Course of Study: Software Engineering

Examiner: Prof. Dr. rer. nat. K. Lagally

Supervisor: Dr. rer. nat. T. Schöbel-Theuer

Thesis No. 2213

Strategies for Automated Porting of Linux Device Drivers to Athomux

Jens-Christian Korth

University of Stuttgart
Institute for Formal Methods in Computer Science
Operating Systems Group
Universitätsstraße 38
D-70569 Stuttgart

Commenced: 5th May 2004

Completed: 4th November 2004

CR-Classification: D.4.7, D.2.13

Abstract

Today's operating systems consist in large parts of device drivers. Developers of new operating systems are confronted with the task of creating drivers that support as much devices as possible with usually very limited manpower.

ATHOMUX, which is the prototype of a new operating system, runs in the Linux userspace and doesn't have any drivers yet. In contrast to that, Linux itself has a large number of device drivers and is available as source code under the same license as ATHOMUX.

This thesis analyzes the possibilities of making the Linux device drivers accessible to ATHOMUX.

Contents

List of Figures

1 Introduction - What is ATHOMUX?

1.1 Why yet another operating system?

ATHOMUX is the implementation of a new operating system architecture which splits up functionality in a fine-grained and orthogonal way by using uniform interfaces. This allows universal composition of the implemented functionality which should lead to dramatic savings in code size, easier maintainability and high code reusability.

Furthermore, ATHOMUX is prepared to run as distributed operating system in a performant way without modification of the application code. This is achieved by using the "Optional Locking" approach as described in [ST04].

1.2 Architecture

The base abstractions of ATHOMUX are *bricks* and *nests*. Each atomic functionality should be implemented in one brick. The bricks usually export one or more interfaces. These interfaces are called nests.

ATHOMUX is an instance oriented operating system. The brick instances can be plugged together like bricks of the LEGO toy system. This can be used to create composite bricks or to wire dynamically configured brick networks.

There are no class hierarchies as known from object oriented programming languages. Every brick instance can be wired to almost any other brick instance. They use anonymous communication; one brick instance doesn't need to know to which other brick instances it's connected to.

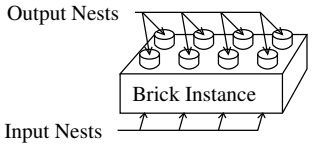

Figure 1: Analogy of LEGO and ATHOMUX bricks

Each nest represents an address space and offers a predefined set of operations on that address space.

There are input and output nests: Each output nest can be wired to one ore more input nests. Usually the output nests offer operations and the input nests use the operations of their peer brick (except for callback operations which should be avoided). There are operations for initialization, data transfer, synchronization, address space mapping, address management and brick management.

A more in-depth description of the ATHOMUX concepts can be found in [ST03].

1.3 Technical Realization

ATHOMUX is designed to allow the easy implementation of functionality with as little code overhead as possible. This is achieved by a special preprocessor and a runtime library.

The preprocessor converts ATHOMUX brick implementations to C code. ATHOMUX bricks are implemented in a mixture of C, preprocessor commands and metadata. The preprocessor commands provide the nest operation call infrastructure, macro substitutions and brick composition statements. All nest operations have a predefined set of parameters which are omitted at the declaration of the operation.

```
  Author: Jens-Christian Korth
  Copyright: Jens-Christian Korth
  License: see files SOFTWARE-LICENSE, PATENT-LICENSE

brick #demo_helloworld
purpose Say hello...
desc
  A simple Hello World implementation.
enddesc


output :>program

operation $output_init
{
  if(@destr) {
    printf("Goodbye world!\n");
  }
  if(@constr) {
    printf("Hello world!\n");
  }
  @success = TRUE;
}
```

Figure 2: ATHOMUX example code

The commodity runtime library provides some generic helper functions for common programming tasks like list management, hashes and pointer management for nests.

1.4 Progress of Implementation

ATHOMUX is a very new project which is still in an experimental phase. During the time when I wrote this thesis, the interfaces were evolving and went through some changes. When I started many features were still missing.

2 Preconsiderations

2.1 Architectural Differences between Linux and ATHOMUX

Linux and ATHOMUX are very different. Linux is a monolithic operating system kernel written in C with assembler optimizations. Its design evolved during the last thirteen years; it doesn't always have a clear concept and it has become very complex.

ATHOMUX can be configured[1] as microkernel or monolithic kernel. It tries to avoid assembler code wherever possible to allow maximum portability and easy understandable code. ATHOMUX has also a very clean design.

2.1.1 Locality

Linux contains many global data structures since its early versions. Every part of the kernel can access the code and data of every other part of the kernel; there are no internal access limitations. From the software engineering point of view this is a very bad design which leads to bad locality in practice. This was also noticed by the Linux kernel developers. As a result of that, the 2.4 and 2.6 kernel series tried to introduce better data encapsulation. But the global data structures are still there, because many drivers depend on them which makes it difficult to substitute them.

ATHOMUX is designed to avoid[2] global code and data. External data structures and code can be accessed solely via the nest interface.

The ATHOMUX design recommends to keep status information separately from the implementation in an external nest. This leads to stateless bricks, having a big advantage for ATHOMUX as network operating system: such bricks can be *migrated* to different nodes easily.

2.1.2 Design of interfaces

Linux uses many different interfaces for different tasks and abstraction levels. Some of them are generic for a class of operation types on a specific abstraction level, but none of them are as generic as the ATHOMUX nest interface which is used on all abstraction levels and operation types.

2.1.3 Layered arrangement of the system

Linux is also missing a clear hierarchy of function calls. It has a hierarchy of abstraction layers, but it allows modules to call functions of each other.

[1] ATHOMUX is an operating system construction kit, not just yet another operating system

[2] The current ATHOMUX prototype contains some global variables. But the average brick doesn't use them, and future version will probably omit them.

There are parts of the Linux kernel having a bidirectional control flow: one coming from a system call from userspace going down the abstraction layers, and one coming from an interrupt call going up the abstraction layers. This leads to unnecessary code complexity.

ATHOMUX brick networks have a clear hierarchy of function calls: The bricks usually receive operation calls on their output nests and make operation calls on their input nests. For special cases which require operation calls in the opposite direction ATHOMUX has a retract operation. But unlike Linux bidirectional control flows are not the default case in ATHOMUX.

2.1.4 Dealing with optimizations

Linux is optimized for performance. It uses the trickiest code and assembler implementations to gain the last bit of speed.

ATHOMUX is optimized for easy implementation and maintenance. This doesn't mean that it ignores performance issues, but for the cases in which a big effort results in only a small gain it prefers a clean and easy understandable implementation.

2.1.5 Functionality provided by one driver module

2.1.5.1 Handling of multiple devices

Each Linux kernel module can only be instantiated once. Most driver modules can handle multiple devices.

Because ATHOMUX is an instance oriented operating system, bricks can be instantiated as often as required. One instance of a driver brick usually handles one device.

2.1.5.2 Device detection

For Linux, every driver module does the hardware detection itself. If no device is found a module doesn't provide its services.

For ATHOMUX, there is a separate strategy level which takes the decision about which brick to instantiate and deinstantiate. As a result, the hardware detection for ATHOMUX drivers should go into these separate strategy bricks.

2.2 Favorable characteristics for Driver Porting

2.2.1 Maintainability

For a future-proof driver system, good maintainability is crucial. This implies to make as less changes as possible to the Linux code, and use automatized code transformations where reasonable.

2.2.2 Doing things the ATHOMUX way

For a good ATHOMUX driver system, the features of ATHOMUX should be used in an advantageous way. This concerns especially the used interfaces.

It's also favorable to keep the bad Linux structures and Linux dependencies out of ATHOMUX as far as possible.

2.2.3 Simplicity

Because the ATHOMUX project has only limited resources, it's favorable to use approaches which work with limited manpower but can be extended on demand.

2.3 Aspects of Driver Porting

In this chapter the different aspects which influence the driver porting will be analyzed.

2.3.1 Runtime environment

ATHOMUX can be configured to run in different runtime environments. The current ATHOMUX prototype runs in the Linux userspace. With only few changes it is possible to run ATHOMUX in the Linux kernelspace. In the future ATHOMUX probably will have its own, Linux independent kernel environment.

- The Linux kernelspace approach requires no porting efforts on the Linux side. It requires only the port of ATHOMUX to the Linux kernelspace and a wrapper which makes the Linux devices accessible to ATHOMUX. This approach does also allow to use Linux binary-only drivers which come without source code.
- There are two possibilities of using the Linux kernel drivers in the userspace: It's possible to use the virtual drivers of the "User Mode Linux" project which provide virtual devices. If real devices are required, they should run in kernelspace and should be bridged using LRPC connector bricks.
- To run Linux drivers in an own ATHOMUX environment, an adaption layer is required which provides the Linux kernelspace base infrastructure.

To become an operating system in the conventional sense, an own ATHOMUX environment is required; however the Linux environments can be used for prototypes.

2.3.2 Usage of Linux code

There are three possibilities of using the Linux code for ATHOMUX: (1) The Linux code can be used as it is; (2) it can be transformed into one or more ATHOMUX bricks, or (3) the Linux binary files can be used if ATHOMUX is configured to run in the Linux kernelspace.

From the ATHOMUX point of view the best solution is to transform the Linux code into bricks. This solution has two issues:

- When building big bricks which contain multiple C files, there may be problems with identical identifiers used for different purposes in different contexts. Linux can be compiled as monolithic kernel without collisions, but this is achieved by compiling each C file separately removing identical identifiers. On object file level there are no collisions.
- As we have seen in 2.1.1, the Linux code has a very bad locality which makes it difficult to split it up into different bricks for some parts. There are other parts which can be split up more easily.

The other two solutions avoid these issues by avoiding code transformation and - as far as possible - modification.

2.3.3 Interface type

When creating ATHOMUX drivers from Linux code, different interface types can be used:

- If the Linux code is split up without any modification, it uses the default C interface. This is probably the most performant interface, but it's not "ATHOMUX style".
- The Linux code can be transformed to use the generic operations of the nest interface; this is a formally correct approach, but it doesn't have any practical advantages.
- The best solution would be to use the nest operations on an address space as interface wherever possible. This is the way ATHOMUX driver interfaces should look like, but this approach requires all different Linux interfaces to be ported manually.

A good compromise can be using the nest address space interfaces for interfaces that will be used by native ATHOMUX bricks (applications or other device drivers). Interfaces which are only used by the ported Linux code can remain as C interfaces.

2.3.4 Granularity

When transforming the Linux code into bricks, there are different possible granularities:

- It is possible to put everything in one monolithic brick; this solution avoids the locality problem, but requires replacement of colliding identifiers.
- Another solution is to put every Linux module into one brick. This should be possible without too big efforts (except when interface changes are required). This solution avoids the identifier collision problem.
- Probably the best solution is to do a manual clustering and put the code of some modules and some subsystems into one brick. This allows minimizing the identifier collision problem and keeping code with bad locality together.

Beside the drivers the Linux code contains other code which cannot be compiled as module. For these parts, manual clustering is required.

2.3.5 Separation of device detection code

True ATHOMUX drivers require the device detection code in separate strategy bricks. This can be achieved by trying to extract the device detection code which might be problematic, or by leaving the device detection code in the driver bricks and creating separate device detection bricks. For modern hardware this shouldn't be too difficult: USB, PCI and ISAPnP devices can be identified by a vendor and product identifier. It should be possible to extract the list of IDs supported by a Linux driver automatically. Other plug and play devices provide a product ID string which can be used for hardware detection. Probably these strings can be extracted too. Only non-PnP devices are problematic and require device-specific detection code.

Without the separation of device detection code, hardware availability would not be known until an instantiation attempt. This would allow to implement a probing strategy which tries to instantiate driver bricks without knowing about the availability of the correspondent hardware.

3 Methods for Driver Porting

The favorable characteristics for driver porting as seen in 2.2 cannot be achieved at the same time. In this chapter I will present three methods which place emphasis on different aspects of the favorable characteristics. As we will see they profit from each other; large parts of the efforts put in the application of one method can be reused for the next one.

3.1 The Wrapping Method

3.1.1 Idea

The idea of this method is to let ATHOMUX run in the normal Linux kernel- and userspace environment and build a generic wrapper around the Linux device interfaces.

3.1.2 Steps to be taken

1. ATHOMUX has to be ported to the Linux kernelspace.
2. Generic wrappers around the Linux block and character device interfaces have to be created.
3. An ATHOMUX-style IO scheduler has to be created in order to support the ATHOMUX IO priorities.
4. A LRPC[3] bridge has to be created to connect the Linux user- and kernelspace.

3.1.3 Required effort

3.1.3.1 The ATHOMUX kernelspace port

The ATHOMUX kernelspace port can be done easily: The current ATHOMUX prototype has only few external dependencies, namely to the glibc library and the pthreads library. For most of the external functions the Linux kernel has similar functions which often have the same signature. Because every external function call requires an *#include* statement it's possible to create a set of replacement include files which use *#define*s to redirect the function calls to the appropriate kernel functions.

3.1.3.2 The Generic wrappers

There are two main classes of Linux devices with different interfaces: The block devices and the character devices. Each of them needs its own, generic wrapper.

[3] Local Remote Procedure Call

Note that these wrappers cover only the devices which export a device interface in the */dev/* directory tree. Devices which are only used by the kernel itself don't have such device interface. The typical devices for that are network adapters and system buses (PCI, USB, SCSI, ...).

Block devices For the block device wrapper, the block IO interface (BIO) which was introduced in the 2.6 kernel series is the ideal interface to use: it manages the IO request blocks which are processed by the drivers. The IO requests are processed asynchronous - thats the behavior the ATHOMUX interface expects.

Character devices For the character device wrapper, suitable interfaces are the *struct file_operations* functions and the VFS[4] functions. The VFS functions use the *struct file_operations* functions and check if the Linux permissions allow the operation.

Network devices The network drivers are a special case, because they usually aren't accessed by an application directly; they need protocol drivers which provide communication sockets for the desired protocols. The socket interface is distantly related to the file interface: it provides similar read and write operations, but uses a completely different addressing scheme for the creation of a socket.

System buses The 2.6 Linux kernel series introduce a common system bus model which makes a generic wrapper possible. Until ATHOMUX has native drivers which would use the system buses, wrapping the Linux bus drivers doesn't make sense.

3.1.3.3 The IO Scheduler

An ATHOMUX-style IO scheduler consists of an IO queue, a scheduling algorithm and an arbiter. The task of the IO queue is to buffer IO request; the scheduling algorithm sorts the buffered requests in an advantageous way; and the arbiter transfers the requests to the driver.

3.1.3.4 The LRPC bridge

The task of the LRPC bridge is to provide a transparent nest link between the Linux user- and kernelspace. The LRPC bridge consists of a client and a server. The server runs in the kernelspace and provides the nest input, and the client runs in the userspace and provides the nest output. The connection can be implemented by introducing a new Linux system call.

[4] Virtual Filesystem Switch

3.1.4 Resulting Linux-independency

The result of this method is very Linux dependent, because it runs in the Linux kernelspace environment. Nevertheless it provides native ATHOMUX nest interfaces for the most important drivers which can be used instead of calling Linux functions directly.

3.1.5 Progress of implementation

A result of my practical work is the port of ATHOMUX to the Linux kernel space, the implementation of the block and character device wrappers, the IO scheduler and the LRPC bridge.

The kernelspace port was done and tested for the *i386* and *x86_64* architectures with little[5] architecture-dependant code.

The device wrappers support the base functionality of the Linux devices. At the time of implementation, the generic operations which are required to support *ioctl*s were not available yet in ATHOMUX.

The IO scheduler supports a FIFO strategy and a scheduling strategy which sorts requests first by priority, then by logical address. Even with the second strategy the performance is below the performance of the Linux IO system which uses a more advanced scheduling strategy and performs caching.

3.1.6 Reusability

- The knowledge gained from creating the generic wrappers can be reused for implementing the automatic code transformations.
- The IO scheduler is independent from the underlying drivers and can be used together with any later ATHOMUX drivers. It doesn't have any Linux dependencies.
- The LRPC bridge can be extended to support real RPC calling bricks on different computers on the same network.
- The ATHOMUX port to the Linux kernelspace can serve as proof of concept that ATHOMUX bricks can run in different environments without huge porting efforts.

[5] ATHOMUX is designed to use 64 bit address spaces. On 32 bit machines the C compiler creates calls to it's runtime library for 64 bit division and multiplication operations. In kernel modules the C runtime library is not available. As solution I wrote some workarounds like replacing divisions by shift operations.

3.2 The Code Transformation Method

3.2.1 Idea

The idea of this method is to create ATHOMUX driver bricks automatically from Linux driver code by using code transformations. This method focuses on the transformations which can be done automatically at the cost of keeping bad Linux structures. As a result, changes in the Linux code can be easily integrated. This is a good compromise between simple wrappers and a complete native ATHOMUX driver system which can by only achieved manually.

3.2.2 Steps to be taken

1. The Linux kernel structures have to be analyzed for suitability of automatic transformations.
2. Appropriate boundaries have to be chosen to split up kernel code into bricks.
3. A code transformer has to be written.
4. Generic bus detection bricks have to be created.

3.2.3 Required effort

To judge the suitability of the Linux code for automatic porting, it's crucial to know how tightly interweaved the different subsystems are. This can be found out by analyzing the dependencies on different granularity levels.

The complexity of a code transformer is due to the previous analysis.

The creation of generic bus detection bricks shouldn't be too difficult.

3.2.4 Resulting Linux-independency

A driver system created by this method can run either in the Linux kernelspace or in an own ATHOMUX kernel environment.

It keeps most of the bad Linux structures, and replaces some interfaces with nest interfaces.

3.2.5 Progress of implementation

Part of my work was the analysis of the Linux structures in 4. The result of this analysis is that this and the following method is infeasible for the Linux code. But it might be possible to apply these methods to other open source operating system kernels.

3.2.6 Reusability

The resulting driver system can be used as foundation for creating a manual optimized driver system. The results of the Linux kernel structure analysis can also serve as precious tool for that.

3.3 The Manual Optimization Method

3.3.1 Idea

The idea of this method is to do manual optimizations on the driver system created by automatic transformation to achieve a fully ATHOMUX-style driver system.

3.3.2 Steps to be taken / required effort

The required effort is probably extreme huge; a more concrete estimation and the concrete steps are due to the realization of the driver system based on automatic transformations.

3.3.3 Resulting Linux-independency

The resulting driver system can be completely Linux independent.

3.3.4 Progress of implementation

This method has not been implemented.

4 Analyzing the Linux Code Structures

In this chapter I will discuss approaches and results from analyzing the Linux kernel code structures.

4.1 Approaches

4.1.1 Existing Works

Analyzing the Linux kernel code structures is quite a complex task. On my investigation I found several attempts of other people:

- Bowman, Siddiqi and Tanuan did an analysis[BST98] of the Linux 2.0 kernel using the "Portable Bookshelf" reengineering tool. This tool tries to extract and analyze the dependencies from the C source files. They noticed that the extracted dependencies weren't accurate enough, so they refined them manually.
- Gleditsch and Gjermshus initiated the Linux Cross-Reference Project which created cross referencing tool for the kernel sources. They mention problems analyzing the kernel sources, too:

  > "Specifically, the heavy use of preprocessor macros makes the parsing a virtual nightmare. We want to index the information in the preprocessor directives as well as the actual C code, so we have to parse both at once, which leads to no end of trouble. (Strict parsing is right out.) Still, we're pretty satisfied with what the indexer manages to get out of it." [GG97]

- A. Melo wrote the *hviz* tool[ME01] which analyzes and visualizes the include file dependency hierarchy. For our purposes, the include files doesn't matter; however I used *hviz* as inspiration for an own analysis tool.
- The Linux *depmod* utility can create a dependency list for Linux kernel modules. However, this is a too coarse grained approach.

Because all these attempts are either outdated or have its focus on different goals I decided to write my own analysis tool.

4.1.2 Creating an Analysis Tool

The first decision for creating a code analysis tool is how to gather the information. Because of the problems analyzing the source code directly which I described in the last section, I choose to extract the imported and exported symbols from the compiled object files.

To judge the structures of the Linux kernel correctly, it is required to analyze the code on different levels of granularity. My analysis tool uses the symbol level, the object file level, the module level, the directory level and the subsystem level.

4.1.3 Analysis Assumptions

For my analysis, I use the vanilla 2.6.9 Linux kernel, configured to build everything thats possible as modules, except drivers for some exotic or very old devices which I disabled completely. The kernel was compiled for the i386 architecture with Pentium 1 optimizations.

As subsystems I use all object files of a directory and its subdirectories. For the overview over the complete kernel and its subsystems in 4.2.1, I used the 12 top-level subdirectories as subsystems unless stated differently. In the showcase analysis of the IDE subsystem in 4.2.2 and in the appendixes, I use a more fine-grained subsystem definition with 48 subsystems by going one subdirectory level deeper for the *drivers* and *arch/i386* subsystems.

4.2 Results

4.2.1 Structure of the Subsystems

In this section I will present the dependencies between the different kernel subsystems and the internal dependencies of the subsystems. The arrows in each graph stand for a "depends-on" relationship; bidirectional arrows stand for mutual dependencies.

4.2.1.1 Overview

Here is an overview over the complete Linux kernel on subsystem level:

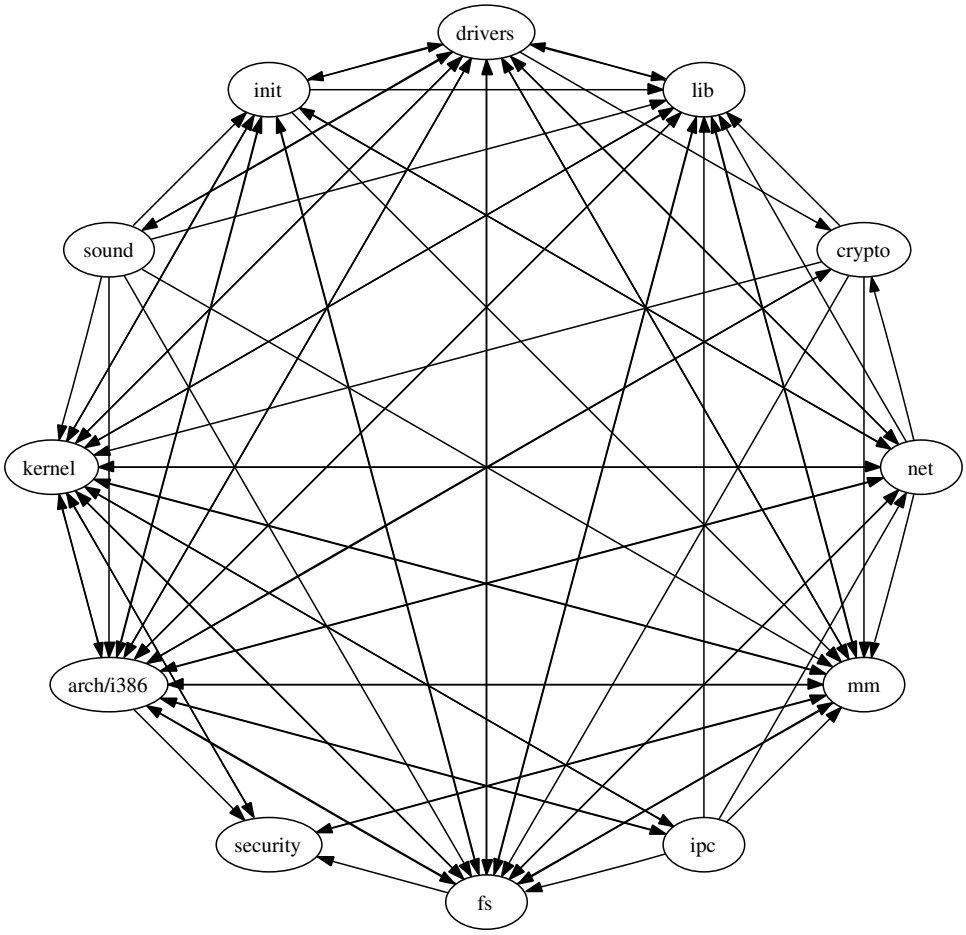

Figure 3: Dependencies between the Linux Kernel Subsystems

Each of the 12 subsystems depends on 2 to 10 other subsystems, 6.8 by average. There are 52 of 66 possible dependency relationships between subsystems ($\hat{=}$79%); 30 of them are mutual dependencies.

For comparison: When using the more fine-grained partitioning with 48 subsystems, each subsystem depends on 1 to 20 other subsystems, 10.25 by average. 10 of the subsystems aren't used by any other subsystem. There are 364 of 1128 possible dependency relationships between subsystems ($\hat{=}$32%); 128 of them are mutual dependencies.

According to software engineering principles, a well designed software system should consist of loosely coupled modules[6] with strong inner cohesion. This can be achieved by building a layered software system in which every layer uses only the services of the layer above.

Linux uses layers too, but there is not always a clear hierarchy; every part of the kernel can use every other part of the kernel. This leads to the tight dependencies which can be seen in figure 3, and makes code reusage difficult.

Another problem are the dependencies of nearly all subsystems on the architecture dependent code (*arch/i386*) which makes it more difficult to create portable ATHOMUX bricks based on Linux code.

In the following sections, the internal structure of the subsystem is being analyzed on directory level. Except for the very small subsystems, the dependencies between the subdirectories are visualized by a dependency graph with one node per subdirectory. To avoid nodes without name, the nodes for the toplevel directories carry the name of the subsystem in capitalized letters.

[6] A module in the sense of software engineering doesn't need to be a module in the sense of a kernel module; it can be a whole subsystem or a part of a kernel module.

4.2.1.2 The architecture dependent subsystem (arch/i386)

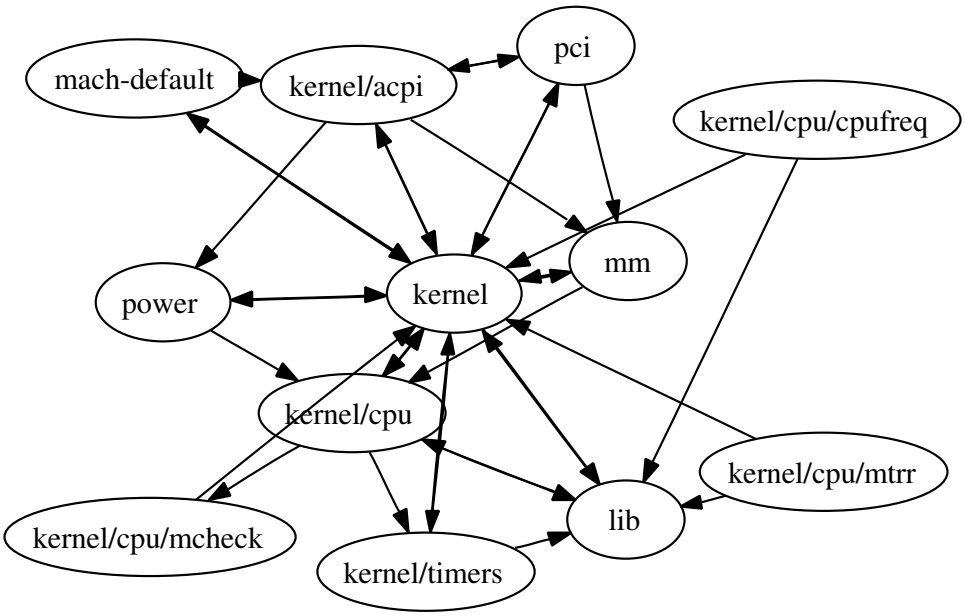

Figure 4: Internal dependencies of the *arch/i386* subsystem

The *arch/i386* subsystem contains the architecture dependent parts of other subsystems. The code from the kernel, lib and mm directories belong logically to the subsystems with the same name.

4.2.1.3 The drivers subsystem (drivers)

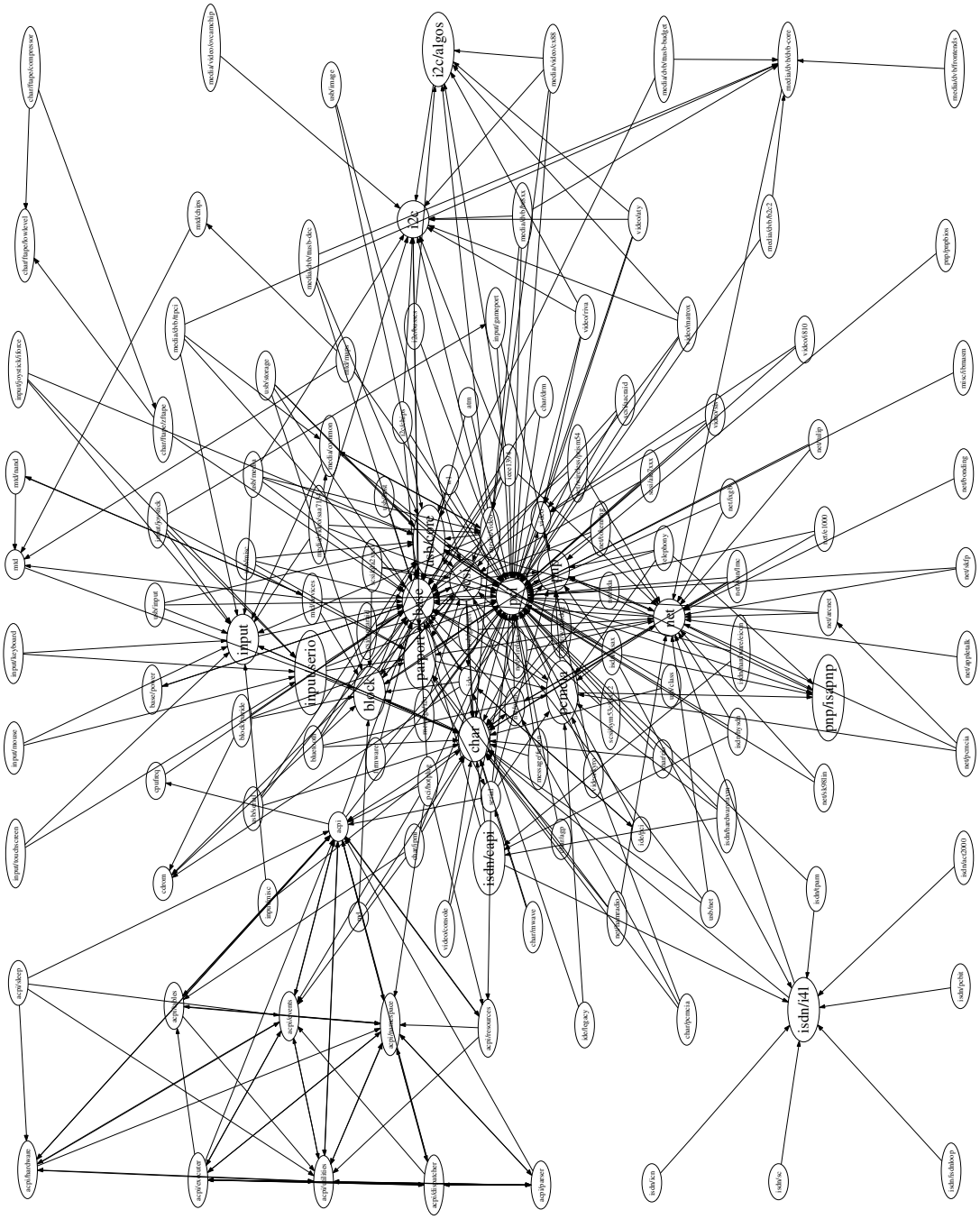

Figure 5: Internal dependencies of the *drivers* subsystem

The *drivers* subsystem is really big. It has 186 subdirectories. The *pci*, *base*, *char* and *net* directories are the most used directories; they are drawn enlarged in the graph together with the other core infrastructure directories of the *drivers* subsystem.

4.2.1.4 The filesystem subsystem (fs)

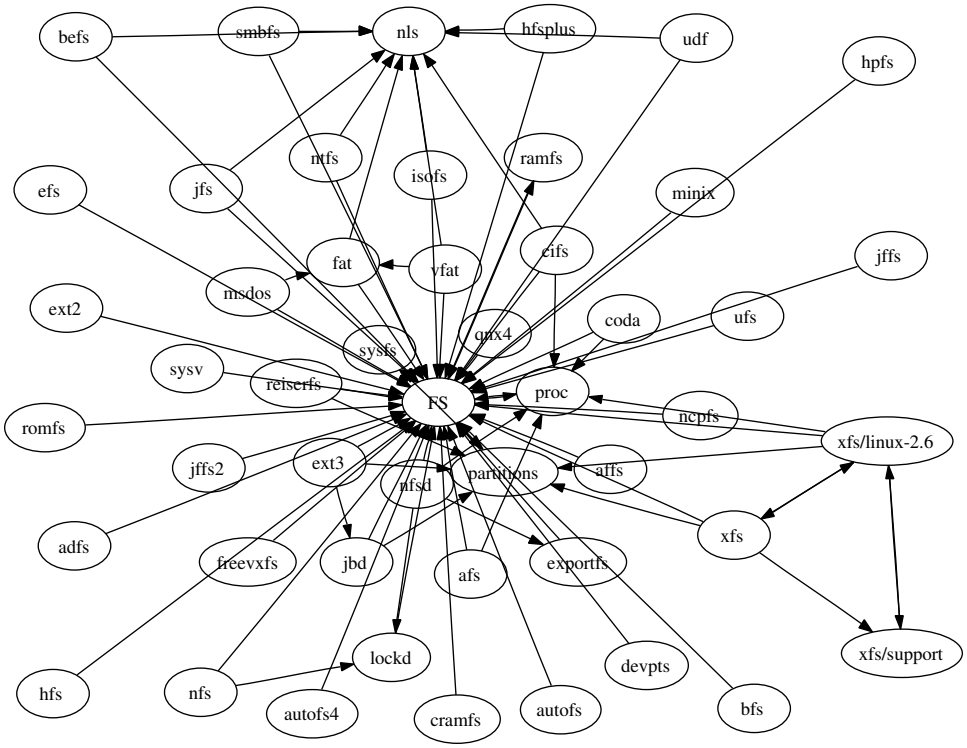

Figure 6: Internal dependencies of the *fs* subsystem

The central infrastructure of the filesystem subsystem is implemented in its toplevel directory ("FS"). Other helper infrastructure is provided by the *proc* (/proc filesystem), *partitions* and *nls* (native language support) directories.

4.2.1.5 The networking subsystem (net)

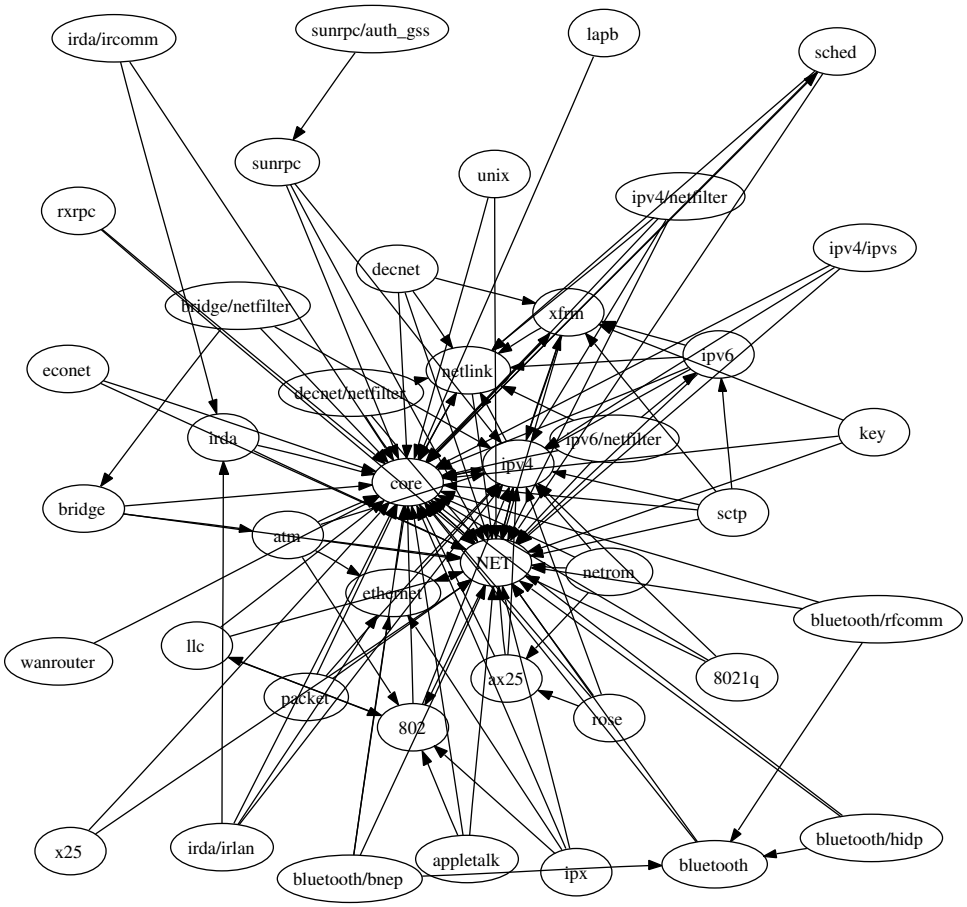

Figure 7: Internal dependencies of the *net* subsystem

The central infrastructure of the networking subsystem is implemented in its toplevel ("NET"), the *core* and the *ipv4* directories.

4.2.1.6 The sound subsystem (sound)

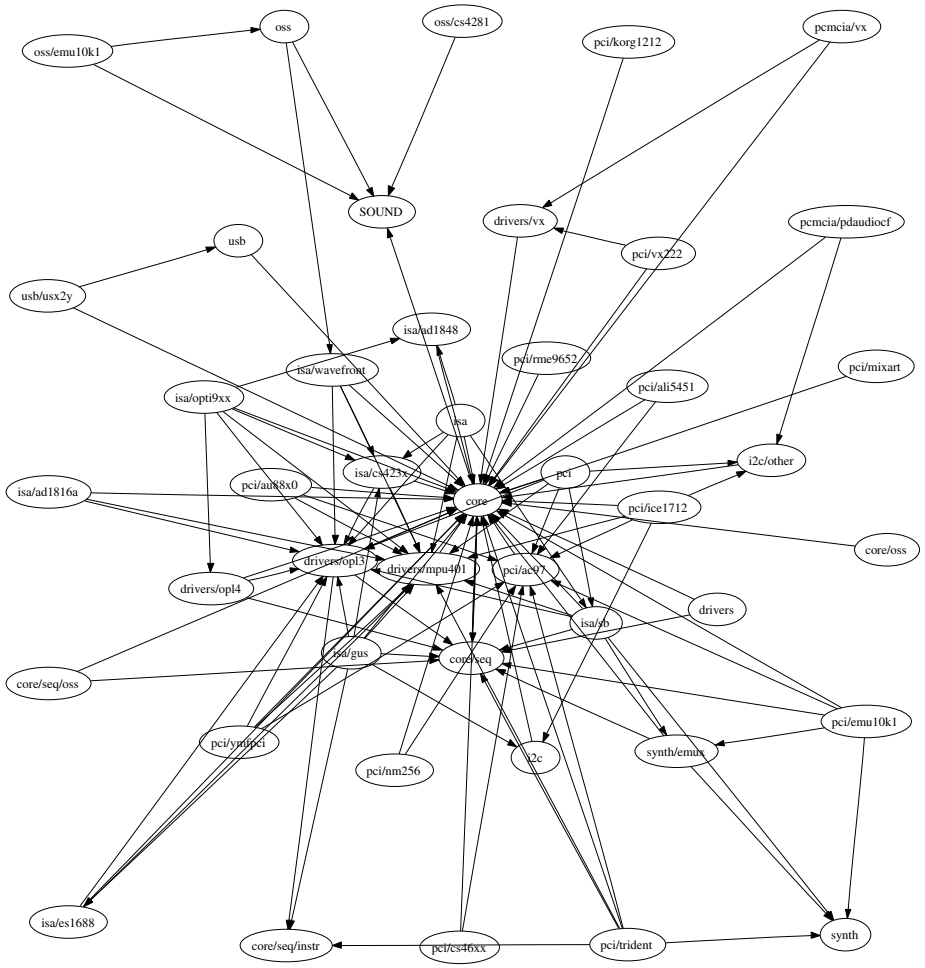

Figure 8: Internal dependencies of the *sound* subsystem

The sound subsystem consists of the old Open Sound System (OSS) and the new Advanced Linux Sound Architecture (ALSA). The OSS code is mainly in the subdirectories containing *oss* in its name; most of the remaining code is ALSA code.

The central infrastructure of this subsystem is implemented in the *core* subdirectory. Helper infrastructure is provided by *pci/ac97*, *drivers/opl3* *drivers/mpu401* (directories with drivers for soundchips used by many soundcard manufacturers) and *core/seq* (sequencer) directories.

4.2.1.7 The other subsystems

The remaining subsystems are much smaller than those described previously. The *security* subsystem has 3 subdirectories; the *lib* subsystem 2, the *kernel* 1 and the *crypto*, *init*, *ipc* (interprocess communication) and *mm* (memory management) have no subdirectories. A dependency graph doesn't make sense for them.

4.2.2 Showcase Analysis of the IDE Subsystem

To judge the difficulties of porting Linux specific drivers to ATHOMUX, a more fine-grained analysis of the kernel structures is required. To avoid getting lost in details, I decided to do a showcase analysis of the IDE subsystem and created dependency graphs for all subsystems which you can find in appendix A.

The IDE subsystem (*drivers/ide* directory of the kernel sources) consists of the generic IDE driver code, chipset specific code and device type specific code. For this showcase analysis, I choose to use a common configuration (omit chipset specific code and support for IDE tape and floppy devices). Even with this restriction, the list of required external symbols is quite long:

object file	**symbols**
arch/i386/kernel/irq.o	disable_irq, disable_irq_nosync, enable_irq, free_irq, probe_irq_off, probe_irq_on, request_irq
arch/i386/kernel/pci-dma.o	dma_alloc_coherent, dma_free_coherent
arch/i386/kernel/semaphore.o	__down_failed, __up_wakeup
arch/i386/lib/delay.o	__const_udelay
arch/i386/lib/memcpy.o	memcpy
arch/i386/lib/strstr.o	strstr
arch/i386/lib/usercopy.o	copy_from_user, copy_to_user
drivers/base/bus.o	bus_register, bus_unregister
drivers/base/core.o	device_register, device_unregister
drivers/base/driver.o	driver_register, driver_unregister
drivers/block/elevator.o	__elv_add_request, elv_next_request, elv_queue_empty, elv_remove_request, elv_requeue_request
drivers/block/genhd.o	add_disk, alloc_disk, blk_register_region, blk_unregister_region, get_disk, put_disk, register_blkdev, unregister_blkdev

object file	**symbols**
drivers/block/ll_rw_blk.o	blk_attempt_remerge, blk_cleanup_queue, blk_dump_rq_flags, blk_execute_rq, blk_get_request, blk_init_queue, blk_max_low_pfn, blk_plug_device, blk_put_request, blk_queue_activity_fn, blk_queue_bounce_limit, blk_queue_dma_alignment, blk_queue_end_tag, blk_queue_hardsect_size, blk_queue_issue_flush_fn, blk_queue_max_hw_segments, blk_queue_max_phys_segments, blk_queue_max_sectors, blk_queue_ordered, blk_queue_prep_rq, blk_queue_segment_boundary, blk_remove_plug, blk_rq_map_sg, blk_rq_prep_restart, blk_start_queue, blk_stop_queue, end_that_request_chunk, end_that_request_first, end_that_request_last, process_that_request_first
drivers/block/scsi_ioctl.o	scsi_cmd_ioctl, scsi_command_size
drivers/cdrom/cdrom.o	cdrom_get_last_written, cdrom_get_media_event, cdrom_ioctl, cdrom_media_changed, cdrom_mode_select, cdrom_mode_sense, cdrom_number_of_slots, cdrom_open, cdrom_release, init_cdrom_command, register_cdrom, unregister_cdrom, add_disk_randomness
drivers/pci/search.o	pci_find_device
fs/block_dev.o	check_disk_change, ioctl_by_bdev
fs/partitions/check.o	del_gendisk
fs/proc/generic.o	proc_mkdir
fs/seq_file.o	seq_printf
init/main.o	system_state
kernel/kmod.o	request_module
kernel/panic.o	panic
kernel/printk.o	printk
kernel/resource.o	ioport_resource, __release_region, __request_region
kernel/sched.o	wait_for_completion, wake_up_process
kernel/timer.o	del_timer, __mod_timer, mod_timer, msleep, schedule_timeout
lib/string.o	strlcpy
lib/vsprintf.o	snprintf, sprintf
mm/memory.o	mem_map
mm/slab.o	kfree, __kmalloc, kmem_cache_alloc, malloc_sizes

It is possible to categorize these dependencies:

- Files in the directory hierarchy of *arch/i386*, *drivers/base*, *drivers/pci*, *fs*, *init*, *kernel*, *lib* and *mm* belong to the required infrastructure of the subsystem.
- Files in the *drivers/block* and *drivers/cdrom* directories belong to the closely related block IO and cdrom subsystems and might be integrated in an ATHOMUX IDE driver. The block IO subsystem does also contain the Linux IO scheduler which is obsolete for ATHOMUX and should be kept out of the ported driver.

Analyzed on file level, 26 of 31 object files required by the IDE subsystems belong to the infrastructure category. On symbol level, 58 of 106 required symbols belong to the related subsystems category.

To reduce the number of external dependencies, it seems worth a try to add the *drivers/block/ll_rw_blk.o* file to our IDE subsystem. But the results are poor: adding it adds 60 new symbols in 15 new and 5 already referenced object files:

object file	**symbols**
arch/i386/kernel/traps.o	dump_stack
drivers/block/as-iosched.o	iosched_as
drivers/block/cfq-iosched.o	iosched_cfq
drivers/block/deadline-iosched.o	iosched_deadline
drivers/block/elevator.o	elevator_exit, elevator_init, elv_add_request, elv_completed_request, elv_former_request, elv_latter_request elv_may_queue, elv_merge, elv_merged_request, elv_merge_requests, elv_put_request, elv_register_queue, elv_set_request, elv_unregister_queue
drivers/block/noop-iosched.o	elevator_noop
fs/bio.o	bio_copy_user, bio_endio, bio_hw_segments, bio_map_user, bio_phys_segments, bio_uncopy_user, bio_unmap_user
fs/partitions/check.o	bdevname
kernel/fork.o	autoremove_wake_function, finish_wait, prepare_to_wait, prepare_to_wait_exclusive
kernel/sched.o	complete, io_schedule, io_schedule_timeout, __wake_up
kernel/workqueue.o	__create_workqueue, flush_workqueue, queue_work
lib/dump_stack.o	dump_stack
lib/kobject.o	kobject_get, kobject_put, kobject_register, kobject_unregister
lib/vsprintf.o	simple_strtoul
mm/bootmem.o	max_low_pfn, max_pfn
mm/highmem.o	blk_queue_bounce, init_emergency_isa_pool
mm/mempool.o	mempool_alloc, mempool_alloc_slab, mempool_create, mempool_destroy, mempool_free, mempool_free_slab
mm/page_alloc.o	per_cpu__page_states
mm/page-writeback.o	block_dump, laptop_io_completion, laptop_mode
mm/slab.o	kmem_cache_create, kmem_cache_free

Adding other files will probably lead to similar results. As a result of that, the reduction of dependencies can mainly be achieved by changing the code manually. As a consequence, it would require a high effort to separate the desired subsystems.

5 Conclusions

As seen in the last chapter, the subsystems of the Linux kernel are interwoven very tightly and cannot be separated automatically with reasonable effort.

The porting methods "automatic transformation" and "manual optimization" depend on automatic methods and become infeasible. To get drivers for ATHOMUX anyway without starting from scratch, I recommend the following approaches:

- Build upon the generic wrapper approach. This results in an ATHOMUX system running under Linux and avoids most problems.
- Split up the Linux kernel code into drivers and infrastructure, and replace the infrastructure parts with an interface-compatible ATHOMUX kernel which has yet to be created. Automatic methods might be used to create nest-style interfaces around the former Linux device interfaces. A long-term goal could be the separation of subsystems using ATHOMUX-style interfaces.
- Use a different driver source: I expect that the DragonFly BSD source code is much more suited for automatic porting than the Linux code. DragonFly BSD is a recent fork of the FreeBSD 4 Unix which was caused by the dissatisfaction of its maintainer about design decisions of the FreeBSD 5 series.
  DragonFly BSD introduces a lightweight messaging API which is used for system calls and IO operations[TDP04]. It is based on abstract address spaces addressed by 64 bit byte offsets. This seems to be a generic interface comparable with the ATHOMUX nest interface which makes it predestined for automatic porting.
  Even without that, DragonFly BSD is probably better structured than Linux: Its ancestor FreeBSD 4 from where most of the code is inherited seems[7] to put an emphasis on good design, e. g. [TFP04] says that unlike other unices they abolished[8] a separate block device driver type to gain code simplicity.

One might compare the driver porting problem with the entropy law: Without putting energy in it; its not possible to improve the structures of a system.

[7] if one can conclude from one good design decision to the whole system; I don't know the BSDs in detail.

[8] This means the separation of the block device drivers and the caching functionality which leads to cleaner structures. Without an own type, block devices are implemented as character devices.

References

[ST03] Schöbel-Theuer, Thomas: *Eine neue Architektur für Betriebssysteme.* Unveröffentlichtes Manuskript einer Monographie, 2003.

[ST04] Schöbel-Theuer, Thomas: *Generalized Optional Locking in Distributed Systems*, 2004.

[BST98] Bowman, Ivan; Siddiqi, Saheem; Tanuan, Meyer C.: *Concrete Architecture of the Linux Kernel* `http://plg.uwaterloo.ca/~itbowman/CS746G/a2`, 1998.

[GG97] Gleditsch, Arne Georg; Gjermshus, Per Kristian: *Cross-Referencing Linux* `http://lxr.linux.no/`, 1997.

[ME01] Melo, Arnaldo Carvalho de: *Include Dependency Graph Script* `http://www.ussg.iu.edu/hypermail/linux/kernel/0112.3/1070.html`, 2001.

[TFP04] The FreeBSD Project: *FreeBSD Architecture Handbook* `http://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/driverbasics-block.html`, 2004.

[TDP04] The DragonFly Project: *DragonFly I/O Device Operations* `http://www.dragonflybsd.org/goals/iomodel.cgi`, 2004.

A External Dependencies of the Linux Kernel Subsystems

These are the external dependencies of each subsystem on object file level as determined by my analysis tool. The circle in the center of each graph contains the name of the subsystem which is being analyzed. The gray boxes represent the subsystems which are required by the analyzed subsystem; they also contain a list of their object files which export symbols required by the analyzed subsystem.

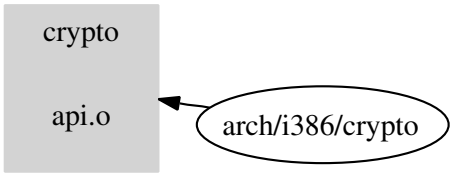

Figure 9: External dependencies of the *arch/i386/crypto* subsystem

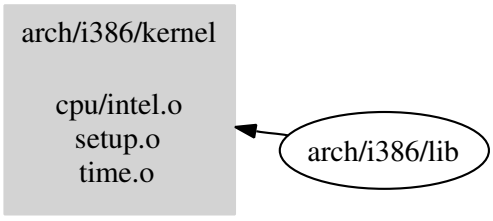

Figure 10: External dependencies of the *arch/i386/lib* subsystem

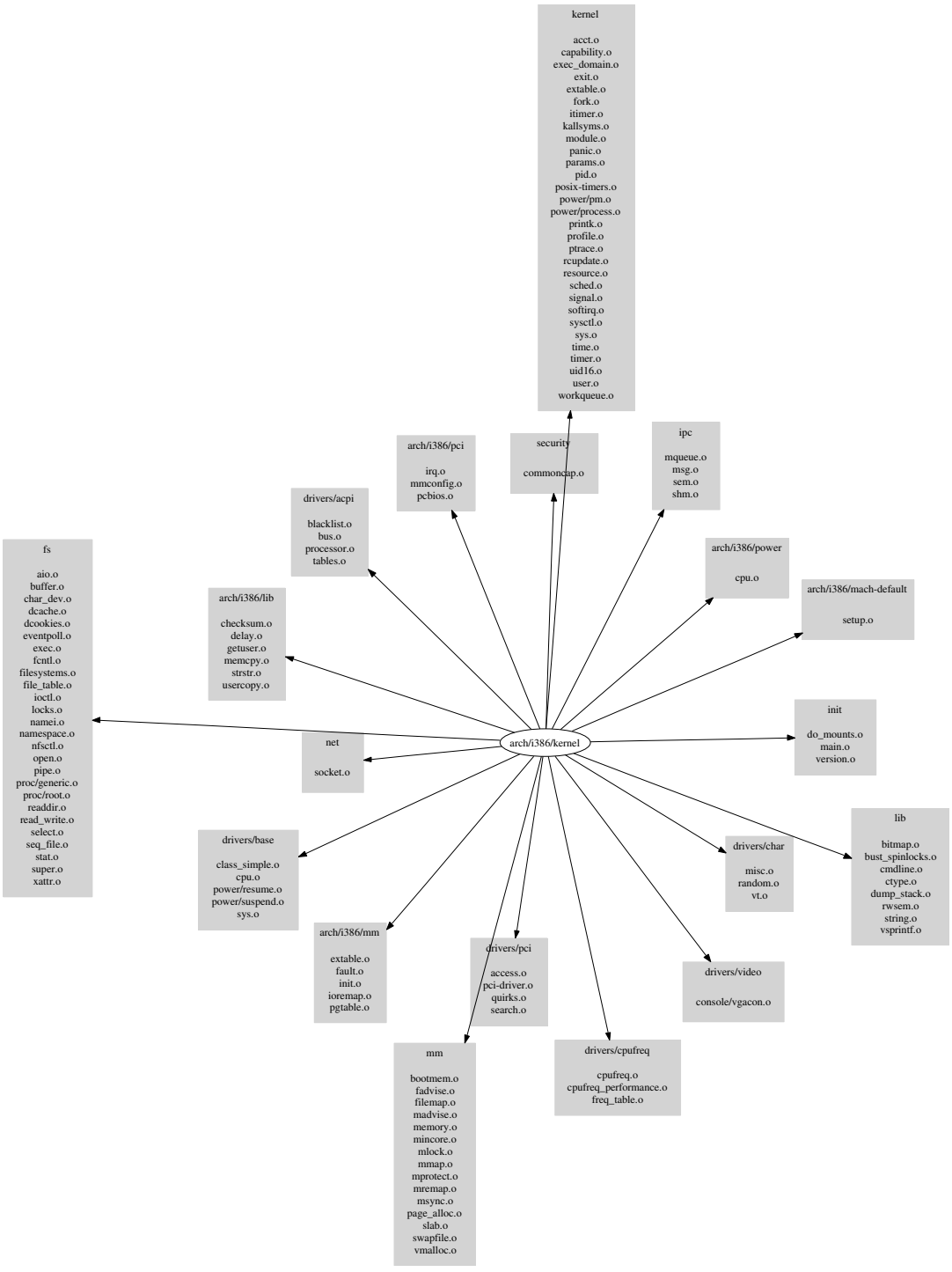

Figure 11: External dependencies of the *arch/i386/kernel* subsystem

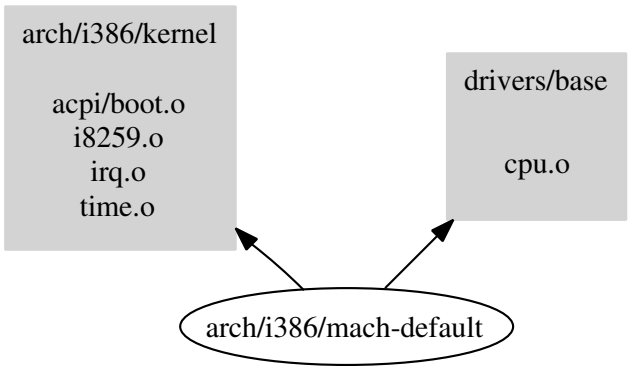

Figure 12: External dependencies of the *arch/i386/mach-default* subsystem

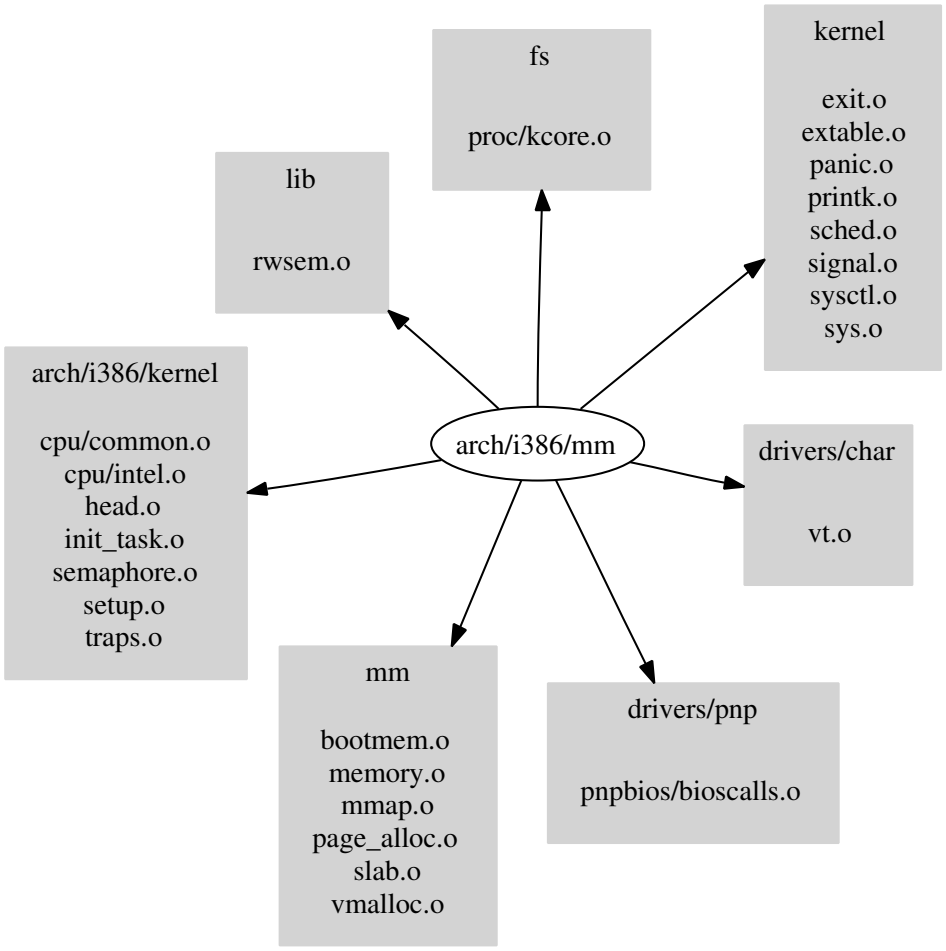

Figure 13: External dependencies of the *arch/i386/mm* subsystem

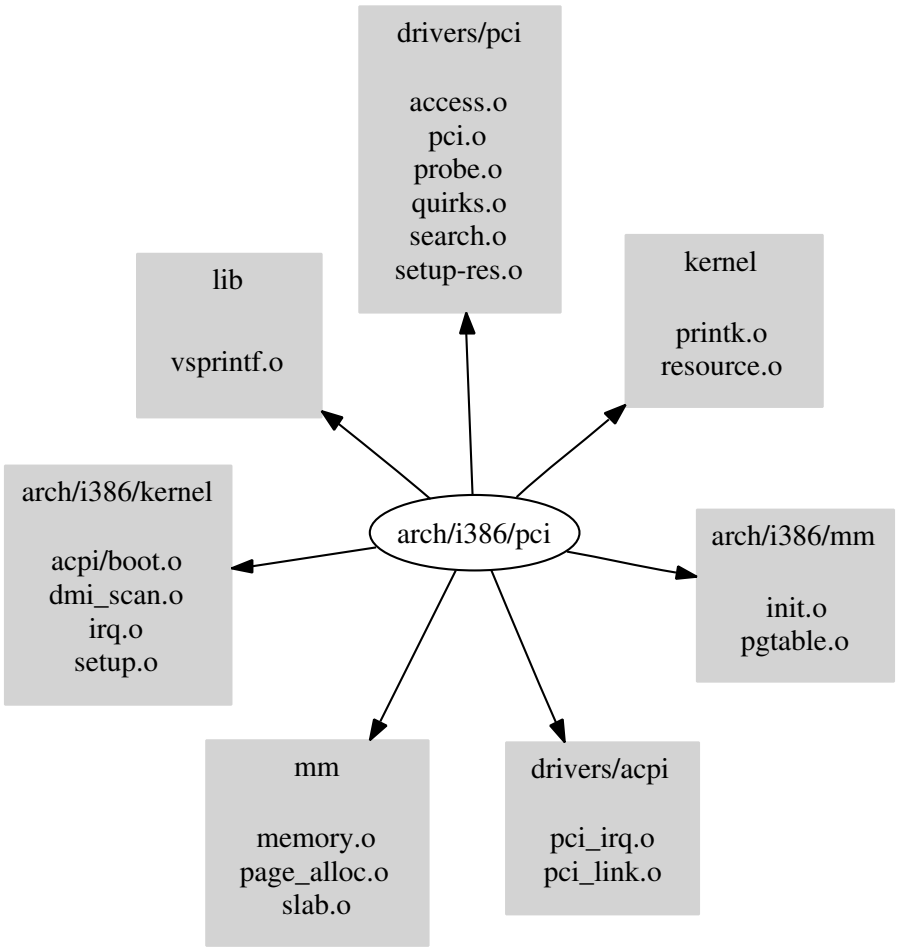

Figure 14: External dependencies of the *arch/i386/pci* subsystem

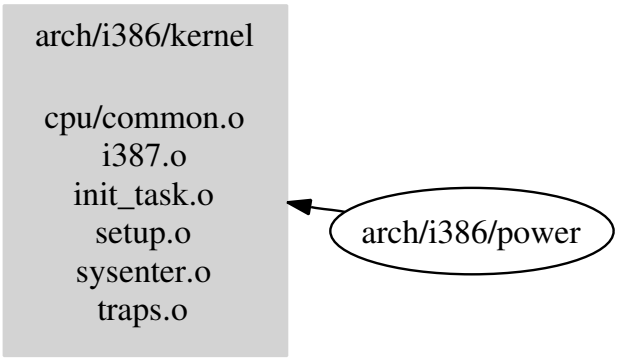

Figure 15: External dependencies of the *arch/i386/power* subsystem

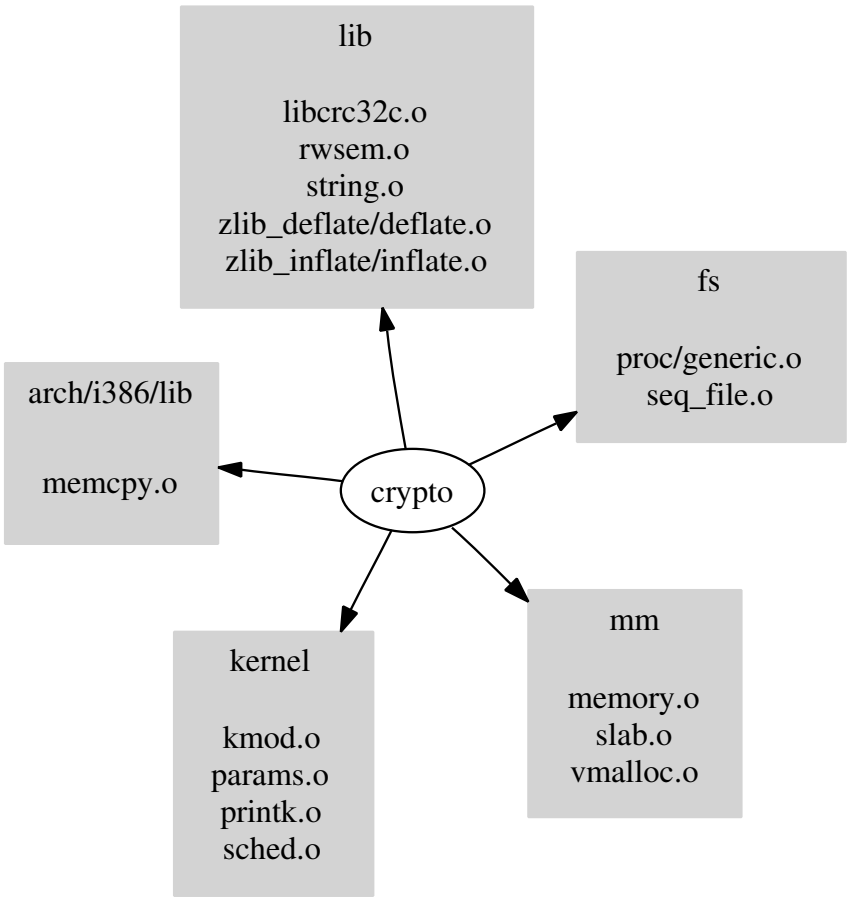

Figure 16: External dependencies of the *crypto* subsystem

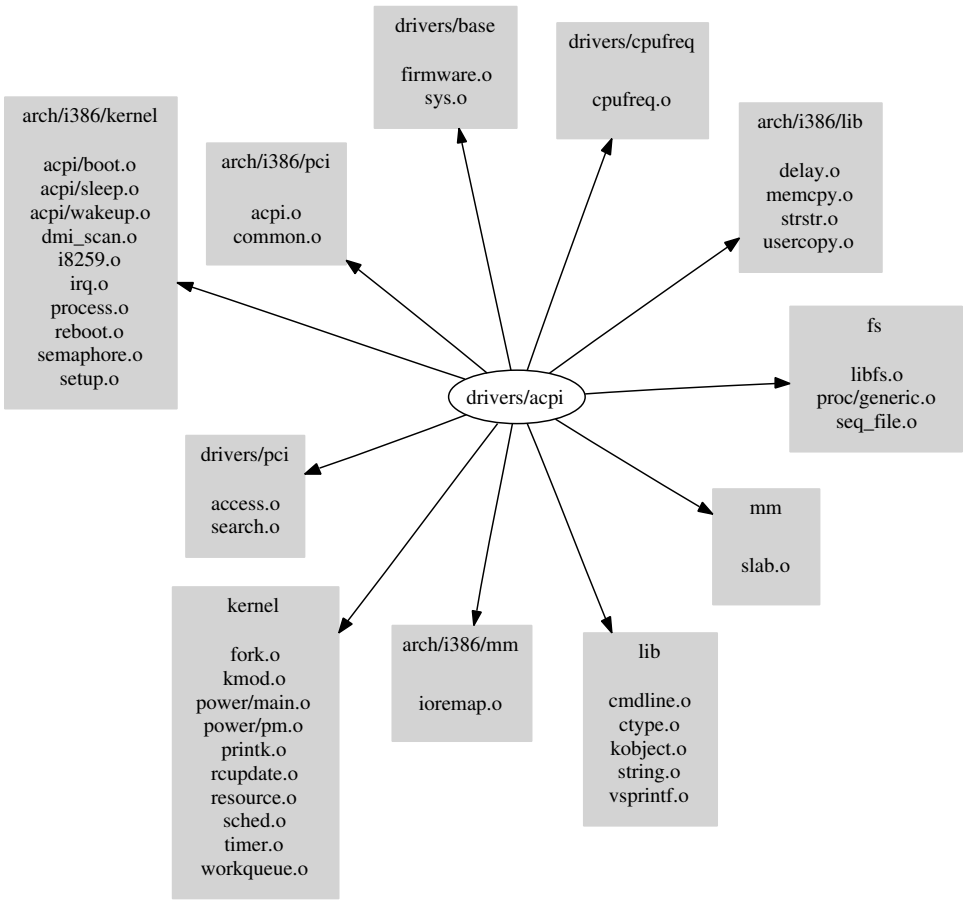

Figure 17: External dependencies of the *drivers/acpi* subsystem

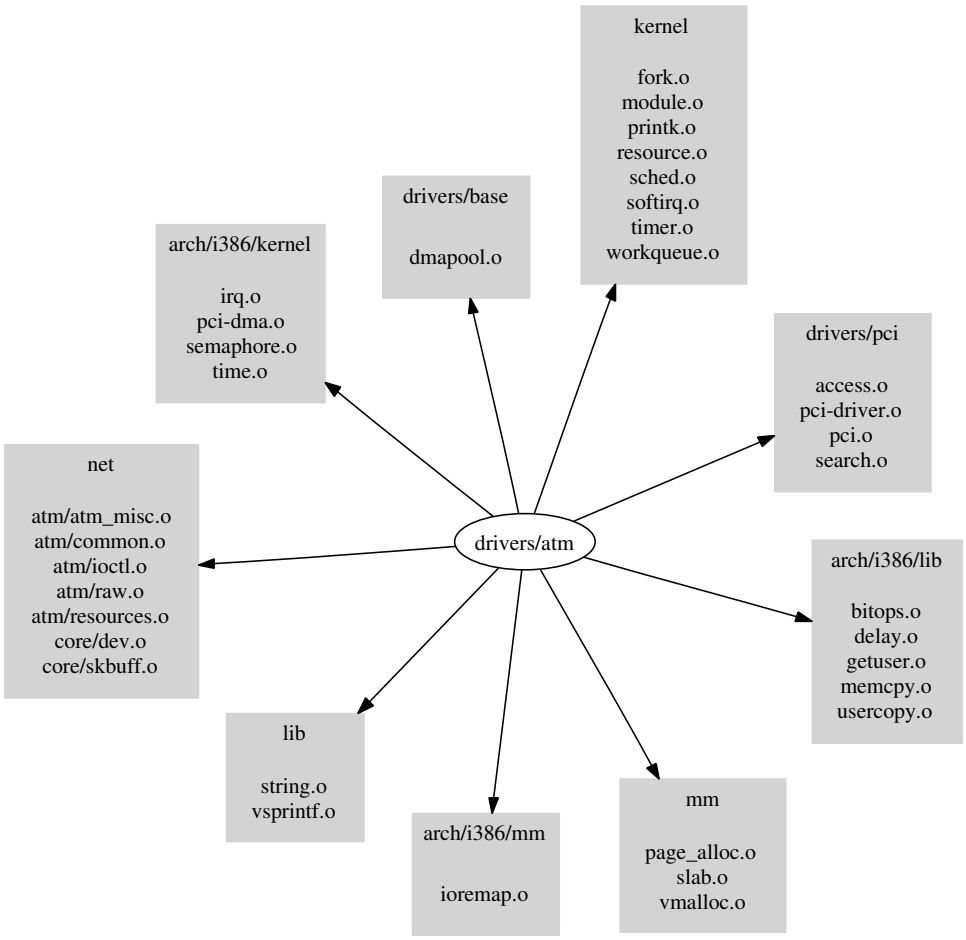

Figure 18: External dependencies of the *drivers/atm* subsystem

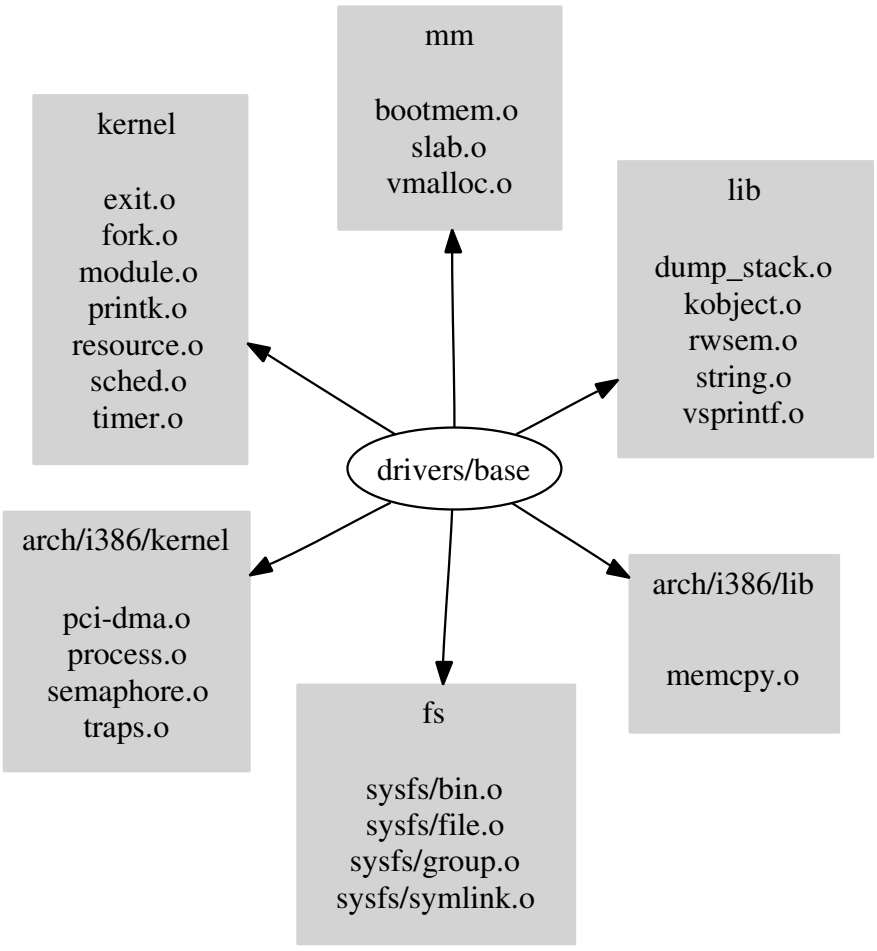

Figure 19: External dependencies of the *drivers/base* subsystem

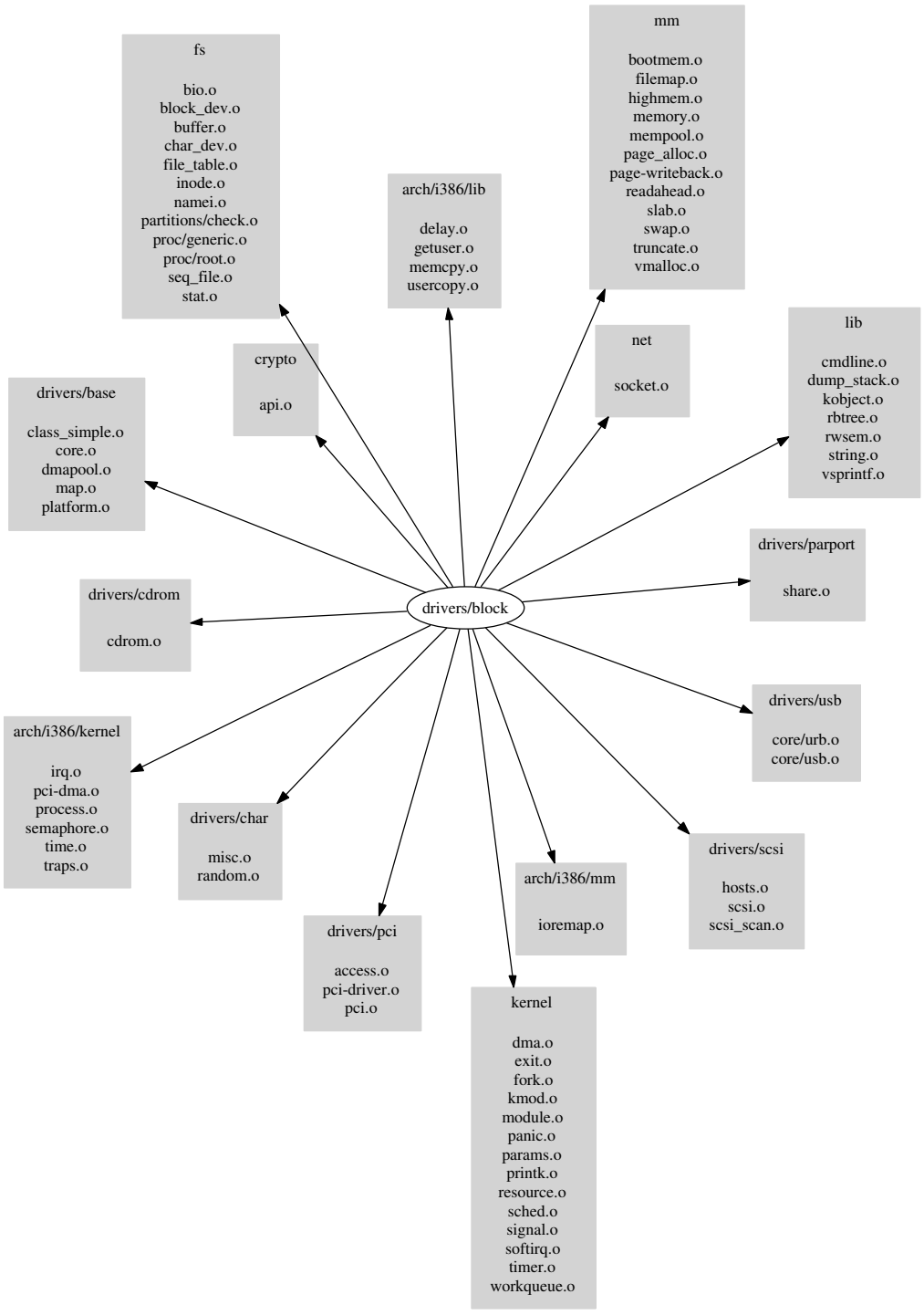

Figure 20: External dependencies of the *drivers/block* subsystem

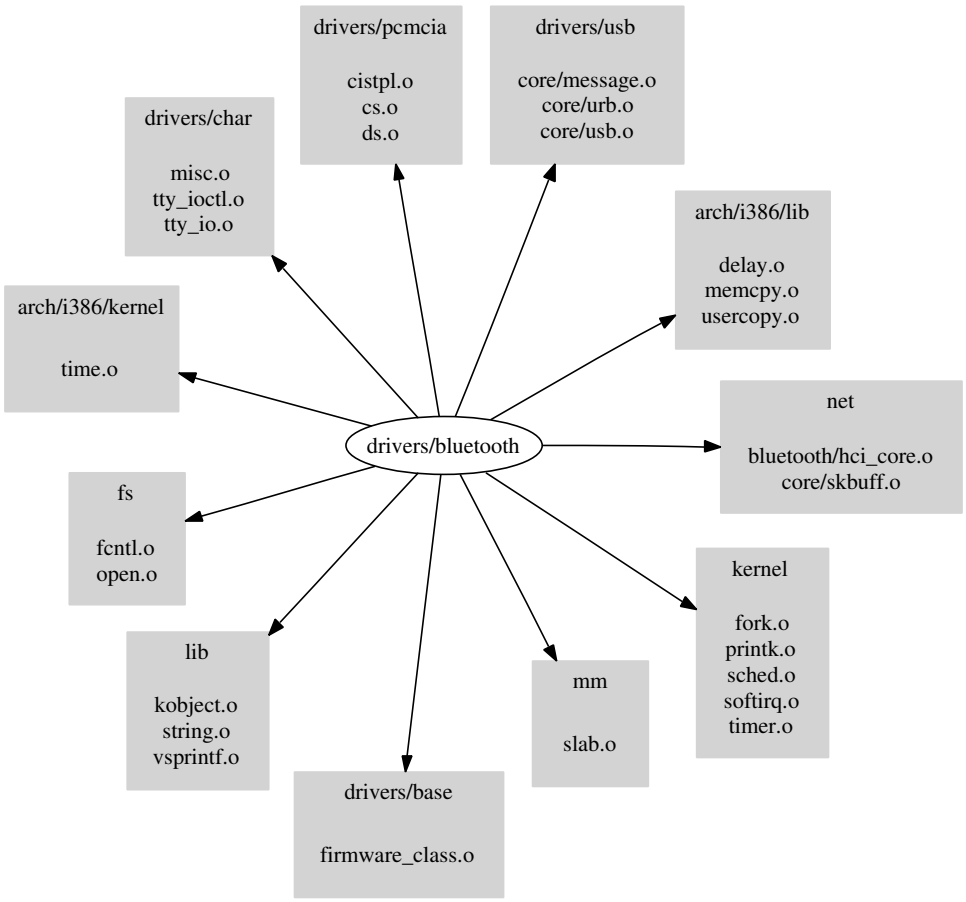

Figure 21: External dependencies of the *drivers/bluetooth* subsystem

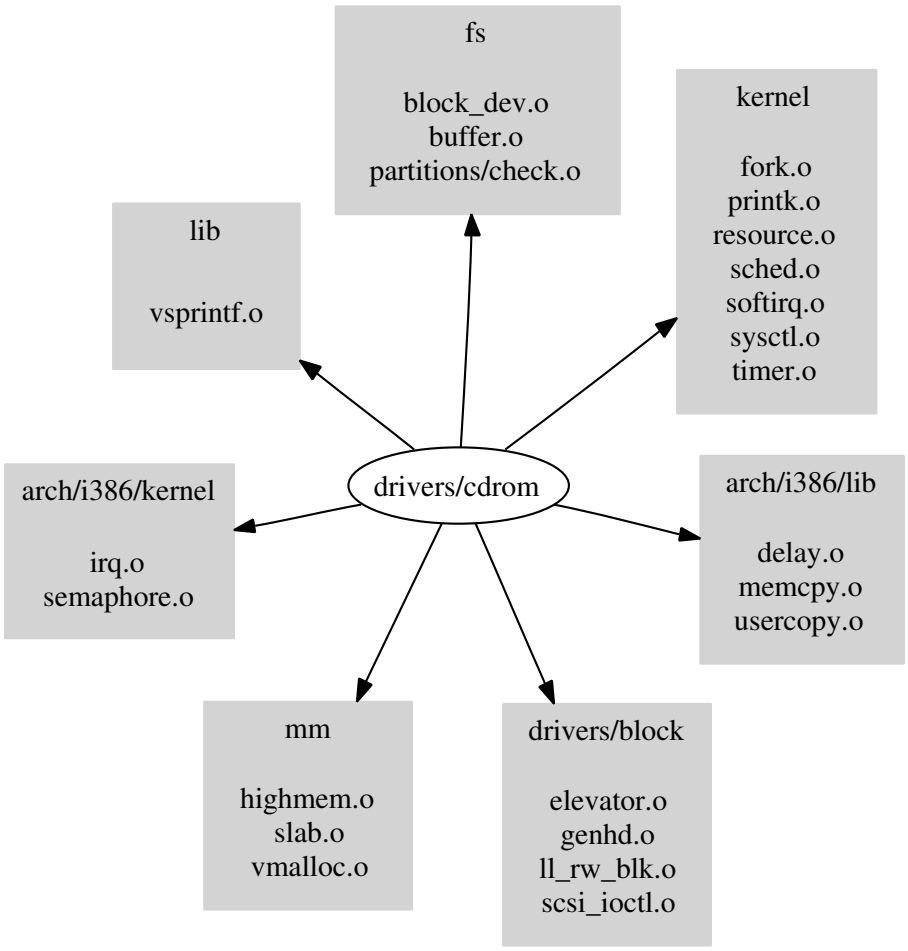

Figure 22: External dependencies of the *drivers/cdrom* subsystem

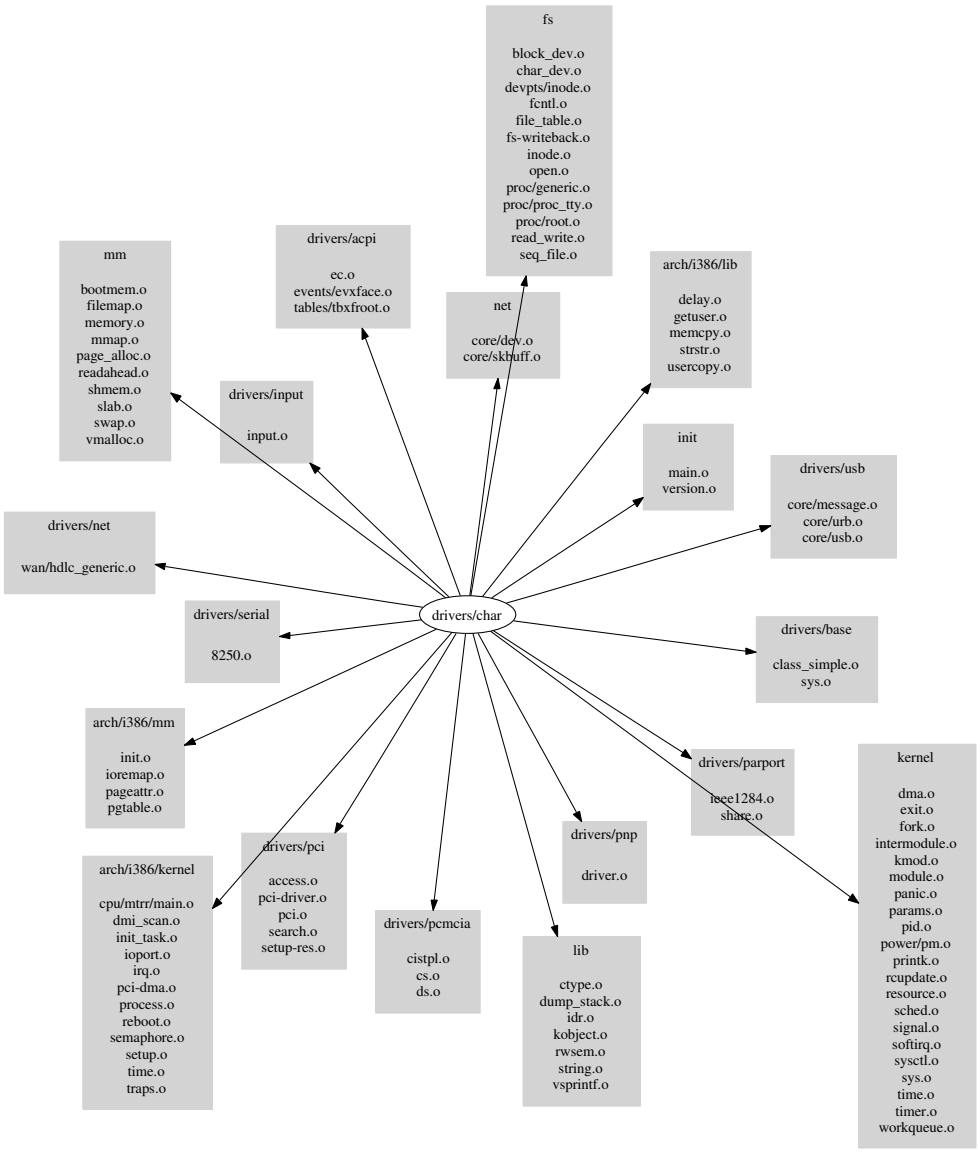

Figure 23: External dependencies of the *drivers/char* subsystem

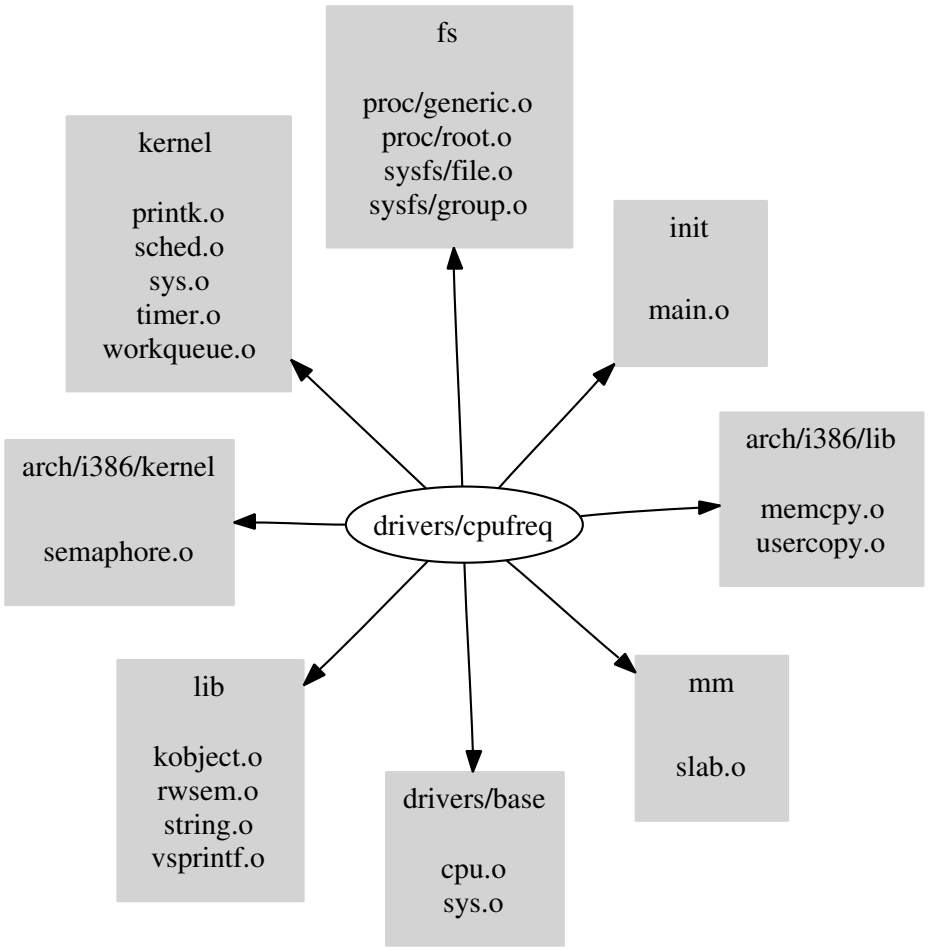

Figure 24: External dependencies of the *drivers/cpufreq* subsystem

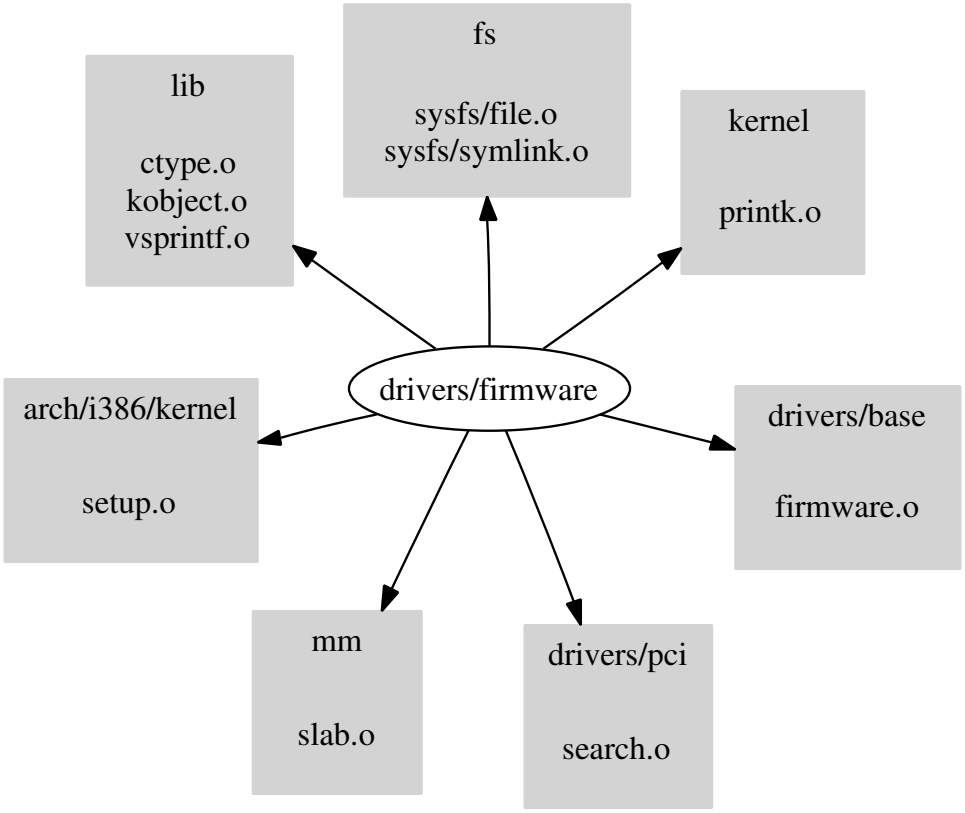

Figure 25: External dependencies of the *drivers/firmware* subsystem

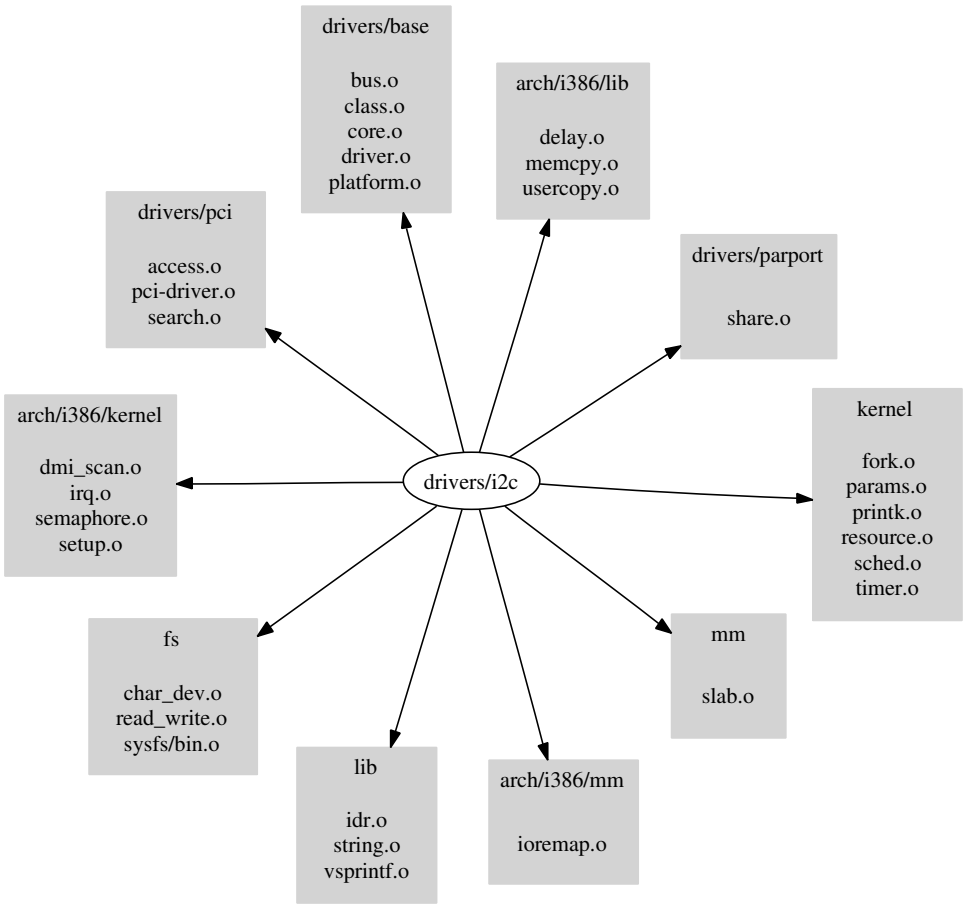

Figure 26: External dependencies of the *drivers/i2c* subsystem

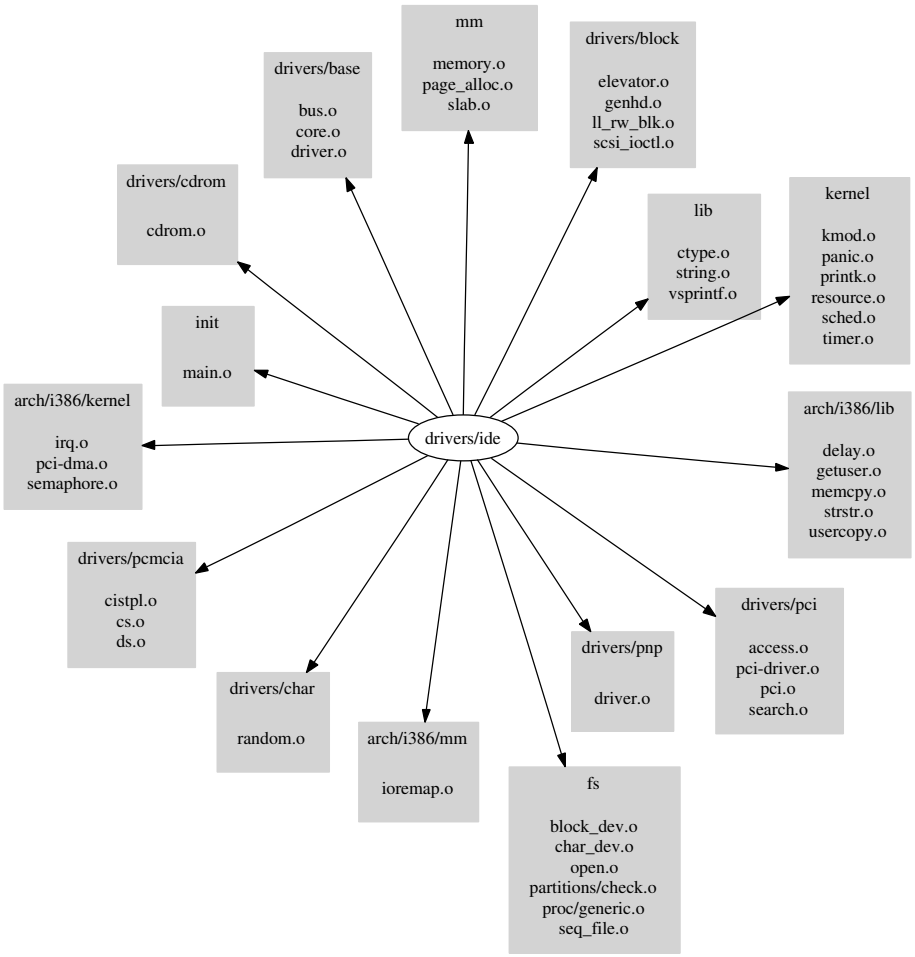

Figure 27: External dependencies of the *drivers/ide* subsystem

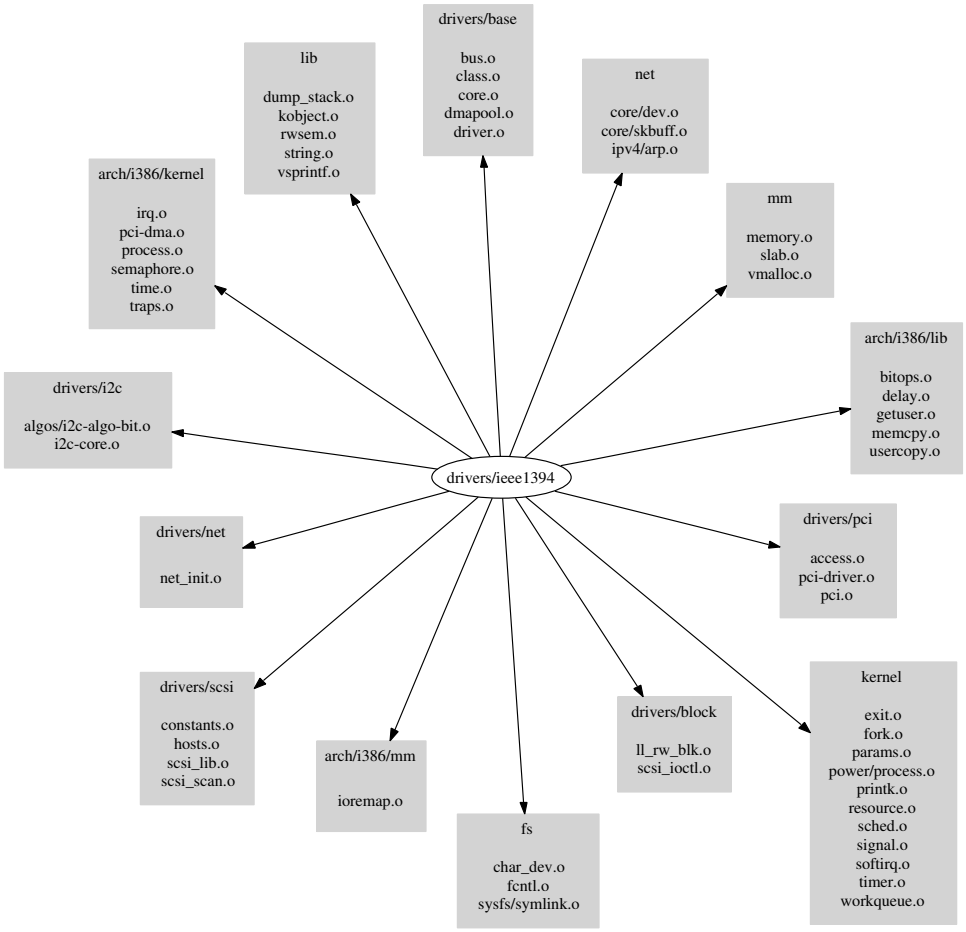

Figure 28: External dependencies of the *drivers/ieee1394* subsystem

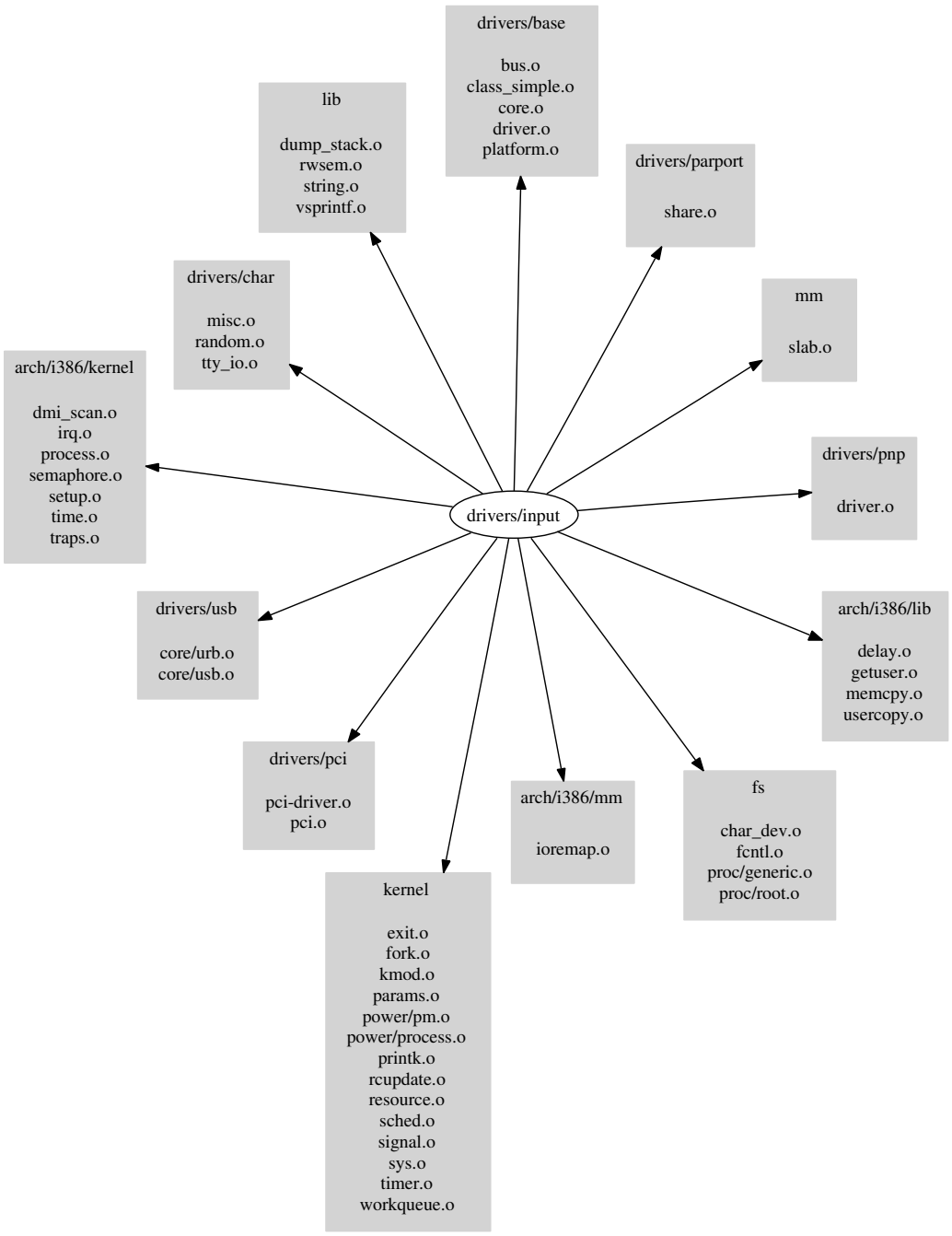

Figure 29: External dependencies of the *drivers/input* subsystem

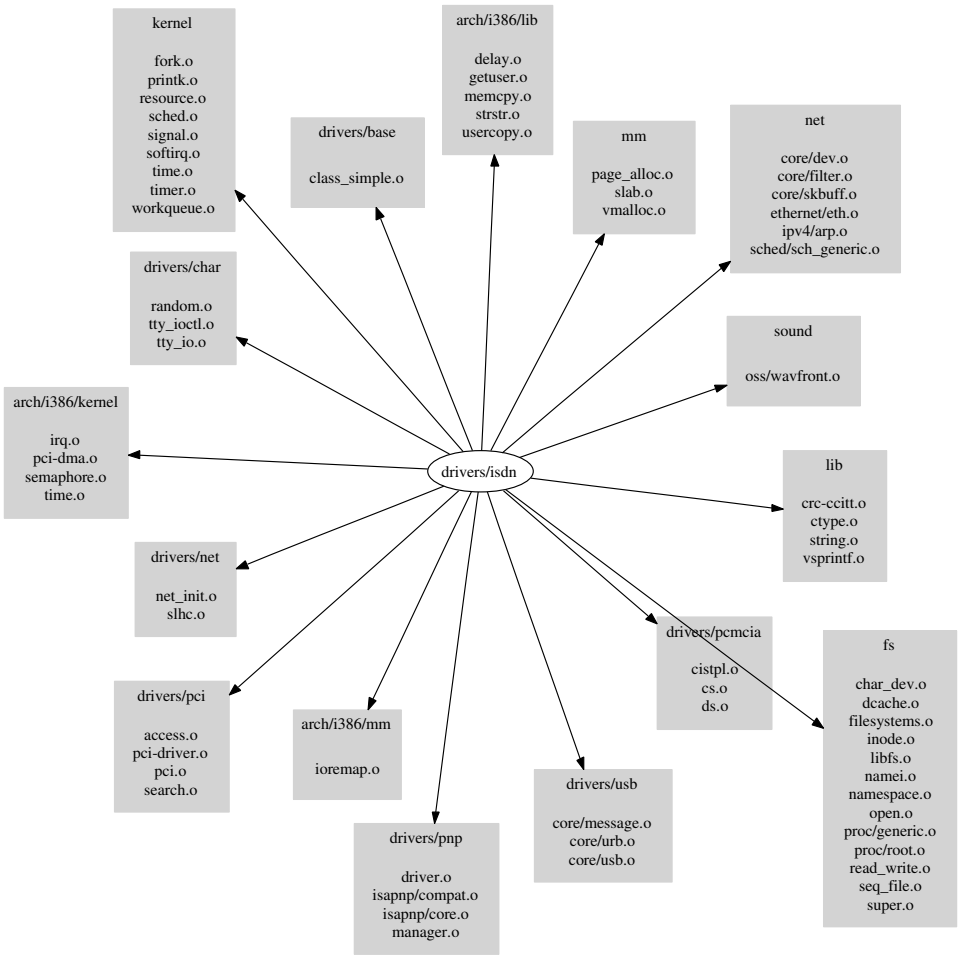

Figure 30: External dependencies of the *drivers/isdn* subsystem

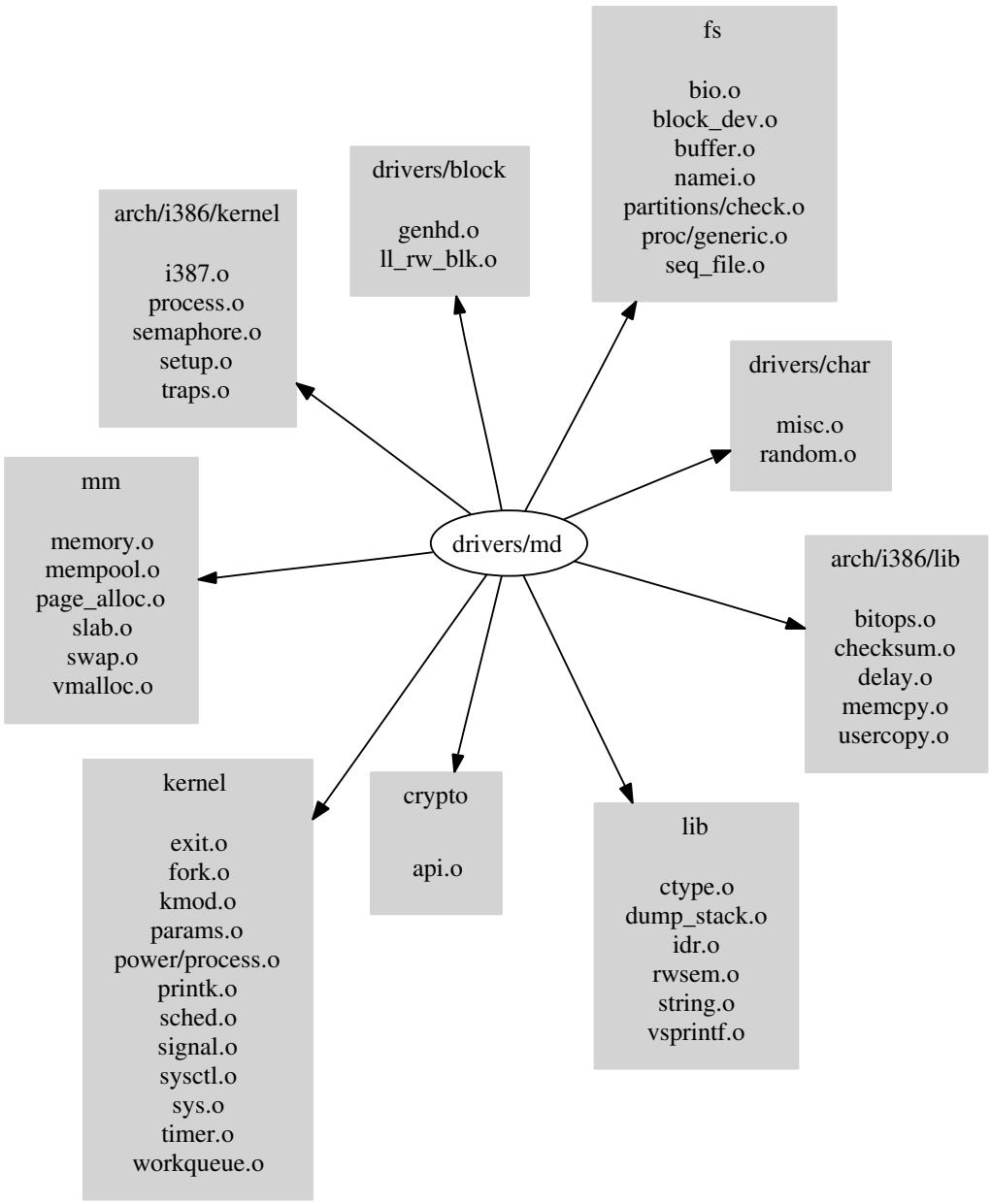

Figure 31: External dependencies of the *drivers/md* subsystem

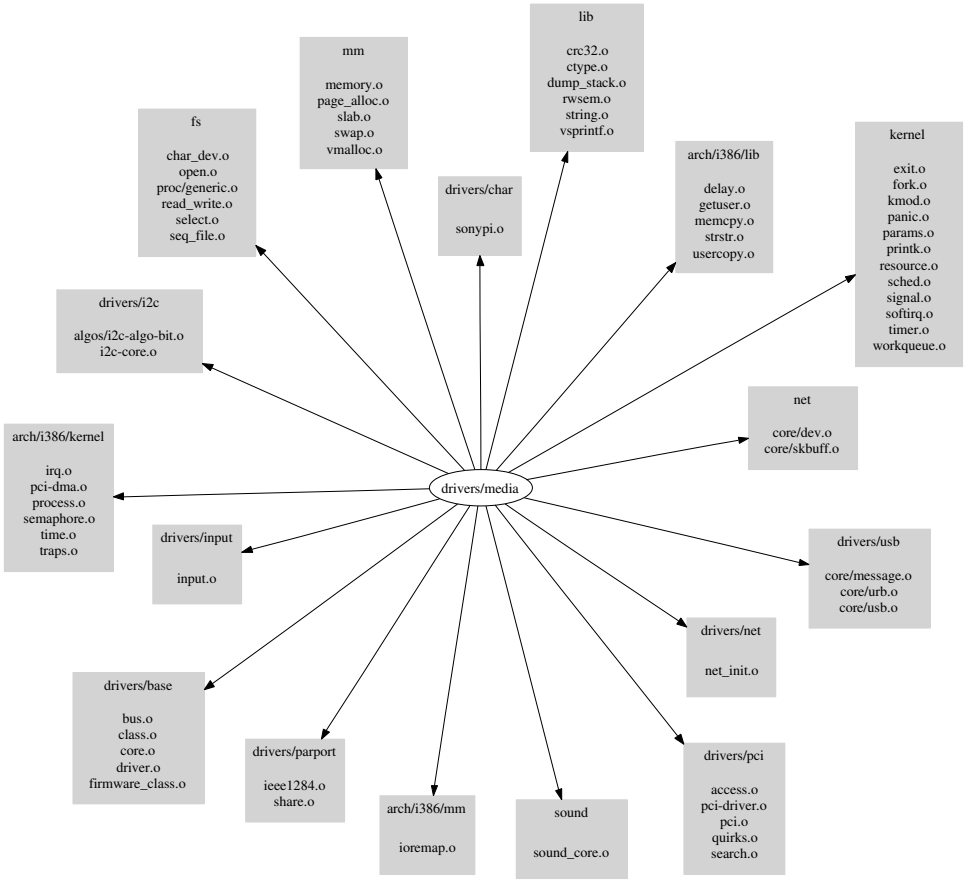

Figure 32: External dependencies of the *drivers/media* subsystem

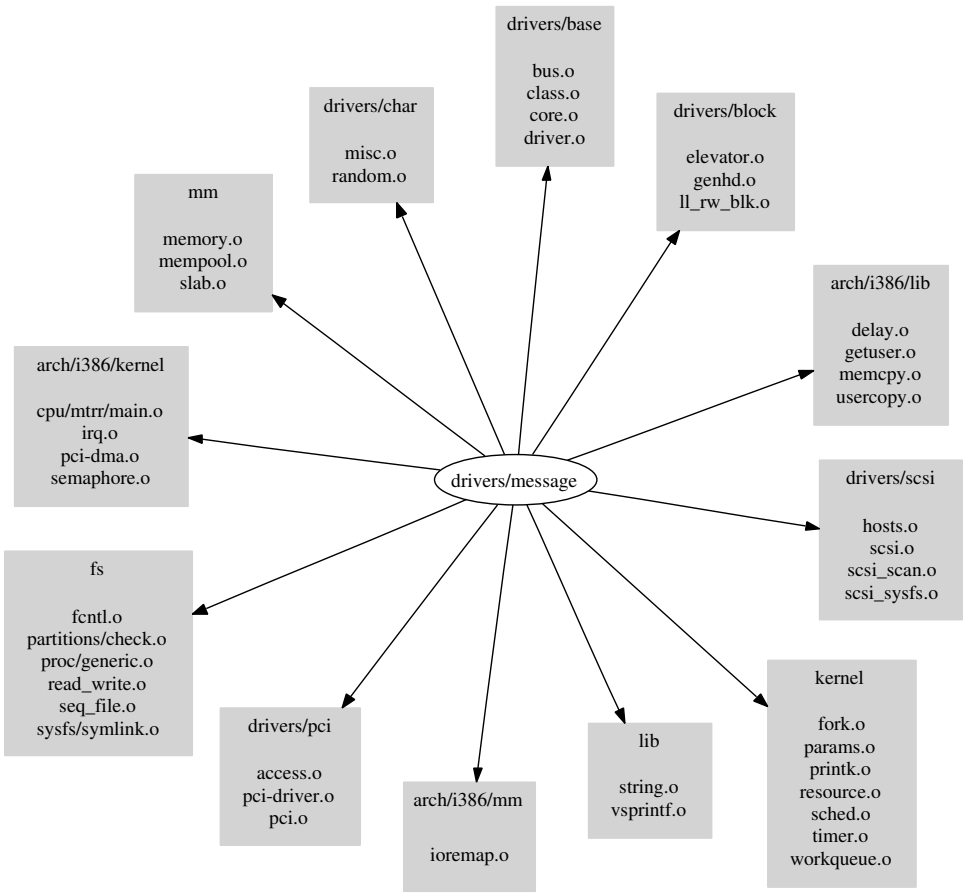

Figure 33: External dependencies of the *drivers/message* subsystem

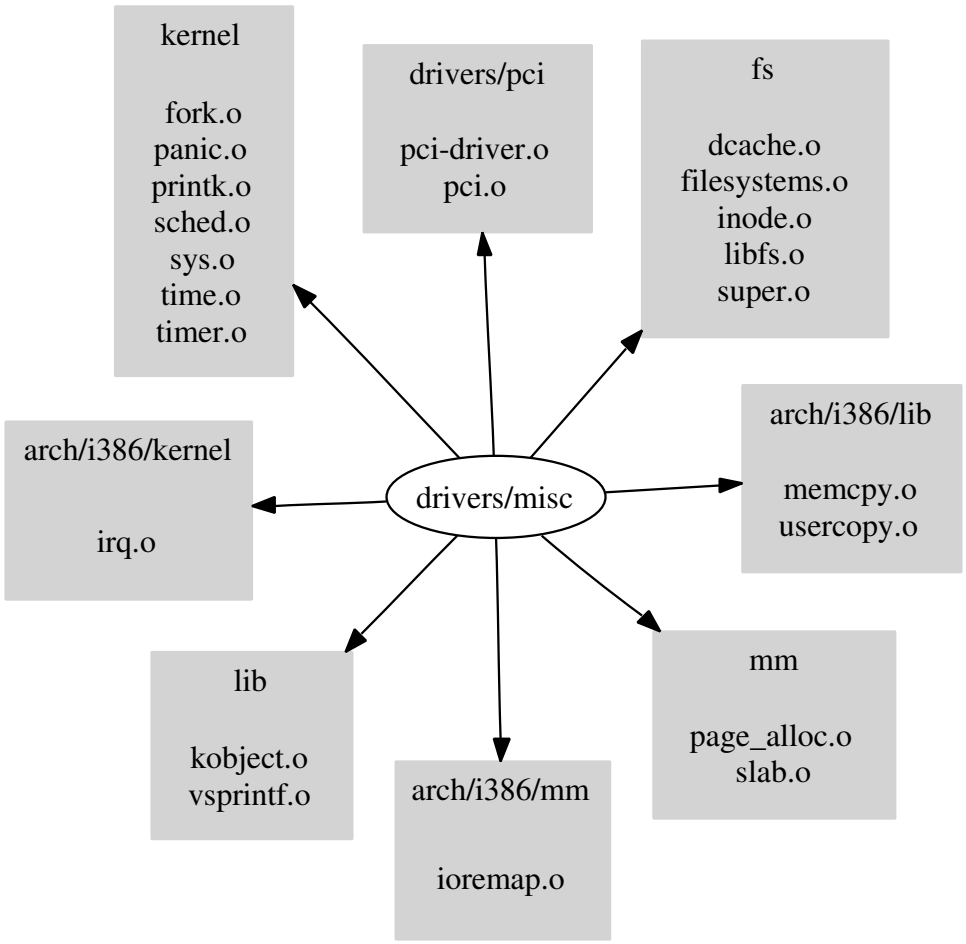

Figure 34: External dependencies of the *drivers/misc* subsystem

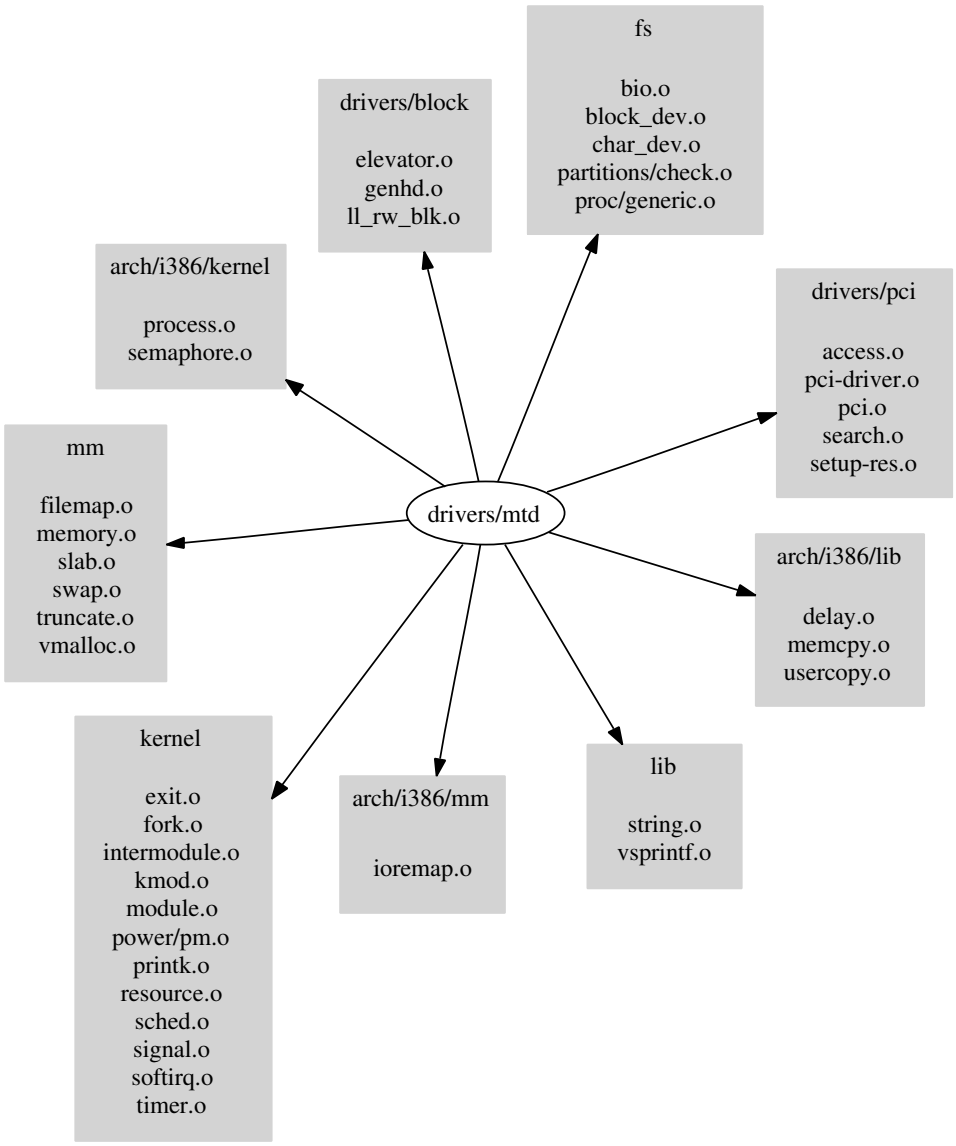

Figure 35: External dependencies of the *drivers/mtd* subsystem

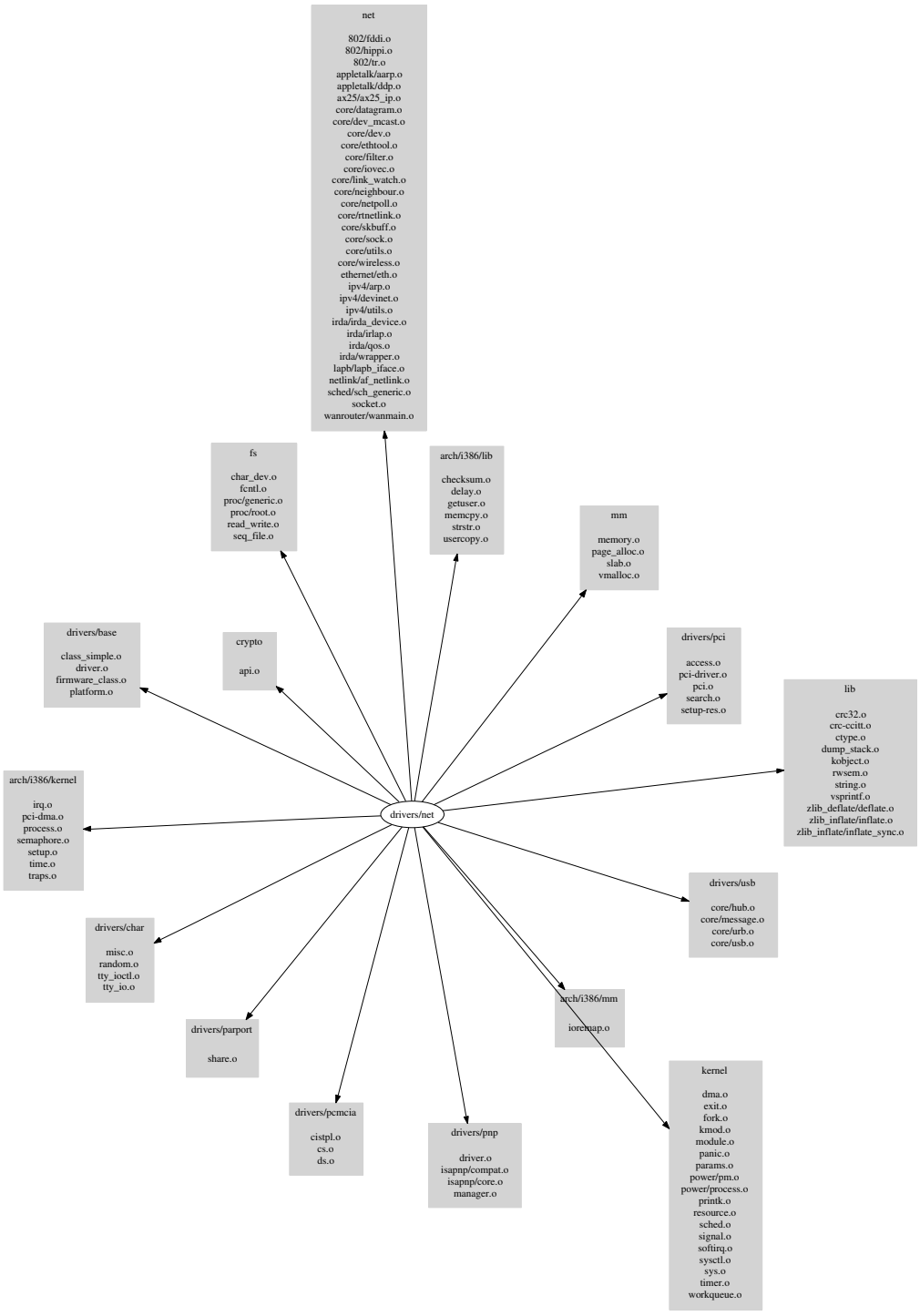

Figure 36: External dependencies of the *drivers/net* subsystem

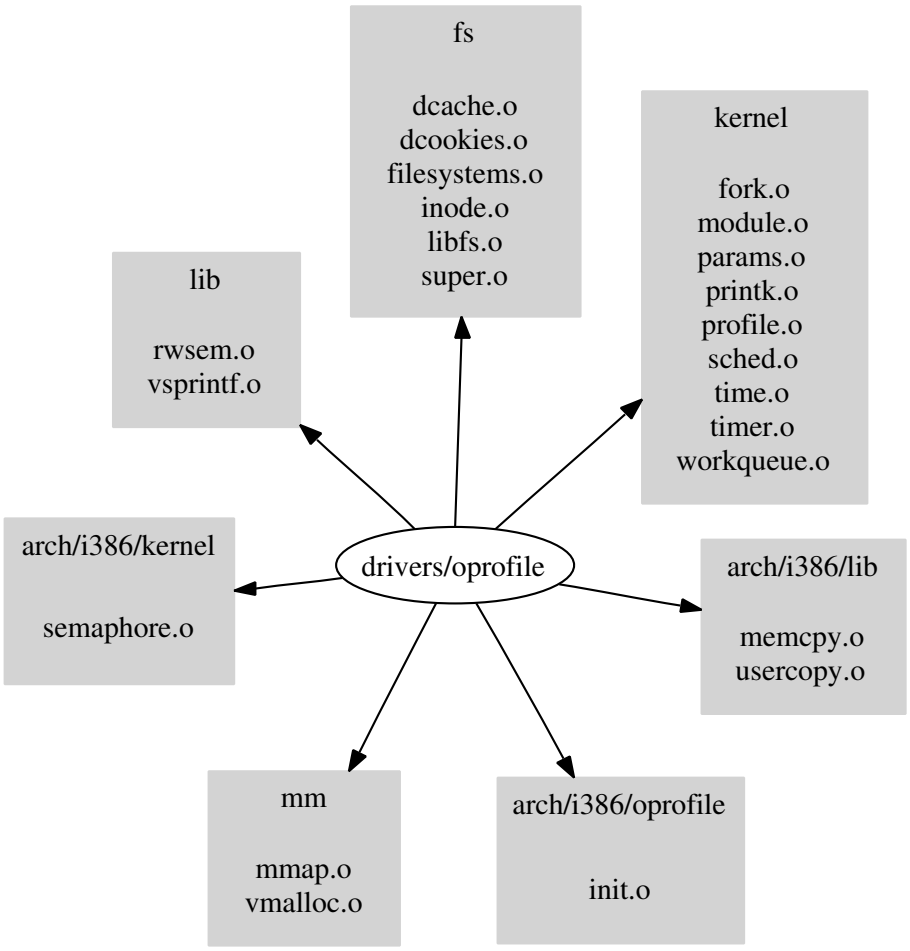

Figure 37: External dependencies of the *drivers/oprofile* subsystem

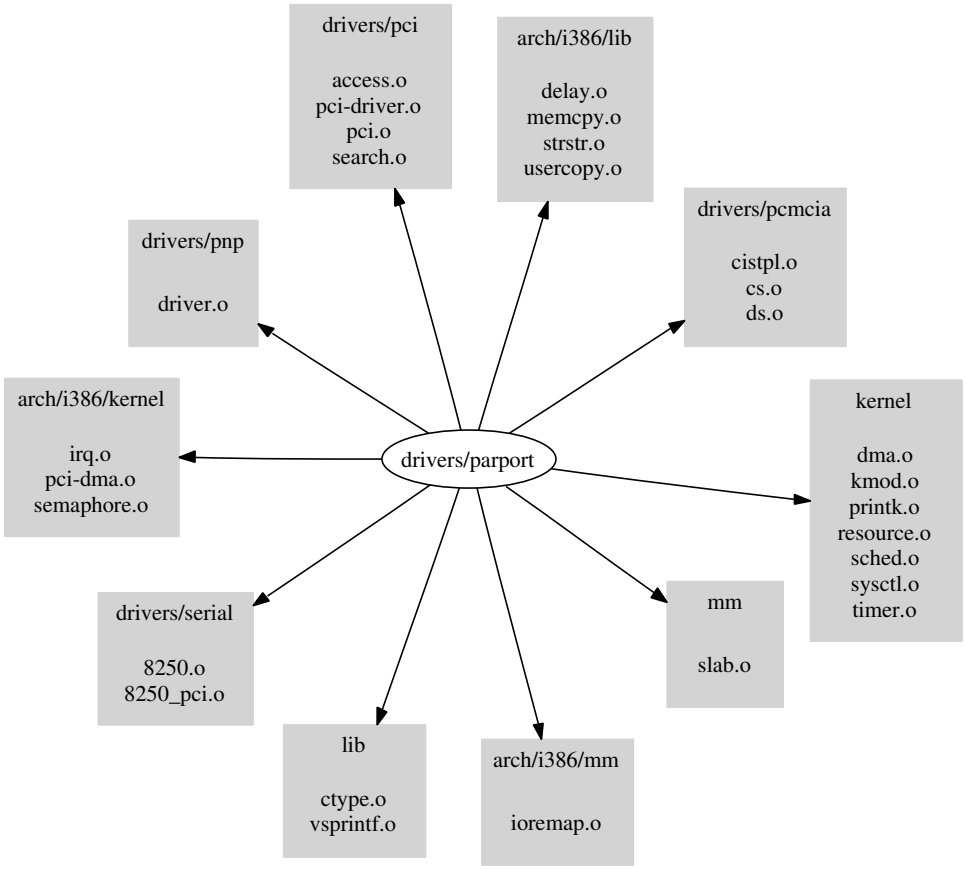

Figure 38: External dependencies of the *drivers/parport* subsystem

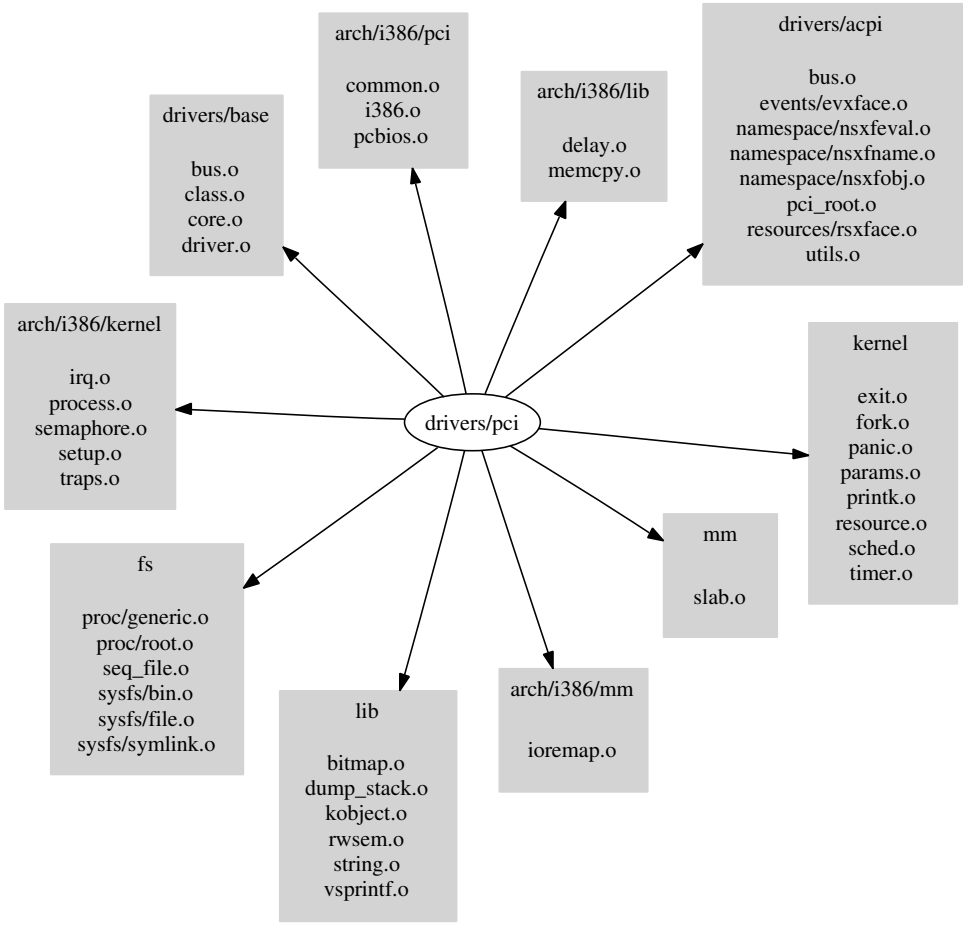

Figure 39: External dependencies of the *drivers/pci* subsystem

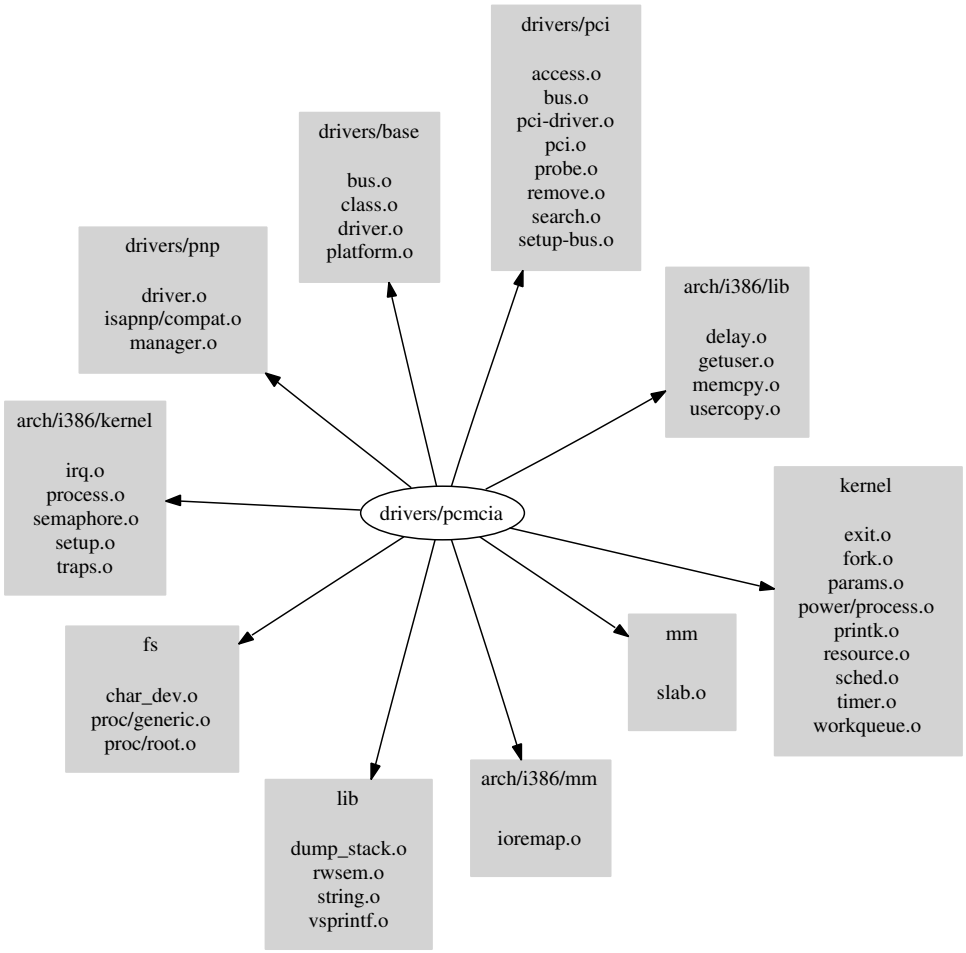

Figure 40: External dependencies of the *drivers/pcmcia* subsystem

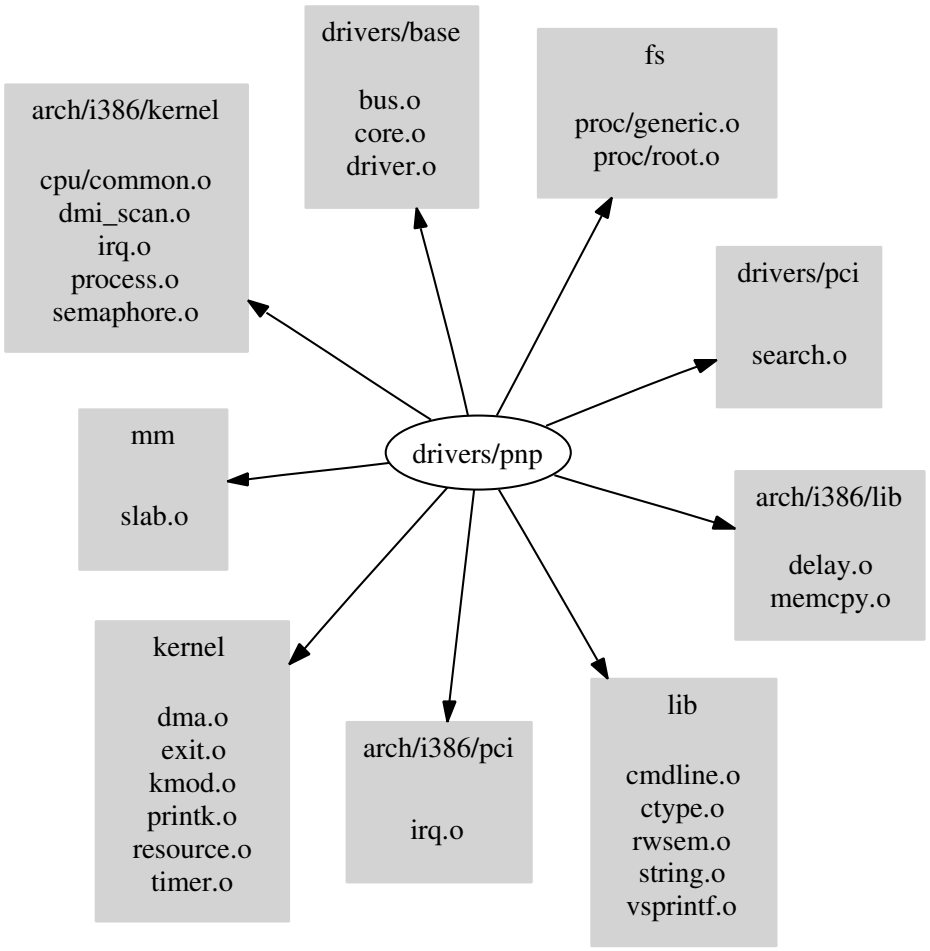

Figure 41: External dependencies of the *drivers/pnp* subsystem

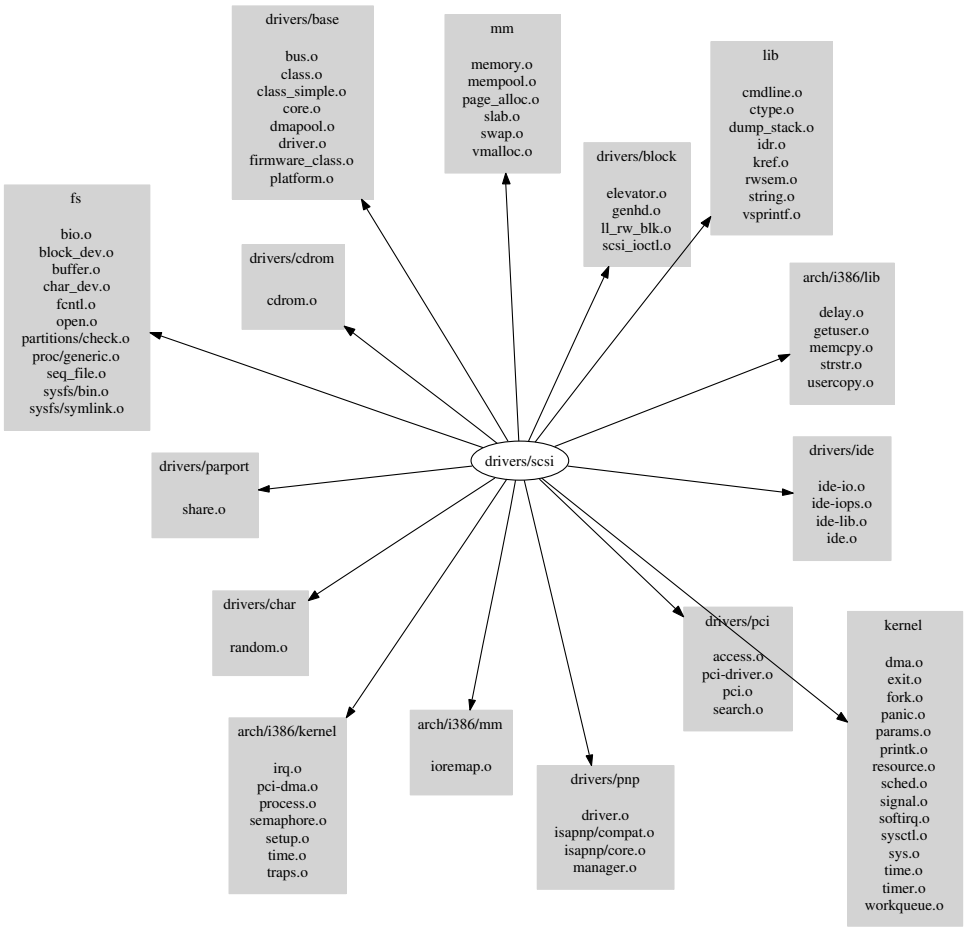

Figure 42: External dependencies of the *drivers/scsi* subsystem

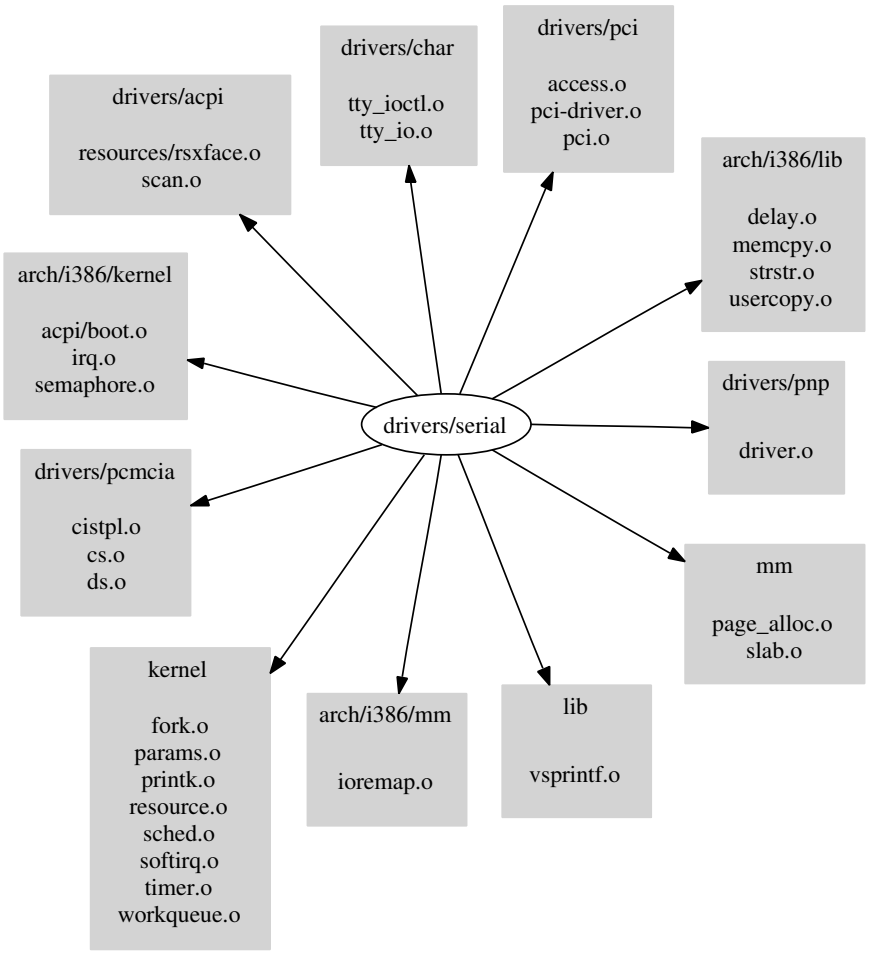

Figure 43: External dependencies of the *drivers/serial* subsystem

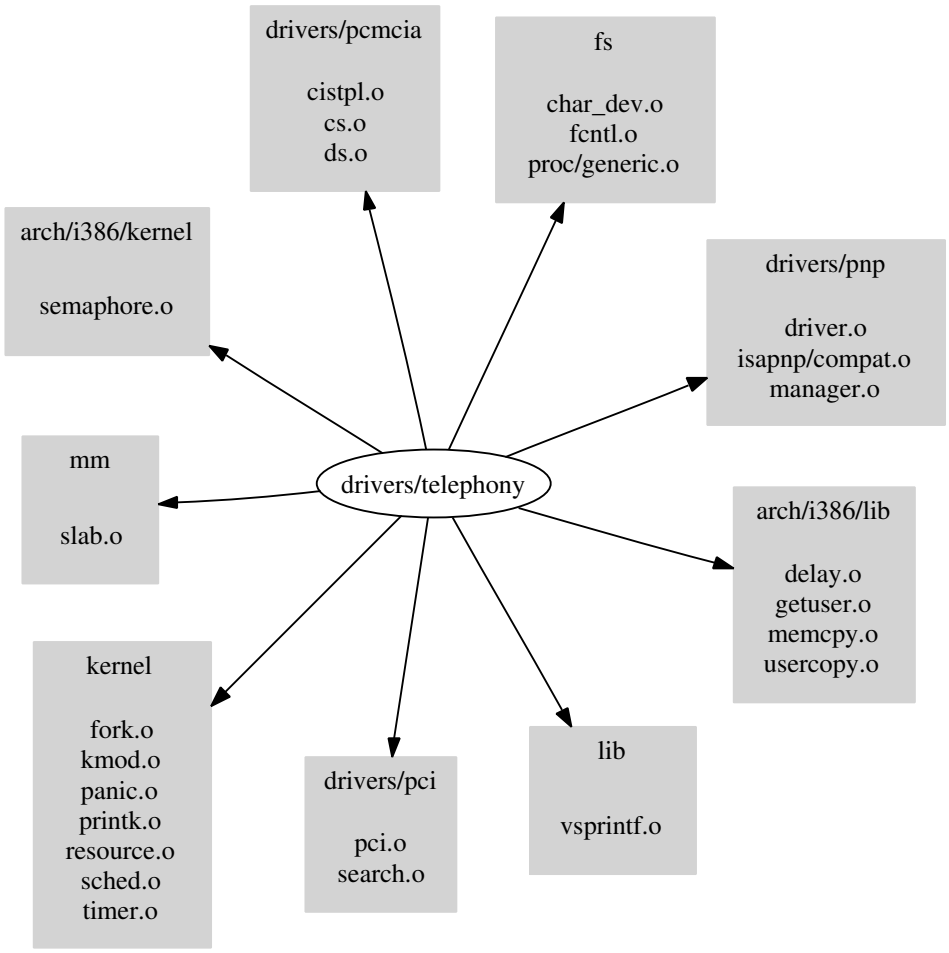

Figure 44: External dependencies of the *drivers/telephony* subsystem

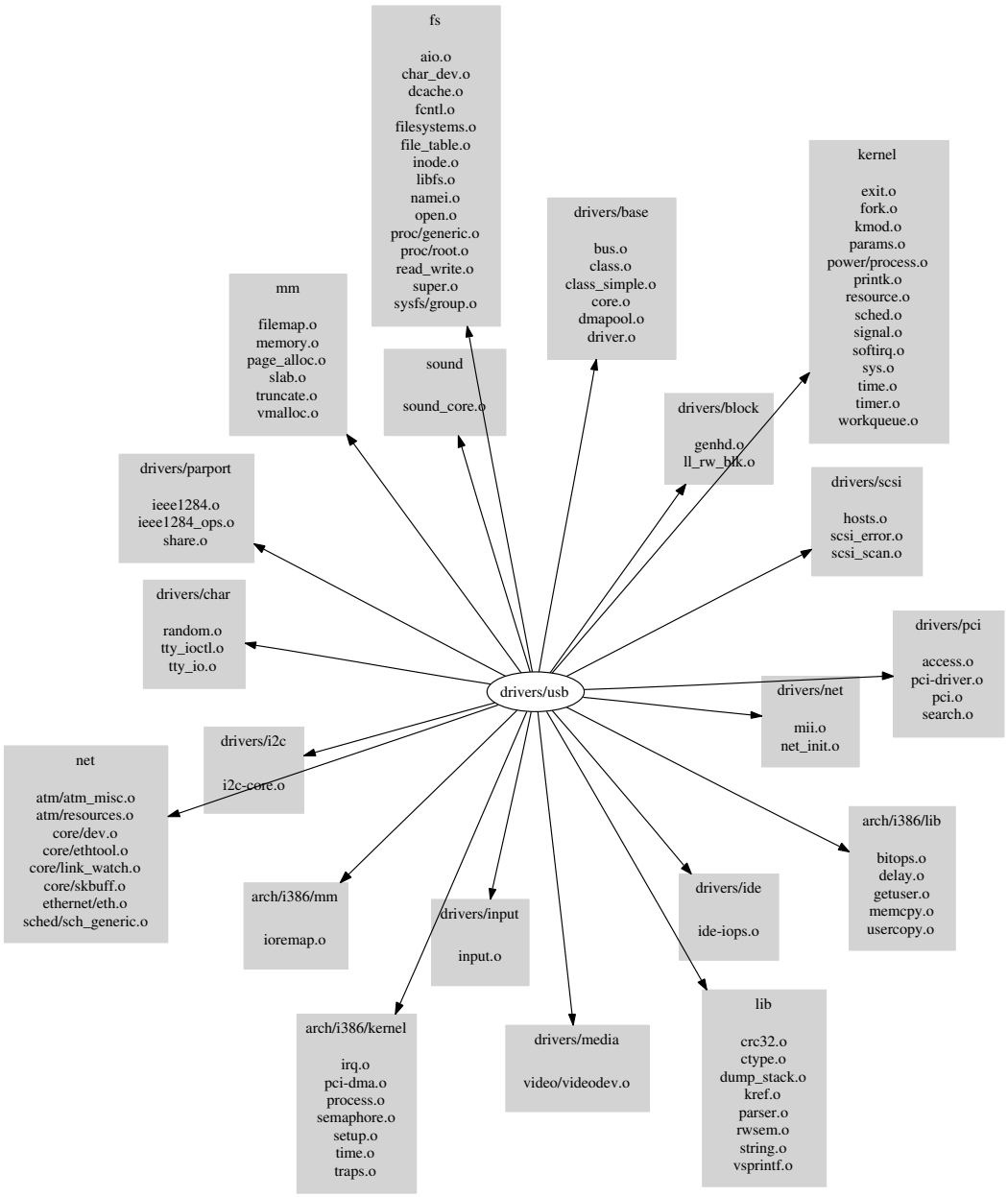

Figure 45: External dependencies of the *drivers/usb* subsystem

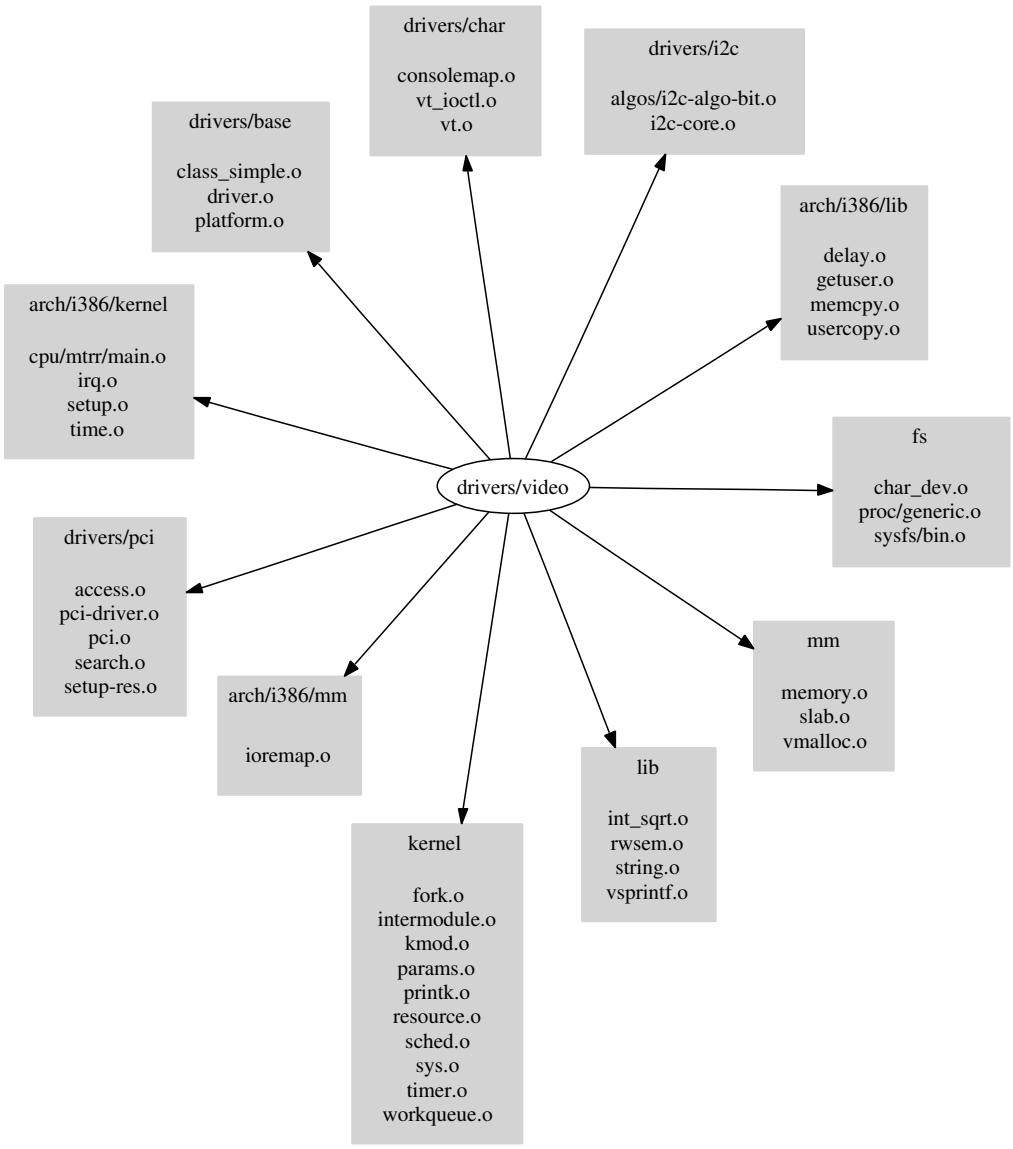

Figure 46: External dependencies of the *drivers/video* subsystem

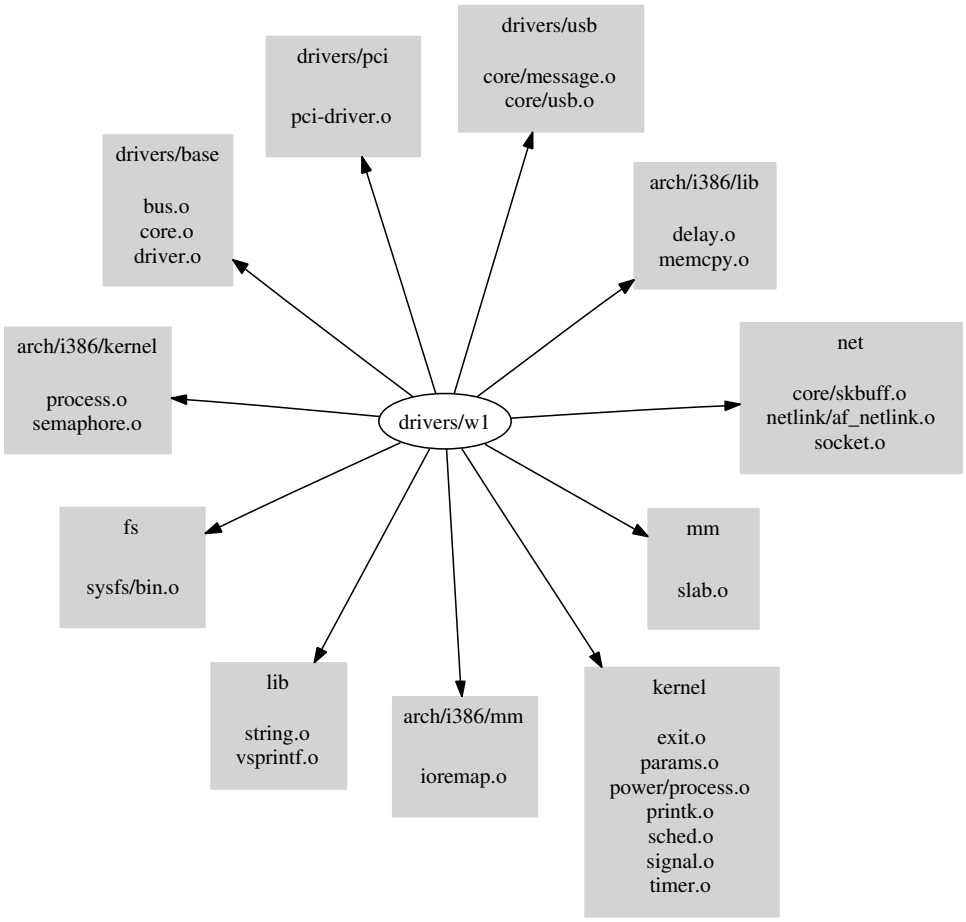

Figure 47: External dependencies of the *drivers/w1* subsystem

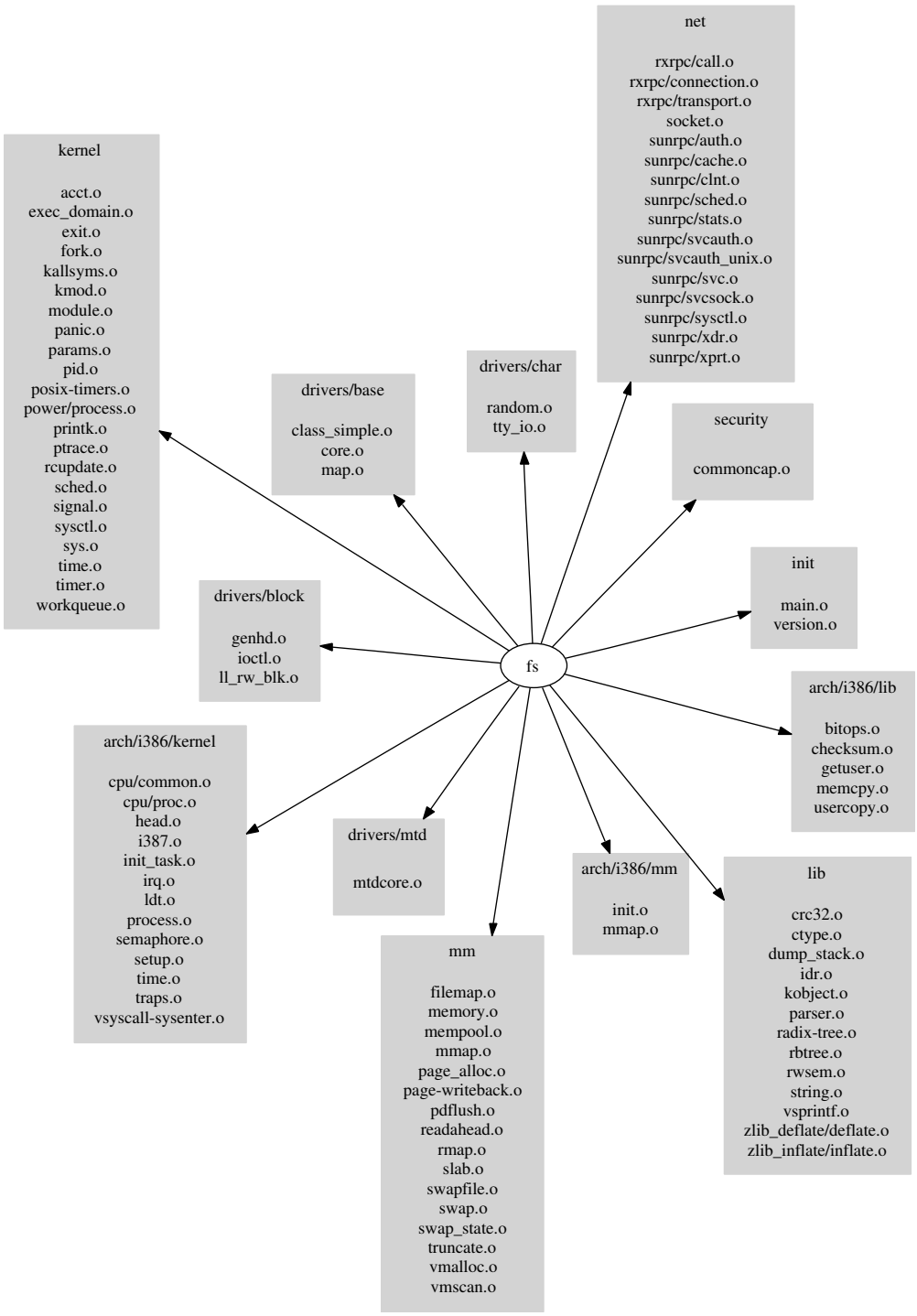

Figure 48: External dependencies of the *fs* subsystem

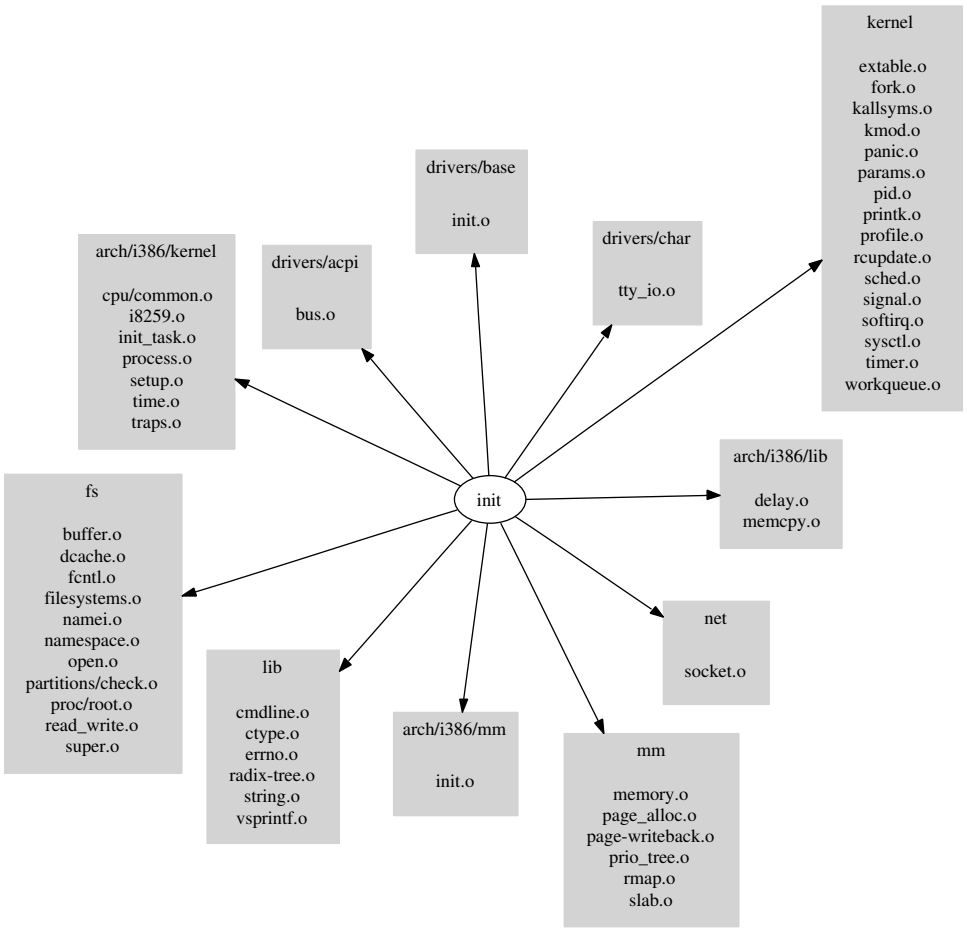

Figure 49: External dependencies of the *init* subsystem

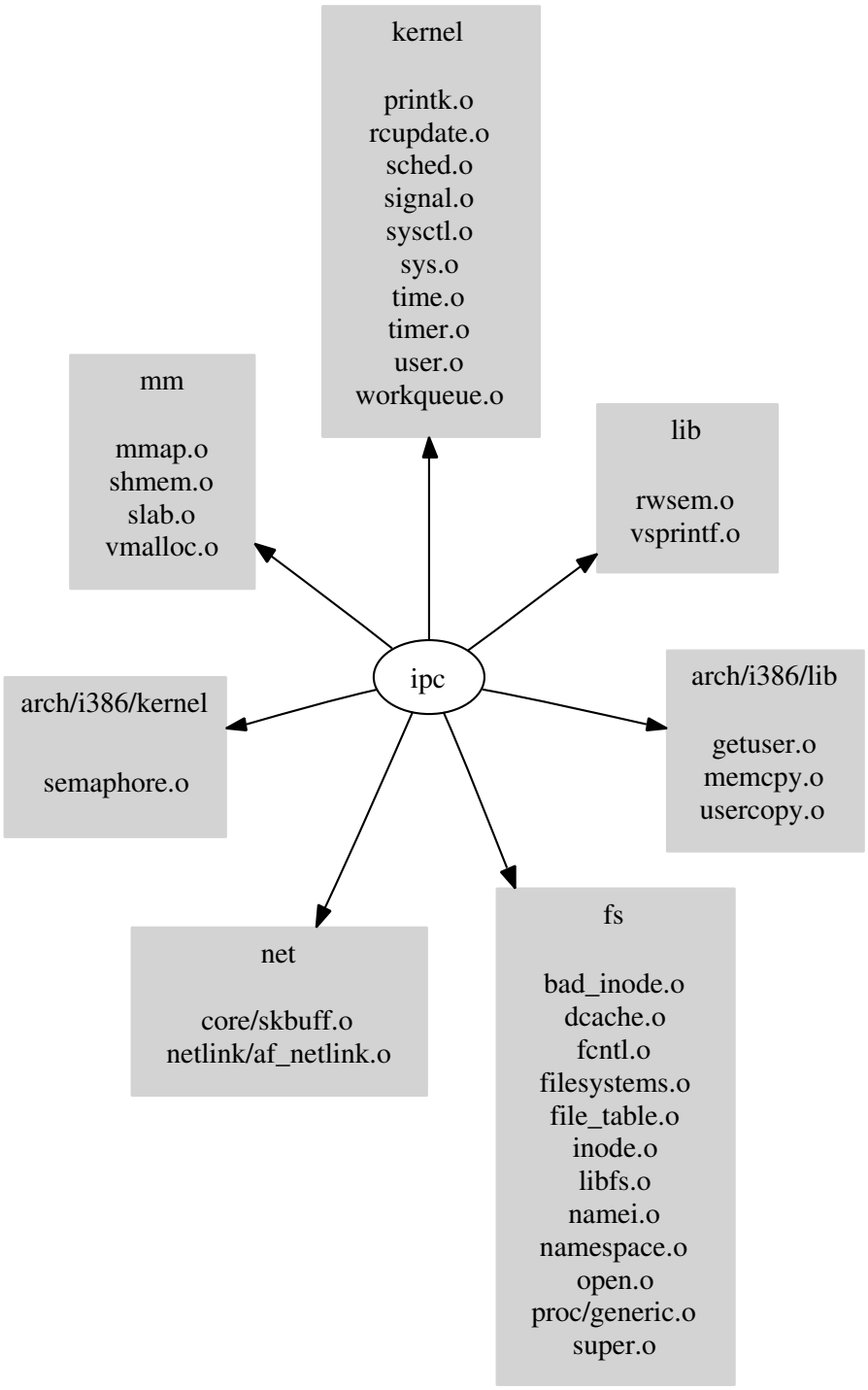

Figure 50: External dependencies of the *ipc* subsystem

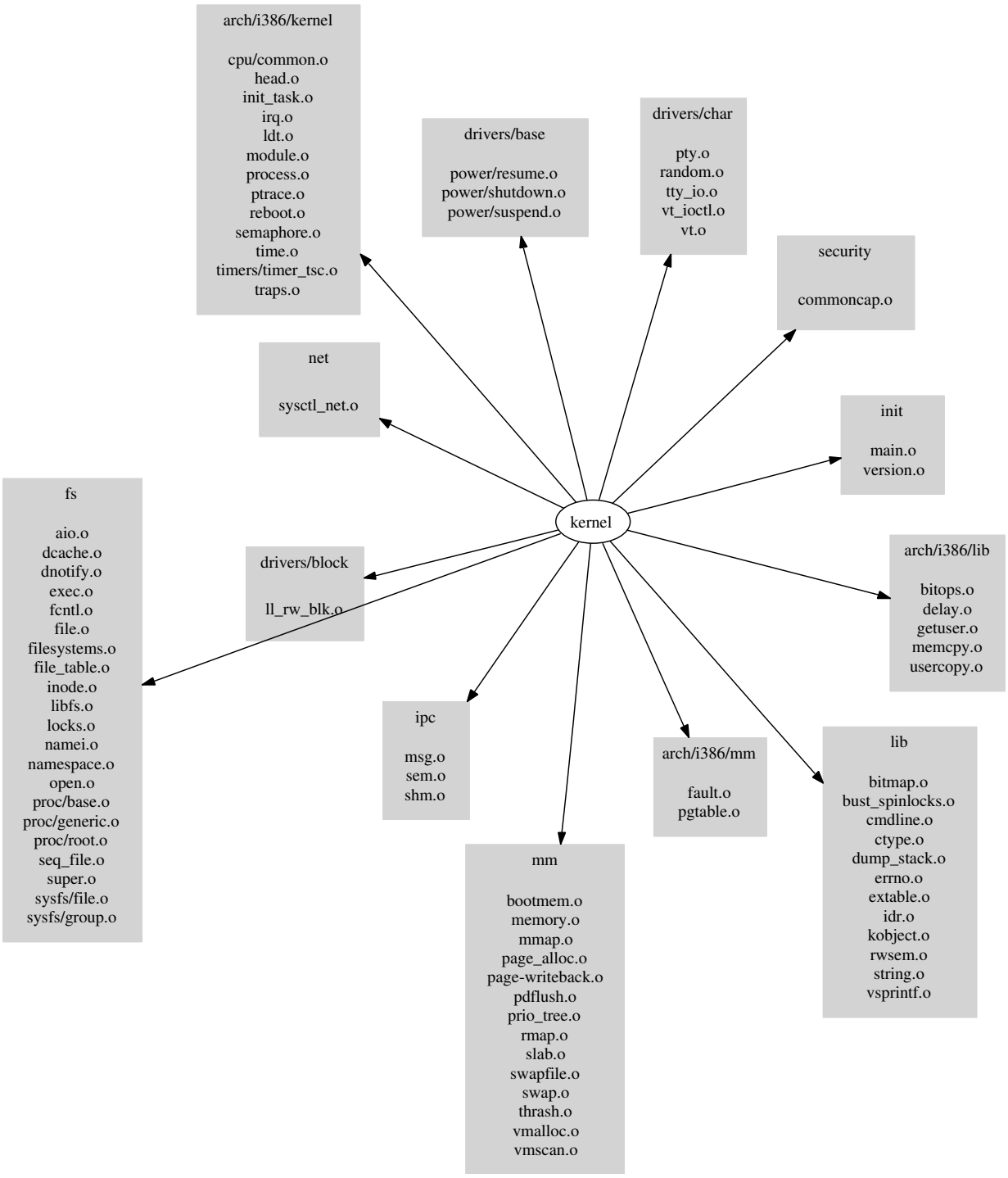

Figure 51: External dependencies of the *kernel* subsystem

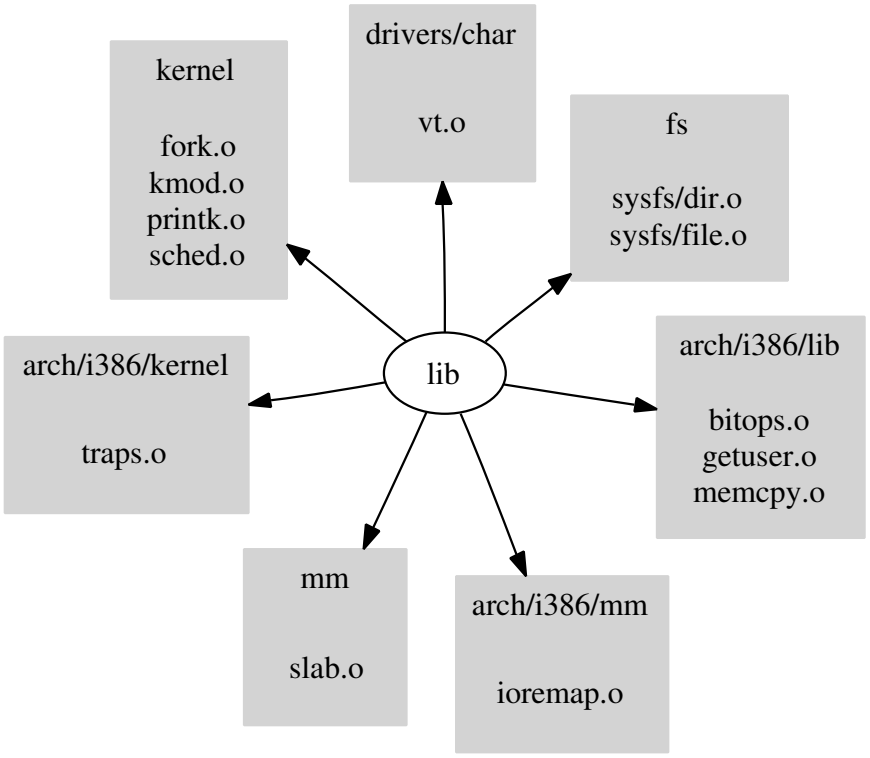

Figure 52: External dependencies of the *lib* subsystem

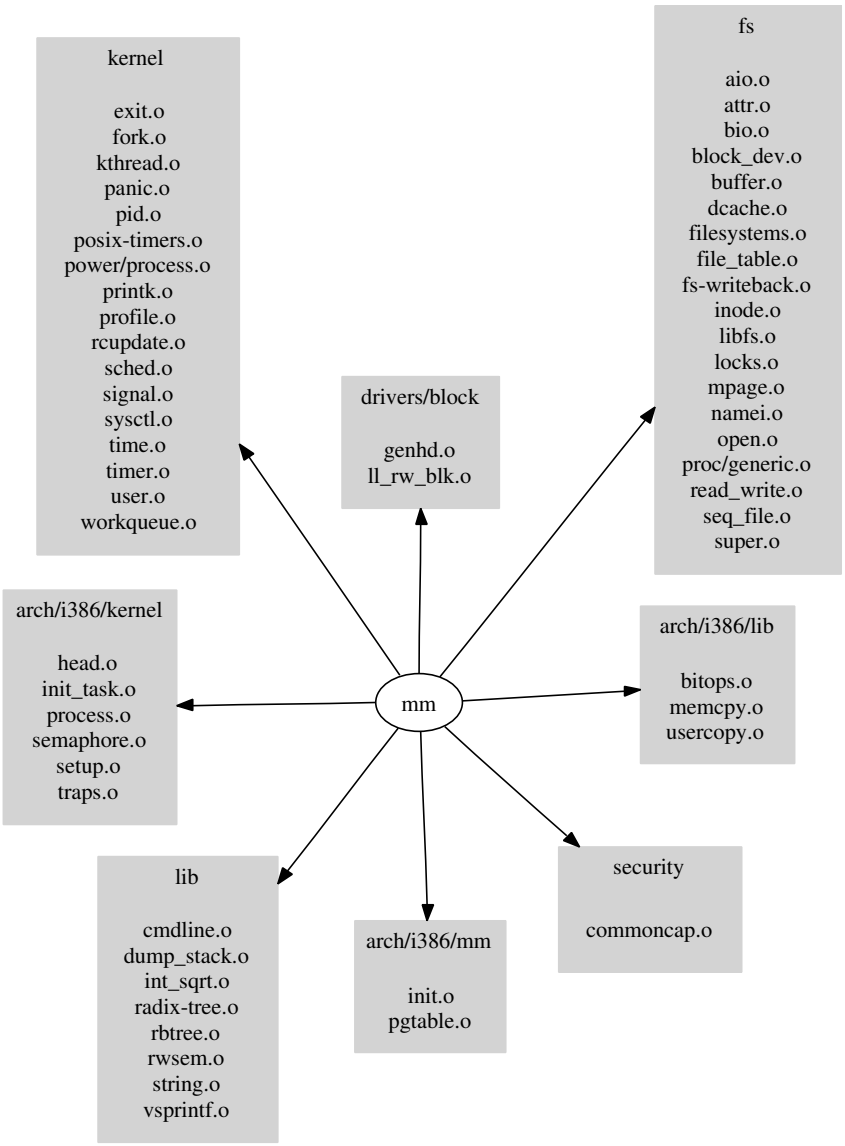

Figure 53: External dependencies of the *mm* subsystem

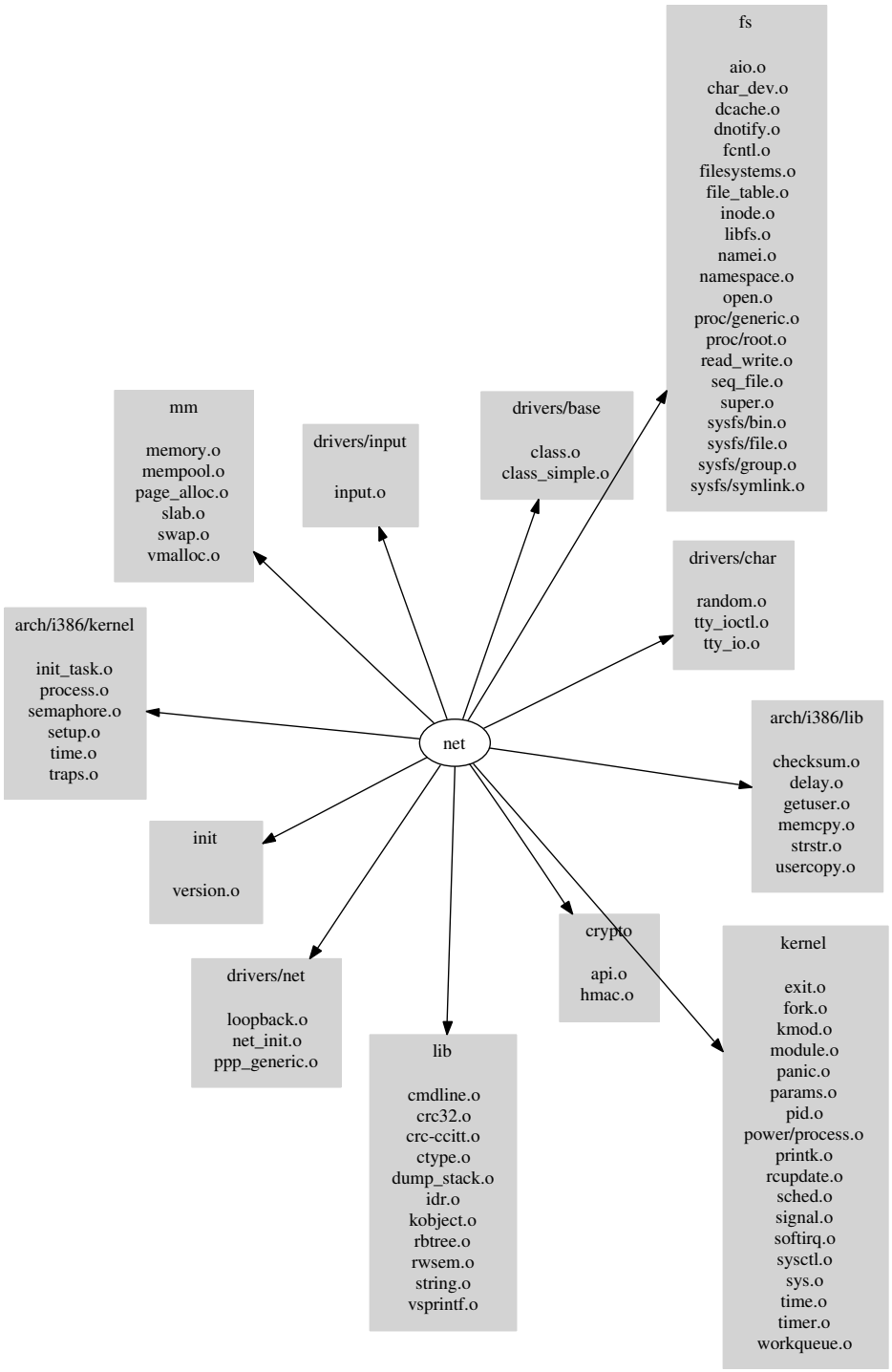

Figure 54: External dependencies of the *net* subsystem

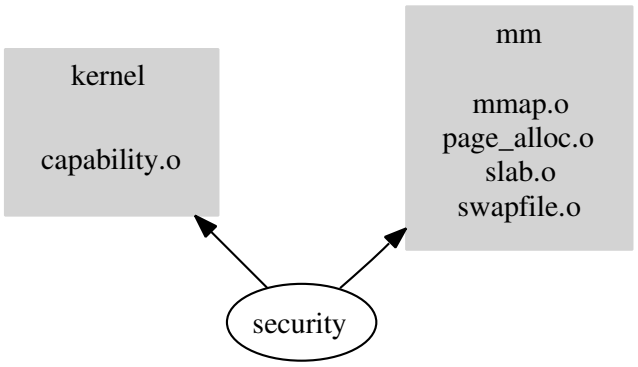

Figure 55: External dependencies of the *security* subsystem

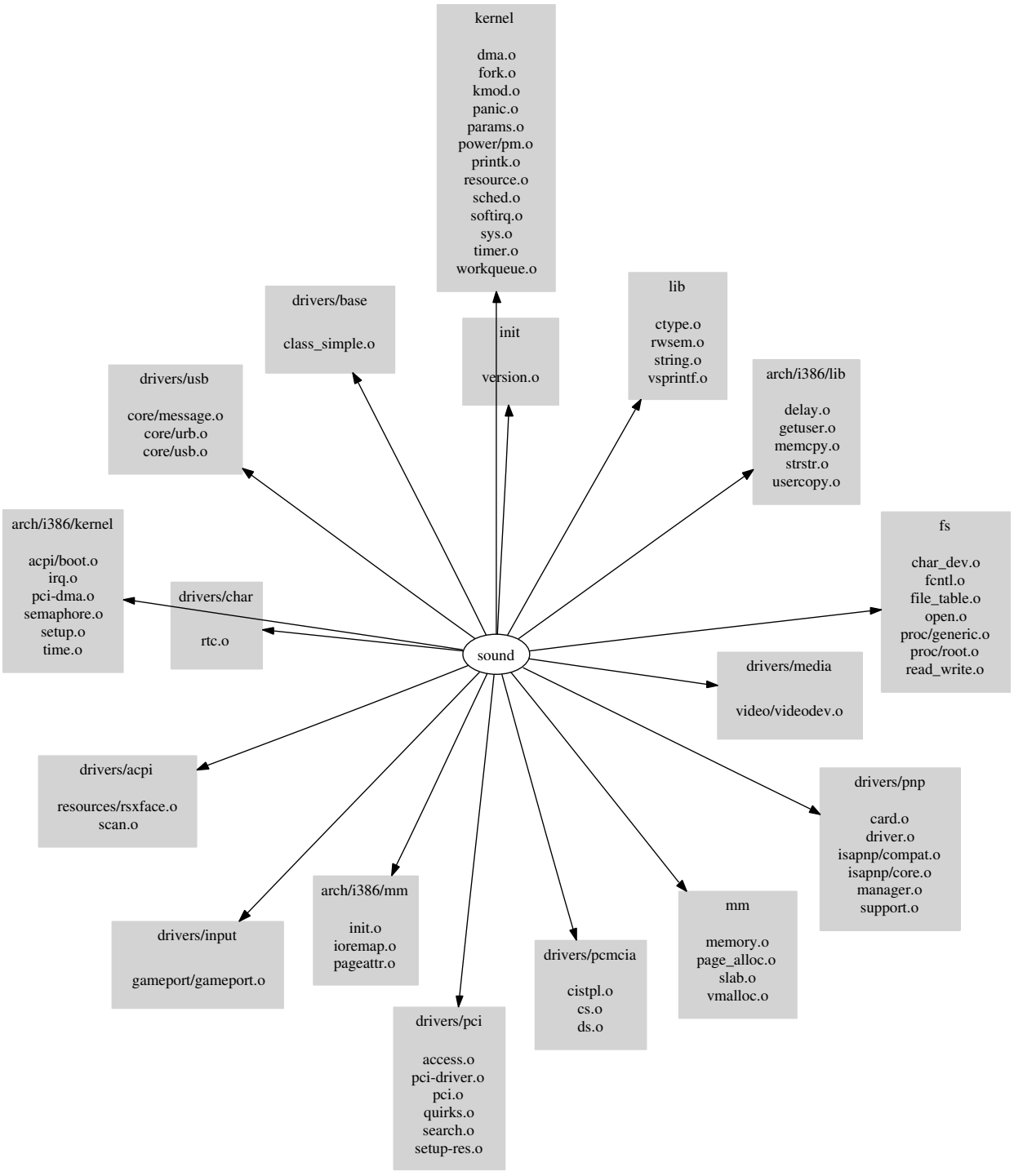

Figure 56: External dependencies of the *sound* subsystem

B Interface Relevance

Another result of my analysis is the frequency of usage of the different interfaces. This analysis is based on the assumption that every object file corresponds with one interface. The following table contains the analysis data. The first column contains the number of object files in different subsystems which depend on the object file which is named in the second column. The number in brackets is the number of different subsystems which depend on that object file. The third column contains the most used symbol of that object file.

This table can be used to judge the relevance of the different interfaces. It might also be used to detect functionality implemented in the wrong place. For example: Why does *kernel/fork.o* (which should and does implement the *fork* syscall) has *remove_wait_queue* as most used symbol? Why are wait queues implemented there?

# of uses	**object file**	**most used symbol**
2219 (42)	kernel/printk.o	printk
1475 (42)	mm/slab.o	kfree
1220 (41)	arch/i386/lib/memcpy.o	memcpy
895 (41)	lib/vsprintf.o	sprintf
761 (37)	kernel/timer.o	del_timer
680 (39)	kernel/sched.o	__wake_up
611 (33)	arch/i386/lib/usercopy.o	copy_from_user
546 (10)	net/core/skbuff.o	__kfree_skb
537 (31)	arch/i386/lib/delay.o	__const_udelay
529 (34)	arch/i386/kernel/semaphore.o	__up_wakeup
493 (33)	lib/string.o	memcmp
460 (17)	kernel/softirq.o	raise_softirq_irqoff
394 (27)	arch/i386/kernel/irq.o	free_irq
392 (7)	net/core/dev.o	per_cpu__softnet_data
389 (27)	kernel/resource.o	__request_region
338 (23)	kernel/params.o	param_get_int
325 (32)	kernel/fork.o	remove_wait_queue
307 (21)	mm/memory.o	mem_map
301 (21)	drivers/pci/pci.o	pci_enable_device
265 (21)	drivers/pci/access.o	pci_bus_read_config_byte
262 (22)	drivers/pci/pci-driver.o	pci_register_driver
251 (25)	arch/i386/mm/ioremap.o	iounmap
224 (22)	arch/i386/lib/getuser.o	__get_user_4
193 (5)	drivers/net/net_init.o	unregister_netdev
177 (26)	fs/proc/generic.o	create_proc_entry
170 (21)	mm/page_alloc.o	free_pages

# of uses	object file	most used symbol
167 (7)	fs/buffer.o	__brelse
165 (25)	kernel/workqueue.o	schedule_work
161 (23)	mm/vmalloc.o	vfree
159 (10)	fs/inode.o	iput
150 (12)	fs/read_write.o	no_llseek
147 (14)	arch/i386/kernel/pci-dma.o	dma_free_coherent
146 (20)	drivers/pci/search.o	pci_find_device
140 (25)	lib/rwsem.o	rwsem_wake
138 (16)	fs/seq_file.o	seq_printf
136 (10)	fs/dcache.o	d_instantiate
127 (20)	arch/i386/kernel/traps.o	dump_stack
118 (11)	kernel/time.o	current_kernel_time
118 (6)	mm/filemap.o	unlock_page
114 (17)	lib/dump_stack.o	dump_stack
111 (19)	arch/i386/kernel/time.o	do_gettimeofday
110 (1)	net/core/sock.o	sk_free
106 (14)	fs/open.o	nonseekable_open
101 (3)	net/ethernet/eth.o	eth_type_trans
100 (21)	arch/i386/kernel/setup.o	boot_cpu_data
100 (19)	kernel/signal.o	send_sig
100 (16)	kernel/panic.o	panic
100 (9)	drivers/usb/core/usb.o	usb_deregister
93 (10)	fs/namei.o	generic_readlink
89 (3)	net/sched/sch_generic.o	__netdev_watchdog_up
88 (9)	mm/swap.o	__page_cache_release
88 (4)	drivers/i2c/i2c-core.o	i2c_add_driver
88 (1)	net/core/utils.o	net_ratelimit
85 (8)	drivers/usb/core/urb.o	usb_submit_urb
84 (15)	fs/proc/root.o	proc_net
84 (15)	kernel/sys.o	register_reboot_notifier
84 (7)	drivers/usb/core/message.o	usb_control_msg
84 (2)	fs/fs-writeback.o	__mark_inode_dirty
83 (20)	kernel/exit.o	daemonize
83 (11)	drivers/block/ll_rw_blk.o	blk_cleanup_queue
77 (21)	arch/i386/kernel/process.o	kernel_thread
75 (10)	drivers/char/tty_io.o	tty_register_driver
73 (13)	drivers/base/core.o	device_create_file
69 (10)	kernel/module.o	module_refcount
69 (1)	net/core/rtnetlink.o	__rta_fill
67 (6)	net/socket.o	sock_release
66 (18)	fs/char_dev.o	register_chrdev

# of uses	object file	most used symbol
65 (4)	drivers/input/input.o	input_event
64 (10)	fs/super.o	get_sb_bdev
61 (9)	fs/libfs.o	generic_read_dir
59 (20)	kernel/kmod.o	request_module
59 (9)	fs/file_table.o	fput
59 (8)	fs/partitions/check.o	del_gendisk
58 (10)	fs/filesystems.o	register_filesystem
57 (4)	drivers/scsi/scsi_scan.o	scsi_scan_host
56 (7)	kernel/dma.o	free_dma
56 (4)	drivers/scsi/hosts.o	scsi_add_host
55 (8)	fs/block_dev.o	sb_set_blocksize
54 (18)	lib/ctype.o	_ctype
52 (1)	net/sunrpc/sysctl.o	rpc_debug
51 (13)	drivers/char/random.o	get_random_bytes
51 (9)	drivers/block/genhd.o	put_disk
51 (5)	lib/crc32.o	crc32_le
51 (1)	fs/bad_inode.o	make_bad_inode
50 (6)	drivers/pnp/manager.o	pnp_activate_dev
48 (13)	kernel/sysctl.o	register_sysctl_table
47 (7)	drivers/char/misc.o	misc_deregister
44 (8)	arch/i386/kernel/init_task.o	init_mm
43 (15)	fs/fcntl.o	fasync_helper
43 (4)	mm/page-writeback.o	set_page_dirty
43 (2)	net/core/link_watch.o	linkwatch_fire_event
43 (1)	drivers/ide/ide.o	noautodma
42 (9)	drivers/pcmcia/cs.o	pcmcia_deregister_client
42 (2)	drivers/scsi/scsi.o	scsi_adjust_queue_depth
42 (1)	drivers/input/gameport/gameport.o	gameport_register_port
41 (12)	drivers/base/class_simple.o	class_simple_device_add
39 (14)	lib/kobject.o	kobject_put
39 (5)	crypto/api.o	crypto_register_alg
38 (5)	arch/i386/lib/checksum.o	csum_partial
37 (9)	drivers/pcmcia/cistpl.o	pcmcia_get_first_tuple
37 (8)	mm/mmap.o	do_mmap_pgoff
35 (9)	drivers/pcmcia/ds.o	pcmcia_register_driver
33 (2)	drivers/ide/ide-iops.o	ide_config_drive_speed
33 (1)	net/core/datagram.o	skb_copy_datagram_iovec
32 (13)	arch/i386/lib/strstr.o	strstr
32 (9)	drivers/base/class.o	class_register
32 (7)	arch/i386/mm/init.o	__PAGE_KERNEL
32 (5)	drivers/block/elevator.o	elv_next_request

# of uses	object file	most used symbol
31 (11)	drivers/pnp/driver.o	pnp_device_attach
31 (6)	drivers/char/tty_ioctl.o	tty_wait_until_sent
31 (1)	drivers/ide/ide-lib.o	ide_get_best_pio_mode
30 (5)	fs/bio.o	bio_endio
30 (2)	sound/sound_core.o	register_sound_mixer
29 (2)	fs/stat.o	inode_add_bytes
28 (2)	net/core/filter.o	sk_run_filter
27 (3)	net/ipv4/arp.o	arp_find
27 (1)	net/core/iovec.o	memcpy_fromiovec
26 (6)	mm/mempool.o	mempool_alloc
26 (2)	drivers/media/video/videodev.o	video_register_device
26 (1)	net/irda/qos.o	irda_qos_bits_to_value
25 (14)	drivers/base/driver.o	driver_unregister
25 (3)	arch/i386/kernel/cpu/mtrr/main.o	mtrr_add
25 (1)	net/irda/irda_device.o	irda_device_set_media_busy
24 (8)	drivers/parport/share.o	parport_release
24 (6)	drivers/pnp/isapnp/compat.o	pnp_find_dev
24 (6)	fs/namespace.o	__mntput
24 (4)	mm/truncate.o	invalidate_inode_pages
23 (12)	drivers/base/bus.o	bus_register
23 (11)	kernel/power/process.o	refrigerator
23 (3)	fs/locks.o	posix_lock_file
23 (3)	net/netlink/af_netlink.o	netlink_broadcast
23 (1)	drivers/pnp/card.o	pnp_register_card_driver
22 (1)	drivers/mtd/mtdcore.o	add_mtd_device
21 (4)	drivers/block/scsi_ioctl.o	scsi_command_size
21 (3)	drivers/acpi/bus.o	acpi_root_dir
21 (1)	fs/attr.o	inode_change_ok
20 (9)	kernel/rcupdate.o	call_rcu
20 (6)	arch/i386/kernel/cpu/common.o	per_cpu__cpu_gdt_table
19 (8)	arch/i386/lib/bitops.o	find_next_zero_bit
19 (6)	kernel/pid.o	find_task_by_pid_type
19 (2)	net/core/ethtool.o	ethtool_op_get_sg
19 (1)	drivers/acpi/namespace/nsxfeval.o	acpi_evaluate_object
19 (1)	drivers/scsi/scsi_lib.o	scsi_wait_req
19 (1)	net/irda/irlap.o	irlap_close
18 (3)	kernel/intermodule.o	inter_module_register
18 (2)	drivers/acpi/scan.o	acpi_bus_register_driver
18 (2)	drivers/cpufreq/cpufreq.o	cpufreq_register_driver
18 (1)	drivers/scsi/constants.o	scsi_print_command
18 (1)	net/ipv4/devinet.o	ipv4_devconf

# of uses	object file	most used symbol
17 (7)	kernel/power/pm.o	pm_register
17 (4)	arch/i386/kernel/head.o	swapper_pg_dir
17 (4)	drivers/pnp/isapnp/core.o	isapnp_present
16 (9)	lib/cmdline.o	get_option
16 (4)	security/commoncap.o	cap_vm_enough_memory
16 (1)	drivers/acpi/utils.o	acpi_evaluate_integer
16 (1)	drivers/net/mii.o	mii_ethtool_gset
16 (1)	net/sunrpc/clnt.o	rpc_clnt_sigmask
15 (4)	drivers/pci/quirks.o	isa_dma_bridge_buggy
15 (3)	drivers/i2c/algos/i2c-algo-bit.o	i2c_bit_add_bus
15 (2)	drivers/pci/probe.o	pci_scan_slot
15 (2)	lib/parser.o	match_token
15 (2)	net/atm/atm_misc.o	atm_pcr_goal
15 (2)	net/atm/resources.o	atm_dev_register
15 (1)	drivers/scsi/scsi_error.o	scsi_block_when_processing_errors
15 (1)	net/core/neighbour.o	neigh_destroy
14 (7)	fs/sysfs/file.o	sysfs_create_file
14 (6)	drivers/pci/setup-res.o	pci_assign_resource
14 (5)	drivers/char/vt.o	fg_console
14 (1)	drivers/acpi/namespace/nsxfname.o	acpi_get_handle
14 (1)	net/sunrpc/sched.o	rpc_delay
13 (6)	init/version.o	system_utsname
13 (5)	arch/i386/kernel/acpi/boot.o	acpi_noirq
13 (2)	drivers/acpi/events/evxface.o	acpi_install_notify_handler
13 (1)	drivers/scsi/scsi_sysfs.o	scsi_register_driver
13 (1)	net/bluetooth/hci_core.o	hci_alloc_dev
12 (7)	drivers/base/platform.o	platform_device_unregister
12 (7)	fs/sysfs/symlink.o	sysfs_create_link
12 (6)	init/main.o	loops_per_jiffy
12 (5)	drivers/base/dmapool.o	dma_pool_alloc
12 (4)	drivers/base/sys.o	sysdev_class_register
12 (1)	net/atm/common.o	vcc_hash
12 (1)	net/sunrpc/xdr.o	xdr_encode_opaque
11 (7)	lib/idr.o	idr_pre_get
11 (6)	mm/bootmem.o	__alloc_bootmem
11 (4)	lib/rbtree.o	rb_next
11 (4)	mm/swapfile.o	swap_free
11 (3)	drivers/acpi/resources/rsxface.o	acpi_walk_resources
11 (3)	lib/crc-ccitt.o	crc_ccitt_table
11 (3)	mm/readahead.o	default_backing_dev_info
11 (2)	arch/i386/pci/common.o	raw_pci_ops

# of uses	object file	most used symbol
11 (2)	fs/exec.o	kernel_read
11 (1)	net/802/tr.o	tr_type_trans
11 (1)	net/rxrpc/call.o	rxrpc_put_call
11 (1)	net/rxrpc/connection.o	rxrpc_put_connection
10 (6)	arch/i386/kernel/dmi_scan.o	dmi_check_system
10 (5)	arch/i386/mm/pgtable.o	__set_fixmap
10 (4)	drivers/base/firmware_class.o	request_firmware
10 (3)	arch/i386/kernel/reboot.o	machine_restart
10 (3)	kernel/posix-timers.o	do_posix_clock_monotonic_gettime
10 (1)	drivers/cpufreq/cpufreq_performance.o	cpufreq_gov_performance
10 (1)	drivers/net/wan/hdlc_generic.o	alloc_hdlcdev
10 (1)	drivers/pci/bus.o	pci_bus_add_devices
10 (1)	drivers/usb/core/hub.o	usb_reset_device
10 (1)	fs/mpage.o	mpage_writepages
10 (1)	net/irda/wrapper.o	async_wrap_skb
9 (5)	fs/aio.o	wait_on_sync_kiocb
9 (4)	kernel/profile.o	create_prof_cpu_mask
9 (3)	arch/i386/kernel/i387.o	kernel_fpu_begin
9 (2)	fs/dnotify.o	dnotify_parent
9 (2)	lib/kref.o	kref_put
9 (1)	drivers/cpufreq/freq_table.o	cpufreq_freq_attr_scaling_available_freqs
9 (1)	mm/swap_state.o	swapper_space
9 (1)	net/ax25/ax25_ip.o	ax25_encapsulate
9 (1)	net/core/dev_mcast.o	dev_mc_add
9 (1)	net/rxrpc/transport.o	rxrpc_put_transport
9 (1)	net/sunrpc/auth.o	put_rpccred
9 (1)	net/sunrpc/xprt.o	xprt_destroy
8 (7)	fs/sysfs/bin.o	sysfs_create_bin_file
8 (3)	drivers/cdrom/cdrom.o	cdrom_ioctl
8 (3)	drivers/parport/ieee1284.o	parport_negotiate
8 (3)	lib/radix-tree.o	radix_tree_lookup
8 (3)	mm/rmap.o	anon_vma_prepare
8 (2)	mm/vmscan.o	set_shrinker
8 (1)	drivers/ide/ide-io.o	ide_do_drive_cmd
8 (1)	drivers/net/loopback.o	loopback_dev
8 (1)	net/atm/ioctl.o	deregister_atm_ioctl
8 (1)	net/sunrpc/cache.o	cache_check
7 (3)	kernel/user.o	free_uid
7 (3)	lib/zlib_inflate/inflate.o	zlib_inflate
7 (2)	arch/i386/pci/pcbios.o	pcibios_set_irq_routing
7 (1)	crypto/hmac.o	crypto_hmac_final

# of uses	object file	most used symbol
7 (1)	drivers/net/ppp_generic.o	ppp_channel_index
7 (1)	drivers/pci/remove.o	pci_remove_bus_device
7 (1)	net/core/wireless.o	iw_handler_get_spy
7 (1)	net/sunrpc/svcsock.o	svc_reserve
6 (5)	fs/sysfs/group.o	sysfs_create_group
6 (2)	drivers/serial/8250.o	register_serial
6 (2)	mm/prio_tree.o	vma_prio_tree_next
6 (1)	drivers/acpi/namespace/nsxfobj.o	acpi_get_parent
6 (1)	drivers/acpi/tables/tbxfroot.o	acpi_get_firmware_table
6 (1)	net/lapb/lapb_iface.o	lapb_connect_request
6 (1)	net/sunrpc/svcauth.o	auth_domain_put
6 (1)	sound/oss/wavfront.o	dev
5 (3)	kernel/extable.o	search_exception_tables
5 (3)	kernel/kallsyms.o	__print_symbol
5 (2)	arch/i386/mm/pageattr.o	change_page_attr
5 (2)	arch/i386/pci/irq.o	pcibios_enable_irq
5 (2)	ipc/sem.o	exit_sem
5 (2)	kernel/acct.o	acct_auto_close
5 (2)	kernel/ptrace.o	access_process_vm
5 (1)	arch/i386/pci/i386.o	pcibios_align_resource
5 (1)	drivers/acpi/processor.o	acpi_processor_register_performance
5 (1)	drivers/char/consolemap.o	con_set_default_unimap
4 (3)	drivers/base/cpu.o	cpu_sysdev_class
4 (3)	lib/zlib_deflate/deflate.o	zlib_deflate
4 (2)	drivers/base/power/resume.o	device_power_up
4 (2)	drivers/char/vt_ioctl.o	reset_vc
4 (2)	kernel/exec_domain.o	default_exec_domain
4 (2)	mm/pdflush.o	pdflush_operation
4 (2)	mm/shmem.o	shmem_zero_setup
4 (1)	arch/i386/mach-default/setup.o	intr_init_hook
4 (1)	drivers/acpi/pci_irq.o	acpi_pci_irq_add_prt
4 (1)	fs/file.o	expand_fd_array
4 (1)	fs/readdir.o	vfs_readdir
4 (1)	mm/thrash.o	grab_swap_token
4 (1)	net/appletalk/aarp.o	aarp_send_ddp
4 (1)	net/sunrpc/stats.o	rpc_proc_register
4 (1)	net/sunrpc/svcauth_unix.o	unix_domain_find
4 (1)	net/sunrpc/svc.o	svc_create
3 (3)	arch/i386/kernel/i8259.o	init_8259A
3 (3)	lib/bitmap.o	bitmap_scnprintf
3 (2)	arch/i386/kernel/cpu/intel.o	early_intel_workaround

# of uses	object file	most used symbol
3 (2)	arch/i386/kernel/ldt.o	init_new_context
3 (2)	arch/i386/mm/fault.o	bust_spinlocks
3 (2)	drivers/base/firmware.o	firmware_register
3 (2)	drivers/base/power/suspend.o	device_power_down
3 (2)	fs/dcookies.o	dcookie_register
3 (2)	fs/select.o	poll_freewait
3 (2)	ipc/msg.o	msg_ctlmax
3 (2)	ipc/shm.o	do_shmat
3 (2)	kernel/capability.o	cap_bset
3 (2)	lib/int_sqrt.o	int_sqrt
3 (2)	mm/highmem.o	blk_queue_bounce
3 (1)	arch/i386/kernel/acpi/sleep.o	acpi_reserve_bootmem
3 (1)	arch/i386/kernel/timers/timer_tsc.o	sched_clock
3 (1)	arch/i386/kernel/vsyscall-sysenter.o	__kernel_rt_sigreturn
3 (1)	drivers/acpi/ec.o	ec_read
3 (1)	drivers/net/slhc.o	slhc_compress
3 (1)	drivers/parport/ieee1284_ops.o	parport_ieee1284_ecp_read_data
3 (1)	fs/proc/base.o	proc_pid_flush
3 (1)	fs/sysfs/dir.o	sysfs_create_dir
3 (1)	kernel/itimer.o	do_setitimer
3 (1)	kernel/kthread.o	kthread_create
3 (1)	net/802/fddi.o	fddi_type_trans
3 (1)	net/appletalk/ddp.o	atalk_find_dev_addr
3 (1)	net/ipv4/utils.o	in_aton
2 (2)	drivers/base/map.o	kobj_lookup
2 (2)	lib/bust_spinlocks.o	bust_spinlocks
2 (2)	lib/errno.o	errno
2 (1)	arch/i386/kernel/acpi/wakeup.o	acpi_copy_wakeup_routine
2 (1)	arch/i386/kernel/ioport.o	sys_ioperm
2 (1)	arch/i386/kernel/ptrace.o	do_syscall_trace
2 (1)	arch/i386/mm/extable.o	fixup_exception
2 (1)	arch/i386/mm/mmap.o	arch_pick_mmap_layout
2 (1)	arch/i386/power/cpu.o	restore_processor_state
2 (1)	drivers/acpi/pci_link.o	acpi_irq_penalty_init
2 (1)	drivers/acpi/pci_root.o	acpi_pci_register_driver
2 (1)	drivers/acpi/tables.o	acpi_get_table_header_early
2 (1)	drivers/base/power/shutdown.o	device_detach_shutdown
2 (1)	drivers/char/pty.o	ptm_driver
2 (1)	drivers/pci/setup-bus.o	pci_bus_assign_resources
2 (1)	drivers/pnp/pnpbios/bioscalls.o	pnpbios_calls_init
2 (1)	drivers/pnp/support.o	pnp_is_active

# of uses	object file	most used symbol
2 (1)	fs/devpts/inode.o	devpts_get_tty
2 (1)	fs/eventpoll.o	eventpoll_init_file
2 (1)	fs/pipe.o	do_pipe
2 (1)	fs/proc/kcore.o	kclist_add
2 (1)	fs/proc/proc_tty.o	proc_tty_init
2 (1)	init/do_mounts.o	prepare_namespace
2 (1)	lib/extable.o	search_extable
2 (1)	lib/zlib_inflate/inflate_sync.o	zlib_inflateIncomp
2 (1)	mm/mlock.o	sys_mlock
2 (1)	net/802/hippi.o	hippi_header
2 (1)	net/atm/raw.o	atm_init_aal5
2 (1)	net/core/netpoll.o	netpoll_cleanup
2 (1)	net/wanrouter/wanmain.o	register_wan_device
1 (1)	arch/i386/kernel/cpu/proc.o	cpuinfo_op
1 (1)	arch/i386/kernel/module.o	apply_relocate
1 (1)	arch/i386/kernel/sysenter.o	enable_sep_cpu
1 (1)	arch/i386/oprofile/init.o	oprofile_arch_exit
1 (1)	arch/i386/pci/acpi.o	pci_acpi_scan_root
1 (1)	arch/i386/pci/mmconfig.o	pci_mmcfg_base_addr
1 (1)	drivers/acpi/blacklist.o	acpi_blacklisted
1 (1)	drivers/base/init.o	driver_init
1 (1)	drivers/block/ioctl.o	blkdev_ioctl
1 (1)	drivers/char/rtc.o	rtc_control
1 (1)	drivers/char/sonypi.o	sonypi_camera_command
1 (1)	drivers/serial/8250_pci.o	pci_siig10x_fn
1 (1)	drivers/video/console/vgacon.o	vga_con
1 (1)	fs/ioctl.o	sys_ioctl
1 (1)	fs/nfsctl.o	sys_nfsservctl
1 (1)	fs/xattr.o	sys_fgetxattr
1 (1)	ipc/mqueue.o	sys_mq_getsetattr
1 (1)	kernel/power/main.o	pm_set_ops
1 (1)	kernel/uid16.o	sys_chown16
1 (1)	lib/libcrc32c.o	crc32c_le
1 (1)	mm/fadvise.o	sys_fadvise64
1 (1)	mm/madvise.o	sys_madvise
1 (1)	mm/mincore.o	sys_mincore
1 (1)	mm/mprotect.o	sys_mprotect
1 (1)	mm/mremap.o	sys_mremap
1 (1)	mm/msync.o	sys_msync
1 (1)	net/sysctl_net.o	net_table

Declaration

All the work contained within this thesis,
except where otherwise acknowledged,
was solely the effort of the author.
At no stage was any collaboration
entered into with any other party.

(Jens-Christian Korth)