Device Configuration in 4.4BSD

Chris Torek [*]
Lawrence Berkeley Laboratory

December 17, 1992

Abstract

Autoconfiguration is the process of finding actual hardware devices on a particular machine. This allows a single kernel to work on different systems, and provides flexibility in hardware configurations. This process is inherently machine-dependent and therefore conflicts with the goal of providing machine-independent abstractions. For instance, SCSI devices use a machine-independent protocol but run underneath machine-dependent adapters. 4.4BSD uses "object-oriented" techniques to resolve this conflict. This document describes this approach and gives details on how to build system "meta-descriptions" for the `config` program. The new scheme makes use of dynamic allocation to remove hard limits on the number of devices supported, and provides a starting point for dynamic loading of drivers.

NOTICE

THIS IS A PRELIMINARY DOCUMENT, AND IS NOT TO BE RECIRCULATED. THIS IS A WORK IN PROGRESS. SEND COMMENTS TO CHRIS TOREK <TOREK@EE.LBL.GOV>.

1 Introduction

Autoconfiguration was first added to Berkeley UNIX[1] for the VAX[2] 11/780 in 4.1BSD. Since then, the system has been ported to a number of new architectures. The old autoconfiguration and device support scheme was not well suited to the newer machines, as detailed in [2]. In essence, the old system allowed machine-specific architectural details to "leak out" of the machine-dependent subsystems.

The 4.4BSD kernel enforces separation of such details through a kind of "multiple inheritence" system. Devices are named via machine-independent data structures; these may inherit from both general system abstractions (e.g., "disk volume") and bus-dependent structures (e.g., for DMA resources). All device drivers share a common machine-independent autoconfiguration interface as well. Details on the higher level machine-independent abstractions are given elsewhere ≪or will be, eventually...≫. This document describes only the "base class", including the autoconfiguration interface.

In its most basic form, autoconfiguration consists of the recursive process of finding and attaching all devices on a bus, including other busses. Here a "bus" is anything that supports other

[*]Supported in part by the Defense Advanced Research Projects Agency under contract BG 91-66.
[1]UNIX is a registered trademark of Unix Systems Laboratories, Inc.
[2]VAX, UNIBUS, and MASSBUS are trademarks of Digital Equipment Corporation.

devices. This includes adapters and what are usually referred to as "controllers" or "masters". In many cases, each bus driver can probe the bus all by itself. This is called *direct configuration*, and is preferred because it finds physically-connected hardware regardless of the presence of proper drivers and configurations. Sometimes, however, the driver needs "hints" as to what to try. This sort of *indirect configuration* is more general and is also provided.

In any case, when a device is found the kernel must somehow "hook up" the appropriate driver code. In 4.4BSD, a user-level program called `config` reads and verifies a "machine description" or "configuration" file. It then produces a table giving needed hints and hook-up details. Autoconfiguration itself is carried out by the bus drivers; the same `config` program selects these drivers based on the same machine descriptions. A description can be targeted for a specific machine—this produces smaller, more efficient kernels—or made generic, so that the resulting kernel can run on a variety of different machines. (Of course, all of these machines must run the same basic instruction set, and hardware deficiencies may limit this flexibility.)

Previous systems also used a `config` program [1]. Unlike that one, however, the new `config` contains no machine dependent code.[3] Instead, the new `config` reads files of what amount to constraints controlling configurations. In essence, `config` provides a "little language" in which one writes machine definition programs that recognize machine descriptions. These programs are much smaller than the equivalent machine-dependent C code found in the old `config`, and correspondingly easier to write. Much of this document concerns itself with writing or altering machine definition programs. Anyone porting 4.4BSD to new hardware will need this information. Other documents ≪unwritten as yet≫ describe the use of `config` for existing machine definitions. In general, such use will be familiar to users of the old `config`.

The terms "machine definition" and "machine description" are unfortunately similar, and this document must refer to both in situations that make them even more difficult to distinguish. The word "configuration" is no better. Instead, we shall adopt as a paradigm a mythical machine called a "SuperSnail-1". The machine being defined, then, will be the `snail`; this really means "any machine definition". We shall picture a particular SuperSnail-1 with a SnailBus adapter and some attached SCSI devices. The configuration file for this particular SuperSnail-1 will be the `SS1`; this plays the role of "any machine description". We will occasionally resort to real machines as examples as well.

2 Common Concepts

Each device must have a name. Externally, this name is simply an alphanumeric string that ends in a *unit number*: "sd0", "sd1", and so forth. These unit numbers identify particular *instances* of a *base device name* (here "sd"); the base name in turn maps directly to a device driver. Within the kernel, this name string is supplanted by other names. Device data structures are allocated dynamically during configuration, giving a unique address for each instance. This is an ideal handle, and hence is the primary internal name. Until this allocation occurs, however, the unit number and driver are used. Thus, every device needs a unit number, even if there can only be one instance (as is commonly true of the "main" backplane bus).

Device unit numbers are completely independent of hardware features such as drive numbers. They are often numerically identical, but this is simply a consequence of sequential numbering. There is no conceptual tie between a device unit number and any address it may have on a bus.

Except as noted below, every device must have a parent, and its position—or one of several

[3]Actually, there is a single special case for `i386` kernels, but only to preserve a historical oddity.

possible positions—is denoted by the pairing "*child* at *parent*". These pairings form the links in a directed graph. At runtime, this graph is turned into a tree[4] by nominating one device as the *root device*—this this should not be confused with the file system root—and doing a depth-first traversal through the graph starting at this root.

For any device to be named as the root at runtime, it must be configured "at root". In effect, the name "root" acts as a special "placeholder" parent, i.e., as an entry point into the graph. By induction, any nonempty configuration must have at least one device that can be found "at root"; otherwise the entire configuration would be unreachable. The `config` program rejects configurations containing unreachable devices. Of course, reachability in the graph does not imply that the device will be found, only that there is a path from some root. It is legal—and sometimes even reasonable—to construct a graph with two completely independent subtrees. The runtime tree will contain only "found" devices.

These "child at parent" pairings are rarely sufficient to locate any particular device precisely. For instance, on one VAX MASSBUS, one might have several different "hp" disks. In the scheme above, there is no way to tell these apart: they are all "hp$n$ at mba$m$". The `config` program therefore allows any parent/child pair to be augmented with one or more *locators*. Each locator value is simply an integer, or a name that ultimately resolves to an integer. Locators typically represent some sort of device address on a bus or controller. This can be a memory address, an I/O port, a drive number, or any other value; `config` does not care.

Locators can sometimes be *wildcarded*, given appropriate hardware and driver support.[5] Alternatively, the same child device may be listed more than once with different parents and/or locators if that particular child might appear in one of several places. This allows arbitrarily complex graphs to be constructed.

During autoconfiguration, the graph itself is represented in compact form as a table of "configuration data". Running, e.g., `config SS1` generates this table from the machine description file `SS1`. The table format is given in Appendix E; for now it suffices to note that it is an array of `cfdata` structures, which in turn point to "configuration drivers". Although the table itself is quite simple—this reduces the size of the kernel autoconfiguration code—it is rather difficult to maintain manually. It uses many numeric cross-references, and in any case it offers no protection against incorrect configurations. This is one of several reasons for using machine description files.

To make direct configuration easier and more efficient, 4.4BSD supports a kind of "cloning" of configuration data. These are denoted by replacing the unit number with an asterisk. Instead of listing dozens of "sd" disks, for instance, the `SS1` file can claim the presence of a single `sd*` (possibly in addition to one or more "regular" sd unit). Making devices clone-capable requires some care: there are a number of ways to make cloning impossible. These are listed below as they occur. Generally, a clone device should also use wildcards for all its locators.

3 The `config` Program

Of course, real machines are not constructed arbitrarily: a leg bone must not connect to a shoulder bone, but only to hip bones above and foot bones below. The restrictions on any machine's device tree are clearly machine-dependent. These constraints are embedded in each machine definitions

[4]More complex structures could be provided, but no current port needs them. This may have to change for systems with "channel"-style disks.

[5]Typically, wildcarding will be allowed only on busses and controllers that support direct configuration (section 5.2). Wildcarding then allows devices to be moved within and between machines, e.g., to give flexibility during hardware failures. The details are, however, entirely up to the device drivers involved.

program. The `config` program reads these constraints, verifies the machine description, and then builds the graph table and a "compilation kit" for the kernel. To do this, `config` needs some additional information.

3.1 Attributes

The verification process is largely controlled by *attributes*. Each base device is associated with zero or more of these attributes. Attributes can carry lists of child devices as well; in this case, they describe a kind of "software interface" and are therefore called *interface attributes*. "Plain" (childless) attributes exist so that some aspect of a device, such as its "Ethernet"-ness, can be denoted. These "plain" attributes then serve to pull in files of common code—in this case, `netinet/if_ether.c`.

Like a plain attribute, an interface attribute can select a kernel file, but this is not its main purpose. By referring to an interface attribute, a device declares its ability to support other devices that use that interface. In addition, an interface attribute can—and usually does—also name one or more locators.[6] A child device may attach to any particular parent through such an interface attribute. The child must then name the associated locators, describing its "address" on the parent. This leads naturally to declarations such as: `hp3 at mba1 drive 0`, a VAX MASSBUS disk (hp3) found at a MASSBUS adapter (mba1), located at drive 0; `bwtwo0 at sbus0 slot 3 offset 0`, a SPARCstation[7] bitmapped display located in Sbus slot 3 at address 0; and `fdc0 at isa0 port IO_FD1 irq 6 drq 2`, a floppy disk controller on an IBM PC,[8] with complex locators. Declarations like this make up the bulk of most machine description files. Note that they may be read either as "I found c_i at p_j", or "look for c_i at p_j". Both interpretations are reasonable, and both have their places; see section 5 for more about this.

3.2 Configuration Files

The `config` program reads three different "kinds" of files. The primary configuration file (`SS1`) has a fairly simple format described in section 3.2.3. This file must begin with a `machine` line naming the machine type (although this may come from an `include` file). The `machine` declaration actually directs `config` to a number of other files, which ultimately determine what configurations are allowed. Our `SS1` file therefore begins with `machine snail`.[9] `Config` will then read the following files, in order:

the first line of `SS1`	(`machine snail`);
`../../conf/files`	(machine-independent definitions);
`./files.snail`	(machine-dependent definitions);
the rest of `SS1`	(the machine description for `SS1`).

(All these paths are relative to the current directory. Typically, this would be `/sys/snail/conf`.)

3.2.1 Devices

The two "files" files define all legal devices and pseudo-devices. They also define all attributes and interfaces, establishing the rules that determine allowable configurations, and list the source files

[6] Note that the *attribute* names the locators. The locators themselves are merely numbers, or strings that resolve to numbers. The names for these locators are attached only to the attribute. We will revisit this concept later.

[7] SPARC is a trademark of Sun Microsystems.

[8] IBM is a registered trademark of International Business Machines, Inc.

[9] In practice, most configurations will begin with an `include` that will name the machine and declare any "standard" devices, such as a clock chip or a built-in console tty.

that make up the kernel. A formal syntax for these files appears in Appendix A; we will work by example in this section. Note that all of these refer to device base names (since no instances exist yet).

Each file consists of a list of statements, typically one per line. Comments may be inserted anywhere using the "`#`" character, and any line that begins with white space continues the previous line. Valid statements are as follows.

`define` *name* $interface_{opt}$
: Attribute definitions. The named attribute springs into existence, and device definitions (below) can then refer to it. If the attribute defines an interface, devices that refer to that attribute are assumed to share that interface. These then serve to constrain device graphs.

`maxusers` *min default max*
: This sets minimum, default, and maximum values for `maxusers` in machine description files; see 3.2.3.

`device` *name* `at` *iflist* $vectors_{opt}$ $interface_{opt}$ $attributes_{opt}$
: The named device springs into existence. The interface list *iflist*, which must not be empty but can contain or consist of the special name `root`, determine which interfaces will support that device. The interface, if any, and any attributes listed are handled as described below. If any *vectors* are listed, `config` will generate appropriate references to interrupt vector stubs (see section 6); these vectors will, however, disallow cloning.

`pseudo-device` *name* $attributes_{opt}$
: The named pseudo-device springs into existence. Interface attributes are legal here, but largely pointless, since no device can connect to a pseudo-device (pseudo-devices have no reason to "live in" the device tree).

`file` *pathname options flags* $rule_{opt}$
: The named *pathname* is added to the list of files used to build the kernel; see section 3.2.2.

`major` *major-list*
: The named devices are associated with major device numbers, allowing them to be used for file system roots, swapping, and crash dumps. Since major device numbers are machine-dependent, they should not appear in the machine-independent `files` file.

Interface definitions consist of a (possibly empty) list of locators surrounded by braces (`{}`). Devices—usually bus adapters and controllers—that support other devices then refer back to the corresponding interface attribute, declaring, in effect, that they export that interface. Some examples should make these clear.

The following defines a simplified VAX UNIBUS (assuming a suitable `sbi` interface exists):

```
define uba { csr }     # create unibus interface
device uba at sbi: uba # uba's export it
device lp at uba       # lp's live on it
device vp at uba       # etc.
```

This means, in essence, that the devices "lp" and "vp" can be found via this "uba" attribute, but in order to do so, a single locator called a "csr" must be specified. This locator is mandatory; a machine description file that, e.g., declares "lp2 at uba0" without also giving it a csr will be rejected.

To allow a locator to be wildcarded, a "devices" file must include default values. Typical default values are 0 (for addresses) and -1 (for index or drive numbers).

```
define sbus { slot = -1, offset = -1 }
device sbus: sbus device le at sbus: ifnet, ether
```

A machine description file can then say something like

```
le0 at sbus0 slot ? offset ?
```

(the question-marks represent "unknown" or "don't care"). `Config` will use the default values for this device's locators. Note that in this case, the "le" device also refers to two plain attributes, because it is both a network interface ("ifnet") and an Ethernet device ("ether").

To allow a machine description to omit a locator entirely—wildcards require the locator name to be listed—enclose the locator in square brackets:

```
define isa { port, irq, [drq = -1], [iomem = 0] }
```

This can be used when some locators do not make sense for some devices, but the software interface requires them anyway. This sacrifices some error checking. Other approaches that avoid this are possible as well:

```
define isa_plain { port, irq }
define isa_drq { port, irq, drq }
define isa_iomem { port, irq, iomem, iosiz }
device isa: isa_plain, isa_drq, isa_iomem
device wdc at isa_plain
device fdc at isa_drq
device we at isa_iomem: ifnet, ether
```

In this case the number of locators associated with the child device will vary: here a "wdc" would have two, an "fdc" would have three, and a "we" would have four.

The approaches can be mixed, if for some reason it is important to have the same number of locators on all children of some parent:

```
define isa_plain { port, irq, ["no drq" = -1], ["no iomem" = 0] }
define isa_drq { port, irq, drq, ["no iomem" = 0] }
define isa_iomem { port, irq, ["no drq" = -1], iomem }
device isa: isa_plain, isa_drq, isa_iomem
device wdc at isa_plain
device fdc at isa_drq
device we at isa_iomem: ifnet, ether
```

Here `config` would reject "drq" and "iomem" values for a "wdc", demand an "iomem" for a "we", and always require a "port" and "irq". This follows from the simple derivation rule given above, but perhaps we should repeat it here:

> *A child device may attach to a particular parent if and only if the parent provides an interface attribute that lists the child.*

In other words, given a parent/child combination (including locators) of the form "c_i `at` p_j l_1 l_2 … l_n", `config` searches the attribute list for p. An attribute a that has a child c is taken. This in turn gives the set of locators required. The actual locators l_k are rearranged so that their names match up with a's required set, and any wildcards and allowable omissions are handled. Any missing or extraneous locators cause a complaint. The locator names are then dropped, leaving only a complete set of values describing c_i's location on p_j.[10] The names `"no drq"` and `"no iomem"` are not "magic"; the blanks simply make them difficult to type, so that they are unlikely to appear accidentally in a machine description file. Their purpose is to act as placeholders for the default -1 and 0 values.

Many bus adapters offer a simple, uniform interface that will be used only once, as a reference for the device of the same name. To avoid having to write separate `define` and `device` lines for these, `config` allows an interface attribute to be defined within the device declaration itself. A line of the form:

`device` *d* `at` *a* `{` *locators*$_{opt}$ `}`

is effectively the same as the two lines:

`define` *d* `{` *locators*$_{opt}$ `}`
`device` *d* `at` *a*`:` *d*

Devices and attributes should generally share names only through this abbreviation mechanism, for reasons explained in section 4.

Some devices can be found on one of several parents. For instance, a VAX `uba` UNIBUS adapter can appear on either a BI bus (as on the 8200) or a "nexus" interface (as on the 8600's SBIA, or the backplane of a 780 or 750). Hence, a more realistic `uba` definition would look like this:

```
device uba at bi, nexus { csr }
```

This defines the "uba" interface attribute, which requires a "csr", and then defines the device "uba" as attaching to anything that provides a "bi" or "nexus" interface. Note that there is no piece of hardware called a "nexus"; the names listed here are interface attributes, not devices.

In any particular "files" file, all of the devices listed in any interface must be defined, though the order of references and device definitions is unimportant. Devices that stand entirely by themselves and support no children need only a single line of the form `device` *d* `at` *a*.

Our `files.snail` now looks like this:

```
# devices.snail
maxusers 2 8 64

# The internal memory/cpu bus has no true name, so we just call it
# the ``main bus''.  Devices on it are directly addressed.
device mainbus at root { addr = 0 }
device clock at mainbus         # time of day & realtime interrupts
device cpu at mainbus           # the SuperSnail chip

# SnailBus adapter (sba).
# Each SnailBus device is at a `slot' index, which may be wildcarded.
```

[10]This is still not the whole story; see section 4 for the real scoop.

```
device sba at mainbus { slot = -1 }
device bd at sba: tty           # bitmapped display
device en at sba: ifnet, ether  # Ethernet interface
device sc at sba: scsi          # SCSI bus adapter
```

Here the `tty`, `ifnet`, `ether`, and `scsi` attributes have been defined in `../../conf/files`. Note that no SCSI devices other than the adapter itself appear here; section 4 explains how this works. (Also, the SuperSnail CPU is just another device. This will be important for future multiprocessor support.)

3.2.2 Kernel Source Files

The various "files" files also list the source files used to build the kernel. A file name in any one such file may override that in any other. For instance, `files.snail` may replace `kern/subr_foo.c` with `snail/snail/subr_foo.c`; if the configuration for machine SS1 contains a directive such as `include files.SLOW`, `files.SLOW` may in turn replace that with `SLOW/subr_foo.c`. More serious conflicts, such as listing both `kern/subr_foo.c` and `vm/subr_foo.c` in the same "files" file, are detected and rejected.

Each kernel file is either standard (always compiled) or optional. In order to be optional, the file line must list one or more names. If any of these names has been selected in the configuration, the file will be included; otherwise it will be omitted. Selection occurs in three ways:

- In the system description file, an `options` *o* selects anything that is optional on *o*.
- Configuring a device whose base name is *d* (e.g., "sd" for "sd0") selects anything that is optional on *d*.
- Configuring a device whose base is marked with attribute *a* selects anything that is optional on *a*, even if *a* is an interface attribute.

Thus, any system configuration that includes a device with the `disk` attribute—say, an `sd` SCSI disk—will pull in any optional `disk` code, as well as the optional `sd` device driver.

`Config` emits a kernel `Makefile` with a separate compilation rule for each source file. For most files, this rule is simply `${NORMAL_C}`. (The prototype `Makefile` must define `${NORMAL_C}`; see Appendix D.) The letter `C` here is really just the last character of the file name; a file named, e.g., `reset.s` would use the rule `${NORMAL_S}` by default. To use a special rule, end the `file` line with `compile-with` *command*.[11] In this case the file name suffix is not appended.

Kernel files may also be labelled with the following flags:

`device-driver`
: This has much the same effect as `compile-with "${DRIVER_C}"`, assuming the file name ends with the letter "c". On some machines, device drivers use special compiler options; this is a shorthand way to express that.

  A `compile-with` will override a `device-driver` declaration.

`config-dependent`
: This modifies the default compilation rule, appending `_C` to it. C source files marked this way will thus be compiled with `${NORMAL_C_C}` or `${DRIVER_C_C}`.

  Again, a `compile-with` will override this declaration.

[11]The command must typically be quoted.

`needs-count`

Some drivers insist on knowing at compile time exactly how many of some particular device or set of devices are configured. An optional file may therefore be marked as “needing a count”, and `config` will count up the number of devices or pseudo-devices of each name on which the file is optional. Thus, a file defined by, e.g., `file snail/sba/foo.c   foo bar needs-count` will cause `config` to generate a header file named `foo.h` containing two `#define`s for `NFOO` and `NBAR`, even if no “foo” or “bar” devices are configured. Any device whose “head count” is taken this way is not clonable, since `config` cannot know in advance how many clonings would occur.

`needs-flag`

If for some reason a header is required, but the driver does not insist on an exact count, an optional file can be marked as “needing a flag”. In this case `config` will generate the header just as for `needs-count`, but the counts will simply be nonzero if the device is configured and 0 if not. Since the count is not exact, `needs-flag` does not prevent autoconfiguration cloning. Also, unlike `needs-count` names, `needs-flag` can apply to non-device file selection names (attributes and options).

A short example will probably make this clearer.

```
file kern/vfs_cache.c
file miscfs/fifofs/fifo_vnops.c   fifo
file net/if_ethersubr.c           ether needs-flag
file ufs/ufs/ufs_ihash.c          ffs lfs mfs
```

Here `vfs_cache.c` will always be compiled; `fifo_vnops.c` will be compiled if `SS1` includes `options FIFO`; `if_ethersubr.c` will be compiled if `SS1` includes any devices with the `ether` attribute, and in any case the file `ether.h` will be created to `#define NETHER` to 0 or 1; and `ufs_ihash.c` will be compiled if any of the three options `FFS`, `LFS`, or `MFS` are given.

3.2.3 Machine Description Files

As mentioned above, the first line of any machine description file—aside from blanks, comments, and `include` directives—must be a machine type-name, such as `snail`. This directs `config` to the appropriate set of “files” files. The remainder of the configuration file specifies global (system-wide) options, `Makefile` options, kernel configuration parameters, and devices. A full formal syntax for machine descriptions appears in Appendix A; we give an overview here.

Most global options are set with `options` lines. In addition to these, however, there are a few special global options. They are:

`makeoptions` *name* `=` *value*

The name and value are copied to the `Makefile`. As elsewhere, if the value contains special characters, it must be quoted.

`maxusers` *n*

Sets the “maximum expected simultaneous users” parameter. This is used to size a number of kernel data structures.

Kernel configuration parameters are comprised of the file system root device; a list of swap devices; and a crash dump area, usually the same as the first swap device. These parameters are tied to a particular kernel image, and a machine description file may list more than one kernel image, each with different parameters. The global options and configured devices will be the same for all these images. Kernel configurations are specified with a `config` line:

> `config` *sysname system-parameters*

The *sysname* is the name of the kernel image, e.g., `vmunix`. The *system-parameters* determine root, swap, and crash dump devices. These simply take the form

> `root` `on`$_{opt}$ *device-parameter*
> `swap` `on`$_{opt}$ *device-parameter* [`and` *device-parameter* ...]
> `dumps` `on`$_{opt}$ *device-parameter*

Each *device-parameter* should name a configured device, optionally with a disk partition letter suffix. If the suffix is omitted, an appropriate default ("a" for root, "b" otherwise) is used. The `swap` and `dumps` parameters may be omitted entirely; in this case `config` will use partition "b" of the file system root device.

As a special case, a "generic" system can be compiled by omitting the root and dumps clauses and giving `swap generic` as the only system parameter. In this case, root, swap, and dump devices will be queried for during boot.

Devices in a machine description file are simply declared using the "child at parent" syntax. Declared devices need not be physically present when `config` is run, nor even when the kernel autoconfigures; however, devices that are physically present but not declared will not be accessible. Device declarations may optionally include the keyword `flags` followed by an integer; these flags are stored in the final table along with the rest of the configuration data. Their interpretation, if any, is up to the driver.

Pseudo-devices are declared with the `pseudo-device` keyword, followed by the name of the pseudo-device and an optional number. As with regular devices, the name must first be defined in a "devices" file. The number, if present, is passed on to the pseudo-device driver; otherwise the default value 1 is used. This normally controls the number of instances of that pseudo-device, but the actual use of that number, if any, is up to the driver. Like real devices, pseudo-devices can be counted via a `needs-count` keyword in a "files" file.

4 The Real Story on Tree Construction

Like locators, parents can be wildcarded. Unlike locators, there are no restraints on parent wildcarding. In the configuration declaration "c_i `at` p_j", the parent device's unit number j may be replaced with a question mark "`?`". In this case the child device c_i can be found at any parent p_j for any configured p_j, including clones. Clonable devices are valid individual parents as well; "c_i `at` $p*$" means just what it suggests.

In fact, however, the declaration "c_i `at` p_j" typically does not refer to a parent *device*, but rather to a parent *attribute*. In most cases these are identical, since most bus adapters use an attribute of the same name via the "shorthand" interface definition. Looking for the attribute directly, rather than looking for the parent device and then finding the corresponding attribute, is slightly more efficient; but the real goal is another kind of device independence.

Recall that these attributes are called "interface attributes". This appellation was not chosen randomly. If a software interface i is properly coded, a child device that uses interface i can be run by any parent that supports interface i. For instance, 4.4BSD includes a machine-independent SCSI subsystem that works through such an interface. The machine-independent `devices` file defines the `scsi` interface attribute to include all the machine-independent SCSI devices. By referring to this `scsi` attribute, the various machine-dependent SCSI host bus adapters listed in any machine-dependent `devices.snail` are automatically declared as supporting the machine-independent SCSI devices. When new SCSI drivers are written, the same host bus adapter driver automatically supports them, and the same `config` file automatically describes this.

At the same time, some machines have available several different—sometimes substantially different—adapters for identical busses. The IBM PC, with its multitude of SCSI adapters, provides a classic example. On these systems it is convenient to be able to write

```
tg0 at scsi? target 0
```

to denote a SCSI target on any device that carries the `scsi` attribute. On the other hand, it may be desirable to tie tg0 to a particular SCSI adapter, so that, e.g., the root or swap partition does not accidentally move if the machine restarts after a hardware failure takes the primary adapter off-line. The `config` program allows for this by searching for a "real device" if the attribute lookup fails. (This is the real reason that devices and attributes should avoid sharing names, except in the usual simplified case.)

Thus, the real rule for tree-building is as follows. Given a child declaration c_i of the form:

c_i `at` $name_j\ l_1\ l_2\ \ldots\ l_n\ flags_{opt}$

where i and j are each valid unit numbers or the clone marker "`*`" or (for j) the wildcard marker "?", and each $l_k, 1 \leq k \leq n$ is a name/value pair, `config` first searches for an attribute a with the given *name*. Only if the name does not correspond to an attribute will `config` search for a base device with that name, then look for a suitable attribute a on that device. In this case, suitability is determined by intersecting the `device` line that defined the parent *name* with that for the child c: c must have been defined `at` some attribute a, and *name* must have referenced the same a. If c does not attach to a, or if there is no suitable a, `config` rejects the declaration. If several different attributes are suitable, `config` chooses one in an unpredictable manner.[12]

(Actually, this is rarely a concern since most devices are allowed `at` only one attribute. For instance, assume that the `isa_plain`, `isa_iomem`, etc., sequence shown at the end of section 3.2.1 is in effect. Then, given a declaration of the form:

```
we0 at isa0 irq 2 iomem 0xc000 port 0x320
```

`config` will not find an "isa" attribute. Instead, it will look up the "isa" device, which refers to three attributes: `isa_plain`, `isa_drq`, and `isa_iomem`. But a `we` is only allowed `at isa_iomem`, so this is the only attribute that can be chosen for a.)

Once the attribute a is verified, `config` collects its locators and applies them against the given locators l_k. These are rearranged if necessary, so that the order of the resulting values corresponds to the "devices" file definition for a. Wildcards are replaced and omitted locators filled in, and if all goes well, this results in a vector of m locator values $v_1, \ldots, v_m$.[13] `Config` then creates a new *alias* for an instance of device c_i with these values, along with the given *flags*. Multiple aliases for c_i, with differing parents and/or locators, are allowed. When all device declarations have been processed, `config` performs a "cross check" to make sure that, for each alias of c_i, the attribute or device $name_j$ corresponds to at least one declared alias. If *name* is an attribute, `config` allows c_i to attach to anything with both the named attribute and unit j; if it is a device, `config` allows c_i to attach only to aliases of device $name_j$.

[12] "We provide rope." —*Keith Bostic*

[13] The vector length m may differ from n due to omitted names in l.

Before emitting the configuration table, `config` attempts to "collapse" aliases. Whenever several aliases for c_i list the same locators and flags, `config` can remove all but one by taking the union of the set of all parents. Wildcarding and collapses can produce long parent vectors. Without further processing these could take significant space, so all parent vectors and locators are in turn packed into arrays. This saves space in the eventual kernel binary.

After collapsing aliases, `config` also checks to make sure that only one clonable alias remains for any device d. This is required by the table format: clonable devices store the "next clone unit" number in the configuration data for the clone. `Config` arranges for this number to start just above the highest explicit unit for d found in the configuration. Thus, if both `sd0` and `sd*` are listed, the configuration data for `sd*` is initialized as unit 1. If no sd0 is listed, `sd*` will be unit 0; if sd0 and sd2 are listed, `sd*` will be unit 3, leaving a "hole". Other arrangements are possible and the kernel may someday be changed to allow multiple clones, but for now only one clone may remain after alias collapsing.

There must be at least one device declaration of the form

> d_i `at root` *flags*$_{opt}$

The base device d must have been marked as an allowable root in a "devices" file. Here i should either be 0 or "`*`", since having the tree rooted at something other than unit 0 is likely to cause confusion. (It can work—nothing in the system itself demands unit 0—but `config` will warn if the final unit number is nonzero.)

5 The Runtime Environment

Each device driver must present to the system a standard autoconfiguration interface. In return, the driver obtains dynamic allocation for its per-device-instance variables. A minimal autoconfiguration interface includes the device base name and a set of "match" and "attach" functions, which are described in section 5.3. Typically there is one such driver in any given source file, although VAX UNIBUS devices often include two per file. If one file includes several drivers, each driver must have its own autoconfiguration interface.

A configuration file such as our `SS1` acts as a list of requests: "look for c_i at p_j". As the system finds devices at runtime, the machine-dependent portion of the autoconfiguration system must make the corresponding declaration "I found c_i at p_j". The machine-independent configuration code ties these together to form the runtime device tree. There are two ways to go about this, described below in section 5.2, but before we get there we need a bit more background.

5.1 Device Classes and Data Structures

4.4BSD groups devices in another way as well. At runtime, found devices are kept on one or more linked lists according to the device "kind" or *class*. These classes are entirely machine-independent, and cover only system-wide abstractions.[14] For instance, disk volumes and network interfaces are treated quite differently within the system, and thus form two of these classes. A catchall class called "dull" covers any device that does not fall into a neat category. Each specific device attach routine should call an appropriate generic "attach" or "establish" routine to get itself on any specific list. The primary list, `alldevs`, lists every device. This list is set up automatically, so "dull" devices need do nothing special.

[14] System monitoring programs such as `systat` and `vmstat` then get their information in machine-independent format via these lists.

```
device mainbus0 at root
disk dk0 at mainbus0 drive 0
disk dk* at mainbus0 drive -1
```

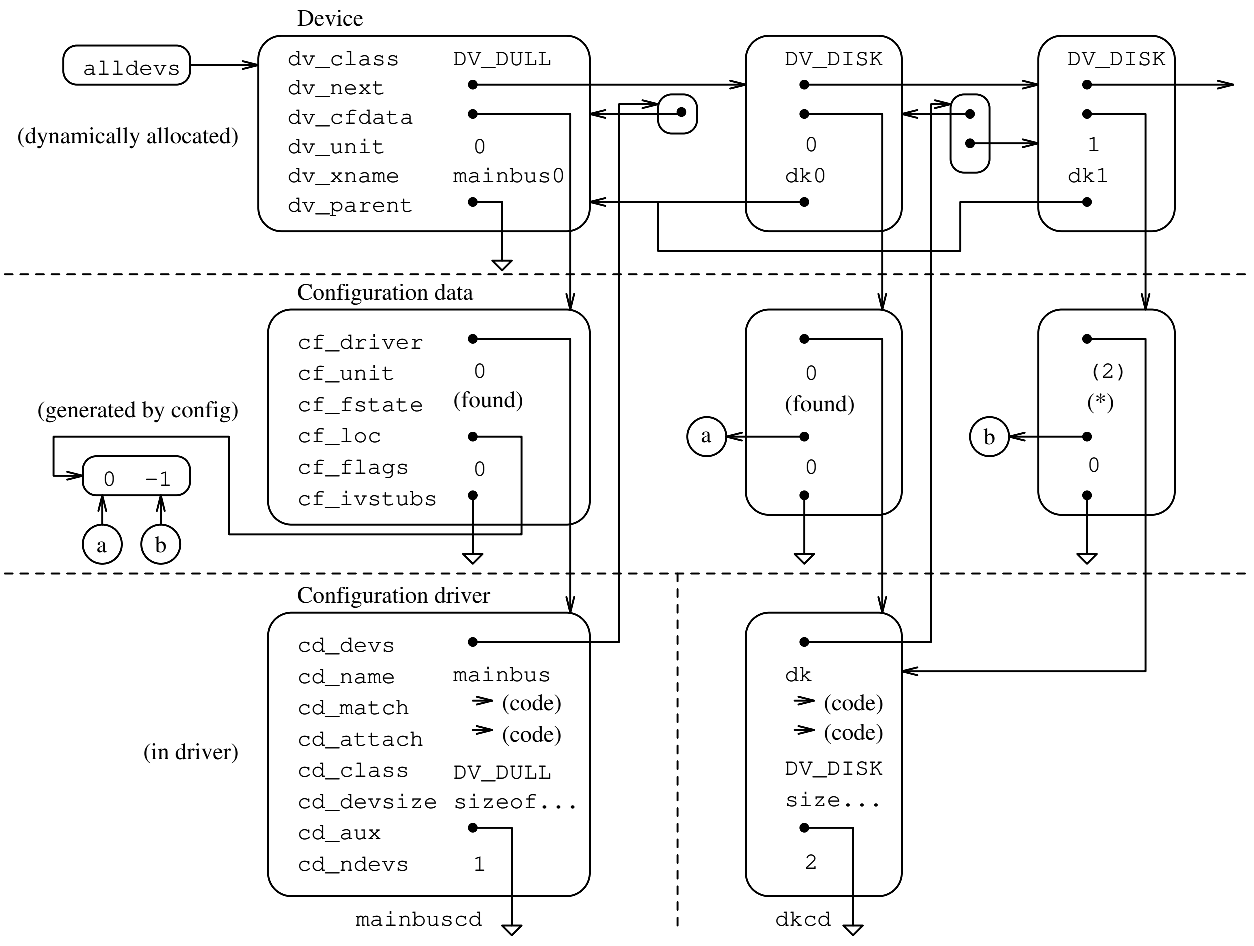

Figure 1: Data Structures

As noted in section 2, config generates a table of configuration data or cfdata structures. These refer to their configuration drivers through pointers to cfdriver structures. The cfdriver carries the name, match, and attach functions, plus the device's class and several more fields described below. For every actual hardware device found, the kernel allocates space that contains a struct device and fills in the device part. For each cfdriver, the kernel also maintains a vector of pointers to these device structures indexed by the device unit number. Figure 1 shows the result of configuring two "dk" disks at the "mainbus". (Several fields have been omitted from this figure for the sake of clarity.) The dotted lines separate the various structures according to which subsystem is responsible for them.

Each cfdriver name is formed by adding the letters "cd" to the device's base name. In Figure 1, for instance, the two cfdrivers are named mainbuscd and dkcd, for the mainbus and dk drivers respectively. The config program uses these names to initialize each cfdata's cf_driver field. Except for the cd_devs and cd_ndevs fields, all of the components of the cfdriver are effectively read-only: the rest of the system may examine them, but not alter them.

Each cfdata carries all data needed to locate one particular device (or in the case of a clone entry, as many as possible). This includes the device unit number and a pointer to any device-specific locators. It also includes a pointer to the interrupt vector stubs, if any; these are described in section 6. Here mainbus0 has been configured with no locators, while dk0 and dk* were each configured with a single locator. These locators have been packed—in this case, with no effect—and appear in a flat array; each cf_loc then points to the first of however many locators there are for each device (even if this number is zero). Note that there can be several cfdata structures with identical cf_driver and cf_unit fields. This occurs whenever the alias collapsing process described in section 4 is unable to remove "extras" due to locator or flags conflicts. The config_attach function (described below) invalidates these other aliases once the first one is attached.

As each device is found, the kernel allocates cd_devsize bytes of space. This must be at least as large as a struct device, but is often larger. The initial part of this space is invariably used as a device structure. The kernel fills in the various fields, assigning a unit number if necessary, and places a pointer to the new device structure in a managed array. Figure 1 shows two of these managed arrays as unnamed boxes just to the right of the device boxes for mainbus0 and dk0 respectively. The kernel also sets cd_devs to point to this array, updating cd_ndevs as necessary to give the array's maximum size. This allows each driver to look up each device by unit number: disk dk1 can be found via dkcd.cd_devs[1]. While this example does not show it, it is normal for some of the device pointers to be NULL. The cd_ndevs field must therefore be considered a limit: values in $[0, 1, \ldots, \texttt{cd\_ndevs})$ are valid subscripts to the cd_devs array; but for any valid i, cd_devs[i] may be NULL.

The base device structure is not sufficient for most devices, and is intended to be "extended". This is why each configuration driver must specify the device data size. A disk device, for instance, must consist at least of a dkdevice structure. A dkdevice begins with, or in C++ parlance, is *derived from*, a generic device, but also contains data structures describing the disk label, transfer rate, and so on. Even this is not enough for many real disk drivers. The SCSI disk driver uses an sd_softc per disk, containing the dkdevice, a struct unit to describe a SCSI unit, and data pertaining to SCSI user-mode format operations.

Auxiliary data may also be required during autoconfiguration. The shape and content of such data are irrelevant to the machine-independent part of the autoconfiguration code, and may vary per device or bus. Each bus adapter will provide some particular configuration interface; anything that can run off that adapter will have to configure under that interface. Generic (void *) pointers to auxiliary data structures are provided in each cfdriver; a separate generic pointer is passed through the configuration functions. These auxiliary data are part of the software interface that must "mesh" to allow a child device to be found on any given parent. This condition is necessary but not sufficient for operation; any other aspects of the parent/child software interface must also mesh. It is up to the authors of the drivers (and devices.snail files) to get this right.

5.2 Autoconfiguration

Early in the bootstrap process, the machine-independent portion of the kernel calls cpu_startup. This function must eventually kick off the autoconfiguration process, usually through a subsidiary called configure. This in turn must choose a root device and start the recursive process that forms the tree.

As noted in section 1, autoconfiguration can be run "forwards" (directly) or "sideways" (indirectly). Direct configuration is better, because it detects hardware that is physically present even if it is not configured into the system, but it can only be used on "self-describing" busses. The

SnailBus, for instance, has a fixed set of "slots" into which I/O cards can be inserted. Each card has an identifying type. By examining the fixed slot addresses, the SnailBus driver can find every device physically present on any particular SnailBus adapter. The SnailBus driver can then inform the system, which will locate an appropriate `cfdata` structure as described below. If there is no such `cfdata`, the system can complain that the device just found needs to be configured.

For indirect configuration, the system gives the bus code an opportunity to test each configured device to see whether it is physically present. This can be done with common code, per driver, or in some hybrid fashion. On the VAX UNIBUS, for instance, each UNIBUS driver must have its own method for deciding whether an I/O card is present, but the UNIBUS driver itself also has some say. All of the details are up to the individual drivers.

5.3 Configuration Functions

With all this in mind, we can now talk about the autoconfiguration process in terms of kernel functions. At this point, not all device instance data are allocated, so the kernel must name individual instances in some other way. Elsewhere, this is done by base device and unit number, but here not even this method suffices—for instance, with direct configuration, a driver for the base device need not even exist.[15] Thus, these functions generally identify a "found device" by the tuple $\langle p, c, a \rangle$. Here p is the parent instance—since child-finding is driven by the parent driver, and happens only after the parent has been established, this is straightforward—and c is a `cfdata` structure from the configuration table. The third element, a, is an "auxiliary" pointer of type `void *`. This auxiliary pointer simply "passed through", allowing parents and children to communicate private configuration data other than through global variables. (See Appendix C for an example of auxiliary data usage.)

As each device is found, it is "attached" via the `cd_attach` pointer in the configuration driver. Attach functions are given three parameters: a pointer p to the parent device data, a pointer s (for "self") to the newly allocated device data, and the auxiliary pointer a. Although s has type `struct device *`, it really points to `cd_devsize` bytes, as noted above. In this way each device instance data can retain all the values needed for that instance.

Device match functions, called via the `cd_match` pointer, also take three parameters. Two are the same as for `cd_attach`, but the middle parameter is not a new instance but rather a pointer t to a `cfdata` structure. Match functions can return 0, meaning "p, t, and a do not go together, so try another", or some nonzero value to try to "take" that configuration.

5.3.1 Direct Configuration

To perform direct configuration, a device driver's `attach` function should probe the device to find any actual hardware. For each "found" device, the driver should then call `config_found`, passing its self pointer s, a pointer a to any auxiliary data desired, and a pointer to a "print function" f_p. The `config_found` function will in turn call all potential `match` functions as determined by the configuration table. These match functions may compete for the device; the largest return value "wins". The winning `cfdata` is used to produce a new `struct device`, which will then be attached via the appropriate `cd_attach`. Thus, a driver for some particular device can "take over" from a generic driver simply by returning a bigger match value. In any case, the newly attached device should recursively find and attach any subdevices as appropriate.

[15] Of course, until one is written and configured in, the system will keep complaining.

If no driver takes a match, `config_found` uses the print function f_p to complain. Clearly this complaint cannot come from the child device driver—after all, no driver acknowledged such a role—so f_p is associated instead with the parent device p. The print function is called with the auxiliary data pointer and the full name (including unit number) of the parent device. If the device *is* matched, the system prints the name of the child and parent devices, and then calls the *same* print function f_p to produce additional information if desired. In the former case, f_p might be expected to produce something like `device type 0x401 at sba0 slot 6 not configured`.[16] In the latter, f_p should just print something like "` slot 6`", the kernel having already output `xyz0 at sba0`. Print functions can distinguish this second case by a `NULL` parent name.

Each print function must return an integer value. Two special strings, "` not configured`" and "` unsupported`", will be appended automatically to no-driver reports if the return value is `UNCONF` or `UNSUPP` respectively; otherwise the function should return the value `QUIET`. When a print function is called simply to produce auxiliary configuration text ("` slot 6`" above), this return value is ignored, so many print functions can simply return `UNCONF`.

The `config_found` function returns 1 if the device gets attached, 0 if not. Most callers can ignore this value, since the system will already have printed a diagnostic.

5.3.2 Indirect Configuration

When indirect configuration is required, a bus attach function can call `config_search`, rather than `config_found`, to apply some function to each configured potential child device on that bus. Unlike `config_found`, `config_search` has no need for a print function, since it simply applies the given function wherever the system configuration offers a possible match.

Specifically, `config_search` takes a pointer to a function f_a, a parent device p, and an auxiliary pointer a. It applies f_a to all configured children of p. If the pointer to function f_a is given as `NULL`, `config_search` applies each configured device's `cd_match` instead.[17] The `config_search` function returns a pointer to the "best match" `cfdata`; callers may ignore this value if appropriate.

A bus adapter driver can therefore have its own code applied (by supplying a non-`NULL` pointer), and make use of or ignore the driver's `cd_match` as it sees fit; or, alternatively, it can have all applicable `cd_match` functions called. Either way, for each device found, some code somewhere must call `config_attach`, passing a parent pointer p, a configuration data pointer t, an auxiliary pointer a, and a print function pointer f_p. The `config_attach` function allocates the new device instance data, invalidates other aliases if necessary, and calls the driver's `cd_attach` function. It also calls the print function f_p just as for direct configuration—and indeed, `config_found` is simply a "wrapper" for searching, and then attaching or complaining depending on the result of that search.

5.3.3 Finding the Root Device

Since the root, as the top of the tree, has no parent, it is impossible to use either `config_search` or `config_found` to find it. Instead, there are two similar functions named `config_rootsearch` and `config_rootfound`. These take the root device's name (as a regular C string, without a unit number suffix) in place of a parent pointer. This name is "magically divined" by the machine-dependent startup code. The `config_rootfound` function also omits the print function; it simply

[16] Names are generally superior to device type numbers, but the name-to-number decoding may well be known only to one driver. If names can be derived in the adapter driver, these can be put in the auxiliary data instead of, or in addition to, the type numbers. See Appendix C for another alternative.

[17] Thus, f_a receives the same arguments and returns the same value as a `cd_match` function.

prints "root device *name* not configured" and returns 0 if it cannot find the given root name in the system configuration. (At this point, the machine-dependent part of the kernel must presumably halt, since there will be no way to get anything done.)

6 Interrupts and Vectors

The code needed to handle interrupts is strongly machine dependent. For the most part, the new configuration system attempts to avoid the issue. Nonetheless, a base device definition can specify a series of interrupt vector function names. These turn into pointers to "vector stubs". Each `cfdata` structure has a `cf_ivstubs` field of type `void (**)()`. If not `NULL`, each `cf_ivstubs` points to the first of n pointers-to-functions, where n is the number of vectors listed on the base device definition. This list is followed by a `NULL` function pointer. The names of these functions are formed by appending the device instance unit number to the vector name, prefixed with an uppercase "X".[18] Thus, `vector dhrint dhxint` for device `dh2` would turn into the sequence `Xdhrint2 Xdhxint2`.

`Config` merely refers to these functions; their definitions are left to other code. In order to assist this other code, however, `config` emits one line per function name, suitable for use in an `awk` script. Each function declaration is preceded with a comment of the form `/* IVEC` *vector unit* `*/`. For `dh2`, for instance, these would become:

```
/* IVEC dhrint 2 */ extern void Xdhrint2();
/* IVEC dhxint 2 */ extern void Xdhxint2();
```

Writing an `awk` script to read these and produce assembly code for the vector stubs is typically straightforward.

Note that any device that includes vectors cannot be cloned, since the vectors encode the unit number. The best solution is usually to generate interrupt vector code "on the fly" when the device is attached. The correct unit, or a pointer to the dynamically-allocated device instance data, can then be inserted directly in the vector. For this reason, hardcoded vectors of this sort are discouraged.

A `Config` syntaxes

These are divided into one for the "files" files (`../../conf/files` and, e.g., `files.snail`) and one for machine description files (e.g., `SS1`). There are some shared non-terminals as well.

A.1 Lexical elements

All the syntaxes have identical lexical elements. Aside from the various keywords, everything is either a pathname, a word, a number, or a special character. Words are strings that match the regular expression `[A-Za-z_][-A-Za-z_0-9]*`, or arbitrary text can be made to act as a single word by enclosing it in double quotes. Pathnames differ from words by containing one or more slashes and/or periods (in fact, all file names should contain at least one period, e.g., "`.c`" or "`.s`"). Numbers begin with a digit. Octal and hexadecimal constants are written as in the C language; all others must be decimal. A "`#`" character introduces a comment, which extends to the end of the line. Lines that begin with whitespace act as continuations of preceding lines; all other line boundaries are significant.

[18]Instead of "X", `config` uses an uppercase "V" for the i386. Don't ask.

Other than when it separates tokens, white space (excluding newlines) is not significant. Note that most word forms, particularly device instance names, are single words (this differs from previous versions of `config`, where names like `dh0` comprised the two tokens `dh` and `0`). Digit suffix requirements are enforced in the semantics, rather than in the grammar.

A.2 Shared nonterminals

file → `file` *pathname foptions fflags rule*$_{opt}$
foptions → *foptions word*
  | ⟨empty⟩
fflags → *fflags fflag*
  | ⟨empty⟩
fflag → `config-dependent`
  | `needs-count`
  | `needs-flag`
rule → `compile-with` *word*

A.3 Grammar for `files` files

devfile-specs → *devfile-specs devfile-spec*
  | ⟨empty⟩
devfile-spec → `define` *attribute interface*$_{opt}$
  | `device` *device* `at` *at-list vectors*$_{opt}$ *interface*$_{opt}$ *attributes*
  | *file*
  | `maxusers` *minusers defaultusers maxusers*
  | `major {` *major-list* `}`
  | `pseudo-device` *device attributes*
interface → `{` *locator-list* `}`
locator-list → *locator-list*`,` *locator*
  | *locator*
locator → *name*
  | *name* `=` *default-value*
  | `[` *name* `=` *default-value* `]`
default-value → *name*
  | *signed-number*
at-list → *at-list*`,` *attribute*
  | *attribute*
vectors → `vector` *vector-names*
vector-names → *vector-names name*
  | *name*
attributes → `:` *attribute-list*
  | ⟨empty⟩
attribute-list → *attribute-list*`,` *attribute*
  | *attribute*
attribute → *word*
minusers → *number*
defaultusers → *number*
maxusers → *number*

major-list → *major-list*`,` *major*
  | *major*
major → *device-name* `=` *number*

A *name* is lexically just a word, but should typically be restricted to valid C identifiers.[19] A *device* is simply a device base name, i.e., an alphanumeric string that does not end in a digit. A *number* is an unsigned integer; a *signed-number* allows negative values as well.

A.4 Machine descriptions

configuration → *includes machine-type system-file*
includes → *includes include*
  | ⟨empty⟩
include → `include` *word*
machine-type → `machine` *name*
system-file → *system-file system-spec*
  | ⟨empty⟩
system-spec → *include*
  | *global-spec*
  | *config-spec*
  | *device-spec*
global-spec → *file*
  | `options` *option-list*
  | `makeoptions` *makeoption-list*
  | `maxusers` *number*
option-list → *option-list*`,` *option*
  | *option*
option → *name* `=` *value*
  | *name*
makeoption-list → *makeoption-list*`,` *makeoption*
  | *makeoption*
makeoption → *name* `=` *value*
value → *name*
  | *signed-number*
config-spec → `config` *name system-parameter-list*
system-parameter-list → *system-parameter-list system-parameter*
  | *system-parameter*
system-parameter → `root` `on`$_{opt}$ *block-device*
  | `swap` `on`$_{opt}$ *swap-spec*
  | `dumps` `on`$_{opt}$ *block-device*
swap-spec → *swapdev-list*
  | `generic`
swapdev-list → *block-device* `and` *swapdev-list*
  | *block-device*

[19]In general, it is wise to avoid embedding special characters into names through the use of double quotes. It not only makes them inconvenient to use, but if such a name is exported to C code, it will cause compilation errors. As noted earlier, "placeholder" locators are a special exception to this rule.

block-device → *suffixed-device*
  | *device*
  | *major-minor*
major-minor → `major` *number* `minor` *number*
device-spec → *device-instance* `at` *attachment locators* flags_{opt}
  | *pseudo-device device* number_{opt}
device-instance → *device*
  | *star-device*
attachment → *parent-finder*
  | `root`
locators → *locators locator*
  | ⟨empty⟩
locator → *name value*
  | *name* `?`
flags → `flags` *number*

This time, a *device* is an instance name, i.e., an alphanumeric string that does end in a digit. A *star-device* is a base device name followed by a "`*`". A *suffixed-device* is an instance name followed by a letter between "a" and "h" inclusive, A *parent-finder* is a device or attribute name followed by either a number, or "`*`" or the wildcard marker "`?`". The remaining terminal symbols are the same as before.

B Changes

The following is a short summary of changes for those who have used the old `config` program.

1. The "cpu" keyword is gone. In general, one simply replaces `cpu` *word* with `options` *word.*

2. The "ident" keyword is gone. This keyword had two effects: it acted as an option, and it included an extra "files" file if that file existed. Each of these is now separately available.

3. The "timezone" keyword is gone, along with its associated syntax. Current systems ignore this timezone parameter completely, but some backward-compatibility code may need the old values. These can be obtained by defining the two options `TIMEZONE` and `DST`. For instance, `timezone 8 dst` can be replaced with:

   ```
   options TIMEZONE=480 # 8*60
   options DST=1 # U.S. style DST
   ```

4. Configuration files no longer describe devices as "master", "disk", "tape", and so forth, and all require a parent device or attribute. Thus, something like:

   ```
   master hpib0 at scode7
   master hpib1 at scode?
   disk   rd0 at hpib? slave 0
   disk   rd1 at hpib? slave ?
   ```

   is now described with:

```
hpib0 at mainbus0 scode 7
hpib1 at mainbus0 scode ?
rd0   at hpib? slave 0
rd1   at hpib? slave ?
```

In addition, clones (`rd*`) are often available, making large configurations simpler.

5. "Ether" is no longer a pseudo-device.

C Code Example

A simplified and idealized version of a bus adapter driver may prove illuminating. We present here the configuration portion of a driver for the SnailBus adapter.

The SnailBus presents a number of "bus slots"; each slot has a type code. A special type code means the slot is unoccupied. All other codes indicate that there is some kind of hardware in that slot. The SnailBus adapter itself occupies slot 0 in its own address space, and has its own type code to describe itself. The adapter registers might be described in an "sbareg.h" file in the following manner.

```
/*
 * SnailBus registers.  Each SnailBus slot has 256 bytes of address space,
 * of which the first 4 bytes are a device type.  There are 16 slots,
 * but slot 0 is used for the SnailBus adapter registers themselves.
 */
struct sb_slot {
        u_int   ss_type;
        char    ss_pad[256 - 4];
};

#define SBA_NSLOTS      16
struct sbareg {
        u_int   sb_typesba;             /* type code: always SB_SBA */
        u_int   sb_csr;                 /* adapter control & status */
        u_int   sb_ver;                 /* version */
        char    sb_pad[256 - 3*sizeof(u_int)];
        struct  sb_slot sb_slots[SBA_NSLOTS - 1];
};

/*
 * SnailBus slot types.
 */
#define SB_EMPTY        0xffffffff      /* empty slot */
#define SB_SBA          0               /* SnailBus Adapter */
#define SB_SCSI         1               /* SCSI host adapter */
#define SB_BD           2               /* bitmapped display */
#define SB_IPIDISK      3               /* IPI disk - unsupported */
#define SB_EN           4               /* Ethernet card */
#define SB_MAXTYPE      5

#ifdef SB_TYPENAMES
static char *sbaslottypes[] =
    { "sba", "SCSI host adapter", "bitmapped display", "IPI disk", "Ethernet" };
```

```
#endif

/*
 * Bits in SB_CSR.
 */
#define SB_RESET        1       /* reset adapter */
#define SB_FAST         2       /* set burst transactions (v2 only) */
```

In order to configure devices on a SnailBus, we will have to probe each of the slots. Each subdevice on the SnailBus will thus need to match against a type code and a slot number. This gives us the following “sbavar.h” variables:

```
/*
 * SnailBus autoconfiguration information.
 */
struct sba_attach_args {
        int     sa_slot;
        u_int   sa_type;
};
```

Finally, the SnailBus adapter configuration driver might look something like this. Note that `sbaprint` is called whether or not the appropriate driver is found. It will typically result in something like “sc0 at xba0 slot 3” (where “sc0 at xba0” has already been printed) or “IPI disk at xba0 slot 9 unsupported” (where “unsupported” is printed because `xbaprint` returns `UNSUPP`). (The file `mainbusvar.h`, which is not shown here, presumably defines the `mb_attach_args` structure appropriately for the machine's main bus.)

```
#include <sys/param.h>
#include <sys/device.h>

#include "mainbusvar.h"
#define SB_TYPENAMES
#include "sbareg.h"
#include "sbavar.h"

/* software state per SnailBus adapter */
struct sba_softc {
        struct  device sc_dev;          /* base device, must be first */
        volatile struct sbareg *sc_sba; /* hardware registers */
};

static int sbamatch(struct device *, struct cfdata *, void *);
static void sbaattach(struct device *, struct device *, void *);
static int sbaprint(void *, char *);
struct cfdriver sbacd =
    { NULL, "sba", sbamatch, sbaattach, DV_DULL, sizeof(struct sba_softc) };

static int
sbamatch(parent, cf, aux)
        struct device *parent;
        struct cfdata *cf;
        void *aux;
{
        struct mb_attach_args *ma = aux;
        volatile struct sbareg *sba;

        /* make sure it is at the right address, or allowed anywhere */
        if (cf->cf_loc[0] != ma->ma_addr && cf->cf_loc[0] != 0)
                return (0);

        /* make sure it claims to be an SBA */
        sba = (volatile struct sbareg *)ma->ma_addr;
        if (sba->sb_typesba != SB_SBA)
                return (0);

        /* we like it; take it */
        return (1);
}

static void
sbaattach(parent, self, aux)
        struct device *parent, *self;
        void *aux;
{
        struct sba_softc *sc = (struct sba_softc *)self;
        struct mb_attach_args *ma = aux;
        struct sba_attach_args args;
        volatile struct sbareg *sba;
        volatile struct sb_slot *ss;
        int slot;

        /* locate registers and print hardware revision */
        sba = (volatile struct sbareg *)ma->ma_addr;
        printf(": SnailBus version %d\n", sba->sb_ver);

        /* remember register address */
        sc->sc_sba = sba;

        /* scan all the slots */
        for (slot = 1; slot < SBA_NSLOTS; slot++) {
                ss = &sba->sba_slots[slot - 1];
                if (ss->ss_type == SB_EMPTY)
                        continue;
                args.sa_type = ss->ss_type;
                args.sa_slot = slot;
                (void)config_found(&sc->sc_dev, (void *)&args, sbaprint);
        }
}

static int
sbaprint(aux, sbaname)
        void *aux;
        char *sbaname;
{
        struct sba_attach_args *sa = aux;
        u_int ty = sa->sa_type;
        if (sbaname != NULL) {
                if (ty >= SB_MAXTYPE)
                        printf("SnailBus type %u", ty);
                else
                        printf("%s", sbaslottypes[ty]);
                printf(" at %s", sbaname);
        }
        printf(" slot %d", sa->sa_slot);
        return (ty == SB_IPIDISK || ty >= SB_MAXTYPE ? UNSUPP : UNCONF);
}
```

D Makefile

As noted earlier, `config` produces a `Makefile` based on a prototype. In our example, this prototype would be called `Makefile.snail`. Most of the prototype file is copied unchanged, but certain "`%`-constructs" are expanded. These are listed below. The description here is rather sketchy, since most of the details are actually machine-dependent; the best way to produce a prototype makefile for a new machine is to modify one for an existing machine. Other changes to prototype makefiles shoudl be straightforward (and rare).

`%OBJS`
: This is replaced with a definition of the "make" variable `OBJS`, which will list all the object files making up the target kernel, according to the various "files" files. Note that on most systems, the prototype makefile will define some "special" objects (such as the traditional `locore.o`) that need special handling (such as being first in the link command). These will not appear in a "files" file, and hence also not in `OBJS`.

`%CFILES`
: This is replaced with a definition for `CFILES`, which will list of all the source files whose names end in "c", according to the "files" files.

`%SFILES`
: This is replaced with a definition for `SFILES`, which will list of all the source files whose names end in "s", according to the "files" files.

`%LOAD`
: This is replaced with several sets of lines, one set per `config` line in the system description file (here `SS1`). Each set of lines makes up a "load" command for a kernel. For a kernel named *k*, `config` will emit the following four lines:

```
k: ${SYSTEM_DEP} swapk.o
        ${SYSTEM_LD_HEAD}
        ${SYSTEM_LD} swapk.o
        ${SYSTEM_LD_TAIL}
```

The rest of the makefile must define the variables used here.

`%RULES`
: This is replaced with individual compilation rules for the various objects involved. Most will simply invoke `${NORMAL_C}` or `${NORMAL_C_C}`. Files marked `device-driver` will invoke

`${DRIVER_C}`; assembly files whose name ends with "`.s`" will use `${NORMAL_S}`; and so on. Again, the rest of the makefile must define these variables.

All source file references will be prefixed with `$S/` unless the file name is absolute (begins with a slash). The prototype makefile must set `S` to the path of the source directory, i.e., "`../..`".

E Output Files

In addition to the various "`.h`" files generated from `needs-count` or `needs-flag` names, `config` emits two or more "`.c`" files and makes a symbolic link. All of these files wind up in a directory of the form `../../compile/`*system-name*, e.g., `../../compile/SS1`.

The symbolic link is named `machine` and points back to `../../`*machine-type*`/include` (here `../../snail/include`). This allows references of the form `<machine/types.h>`, given that the compiler is run with appropriate `-I` flags.

For each kernel *k* named in a `config` line, the `config` program emits a file called `swap`*k*`.c`. Thus, for `config vmunix root on sd0`, `config` would generate a `swapvmunix.c` giving `sd0` as the root, swap, and crash dump device. These same names appear in link lines from the `%LOAD` directive. (The `config` program also emits compilation rules for these, using `${NORMAL_C}`.)

The last file is called `ioconf.c`. This file contains the `cfdata` table, along with tables of locators, parent indicies, interrupt vectors, and tree-root indices. The top of the file is copied from `ioconf.incl.`*machine-name* (`ioconf.incl.snail`), if it exists. If not, `config` emits a default set of `#include` lines. The `ioconf.incl` file, if any, can override or augment these, e.g., in order to define symbolic locators.[20] The remainder of the file consists of, in order:

- Interrupt vectors, if any.
- External declarations for configuration drivers.
- The locator table.
- The parent vector index table.
- The table of configuration data.
- The table of root device indicies.
- A set of pseudo-device initializations.[21]

The configuration data table is simply an array of `struct cfdata`, with one entry for each device alias. Each `cfdata` carries a driver pointer, a unit number, a "state" flag (found/not-found/clone), a pointer to the first of its locators, flags from the machine description file (default 0), a pointer to possible parent devices, and a (possibly `NULL`) pointer to the first of its interrupt vector stubs. Locator vector lengths are implicit. Parent device vectors are terminated by a -1 value, and interrupt stubs are terminated by a `NULL` pointer. The entire `cfdata` table itself is terminated by an entry with a `NULL` driver pointer.

For obvious reasons, device drivers should avoid using these "internals", instead going through the official `config_search` and `config_found` interfaces. These are described here mainly for completeness (and for anyone who needs to work on `config`).

[20] Note that `config` cannot do as much to minimize locator duplication when symbolic names are used. This effect is usually small. It may someday be worth fixing by having `config` expand the symbolic names; this would make the `ioconf.incl` file unnecessary.

[21] Currently this is disabled with `#ifdef`, as the rest of the system has not been converted for them.

References

[1] Leffer, Samuel J. and Karels, Michael J. "Building Berkeley UNIX Kernels with Config," 4.3BSD UNIX System Manager's Manual, online.

[2] Torek, Chris. "A New Framework for Device Support in Berkeley UNIX," *Proceedings of the UKUUG*, London (Summer 1990).