to the controller should be stopped by

`xpt_freeze_simq()`. The controller queue will be restarted after the SIM driver overcomes the shortage and informs CAM by returning some CCB with this flag set.

- *CAM_SIM_QUEUED* - when SIM puts a CCB into its request queue this flag should be set (and removed when this CCB gets dequeued before being returned back to CAM). This flag is not used anywhere in the CAM code now, so its purpose is purely diagnostic.

The function `xxx_action()` is not allowed to sleep, so all the synchronization for resource access must be done using SIM or device queue freezing. Besides the aforementioned flags the CAM subsystem provides functions `xpt_selease_simq()` and `xpt_release_devq()` to unfreeze the queues directly, without passing a CCB to CAM.

The CCB header contains the following fields:

- *path* - path ID for the request
- *target_id* - target device ID for the request
- *target_lun* - LUN ID of the target device
- *timeout* - timeout interval for this command, in milliseconds
- *timeout_ch* - a convenience place for the SIM driver to store the timeout handle (the CAM subsystem itself does not make any assumptions about it)
- *flags* - various bits of information about the request spriv_ptr0, spriv_ptr1 - fields reserved for private use by the SIM driver (such as linking to the SIM queues or SIM private control blocks); actually, they exist as unions: spriv_ptr0 and spriv_ptr1 have the type (void *), spriv_field0 and spriv_field1 have the type unsigned long, sim_priv.entries[0].bytes and sim_priv.entries[1].bytes are byte arrays of the size consistent with the other incarnations of the union and sim_priv.bytes is one array, twice bigger.

The recommended way of using the SIM private fields of CCB is to define some meaningful names for them and use these meaningful names in the driver, like:

```
#define ccb_some_meaningful_name    sim_priv.entries[0].bytes
#define ccb_hcb spriv_ptr1 /* for hardware control block */
```

The most common initiator mode requests are:

- *XPT_SCSI_IO* - execute an I/O transaction

  The instance "struct ccb_scsiio csio" of the union ccb is used to transfer the arguments. They are:

  - *cdb_io* - pointer to the SCSI command buffer or the buffer itself
  - *cdb_len* - SCSI command length
  - *data_ptr* - pointer to the data buffer (gets a bit complicated if scatter/gather is used)

- *dxfer_len* - length of the data to transfer
- *sglist_cnt* - counter of the scatter/gather segments
- *scsi_status* - place to return the SCSI status
- *sense_data* - buffer for the SCSI sense information if the command returns an error (the SIM driver is supposed to run the REQUEST SENSE command automatically in this case if the CCB flag CAM_DIS_AUTOSENSE is not set)
- *sense_len* - the length of that buffer (if it happens to be higher than size of sense_data the SIM driver must silently assume the smaller value) resid, sense_resid - if the transfer of data or SCSI sense returned an error these are the returned counters of the residual (not transferred) data. They do not seem to be especially meaningful, so in a case when they are difficult to compute (say, counting bytes in the SCSI controller's FIFO buffer) an approximate value will do as well. For a successfully completed transfer they must be set to zero.
- *tag_action* - the kind of tag to use:
  - CAM_TAG_ACTION_NONE - don't use tags for this transaction
  - MSG_SIMPLE_Q_TAG, MSG_HEAD_OF_Q_TAG, MSG_ORDERED_Q_TAG - value equal to the appropriate tag message (see /sys/cam/scsi/scsi_message.h); this gives only the tag type, the SIM driver must assign the tag value itself

The general logic of handling this request is the following:

The first thing to do is to check for possible races, to make sure that the command did not get aborted when it was sitting in the queue:

```
    struct ccb_scsiio *csio = &ccb->csio;

    if ((ccb_h->status & CAM_STATUS_MASK) != CAM_REQ_INPROG) {
        xpt_done(ccb);
        return;
    }
```

Also we check that the device is supported at all by our controller:

```
    if(ccb_h->target_id > OUR_MAX_SUPPORTED_TARGET_ID
    || cch_h->target_id == OUR_SCSI_CONTROLLERS_OWN_ID) {
        ccb_h->status = CAM_TID_INVALID;
        xpt_done(ccb);
        return;
    }
    if(ccb_h->target_lun > OUR_MAX_SUPPORTED_LUN) {
        ccb_h->status = CAM_LUN_INVALID;
        xpt_done(ccb);
        return;
    }
```

Then allocate whatever data structures (such as card-dependent hardware control block) we need to process this request. If we can't then freeze the SIM queue and remember that we have a pending operation, return the CCB back and ask CAM to re-queue it. Later when the resources become available the SIM queue must be unfrozen by returning a ccb with the CAM_SIMQ_RELEASE bit set in its status. Otherwise, if all went well, link the CCB with the hardware control block (HCB) and mark it as queued.

```
struct xxx_hcb *hcb = allocate_hcb(softc, unit, bus);

if(hcb == NULL) {
    softc->flags |= RESOURCE_SHORTAGE;
    xpt_freeze_simq(sim, /*count*/1);
    ccb_h->status = CAM_REQUEUE_REQ;
    xpt_done(ccb);
    return;
}

hcb->ccb = ccb; ccb_h->ccb_hcb = (void *)hcb;
ccb_h->status |= CAM_SIM_QUEUED;
```

Extract the target data from CCB into the hardware control block. Check if we are asked to assign a tag and if yes then generate an unique tag and build the SCSI tag messages. The SIM driver is also responsible for negotiations with the devices to set the maximal mutually supported bus width, synchronous rate and offset.

```
hcb->target = ccb_h->target_id; hcb->lun = ccb_h->target_lun;
generate_identify_message(hcb);
if( ccb_h->tag_action != CAM_TAG_ACTION_NONE )
    generate_unique_tag_message(hcb, ccb_h->tag_action);
if( !target_negotiated(hcb) )
    generate_negotiation_messages(hcb);
```

Then set up the SCSI command. The command storage may be specified in the CCB in many interesting ways, specified by the CCB flags. The command buffer can be contained in CCB or pointed to, in the latter case the pointer may be physical or virtual. Since the hardware commonly needs physical address we always convert the address to the physical one.

A NOT-QUITE RELATED NOTE: Normally this is done by a call to vtophys(), but for the PCI device (which account for most of the SCSI controllers now) drivers' portability to the Alpha architecture the conversion must be done by vtobus() instead due to special Alpha quirks. [IMHO it would be much better to have two separate functions, vtop() and ptobus() then vtobus() would be a simple superposition of them.] In case if a physical address is requested it's OK to return the CCB with the status CAM_REQ_INVALID, the current drivers do that. But it's also possible to compile the Alpha-specific piece of code, as in this example (there should be a more direct way to do that, without conditional compilation in the drivers). If necessary a physical address can be also converted or mapped back to a virtual address but with big pain, so we don't do that.

```
if(ccb_h->flags & CAM_CDB_POINTER) {
    /* CDB is a pointer */
```

```
            if(!(ccb_h->flags & CAM_CDB_PHYS)) {
                /* CDB pointer is virtual */
                hcb->cmd = vtobus(csio->cdb_io.cdb_ptr);
            } else {
                /* CDB pointer is physical */
    #if defined(__alpha__)
                hcb->cmd = csio->cdb_io.cdb_ptr | alpha_XXX_dmamap_or ;
    #else
                hcb->cmd = csio->cdb_io.cdb_ptr ;
    #endif
            }
        } else {
            /* CDB is in the ccb (buffer) */
            hcb->cmd = vtobus(csio->cdb_io.cdb_bytes);
        }
        hcb->cmdlen = csio->cdb_len;
```

Now it’s time to set up the data. Again, the data storage may be specified in the CCB in many interesting ways, specified by the CCB flags. First we get the direction of the data transfer. The simplest case is if there is no data to transfer:

```
        int dir = (ccb_h->flags & CAM_DIR_MASK);

        if (dir == CAM_DIR_NONE)
            goto end_data;
```

Then we check if the data is in one chunk or in a scatter-gather list, and the addresses are physical or virtual. The SCSI controller may be able to handle only a limited number of chunks of limited length. If the request hits this limitation we return an error. We use a special function to return the CCB to handle in one place the HCB resource shortages. The functions to add chunks are driver-dependent, and here we leave them without detailed implementation. See description of the SCSI command (CDB) handling for the details on the address-translation issues. If some variation is too difficult or impossible to implement with a particular card it’s OK to return the status CAM_REQ_INVALID. Actually, it seems like the scatter-gather ability is not used anywhere in the CAM code now. But at least the case for a single non-scattered virtual buffer must be implemented, it’s actively used by CAM.

```
        int rv;

        initialize_hcb_for_data(hcb);

        if((!(ccb_h->flags & CAM_SCATTER_VALID)) {
            /* single buffer */
            if(!(ccb_h->flags & CAM_DATA_PHYS)) {
                rv = add_virtual_chunk(hcb, csio->data_ptr, csio->dxfer_len, dir);
                }
            } else {
                rv = add_physical_chunk(hcb, csio->data_ptr, csio->dxfer_len, dir);
            }
        } else {
            int i;
            struct bus_dma_segment *segs;
            segs = (struct bus_dma_segment *)csio->data_ptr;

            if ((ccb_h->flags & CAM_SG_LIST_PHYS) != 0) {
                /* The SG list pointer is physical */
                rv = setup_hcb_for_physical_sg_list(hcb, segs, csio->sglist_cnt);
            } else if (!(ccb_h->flags & CAM_DATA_PHYS)) {
                /* SG buffer pointers are virtual */
                for (i = 0; i < csio->sglist_cnt; i++) {
                    rv = add_virtual_chunk(hcb, segs[i].ds_addr,
                        segs[i].ds_len, dir);
                    if (rv != CAM_REQ_CMP)
                        break;
                }
            } else {
                /* SG buffer pointers are physical */
                for (i = 0; i < csio->sglist_cnt; i++) {
                    rv = add_physical_chunk(hcb, segs[i].ds_addr,
                        segs[i].ds_len, dir);
                    if (rv != CAM_REQ_CMP)
                        break;
                }
            }
        }
        if(rv != CAM_REQ_CMP) {
            /* we expect that add_*_chunk() functions return CAM_REQ_CMP
             * if they added a chunk successfully, CAM_REQ_TOO_BIG if
             * the request is too big (too many bytes or too many chunks),
             * CAM_REQ_INVALID in case of other troubles
             */
            free_hcb_and_ccb_done(hcb, ccb, rv);
            return;
        }
        end_data:
```

If disconnection is disabled for this CCB we pass this information to the hcb:

```
        if(ccb_h->flags & CAM_DIS_DISCONNECT)
            hcb_disable_disconnect(hcb);
```

If the controller is able to run REQUEST SENSE command all by itself then the value of the flag CAM_DIS_AUTOSENSE should also be passed to it, to prevent automatic REQUEST SENSE if the CAM subsystem does not want it.

The only thing left is to set up the timeout, pass our hcb to the hardware and return, the rest will be done by the interrupt handler (or timeout handler).

```
ccb_h->timeout_ch = timeout(xxx_timeout, (caddr_t) hcb,
    (ccb_h->timeout * hz) / 1000); /* convert milliseconds to ticks */
put_hcb_into_hardware_queue(hcb);
return;
```

And here is a possible implementation of the function returning CCB:

```
static void
free_hcb_and_ccb_done(struct xxx_hcb *hcb, union ccb *ccb, u_int32_t status)
{
    struct xxx_softc *softc = hcb->softc;

    ccb->ccb_h.ccb_hcb = 0;
    if(hcb != NULL) {
        untimeout(xxx_timeout, (caddr_t) hcb, ccb->ccb_h.timeout_ch);
        /* we’re about to free a hcb, so the shortage has ended */
        if(softc->flags & RESOURCE_SHORTAGE)  {
            softc->flags &= ~RESOURCE_SHORTAGE;
            status |= CAM_RELEASE_SIMQ;
        }
        free_hcb(hcb); /* also removes hcb from any internal lists */
    }
    ccb->ccb_h.status = status |
        (ccb->ccb_h.status & ~(CAM_STATUS_MASK|CAM_SIM_QUEUED));
    xpt_done(ccb);
}
```

- *XPT_RESET_DEV* - send the SCSI "BUS DEVICE RESET" message to a device

  There is no data transferred in CCB except the header and the most interesting argument of it is target_id. Depending on the controller hardware a hardware control block just like for the XPT_SCSI_IO request may be constructed (see XPT_SCSI_IO request description) and sent to the controller or the SCSI controller may be immediately programmed to send this RESET message to the device or this request may be just not supported (and return the status CAM_REQ_INVALID). Also on completion of the request all the disconnected transactions for this target must be aborted (probably in the interrupt routine).

  Also all the current negotiations for the target are lost on reset, so they might be cleaned too. Or they clearing may be deferred, because anyway the target would request re-negotiation on the next transaction.

- *XPT_RESET_BUS* - send the RESET signal to the SCSI bus

  No arguments are passed in the CCB, the only interesting argument is the SCSI bus indicated by the struct sim pointer.

  A minimalistic implementation would forget the SCSI negotiations for all the devices on the bus and return the status CAM_REQ_CMP.

  The proper implementation would in addition actually reset the SCSI bus (possible also reset the SCSI controller) and mark all the CCBs being processed, both those in the hardware queue and those being disconnected, as done with the status CAM_SCSI_BUS_RESET. Like:

```
int targ, lun;
struct xxx_hcb *h, *hh;
struct ccb_trans_settings neg;
struct cam_path *path;

/* The SCSI bus reset may take a long time, in this case its completion
 * should be checked by interrupt or timeout. But for simplicity
 * we assume here that it’s really fast.
 */
reset_scsi_bus(softc);

/* drop all enqueued CCBs */
for(h = softc->first_queued_hcb; h != NULL; h = hh) {
    hh = h->next;
    free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
}

/* the clean values of negotiations to report */
neg.bus_width = 8;
neg.sync_period = neg.sync_offset = 0;
neg.valid = (CCB_TRANS_BUS_WIDTH_VALID
    | CCB_TRANS_SYNC_RATE_VALID | CCB_TRANS_SYNC_OFFSET_VALID);

/* drop all disconnected CCBs and clean negotiations  */
for(targ=0; targ <= OUR_MAX_SUPPORTED_TARGET; targ++) {
    clean_negotiations(softc, targ);

    /* report the event if possible */
    if(xpt_create_path(&path, /*periph*/NULL,
            cam_sim_path(sim), targ,
            CAM_LUN_WILDCARD) == CAM_REQ_CMP) {
        xpt_async(AC_TRANSFER_NEG, path, &neg);
        xpt_free_path(path);
    }

    for(lun=0; lun <= OUR_MAX_SUPPORTED_LUN; lun++)
        for(h = softc->first_discon_hcb[targ][lun]; h != NULL; h = hh) {
            hh=h->next;
            free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
        }
}

ccb->ccb_h.status = CAM_REQ_CMP;
xpt_done(ccb);

/* report the event */
xpt_async(AC_BUS_RESET, softc->wpath, NULL);
return;
```

Implementing the SCSI bus reset as a function may be a good idea because it would be re-used by the timeout function as a last resort if the things go wrong.

- *XPT_ABORT* - abort the specified CCB

  The arguments are transferred in the instance "struct ccb_abort cab" of the union ccb. The only argument field in it is:

  abort_ccb - pointer to the CCB to be aborted

  If the abort is not supported just return the status CAM_UA_ABORT. This is also the easy way to minimally implement this call, return CAM_UA_ABORT in any case.

  The hard way is to implement this request honestly. First check that abort applies to a SCSI transaction:

```
struct ccb *abort_ccb;
abort_ccb = ccb->cab.abort_ccb;

if(abort_ccb->ccb_h.func_code != XPT_SCSI_IO) {
    ccb->ccb_h.status = CAM_UA_ABORT;
    xpt_done(ccb);
    return;
}
```

  Then it’s necessary to find this CCB in our queue. This can be done by walking the list of all our hardware control blocks in search for one associated with this CCB:

```
struct xxx_hcb *hcb, *h;

hcb = NULL;
```

```
/* We assume that softc->first_hcb is the head of the list of all
 * HCBs associated with this bus, including those enqueued for
 * processing, being processed by hardware and disconnected ones.
 */
for(h = softc->first_hcb; h != NULL; h = h->next) {
    if(h->ccb == abort_ccb) {
        hcb = h;
        break;
    }
}

if(hcb == NULL) {
    /* no such CCB in our queue */
    ccb->ccb_h.status = CAM_PATH_INVALID;
    xpt_done(ccb);
    return;
}

hcb=found_hcb;
```

Now we look at the current processing status of the HCB. It may be either sitting in the queue waiting to be sent to the SCSI bus, being transferred right now, or disconnected and waiting for the result of the command, or actually completed by hardware but not yet marked as done by software. To make sure that we don’t get in any races with hardware we mark the HCB as being aborted, so that if this HCB is about to be sent to the SCSI bus the SCSI controller will see this flag and skip it.

```
int hstatus;

/* shown as a function, in case special action is needed to make
 * this flag visible to hardware
 */
set_hcb_flags(hcb, HCB_BEING_ABORTED);

abort_again:

hstatus = get_hcb_status(hcb);
switch(hstatus) {
case HCB_SITTING_IN_QUEUE:
    remove_hcb_from_hardware_queue(hcb);
    /* FALLTHROUGH */
case HCB_COMPLETED:
    /* this is an easy case */
    free_hcb_and_ccb_done(hcb, abort_ccb, CAM_REQ_ABORTED);
    break;
```

If the CCB is being transferred right now we would like to signal to the SCSI controller in some hardware-dependent way that we want to abort the current transfer. The SCSI controller would set the SCSI ATTENTION signal and when the target responds to it send an ABORT message. We also reset the timeout to make sure that the target is not sleeping forever. If the command would not get aborted in some reasonable time like 10 seconds the timeout routine would go ahead and reset the whole SCSI bus. Because the command will be aborted in some reasonable time we can just return the abort request now as successfully completed, and mark the aborted CCB as aborted (but not mark it as done yet).

```
        case HCB_BEING_TRANSFERRED:
            untimeout(xxx_timeout, (caddr_t) hcb, abort_ccb->ccb_h.timeout_ch);
            abort_ccb->ccb_h.timeout_ch =
                timeout(xxx_timeout, (caddr_t) hcb, 10 * hz);
            abort_ccb->ccb_h.status = CAM_REQ_ABORTED;
            /* ask the controller to abort that HCB, then generate
             * an interrupt and stop
             */
            if(signal_hardware_to_abort_hcb_and_stop(hcb) < 0) {
                /* oops, we missed the race with hardware, this transaction
                 * got off the bus before we aborted it, try again */
                goto abort_again;
            }

            break;
```

If the CCB is in the list of disconnected then set it up as an abort request and re-queue it at the front of hardware queue. Reset the timeout and report the abort request to be completed.

```
        case HCB_DISCONNECTED:
            untimeout(xxx_timeout, (caddr_t) hcb, abort_ccb->ccb_h.timeout_ch);
            abort_ccb->ccb_h.timeout_ch =
                timeout(xxx_timeout, (caddr_t) hcb, 10 * hz);
            put_abort_message_into_hcb(hcb);
            put_hcb_at_the_front_of_hardware_queue(hcb);
            break;
        }
        ccb->ccb_h.status = CAM_REQ_CMP;
        xpt_done(ccb);
        return;
```

That’s all for the ABORT request, although there is one more issue. Because the ABORT message cleans all the ongoing transactions on a LUN we have to mark all the other active transactions on this LUN as aborted. That should be done in the interrupt routine, after the transaction gets aborted.

Implementing the CCB abort as a function may be quite a good idea, this function can be re-used if an I/O transaction times out. The only difference would be that the timed out transaction would return the status CAM_CMD_TIMEOUT for the timed out request. Then the case XPT_ABORT would be small, like that:

```
        case XPT_ABORT:
            struct ccb *abort_ccb;
            abort_ccb = ccb->cab.abort_ccb;

            if(abort_ccb->ccb_h.func_code != XPT_SCSI_IO) {
                ccb->ccb_h.status = CAM_UA_ABORT;
                xpt_done(ccb);
                return;
            }
            if(xxx_abort_ccb(abort_ccb, CAM_REQ_ABORTED) < 0)
                /* no such CCB in our queue */
                ccb->ccb_h.status = CAM_PATH_INVALID;
            else
                ccb->ccb_h.status = CAM_REQ_CMP;
            xpt_done(ccb);
            return;
```

- *XPT_SET_TRAN_SETTINGS* - explicitly set values of SCSI transfer settings

  The arguments are transferred in the instance "struct ccb_trans_setting cts" of the union ccb:

  - *valid* - a bitmask showing which settings should be updated:
  - *CCB_TRANS_SYNC_RATE_VALID* - synchronous transfer rate
  - *CCB_TRANS_SYNC_OFFSET_VALID* - synchronous offset
  - *CCB_TRANS_BUS_WIDTH_VALID* - bus width
  - *CCB_TRANS_DISC_VALID* - set enable/disable disconnection
  - *CCB_TRANS_TQ_VALID* - set enable/disable tagged queuing
  - *flags* - consists of two parts, binary arguments and identification of sub-operations. The binary arguments are :
    - *CCB_TRANS_DISC_ENB* - enable disconnection
    - *CCB_TRANS_TAG_ENB* - enable tagged queuing
  - the sub-operations are:
    - *CCB_TRANS_CURRENT_SETTINGS* - change the current negotiations
    - *CCB_TRANS_USER_SETTINGS* - remember the desired user values sync_period, sync_offset - self-explanatory, if sync_offset==0 then the asynchronous mode is requested bus_width - bus width, in bits (not bytes)

Two sets of negotiated parameters are supported, the user settings and the current settings. The user settings are not really used much in the SIM drivers, this is mostly just a piece of memory where the upper levels can store (and later recall) its ideas about the parameters. Setting the user parameters does not cause re-negotiation of the transfer rates. But when the SCSI controller does a negotiation it must never set the values higher than the user parameters, so it's essentially the top boundary.

The current settings are, as the name says, current. Changing them means that the parameters must be re-negotiated on the next transfer. Again, these "new current settings" are not supposed to be forced on the device, just they are used as the initial step of negotiations. Also they must be limited by actual capabilities of the SCSI controller: for example, if the SCSI controller has 8-bit bus and the request asks to set 16-bit wide transfers this parameter must be silently truncated to 8-bit transfers before sending it to the device.

One caveat is that the bus width and synchronous parameters are per target while the disconnection and tag enabling parameters are per lun.

The recommended implementation is to keep 3 sets of negotiated (bus width and synchronous transfer) parameters:

- *user* - the user set, as above
- *current* - those actually in effect
- *goal* - those requested by setting of the "current" parameters

The code looks like:

```
        struct ccb_trans_settings *cts;
        int targ, lun;
        int flags;

        cts = &ccb->cts;
        targ = ccb_h->target_id;
        lun = ccb_h->target_lun;
        flags = cts->flags;
        if(flags & CCB_TRANS_USER_SETTINGS) {
            if(flags & CCB_TRANS_SYNC_RATE_VALID)
                softc->user_sync_period[targ] = cts->sync_period;
            if(flags & CCB_TRANS_SYNC_OFFSET_VALID)
                softc->user_sync_offset[targ] = cts->sync_offset;
            if(flags & CCB_TRANS_BUS_WIDTH_VALID)
                softc->user_bus_width[targ] = cts->bus_width;

            if(flags & CCB_TRANS_DISC_VALID) {
                softc->user_tflags[targ][lun] &= ~CCB_TRANS_DISC_ENB;
                softc->user_tflags[targ][lun] |= flags & CCB_TRANS_DISC_ENB;
            }
            if(flags & CCB_TRANS_TQ_VALID) {
                softc->user_tflags[targ][lun] &= ~CCB_TRANS_TQ_ENB;
                softc->user_tflags[targ][lun] |= flags & CCB_TRANS_TQ_ENB;
            }
        }
        if(flags & CCB_TRANS_CURRENT_SETTINGS) {
            if(flags & CCB_TRANS_SYNC_RATE_VALID)
                softc->goal_sync_period[targ] =
                    max(cts->sync_period, OUR_MIN_SUPPORTED_PERIOD);
            if(flags & CCB_TRANS_SYNC_OFFSET_VALID)
                softc->goal_sync_offset[targ] =
                    min(cts->sync_offset, OUR_MAX_SUPPORTED_OFFSET);
            if(flags & CCB_TRANS_BUS_WIDTH_VALID)
                softc->goal_bus_width[targ] = min(cts->bus_width, OUR_BUS_WIDTH);

            if(flags & CCB_TRANS_DISC_VALID) {
                softc->current_tflags[targ][lun] &= ~CCB_TRANS_DISC_ENB;
                softc->current_tflags[targ][lun] |= flags & CCB_TRANS_DISC_ENB;
            }
            if(flags & CCB_TRANS_TQ_VALID) {
                softc->current_tflags[targ][lun] &= ~CCB_TRANS_TQ_ENB;
                softc->current_tflags[targ][lun] |= flags & CCB_TRANS_TQ_ENB;
            }
        }
        ccb->ccb_h.status = CAM_REQ_CMP;
        xpt_done(ccb);
        return;
```

Then when the next I/O request will be processed it will check if it has to re-negotiate, for example by calling the function target_negotiated(hcb). It can be implemented like this:

```
    int
    target_negotiated(struct xxx_hcb *hcb)
    {
        struct softc *softc = hcb->softc;
        int targ = hcb->targ;

        if( softc->current_sync_period[targ] != softc->goal_sync_period[targ]
        || softc->current_sync_offset[targ] != softc->goal_sync_offset[targ]
        || softc->current_bus_width[targ] != softc->goal_bus_width[targ] )
            return 0; /* FALSE */
        else
            return 1; /* TRUE */
    }
```

After the values are re-negotiated the resulting values must be assigned to both current and goal parameters, so for future I/O transactions the current and goal parameters would be the same and `target_negotiated()` would return TRUE. When the card is initialized (in `xxx_attach()`) the current negotiation values must be initialized to narrow asynchronous mode, the goal and current values must be initialized to the maximal values supported by controller.

- *XPT_GET_TRAN_SETTINGS* - get values of SCSI transfer settings

  This operations is the reverse of XPT_SET_TRAN_SETTINGS. Fill up the CCB instance "struct ccb_trans_setting cts" with data as requested by the flags CCB_TRANS_CURRENT_SETTINGS or CCB_TRANS_USER_SETTINGS (if both are set then the existing drivers return the current settings). Set all the bits in the valid field.

- *XPT_CALC_GEOMETRY* - calculate logical (BIOS) geometry of the disk

  The arguments are transferred in the instance "struct ccb_calc_geometry ccg" of the union ccb:

  - *block_size* - input, block (A.K.A sector) size in bytes
  - *volume_size* - input, volume size in bytes
  - *cylinders* - output, logical cylinders
  - *heads* - output, logical heads
  - *secs_per_track* - output, logical sectors per track

  If the returned geometry differs much enough from what the SCSI controller BIOS thinks and a disk on this SCSI controller is used as bootable the system may not be able to boot. The typical calculation example taken from the aic7xxx driver is:

```
        struct    ccb_calc_geometry *ccg;
        u_int32_t size_mb;
        u_int32_t secs_per_cylinder;
        int   extended;

        ccg = &ccb->ccg;
        size_mb = ccg->volume_size
            / ((1024L * 1024L) / ccg->block_size);
        extended = check_cards_EEPROM_for_extended_geometry(softc);

        if (size_mb > 1024 && extended) {
            ccg->heads = 255;
            ccg->secs_per_track = 63;
        } else {
            ccg->heads = 64;
            ccg->secs_per_track = 32;
        }
        secs_per_cylinder = ccg->heads * ccg->secs_per_track;
        ccg->cylinders = ccg->volume_size / secs_per_cylinder;
        ccb->ccb_h.status = CAM_REQ_CMP;
        xpt_done(ccb);
        return;
```

This gives the general idea, the exact calculation depends on the quirks of the particular BIOS. If BIOS provides no way set the "extended translation" flag in EEPROM this flag should normally be assumed equal to 1. Other popular geometries are:

```
        128 heads, 63 sectors - Symbios controllers
        16 heads, 63 sectors - old controllers
```

Some system BIOSes and SCSI BIOSes fight with each other with variable success, for example a combination of Symbios 875/895 SCSI and Phoenix BIOS can give geometry 128/63 after power up and 255/63 after a hard reset or soft reboot.

- *XPT_PATH_INQ* - path inquiry, in other words get the SIM driver and SCSI controller (also known as HBA - Host Bus Adapter) properties

  The properties are returned in the instance "struct ccb_pathinq cpi" of the union ccb:

  - version_num - the SIM driver version number, now all drivers use 1
  - hba_inquiry - bitmask of features supported by the controller:
  - PI_MDP_ABLE - supports MDP message (something from SCSI3?)
  - PI_WIDE_32 - supports 32 bit wide SCSI
  - PI_WIDE_16 - supports 16 bit wide SCSI
  - PI_SDTR_ABLE - can negotiate synchronous transfer rate
  - PI_LINKED_CDB - supports linked commands
  - PI_TAG_ABLE - supports tagged commands
  - PI_SOFT_RST - supports soft reset alternative (hard reset and soft reset are mutually exclusive within a SCSI bus)
  - target_sprt - flags for target mode support, 0 if unsupported
  - hba_misc - miscellaneous controller features:
  - PIM_SCANHILO - bus scans from high ID to low ID
  - PIM_NOREMOVE - removable devices not included in scan
  - PIM_NOINITIATOR - initiator role not supported

- PIM_NOBUSRESET - user has disabled initial BUS RESET
- hba_eng_cnt - mysterious HBA engine count, something related to compression, now is always set to 0
- vuhba_flags - vendor-unique flags, unused now
- max_target - maximal supported target ID (7 for 8-bit bus, 15 for 16-bit bus, 127 for Fibre Channel)
- max_lun - maximal supported LUN ID (7 for older SCSI controllers, 63 for newer ones)
- async_flags - bitmask of installed Async handler, unused now
- hpath_id - highest Path ID in the subsystem, unused now
- unit_number - the controller unit number, cam_sim_unit(sim)
- bus_id - the bus number, cam_sim_bus(sim)
- initiator_id - the SCSI ID of the controller itself
- base_transfer_speed - nominal transfer speed in KB/s for asynchronous narrow transfers, equals to 3300 for SCSI
- sim_vid - SIM driver's vendor id, a zero-terminated string of maximal length SIM_IDLEN including the terminating zero
- hba_vid - SCSI controller's vendor id, a zero-terminated string of maximal length HBA_IDLEN including the terminating zero
- dev_name - device driver name, a zero-terminated string of maximal length DEV_IDLEN including the terminating zero, equal to cam_sim_name(sim)

The recommended way of setting the string fields is using strncpy, like:

```
        strncpy(cpi->dev_name, cam_sim_name(sim), DEV_IDLEN);
```

After setting the values set the status to CAM_REQ_CMP and mark the CCB as done.

21.3. Polling

```
static void xxx_poll ( struct cam_sim *sim );
```

The poll function is used to simulate the interrupts when the interrupt subsystem is not functioning (for example, when the system has crashed and is creating the system dump). The CAM subsystem sets the proper interrupt level before calling the poll routine. So all it needs to do is to call the interrupt routine (or the other way around, the poll routine may be doing the real action and the interrupt routine would just call the poll routine). Why bother about a separate function then ? Because of different calling conventions. The `xxx_poll` routine gets the struct cam_sim pointer as its argument when the PCI interrupt routine by common convention gets pointer to the struct xxx_softc and the ISA interrupt routine gets just the the device unit number. So the poll routine would normally look as:

```
static void
xxx_poll(struct cam_sim *sim)
{
    xxx_intr((struct xxx_softc *)cam_sim_softc(sim)); /* for PCI device */
}
```

or

```
static void
xxx_poll(struct cam_sim *sim)
{
    xxx_intr(cam_sim_unit(sim)); /* for ISA device */
}
```

21.4. Asynchronous Events

If an asynchronous event callback has been set up then the callback function should be defined.

```
static void
ahc_async(void *callback_arg, u_int32_t code, struct cam_path *path, void *arg)
```

- callback_arg - the value supplied when registering the callback
- code - identifies the type of event
- path - identifies the devices to which the event applies
- arg - event-specific argument

Implementation for a single type of event, AC_LOST_DEVICE, looks like:

```
    struct xxx_softc *softc;
    struct cam_sim *sim;
    int targ;
    struct ccb_trans_settings neg;

    sim = (struct cam_sim *)callback_arg;
    softc = (struct xxx_softc *)cam_sim_softc(sim);
```

```
switch (code) {
case AC_LOST_DEVICE:
    targ = xpt_path_target_id(path);
    if(targ <= OUR_MAX_SUPPORTED_TARGET) {
        clean_negotiations(softc, targ);
        /* send indication to CAM */
        neg.bus_width = 8;
        neg.sync_period = neg.sync_offset = 0;
        neg.valid = (CCB_TRANS_BUS_WIDTH_VALID
            | CCB_TRANS_SYNC_RATE_VALID | CCB_TRANS_SYNC_OFFSET_VALID);
        xpt_async(AC_TRANSFER_NEG, path, &neg);
    }
    break;
default:
    break;
}
```

21.5. Interrupts

The exact type of the interrupt routine depends on the type of the peripheral bus (PCI, ISA and so on) to which the SCSI controller is connected.

The interrupt routines of the SIM drivers run at the interrupt level splcam. So `splcam()` should be used in the driver to synchronize activity between the interrupt routine and the rest of the driver (for a multiprocessor-aware driver things get yet more interesting but we ignore this case here). The pseudo-code in this document happily ignores the problems of synchronization. The real code must not ignore them. A simple-minded approach is to set `splcam()` on the entry to the other routines and reset it on return thus protecting them by one big critical section. To make sure that the interrupt level will be always restored a wrapper function can be defined, like:

```
static void
xxx_action(struct cam_sim *sim, union ccb *ccb)
{
    int s;
    s = splcam();
    xxx_action1(sim, ccb);
    splx(s);
}

static void
xxx_action1(struct cam_sim *sim, union ccb *ccb)
{
    ... process the request ...
```

```
        }
```

This approach is simple and robust but the problem with it is that interrupts may get blocked for a relatively long time and this would negatively affect the system's performance. On the other hand the functions of the `spl()` family have rather high overhead, so vast amount of tiny critical sections may not be good either.

The conditions handled by the interrupt routine and the details depend very much on the hardware. We consider the set of "typical" conditions.

First, we check if a SCSI reset was encountered on the bus (probably caused by another SCSI controller on the same SCSI bus). If so we drop all the enqueued and disconnected requests, report the events and re-initialize our SCSI controller. It is important that during this initialization the controller won't issue another reset or else two controllers on the same SCSI bus could ping-pong resets forever. The case of fatal controller error/hang could be handled in the same place, but it will probably need also sending RESET signal to the SCSI bus to reset the status of the connections with the SCSI devices.

```
        int fatal=0;
        struct ccb_trans_settings neg;
        struct cam_path *path;

        if( detected_scsi_reset(softc)
        || (fatal = detected_fatal_controller_error(softc)) ) {
            int targ, lun;
            struct xxx_hcb *h, *hh;

            /* drop all enqueued CCBs */
            for(h = softc->first_queued_hcb; h != NULL; h = hh) {
                hh = h->next;
                free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
            }

            /* the clean values of negotiations to report */
            neg.bus_width = 8;
            neg.sync_period = neg.sync_offset = 0;
            neg.valid = (CCB_TRANS_BUS_WIDTH_VALID
                | CCB_TRANS_SYNC_RATE_VALID | CCB_TRANS_SYNC_OFFSET_VALID);

            /* drop all disconnected CCBs and clean negotiations  */
            for(targ=0; targ <= OUR_MAX_SUPPORTED_TARGET; targ++) {
                clean_negotiations(softc, targ);

                /* report the event if possible */
                if(xpt_create_path(&path, /*periph*/NULL,
                        cam_sim_path(sim), targ,
                        CAM_LUN_WILDCARD) == CAM_REQ_CMP) {
```

```
                xpt_async(AC_TRANSFER_NEG, path, &neg);
                xpt_free_path(path);
            }

            for(lun=0; lun <= OUR_MAX_SUPPORTED_LUN; lun++)
                for(h = softc->first_discon_hcb[targ][lun]; h != NULL; h = hh) {
                    hh=h->next;
                    if(fatal)
                        free_hcb_and_ccb_done(h, h->ccb, CAM_UNREC_HBA_ERROR);
                    else
                        free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
                }
        }

        /* report the event */
        xpt_async(AC_BUS_RESET, softc->wpath, NULL);

        /* re-initialization may take a lot of time, in such case
         * its completion should be signaled by another interrupt or
         * checked on timeout - but for simplicity we assume here that
         * it’s really fast
         */
        if(!fatal) {
            reinitialize_controller_without_scsi_reset(softc);
        } else {
            reinitialize_controller_with_scsi_reset(softc);
        }
        schedule_next_hcb(softc);
        return;
    }
```

If interrupt is not caused by a controller-wide condition then probably something has happened to the current hardware control block. Depending on the hardware there may be other non-HCB-related events, we just do not consider them here. Then we analyze what happened to this HCB:

```
    struct xxx_hcb *hcb, *h, *hh;
    int hcb_status, scsi_status;
    int ccb_status;
    int targ;
    int lun_to_freeze;

    hcb = get_current_hcb(softc);
    if(hcb == NULL) {
        /* either stray interrupt or something went very wrong
         * or this is something hardware-dependent
```

```
             */
            handle as necessary;
            return;
        }

        targ = hcb->target;
        hcb_status = get_status_of_current_hcb(softc);
```

First we check if the HCB has completed and if so we check the returned SCSI status.

```
        if(hcb_status == COMPLETED) {
            scsi_status = get_completion_status(hcb);
```

Then look if this status is related to the REQUEST SENSE command and if so handle it in a simple way.

```
            if(hcb->flags & DOING_AUTOSENSE) {
                if(scsi_status == GOOD) { /* autosense was successful */
                    hcb->ccb->ccb_h.status |= CAM_AUTOSNS_VALID;
                    free_hcb_and_ccb_done(hcb, hcb->ccb, CAM_SCSI_STATUS_ERROR);
                } else {
            autosense_failed:
                    free_hcb_and_ccb_done(hcb, hcb->ccb, CAM_AUTOSENSE_FAIL);
                }
                schedule_next_hcb(softc);
                return;
            }
```

Else the command itself has completed, pay more attention to details. If auto-sense is not disabled for this CCB and the command has failed with sense data then run REQUEST SENSE command to receive that data.

```
            hcb->ccb->csio.scsi_status = scsi_status;
            calculate_residue(hcb);

            if( (hcb->ccb->ccb_h.flags & CAM_DIS_AUTOSENSE)==0
            && ( scsi_status == CHECK_CONDITION
                    || scsi_status == COMMAND_TERMINATED) ) {
                /* start auto-SENSE */
                hcb->flags |= DOING_AUTOSENSE;
                setup_autosense_command_in_hcb(hcb);
                restart_current_hcb(softc);
                return;
            }
            if(scsi_status == GOOD)
                free_hcb_and_ccb_done(hcb, hcb->ccb, CAM_REQ_CMP);
            else
```

```
                free_hcb_and_ccb_done(hcb, hcb->ccb, CAM_SCSI_STATUS_ERROR);
            schedule_next_hcb(softc);
            return;
        }
```

One typical thing would be negotiation events: negotiation messages received from a SCSI target (in answer to our negotiation attempt or by target's initiative) or the target is unable to negotiate (rejects our negotiation messages or does not answer them).

```
        switch(hcb_status) {
        case TARGET_REJECTED_WIDE_NEG:
            /* revert to 8-bit bus */
            softc->current_bus_width[targ] = softc->goal_bus_width[targ] = 8;
            /* report the event */
            neg.bus_width = 8;
            neg.valid = CCB_TRANS_BUS_WIDTH_VALID;
            xpt_async(AC_TRANSFER_NEG, hcb->ccb.ccb_h.path_id, &neg);
            continue_current_hcb(softc);
            return;
        case TARGET_ANSWERED_WIDE_NEG:
            {
                int wd;

                wd = get_target_bus_width_request(softc);
                if(wd <= softc->goal_bus_width[targ]) {
                    /* answer is acceptable */
                    softc->current_bus_width[targ] =
                    softc->goal_bus_width[targ] = neg.bus_width = wd;

                    /* report the event */
                    neg.valid = CCB_TRANS_BUS_WIDTH_VALID;
                    xpt_async(AC_TRANSFER_NEG, hcb->ccb.ccb_h.path_id, &neg);
                } else {
                    prepare_reject_message(hcb);
                }
            }
            continue_current_hcb(softc);
            return;
        case TARGET_REQUESTED_WIDE_NEG:
            {
                int wd;

                wd = get_target_bus_width_request(softc);
                wd = min (wd, OUR_BUS_WIDTH);
                wd = min (wd, softc->user_bus_width[targ]);
                if(wd != softc->current_bus_width[targ]) {
                    /* the bus width has changed */
                    softc->current_bus_width[targ] =
                    softc->goal_bus_width[targ] = neg.bus_width = wd;

                    /* report the event */
                    neg.valid = CCB_TRANS_BUS_WIDTH_VALID;
                    xpt_async(AC_TRANSFER_NEG, hcb->ccb.ccb_h.path_id, &neg);
                }
                prepare_width_nego_rsponse(hcb, wd);
            }
            continue_current_hcb(softc);
            return;
        }
```

Then we handle any errors that could have happened during auto-sense in the same simple-minded way as before. Otherwise we look closer at the details again.

```
        if(hcb->flags & DOING_AUTOSENSE)
            goto autosense_failed;

        switch(hcb_status) {
```

The next event we consider is unexpected disconnect. Which is considered normal after an ABORT or BUS DEVICE RESET message and abnormal in other cases.

```
        case UNEXPECTED_DISCONNECT:
            if(requested_abort(hcb)) {
                /* abort affects all commands on that target+LUN, so
                 * mark all disconnected HCBs on that target+LUN as aborted too
                 */
                for(h = softc->first_discon_hcb[hcb->target][hcb->lun];
                        h != NULL; h = hh) {
                    hh=h->next;
                    free_hcb_and_ccb_done(h, h->ccb, CAM_REQ_ABORTED);
                }
                ccb_status = CAM_REQ_ABORTED;
            } else if(requested_bus_device_reset(hcb)) {
                int lun;

                /* reset affects all commands on that target, so
                 * mark all disconnected HCBs on that target+LUN as reset
                 */
                for(lun=0; lun <= OUR_MAX_SUPPORTED_LUN; lun++)
                    for(h = softc->first_discon_hcb[hcb->target][lun];
                            h != NULL; h = hh) {
                        hh=h->next;
                        free_hcb_and_ccb_done(h, h->ccb, CAM_SCSI_BUS_RESET);
                    }

                /* send event */
                xpt_async(AC_SENT_BDR, hcb->ccb->ccb_h.path_id, NULL);

                /* this was the CAM_RESET_DEV request itself, it’s completed */
                ccb_status = CAM_REQ_CMP;
            } else {
                calculate_residue(hcb);
                ccb_status = CAM_UNEXP_BUSFREE;
                /* request the further code to freeze the queue */
                hcb->ccb->ccb_h.status |= CAM_DEV_QFRZN;
                lun_to_freeze = hcb->lun;
            }
            break;
```

If the target refuses to accept tags we notify CAM about that and return back all commands for this LUN:

```
    case TAGS_REJECTED:
        /* report the event */
        neg.flags = 0 & ~CCB_TRANS_TAG_ENB;
        neg.valid = CCB_TRANS_TQ_VALID;
        xpt_async(AC_TRANSFER_NEG, hcb->ccb.ccb_h.path_id, &neg);

        ccb_status = CAM_MSG_REJECT_REC;
        /* request the further code to freeze the queue */
        hcb->ccb->ccb_h.status |= CAM_DEV_QFRZN;
        lun_to_freeze = hcb->lun;
        break;
```

Then we check a number of other conditions, with processing basically limited to setting the CCB status:

```
    case SELECTION_TIMEOUT:
        ccb_status = CAM_SEL_TIMEOUT;
        /* request the further code to freeze the queue */
        hcb->ccb->ccb_h.status |= CAM_DEV_QFRZN;
        lun_to_freeze = CAM_LUN_WILDCARD;
        break;
    case PARITY_ERROR:
        ccb_status = CAM_UNCOR_PARITY;
```

```
                break;
            case DATA_OVERRUN:
            case ODD_WIDE_TRANSFER:
                ccb_status = CAM_DATA_RUN_ERR;
                break;
            default:
                /* all other errors are handled in a generic way */
                ccb_status = CAM_REQ_CMP_ERR;
                /* request the further code to freeze the queue */
                hcb->ccb->ccb_h.status |= CAM_DEV_QFRZN;
                lun_to_freeze = CAM_LUN_WILDCARD;
                break;
            }
```

Then we check if the error was serious enough to freeze the input queue until it gets proceeded and do so if it is:

```
            if(hcb->ccb->ccb_h.status & CAM_DEV_QFRZN) {
                /* freeze the queue */
                xpt_freeze_devq(ccb->ccb_h.path, /*count*/1);

                /* re-queue all commands for this target/LUN back to CAM */

                for(h = softc->first_queued_hcb; h != NULL; h = hh) {
                    hh = h->next;

                    if(targ == h->targ
                    && (lun_to_freeze == CAM_LUN_WILDCARD || lun_to_freeze == h->lun) )
                        free_hcb_and_ccb_done(h, h->ccb, CAM_REQUEUE_REQ);
                }
            }
            free_hcb_and_ccb_done(hcb, hcb->ccb, ccb_status);
            schedule_next_hcb(softc);
            return;
```

This concludes the generic interrupt handling although specific controllers may require some additions.

21.6. Errors Summary

When executing an I/O request many things may go wrong. The reason of error can be reported in the CCB status with great detail. Examples of use are spread throughout this document. For completeness here is the summary of recommended responses for the typical error conditions:

- *CAM_RESRC_UNAVAIL* - some resource is temporarily unavailable and the SIM driver can not generate an event when it will become available. An example of this resource would be some intra-controller hardware resource for which the controller does not generate an interrupt when it becomes available.
- *CAM_UNCOR_PARITY* - unrecovered parity error occurred
- *CAM_DATA_RUN_ERR* - data overrun or unexpected data phase (going in other direction than specified in CAM_DIR_MASK) or odd transfer length for wide transfer
- *CAM_SEL_TIMEOUT* - selection timeout occurred (target does not respond)
- *CAM_CMD_TIMEOUT* - command timeout occurred (the timeout function ran)
- *CAM_SCSI_STATUS_ERROR* - the device returned error
- *CAM_AUTOSENSE_FAIL* - the device returned error and the REQUEST SENSE COMMAND failed
- *CAM_MSG_REJECT_REC* - MESSAGE REJECT message was received
- *CAM_SCSI_BUS_RESET* - received SCSI bus reset
- *CAM_REQ_CMP_ERR* - "impossible" SCSI phase occurred or something else as weird or just a generic error if further detail is not available
- *CAM_UNEXP_BUSFREE* - unexpected disconnect occurred
- *CAM_BDR_SENT* - BUS DEVICE RESET message was sent to the target
- *CAM_UNREC_HBA_ERROR* - unrecoverable Host Bus Adapter Error
- *CAM_REQ_TOO_BIG* - the request was too large for this controller
- *CAM_REQUEUE_REQ* - this request should be re-queued to preserve transaction ordering. This typically occurs when the SIM recognizes an error that should freeze the queue and must place other queued requests for the target at the sim level back into the XPT queue. Typical cases of such errors are selection timeouts, command timeouts and other like conditions. In such cases the troublesome command returns the status indicating the error, the and the other commands which have not be sent to the bus yet get re-queued.
- *CAM_LUN_INVALID* - the LUN ID in the request is not supported by the SCSI controller
- *CAM_TID_INVALID* - the target ID in the request is not supported by the SCSI controller

21.7. Timeout Handling

When the timeout for an HCB expires that request should be aborted, just like with an XPT_ABORT request. The only difference is that the returned status of aborted request should be CAM_CMD_TIMEOUT instead of CAM_REQ_ABORTED (that’s why implementation of the abort better be done as a function). But there is one more possible problem: what if the abort request itself will get stuck? In this case the SCSI bus should be reset, just like with an XPT_RESET_BUS request (and the idea about implementing it as a function called from both places applies here too). Also we should reset the whole SCSI bus if a device reset request got stuck. So after all the timeout function would look like:

```
static void
xxx_timeout(void *arg)
{
    struct xxx_hcb *hcb = (struct xxx_hcb *)arg;
    struct xxx_softc *softc;
    struct ccb_hdr *ccb_h;

    softc = hcb->softc;
    ccb_h = &hcb->ccb->ccb_h;

    if(hcb->flags & HCB_BEING_ABORTED
    || ccb_h->func_code == XPT_RESET_DEV) {
        xxx_reset_bus(softc);
    } else {
        xxx_abort_ccb(hcb->ccb, CAM_CMD_TIMEOUT);
    }
}
```

When we abort a request all the other disconnected requests to the same target/LUN get aborted too. So there appears a question, should we return them with status CAM_REQ_ABORTED or CAM_CMD_TIMEOUT ? The current drivers use CAM_CMD_TIMEOUT. This seems logical because if one request got timed out then probably something really bad is happening to the device, so if they would not be disturbed they would time out by themselves.

Chapter 22. USB Devices

This chapter was written by Nick Hibma <`n_hibma@FreeBSD.org`>. Modifications made for the handbook by Murray Stokely <`murray@FreeBSD.org`>.

22.1. Introduction

The Universal Serial Bus (USB) is a new way of attaching devices to personal computers. The bus architecture features two-way communication and has been developed as a response to devices becoming smarter and requiring more interaction with the host. USB support is included in all current PC chipsets and is therefore available in all recently built PCs. Apple's introduction of the USB-only iMac has been a major incentive for hardware manufacturers to produce USB versions of their devices. The future PC specifications specify that all legacy connectors on PCs should be replaced by one or more USB connectors, providing generic plug and play capabilities. Support for USB hardware was available at a very early stage in NetBSD and was developed by Lennart Augustsson for the NetBSD project. The code has been ported to FreeBSD and we are currently maintaining a shared code base. For the implementation of the USB subsystem a number of features of USB are important.

Lennart Augustsson has done most of the implementation of the USB support for the NetBSD project. Many thanks for this incredible amount of work. Many thanks also to Ardy and Dirk for their comments and proofreading of this paper.

- Devices connect to ports on the computer directly or on devices called hubs, forming a treelike device structure.
- The devices can be connected and disconnected at run time.
- Devices can suspend themselves and trigger resumes of the host system
- As the devices can be powered from the bus, the host software has to keep track of power budgets for each hub.
- Different quality of service requirements by the different device types together with the maximum of 126 devices that can be connected to the same bus, require proper scheduling of transfers on the shared bus to take full advantage of the 12Mbps bandwidth available. (over 400Mbps with USB 2.0)
- Devices are intelligent and contain easily accessible information about themselves

The development of drivers for the USB subsystem and devices connected to it is supported by the specifications that have been developed and will be developed. These specifications are publicly available from the USB home pages. Apple has been very strong in pushing for standards based drivers, by making drivers for the generic classes available in their operating system MacOS and discouraging the use of separate drivers for each new device. This chapter tries to collate essential information for a basic understanding of the present implementation of the USB stack in FreeBSD/NetBSD. It is recommended however to read it together with the relevant specifications mentioned in the references below.

22.1.1. Structure of the USB Stack

The USB support in FreeBSD can be split into three layers. The lowest layer contains the host controller driver, providing a generic interface to the hardware and its scheduling facilities. It supports initialisation of the hardware, scheduling of transfers and handling of completed and/or failed transfers. Each host controller driver implements a virtual hub providing hardware independent access to the registers controlling the root ports on the back of the machine.

The middle layer handles the device connection and disconnection, basic initialisation of the device, driver selection, the communication channels (pipes) and does resource management. This services layer also controls the default pipes and the device requests transferred over them.

The top layer contains the individual drivers supporting specific (classes of) devices. These drivers implement the protocol that is used over the pipes other than the default pipe. They also implement additional functionality to make the device available to other parts of the kernel oruserland. They use the USB driver interface (USBDI) exposed by the services layer.

22.2. Host Controllers

The host controller (HC) controls the transmission of packets on the bus. Frames of 1 millisecond are used. At the start of each frame the host controller generates a Start of Frame (SOF) packet.

The SOF packet is used to synchronise to the start of the frame and to keep track of the frame number. Within each frame packets are transferred, either from host to device (out) or from device to host (in). Transfers are always initiated by the host (polled transfers). Therefore there can only be one host per USB bus. Each transfer of a packet has a status stage in which the recipient of the data can return either ACK (acknowledge reception), NAK (retry), STALL (error condition) or nothing (garbled data stage, device not available or disconnected). Section 8.5 of the USB specification (http://www.usb.org/developers/docs.html) explains the details of packets in more detail. Four different types of transfers can occur on a USB bus: control, bulk, interrupt and isochronous. The types of transfers and their characteristics are described below (‘Pipes’ subsection).

Large transfers between the device on the USB bus and the device driver are split up into multiple packets by the host controller or the HC driver.

Device requests (control transfers) to the default endpoints are special. They consist of two or three phases: SETUP, DATA (optional) and STATUS. The set-up packet is sent to the device. If there is a data phase, the direction of the data packet(s) is given in the set-up packet. The direction in the status phase is the opposite of the direction during the data phase, or IN if there was no data phase. The host controller hardware also provides registers with the current status of the root ports and the changes that have occurred since the last reset of the status change register. Access to these registers is provided through a virtualised hub as suggested in the USB specification [2]. Thevirtual hub must comply with the hub device class given in chapter 11 of that specification. It must provide a default pipe through which device requests can be sent to it. It returns the standard andhub class specific set of descriptors. It should also provide an interrupt pipe that reports changes happening at its ports. There are currently two specifications for host controllers available: Universal Host Controller Interface (http://developer.intel.com/design/USB/UHCI11D.htm) (UHCI; Intel) and Open Host Controller Interface (http://www.compaq.com/productinfo/development/openhci.html) (OHCI; Compaq, Microsoft, National Semiconductor). The UHCI specification has been designed to reduce hardware complexity byrequiring the host controller driver to supply a complete schedule of the transfers for each frame. OHCI type controllers are much more independent by providing a more abstract interface doing alot of work themselves.

22.2.1. UHCI

The UHCI host controller maintains a framelist with 1024 pointers to per frame data structures. It understands two different data types: transfer descriptors (TD) and queue heads (QH). Each TD represents a packet to be communicated to or from a device endpoint. QHs are a means to groupTDs (and QHs) together.

Each transfer consists of one or more packets. The UHCI driver splits large transfers into multiple packets. For every transfer, apart from isochronous transfers, a QH is allocated. For every type of transfer these QHs are collected at a QH for that type. Isochronous transfers have to be executed first because of the fixed latency requirement and are directly referred to by the pointer in the framelist. The last isochronous TD refers to the QH for interrupt transfers for that frame. All QHs for interrupt transfers point at the QH for control transfers, which in turn points at the QH for bulk transfers. The following diagram gives a graphical overview of this:

This results in the following schedule being run in each frame. After fetching the pointer for the current frame from the framelist the controller first executes the TDs for all the isochronous packets in that frame. The last of these TDs refers to the QH for the interrupt transfers for thatframe. The host controller will then descend from that QH to the QHs for the individual interrupt transfers. After finishing that queue, the QH for the interrupt transfers will refer the controller to the QH for all control transfers. It will execute all the subqueues scheduled there, followed by all the transfers queued at the bulk QH. To facilitate the handling of finished or failed transfers different types of interrupts are generatedby the hardware at the end of each frame. In the last TD for a transfer the Interrupt-On Completion bit is set by the HC driver to flag an interrupt when the transfer has completed. An error interrupt is flagged if a TD reaches its maximum error count. If the short packet detect bit is set in a TD and less than the set packet length is transferred this interrupt is flagged to notify the controller driver of the completed transfer. It is the host controller driver’s task to find out which transfer has completed or produced an error. When called the interrupt service routine will locate all the finished transfers and call their callbacks.

See for a more elaborate description the UHCI specification. (http://developer.intel.com/design/USB/UHCI11D.htm)

22.2.2. OHCI

Programming an OHCI host controller is much simpler. The controller assumes that a set of endpoints is available, and is aware of scheduling priorities and the ordering of the types of transfers in a frame. The main data structure used by the host controller is the endpoint descriptor (ED) to which aqueue of transfer descriptors (TDs) is attached.

The ED contains the maximum packet size allowed for an endpoint and the controller hardware does the splitting into packets. The pointers to the data buffers are updated after each transfer and when the start and end pointer are equal, the TD is retired to the done-queue. The four types of endpoints have their own queues. Control and bulk endpoints are queued each at their own queue. Interrupt EDs are queued in a tree, with the level in the tree defining the frequency at which they run.

framelist interruptisochronous control bulk

The schedule being run by the host controller in each frame looks as follows. The controller will first run the non-periodic control and bulk queues, up to a time limit set by the HC driver. Then the interrupt transfers for that frame number are run, by using the lower five bits of the frame number as an index into level 0 of the tree of interrupts EDs. At the end of this tree the isochronous EDs are connected and these are traversed subsequently. The isochronous TDs contain the frame number of the first frame the transfer should be run in. After all the periodic transfers have been run, the control and bulk queues are traversed again. Periodically the interrupt service routine is called to process the done queue and call the callbacks for each transfer and reschedule interrupt and isochronous endpoints.

See for a more elaborate description the OHCI specification (http://www.compaq.com/productinfo/development/openhci.html). Services layer The middle layer provides access to the device in a controlled way and maintains resources inuse by the different drivers and the services layer. The layer takes care of the following aspects:

- The device configuration information
- The pipes to communicate with a device
- Probing and attaching and detaching form a device.

22.3. USB Device Information

22.3.1. Device configuration information

Each device provides different levels of configuration information. Each device has one or more configurations, of which one is selected during probe/attach. A configuration provides power and bandwidth requirements. Within each configuration there can be multiple interfaces. A device interface is a collection of endpoints. For example USB speakers can have an interface for the audio data (Audio Class) and an interface for the knobs, dials and buttons (HID Class). All interfaces in a configuration areactive at the same time and can be attached to by different drivers. Each interface can have alternates, providing different quality of service parameters. In for example cameras this is used to provide different frame sizes and numbers of frames per second.

Within each interface 0 or more endpoints can be specified. Endpoints are the unidirectional access points for communicating with a device. They provide buffers to temporarily store incoming or outgoing data from the device. Each endpoint has a unique address within a configuration, the endpoint's number plus its direction. The default endpoint, endpoint 0, is not part of any interface and available in all configurations. It is managed by the services layer and not directly available to device drivers.

Level 0 Level 1 Level 2 Slot 0

Slot 3 Slot 2 Slot 1

(Only 4 out of 32 slots shown)

This hierarchical configuration information is described in the device by a standard set of descriptors (see section 9.6 of the USB specification [2]). They can be requested through the Get Descriptor Request. The services layer caches these descriptors to avoid unnecessary transferson the USB bus. Access to the descriptors is provided through function calls.

- Device descriptors: General information about the device, like Vendor, Product and Revision Id, supported device class, subclass and protocol if applicable, maximum packet size for the default endpoint, etc.
- Configuration descriptors: The number of interfaces in this configuration, suspend and resume functionality supported and power requirements.
- Interface descriptors: interface class, subclass and protocol if applicable, number of alternate settings for the interface and the number of endpoints.
- Endpoint descriptors: Endpoint address, direction and type, maximum packet size supported and polling frequency if type is interrupt endpoint. There is no descriptor for thedefault endpoint (endpoint 0) and it is never counted in an interface descriptor.
- String descriptors: In the other descriptors string indices are supplied for some fields.These can be used to retrieve descriptive strings, possibly in multiple languages.

Class specifications can add their own descriptor types that are available through the GetDescriptor Request.

Pipes Communication to end points on a device flows through so-called pipes. Drivers submit transfers to endpoints to a pipe and provide a callback to be called on completion or failure of the transfer (asynchronous transfers) or wait for completion (synchronous transfer). Transfers to an endpoint are serialised in the pipe. A transfer can either complete, fail or time-out (if a time-out has been set). There are two types of time-outs for transfers. Time-outs can happen due to time-out on the USBbus (milliseconds). These time-outs are seen as failures and can be due to disconnection of the device. A second form of time-out is implemented in software and is triggered when a transfer does not complete within a specified amount of time (seconds). These are caused by a device acknowledging negatively (NAK) the transferred packets. The cause for this is the device not being ready to receive data, buffer under- or overrun or protocol errors.

If a transfer over a pipe is larger than the maximum packet size specified in the associated endpoint descriptor, the host controller (OHCI) or the HC driver (UHCI) will split the transfer into packets of maximum packet size, with the last packet possibly smaller than the maximum packetsize.

Sometimes it is not a problem for a device to return less data than requested. For example abulk-in-transfer to a modem might request 200 bytes of data, but the modem has only 5 bytes available at that time. The driver can set the short packet (SPD) flag. It allows the host controller to accept a packet even if the amount of data transferred is less than requested. This flag is only valid for in-transfers, as the amount of data to be sent to a device is always known beforehand. If an unrecoverable error occurs in a device during a transfer the pipe is stalled. Before any more data is accepted or sent the driver needs to resolve the cause of the stall and clear the endpoint stall condition through send the clear endpoint halt device request over the default pipe. The default endpoint should never stall.

There are four different types of endpoints and corresponding pipes: - Control pipe / default pipe: There is one control pipe per device, connected to the default endpoint (endpoint 0). The pipe carries the device requests and associated data. The difference between transfers over the default pipe and other pipes is that the protocol for thetransfers is described in the USB specification [2]. These requests are used to reset and configure the device. A basic set of commands that must be supported by each device is provided in chapter 9 of the USB specification [2]. The commands supported on this pipe canbe extended by a device class specification to support additional functionality.

- Bulk pipe: This is the USB equivalent to a raw transmission medium.
- Interrupt pipe: The host sends a request for data to the device and if the device has nothing to send, it will NAK the data packet. Interrupt transfers are scheduled at a frequency specifiedwhen creating the pipe.
- Isochronous pipe: These pipes are intended for isochronous data, for example video oraudio streams, with fixed latency, but no guaranteed delivery. Some support for pipes of this type is available in the current implementation. Packets in control, bulk and interrupt transfers are retried if an error occurs during transmission or the device acknowledges the packet negatively (NAK) due to for example lack of buffer space to store the incoming data. Isochronous packets are however not retried in case of failed delivery or NAK of a packet as this might violate the timing constraints.

The availability of the necessary bandwidth is calculated during the creation of the pipe. Transfersare scheduled within frames of 1 millisecond. The bandwidth allocation within a frame is prescribed by the USB specification, section 5.6 [2]. Isochronous and interrupt transfers areallowed to consume up to 90% of the bandwidth within a frame. Packets for control and bulk transfers are scheduled after all isochronous and interrupt packets and will consume all the remaining bandwidth.

More information on scheduling of transfers and bandwidth reclamation can be found in chapter 5of the USB specification [2], section 1.3 of the UHCI specification [3] and section 3.4.2 of the OHCI specification [4].

22.4. Device probe and attach

After the notification by the hub that a new device has been connected, the service layer switcheson the port, providing the device with 100 mA of current. At this point the device is in its default state and listening to device address 0. The services layer will proceed to retrieve the various descriptors through the default pipe. After that it will send a Set Address request to move the device away from the default device address (address 0). Multiple device drivers might be able to support the device. For example a modem driver might beable to support an ISDN TA through the AT compatibility interface. A driver for that specific model of the ISDN adapter might however be able to provide much better support for this device. To support this flexibility, the probes return priorities indicating their level of support. Support for a specific revision of a product ranks the highest and the generic driver the lowest priority. It might also be that multiple drivers could attach to one device if there are multiple interfaceswithin one configuration. Each driver only needs to support a subset of the interfaces.

The probing for a driver for a newly attached device checks first for device specific drivers. If notfound, the probe code iterates over all supported configurations until a driver attaches in a configuration. To support devices with multiple drivers on different interfaces, the probe iteratesover all interfaces in a configuration that have not yet been claimed by a driver. Configurations that exceed the power budget for the hub are ignored. During attach the driver should initialise the device to its proper state, but not reset it, as this will make the device disconnect itself from the bus and restart the probing process for it. To avoid consuming unnecessary bandwidth should not claim the interrupt pipe at attach time, but should postpone allocating the pipe until the file is opened and the data is actually used. When the file is closed the pipe should be closed again, eventhough the device might still be attached.

22.4.1. Device disconnect and detach

A device driver should expect to receive errors during any transaction with the device. The designof USB supports and encourages the disconnection of devices at any point in time. Drivers should make sure that they do the right thing when the device disappears.

Furthermore a device that has been disconnected and reconnected will not be reattached at the same device instance. This might change in the future when more devices support serial numbers (see the device descriptor) or other means of defining an identity for a device have been developed.

The disconnection of a device is signalled by a hub in the interrupt packet delivered to the hub driver. The status change information indicates which port has seen a connection change. The device detach method for all device drivers for the device connected on that port are called and the structures cleaned up. If the port status indicates that in the mean time a device has been connected to that port, the procedure for probing and attaching the device will be started. A device reset will produce a disconnect-connect sequence on the hub and will be handled as described above.

22.5. USB Drivers Protocol Information

The protocol used over pipes other than the default pipe is undefined by the USB specification. Information on this can be found from various sources. The most accurate source is the developer’s section on the USB home pages [1].

From these pages a growing number of deviceclass specifications are available. These specifications specify what a compliant device should look like from a driver perspective, basic functionality it needs to provide and the protocol that is to be used over the communication channels. The USB specification [2] includes the description of the Hub Class. A class specification for Human Interface Devices (HID) has been created to cater for keyboards, tablets, bar-code readers, buttons, knobs, switches, etc. A third example is the class specification for mass storage devices. For a full list of device classes see the developers sectionon the USB home pages [1].

For many devices the protocol information has not yet been published however. Information on the protocol being used might be available from the company making the device. Some companies will require you to sign a Non -Disclosure Agreement (NDA) before giving you the specifications. This in most cases precludes making the driver open source.

Another good source of information is the Linux driver sources, as a number of companies have started to provide drivers for Linux for their devices. It is always a good idea to contact the authors of those drivers for their source of information.

Example: Human Interface Devices The specification for the Human Interface Devices like keyboards, mice, tablets, buttons, dials,etc. is referred to in other device class specifications and is used in many devices.

For example audio speakers provide endpoints to the digital to analogue converters and possibly an extra pipe for a microphone. They also provide a HID endpoint in a separate interface for the buttons and dials on the front of the device. The same is true for the monitor control class. It is straightforward to build support for these interfaces through the available kernel and userland libraries together with the HID class driver or the generic driver. Another device that serves as an example for interfaces within one configuration driven by different device drivers is a cheap keyboard with built-in legacy mouse port. To avoid having the cost of including the hardware for a USB hub in the device, manufacturers combined the mouse data received from the PS/2 port on the back of the keyboard and the keypresses from the keyboard into two separate interfaces in the same configuration. The mouse and keyboard drivers each attach to the appropriate interface and allocate the pipes to the two independent endpoints.

Example: Firmware download Many devices that have been developed are based on a general purpose processor with anadditional USB core added to it. Because the development of drivers and firmware for USB devices is still very new, many devices require the downloading of the firmware after they have been connected.

The procedure followed is straightforward. The device identifies itself through a vendor and product Id. The first driver probes and attaches to it and downloads the firmware into it. After that the device soft resets itself and the driver is detached. After a short pause the devicere announces its presence on the bus. The device will have changed its vendor/product/revision Id to reflect the fact that it has been supplied with firmware and as a consequence a second driver will probe it and attach to it.

An example of these types of devices is the ActiveWire I/O board, based on the EZ-USB chip. For this chip a generic firmware downloader is available. The firmware downloaded into the ActiveWire board changes the revision Id. It will then perform a soft reset of the USB part of the EZ-USB chip to disconnect from the USB bus and again reconnect.

Example: Mass Storage Devices Support for mass storage devices is mainly built around existing protocols. The

Iomega USB Zipdrive is based on the SCSI version of their drive. The SCSI commands and status messages are wrapped in blocks and transferred over the bulk pipes to and from the device, emulating a SCSI controller over the USB wire. ATAPI and UFI commands are supported in a similar fashion.

The Mass Storage Specification supports 2 different types of wrapping of the command block.The initial attempt was based on sending the command and status through the default pipe and using bulk transfers for the data to be moved between the host and the device. Based on experience a second approach was designed that was based on wrapping the command and status blocks and sending them over the bulk out and in endpoint. The specification specifies exactly what has to happen when and what has to be done in case an error condition is encountered. The biggest challenge when writing drivers for these devices is to fit USB based protocol into theexisting support for mass storage devices. CAM provides hooks to do this in a fairly straight forward way. ATAPI is less simple as historically the IDE interface has never had many different appearances.

The support for the USB floppy from Y-E Data is again less straightforward as a new command set has been designed.

Chapter 23. NewBus

This chapter will talk about the FreeBSD NewBus architecture.

XII. Architectures

Chapter 24. x86 Assembly Language Programming

This chapter written by G. Adam Stanislav. Whiz Kid Technomagic (http://www.whizkidtech.net/) Modifications for the Handbook made by Wylie Stilwell <`wylie@osd.bsdi.com`>, &.logo;, and Murray Stokely <`murray@FreeBSD.org`>

24.1. Synopsis

Assembly language programing under Unix is highly undocumented. It is generally assumed that no one would ever want to use it because various Unix systems run on different microprocessors, so everything should be written in C for portability.

In reality, C portability is quite a myth. Even C programs need to be modified when ported from one Unix to another, regardless of what processor each runs on. Typically, such a program is full of conditional statements depending on the system it is compiled for.

Even if we believe that all of Unix software should be written in C, or some other high-level language, we still need assembly language programmers: Who else would write the section of C library that accesses the kernel?

In this tutorial, which is quite brief at this time, I will attempt to show you how you can use assembly language writing Unix programs, specifically under FreeBSD. I hope to turn it into a complete course of FreeBSD assembly language eventually.

This tutorial does not explain the basics of assembly language. There are enough resources about that (for a complete online course in assembly language, see Randall Hyde's Art of Assembly Language (http://webster.cs.ucr.edu/); or if you prefer a printed book, take a look at Jeff Duntemann's Assembly Language Step-by-Step. However, once the tutorial is finished, any assembly language programmer will be able to write programs for FreeBSD quickly and efficiently.

24.2. The Tools

24.2.1. The Assembler

The most important tool for assembly language programming is the assembler, the software that converts assembly language code into machine language.

Two very different assemblers are available for FreeBSD. One is as(1), which uses the traditional Unix assembly language syntax. It comes with the system.

The other is `/usr/ports/devel/nasm`. It uses the Intel syntax. Its main advantage is that it can assemble code for many operating systems. It needs to be installed separately, but is completely free.

This tutorial uses *nasm* syntax because most assembly language programmers coming to FreeBSD from other operating systems will find it easier to understand. And, because, quite frankly, that is what I am used to.

24.2.2. The Linker

The output of the assembler, like that of any compiler, needs to be linked to form an executable file.

The standard ld(1) linker comes with FreeBSD. It works with the code assembled with either assembler.

24.3. System Calls

24.3.1. Default Calling Convention

By default, the FreeBSD kernel uses the C calling convention. Further, although the kernel is accessed using int 80h, it is assumed the program will call a function that issues int 80h, rather than issuing int 80h directly.

This convention is very convenient, and quite superior to the Microsoft convention used by MS DOS. Why? Because the Unix convention allows any program written in any language to access the kernel.

An assembly language program can do that as well. For example, we could open a file:

```
kernel:
 int 80h ; Call kernel
 ret

open:
 push dword mode
 push dword flags
 push dword path
 mov eax, 5
 call kernel
 add esp, byte 12
 ret
```

This is a very clean and portable way of coding. If you need to port the code to a Unix system which uses a different interrupt, or a different way of passing parameters, all you need to change is the kernel procedure.

But assembly language programmers like to shave off cycles. The above example requires a `call/ret` combination. We can eliminate it by pushing an extra dword:

```
open:
 push dword mode
 push dword flags
 push dword path
 mov eax, 5
 push eax ; Or any other dword
 int 80h
 add esp, byte 16
```

The *5* that we have placed in *EAX* identifies the kernel function, in this case *open*.

24.3.2. Alternate Calling Convention

FreeBSD is an extremely flexible system. It offers other ways of calling the kernel. For it to work, however, the system must have Linux emulation installed.

Linux is a Unix-like system. However, its kernel uses the Microsoft system-call convention of passing parameters in registers. As with the Unix convention, the function number is placed in *EAX*. The parameters, however, are not passed on the stack but *EBX, ECX, EDX, ESI, EDI, EBP*:

```
open:
 mov eax, 5
 mov ebx, path
 mov ecx, flags
 mov edx, mode
 int 80h
```

This convention has a great disadvantage over the Unix way, at least as far as assembly language programming is concerned: Every time you make a kernel call you must *push* the registers, then *pop* them later. This makes your code bulkier and slower. Nevertheless, FreeBSD gives you a choice.

If you do choose the Microsoft/Linux convention, you must let the system know about it. After your program is assembled and linked, you need to brand the executable:

```
% brandelf -f Linux filename
```

24.3.3. Which convention should you use?

If you are coding specifically for FreeBSD, you should always use the Unix convention: It is faster, you can store global variables in registers, you do not have to brand the executable, and you do not impose the installation of the Linux emulation package on the target system.

If you want to create portable code that can also run on Linux, you will probably still want to give the FreeBSD users as efficient a code as possible. I will show you how you can accomplish that after I have explained the basics.

24.3.4. Call Numbers

To tell the kernel which system service you are calling, place its number in *EAX*. Of course, you need to know what the number is.

24.3.4.1. The Syscalls File

The numbers are listed in `syscalls`. *locate syscalls* finds this file in several different formats, all produced automatically from `syscalls.master`.

You can find the master file for the default Unix calling convention in `/usr/src/sys/kern/syscalls.master`. If you need to use the Microsoft convention implemented in the Linux emulation mode, read `/usr/src/sys/i386/linux/syscalls.master`.

N.B.: Not only do FreeBSD and Linux use different calling conventions, they sometimes use different numbers for the same functions.

`syscalls.master` describes how the call is to be made:

```
0     STD     NOHIDE         { int nosys(void); } syscall nosys_args int
1     STD     NOHIDE         { void exit(int rval); } exit rexit_args void
2     STD     POSIX         { int fork(void); }
3     STD     POSIX         { ssize_t read(int fd, void *buf, size_t nbyte); }
4     STD     POSIX         { ssize_t write(int fd, const void *buf, size_t nbyte);
5     STD     POSIX         { int open(char *path, int flags, int mode); }
6     STD     POSIX         { int close(int fd); }
etc...
```

It is the leftmost column that tells us the number to place in *EAX*.

The rightmost column tells us what parameters to *push*. They are *push*ed from right to left.

EXAMPLE 3.1: For example, to *open* a file, we need to *push* the *mode* first, then *flags*, then the address at which the *path* is stored.

24.4. Return Values

A system call would not be useful most of the time if it did not return some kind of a value: The file descriptor of an open file, the number of bytes read to a buffer, the system time, etc.

Additionally, the system needs to inform us if an error occurs: A file does not exist, system resources are exhausted, we passed an invalid parameter, etc.

24.4.1. man-pages

The traditional place to look for information about various system calls under Unix systems are the man pages. FreeBSD describes its system calls in section 2, sometimes in section 3.

For example, `open(2)` says:

If successful, `open()` returns a non-negative integer, termed a file descriptor. It returns *-1* on failure, and sets *errno* to indicate the error.

The assembly language programmer new to Unix and FreeBSD will immediately ask the puzzling question: Where is *errno* and how do I get to it?

N.B.: The information presented in the man pages applies to C programs. The assembly language programmer needs additional information.

24.4.2. Where are the return values?

Unfortunately, it depends... For most system calls it is in *EAX*, but not for all. A good rule of thumb, when working with a system call for the first time, look for the return value in *EAX*. If it is not there, you need further research.

N.B.: I am aware of one system call that returns the value in *EDX*: *SYS_fork*. All others I have worked with use *EAX*. But I have not worked with them all yet.

TIP: If you cannot find the answer here or anywhere else, study libc source code and see how it interfaces with the kernel.

24.4.3. Where is `errno`>?

Actually, nowhere...

`errno` is part of the C language, not the Unix kernel. When accessing kernel services directly, the error code is returned in *EAX*, the same register the proper return value generally ends up in.

This makes perfect sense. If there is no error, there is no error code. If there is an error, there is no return value. One register can contain either.

24.4.4. Determining an Error Occurred

When using the standard FreeBSD calling convention, the *carry flag* is cleared upon success, set upon failure.

When using the Linux emulation mode, the signed value in *EAX* is non-negative upon success, and contains the return value. In case of an error, the value is negative, i.e., *-errno*.

24.5. Creating Portable Code

Portability is generally not one of the strengths of assembly language. Yet, writing assembly language programs for different platforms is possible, especially with *nasm*. I have written assembly language libraries that can be assembled for such different operating systems as Windows and FreeBSD.

It is all the more possible when you want your code to run on two platforms which, while different, are based on similar architectures.

For example, FreeBSD is Unix, Linux is Unix-like. I only mentioned three differences between them (from an assembly language programmerâs perspective): The calling convention, the function numbers, and the way of returning values.

24.5.1. Dealing With Function Numbers

In many cases the function numbers are the same. However, even when they are not, the problem is easy to deal with: Instead of using numbers in your code, use constants which you have declared differently depending on the target architecture:

```
%ifdef LINUX
%define SYS_execve 11
%else
%define SYS_execve 59
%endif
```

24.5.2. Dealing With Conventions

Both, the calling convention, and the return value (the *errno* problem) can be resolved with macros:

```
%ifdef LINUX

%macro system 0
 call kernel
%endmacro

align 4
kernel:
 push ebx
 push ecx
 push edx
 push esi
 push edi
 push ebp

 mov ebx, [esp+32]
 mov ecx, [esp+36]
 mov edx, [esp+40]
 mov esi, [esp+44]
 mov ebp, [esp+48]
 int 80h

 pop ebp
 pop edi
 pop esi
 pop edx
 pop ecx
 pop ebx

 or eax, eax
 js .errno
 clc
 ret

.errno:
 neg eax
 stc
 ret

%else
```

```
%macro system 0
 int 80h
%endmacro

%endif
```

24.5.3. Dealing With Other Portability Issues

The above solutions can handle most cases of writing code portable between FreeBSD and Linux. Nevertheless, with some kernel services the differences are deeper.

In that case, you need to write two different handlers for those particular system calls, and use conditional assembly. Luckily, most of your code does something other than calling the kernel, so usually you will only need a few such conditional sections in your code.

24.5.4. Using a Portable Library

You can avoid portability issues in your main code altogether by writing a library of system calls. Create a separate library for FreeBSD, a different one for Linux, and yet other libraries for more operating systems.

In your library, write a separate function (or procedure, if you prefer the traditional assembly language terminology) for each system call. Use the C calling convention of passing parameters. But still use *EAX* to pass the call number in. In that case, your FreeBSD library can be very simple, as many seemingly different functions can be just labels to the same code:

```
sys.open:
sys.close:
[etc...]
 int 80h
 ret
```

Your Linux library will require more different functions. But even here you can group system calls using the same number of parameters:

```
sys.exit:
sys.close:
[etc... one-parameter functions]
 push ebx
 mov ebx, [esp+12]
 int 80h
 pop ebx
 jmp sys.return

...

sys.return:
 or eax, eax
 js sys.err
 clc
 ret

sys.err:
 neg eax
 stc
 ret
```

The library approach may seem inconvenient at first because it requires you to produce a separate file your code depends on. But it has many advantages: For one, you only need to write it once and can use it for all your programs. You can even let other assembly language programmers use it, or perhaps use one written by someone else. But perhaps the greatest advantage of the library is that your code can be ported to other systems, even by other programmers, by simply writing a new library without any changes to your code.

If you do not like the idea of having a library, you can at least place all your system calls in a separate assembly language file and link it with your main program. Here, again, all porters have to do is create a new object file to link with your main program.

24.5.5. Using a Portable Include File

If you are releasing your software as (or with) source code, you can use macros and place them in a separate file, which you include in your code.

Porters of your software will simply write a new include file. No library or external object file is necessary, yet your code is portable without any need to edit the code.

N.B.: This is the approach we will use throughout this tutorial. We will name our include file *system.inc*, and add to it whenever we deal with a new system call.

We can start our `system.inc` by declaring the standard file descriptors:

```
%define    stdin 0
%define    stdout 1
%define    stderr 2
```

Next, we create a symbolic name for each system call:

```
%define         SYS_nosys 0
%define         SYS_exit 1
%define         SYS_fork 2
%define         SYS_read 3
%define         SYS_write 4
; [etc...]
```

We add a short, non-global procedure with a long name, so we do not accidentally reuse the name in our code:

```
section .code
align 4
access.the.bsd.kernel:
     int     80h
     ret
```

We create a macro which takes one argument, the syscall number:

```
%macro system 1
 mov eax, %1
 call access.the.bsd.kernel
%endmacro
```

Finally, we create macros for each syscall. These macros take no arguments.

```
%macro sys.exit 0
 system SYS_exit
%endmacro

%macro sys.fork 0
 system         SYS_fork
%endmacro

%macro sys.read 0
 system   SYS_read
%endmacro

%macro sys.write 0
 system SYS_write
```

```
%endmacro

; [etc...]
```

Go ahead, enter it into your editor and save it as `system.inc`. We will add more to it as we discuss more syscalls.

24.6. Our First Program

We are now ready for our first program, the mandatory *Hello, World!*

```
 1:  %include ’system.inc’
 2:
 3:  section .data
 4:  hello db ’Hello, World!’, 0Ah
 5:  hbytes equ $-hello
 6:
 7:  section .code
 8:  global _start
 9:  _start:
10:  push dword hbytes
11:  push dword hello
12:  push dword stdout
13:  sys.write
14:
15:  push dword 0
16:  sys.exit
```

Here is what it does: Line 1 includes the defines, the macros, and the code from `system.inc`.

Lines 3-5 are the data: Line 3 starts the data section/segment. Line 4 contains the string "Hello, World!" followed by a new line (0Ah). Line 5 creates a constant that contains the length of the string from line 4 in bytes.

Lines 7-16 contain the code. Note that FreeBSD uses the elf file format for its executables, which requires every program to start at the point labeled _start (or, more precisely, the linker expects that). This label has to be global.

Lines 10-13 ask the system to write *hbytes* bytes of the *hello* string to `stdout`.

Lines 15-16 ask the system to end the program with the return value of *0*. The `SYS_exit` syscall never returns, so the code ends there.

N.B.: If you have come to Unix from MS DOS assembly language background, you may be used to writing directly to the video hardware. You will never have to worry about this in FreeBSD, or any other flavor of Unix. As far as you are concerned, you are writing to a file known as `stdout`. This can be the video screen, or a Telnet terminal, or an actual file, or even the input of another program. Which it is, is for the system to figure out.

24.6.1. Assembling the Code

Type the code (except the line numbers) in an editor, and save it in a file named `hello.asm`. You need *nasm* to assemble it.

24.6.1.1. Installing NASM

If you do not have *nasm*, type:

```
% su
Password: your root password
# cd /usr/ports/devel/nasm
# make install
# exit
%
```

```
% su
Password: your root password
# cd /usr/ports/devel/nasm
# make install
# exit
%
```

You may type *make install clean* instead of just *make install* if you do not want to keep *nasm* source code.

Either way, FreeBSD will automatically download *nasm* from the Internet, compile it, and install it on your system.

N.B.: If your system is not FreeBSD, you need to get *nasm* from its home page (http://www.web-sites.co.uk/nasm/). You can still use it to assemble FreeBSD code.

24.6.2. Assemble, link and run

Now you can assemble, link, and run the code:

```
% nasm -f elf hello.asm
% ld -s -o hello hello.o
% ./hello
Hello, World!
%
```

24.7. Writing Unix Filters

A common type of Unix application is a filterâa program that reads data from the *stdin*, processes it somehow, then writes the result to *stdout*.

In this chapter, we shall develop a simple filter, and learn how to read from *stdin* and write to *stdout*. This filter will convert each byte of its input into a hexadecimal number followed by a blank space.

```
       %include 'system.inc'

section .data
hex   db '0123456789ABCDEF'
buffer db 0, 0, ' '

section .code
global _start
_start:
; read a byte from stdin
push dword 1
push dword buffer
push dword stdin
sys.read
add esp, byte 12
or eax, eax
je .done

 ; convert it to hex
movzx eax, byte [buffer]
mov edx, eax
shr dl, 4
mov dl, [hex+edx]
mov [buffer], dl
and al, 0Fh
mov al, [hex+eax]
mov [buffer+1], al

; print it
push dword 3
push dword buffer
push dword stdout
sys.write
add esp, byte 12
jmp short _start

       .done        :
push         dword 0
sys.exit
```

In the data section we create an array called *hex*. It contains the 16 hexadecimal digits in ascending order. The array is followed by a buffer which we will use for both input and output. The first two bytes of the buffer are initially set to *0*. This is where we will write the two hexadecimal digits (the first byte also is where we will read the input). The third byte is a space.

The code section consists of four parts: Reading the byte, converting it to a hexadecimal number, writing the result, and eventually exiting the program.

To read the byte, we ask the system to read one byte from *stdin*, and store it in the first byte of the *buffer*. The system returns the number of bytes read in *EAX*. This will be *1* while data is coming, or *0*, when no more input data is available. Therefore, we check the value of *EAX*. If it is *0*, we jump to *.done*, otherwise we continue.

N.B.: For simplicity sake, we are ignoring the possibility of an error condition at this time.

The hexadecimal conversion reads the byte from the *buffer* into *EAX*, or actually just *AL*, while clearing the remaining bits of *EAX* to zeros. We also copy the byte to *EDX* because we need to convert the upper four bits (nibble) separately from the lower four bits. We store the result in the first two bytes of the buffer.

Next, we ask the system to write the three bytes of the buffer, i.e., the two hexadecimal digits and the blank space, to *stdout*. We then jump back to the beginning of the program and process the next byte.

Once there is no more input left, we ask the system to exit our program, returning a zero, which is the traditional value meaning the program was successful.

Go ahead, and save the code in a file named `hex.asm`, then type the following (the *^D* means press the control key and type *D* while holding the control key down):

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
Here I come!
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D
%
```

N.B.: If you are migrating to Unix from MS DOS, you may be wondering why each line ends with *0A* instead of *0D 0A*. This is because Unix does not use the cr/lf convention, but a ânew lineâ convention, which is *0A* in hexadecimal.

Can we improve this? Well, for one, it is a bit confusing because once we have converted a line of text, our input no longer starts at the begining of the line. We can modify it to print a new line instead of a space after each *0A*:

```
%include ’system.inc’
```

```
section .data
hex db ’0123456789ABCDEF’
buffer db 0, 0, ’ ’

section .code
global _start
_start:
 mov cl, ’ ’

.loop:
 ; read a byte from stdin
 push dword 1
 push dword buffer
 push dword stdin
 sys.read
 add esp, byte 12
 or eax, eax
 je .done

 ; convert it to hex
 movzx eax, byte [buffer]
 mov [buffer+2], cl
 cmp al, 0Ah
 jne .hex
 mov [buffer+2], al

.hex:
 mov edx, eax
 shr dl, 4
 mov dl, [hex+edx]
 mov [buffer], dl
 and al, 0Fh
 mov al, [hex+eax]
 mov [buffer+1], al

 ; print it
 push dword 3
 push dword buffer
 push dword stdout
 sys.write
 add esp, byte 12
 jmp short .loop

.done:
	push	dword 0
	sys.exit
```

We have stored the space in the *CL* register. We can do this safely because, unlike Microsoft Windows, Unix system calls do not modify the value of any register they do not use to return a value in.

That means we only need to set *CL* once. We have, therefore, added a new label *.loop* and jump to it for the next byte instead of jumping at *_start*. We have also added the *.hex* label so we can either have a blank space or a new line as the third byte of the *buffer*.

Once you have changed *hex.asm* to reflect these changes, type:

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
Here I come!
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D
%
```

That looks better. But this code is quite inefficient! We are making a system call for every single byte twice (once to read it, another time to write the output).

24.8. Buffered Input and Output

We can improve the efficiency of our code by buffering our input and output. We create an input buffer and read a whole sequence of bytes at one time. Then we fetch them one by one from the buffer.

We also create an output buffer. We store our output in it until it is full. At that time we ask the kernel to write the contents of the buffer to *stdout*.

The program ends when there is no more input. But we still need to ask the kernel to write the contents of our output buffer to *stdout* one last time, otherwise some of our output would make it to the output buffer, but never be sent out. Do not forget that, or you will be wondering why some of your output is missing.

```
%include 'system.inc'

%define	BUFSIZE	2048

section	.data
hex	db	'0123456789ABCDEF'
```

```
section .bss
ibuffer resb BUFSIZE
obuffer resb BUFSIZE

section .code
global _start
_start:
 sub eax, eax
 sub ebx, ebx
 sub ecx, ecx
 mov edi, obuffer

.loop:
 ; read a byte from stdin
 call getchar

 ; convert it to hex
 mov dl, al
 shr al, 4
 mov al, [hex+eax]
 call putchar

 mov al, dl
 and al, 0Fh
 mov al, [hex+eax]
 call putchar

 mov al, ' '
 cmp dl, 0Ah
 jne .put
 mov al, dl

.put:
 call putchar
 jmp short .loop

align 4
getchar:
 or ebx, ebx
 jne .fetch

 call read

.fetch:
 lodsb
 dec ebx
 ret

read:
 push dword BUFSIZE
 mov esi, ibuffer
 push esi
 push dword stdin
 sys.read
 add esp, byte 12
 mov ebx, eax
 or eax, eax
 je .done
 sub eax, eax
 ret

align 4
.done:
 call write ; flush output buffer
 push dword 0
 sys.exit

align 4
putchar:
 stosb
 inc ecx
 cmp ecx, BUFSIZE
 je write
 ret

align 4
write:
 sub edi, ecx ; start of buffer
 push ecx
 push edi
 push dword stdout
 sys.write
 add esp, byte 12
 sub eax, eax
 sub ecx, ecx ; buffer is empty now
 ret
```

We now have a third section in the source code, named *.bss*. This section is not included in our executable file, and, therefore, cannot be initialized. We use *resb* instead of *db*. It simply reserves the requested size of uninitialized memory for our use.

We take advantage of the fact that the system does not modify the registers: We use registers for what, otherwise, would have to be global variables stored in the *.data* section. This is also why the Unix convention of passing parameters to system calls on the stack is superior to the Microsoft convention of passing them in the registers: We can keep the registers for our own use.

We use *EDI* and *ESI* as pointers to the next byte to be read from or written to. We use *EBX* and *ECX* to keep count of the number of bytes in the two buffers, so we know when to dump the output to, or read more input from, the system.

Let us see how it works now:

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
Here I come!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D
%
```

Not what you expected? The program did not print the output until we pressed *^D*. That is easy to fix by inserting three lines of code to write the output every time we have converted a new line to *0A*. I have marked the three lines with > (do not copy the > in your `hex.asm`

```
%include 'system.inc'

%define BUFSIZE 2048

section .data
hex db '0123456789ABCDEF'

section .bss
ibuffer resb BUFSIZE
obuffer resb BUFSIZE

section .code
global _start
_start:
 sub eax, eax
 sub ebx, ebx
 sub ecx, ecx
 mov edi, obuffer

.loop:
 ; read a byte from stdin
 call getchar

 ; convert it to hex
 mov dl, al
 shr al, 4
 mov al, [hex+eax]
 call putchar

 mov al, dl
 and al, 0Fh
 mov al, [hex+eax]
 call putchar

 mov al, ' '
 cmp dl, 0Ah
 jne .put
 mov al, dl

.put:
 call putchar
> cmp al, 0Ah
> jne .loop
> call write
 jmp short .loop

align 4
getchar:
 or ebx, ebx
 jne .fetch

 call read

.fetch:
 lodsb
 dec ebx
 ret

read:
 push dword BUFSIZE
 mov esi, ibuffer
 push esi
 push dword stdin
 sys.read
 add esp, byte 12
 mov ebx, eax
 or eax, eax
 je .done
 sub eax, eax
 ret

align 4
.done:
 call write ; flush output buffer
 push dword 0
 sys.exit

align 4
putchar:
 stosb
 inc ecx
 cmp ecx, BUFSIZE
 je write
 ret

align 4
write:
 sub edi, ecx ; start of buffer
 push ecx
 push edi
 push dword stdout
 sys.write
 add esp, byte 12
 sub eax, eax
 sub ecx, ecx ; buffer is empty now
 ret
```

Now, let us see how it works:

```
% nasm -f elf hex.asm
% ld -s -o hex hex.o
% ./hex
Hello, World!
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 0A
Here I come!
48 65 72 65 20 49 20 63 6F 6D 65 21 0A
^D
%
```

Not bad for a 644-byte executable, is it!

24.8.1. How to Unread a Character

WARNING: This may be a somewhat advanced topic, mostly of interest to programmers familiar with the theory of compilers. If you wish, you may skip to the next chapter (#command-line), and perhaps read this later.

While our sample program does not require it, more sophisticated filters often need to look ahead. In other words, they may need to see what the next character is (or even several characters). If the next character is of a certain value, it is part of the token currently being processed. Otherwise, it is not.

For example, you may be parsing the input stream for a textual string (e.g., when implementing a language compiler): If a character is followed by another character, or perhaps a digit, it is part of the token you are processing. If it is followed by white space, or some other value, then it is not part of the current token.

This presents an interesting problem: How to return the next character back to the input stream, so it can be read again later?

One possible solution is to store it in a character variable, then set a flag. We can modify `getchar` to check the flag, and if it is set, fetch the byte from that variable instead of the input buffer, and reset the flag. But, of course, that slows us down.

The C language has an `ungetc()` function, just for that purpose. Is there a quick way to implement it in our code? I would like you to scroll back up and take a look at the `getchar` procedure and see if you can find a nice and fast solution before reading the next paragraph. Then come back here and see my own solution.

The key to returning a character back to the stream is in how we are getting the characters to start with:

First we check if the buffer is empty by testing the value of *EBX*. If it is zero, we call the *read* procedure.

If we do have a character available, we use *lodsb*, then decrease the value of *EBX*. The *lodsb* instruction is effectively identical to:

```
mov al, [esi]
inc esi
```

The byte we have fetched remains in the buffer until the next time `read` is called. We do not know when that happens, but we do know it will not happen until the next call to `getchar`. Hence, to âreturnâ the last-read byte back to the stream, all we have to do is decrease the value of *ESI* and increase the value of *EBX*:

```
ungetc:
 dec esi
 inc ebx
 ret
```

But, be careful! We are perfectly safe doing this if our look-ahead is at most one character at a time. If we are examining more than one upcoming character and call `ungetc` several times in a row, it will work most of the time, but not all the time (and will be tough to debug). Why?

Because as long as `getchar` does not have to call `read` all of the pre-read bytes are still in the buffer, and our `ungetc` works without a glitch. But the moment `getchar` calls `read`, the contents of the buffer change.

We can always rely on `ungetc` working properly on the last character we have read with `getchar`, but not on anything we have read before that.

If your program reads more than one byte ahead, you have at least two choices:

If possible, modify the program so it only reads one byte ahead. This is the simplest solution.

If that option is not available, first of all determine the maximum number of characters your program needs to return to the input stream at one time. Increase that number slightly, just to be sure, preferably to a multiple of 16âso it aligns nicely. Then modify the *.bss* section of your code, and create a small âspareâ buffer right before your input buffer, something like this:

```
section .bss
 resb 16 ; or whatever the value you came up with
ibuffer resb BUFSIZE
obuffer resb BUFSIZE
```

You also need to modify your `ungetc` to pass the value of the byte to unget in *AL*:

```
ungetc:
 dec esi
 inc ebx
 mov [esi], al
 ret
```

With this modification, you can call `ungetc` up to 17 times in a row safely (the first call will still be within the buffer, the remaining 16 may be either within the buffer or within the âspareâ).

24.9. Command-line Arguments

Our *hex* program will be more useful if it can read the names of an input and output file from its command line, i.e., if it can process the command line arguments. But... Where are they?

Before a Unix system starts a program, it pushes some data on the stack, then jumps at the *_start* label of the program. Yes, I said jumps, not calls. That means the data can be accessed by reading *[esp+offset]*, or by simply *pop*ping it.

The value at the top of the stack contains the number of command line arguments. It is traditionally called *argc*, for âargument count.â

Command line arguments follow next, all *argc* of them. These are typically referred to as *argv*, for âargument value(s).â That is, we get *argv[0]*, *argv[1]*, ..., *argv[argc-1]*. These are not the actual arguments, but pointers to arguments, i.e., memory addresses of the actual arguments. The arguments themselves are NUL-terminated character strings.

The *argv* list is followed by a *NULL* pointer, which is simply a *0*. There is more, but this is enough for our purposes right now.

N.B.: If you have come from the MS DOS programming environment, the main difference is that each argument is in a separate string. The second difference is that there is no practical limit on how many arguments there can be.

Armed with this knowledge, we are almost ready for the next version of `hex.asm`. First, however, we need to add a few lines to `system.inc`:

First, we need to add two new entries to our list of system call numbers:

```
%define SYS_open 5
%define SYS_close 6
```

Then we add two new macros at the end of the file:

```
%macro sys.open 0
 system SYS_open
%endmacro

%macro sys.close 0
 system SYS_close
%endmacro
```

Here, then, is our modified source code:

```
%include 'system.inc'

%define BUFSIZE 2048

section .data
fd.in dd stdin
fd.out dd stdout
hex db '0123456789ABCDEF'

section .bss
ibuffer resb BUFSIZE
obuffer resb BUFSIZE

section .code
align 4
```

```
err:
 push dword 1 ; return failure
 sys.exit

align 4
global _start
_start:
 add esp, byte 8 ; discard argc and argv[0]

 pop ecx
 jecxz .init ; no more arguments

 ; ECX contains the path to input file
 push dword 0 ; O_RDONLY
 push ecx
 sys.open
 jc err ; open failed

 add esp, byte 8
 mov [fd.in], eax

 pop ecx
 jecxz .init ; no more arguments

 ; ECX contains the path to output file
 push dword 420 ; file mode (644 octal)
 push dword 0200h | 0400h | 01h
 ; O_CREAT | O_TRUNC | O_WRONLY
 push ecx
 sys.open
 jc err

 add esp, byte 12
 mov [fd.out], eax

.init:
 sub eax, eax
 sub ebx, ebx
 sub ecx, ecx
 mov edi, obuffer

.loop:
 ; read a byte from input file or stdin
 call getchar
 ; convert it to hex
 mov dl, al
 shr al, 4
 mov al, [hex+eax]
 call putchar

 mov al, dl
 and al, 0Fh
 mov al, [hex+eax]
 call putchar

 mov al, ’ ’
 cmp dl, 0Ah
 jne .put
 mov al, dl

.put:
 call putchar
 cmp al, dl
 jne .loop
 call write
 jmp short .loop

align 4
getchar:
 or ebx, ebx
 jne .fetch

 call read

.fetch:
 lodsb
 dec ebx
 ret

read:
 push dword BUFSIZE
 mov esi, ibuffer
 push esi
 push dword [fd.in]
 sys.read
 add esp, byte 12
 mov ebx, eax
 or eax, eax
 je .done
 sub eax, eax
 ret

align 4
.done:
 call write ; flush output buffer

 ; close files
 push dword [fd.in]
 sys.close

 push dword [fd.out]
 sys.close

 ; return success
 push dword 0
 sys.exit

align 4
putchar:
 stosb
 inc ecx
 cmp ecx, BUFSIZE
 je write
 ret

align 4
write:
 sub edi, ecx ; start of buffer
 push ecx
 push edi
 push dword [fd.out]
 sys.write
 add esp, byte 12
 sub eax, eax
 sub ecx, ecx ; buffer is empty now
 ret
```

In our *.data* section we now have two new variables, *fd.in* and *fd.out*. We store the input and output file descriptors here.

In the *.code* section we have replaced the references to *stdin* and *stdout* with *[fd.in]* and *[fd.out]*.

The *.code* section now starts with a simple error handler, which does nothing but exit the program with a return value of *1*. The error handler is before *_start* so we are within a short distance from where the errors occur.

Naturally, the program execution still begins at *_start*. First, we remove *argc* and *argv[0]* from the stack: They are of no interest to us (in this program, that is).

We pop *argv[1]* to *ECX*. This register is particularly suited for pointers, as we can handle *NULL* pointers with *jecxz*. If *argv[1]* is not *NULL*, we try to open the file named in the first argument. Otherwise, we continue the program as before: Reading from *stdin*, writing to *stdout*. If we fail to open the input file (e.g., it does not exist), we jump to the error handler and quit.

If all went well, we now check for the second argument. If it is there, we open the output file. Otherwise, we send the output to *stdout*. If we fail to open the output file (e.g., it exists and we do not have the write permission), we, again, jump to the error handler.

The rest of the code is the same as before, except we close the input and output files before exiting, and, as mentioned, we use *[fd.in]* and *[fd.out]*.

Our executable is now a whopping 768 bytes long.

Can we still improve it? Of course! Every program can be improved. Here are a few ideas of what we could do:

- Have our error handler print a message to *stderr*.
- Add error handlers to the `read` and `write` functions.
- Close *stdin* when we open an input file, *stdout* when we open an output file.
- Add command line switches, such as *-i* and *-o*, so we can list the input and output files in any order, or perhaps read from *stdin* and write to a file.
- Print a usage message if command line arguments are incorrect.

I shall leave these enhancements as an exercise to the reader: You already know everything you need to know to implement them.

24.10. Unix Environment

An important Unix concept is the environment, which is defined by *environment variables*. Some are set by the system, others by you, yet others by the *shell*, or any program that loads another program.

24.10.1. How to Find Environment Variables

I said earlier that when a program starts executing, the stack contains *argc* followed by the *NULL*-terminated *argv* array, followed by something else. The âsomething elseâ is the *environment*, or, to be more precise, a *NULL*-terminated array of pointers to *environment variables*. This is often referred to as *env*.

The structure of *env* is the same as that of *argv*, a list of memory addresses followed by a *NULL* (*0*). In this case, there is no *âenvcâ*âwe figure out where the array ends by searching for the final *NULL*.

The variables usually come in the *name=value* format, but sometimes the =value part may be missing. We need to account for that possibility.

24.10.2. Webvar

I could just show you some code that prints the environment the same way the Unix *env* command does. But I thought it would be more interesting to write a simple assembly language CGI utility.

24.10.2.1. CGI: A Quick Overview

10.2.1. CGI: A Quick Overview

I have a detailed CGI tutorial (http://www.whizkidtech.net/cgi-bin/tutorial) on my web site, but here is a very quick overview of CGI:

- The web server communicates with the CGI program by setting *environment variables*.
- The CGI program sends its output to *stdout*. The web server reads it from there.
- It must start with an HTTP header followed by two blank lines.
- It then prints the HTML code, or whatever other type of data it is producing.

N.B.: While certain *environment variables* use standard names, others vary, depending on the web server. That makes *webvars* quite a useful diagnostic tool.

24.10.3. Webvar continued...

Our *webvar* program, then, must send out the HTTP header followed by some HTML mark-up. It then must read the *environment variable* one by one and send them out as part of the HTML page.

The code follows. I placed comments and explanations right inside the code:

```
;;;;;;; webvars.asm ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Copyright (c) 2000 G. Adam Stanislav
; All rights reserved.
;
; Redistribution and use in source and binary forms, with or without
; modification, are permitted provided that the following conditions
; are met:
; 1. Redistributions of source code must retain the above copyright
;    notice, this list of conditions and the following disclaimer.
; 2. Redistributions in binary form must reproduce the above copyright
;    notice, this list of conditions and the following disclaimer in the
;    documentation and/or other materials provided with the distribution.
;
; THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS “AS IS” AND
; ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
; IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
; ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
; FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
; DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
; OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
; HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
; LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
; OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
; SUCH DAMAGE.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Version 1.0
;
; Started:  8-Dec-2000
; Updated:  8-Dec-2000
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%include ’system.inc’

section .data
http db ’Content-type: text/html’, 0Ah, 0Ah
 db ’<?xml version="1.0" encoding="UTF-8"?>’, 0Ah
 db ’<!DOCTYPE html PUBLIC "-//W3C/DTD XHTML Strict//EN" ’
 db ’"DTD/xhtml1-strict.dtd">’, 0Ah
 db ’<html xmlns="http://www.w3.org/1999/xhtml" ’
 db ’xml.lang="en" lang="en">’, 0Ah
 db ’<head>’, 0Ah
 db ’<title>Web Environment</title>’, 0Ah
 db ’<meta name="author" content="G. Adam Stanislav" />’, 0Ah
 db ’</head>’, 0Ah, 0Ah
 db ’<body bgcolor="#ffffff" text="#000000" link="#0000ff" ’
 db ’vlink="#840084" alink="#0000ff">’, 0Ah
 db ’<div class="webvars">’, 0Ah
 db ’<h1>Web Environment</h1>’, 0Ah
 db ’<p>The following <b>environment variables</b> are defined ’
 db ’on this web server:</p>’, 0Ah, 0Ah
 db '<table align="center" width="80" border="0" cellpadding="10" '
 db 'cellspacing="0" class="webvars">', 0Ah
httplen equ $-http
left db '<tr>', 0Ah
 db '<td class="name"><tt>'
leftlen equ $-left
middle db '</tt></td>', 0Ah
 db '<td class="value"><tt><b>'
midlen equ $-middle
undef db '<i>(undefined)</i>'
undeflen equ $-undef
right db '</b></tt></td>', 0Ah
 db '</tr>', 0Ah
rightlen equ $-right
wrap db '</table>', 0Ah
 db '</div>', 0Ah
 db '</body>', 0Ah
 db '</html>', 0Ah, 0Ah
wraplen equ $-wrap

section .code
global _start
_start:
 ; First, send out all the http and xhtml stuff that is
 ; needed before we start showing the environment
 push dword httplen
 push dword http
 push dword stdout
 sys.write

 ; Now find how far on the stack the environment pointers
 ; are. We have 12 bytes we have pushed before "argc"
 mov eax, [esp+12]

 ; We need to remove the following from the stack:
 ;
 ; The 12 bytes we pushed for sys.write
 ; The  4 bytes of argc
 ; The EAX*4 bytes of argv
 ; The  4 bytes of the NULL after argv
 ;
 ; Total:
 ; 20 + eax * 4
 ;
 ; Because stack grows down, we need to ADD that many bytes
 ; to ESP.
 lea esp, [esp+20+eax*4]
 cld ; This should already be the case, but let’s be sure.

 ; Loop through the environment, printing it out
.loop:
 pop edi
 or edi, edi ; Done yet?
 je near .wrap

 ; Print the left part of HTML
 push dword leftlen
 push dword left
 push dword stdout
 sys.write

 ; It may be tempting to search for the ’=’ in the env string next.
 ; But it is possible there is no ’=’, so we search for the
 ; terminating NUL first.
 mov esi, edi ; Save start of string
 sub ecx, ecx
 not ecx ; ECX = FFFFFFFF
 sub eax, eax
repne scasb
 not ecx ; ECX = string length + 1
 mov ebx, ecx ; Save it in EBX

 ; Now is the time to find ’=’
 mov edi, esi ; Start of string
 mov al, ’=’
repne scasb
 not ecx
 add ecx, ebx ; Length of name

 push ecx
 push esi
 push dword stdout
 sys.write

 ; Print the middle part of HTML table code
 push dword midlen
 push dword middle
 push dword stdout
 sys.write
	; Find the length of the value
	not ecx
	lea ebx, [ebx+ecx-1]

	; Print "undefined" if 0
	or ebx, ebx
	jne .value

	mov ebx, undeflen
	mov edi, undef

.value:
	push ebx
	push edi
	push dword stdout
	sys.write

	; Print the right part of the table row
	push dword rightlen
	push dword right
	push dword stdout
	sys.write

	; Get rid of the 60 bytes we have pushed
	add esp, byte 60

	; Get the next variable
	jmp .loop

.wrap:
	; Print the rest of HTML
	push dword wraplen
	push dword wrap
	push dword stdout
	sys.write

	; Return success
	push dword 0
	sys.exit
```

This code produces a 1,396-byte executable. Most of it is data, i.e., the HTML mark-up we need to send out.

Assemble and link it as usual:

```
% nasm -f elf webvars.asm
% ld -s -o webvars webvars.o
```

To use it, you need to upload *webvars* to your web server. Depending on how your web server is set up, you may have to store in a special *cgi-bin* directory, or perhaps rename it with a *.cgi* extension.

Then you need to use your browser to view its output. To see its output on my web server, please instruct your browser to go to *http://www.int80h.org/webvars/*. I am deliberately not placing a regular link here because I do not want its output to appear on all the search engines...

24.11. Working with Files

We have already done some basic file work: We know how to open and close them, how to read and write them using buffers. But Unix offers much more functionality when it comes to files. We will examine some of it in this section, and end up with a nice file conversion utility.

Indeed, let us start at the end, that is, with the file conversion utility. It always makes programming easier when we know from the start what the end product is supposed to do.

One of the first programs I wrote for Unix was *tuc* (ftp://ftp.int80h.org/unix/tuc/), a text-to-Unix file converter. It converts a text file from other operating systems to a Unix text file. In other words, it changes from different kind of line endings to the newline convention of Unix. It saves the output in a different file. Optionally, it converts a Unix text file to a DOS text file.

I have used *tuc* extensively, but always only to convert from some other OS to Unix, never the other way. I have always wished it would just overwrite the file instead of me having to send the output to a different file. Most of the time, I end up using it like this:

```
% tuc myfile tempfile
% mv tempfile myfile
```

It would be nice to have a `ftuc`, i.e., `fast tuc`, and use it like this:

```
% ftuc myfile
```

In this chapter, then, we will write *ftuc* in assembly language (the original *tuc* is in *C*), and study various file-oriented kernel services in the process.

At first sight, such a file conversion is very simple: All you have to do is strip the carriage returns, right?

If you answered yes, think again: That approach will work most of the time (at least with MS DOS text files), but will fail occasionally.

The problem is that not all non-Unix text files end their line with the carriage return / line feed sequence. Some use carriage returns without line feeds. Others combine several blank lines into a single carriage return followed by several line feeds. And so on.

A text file converter, then, must be able to handle any possible line endings:

- carriage return / line feed
- carriage return
- line feed / carriage return
- line feed

It should also handle files that use some kind of a combination of the above (e.g., carriage return followed by several line feeds).

24.11.1. Finite State Machine

The problem is easily solved by the use of a technique called *finite state machine*, originally developed by the designers of digital electronic circuits. A *finite state machine* is a digital circuit whose output is dependent not only on its input but on its previous input, i.e., on its state. The microprocessor is an example of a *finite state machine*: Our assembly language code is assembled to machine language in which some assembly language code produces a single byte of machine language, while others produce several bytes. As the microprocessor fetches the bytes from the memory one by one, some of them simply change its state rather than produce some output. When all the bytes of the op code are fetched, the microrpocessor produces some output, or changes the value of a register, etc.

Because of that, all software is essentially a sequence of state instructions for the microprocessor. Nevertheless, the concept of *finite state machine* is useful in software design as well.

Our text file converter can be designed as a *finite state machine* with three possible states. We could call them states 0-2, but it will make our life easier if we give them symbolic names:

- *ordinary*
- *cr*
- *lf*

Our program will start in the *ordinary* state. During this state, the program action depends on its input as follows:

- If the input is anything other than a carriage return or line feed, the input is simply passed on to the output. The state remains unchanged.
- If the input is a carriage return, the state is changed to *cr*. The input is then discarded, i.e., no output is made.
- If the input is a line feed, the state is changed to *lf*. The input is then discarded.

Whenever we are in the *cr* state, it is because the last input was a carriage return, which was unprocessed. What our software does in this state again depends on the current input:

- If the input is anything other than a carriage return or line feed, output a line feed, then output the input, then change the state to *ordinary*.
- If the input is a carriage return, we have received two (or more) carriage returns in a row. We discard the input, we output a line feed, and leave the state unchanged.
- If the input is a line feed, we output the line feed and change the state to *ordinary*. Note that this is not the same as the first case above â if we tried to combine them, we would be outputting two line feeds instead of one.

Finally, we are in the *lf* state after we have received a line feed that was not preceded by a carriage return. This will happen when our file already is in Unix format, or whenever several lines in a row are expressed by a single carriage return followed by several line feeds, or when line ends with a line feed / carriage return sequence. Here is how we need to handle our input in this state:

- If the input is anything other than a carriage return or line feed, we output a line feed, then output the input, then change the state to *ordinary*. This is exactly the same action as in the *cr* state upon receiving the same kind of input.
- If the input is a carriage return, we discard the input, we output a line feed, then change the state to *ordinary*.
- If the input is a line feed, we output the line feed, and leave the state unchanged.

24.11.1.1. The Final State

The above *finite state machine* works for the entire file, but leaves the possibility that the final line end will be ignored. That will happen whenever the file ends with a single carriage return or a single line feed. I did not think of it when I wrote *tuc*, just to discover that occasionally it strips the last line ending.

This problem is easily fixed by checking the state after the entire file was processed. If the state is not *ordinary*, we simply need to output one last line feed.

N.B.: Now that we have expressed our algorithm as a *finite state machine*, we could easily design a dedicated digital electronic circuit (a âchipâ) to do the conversion for us. Of course, doing so would be considerably more expensive than writing an assembly language program.

24.11.1.2. The Output Counter

Because our file conversion program may be combining two characters into one, we need to use an output counter. We initialize it to 0, and increase it every time we send a character to the output. At the end of the program, the counter will tell us what size we need to set the file to.

24.11.2. Implementing FSM in Software

The hardest part of working with a *finite state machine* is analyzing the problem and expressing it as a *finite state machine*. That accomplished, the software almost writes itself.

In a high-level language, such as C, there are several main approaches. One is to use a *switch* statement which chooses what function should be run. For example,

```
switch (state) {
default:
case REGULAR:
regular(inputchar);
break;
case CR:
cr(inputchar);
break;
case LF:
lf(inputchar);
break;
}
```

Another approach is by using an array of function pointers, something like this:

```
(output[state])(inputchar);
```

Yet another is to have *state* be a function pointer, set to point at the appropriate function:

```
(*state)(inputchar);
```

This is the approach we will use in our program because it is very easy to do in assembly language, and very fast, too. We will simply keep the address of the right procedure in *EBX*, and then just issue:

```
call ebx
```

This is possibly faster than hardcoding the address in the code because the microprocessor does not have to fetch the address from the memoryâit is already stored in one of its registers. I said *possibly* because with the caching modern microprocessors do, either way may be equally fast.

24.11.3. Memory Mapped Files

Because our program works on a single file, we cannot use the approach that worked for us before, i.e., to read from an input file and to write to an output file.

Unix allows us to map a file, or a section of a file, into memory. To do that, we first need to open the file with the appropriate read/write flags. Then we use the *mmap* system call to map it into the memory. One nice thing about *mmap* is that it automatically works with virtual memory: We can map more of the file into the memory than we have physical memory available, yet still access it through regular memory op codes, such as *mov*, *lods*, and *stos*. Whatever changes we make to the memory image of the file will be written to the file by the system. We do not even have to keep the file open: As long as it stays mapped, we can read from it and write to it.

The 32-bit Intel microprocessors can access up to four gigabytes of memory â physical or virtual. The FreeBSD system allows us to use up to a half of it for file mapping.

For simplicity sake, in this tutorial we will only convert files that can be mapped into the memory in their entirety. There are probably not too many text files that exceed two gigabytes in size. If our program encounters one, it will simply display a message suggesting we use the original *tuc* instead.

If you examine your copy of `syscalls.master`, you will find two separate syscalls named *mmap*. This is because of evolution of Unix: There was the traditional BSD *mmap*, syscall 71. That one was superceded by the POSIX *mmap*, syscall 197. The FreeBSD system supports both because older programs were written by using the original BSD version. But new software uses the POSIX version, which is what we will use.

The `syscalls.master` file lists the POSIX version like this:

```
197 STD BSD { caddr_t mmap(caddr_t addr, size_t len, int prot, \
     int flags, int fd, long pad, off_t pos); }
```

This differs slightly from what `mmap(2)` says. That is because `mmap(2)` describes the C version.

The difference is in the *long pad* argument, which is not present in the C version. However, the FreeBSD syscalls add a 32-bit pad after pushing a 64-bit argument. In this case, *off_t* is a 64-bit value.

When we are finished working with a memory-mapped file, we unmap it with the *munmap* syscall:

TIP: For an in-depth treatment of `mmap`, see W. Richard Stevensâ Unix Network Programming, Volume 2, Chapter 12 (http://www.int80h.org/cgi-bin/isbn?isbn=0130810819).

24.11.4. Determining File size

Because we need to tell `mmap` how many bytes of the file to map into the memory, and because we want to map the entire file, we need toq determine the size of the file.

We can use the `fstat` syscall to get all the information about an open file that the system can give us. That includes the file size.

Again, `syscalls.master` lists two versions of `fstat`, a traditional one (syscall 69), and a POSIX one (syscall 189)Naturally, we will use the POSIX version:

```
189 STD POSIX { int fstat(int fd, struct stat *sb); }
```

This is a very straightforward call: We pass to it the address of a *stat* structure and the descriptor of an open file. It will fill out the contents of the *stat* structure.

I do, however, have to say that I tried to declare the `stat` structure in the `.bss` section, and `fstat` did not like it: It set the carry flag indicating an error. After I changed the code to allocate the structure on the stack, everything was working fine.

24.11.5. Changing the File Size

11.5. Changing the File Size

Because our program may combine carriage return / line feed sequences into straight line feeds, our output may be smaller than our input. However, since we are placing our output into the same file we read the input from, we may have to change the size of the file.

The `ftruncate` system call allows us to do just that. Despite its somewhat misleading name, the `ftruncate` system call can be used to both truncate the file (make it smaller) and to grow it.

And yes, we will find two versions of `ftruncate` in `syscalls.master`, an older one (130), and a newer one (201). We will use the newer one:

```
201 STD BSD { int ftruncate(int fd, int pad, off_t length); }
```

Please note that this one contains a `int pad` again.

24.11.6. ftuc

We now know everything we need to write *ftuc*. We start by adding some new lines in `system.inc`. First, we define some constants and structures, somewhere at or near the beginning of the file:

```
;;;;;;; open flags
%define O_RDONLY 0
%define O_WRONLY 1
```

```
%define O_RDWR 2

;;;;;;; mmap flags
%define PROT_NONE 0
%define PROT_READ 1
%define PROT_WRITE 2
%define PROT_EXEC 4
;;
%define MAP_SHARED 0001h
%define MAP_PRIVATE 0002h

;;;;;;; stat structure
struc stat
st_dev resd 1 ; = 0
st_ino resd 1 ; = 4
st_mode resw 1 ; = 8, size is 16 bits
st_nlink resw 1 ; = 10, ditto
st_uid resd 1 ; = 12
st_gid resd 1 ; = 16
st_rdev resd 1 ; = 20
st_atime resd 1 ; = 24
st_atimensec resd 1 ; = 28
st_mtime resd 1 ; = 32
st_mtimensec resd 1 ; = 36
st_ctime resd 1 ; = 40
st_ctimensec resd 1 ; = 44
st_size resd 2 ; = 48, size is 64 bits
st_blocks resd 2 ; = 56, ditto
st_blksize resd 1 ; = 64
st_flags resd 1 ; = 68
st_gen resd 1 ; = 72
st_lspare resd 1 ; = 76
st_qspare resd 4 ; = 80
endstruc
```

We define the new syscalls:

```
%define SYS_mmap 197
%define SYS_munmap 73
%define SYS_fstat 189
%define SYS_ftruncate 201
```

We add the macros for their use:

```
%macro sys.mmap 0
```

```
 system SYS_mmap
%endmacro

%macro sys.munmap 0
 system SYS_munmap
%endmacro

%macro sys.ftruncate 0
 system SYS_ftruncate
%endmacro

%macro sys.fstat 0
 system SYS_fstat
%endmacro
```

And here is our code:

```
;;;;;;; Fast Text-to-Unix Conversion (ftuc.asm) ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; Started: 21-Dec-2000
;; Updated: 22-Dec-2000
;;
;; Copyright 2000 G. Adam Stanislav.
;; All rights reserved.
;;
;;;;;;; v.1 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%include ’system.inc’

section .data
 db ’Copyright 2000 G. Adam Stanislav.’, 0Ah
 db ’All rights reserved.’, 0Ah
usg db ’Usage: ftuc filename’, 0Ah
usglen equ $-usg
co db "ftuc: Can’t open file.", 0Ah
colen equ $-co
fae db ’ftuc: File access error.’, 0Ah
faelen equ $-fae
ftl db ’ftuc: File too long, use regular tuc instead.’, 0Ah
ftllen equ $-ftl
mae db ’ftuc: Memory allocation error.’, 0Ah
maelen equ $-mae

section .code

align 4
```

```
memerr:
 push dword maelen
 push dword mae
 jmp short error

align 4
toolong:
 push dword ftllen
 push dword ftl
 jmp short error

align 4
facerr:
 push dword faelen
 push dword fae
 jmp short error

align 4
cantopen:
 push dword colen
 push dword co
 jmp short error

align 4
usage:
 push dword usglen
 push dword usg

error:
 push dword stderr
 sys.write

 push dword 1
 sys.exit

align 4
global _start
_start:
 pop eax ; argc
 pop eax ; program name
 pop ecx ; file to convert
 jecxz usage

 pop eax
 or eax, eax ; Too many arguments?
```

```
jne usage

; Open the file
push dword O_RDWR
push ecx
sys.open
jc cantopen

mov ebp, eax ; Save fd

sub esp, byte stat_size
mov ebx, esp

; Find file size
push ebx
push ebp ; fd
sys.fstat
jc facerr

mov edx, [ebx + st_size + 4]

; File is too long if EDX != 0 ...
or edx, edx
jne near toolong
mov ecx, [ebx + st_size]
; ... or if it is above 2 GB
or ecx, ecx
js near toolong

; Do nothing if the file is 0 bytes in size
jecxz .quit

; Map the entire file in memory
push edx
push edx ; starting at offset 0
push edx ; pad
push ebp ; fd
push dword MAP_SHARED
push dword PROT_READ | PROT_WRITE
push ecx ; entire file size
push edx ; let system decide on the address
sys.mmap
jc near memerr

mov edi, eax
```

```
 mov esi, eax
 push ecx ; for SYS_munmap
 push edi

 ; Use EBX for state machine
 mov ebx, ordinary
 mov ah, 0Ah
 cld

.loop:
 lodsb
 call ebx
 loop .loop

 cmp ebx, ordinary
 je .filesize

 ; Output final lf
 mov al, ah
 stosb
 inc edx

.filesize:
 ; truncate file to new size
 push dword 0 ; high dword
 push edx ; low dword
 push eax ; pad
 push ebp
 sys.ftruncate

 ; close it (ebp still pushed)
 sys.close

 add esp, byte 16
 sys.munmap

.quit:
 push dword 0
 sys.exit

align 4
ordinary:
 cmp al, 0Dh
 je .cr
 cmp al, ah
 je .lf

 stosb
 inc edx
 ret

align 4
.cr:
 mov ebx, cr
 ret

align 4
.lf:
 mov ebx, lf
 ret

align 4
cr:
 cmp al, 0Dh
 je .cr

 cmp al, ah
 je .lf

 xchg al, ah
 stosb
 inc edx

 xchg al, ah
 ; fall through

.lf:
 stosb
 inc edx
 mov ebx, ordinary
 ret

align 4
.cr:
 mov al, ah
 stosb
 inc edx
 ret
```

```
align 4
lf:
 cmp al, ah
 je .lf

 cmp al, 0Dh
 je .cr

 xchg al, ah
 stosb
 inc edx

 xchg al, ah
 stosb
 inc edx
 mov ebx, ordinary
 ret

align 4
.cr:
 mov ebx, ordinary
 mov al, ah
 ; fall through

.lf:
 stosb
 inc edx
 ret
```

WARNING: Do not use this program on files stored on a disk formated by MS DOS or Windows. There seems to be a subtle bug in the FreeBSD code when using `mmap` on these drives mounted under FreeBSD: If the file is over a certain size, `mmap` will just fill the memory with zeros, and then copy them to the file overwriting its contents.

24.12. Caveats

Assembly language programmers who "grew up" under MS DOS and Windows often tend to take shortcuts. Reading the keyboard scan codes and writing directly to video memory are two classical examples of practices which, under MS DOS are not frowned upon but considered the right thing to do.

The reason? Both the PC BIOS and MS DOS are notoriously slow when performing these operations.

You may be tempted to continue similar practices in the Unix environment. For example, I have seen a web site which explains how to access the keyboard scan codes on a popular Unix clone.

That is generally a *very bad idea* in Unix environment! Let me explain why.

24.12.1. Unix Is Protected

For one thing, it may simply not be possible. Unix runs in protected mode. Only the kernel and device drivers are allowed to access hardware directly. Perhaps a particular Unix clone will let you read the keyboard scan codes, but chances are a real Unix operating system will not. And even if one version may let you do it, the next one may not, so your carefully crafted software may become a dinosaur overnight.

24.12.2. Unix is an Abstraction

But there is a much more important reason not to try accessing the hardware directly (unless, of course, you are writing a device driver), even on the Unix-like systems that let you do it:

Unix is an abstraction!

There is a major difference in the philosophy of design between MS DOS and Unix. MS DOS was designed as a single-user system. It is run on a computer with a keyboard and a video screen attached directly to that computer. User input is almost guaranteed to come from that keyboard. Your programâs output virtually always ends up on that screen.

This is NEVER guaranteed under Unix. It is quite common for a Unix user to pipe and redirect program input and output:

```
% program1 | program2 | program3 > file1
```

If you have written *program2*, your input does not come from the keyboard but from the output*program1*. Similarly, your output does not go to the screen but becomes the input for *program3* whose output, in turn, goes to `file1`.

But there is more! Even if you made sure that your input comes from, and your output goes to, the terminal, there is no guarantee the terminal is a PC: It may not have its video memory where you expect it, nor may its keyboard be producing PC-style scan codes. It may be a Macintosh, or any other computer.

Now you may be shaking your head: My software is in assembly language, how can it run on a Macintosh? But I did not say your software would be running on a Macintosh, only that its terminal may be a Macintosh.

Under Unix, the terminal does not have to be directly attached to the computer that runs your software, it can even be on another continent, or, for that matter, on another planet. It is perfectly possible that a Macintosh user in Australia connects to a Unix system in North America (or anywhere else) via *telnet*. The software then runs on one computer, while the terminal is on a different computer: If you try to read the scan codes, you will get the wrong input!

Same holds true about any other hardware: A file you are reading may be on a disk you have no direct access to. A camera you are reading images from may be on a space shuttle, connected to you via satellites.

That is why under Unix you must never make any assumptions about where your data is coming from and going to. Always let the system handle the physical access to the hardware.

N.B.: These are caveats, not absolute rules. Exceptions are possible. For example, if a text editor has determined it is running on a local machine, it may want to read the scan codes directly for improved control. I am not mentioning these caveats to tell you what to do or what not to do, just to make you aware of certain pitfalls that await you if you have just arrived to Unix form MS DOS. Of course, creative people often break rules, and it is OK as long as they know they are breaking them and why.

24.13. Acknowledgements

This tutorial would never have been possible without the help of many experienced FreeBSD programmers from the FreeBSD hackers (mailto:freebsd-hackers@FreeBSD.org) mailing list, many of whom have patiently answered my questions, and pointed me in the right direction in my attempts to explore the inner workings of Unix system programming in general and FreeBSD in particular.

Thomas M. Sommers opened the door for me. His How do I write "Hello, world" in FreeBSD assembler? (http://home.ptd.net/~tms2/hello.html) web page was my first encounter with an example of assembly language programming under FreeBSD.

Jake Burkholder has kept the door open by willingly answering all of my questions and supplying me with example assembly language source code.

Chapter 25. Alpha

Talk about the architectural specifics of FreeBSD/alpha.

Explanation of allignment errors, how to fix, how to ignore.

Example assembly language code for FreeBSD/alpha.

Chapter 26. IA-64

Talk about the architectural specifics of FreeBSD/ia64.

XIII. Debugging

Chapter 27. Truss

various descriptions on how to debug certain aspects of the system using truss, ktrace, gdb, kgdb, etc

XIV. Compatibility Layers

Chapter 28. Linux

Linux, SVR4, etc

XV. Appendices

Bibliography

[1] Dave A Patterson and John L Hennessy, 1998, 1-55860-428-6, Morgan Kaufmann Publishers, Inc., *Computer Organization and Design: The Hardware / Software Interface, 1-2.*

[2] W. Richard Stevens, 1993, 0-201-56317-7, Addison Wesley Longman, Inc., *Advanced Programming in the Unix Environment, 1-2.*

[3] Marshall Kirk McKusick, Keith Bostic, Michael J Karels, and John S Quarterman, 1996, 0-201-54979-4, Addison-Wesley Publishing Company, Inc., *The Design and Implementation of the 4.4 BSD Operating System, 1-2.*

[4] Aleph One, *Phrack 49; "Smashing the Stack for Fun and Profit".*

[5] Chrispin Cowan, Calton Pu, and Dave Maier, *StackGuard; Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.*

[6] Todd Miller and Theo de Raadt, *strlcpy and strlcat -- consistent, safe string copy and concatenation..*