# Porting the BSD net80211 wireless stack to the Mac OS X kernel

**Prashant Vaibhav**

*Electrical Engineering & Computer Science*
*Jacobs University Bremen*
*Campus Ring 1*
*28759 Bremen*
*Germany*

*Type:* *Bachelor Thesis Report*
*Major:* *Electrical Engineering & Computer Science*
*Date:* *Fall 2011*
*Supervisor:* *Prof. J. Schoenwaelder; Anuj Sehgal*

**Abstract**

This paper describes a project to create a robust IEEE 802.11 stack for the XNU kernel used by Mac OS X and Darwin operating systems, which can be used by third party wireless device drivers in a standard way. This reduces duplication of code, improves code reusability, and provides end-users with a set of standard configuration tools for third party wireless hardware. The wireless stack is largely based on existing code from the OpenBSD version of `net80211` infrastructure. The project reuses code imported from the BSD source tree with as few modifications as reasonable, so that any bug fixes and enhancements from upstream can be merged back without copious effort. Since the BSD and XNU kernel functions and data structures have diverged over the years, a small compatibility layer is implemented to enable the `net80211` code to compile and function properly. This is done primarily through encapsulation inside a C++ class to shadow existing functions in the same process space, and importing any missing functionality directly from BSD source code. Furthermore, code was written to handle connection to the operating system's configuration and networking layer, and marshall data and configuration parameters back and forth. Finally, emulation routines to wrap BSD hardware and memory access services were written to translate them into XNU kernel calls.

# Contents

# 1 Introduction

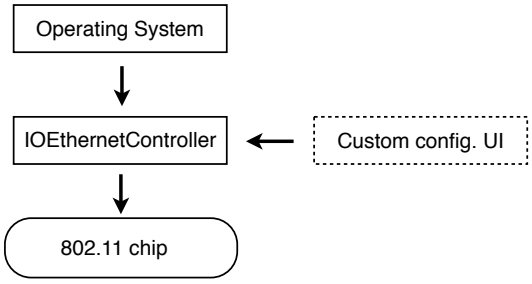
Mac OS X includes the BSD networking stack in its XNU kernel[1]. The XNU kernel currently provides robust support for Ethernet networking. However, support for IEEE 802.11 wireless networking is limited to a set of extensions in the form of `ioctl` calls that the driver must support to allow configuration and reporting. Darwin, the free and open source version of the base operating system used by Mac OS X, entirely omits this support. Third party device drivers for wireless hardware typically have to re-implement this functionality, and masquerade as an Ethernet device with custom configuration GUIs. Ethernet 802.3 packets coming from the operating system are converted to 802.11 frames and vice versa. This requires inclusion of a large amount of 802.11 logic in every driver.

This project aimed to create a robust wireless stack for the XNU kernel, which could be used by IEEE 802.11 device drivers in a standard way. This will reduce duplication of code, improve code reusability, and provide end-users with a set of standard configuration tools for third party wireless hardware.

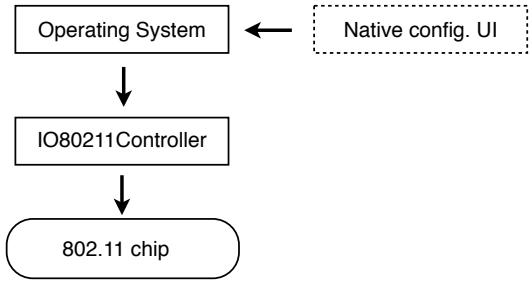
The BSD `net80211` infrastructure is currently one of the most portable and well tested wireless stacks available. Our wireless stack is thus largely based on existing code from `net80211`. We aimed to reuse code imported from the BSD source tree with minimal modifications, so that any bug fixes and enhancements from upstream can be merged back without copious effort. Although several intrusive modifications were still necessary to conform to IOKit (the driver programming layer of the XNU kernel) guidelines, these changes were kept limited. Since the BSD and XNU kernel functions and data structures have diverged over the years[2], a compatibility layer was needed to enable the `net80211` code to compile and function properly. This was done primarily through encapsulation inside a C++ class to shadow existing functions in the same process space, and importing any missing functionality directly from BSD source code. Furthermore, code was written to handle connection to the operating system's configuration and networking layer, and marshall data and configuration parameters back and forth. Finally, emulation routines to wrap BSD hardware and memory access services were written to translate them into XNU kernel calls.

# 2 Motivation

The wireless stack used in Mac OS X is ad-hoc built for each device driver that Apple includes in their hardware products. A standard base class, `IO80211Controller` exists which adds certain functionality to the base class for ethernet controllers: `IOEthernetController`, allowing configuration parameters to be passed back and forth between the kernel device driver and the configuration GUI. This entire code is completely proprietary. Although the header files to compile against the




Existing approach - masquerade hardware as ethernet device




Our approach - expose hardware with full wireless capability

Figure 1: Exposing the hardware to the operating system: comparing two approaches

`IO80211*` interfaces were included with earlier versions of Mac OS X, these have been discontinued in newer iterations. The `IOEthernetController` class, moreover, expects to receive network packets in IEEE 802.3 frame format, and provides the same to the drivers to be sent out. Third party wireless hardware manufacturers therefore have to implement their own wireless stack and configuration tools.

Open source implementations of the Darwin OS (on which Mac OS X is based) exist in the form of OpenDarwin and PureDarwin. These operating systems aim to provide a drop-in replacement for a more traditional Linux / BSD based system. However, these are plagued by extremely limited hardware support, most often limited to the drivers Apple provides as part of source code releases for Mac OS X. Support is entirely absent for wireless networking. Having a standard set of UNIX-compliant configuration tools on the user end, and a standard wireless stack on the kernel end would greatly improve the suitability of Darwin-based operating systems when run on commodity hardware.

A further problem surfaces when one wishes to dual-boot a Macintosh with Mac

OS X and one of the BSDs. Broadcom wireless chips included with most Macintosh computers today do not have a stable driver for BSD platforms. Under BSD, a user is forced to use either a Windows driver under emulation (`NDISulator`), or replace their wireless card with a well-supported one, e.g. Intel PRO/Wireless. The latter has no official driver support under Mac OS X. The user is thus in a situation where only one or the other operating system will support their wireless hardware. Having a BSD `net80211`-like stack would enable creation of OS X drivers for the vast amount of wireless hardware that BSDs already support.

Another scenario where this project's existence can be vastly useful is research environments. A researcher might want to simply monitor packets flowing through the air via 802.11 protocol to track down a bug. A security researcher might further want to inject raw 802.11 packets through his off-the-shelf hardware. Such functionality is only possible if the hardware driver supports it. OS X and Darwin currently do not include any hardware drivers which allow these features. Creating third-party drivers which include monitor mode and raw packet injection support will be very useful in such scenarios.

From an academic point of view, the project highlights the differences between the explicitly C++ object-oriented driver model of Mac OS X, and the C-based "UNIX-like" driver model of the BSDs. Various design challenges exist when converting source code from one kernel to another within the same family (e.g. OpenBSD to NetBSD), most notably the issue of avoiding symbol naming conflicts, the differing threading and synchronization models, and different networking stack APIs. These are described in more detail under section 6.

In terms of scale, the `net80211` code base is massive, with about 12000 lines of code spread across more than 15 files and several hundred API functions. In addition, supporting functionality (e.g. cryptography) amount to 7000 additional lines of code. Porting a massive code base between two different kernel architectures proved to be an interesting case study of code sharing and reuse.

# 3 Current state of the art

Prior work in this area falls under three categories:

1. Third party wireless device drivers ported to Mac OS X or written from scratch.
2. Porting efforts to bring BSD ethernet and wireless code to other operating systems.
3. Simulating (portions of) one operating system's kernel under another operating system

A survey of open source wireless drivers can also be found in [3].

### 3.1 Commercial drivers - Ralink, Prism *et al*

Commercial wireless device drivers are currently only available from Ralink Technologies, which builds 802.11 chips for various USB and PCI form factors. Drivers for the Prism/Prism2 range of 802.11 chips are available from a third party, but this project has been stagnant for several years. These drivers masquerade as a regular wired device, include private 802.11 stacks within the driver, and have their own configuration utilities.

### 3.2 Open source drivers - iwidarwin, Project Camphor *et al*

Other open source efforts have typically focused on the Intel range of wireless devices, which has no official support in OS X. The `iwidarwin` project was an effort to port the Linux drivers of some Intel cards to OS X. These drivers came with a custom configuration utility and also appeared as regular wired ethernet device to operating system.

More recently, the author's own "Project Camphor" drivers have attempted to implement Intel drivers from the ground up using FreeBSD driver code as reference[4]. This project included a nano-size 802.11 stack written from scratch and supporting the bare minimum functionality to get STA (station) mode working. This 802.11 stack is available as a separate framework called "VoodooWireless."[1] These drivers also support the OS-native "Airport" configuration interface.

### 3.3 Porting efforts - MadWifi, Haiku *et al*

In terms of porting the BSD 802.11 stack to another operating system, Sam Leffler[5], the originator of FreeBSD `net80211` stack, himself ported it to Linux to provide support for Atheros wireless driver for Linux. This port is now maintained under the umbrella of the "MadWiFi" project.

The free operating system Haiku currently includes a compatibility layer for FreeBSD wireless and wired networking drivers[6]. This layer re-implements several portions of the FreeBSD networking stack, including re-implementing most of the `mbuf` routines to manage network packets. This was possible in Haiku's case because their network stack shares little to no code with other BSDs, thus avoiding most naming conflicts.

[1] The project described in this paper is a natural evolution of VoodooWireless, replacing the custom 802.11 code with BSD's `net80211`.

### 3.4 Simulated kernel approaches - NDISulator *et al*

A different approach is taken by the projects "NDISulator" (aka Project Evil) and "ndiswrapper" to make Microsoft Windows network drivers work under BSD and Linux respectively. In this approach, a portion of the Windows networking subsystem ("NDIS") is re-implemented, while function calling and system call conventions are converted between native Windows code and the BSD or Linux kernel. This allows unmodified binary drivers to operate correctly under a simulated kernel environment. This approach, although interesting, seeks to solve a different problem than ours (i.e. supporting hardware with no known open-source drivers or documentation).

## 4 Background information on the design challenges

In order to correctly understand the execution and design of this project, it is necessary to know about the driver models followed by BSD and Mac OS X, in particular how the network stack operates. A very simplified description follows:

### 4.1 The BSD driver model

Under BSD, the drivers are generally external modules loaded into the kernel's memory space. The API is written in C. Every driver exposes certain information about itself in a structure defined by the kernel, before registering itself. The driver loading procedure consists of various stages: probe, init, attach, detach, unload etc. During each stage, the kernel passes a structure `device_t` to the driver which identifies the device for which the driver is being loaded. If during 'probe' the driver determines the device to be a suitable match, the driver is loaded. The driver then initializes the hardware and makes it ready for use. Hardware access is done via a C-based API to which various `typedef`'d opaque data structures are passed.[7]

For network drivers, applications send packets to the kernel which then queues them to the appropriate device's packet queue. The driver is then responsible for dequeueing packets from a packet queue (or multiple queues if it supports QoS), and sending it to the hardware. Upon receiving an interrupt, the driver must receive the packet from the hardware and queue it to the kernel's input queue. These interrupts can be primary interrupts in the processor's interrupt context and great care must be taken during this operation. Enqueueing and dequeueing is typically done using preprocessor macros. Threading and locking issues must be manually dealt with as driver functions can be called from different threads.

The packets sent to the driver are in the Ethernet 802.3 packet format and must be

converted to/from 802.11 format. This work is done by the `net80211` stack. For wireless devices, before any "data" packets can be sent or received, however, the hardware must associate to an access point. Configuration data (e.g. ESSID name or scan results) are transferred from user-space configuration utility to the in-kernel driver and vice versa, via `ioctl` calls. The driver, together with the `net80211` system is responsible for creating and sending appropriate management packets to carry out association, dissociation, authentication etc. Wireless adaptors that are soft-radio based fall back on `net80211` layer for this functionality, while those that are firmware microcode-based construct these frames in hardware.

Similarly, link cryptography is handled either by the hardware, or if such is not available, the driver falls back to software crypto. This fallback functionality is implemented as a `struct` of function pointers with the default implementation set to `net80211`'s generic versions.

## 4.2 The Mac OS X driver model

Under Mac OS X, the driver model is called `I/OKit`. It is an object-oriented API written in C++ with certain restrictions. Every device driver derives from a base class of its "family" (i.e. device type). For network drivers, this is typically `IONetworkController` while for wireless network drivers the more specific family class is `IO80211Controller`. The driver loading model is similar to the BSD version described above, where the probe, init and other functions are part of the driver class. Hardware access is handled by a set of C++ based APIs and classes. Concurrency is handled in a rather interesting way: every driver registers a "workloop" with the IOKit (or shares the base class' workloop). Thereafter, all access to the driver and its data go through the workloop, which acts as a gating mechanism to permit only single-threaded access.[8]

Similar to BSD, for network drivers, the kernel queues packets to the appropriate device's queue(s). From here on, a driver can either choose to do its own packet management, or follow the more recommended method of letting the operating system manage its queue. Under the latter case, the driver's `::outputPacket()` function is called for every packet to be transmitted. Upon receiving a hardware interrupt, the primary interrupt is converted to a secondary interrupt and queued on the workloop. Thus a driver has significantly more leeway in the operations it can perform while handling the secondary interrupt.

Since no wireless stack exists under Mac OS X, drivers must handle the task of converting 802.3 frames to 802.11 format, constructing and sending management frames etc. There is no generic implementation to fall back on. However, if a driver subclasses the (currently undocumented) `IO80211Controller` base class, configuration mechanism for 802.11 parameters is provided in the form of an `ioctl`-like function that should be overloaded by the driver. User-space to kernel space

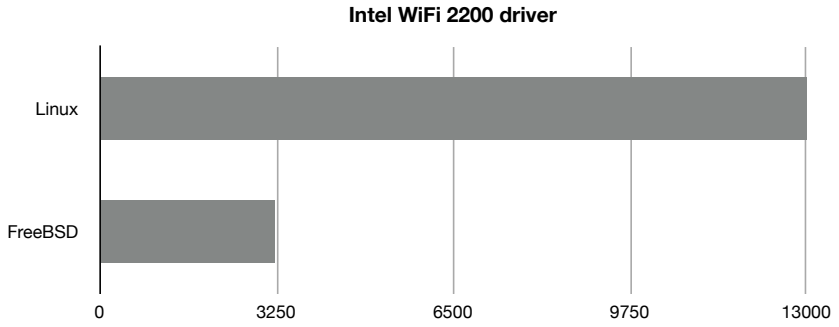



Figure 2: Size of device driver for the Intel 2200 adaptor in FreeBSD and Linux (lines of code)

conversion (and vice versa) is in this case handled by the base class. Apart from configuration data, any other 802.11-specific features such as link cryptography are solely the driver's responsibility.

### 4.3 Choice of `net80211` variant

The different flavors of BSD (FreeBSD, OpenBSD, NetBSD, DragonflyBSD *et al*) all have slightly differing versions of `net80211`. Notably, OpenBSD has a variant of `net80211` based on an early version from FreeBSD. Thereafter, the FreeBSD version included substantial new features which were imported into the remaining BSD flavors.[9] For this project, the OpenBSD variant was chosen due to its simplicity and brevity, while still including most functionality desired by end users and driver developers. The 802.11 codebase of Linux was not chosen due to its complexity and dependence on the Linux kernel, e.g. the source code of the Intel 2200 wireless driver under Linux is well in excess of 13000 lines of code, while that of BSD is about 3200 lines of code for similar functionality[5].

### 4.4 Porting `net80211` code

It was not expected that the `net80211` code from OpenBSD will work without any changes, even if a compatibility layer were implemented. This is because the XNU kernel shares a lot of code with BSD. There are a number of symbol naming conflicts. For example, the `ifnet` structures are used under BSD to represent network interfaces. Under XNU, `IONetworkInterface` and derived classes are used by drivers, however they are in turn based on BSD `ifnet` structure. The XNU kernel programming interface does not allow direct access to this, and to several other structures.

Code must thus be ported manually by scanning all such relevant functionality

that the net80211 layer requires, and modifying them to make use of XNU-native access mechanism. Most changes needed to be done in code that accesses network packet buffers. The network packet routines in BSD operate directly on mbuf structures, while under XNU they have been homogenized to use the new mbuf_t data type and its associated functions. These needed to be manually converted to ensure safe operation and compatibility with future releases of XNU. Other changes include re-architecting the packet input and output points, threading and synchronization (e.g. emulating tsleep() and wakeup() functionality), replacing timing code (e.g. emulating timeout functions and use of "hz" variable which does not exist on tickless kernels like XNU) and re-implementation of several in-kernel cryptography services.

### 4.5 Hooking into XNU networking stack

As described earlier, most of the net80211 code accesses BSD network stack directly.[10] Access to it is not provided for hardware drivers running under XNU. The interface between net80211 and the XNU networking stack therefore needed to be specifically written to marshall data and control parameters appropriately. This constitutes the wrapper layer around our modified net80211. We implemented this by subclassing the IO80211Controller class to create a new 802.11-compliant network interface, and encapsulating the entirety of net80211 functions as member functions of this class. The class would thus receive commands from the networking stack and appropriately marshall it to net80211 functions.

Configuration of 802.11 interfaces on OS X is typically done using ioctl-like function calls made to the IO80211Controller class. The class wrapper layer was implemented as an overriden member function IO80211Controller::apple80211request(), converting control parameters back and forth between net80211 code. Functionality to pass network packets between the networking stack and the net80211 subsystem was also implemented inside the wrapper, with significant re-architecting of packet queue handling, replacing direct ifnet access with IO80211Interface-based access. Additionally, sufficient power management functionality (i.e. suspend, resume) etc. was implemented to correctly reset the driver and net80211 subsystem when the operating system notifies the wrapper of these events.

## 5 The operating system glue

The following section describes the process we followed and the challenges we faced while writing the wrapper and glue code to interface net80211 code with the OS X kernel.

## 5.1 Avoiding symbol naming conflicts and overriding existing kernel functions

Any network driver in IOKit must be a subclass of the IONetworkController class. A wireless device should be subclassed from the (now private) IO80211Controller base class, which itself subclasses IONetworkController and adds some wireless configuration functionality plus glue code to interface with the operating system. Since we decided to support the Airport configuration GUI of OS X, our driver is implemented as a subclass of IO80211Controller.

The actual net80211 functionality is provided as a set of (several dozen) C-style functions in the BSD code base. Separate header files provide prototypes to these functions which a device driver can make use of. Almost every net80211 function also takes as an argument a pointer to a `struct ieee80211com`. This data structure contains the per-instance context for a particular wireless device. A lot of BSD functionality is also used by the net80211 layer, most of which are either not exposed or have slightly differing behavior in IOKit. Simply implementing these functions in our port can potentially give rise to symbol naming conflicts. Every kernel extension is loaded into the same process space - that of the kernel. If, for instance, the function `tsleep()` is defined as a C-function in our kernel extension, and is also present in the XNU kernel already (due to having a large portion of FreeBSD code), the kernel extension will fail to load because of the attempt to map a function that already exists in the kernel space.

One solution to this problem is to rename every such function to something that does not interfere with existing functions. This is very time consuming and requires many intrusive code changes which we could do without. A second solution is to wrap everything inside a namespace. However, namespaces are not well supported by IOKit, and their use comes with several restrictions; e.g. not being able to put any class inside a namespace, which derives from OSObject (all IOKit classes derive from this superclass). Furthermore, using plain-old C-style functions makes interfacing with multiple driver instances difficult, and in some cases makes certain functionality impossible. For instance, these functions would always need a reference to the driver class to be able to operate on it.

The chosen solution to this was to encapsulate all net80211 functions as members of the driver base class. The biggest benefit of this is that shadowing existing kernel functions is made possible. A function defined inside a class takes precedence over a function of the same name defined outside the class. Further, due to C++ name mangling, the function's name is almost guaranteed not to clash with any existing kernel function (as long as the parent class is named appropriately). Lastly, since all functions of net80211 are now member functions of the driver class, they operate in the context of the class instance and can access all private data of the driver class. This also delegates handling of multiple driver instances to the IOKit runtime, as the device-private `struct ieee8021com` data structure is simply a member

variable of the driver class.

Thus, a "Voodoo80211Device" class was declared as a subclass of IO80211Controller, representing the base class for any wireless driver which intends to use our net80211 port. Any driver wishing to make use of net80211 is expected to derive from this base class. It can then transparently call net80211 functions (which are prefixed with "ieee80211_"), which are implemented already in the base class. Any other BSD functionality required by these drivers (e.g. hardware access services) are also implemented as member functions of the same class. The driver code can thus operate transparently, as long as its driver class derives from Voodoo80211Device and all driver functions are declared to be member functions of it.

## 5.2 Deciphering Apple 80211 headers

Usually the base class' implementation is in a separate binary from the derived class implementation. When a driver module is loaded into a running XNU kernel, runtime linking occurs to link the functions in the derived class with the base class. Of particular importance is the vtable linking. A vtable is a table of function pointers which facilitates runtime overriding of functions tagged as "virtual" in the base class. This enables polymorphism in C++, and IOKit is heavily dependent on this functionality.

The trouble starts when the definition of the base class used to compile a derived class differs from one that is actually present on the system. If the vtable differs, runtime linking cannot occur. This has happened in almost every major iteration of OS X, where the base class definition changed, making drivers compiled with older header files unable to load. Apple had, up to Mac OS X 10.5, provided updated header files [11] which could be used to compile a working IO80211Controller-derived class. Since OS X 10.6, these headers were removed, due to unknown reasons. The author's efforts to contact Apple for an explanation yielded only responses to the effect of "we don't know why they were removed".

Since the original header files were distributed under the terms of the APSL (Apple Public Source License), they could be used as the basis for any modified versions as long as the terms are adhered to. Keeping in adherence to the APSL, and the End User License Agreement of OS X which prohibits disassembly, but not examination of binaries, it was possible to reverse-engineer the changes and modify the header files so that they conform to the new class definitions. What was required is to find out what changes, if any, have been made to the ordering of functions in the vtable, and their arguments.

To achieve this, the binary file which includes the object code of the various IO80211* classes was examined with the `otool` command line utility supplied with OS X. The excellent package "IDA Pro" was also used for this purpose. The symbol table defined in the binary file was examined, and all function names found in the

vtable of each class were passed through the "c++filt" command line tool which de-mangles the function names into human-readable format, very similar to the prototype present in the original source file. In this way, the changes made to the base class member functions was tabulated and the changes were applied to the header files we had available from OS X 10.5. This enabled us to compile loadable drivers for OS X versions 10.6 and 10.7 in addition to 10.5. These header files are available in the `apple80211/SL/` and `apple80211/Lion` folders of the source tree.

### 5.3 Responding to configuration commands

Under OS X, configuration of wireless devices is done via the Airport configuration interface in OS X. At the driver layer, these get translated into ioctl-like calls in the IO80211Controller class. The function `::apple80211Request()` is called along with the request type (whether "set" or "get"), the request number, and a pointer to a data structure which is specific to the request number. These request numbers and associated data structures are provided in the apple80211_ioctl.h header file supplied by Apple. However, any sort of documentation related to the ordering of requests and the exact format of responses is omitted. To decipher this, a "FakeAirport" driver was created which subclassed from IO80211Controller. It would print out every incoming 802.11 request along with the request number and any provided data. For "get" requests, the driver would report some dummy data to make the operating system believe it is interfacing with a real wireless device. With trial and error, more information on the API and semantics of how the configuration UI interfaces with a wireless driver was deciphered to a reasonable degree.

## 6 Adapting the net80211 code base

The following section describes the modifications and re-architecting done to the net80211 code base to make it work under OS X.

### 6.1 The `malloc()` and `free()` functions

The `malloc()` and `free()` functions in BSD, unsurprisingly, work the same way as in userspace programs - they allocate dynamic memory on the heap and free them on demand. There are a few important differences though. Memory allocated by `malloc()` is usually wired in kernel space - that is, it cannot be paged out to disk. And every memory allocation occurs in one of several predefined "zones" - for most of net80211 and device drivers, this is the M_DEVBUF zone, or the "device memory" buffer. In IOKit, memory allocation is done via the

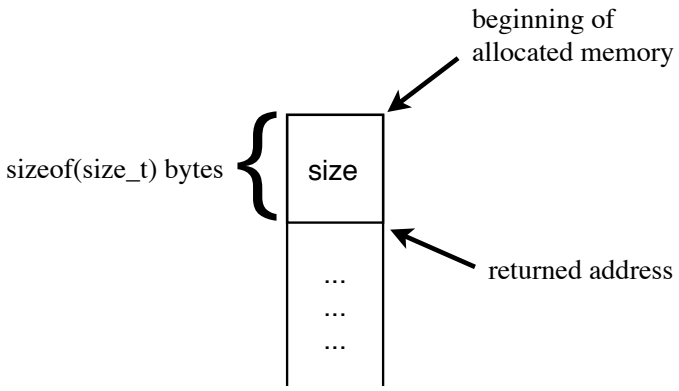



Figure 3: Layout of memory allocations done via re-implemented `malloc()`

`IOMalloc()` and `IOFree()` functions, while the alternative kernel-space memory allocation functions are `OSMalloc()` and `OSFree()`. The latter operates in specific memory allocation zones similar to the BSD drivers, while the former is a general purpose memory allocation service for device drivers. In the Darwin kernel, the BSD `malloc()` as well as other cousins like kalloc() and zalloc() are present, but they are not exposed to the IOKit layer. The `malloc()` and `free()` functions were thus re-implemented as a wrapper around the `IOMalloc()` and `IOFree()` function calls. One important difference is that the IOKit `IOFree()` function requires the size of the previously allocated memory to be freed, while the BSD `free()` function does not. The size was thus not provided in any invocation of `free()` in net80211 code. One way to get around it would be modifying all `free()` functions by explicitly including the size of the struct to be freed as thus:

- Original:
  ```
  struct ieee80211_ccmp_ctx *ctx;
  free(ctx, M_DEVBUF);
  ```
- Modified:
  ```
  struct ieee80211_ccmp_ctx *ctx;
  free(ctx, sizeof(*ctx), M_DEVBUF);
  ```

And then wrap the function call to provide this new size information to `IOFree()`. However, in various parts of net80211, particularly the cryptography routines, the size of the memory allocation is not always equal to the size of the data structure. For example, the size of private data for a cryptographic context (`k_priv` member of `struct ieee80211_key`) can be variable, and is typed as void*. c.f. ieee80211_crypto_ccmp.cpp function ieee80211_ccmp_delete_key(). In this scenario, simply passing in `sizeof(*k->k_priv)` would attempt to get the size of a dereferenced void pointer, which is undefined.

To combat this issue, a simple trick was used. For every call to `malloc()`, the function would allocate some extra bytes (equal to the sizeof(size_t)). Then the size of the allocation (a value of type size_t) is stored at the beginning of this newly allocated memory. The function then skips over this and returns an address beginning after this "hidden" extra value. Code can then continue using the allocated memory without touching the extra size data we stored as a prefix to this memory block. When `free()` is called with a particular address, the size of this allocation is read off from the memory bytes just before the passed-in value. This size information includes the size of the extra value itself. `IOFree()` is then called to free "size" bytes starting from the original address of the memory allocation (i.e. including the extra prefix to store the size). In this way, the `malloc()` and `free()` functions could be used transparently in net80211 and device driver code without having to provide the number of bytes to be free each time. This code is implemented around line 530 in the file Voodoo80211Device.cpp.

## 6.2 The mbuf Kernel Programming Interface (KPI)

BSD code uses the mbuf structure directly to operate on network packets. Several macros are used, e.g. MGET() and MGETHDR() to allocate packets, while the lengths are set/accessed directly as m-¿m_len. In IOKit one must use the mbuf KPI which operates on opaque datatype of mbuf_t representing a single mbuf. This is done so that code written in the past is compatible with future versions of the operating system, even if the underlying mbuf data structure changes.

A bulk of the changes made to net80211 deals with porting BSD-style mbuf routines to use IOKit mbuf_t KPI. Every function which accepted or returned a struct mbuf* was modified to accept or return an mbuf_t. Internally these functions were scanned and all mbuf operations were replaced with their equivalent mbuf_t KPI operations. For example, accesses to the mbuf length via `m->m_len` were replaced with `mbuf_len(m)`, while setting the length via `m->m_len = xx` was replaced with `mbuf_setlen(m, xx)`. The exact changes are too numerous to mention individually, but a look at any mbuf-handling functions will make for a good illustration. An excerpt is reproduced below:

BSD:

```
if (M_TRAILINGSPACE(n) < IEEE80211_CCMP_MICLEN) {
    MGET(n->m_next, M_DONTWAIT, n->m_type);
    if (n->m_next == NULL)
        goto nospace;
    n = n->m_next;
    n->m_len = 0;
}
/* finalize MIC, U := T XOR first-M-bytes( S_0 ) */
```

```
mic = mtod(n, u_int8_t *) + n->m_len;
for (i = 0; i < IEEE80211_CCMP_MICLEN; i++)
    mic[i] = b[i] ^ s0[i];
n->m_len += IEEE80211_CCMP_MICLEN;
n0->m_pkthdr.len += IEEE80211_CCMP_MICLEN;
m_freem(m0);
return n0;
```

Ported:

```
if (mbuf_trailingspace(m) < IEEE80211_CCMP_MICLEN) {
    mbuf_get(MBUF_DONTWAIT, mbuf_type(n), &n);
    if (mbuf_next(n) == NULL)
        goto nospace;
    n = mbuf_next(n);
    mbuf_setlen(n, 0);
}
/* finalize MIC, U := T XOR first-M-bytes( S_0 ) */
mic = mtod(n, u_int8_t *) + mbuf_len(n);
for (i = 0; i < IEEE80211_CCMP_MICLEN; i++)
    mic[i] = b[i] ^ s0[i];
mbuf_adjustlen(n, IEEE80211_CCMP_MICLEN);
mbuf_pkthdr_adjustlen(n0, IEEE80211_CCMP_MICLEN);
mbuf_freem(m0);
return n0;
```

### 6.3 Input and output of packets

In BSD, there are two methods of queueing packets for output: managed stack-provided output queue, and unmanaged manual queueing. In the former, the networking stack provides a queue of packets that are then dequeued by the driver in its `if_start` function and sent out via hardware; while in the latter, the packets are sent directly to the driver which does its own queueing. With the 802.11 stack, a slight complication arises because of the addition of several new types of packets which are not simple network payload. These packets can be, for instance, management or control packets. In addition, 802.11 standard supports Quality of Service (QoS) in the form of WME (Wireless Media Extensions). This results in not one, but four different packet queues, each containing packets of one of the four QoS classes (Video, Voice, Best-Effort and Background). Further, devices that support sleep mode also have a power-management queue which contains packets to be sent when the device wakes up. All of this means that to support all such features, about 6-7 independent queues are needed.

IOKit provides similar functionality wrapped around C++ classes, e.g. `IOPacketQueue`

and friends. Drivers have the option of using their own queueing, akin to the BSD unmanaged interface, or delegate the queuing (as well as dequeueing - in contrast to BSD) to the network stack. In the managed scenario, the network stack dequeues packets from the output queue, and passes them to the driver by calling `outputPacket()` function. This brings certain extra features like making sure every passed mbuf is a self-contained network packet and not a chain. This dequeuing also happens in the single-threaded workloop context, making it very easy for drivers to handle synchronization of hardware resources. Note that packets are received typically via an interrupt handler, which in IOKit is also invoked in the workloop context. This means at a certain point of time, a packet can be either received or transmitted, but not simultaneously. Wireless hardware is inherently half-duplex in the sense that the medium (electromagnetic waves) only supports either sending or receiving at one time. This is a prime candidate for IOKit's single-threaded packet queueing model without loss of (too much) performance.

However, as mentioned earlier, net80211 also supports QoS, power management and control and management frames. These are all put on separate queues. Dequeueing should also happen independently of each queue. This makes it very complicated to adapt to IOKit's much simpler single-threaded model. Adapting it in its existing form would have required using the unmanaged input/output mode with IOKit and handling packet queueing in the driver and/or its base class.

After a study of the driver architecture, it was found that for the most part, drivers dequeue from only two packet queues - the data/payload, and any management frames. This further happens in a single thread rather than simultaneously. The management frames are simply dequeued and sent out first, before any data packets are dequeued. Management frames are typically sent out directly by the hardware while data packets are first passed through net80211 `ieee80211_encap()` to convert from 802.3 ethernet frame format to 802.11 wireless format. Some hardware drivers (e.g. Intel 2200) do not even support sending out management frames.

This means, with a small loss in functionality and performance, we can effectively collapse this into a single-threaded single-queue scheme, with management frames appropriately tagged as such so that the driver can recognize them. This scheme was therefore chosen. Every time a new packet originates in the net80211 part of the code, it is simply enqueued on the output queue of the driver and the queue is kick-started. This queue is initialized by the derived class as usual for an IOKit network driver (i.e. during `createOutputQueue()` function call) [12]. These packets can then be sent out by the hardware driver when the queue calls its `outputPacket()` function. The only remaining problem is that of tagging the packets as being 802.11-format (for management frames). There are a few possibilities for this:

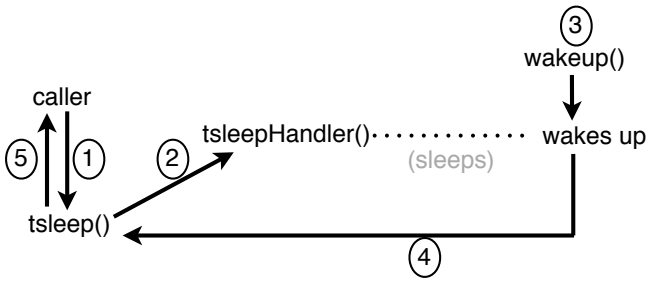
- Use an mbuf tag, which is simply auxiliary data attached to every mbuf. However, this is not a very efficient operation and adds considerably to the overhead of handling each packet.

- Use the "rcvif" (receiving interface) field somehow (in a very non-standard way) to include information about the type of packet. Outgoing packets do not use this field, so it can be overloaded with another type of information. However, this field is already abused enough, with Apple's private implementation using it to indicate the QoS class, and net80211 itself using it to pass along a reference to the "node" (a data structure containing comprehensive information about the intended recipient of the packet) down to the driver's output routine.
- Finally, the chosen method: every time a packet is queued, IOKit makes it possible to supply an additional argument. This argument (a void-pointer) is subsequently passed as-is to the drivers' `outputPacket()` function when the associated packet is dequeued. To take advantage of this facility to solve the problem at hand, the following scheme was used: A data structure `ExtraMbufParams` was declared, with (currently) only one field - a boolean value named "`is80211ManagementFrame`" indicating if the associated packet is a management frame and should be handled differently by the driver. For efficiency, a `const struct ExtraMbufParams` variable is declared with the `is80211ManagementFrame` field set to true, and its address is passed as the argument for every management frame, avoiding the creation of new instances of the struct for every mbuf.

Of course this data structure can also be used to pass along other information, e.g. the node, instead of stuffing it into the "rcvif" field of the mbuf. Such was not done to preserve as much of the net80211 semantics as possible, and to leave open the possibility of implementing manual queueing support later (which would negate the need to tag packets in this way).

A related design choice was that we do not support sending out raw 802.11 packets from the Data Link layer (which should ideally only send data payloads and not 802.11 frames). In net80211 this depends on a tag being present in the mbuf (set by the data link layer) to indicate that a packet is 802.11 format. This entire code was disabled, making processing of outgoing packets not only efficient (by avoiding an expensive tag-search operation on the packet), but also reducing the points in the code which can send out raw 802.11 frames.

Input of packets is comparatively much simpler. In BSD net80211, the `ieee80211_input()` function is eventually called when the driver receives a packet from hardware. The passed mbuf contains the 802.11-format frame, which is then decapsulated by the net80211 stack and passed up to the data link layer through the `ieee80211_deliver_data()` function, while non-data packets are handled internally in net80211. In IOKit, the driver passes along an 802.3 format data frame to the upper layers of the protocol by calling the `inputPacket()` function, which is equivalent to the latter part of the BSD scheme. This scheme was thus modified so that drivers will only call the `ieee80211_input()` to pass up the received 802.11-format frame to net80211; same as BSD drivers. Management and control frames are then handled




1. caller calls tsleep()
2. tsleep() calls tsleepHandler() on the workloop
3. tsleepHandler() drops workloop lock and sleeps until woken up via wakeup()
4. Upon waking up, tsleepHandler() returns to tsleep() with return value
5. tsleep() checks return value and returns to caller with a code indicating whether thread was woken or timed out

Figure 4: How tsleep is emulated via command-gate

by net80211, while data packets are passed to `ieee80211_deliver_data()` which decapsulates and subsequently passes it to the operating system's upper network layer by calling the `inputPacket()` function. This can be examined in the file ieee80211_input.cpp.

### 6.4 Synchronization: `splnet()`, `splx()`, `tsleep()`, `wakeup()`

In BSD, a thread can sleep by calling the `tsleep()` function. It specifies a memory location on which to wait (acting as an identity) and an optional deadline after which to wakeup the thread anyway if not woken up explicitly. This functionality is crucial to net80211 and in particular device driver code to implement asynchronous functionality. IOKit provides `msleep()` functionality (via BSD) which is almost the same as tsleep, except it takes an explicit mutex which is dropped before sleeping and re-acquired on being woken up. However, since our design decision was to avoid manual threading and use IOKit services, the workloop mutex would need to be used for msleep to work as expected. The workloop's mutex is not exposed to clients, however. The recommended alternative is to use `IOCommandGate`, a class that facilitates running operations on the driver's workloop. A workloop is simply a recursive lock that is used throughout the IOKit framework to serialize most accesses to hardware (and thus entry points into driver code). An `IOCommandGate` can be initialized with a callback function and added to the driver's workloop. Then, the `runCommand()` function can be used at any time from any thread context, to call the previously set callback function "on the workloop" with the given arguments. "On the workloop" means the function will be called with the workloop's lock held, meaning that any other part of the code which works in the workloop context (i.e. most of the driver code) is not simultaneously running in any other thread, effectively guaranteeing single threaded con-

text. In this context, the driver can call `IOCommandGate::commandSleep()` function to drop the workloop lock and sleep the current thread. When another part of the code (e.g. an interrupt) calls `IOCommandGate::wakeupGate()`, the original thread resumes execution and grabs the lock again. This effectively emulates the working of `tsleep()` / `wakeup()` pair. These functions were thus re-implemented (in Voodoo80211Device.cpp line 104 onwards) to use the command gate mechanism to sleep on the workloop only. A `tsleepHandler()` callback function is run on the workloop context every time `tsleep()` is called. This function then drops the workloop lock and sleeps. When it is woken up (or the timeout expires), the appropriate reason for waking up is returned, which is then passed on the the original called of the `tsleep()` function. The `wakeup()` function is also reimplemented to simply wake up the command gate instead. [13]

A side-effect of this scheme is that the `splnet()` and `splx()` functions used liberally throughout BSD network stack are effectively rendered unnecessary. The `splnet()` function acquires the global network-stack spinlock while the `splx()` function restores it to its previous state. These functions serialize access to critical sections in BSD networking code. However, by using the workloop semantics, IOKit based drivers (and thus most of net80211) would be running in a single-threaded context, obviating the use of spinlocks. Furthermore, these functions cannot be emulated for the following reason. Most invocations of `splnet()` have a subsequent `tsleep()` function call which would drop the workloop lock and sleep the thread. However, IOKit does not allow holding a spinlock while sleeping a thread, and causes a kernel panic if this ever happens. Thus these two functions were safely replaced with empty bodies (no operation).

### 6.5 timeout functionality

In BSD, time-based callbacks are handled by the timeout service. A client first initializes a `struct timeout*` data structure by calling `timeout_set()`, along with information on which callback function to call and what argument to pass it. Thereafter, the `timeout_add_xx()` functions can be called to add a timer (in the specified unit, e.g. milliseconds). When this time expires, the callback function specified during timeout initialization is called along with an optional argument. This functionality was wrapped around IOKit's existing timer functionality. In IOKit, a timer is represented by an `IOTimerEventSource` class. To use it, a new instance is initialized which is akin to `timeout_set`, and then added to the workloop. To emulate this, the `struct timeout*` data type was replaced with a `VoodooTimeout` class which subclasses `OSObject`. An `OSObject` derivative is required to be passed as an argument to any callback function registered via an `IOTimerEventSource`. The emulated timeout_set() function simply creates a new instance of a `VoodooTimeout`, and initializes it. The callback function is set to the function `voodooTimeoutOccurred()` which

is simply a trampoline which gets the actual function pointer and argument from the VoodooTimeout class and calls the function. The trampoline was required because the signatures of the callback function expected by timeout_set and IOTimerEventSource are different. The timeout_add_xx() then creates a new IOTimerEventSource, adds it to the workloop, sets its timeout and enables it. When the timer expires, the trampoline is called which in turn calls the actual callback function. The timeout_del() function deletes a timeout, which can be emulated with a IOTimerEventSource::cancelTimeout() function call.

However, because there is no timeout_unset()-like function to free a previously allocated timeout, the allocation and freeing of the IOTimerEventSource function is done in the timeout_add_xx() and timeout_del() functions. This of course adds a minuscule amount of lag, but since most timeouts are of the order of several hundred milliseconds (or seconds), the skew is negligible. Another issue is that the VoodooTimeout object created for every timeout_set() call is never destroyed, and this is a potentially grave memory leak. Due to lack of a timeout_unset() function, there seems to be no easy workaround short of adding such a function at judiciously chosen points in net80211 code. However, since the semantics of timer usage are not very clear, when exactly to destroy them is usually unknown. One way would be to destroy it in the timeout_del() function, but that would cause all subsequent timeout_add calls to fail. Because the number of used timers is generally in single-digits or low-double-digits at the max, this memory leak was left as-is.

### 6.6 Network media support

net80211 provides extensive support for different media types - 802.11a/b/g/n, 2GHz or 5GHz channels, 10, 20 or 40 MHz wide channels, TURBO mode and others. These are implemented via the functions ieee80211_media_init(), ieee80211_media_status() and ieee80211_media_change(), in the file ieee80211.c. They make use of callback functions of type ifm_change_cb_t and ifm_stat_cb_t, while the media types themselves are represented by ifmedia_t instances. Initially this was adapted to use IOKit's own IONetworkMedium method. However, this led to difficulty of debugging any problems with this approach and the danger of memory leaks (because every media type in IOKit is a dynamically allocated object, created during ieee80211_media_init(), but never destroyed due to lack of an ieee80211_media_deinit()-like function). It was thus replaced with a sole media type, the "auto" media type which is already well supported by both net80211 and the IOKit network stack. Most of the code was disabled by putting them inside #if 0 blocks, and the functions only returned or set the "auto" media type regardless of the requested media. For the most part, configuring precise media type (e.g. 802.11g-only) was not immediately useful be-

cause there is a lack of any GUI or tool to configure this parameter and the driver would have run in "auto" media mode in either case.

## 6.7 Overloading functions and function pointers

A sizable number of functions in net80211 provide default functionality, but are really intended to be replaced by a hardware-specific implementation provided by the device driver. These functions were represented as function pointers within the `ieee80211com` data structure. Because IOKit is object oriented, and the entire net80211 functionality is encapsulated inside a single class meant to be subclassed by device driver, it made sense to replace these function pointers with virtual member functions. This makes it easy for device drivers to override net80211-provided functionality with hardware-specific routines, simply by reimplementing the virtual functions. It also avoided member-function casts using `OSMemberFunctionCast()` macro to set the function pointers in the `ieee80211com` structure. These functions can be found in Voodoo80211Device.h. All functions tagged as "virtual" are intended to be re-implemented by device drivers.

## 6.8 Other miscellaneous modifications

The following is a list of other changes made to the net80211 source code, many of which apply to more than one source file:

- All functions that took a `struct ifnet *ifp` parameter were modified to take `struct ieee80211com* ic` parameter instead. This was done because almost universally the parameter was cast to a struct ieee80211com* within the function before being used. In the isolated cases where the `ifnet` structure was used directly, this generally meant a point of interface with the operating system's network stack, requiring manual modification to adapt to IOKit in any case.
- All instances of debug printfs which made use of `ifp->if_xname` were replaced with "voodoo_wifi" to maintain a consistent prefix in the debug output. This made possible the use of a single, uniform "grep voodoo_wifi" filter to the "dmesg" output for examining all debug messages printed by net80211, rather than customizing it based on the interface name.
- All instances where a function expects the datatype of a parameter to be an enum, were explicitly cast to that enum type, to silence compiler warnings (or errors depending on the used compiler options). E.g. ieee80211.cpp line 140.
- The functions `ieee80211_attach()` and `ieee80211_detach()` in ieee80211.cpp were modified to hook directly into IOKit network stack and

call ::attachInterface() or ::detachInterface() to attach an interface nub to the controller. A private member variable fInterface stored the returned interface instance for later use.

- Explicit datatype casts were inserted at every instance which required one (e.g. cast to u_int in ieee80211.cpp line 201, ieee80211_chan2ieee()).
- sys/endian.h was re-created to add several functions to deal with little-and-big-endian conversion, which are not defined in IOKit (except as differently-named functions). These functions were simply defined to be their IOKit counterparts, e.g.: #define htole16 OSSwapHostToLittleInt16
- Red-black trees are used extensively in net80211 to store and access node lists, for instance. The Darwin kernel API exposes linked lists and similar data structures but not RB-trees. This was imported from BSD in the form of a header file sys/tree.h. All files which included the system version of this header were replaced to use the local version instead.
- Most cryptography routine files were simply copied over from BSD with only minor modifications. These are in the crypto/ folder. All source code files which included any crypto header files were switched to include the local copy rather than the system copy, since some of the headers are exposed by IOKit, but do not contain the necessary definitions.
- explicit_bzero() was replaced with bzero() and ovbcopy() was replaced with bcopy() because the Darwin implementations of these functions are explicit (won't be optimized out by the compiler) and can handle overlapping ranges. [1]
- Most accesses to the network interface's if_flags field were replaced with fInterface->getFlags(). In some cases, it was more appropriate to use fInterface->linkStatus() to check for the status of the network link.
- The initialization of ieee80211_edca_table[] data structure in ieee80211_output.cpp used a GCC extension to refer to array elements by name. This is not supported on all compilers or even all versions of GCC. It was thus reordered and reformatted to use the standard C array-initialization syntax.

## 6.9 Limitations

Of course certain tradeoffs needed to be made between feature-support and simplicity. To keep the initial attempt at the port a relatively simple, yet functional affair, a few features were not included. The following is a short list of the major features that were not included in the port:

- QoS is initially not supported, and `ieee80211_classify()` function always returns 0 (i.e. default to "Best Effort" class).
- WEP, BIP (Cisco proprietary crypto) and WPA1 (TKIP) are not included in the IEEE 802.11-2007 standard anymore due to poor security performance, and were thus not ported. TKIP in particular, involves copious amounts of operations on mbuf packets and complicated use of timers. With time these will be implemented, but only CCMP (WPA2) is supported at present, and WEP, BIP and TKIP crypto functions have empty implementations.
- Since only STA mode was supported, all code within `#ifndef IEEE80211_STA_ONLY` blocks was removed to maintain brevity.
- All code within `#ifdef NBPFILTER` blocks was removed because Berkeley Packet Filter functionality was not supported. This is arguably a change that deviates from one of the original motivations of this project, however, simplicity and brevity demanded this. Support for BPF can be added at any time later by examining a diff between original net80211 code and the ported code, then inserting the needed BPF calls at the appropriate positions. Usually these code blocks consisted of only 1 or 2 lines.

# 7 Hardware and memory access services

Hardware drivers in BSD use BSD-specific hardware access functions. This can be, for example, functions to map the PCI device memory into the computer's main memory. Other facilities include functions to allocate DMA-safe memory with certain specifications, and mapping an mbuf's data into a list of memory segments to perform hardware scatter/gather operations. These functions are obviously not present in the XNU kernel. To make it easier to port BSD hardware drivers with minimal changes, a lot of these functions were emulated in the form of wrappers around equivalent IOKit functionality. Where possible, the data types of the parameters to the functions were kept the same as in BSD. In most cases these were simply redefined datatypes `typedef`'d to its equivalent IOKit counterpart. In some cases, they were implemented as a data structure containing all relevant information needed to operate on it. These functions can be found in the file compat.h and their implementations can be found in compat.cpp. Note that these were primarily written to aid rapid development of the test driver (described in section 7.2) and are thus not exhaustive.

## 7.1 Allocating aligned memory

Of particular note is the function `allocDmaMemory()` implemented as a member function of Voodoo80211Device. A common operation in device drivers is to

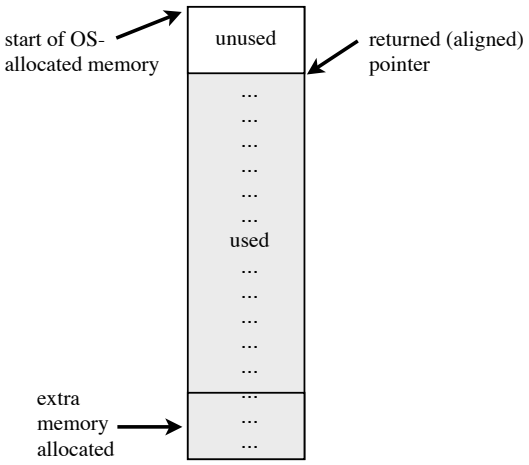



Figure 5: Manually aligning physically contiguous memory

allocate physically contiguous memory whose physical starting address is aligned to a particular boundary (say, 16 KB). Hardware which has specific restrictions needs memory to be aligned in this manner before it can operate on it. IOKit provides the facility to allocate such memory via the IOBufferMemoryDescriptor class. However, in our testing we found that OS X has trouble allocating memory aligned on boundaries larger than the page size (i.e. 4 KB). To get around this, the `allocDmaMemory()` function was implemented. What this function does is the following: if the requested alignment is less than or equal to the page size, nothing special is done. However, if the requested alignment is larger than the page size, the alignment is set to the page size, and extra memory equal to the requested alignment is allocated in addition to the requested amount. The virtual and physical addresses are then incremented manually until the required alignment is achieved by skipping the initial parts of the allocated memory. The newly aligned pointers are then returned to the caller. The extra memory that is allocated acts as a buffer so that the caller does not end up using memory outside the bounds of what was actually allocated. Figure 5 shows how this is achieved.

## 7.2 Test driver

To test the `net80211` subsystem along with its wrapper, we could either have implemented a dummy driver, or a live working one using real hardware. The latter was preferred as it would not only test every aspect of the project thoroughly but also provide something useful in the end.

To this effect, the “Intel PRO/Wireless 3945ABG” card was chosen. This 802.11 chip by Intel has very good Linux, BSD and Windows support, but lacks support for OS X. Since well-documented and stable driver along with source code is avail-

able for this card, it was deemed possible to port it to OS X with fairly reasonable effort. Project Camphor mentioned earlier includes a proof-of-concept driver written by the author. Again the OpenBSD variant was chosen, as it not only relies on OpenBSD's `net80211` variant, but is also more feature-complete compared to other BSDs. Code from this was thus reused, replacing its internal 802.11 code with hooks into our `net80211` port.

The BSD driver needs access to the hardware, and does so using BSD hardware access services. These can either be converted to native XNU model (I/OKit), or wrapped around another compatibility later. We chose a mix of the former and the latter options, emulating BSD hardware access services where feasible, and using IOKit native service directly where required.

However, due to late availability of the test hardware, complications in debugging the driver could not be solved in due time. As of now, the driver loads, attaches to the hardware, hooks into the `net80211` stack and initializes the card. It is then able to read the hardware MAC address and other configuration parameters from the card's EEPROM (e.g. the supported channels and transmission power). The driver is currently unable to load the main firmware image into the hardware, causing a time-out and failure to initialize the hardware to a state where it is capable of transmitting and receiving packets. The functions from net80211 can already be seen generating valid debug output, though.

# 8 Conclusion and future work

We have thus described the effort to port BSD net80211 stack to OS X, the various challenges we faced and the ways in which they were worked around. Although complete testing of the port was not possible within the time constraints, it is believed that once issues with the test driver are ironed out, a working driver can be created rapidly and testing and evaluation of the port can be completed. Overall code composition of the port can be see in Figure 6. Less than half of the code comes from the actual BSD net80211 code base. A sizable part of the code consists of apple80211 header files which make it possible to interface with the operating system. Another significant part of the source tree consists of cryptography services imported directly from BSD. The rest of the source tree contains the compatibility wrappers and test hardware driver. In total, the port consist of more than 22,000 lines of code.

We believe the existence of this port will fuel development of OS X-compatible wireless drivers, and reduce code duplication between these drivers. We further hope that this port continues to be enhanced with new features added to make it more useful. For instance, monitor mode and raw packet injection could be added to make it more useful for security researchers.

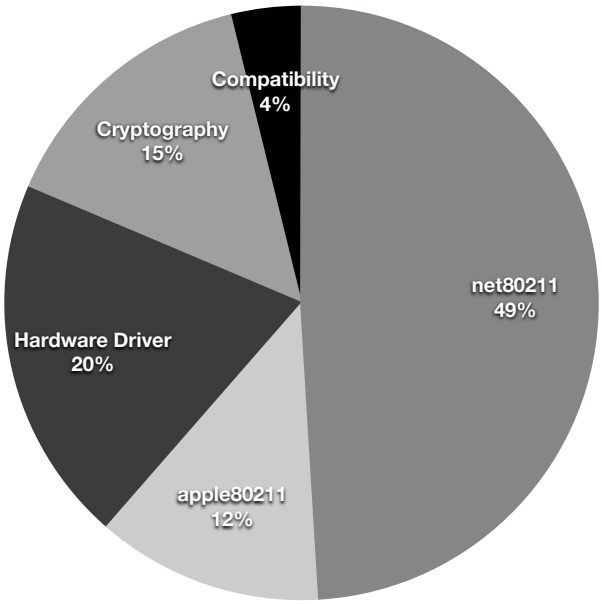



Figure 6: Composition of the code base

Furthermore, several facilities in other variants of net80211 could be incorporated, e.g. the "virtual interface" facility from FreeBSD's net80211, which is also supported in OS X 10.7, will make it possible for the same wireless adaptor to expose several wireless interfaces, each with its own configuration and operating mode. Yet more hardware access services could be emulated to make porting of existing BSD drivers easier. Lastly, a command line configuration tool could be created which allows configuration of the driver without having to use Apple's Airport user interface.

We believe we have only scratched the surface of the possibilities, and are looking forward to seeing how developers make use of it.

**Acknowledgements**

I am grateful to my supervisor Prof. Juergen Schoenwaelder for his continued support during this project. I am also thankful for additional support and advice provided by Anuj Sehgal. Many thanks to Vaibhav Bajpai for providing the test machine.

# List of Figures

# References


[1] Amit Singh. *Mac OS X Internals*. Addison-Wesley Professional, 2006.

[2] M K McKusick and George V Neville-Neil. *Design And Implementation Of The FreeBSD Operating System*. Addison Wesley, 2004.

[3] M. Vipin and S. Srikanth. Analysis of open source drivers for ieee 802.11 wlans. In *Wireless Communication and Sensor Computing, 2010. ICWCSC 2010. International Conference on*, pages 1 –5, jan. 2010.

[4] Prashant Vaibhav. Project camphor. `http://projectcamphor.mercurysquad.com/`.

[5] S. Leffler. Multiband atheros driver for wifi (madwifi), 2005, 2008.

[6] Colin Guenther. Haiku: Wifi stack prototype works. `http://www.haiku-os.org/blog/coling/2009-07-12/wifi_stack_prototype_works`, 2009.

[7] John Baldwin. Writing and adapting device drivers for freebsd. `http://people.freebsd.org/~jhb/papers/drivers/slides.pdf`, 2011.

[8] Inc. Apple. *Kernel Programming Guide - Mac OS X Developer Library*, 2011.

[9] Sam Leffler. Wireless networking in the open source community. `http://people.freebsd.org/~sam/SANE2006-Wireless.pdf`, 2006. The 5th System Administration and Network Engineering Conference.

[10] P. Gangwal. Implementation and experimental study of rate adaptation algorithms in ieee 802.11 wireless networks, 2009. pages 3–6.

[11] Mike. Io80211interface / io80211controller finally (somewhat?) open. `http://lists.apple.com/archives/xcode-users/2007/Nov/msg00544.html`, 2007.

[12] How to write an iokit ethernet driver. `http://arstechnica.com/civis/viewtopic.php?f=19&t=793112`.

[13] J. Levine, O.H. Halvorsen, and D. Clarke. *OS X and IOS Kernel Programming*. Apress Series. Apress, 2011.