Building 4.2BSD UNIX† Systems with Config
June, 1983

Samuel J. Leffler

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

ABSTRACT

This document describes the use of *config* (8) to configure and create bootable 4.2BSD system images. It discusses the structure of system configuration files and how to configure systems with non-standard hardware configurations. Sections describing the preferred way to add new code to the system and how the system's autoconfiguration process operates are included. An appendix contains a summary of the rules used by the system in calculating the size of system data structures, and also indicates some of the standard system size limitations (and how to change them).

†UNIX is a Trademark of Bell Laboratories.

1. INTRODUCTION

Config is a tool used in building 4.2BSD system images. It takes a file describing a system's tunable parameters and hardware support, and generates a collection of files which are then used to build a copy of UNIX appropriate to that configuration. *Config* simplifies system maintenance by isolating system dependencies in a single, easy to understand, file.

This document describes the content and format of system configuration files and the rules which must be followed when creating these files. Example configuration files are constructed and discussed.

Later sections suggest guidelines to be used in modifying system source and explain some of the inner workings of the autoconfiguration process. Appendix D summarizes the rules used in calculating the most important system data structures and indicates some inherent system data structure size limitations (and how to go about modifying them).

2. CONFIGURATION FILE CONTENTS

A system configuration must include at least the following pieces of information:

- machine type
- cpu type
- system identification
- timezone
- maximum number of users
- location of the root file system
- available hardware

Config allows multiple system images to be generated from a single configuration description. Each system image is configured for identical hardware, but may have different locations for the root file system and, possibly, other system devices.

0.1. Machine type

The *machine type* indicates if the system is going to operate on a DEC VAX-11 computer, or some other machine on which 4.2BSD operates. The machine type is used to locate certain data files which are machine specific and, also, to select rules used in constructing the resultant configuration files.

0.2. Cpu type

The *cpu type* indicates which, of possibly many, cpu's the system is to operate on. For example, if the system is being configured for a VAX-11, it could be running on a VAX-11/780, VAX-11/750, or VAX-11/730. Specifying more than one cpu type implies the system should be configured to run on all the cpu's specified. For some types of machines this is not possible and *config* will print a diagnostic indicating such.

0.3. System identification

The *system identification* is a moniker attached to the system, and often the machine on which the system is to run. For example, at Berkeley we have machines named Ernie (Co-VAX), Kim (No-VAX), and so on. The system identifier selected is used to create a global C ``#define'' which may be used to isolate system dependent pieces of code in the kernel. For example, Ernie's Varian driver used to be special cased because its interrupt vectors were wired together. The code in the driver which understood how to handle this non-standard hardware configuration was conditionally compiled in only if the system was for Ernie.

The system identifier ``GENERIC'' is given to a system which will run on any cpu of a particular machine type; it should not otherwise be used for a system identifier.

0.4. Timezone

The timezone in which the system is to run is used to define the information returned by the *gettimeofday*(2) system call. This value is specified as the number of hours east or west of GMT. Negative numbers indicate a value east of GMT. The timezone specification may also indicate the type of daylight savings time rules to be applied.

0.5. Maximum number of users

The system allocates many system data structures at boot time based on the maximum number of users the system will support. This number is normally between 8 and 40, depending on the hardware and expected job mix. The rules used to calculate system data structures are discussed in Appendix D.

0.6. Root file system location

When the system boots it must know the location of the root of the file system tree. This location and the part(s) of the disk(s) to be used for paging and swapping must be specified in order to create a complete configuration description. *Config* uses many rules to calculate default locations for these items; these are described in Appendix B.

When a generic system is configured, the root file system is left undefined until the system is booted. In this case, the root file system need not be specified, only that the system is a generic system.

0.7. Hardware devices

When the system boots it goes through an *autoconfiguration* phase. During this period, the system searches for all those hardware devices which the system builder has indicated might be present. This probing sequence requires certain pieces of information such as register addresses, bus interconnects, etc. A system's hardware may be configured in a very flexible manner or be specified without any flexibility whatsoever. Most people do not configure hardware devices into the system unless they are currently present on the machine, expect them to be present in the near future, or are simply guarding against a hardware failure somewhere else at the site (it is often wise to configure in extra disks in case an emergency requires moving one off a machine which has hardware problems).

The specification of hardware devices usually occupies the majority of the configuration file. As such, a large portion of this document will be spent understanding it. Section 6.3 contains a description of the autoconfiguration process, as it applies to those planning to write, or modify existing, device drivers.

0.8. Optional items

Other than the mandatory pieces of information described above, it is also possible to include various optional system facilities. For example, 4.2BSD can be configured to support binary compatibility for programs built under 4.1BSD. Also, optional support is provided for disk quotas and tracing the performance of the virtual memory subsystem. Any optional facilities to be configured into the system are specified in the configuration file. The resultant files generated by *config* will automatically include the necessary pieces of the system.

3. SYSTEM BUILDING PROCESS

In this section we consider the steps necessary to build a bootable system image. We assume the system source is located in the ''/sys'' directory and that, initially, the system is being configured from source code.

Under normal circumstances there are 5 steps in building a system.

1) Create a configuration file for the system.

2) Make a directory for the system to be constructed in.

3) Run *config* on the configuration file to generate the files required to compile and load the system image.

4) Construct the source code interdependency rules for the configured system.

5) Compile and load the system with *make*(1).

Steps 1 and 2 are usually done only once. When a system configuration changes it usually suffices to just run *config* on the modified configuration file, rebuild the source code dependencies, and remake the system. Sometimes, however, configuration dependencies may not be noticed in which case it is necessary to clean out the relocatable object files saved in the system's directory; this will be discussed later.

3.1. Creating a configuration file

Configuration files normally reside in the directory ''/sys/conf''. A configuration file is most easily constructed by copying an existing configuration file and modifying it. The 4.2BSD distribution contains a number of configuration files for machines at Berkeley, one may be suitable or, in worst case, you may take the generic configuration file and edit that.

The configuration file must have the same name as the directory in which the configured system is to be built. Further, *config* assumes this directory is located in the parent directory of the directory in which it is run. For example, the generic system has a configuration file ''/sys/conf/GENERIC'' and an accompanying directory named ''/sys/GENERIC''. In general it is unwise to move your configuration directories out of ''/sys'' as most of the system code and the files created by *config* use pathnames of the form ''../''. If you are running out of space on the file system where the configuration directories are located there is a mechanism for sharing relocatable object files between systems; this is described later.

When building your configuration file, be sure to include the items described in section 2. In particular, the machine type, cpu type, timezone, system identifier, maximum users, and root device must be specified. The specification of the hardware present may take a bit of work; particularly if your hardware is configured at non-standard places (e.g. device registers located at funny places or devices not supported by the system). Section 4 of this document gives a detailed description of the configuration file syntax, section 5 explains some sample configuration files, and section 6 discusses how to add new devices to the system. If the devices to be configured are not already described in one of the existing configuration files you should check the manual pages in section 4 of the UNIX Programmers Manual. For each supported device, the manual page synopsis entry gives a sample configuration line.

Once the configuration file is complete, run it through *config* and look for any errors. Never try and use a system which *config* has complained about; the results are unpredictable. For the most part, *config*'s error diagnostics are self explanatory. It may be the case that the line numbers given with the error messages are off by one.

A successful run of *config* on your configuration file will generate a number of files in the configuration directory. These files are:

- A file to be used by *make* (1) in compiling and loading the system.
- One file for each possible system image for your machine which describes where swapping, the root file system, and other miscellaneous system devices are located.
- A collection of header files, one per possible device the system supports, which define the hardware configured.

- A file containing the i/o configuration tables used by the system during its *autoconfiguration* phase.
- An assembly language file of interrupt vectors which connect interrupts from your machine's external buses to the main system path for handling interrupts.

Unless you have reason to doubt *config*, or are curious how the system's autoconfiguration scheme works, you should never have to look at any of these files.

3.2. Constructing source code dependencies

When *config* is done generating the files needed to compile and link your system it will terminate with a message of the form ‘‘Don't forget to run make depend’’. This is a reminder that you should change over to the configuration directory for the system just configured and type ‘‘make depend’’ to build the rules used by *make* to recognize interdependencies in the system source code. This will insure that any changes to a piece of the system source code will result in the proper modules being recompiled the next time *make* is run.

This step is particularly important if your site makes changes to the system include files. The rules generated specify which source code files are dependent on which include files. Without these rules, *make* will not recognize when it must rebuild modules due to a system header file being modified. Note that dependency rules created by this step only reflect directly included files. That is, if file ‘‘a’’ includes another file ‘‘b’’, which includes yet another, say ‘‘c’’, and then ‘‘c’’ is modified, *make* will not recognize that ‘‘a’’ should be recompiled. It is best to keep include file dependencies only one level deep.

3.3. Building the system

The makefile constructed by *config* should allow a new system to be rebuilt by simply typing ‘‘make image-name’’. For example, if you have named your bootable system image ‘‘vmunix’’, then ‘‘make vmunix’’ will generate a bootable image named ‘‘vmunix’’. Alternate system image names are used when the root file system location and/or swapping configuration is done in more than one way. The makefile which *config* creates has entry points for each system image defined in the configuration file. Thus, if you have configured ‘‘vmunix’’ to be a system with the root file system on an ‘‘hp’’ device and ‘‘hkvmunix’’ to be a system with the root file system on an ‘‘hk’’ device, then ‘‘make vmunix hkvmunix’’ will generate binary images for each.

Note that the name of a bootable image is different from the system identifier. All bootable images are configured for the same system; only the information about the root file system and paging devices differ. (This is described in more detail in section 4.)

The last step in the system building process is to rearrange certain commonly used symbols in the symbol table of the system image; the makefile generated by *config* does this automatically for you. This is advantageous for programs such as *ps*(1) and *vmstat*(1), which run much faster when the symbols they need are located at the front of the symbol table. Remember also that many programs expect the currently executing system to be named ‘‘/vmunix’’. If you install a new system and name it something other than ‘‘/vmunix’’, many programs are likely to give strange results.

3.4. Sharing object modules

If you have many systems which are all built on a single machine there are at least two approaches to saving time in building system images. The best way is to have a single system image which is run on all machines. This is attractive since it minimizes disk space used and time required to rebuild systems after making changes. However, it is often the case that one or more systems will require a separately configured system image. This may be due to limited memory (building a system with many unused device drivers can be expensive), or to configuration requirements (one machine may be a development machine where disk quotas are not needed, while another is a production machine where they are), etc. In these cases it is possible for common systems to share relocatable object modules which are not configuration dependent; most of the module in the directory ‘‘/sys/sys’’ are of this sort.

To share object modules, a generic system should be built. Then, for each system configure the system as before, but before recompiling and linking the system, type ‘‘make links’’. This will cause the system to be searched for source modules which are safe to share between systems and generate symbolic links in the current directory to the appropriate object modules in the directory ‘‘../GENERIC’’. A shell script, ‘‘makelinks’’ is generated with this request and may be checked for correctness. The file ‘‘/sys/conf/defines’’ contains a list of symbols which we believe are safe to ignore when checking the source code for modules which may be shared. Note that this list includes the definitions used to conditionally compile in the virtual memory tracing facilities, and the trace point support used only rarely (even at Berkeley). It may be necessary to modify this file to reflect local needs. Note further, that as described previously, interdependencies which are not directly visible in the source code are not caught. This means that if you place per-system dependencies in an include file, they will not be recognized and the shared code may be selected in an unexpected fashion.

3.5. Building profiled systems

It is simple to configure a system which will automatically collect profiling information as it operates. The profiling data may be collected with *kgmon* (8) and processed with *gprof* (1) to obtain information regarding the system’s operation. Profiled systems maintain histograms of the program counter as well as the number of invocations of each routine. The *gprof* (1) command will also generate a dynamic call graph of the executing system and propagate time spent in each routine along the arcs of the call graph (consult the gprof documentation for elaboration). The program counter sampling can be driven by the system clock, or if you have an alternate real time clock this can be used. The latter is highly recommended as use of the system clock will result in statistical anomalies and time spent in the clock routine will not be accurately accounted for.

To configure a profiled system, the **−p** option should be supplied to *config*. A profiled system is about 5-10% larger in its text space due to the calls to count the subroutine invocations. When the system executes, the profiling data is stored in a buffer which is 1.2 times the size of the text space. The overhead for running a profiled system varies; under normal load we see anywhere from 5-25% of the system time spent in the profiling code.

Note that systems configured for profiling should not be shared as described above unless all the other shared systems are also to be profiled.

4. CONFIGURATION FILE SYNTAX

In this section we consider the specific rules used in writing a configuration file. A complete grammar for the input language can be found in Appendix A and may be of use if you should have problems with syntax errors.

A configuration file is broken up into three logical pieces:

- configuration parameters global to all system images specified in the configuration file,
- parameters specific to each system image to be generated, and
- device specifications.

4.1. Global configuration parameters

The global configuration parameters are the type of machine, cpu types, options, timezone, system identifier, and maximum users. Each is specified with a separate line in the configuration file.

machine *type*

The system is to run on the machine type specified. No more than one machine type can appear in the configuration file. Legal values are **vax** and **sun**.

cpu ‘‘*type*’’

This system is to run on the cpu type specified. More than one cpu type specification can appear in a configuration file. Legal types for a **vax** machine are **VAX780**,**VAX750**, and **VAX730**.

options *optionlist*

Compile the listed optional code into the system. Options in this list are separated by commas. Possible options are listed at the top of the generic makefile. A line of the form ‘‘options FUNNY,HAHA’’ generates global ‘‘#define’’s −DFUNNY −DHAHA in the resultant makefile. An option may be given a value by following its name with ‘‘=’’, then the value enclosed in (double) quotes. None of the standard options use such a value. The following options are currently in use: COMPAT (include code for compatiblity with 4.1BSD binaries), INET (Internet communication protocols), PUP (support for a PUP raw interface), and QUOTA (enable disk quotas). There are additional options which are associated with certain peripheral devices; those are listed in the Synopsis section of the manual page for the device.

timezone *number* [**dst** [*number*]]

Specifies the timezone you are in. This is measured in the number of hours your timezone is west of GMT. EST is 5 hours west of GMT, PST is 8. Negative numbers indicate hours east of GMT. If you specify **dst**, the system will operate under daylight savings time. An optional integer or floating point number may be included to specify a particular daylight saving time correction algorithm; the default value is 1, indicating the United States. Other values are: 2 (Australian style), 3 (Western European), 4 (Middle European), and 5 (Eastern European). See *gettimeofday* (2) and *ctime* (3) for more information.

ident *name*

This system is to be known as *name*. This is usually a cute name like ERNIE (short for Ernie Co-Vax) or VAXWELL (for Vaxwell Smart).

maxusers *number*

The maximum expected number of simultaneously active user on this system is *number*. This number is used to size several system data structures.

4.2. System image parameters

Multiple bootable images may be specified in a single configuration file. The systems will have the same global configuration parameters and devices, but the location of the root file system and other system specific devices may be different. A system image is specified with a ‘‘config’’ line:

config *sysname config-clauses*

The *sysname* field is the name given to the loaded system image; almost everyone names their standard system image ‘‘vmunix’’. The configuration clauses are one or more specifications indicating where the root file system is located, how many paging devices there are and where they go. The device used by the system to process argument lists during *execve*(2) calls may also be specified, though in practice this is almost always selected by *config* using one of its rules for selecting default locations for system devices.

A configuration clause is one of the following

root [**on**] *root-device*
swap [**on**] *swap-device* [**and** *swap-device*]
dumps [**on**] *dump-device*
args [**on**] *arg-device*

(the ‘‘on’’ is optional.) Multiple configuration clauses are separated by white space; *config* allows specifications to be continued across multiple lines by beginning the continuation line with a tab character. The ‘‘root’’ clause specifies where the root file system is located, the ‘‘swap’’ clause indicates swapping and paging area(s), the ‘‘dumps’’ clause can be used to force system dumps to be taken on a particular device, and the ‘‘args’’ clause can be used to specify that argument list processing for *execve* should be done on a particular disk.

The device names supplied in the clauses may be fully specified as a device, unit, and file system partition; or underspecified in which case *config* will use builtin rules to select default unit numbers and file system partitions. The defaulting rules are a bit complicated as they are dependent on the overall system configuration. For example, the swap area need not be specified at all if the root device is specified; in this case the swap area is placed in the ‘‘b’’ partition of the same disk where the root file system is located. Appendix B contains a complete list of the defaulting rules used in selecting system configuration devices.

The device names are translated to the appropriate major and minor device numbers on a per-machine basis. A file, ‘‘/sys/conf/devices.machine’’ (where ‘‘machine’’ is the machine type specified in the configuration file), is used to map a device name to its major block device number. The minor device number is calculated using the standard disk partitioning rules: on unit 0, partition ‘‘a’’ is minor device 0, partition ‘‘b’’ is minor device 1, and so on; for units other than 0, add 8 times the unit number to get the minor device.

If the default mapping of device name to major/minor device number is incorrect for your configuration, it can be replaced by an explicit specification of the major/minor device. This is done by substituting

major *x* **minor** *y*

where the device name would normally be found. For example,

config vmunix **root on major** 99 **minor** 1

Normally, the areas configured for swap space are sized by the system at boot time. If a non-standard partition size is to be used for one or more swap areas, this can also be specified. To do this, the device name specified for a swap area should have a ‘‘size’’ specification appended. For example,

config vmunix **root on** hp0 **swap on** hp0b **size** 1200

would force swapping to be done in partition ‘‘b’’ of ‘‘hp0’’ and the swap partition size would be set to 1200 sectors. A swap area sized larger than the associated disk partition is trimmed to the partition size.

To create a generic configuration, only the clause ‘‘swap generic’’ should be specified; any extra clauses will cause an error.

4.3. Device specifications

Each device attached to a machine must be specified to *config* so that the system generated will know to probe for it during the autoconfiguration process carried out at boot time. Hardware specified in the configuration need not actually be present on the machine where the generated system is to be run. Only the hardware actually found at boot time will be used by the system.

The specification of hardware devices in the configuration file parallels the interconnection hierarchy of the machine to be configured. On the VAX, this means a configuration file must indicate what

MASSBUS and UNIBUS adapters are present, and to which *nexi* they might be connected*. Similarly, devices and controllers must be indicated as possibly being connected to one or more adapters. A device description may provide a complete definition of the possible configuration parameters or it may leave certain parameters undefined and make the system probe for all the possible values. The latter allows a single device configuration list to match many possible physical configurations. For example, a disk may be indicated as present at UNIBUS adapter 0, or at any UNIBUS adapter which the system locates at boot time. The latter scheme, termed *wildcarding*, allows more flexibility in the physical configuration of a system; if a disk must be moved around for some reason, the system will still locate it at the alternate location.

A device specification takes one of the following forms:

master *device-name device-info*
controller *device-name device-info* [*interrupt-spec*]
device *device-name device-info interrupt-spec*
disk *device-name device-info*
tape *device-name device-info*

A ‘‘master’’ is a MASSBUS tape controller; a ‘‘controller’’ is a disk controller, a UNIBUS tape controller, a MASSBUS adapter, or a UNIBUS adapter. A ‘‘device’’ is an autonomous device which connects directly to a UNIBUS adapter (as opposed to something like a disk which connects through a disk controller). ‘‘Disk’’ and ‘‘tape’’ identify disk drives and tape drives connected to a ‘‘controller’’ or ‘‘master’’.

The *device-name* is one of the standard device names, as indicated in section 4 of the UNIX Programmers Manual, concatenated with the *logical* unit number to be assigned the device (the *logical* unit number may be different than the *physical* unit number indicated on the front of something like a disk; the *logical* unit number is used to refer to the UNIX device, not the physical unit number). For example, ‘‘hp0’’ is logical unit 0 of a MASSBUS storage device, even though it might be physical unit 3 on MASSBUS adapter 1.

The *device-info* clause specifies how the hardware is connected in the interconnection hierarchy. On the VAX, UNIBUS and MASSBUS adapters are connected to the internal system bus through a *nexus*. Thus, one of the following specifications would be used:

```
controller      mba0        at nexus x
controller      uba0        at nexus x
```

To tie a controller to a specific nexus, ‘‘x’’ would be supplied as the number of that nexus; otherwise ‘‘x’’ may be specified as ‘‘?’’, in which case the system will probe all nexi present looking for the specified controller.

The remaining interconnections on the VAX are:

- a controller may be connected to another controller (e.g. a disk controller attached to a UNIBUS adapter),
- a master is always attached to a controller (a MASSBUS adaptor),
- a tape is always attached to a master (for MASSBUS tape drives),
- a disk is always attached to a controller, and
- devices are always attached to controllers (e.g. UNIBUS controllers attached to UNIBUS adapters).

The following lines give an example of each of these interconnections:

```
controller      hk0         at uba0 ...
master          ht0         at mba0 ...
tape            tu0         at ht0 ...
disk            rk1         at hk0 ...
device          dz0         at uba0 ...
```

Any piece of hardware which may be connected to a specific controller may also be wildcarded across multiple controllers.

* While VAX-11/750’s and VAX-11/730 do not actually have nexi, the system treats them as having *simulated nexi* to simplify device configuration.

The final piece of information needed by the system to configure devices is some indication of where or how a device will interrupt. For tapes and disks, simply specifying the *slave* or *drive* number is sufficient to locate the control status register for the device. For controllers, the control status register must be given explicitly, as well how many interrupt vectors are used and the names of the routines to which they should be bound. Thus the example lines given above might be completed as:

```
controller        hk0         at uba0 csr 0177440      vector rkintr
master            ht0         at mba0 drive 0
tape              tu0         at ht0 slave 0
disk              rk1         at hk0 drive 1
device            dz0         at uba0 csr 0160100      vector dzrint dzxint
```

Certain device drivers require extra information passed to them at boot time to tailor their operation to the actual hardware present. The line printer driver, for example, needs to know how many columns are present on each non-standard line printer (i.e. a line printer with other than 80 columns). The drivers for the terminal multiplexors need to know which lines are attached to modem lines so that no one will be allowed to use them unless a connection is present. For this reason, one last parameter may be specified to a *device*, a *flags* field. It has the syntax

flags *number*

and is usually placed after the *csr* specification. The *number* is passed directly to the associated driver. The manual pages in section 4 should be consulted to determine how each driver uses this value (if at all). Communications interface drivers commonly use the flags to indicate whether modem control signals are in use.

The exact syntax for each specific device is given in the Synopsis section of its manual page in section 4 of the manual.

4.4. Pseudo-devices

A number of drivers and software subsystems are treated like device drivers without any associated hardware. To include any of these pieces, a ‘‘pseudo-device’’ specification must be used. A specification for a pseudo device takes the form

pseudo-device   *device-name* [*howmany*]

Examples of pseudo devices are **bk**, the Berknet line discipline, **pty**, the pseudo terminal driver (where the optional *howmany* value indicates the number of pseudo terminals to configure, 32 default), and **inet**, the DARPA Internet protocols (one must also specify INET in the ‘‘options’’). Other pseudo devices for the network include **loop**, the software loopback interface, **imp** (required when a CSS or ACC imp is configured), and **ether** (used by the Address Resolution Protocol on 10 Mb/sec ethernets). More information on configuring each of these can also be found in section 4 of the manual.

5. SAMPLE CONFIGURATION FILES

In this section we will consider how to configure a sample VAX-11/780 system on which the hardware can be reconfigured to guard against various hardware mishaps. We then study the rules needed to configure a VAX-11/750 to run in a networking environment.

5.1. VAX-11/780 System

Our VAX-11/780 is configured with hardware recommended in the document ‘‘Hints on Configuring a VAX for 4.2BSD’’ (this is one of the high-end configurations). Table 1 lists the pertinent hardware to be configured.

Item	Vendor	Connection	Name	Reference
cpu	DEC		VAX780	
MASSBUS controller	Emulex	nexus ?	mba0	hp(4)
disk	Fujitsu	mba0	hp0	
disk	Fujitsu	mba0	hp1	
MASSBUS controller	Emulex	nexus ?	mba1	
disk	Fujitsu	mba1	hp2	
disk	Fujitsu	mba1	hp3	
UNIBUS adapter	DEC	nexus ?		
tape controller	Emulex	uba0	tm0	tm(4)
tape drive	Kennedy	tm0	te0	
tape drive	Kennedy	tm0	te1	
terminal multiplexor	Emulex	uba0	dh0	dh(4)
terminal multiplexor	Emulex	uba0	dh1	
terminal multiplexor	Emulex	uba0	dh2	

Table 1. VAX-11/780 Hardware support.

We will call this machine ANSEL and construct a configuration file one step at a time.

The first step is to fill in the global configuration parameters. The machine is a VAX, so the *machine type* is ‘‘vax’’. We will assume this system will run only on this one processor, so the *cpu type* is ‘‘VAX780’’. The options are empty since this is going to be a ‘‘vanilla’’ VAX. The system identifier, as mentioned before, is ‘‘ANSEL’’ and the maximum number of users we plan to support is about 40. Thus the beginning of the configuration file looks like this:

```
#
# ANSEL VAX (a picture perfect machine)
#
machine         vax
cpu             VAX780
timezone        8 dst
ident           ANSEL
maxusers        40
```

To this we must then add the specifications for three system images. The first will be our standard system with the root on ‘‘hp0’’ and swapping on the same drive as the root. The second will have the root file system in the same location, but swap space interleaved among drives on each controller. Finally, the third will be a generic system, to allow us to boot off any of the four disk drives.

```
config          vmunix          root on hp0
config          hpvmunix        root on hp0 swap on hp0 and hp2
config          genvmunix       swap generic
```

Finally, the hardware must be specified. Let us first just try transcribing the information from Table 1.

```
controller     mba0     at nexus ?
disk           hp0      at mba0 disk 0
disk           hp1      at mba0 disk 1
controller     mba1     at nexus ?
disk           hp2      at mba1 disk 2
disk           hp3      at mba1 disk 3
controller     uba0     at nexus ?
controller     tm0      at uba0 csr 0172520     vector tmintr
tape           te0      at tm0 drive 0
tape           te1      at tm0 drive 1
device         dh0      at uba0 csr 0160020     vector dhrint dhxint
device         dm0      at uba0 csr 0170500     vector dmintr
device         dh1      at uba0 csr 0160040     vector dhrint dhxint
device         dh2      at uba0 csr 0160060     vector dhrint dhxint
```

(Oh, I forgot to mention one panel of the terminal multiplexor has modem control, thus the ‘‘dm0’’ device.)

This will suffice, but leaves us with little flexibility. Suppose our first disk controller were to break. We would like to recable the drives normally on the second controller so that all our disks could still be used without reconfiguring the system. To do this we wildcard the MASSBUS adapter connections and also the slave numbers. Further, we wildcard the UNIBUS adapter connections in case we decide some time in the future to purchase another adapter to offload the single UNIBUS we currently have. The revised device specifications would then be:

```
controller     mba0     at nexus ?
disk           hp0      at mba? disk ?
disk           hp1      at mba? disk ?
controller     mba1     at nexus ?
disk           hp2      at mba? disk ?
disk           hp3      at mba? disk ?
controller     uba0     at nexus ?
controller     tm0      at uba? csr 0172520     vector tmintr
tape           te0      at tm0 drive 0
tape           te1      at tm0 drive 1
device         dh0      at uba? csr 0160020     vector dhrint dhxint
device         dm0      at uba? csr 0170500     vector dmintr
device         dh1      at uba? csr 0160040     vector dhrint dhxint
device         dh2      at uba? csr 0160060     vector dhrint dhxint
```

The completed configuration file for ANSEL is shown in Appendix C.

5.2. VAX-11/750 with network support

Our VAX-11/750 system will be located on two 10Mb/s Ethernet local area networks and also the DARPA Internet. The system will have a MASSBUS drive for the root file system and two UNIBUS drives. Paging is interleaved among all three drives. We have sold our standard DEC terminal multiplexors since this machine will be accessed solely through the network. This machine is not intended to have a large user community, it does not have a great deal of memory. First the global parameters:

```
#
# UCBVAX (Gateway to the world)
#
machine         vax
cpu             "VAX780"
cpu             "VAX750"
ident           UCBVAX
timezone        8 dst
maxusers        32
options         INET
```

The multiple cpu types allow us to replace UCBVAX with a more powerful cpu without reconfiguring the system. The value of 32 given for the maximum number of users is done to force the system data structures to be over-allocated. That is desirable on this machine because, while it is not expected to support many users, it is expected to perform a great deal of work. Upping this value results in a larger disk buffer cache than would normally be allocated if the true number of users were given. The ‘‘INET’’ indicates we plan to use the DARPA standard Internet protocols on this machine.

The system images and disks are configured in next.

```
config          vmunix          root on hp swap on hp and rk0 and rk1
config          upvmunix        root on up
config          hkvmunix        root on hk swap on rk0 and rk1

controller      mba0            at nexus ?
controller      uba0            at nexus ?
disk            hp0             at mba? drive 0
disk            hp1             at mba? drive 1
controller      sc0             at uba? csr 0176700     vector upintr
disk            up0             at sc0 drive 0
disk            up1             at sc0 drive 1
controller      hk0             at uba? csr 0177440     vector rkintr
disk            rk0             at hk0 drive 0
disk            rk1             at hk0 drive 1
```

UCBVAX requires heavy interleaving of its paging area to keep up with all the mail traffic it handles. The limiting factor on this system’s performance is usually the number of disk arms, as opposed to memory or cpu cycles. The extra UNIBUS controller, ‘‘sc0’’, is in case the MASSBUS controller breaks and a spare controller must be installed (most of our old UNIBUS controllers have been replaced with the newer MASSBUS controllers, so we have a number of these around as spares).

Finally, we add in the network support. The Internet protocols require an ‘‘inet’’ *pseudo-device* in addition to the global ‘‘INET’’ option specified above. Pseudo terminals are needed to allow users to log in across the network (remember the only hardwired terminal is the console). The connection to the Internet is through an IMP, this requires yet another *pseudo-device* (in addition to the actual hardware device used by the IMP software). And, finally, there are the two Ethernet devices. These use a special protocol, the Address Resolution Protocol (ARP), to map between Internet and Ethernet addresses. Thus, yet another *pseudo-device* is needed. The additional device specifications are show below.

```
pseudo-device     inet
pseudo-device     pty
# software loopback device for testing
pseudo-device     loop
pseudo-device     imp
device            acc0      at uba? csr 0167600     vector accrint accxint
pseudo-device     ether
device            ec0       at uba? csr 0164330     vector ecrint eccollide ecxint
device            il0       at uba? csr 0164000     vector ilrint ilcint
```

The completed configuration file for UCBVAX is shown in Appendix C.

5.3. Miscellaneous comments

It should be noted in these examples that neither system was configured to use disk quotas or the 4.1BSD compatibility mode. To use these optional facilities, and others, we would probably clean out our current configuration, reconfigure the system, then recompile and relink the system image(s). This could, of course, be avoided by figuring out which relocatable object files are affected by the reconfiguration, then reconfiguring and recompiling only those files affected by the configuration change. This technique should be used carefully.

6. ADDING NEW SYSTEM SOFTWARE

This section is not for the novice, it describes some of the inner workings of the configuration process as well as the pertinent parts of the system autoconfiguration process. It is intended to give those people who intend to install new device drivers and/or other system facilities sufficient information to do so in the manner which will allow others to easily share the changes.

This section is broken into four parts:

- general guidelines to be followed in modifying system code,
- how to add a device driver to 4.2BSD,
- how UNIBUS device drivers are autoconfigured under 4.2BSD on the VAX, and
- how to add non-standard system facilities to 4.2BSD.

6.1. Modifying system code

If you wish to make site-specific modifications to the system it is best to bracket them with

```
#ifdef SITENAME
...
#endif
```

to allow your source to be easily distributed to others, and also to simplify *diff*(1) listings. If you choose not to use a source code control system (e.g. SCCS, RCS), and perhaps even if you do, it is recommended that you save the old code with something of the form:

```
#ifndef SITENAME
...
#endif
```

We try to isolate our site-dependent code in individual files which may be configured with pseudo-device specifications.

Indicate machine specific code with ‘‘#ifdef vax’’. 4.2BSD has undergone extensive work to make it extremely portable to machines with similar architectures − you may someday find yourself trying to use a single copy of the source code on multiple machines.

Use *lint* periodically if you make changes to the system. The 4.2BSD release has only one line of lint in it. It is very simple to lint the kernel. Use the LINT configuration file, designed to pull in as much of the kernel source code as possible, in the following manner.

```
$ cd /sys/conf
$ mkdir ../LINT
$ config LINT
$ cd ../LINT
$ make depend
$ make assym.s
$ make −k lint > linterrs 2>&1 &
(or for users of csh(1))
% make −k >& linterrs
```

This takes about 45 minutes on a lightly loaded VAX-11/750, but is well worth it.

6.2. Adding device drivers to 4.2BSD

The i/o system and *config* have been designed to easily allow new device support to be added. As described in ‘‘Installing and Operating 4.2BSD on the VAX’’, the system source directories are organized as follows:

/sys/h	machine independent include files
/sys/sys	machine independent system source files
/sys/conf	site configuration files and basic templates
/sys/net	network independent, but network related code
/sys/netinet	DARPA Internet code
/sys/netimp	IMP support code
/sys/netpup	PUP-1 support code
/sys/vax	VAX specific mainline code
/sys/vaxif	VAX network interface code
/sys/vaxmba	VAX MASSBUS device drivers and related code
/sys/vaxuba	VAX UNIBUS device drivers and related code

Existing block and character device drivers for the VAX reside in ‘‘/sys/vax’’, ‘‘/sys/vaxmba’’, and ‘‘/sys/vaxuba’’. Network interface drivers reside in ‘‘/sys/vaxif’’. Any new device drivers should be placed in the appropriate source code directory and named so as not to conflict with existing devices. Normally, definitions for things like device registers are placed in a separate file in the same directory. For example, the ‘‘dh’’ device driver is named ‘‘dh.c’’ and its associated include file is named ‘‘dhreg.h’’.

Once the source for the device driver has been placed in a directory, the file ‘‘/sys/conf/files.machine’’, and possibly ‘‘/sys/conf/devices.machine’’ should be modified. The *files* files in the conf directory contain a line for each source or binary-only file in the system. Those files which are machine independent are located in ‘‘/sys/conf/files’’ while machine specific files are in ‘‘/sys/conf/files.machine’’. The ‘‘devices.machine’’ file is used to map device names to major block device numbers. If the device driver being added provides support for a new disk you will want to modify this file (the format is obvious).

The format of the *files* file has grown somewhat complex over time. Entries are normally of the form

vaxuba/foo.c **optional** foo **device-driver**

where the keyword *optional* indicates that to compile the ‘‘foo’’ driver into the system it must be specified in the configuration file. If instead the driver is specified as *standard*, the file will be loaded no matter what configuration is requested. This is not normally done with device drivers. The fact that the file is specified as a *device-driver* results, on the VAX, in the compilation including a **−i** option for the C optimizer. This is required when pointer references are made to memory locations in the VAX i/o address space.

Aside from including the driver in the *files* file, it must also be added to the device configuration tables. These are located in ‘‘/sys/vax/conf.c’’, or similar for machines other than the VAX. If you don’t understand what to add to this file, you should study an entry for an existing driver. Remember that the position in the block device table specifies what the major block device number is, this is needed in the ‘‘devices.machine’’ file if the device is a disk.

With the configuration information in place, your configuration file appropriately modified, and a system reconfigured and rebooted you should incorporate the shell commands needed to install the special files in the file system to the file ‘‘/dev/MAKEDEV’’ or ‘‘/dev/MAKEDEV.local’’. This is discussed in the document ‘‘Installing and Operating 4.2BSD on the VAX’’.

6.3. Autoconfiguration on the VAX

4.2BSD (and 4.1BSD) require all device drivers to conform to a set of rules which allow the system to:

1) support multiple UNIBUS and MASSBUS adapters,

2) support system configuration at boot time, and

3) manage resources so as not to crash when devices request resources which are unavailable.

In addition, devices such as the RK07 which require everyone else to get off the UNIBUS when they are running need cooperation from other DMA devices if they are to work. Since it is unlikely that you will be writing a device driver for a MASSBUS device, this section is devoted exclusively to describing the i/o system and autoconfiguration process as it applies to UNIBUS devices.

Each UNIBUS on a VAX has a set of resources:

- 496 map registers which are used to convert from the 18 bit UNIBUS addresses into the much larger VAX address space.
- Some number of buffered data paths (3 on an 11/750, 15 on an 11/780, 0 on an 11/730) which are used by high speed devices to transfer data using fewer bus cycles.

There is a structure of type *struct uba_hd* in the system per UNIBUS adapter used to manage these resources. This structure also contains a linked list where devices waiting for resources to complete DMA UNIBUS activity have requests waiting.

There are three central structures in the writing of drivers for UNIBUS controllers; devices which do not do DMA i/o can often use only two of these structures. The structures are *struct uba_ctlr*, the UNIBUS controller structure, *struct uba_device* the UNIBUS device structure, and *struct uba_driver*, the UNIBUS driver structure. The *uba_ctlr* and *uba_device* structures are in one-to-one correspondence with the definitions of controllers and devices in the system configuration. Each driver has a *struct uba_driver* structure specifying an internal interface to the rest of the system.

Thus a specification

```
controller sc0 at uba0 csr 0176700 vector upintr
```

would cause a *struct uba_ctlr* to be declared and initialized in the file *ioconf.c* for the system configured from this description. Similarly specifying

```
disk up0 at sc0 drive 0
```

would declare a related *uba_device* in the same file. The *up.c* driver which implements this driver specifies in its declarations:

```
int     upprobe(), upslave(), upattach(), updgo(), upintr();
struct  uba_ctlr *upminfo[NSC];
struct  uba_device *updinfo[NUP];
u_short upstd[] = { 0776700, 0774400, 0776300, 0 };
struct  uba_driver scdriver =
    { upprobe, upslave, upattach, updgo, upstd, "up", updinfo, "sc", upminfo };
```

initializing the *uba_driver* structure. The driver will support some number of controllers named *sc0*, *sc1*, etc, and some number of drives named *up0*, *up1*, etc. where the drives may be on any of the controllers (that is there is a single linear name space for devices, separate from the controllers.)

We now explain the fields in the various structures. It may help to look at a copy of *vaxuba/ubareg.h*, *h/ubavar.h* and drivers such as *up.c* and *dz.c* while reading the descriptions of the various structure fields.

uba_driver structure

One of these structures exists per driver. It is initialized in the driver and contains functions used by the configuration program and by the UNIBUS resource routines. The fields of the structure are:

ud_probe

A routine which is given a *caddr_t* address as argument and should cause an interrupt on the device whose control-status register is at that address in virtual memory. It may be the case that the device does not exist, so the probe routine should use delays (via the DELAY(n) macro which delays for *n* microseconds) rather than waiting for specific events to occur. The routine must **not** declare its argument as a *register* parameter, but **must** declare

```
register int br, cvec;
```

as local variables. At boot time the system takes special measures that these variables are ‘‘value-result’’ parameters. The *br* is the IPL of the device when it interrupts, and the *cvec* is the interrupt vector address on the UNIBUS. These registers are actually filled in in the interrupt handler when an interrupt occurs.

As an example, here is the *up.c* probe routine:

```
upprobe(reg)
        caddr_t reg;
{
        register int br, cvec;

#ifdef lint
        br = 0; cvec = br; br = cvec;
#endif
        ((struct updevice *)reg)->upcs1 = UP_IE|UP_RDY;
        DELAY(10);
        ((struct updevice *)reg)->upcs1 = 0;
        return (sizeof (struct updevice));
}
```

The definitions for *lint* serve to indicate to it that the *br* and *cvec* variables are value-result. The statements here interrupt enable the device and write the ready bit UP_RDY. The 10 microsecond delay insures that the interrupt enable will not be canceled before the interrupt can be posted. The return of ‘‘sizeof (struct updevice)’’ here indicates that the probe routine is satisfied that the device is present (the value returned is not currently used, but future plans dictate you should return the amount of space in the device’s register bank). A probe routine may use the function ‘‘badaddr’’ to see if certain other addresses are accessible on the UNIBUS (without generating a machine check), or look at the contents of locations where certain registers should be. If the registers contents are not acceptable or the addresses don’t respond, the probe routine can return 0 and the device will not be considered to be there.

One other thing to note is that the action of different VAXen when illegal addresses are accessed on the UNIBUS may differ. Some of the machines may generate machine checks and some may cause UNIBUS errors. Such considerations are handled by the configuration program and the driver writer need not be concerned with them.

It is also possible to write a very simple probe routine for a one-of-a-kind device if probing is difficult or impossible. Such a routine would include statements of the form:

```
br = 0x15;
cvec = 0200;
```

for instance, to declare that the device ran at UNIBUS br5 and interrupted through vector 0200 on the UNIBUS. The current UDA-50 driver does something similar to this because the device is so difficult to force an interrupt on that it hardly seems worthwhile.

ud_slave

This routine is called with a *uba_device* structure (yet to be described) and the address of the device controller. It should determine whether a particular slave device of a controller is present, returning 1 if it is and 0 if it is not. As an example here is the slave routine for *up.c*.

```
upslave(ui, reg)
        struct uba_device *ui;
        caddr_t reg;
{
        register struct updevice *upaddr = (struct updevice *)reg;

        upaddr->upcs1 = 0;              /* conservative */
        upaddr->upcs2 = ui->ui_slave;
        if (upaddr->upcs2&UPCS2_NED) {
                upaddr->upcs1 = UP_DCLR|UP_GO;
                return (0);
        }
        return (1);
}
```

Here the code fetches the slave (disk unit) number from the *ui_slave* field of the *uba_device* structure, and sees if the controller responds that that is a non-existent driver (NED). If the drive a drive clear is issued to clean the state of the controller, and 0 is returned indicating that the slave is not there. Otherwise a 1 is returned.

ud_attach

The attach routine is called after the autoconfigure code and the driver concur that a peripheral exists attached to a controller. This is the routine where internal driver state about the peripheral can be initialized. Here is the *attach* routine from the *up.c* driver:

```
upattach(ui)
        register struct uba_device *ui;
{
        register struct updevice *upaddr;

        if (upwstart == 0) {
                timeout(upwatch, (caddr_t)0, hz);
                upwstart++;
        }
        if (ui->ui_dk >= 0)
                dk_mspw[ui->ui_dk] = .0000020345;
        upip[ui->ui_ctlr][ui->ui_slave] = ui;
        up_softc[ui->ui_ctlr].sc_ndrive++;
          ui->ui_type = upmaptype(ui);
}
```

The attach routine here performs a number of functions. The first time any drive is attached to the controller it starts the timeout routine which watches the disk drives to make sure that interrupts aren’t lost. It also initializes, for devices which have been assigned *iostat* numbers (when ui->ui_dk >= 0), the transfer rate of the device in the array *dk_mspw*, the fraction of a second it takes to transfer 16 bit word. It then initializes an inverting pointer in the array *upip* which will be used later to determine, for a particular *up* controller and slave number, the corresponding *uba_device*. It increments the count of the number of devices on this controller, so that search commands can later be avoided if the count is exactly 1. It then attempts to decipher the actual type of drive attached to the controller in a controller-specific way. On the EMULEX SC-21 it may ask for the number of tracks on the device and use this to decide what the drive type is. The drive type is used to setup disk partition mapping tables and other device specific information.

ud_dgo

Is the routine which is called by the UNIBUS resource management routines when an operation is ready to be started (because the required resources have been allocated). The routine in *up.c* is:

```
updgo(um)
	struct uba_ctlr *um;
{
	register struct updevice *upaddr = (struct updevice *)um->um_addr;

	upaddr->upba = um->um_ubinfo;
	upaddr->upcs1 = um->um_cmd|((um->um_ubinfo>>8)&0x300);
}
```

This routine uses the field *um_ubinfo* of the *uba_ctlr* structure which is where the UNIBUS routines store the UNIBUS map allocation information. In particluar, the low 18 bits of this word give the UNIBUS address assigned to the transfer. The assignment to *upba* in the go routine places the low 16 bits of the UNIBUS address in the disk UNIBUS address register. The next assignment places the disk operation command and the extended (high 2) address bits in the device control-status register, starting the i/o operation. The field *um_cmd* was initialized with the command to be stuffed here in the driver code itself before the call to the *ubago* routine which eventually resulted in the call to *updgo*.

ud_addr

Are the conventional addresses for the device control registers in UNIBUS space. This information is used by the system to look for instances of the device supported by the driver. When the system probes for the device it first checks for a control-status register located at the address indicated in the configuration file (if supplied), then uses the list of conventional addresses pointed to be *ud_addr*.

ud_dname

Is the name of a *device* supported by this controller; thus the disks on a SC-21 controller are called *up0*, *up1*, etc. That is because this field contains *up*.

ud_dinfo

Is an array of back pointers to the *uba_device* structures for each device attached to the controller. Each driver defines a set of controllers and a set of devices. The device address space is always one-dimensional, so that the presence of extra controllers may be masked away (e.g. by pattern matching) to take advantage of hardware redundancy. This field is filled in by the configuration program, and used by the driver.

ud_mname

The name of a controller, e.g. *sc* for the *up.c* driver. The first SC-21 is called *sc0*, etc.

ud_minfo

The backpointer array to the structures for the controllers.

ud_xclu

If non-zero specifies that the controller requires exclusive use of the UNIBUS when it is running. This is non-zero currently only for the RK611 controller for the RK07 disks to map around a hardware problem. It could also be used if 6250bpi tape drives are to be used on the UNIBUS to insure that they get the bandwidth that they need (basically the whole bus).

uba_ctlr structure

One of these structures exists per-controller. The fields link the controller to its UNIBUS adapter and contain the state information about the devices on the controller. The fields are:

um_driver

A pointer to the *struct uba_device* for this driver, which has fields as defined above.

um_ctlr

The controller number for this controller, e.g. the 0 in *sc0*.

um_alive

Set to 1 if the controller is considered alive; currently, always set for any structure encountered during normal operation. That is, the driver will have a handle on a *uba_ctlr* structure only if the configuration routines set this field to a 1 and entered it into the driver tables.

um_intr
The interrupt vector routines for this device. These are generated by *config* and this field is initialized in the *ioconf.c* file.

um_hd
A back-pointer to the UNIBUS adapter to which this controller is attached.

um_cmd
A place for the driver to store the command which is to be given to the device before calling the routine *ubago* with the devices *uba_device* structure. This information is then retrieved when the device go routine is called and stuffed in the device control status register to start the i/o operation.

um_ubinfo
Information about the UNIBUS resources allocated to the device. This is normally only used in device driver go routine (as *updgo* above) and occasionally in exceptional condition handling such as ECC correction.

um_tab
This buffer structure is a place where the driver hangs the device structures which are ready to transfer. Each driver allocates a buf structure for each device (e.g. *updtab* in the *up.c* driver) for this purpose. You can think of this structure as a device-control-block, and the buf structures linked to it as the unit-control-blocks. The code for dealing with this structure is stylized; see the *rk.c* or *up.c* driver for the details. If the *ubago* routine is to be used, the structure attached to this *buf* structure must be:

- A chain of *buf* structures for each waiting device on this controller.
- On each waiting *buf* structure another *buf* structure which is the one containing the parameters of the i/o operation.

uba_device structure

One of these structures exist for each device attached to a UNIBUS controller. Devices which are not attached to controllers or which perform no buffered data path DMA i/o may have only a device structure. Thus *dz* and *dh* devices have only *uba_device* structures. The fields are:

ui_driver
A pointer to the *struct uba_driver* structure for this device type.

ui_unit
The unit number of this device, e.g. 0 in **up0**, or 1 in **dh1**.

ui_ctlr
The number of the controller on which this device is attached, or −1 if this device is not on a controller.

ui_ubanum
The number of the UNIBUS on which this device is attached.

ui_slave
The slave number of this device on the controller which it is attached to, or −1 if the device is not a slave. Thus a disk which was unit 2 on a SC-21 would have *ui_slave* 2; it might or might not be *up2*, that depends on the system configuration specification.

ui_intr
The interrupt vector entries for this device, copied into the UNIBUS interrupt vector at boot time. The values of these fields are filled in by *config* to small code segments which it generates in the file *ubglue.s*.

ui_addr
The control-status register address of this device.

ui_dk
The iostat number assigned to this device. Numbers are assigned to disks only, and are small positive integers which index the various *dk_** arrays in <*sys/dk.h*>.

ui_flags

The optional ‘‘flags *xxx*’’ parameter from the configuration specification was copied to this field, to be interpreted by the driver. If *flags* was not specified, then this field will contain a 0.

ui_alive

The device is really there. Presently set to 1 when a device is determined to be alive, and left 1.

ui_type

The device type, to be used by the driver internally.

ui_physaddr

The physical memory address of the device control-status register. This is used in the device dump routines typically.

ui_mi

A *struct uba_ctlr* pointer to the controller (if any) on which this device resides.

ui_hd

A *struct uba_hd* pointer to the UNIBUS on which this device resides.

UNIBUS resource management routines

UNIBUS drivers are supported by a collection of utility routines which manage UNIBUS resources. If a driver attempts to bypass the UNIBUS routines, other drivers may not operate properly. The major routines are: *uballoc* to allocate UNIBUS resources, *ubarelse* to release previously allocated resources, and *ubago* to initiate DMA. When allocating UNIBUS resources you may request that you

NEEDBDP

if you need a buffered data path,

HAVEBDP

if you already have a buffered data path and just want new mapping registers (and access to the UNIBUS), and

CANTWAIT

if you are calling (potentially) from interrupt level

If the presentation here does not answer all the questions you may have, consult the file /sys/vaxuba/uba.c

Autoconfiguration requirements

Basically all you have to do is write a *ud_probe* and a *ud_attach* routine for the controller. It suffices to have a *ud_probe* routine which just initializes *br* and *cvec*, and a *ud_attach* routine which does nothing. Making the device fully configurable requires, of course, more work, but is worth it if you expect the device to be in common usage and want to share it with others.

If you managed to create all the needed hooks, then make sure you include the necessary header files; the ones included by *vaxuba/ct.c* are nearly minimal. Order is important here, don’t be surprised at undefined structure complaints if you order the includes wrongly. Finally if you get the device configured in, you can try bootstrapping and see if configuration messages print out about your device. It is a good idea to have some messages in the probe routine so that you can see that you are getting called and what is going on. If you do not get called, then you probably have the control-status register address wrong in your system configuration. The autoconfigure code notices that the device doesn’t exist in this case and you will never get called.

Assuming that your probe routine works and you manage to generate an interrupt, then you are basically back to where you would have been under older versions of UNIX. Just be sure to use the *ui_ctlr* field of the *uba_device* structures to address the device; compiling in funny constants will make your driver only work on the CPU type you have (780, 750, or 730).

Other bad things that might happen while you are setting up the configuration stuff:

- You get ‘‘nexus zero vector’’ errors from the system. This will happen if you cause a device to interrupt, but take away the interrupt enable so fast that the UNIBUS adapter cancels the interrupt and confuses the processor. The best thing to do it to put a modest delay in the probe code between the instructions which should cause and interrupt and the clearing of the interrupt enable. (You should clear interrupt enable before you leave the probe routine so the device doesn't interrupt more and confuse the system while it is configuring other devices.)

- The device refuses to interrupt or interrupts with a ''zero vector''. This typically indicates a problem with the hardware or, for devices which emulate other devices, that the emulation is incomplete. Devices may fail to present interrupt vectors because they have configuration switches set wrong, or because they are being accessed in inappropriate ways. Incomplete emulation can cause ''maintenance mode'' features to not work properly, and these features are often needed to force device interrupts.

6.4. Adding non-standard system facilities

This section considers the work needed to augment *config*'s data base files for non-standard system facilities.

As far as *config* is concerned non-standard facilities may fall into two categories. *Config* understands that certain files are used especially for kernel profiling. These files are indicated in the *files* files with a *profiling-routine* keyword. For example, the current profiling subroutines are sequestered off in a separate file with the following entry:

sys/subr_mcount.c **optional profiling-routine**

The *profiling-routine* keyword forces *config* to not compile the source file with the **−pg** option.

The second keyword which can be of use is the *config-dependent* keyword. This causes *config* to compile the indicated module with the global configuration parameters. This allows certain modules, such as *machdep.c* to size system data structures based on the maximum number of users configured for the system.

APPENDIX A. CONFIGURATION FILE GRAMMAR

The following grammar is a compressed form of the actual *yacc* (1) grammar used by *config* to parse configuration files. Terminal symbols are shown all in upper case, literals are emboldened; optional clauses are enclosed in brackets, ‘‘[’’ and ‘‘]’’; zero or more instantiations are denoted with ‘‘*’’.

Configuration ::= [Spec **;**]*

Spec ::= Config_spec
| Device_spec
| **trace**
| /* lambda */

/* configuration specifications */

Config_spec ::= **machine** ID
| **cpu** ID
| **options** Opt_list
| **ident** ID
| System_spec
| **timezone** [**−**] NUMBER [**dst** [NUMBER]]
| **timezone** [**−**] FPNUMBER [**dst** [NUMBER]]
| **maxusers** NUMBER

/* system configuration specifications */

System_spec ::= **config** ID System_parameter [System_parameter]*

System_parameter ::= swap_spec | root_spec | dump_spec | arg_spec

swap_spec ::= **swap** [**on**] swap_dev [**and** swap_dev]*

swap_dev ::= dev_spec [**size** NUMBER]

root_spec ::= **root** [**on**] dev_spec

dump_spec ::= **dumps** [**on**] dev_spec

arg_spec ::= **args** [**on**] dev_spec

dev_spec ::= dev_name | major_minor

major_minor ::= **major** NUMBER **minor** NUMBER

dev_name ::= ID [NUMBER [ID]]

/* option specifications */

Opt_list ::= Option [**,** Option]*

Option ::= ID [**=** Opt_value]

Opt_value ::= ID | NUMBER

/* device specifications */

Device_spec ::= **device** Dev_name Dev_info Int_spec
    | **master** Dev_name Dev_info
    | **disk** Dev_name Dev_info
    | **tape** Dev_name Dev_info
    | **controller** Dev_name Dev_info [Int_spec]
    | **pseudo-device** Dev [NUMBER]

Dev_name ::= Dev NUMBER

Dev ::= **uba** | **mba** | ID

Dev_info ::= Con_info [Info]*

Con_info ::= **at** Dev NUMBER
    | **at nexus** NUMBER

Info ::= **csr** NUMBER
    | **drive** NUMBER
    | **slave** NUMBER
    | **flags** NUMBER

Int_spec ::= **vector** ID [ID]*
    | **priority** NUMBER

Lexical Conventions

The terminal symbols are loosely defined as:

ID

One or more alphabetics, either upper or lower case, and underscore, ‘‘_’’.

NUMBER

Approximately the C language specification for an integer number. That is, a leading ‘‘0x’’ indicates a hexadecimal value, a leading ‘‘0’’ indicates an octal value, otherwise the number is expected to be a decimal value. Hexadecimal numbers may use either upper or lower case alphabetics.

FPNUMBER

A floating point number without exponent. That is a number of the form ‘‘nnn.ddd’’, where the fractional component is optional.

In special instances a question mark, ‘‘?’’, can be substituted for a ‘‘NUMBER’’ token. This is used to effect wildcarding in device interconnection specifications.

Comments in configuration files are indicated by a ‘‘#’’ character at the beginning of the line; the remainder of the line is discarded.

A specification is interpreted as a continuation of the previous line if the first character of the line is tab.

APPENDIX B. RULES FOR DEFAULTING SYSTEM DEVICES

When *config* processes a ‘‘config’’ rule which does not fully specify the location of the root file system, paging area(s), device for system dumps, and device for argument list processing it applies a set of rules to define those values left unspecified. The following list of rules are used in defaulting system devices.

1) If a root device is not specified, the swap specification must indicate a ‘‘generic’’ system is to be built.
2) If the root device does not specify a unit number, it defaults to unit 0.
3) If the root device does not include a partition specification, it defaults to the ‘‘a’’ partition.
4) If no swap area is specified, it defaults to the ‘‘b’’ partition of the root device.
5) If no device is specified for processing argument lists, the first swap partition is selected.
6) If no device is chosen for system dumps, the first swap partition is selected (see below to find out where dumps are placed within the partition).

The following table summarizes the default partitions selected when a device specification is incomplete, e.g. ‘‘hp0’’.

Type	Partition
root	‘‘a’’
swap	‘‘b’’
args	‘‘b’’
dumps	‘‘b’’

Multiple swap/paging areas

When multiple swap partitions are specified, the system treats the first specified as a ‘‘primary’’ swap area which is always used. The remaining partitions are then interleaved into the paging system at the time a *swapon*(2) system call is made. This is normally done at boot time with a call to *swapon*(8) from the /etc/rc file.

System dumps

System dumps are automatically taken after a system crash, provided the device driver for the ‘‘dumps’’ device supports this. The dump contains the contents of memory, but not the swap areas. Normally the dump device is a disk in which case the information is copied to a location near the back of the partition. The dump is placed in the back of the partition because the primary swap and dump device are commonly the same device and this allows the system to be rebooted without immediately overwriting the saved information. When a dump has occurred, the system variable *dumpsize* is set to a non-zero value indicating the size (in bytes) of the dump. The *savecore* (8) program then copies the information from the dump partition to a file in a ‘‘crash’’ directory and also makes a copy of the system which was running at the time of the crash (usually ‘‘/vmunix’’). The offset to the system dump is defined in the system variable *dumplo* (a sector offset from the front of the dump partition). The *savecore* program operates by reading the contents of *dumplo*, *dumpdev*, and *dumpmagic* from /dev/kmem, then comparing the value of *dumpmagic* read from /dev/kmem to that located in corresponding location in the dump area of the dump partition. If a match is found, *savecore* assumes a crash occurred and reads *dumpsize* from the dump area of the dump partition. This value is then used in copying the system dump. Refer to *savecore* (8) for more information about its operation.

The value *dumplo* is calculated to be

dumpdev-size − DUMPDEV

where *dumpdev-size* is the size of the disk partition where system dumps are to be placed, and DUMPDEV is 10 Megabytes. If the disk partition is not large enough to hold a 10 Megabyte dump, *dumplo* is set to 0 (the front of the partition). For sites with more than 10 Megabytes of memory the definition of DUMPDEV in /sys/vax/autoconf.c will have to be changed.

APPENDIX C. SAMPLE CONFIGURATION FILES

The following configuration files are developed in section 5; they are included here for completeness.

```
#
# ANSEL VAX (a picture perfect machine)
#
machine         vax
cpu             VAX780
timezone        8 dst
ident           ANSEL
maxusers        40

config          vmunix          root on hp0
config          hpvmunix        root on hp0 swap on hp0 and hp2
config          genvmunix       swap generic

controller      mba0            at nexus ?
disk            hp0             at mba? disk ?
disk            hp1             at mba? disk ?
controller      mba1            at nexus ?
disk            hp2             at mba? disk ?
disk            hp3             at mba? disk ?
controller      uba0            at nexus ?
controller      tm0             at uba? csr 0172520     vector tmintr
tape            te0             at tm0 drive 0
tape            te1             at tm0 drive 1
device          dh0             at uba? csr 0160020     vector dhrint dhxint
device          dm0             at uba? csr 0170500     vector dmintr
device          dh1             at uba? csr 0160040     vector dhrint dhxint
device          dh2             at uba? csr 0160060     vector dhrint dhxint
```

```
#
# UCBVAX - Gateway to the world
#
machine         vax
cpu             "VAX780"
cpu             "VAX750"
ident           UCBVAX
timezone        8 dst
maxusers        32
options         INET

config          vmunix          root on hp swap on hp and rk0 and rk1
config          upvmunix        root on up
config          hkvmunix        root on hk swap on rk0 and rk1

controller      mba0            at nexus ?
controller      uba0            at nexus ?
disk            hp0             at mba? drive 0
disk            hp1             at mba? drive 1
controller      sc0             at uba? csr 0176700     vector upintr
disk            up0             at sc0 drive 0
disk            up1             at sc0 drive 1
controller      hk0             at uba? csr 0177440     vector rkintr
disk            rk0             at hk0 drive 0
disk            rk1             at hk0 drive 1
pseudo-device   inet
pseudo-device   pty
# software loopback device for testing
pseudo-device   loop
pseudo-device   imp
device          acc0            at uba? csr 0167600     vector accrint accxint
pseudo-device   ether
device          ec0             at uba? csr 0164330     vector ecrint eccollide ecxint
device          il0             at uba? csr 0164000     vector ilrint ilcint
```

APPENDIX D. VAX KERNEL DATA STRUCTURE SIZING RULES

Certain system data structures are sized at compile time according to the maximum number of simultaneous users expected, while others are calculated at boot time based on the physical resources present; e.g. memory. This appendix lists both sets of rules and also includes some hints on changing built-in limitations on certain data structures.

Compile time rules

The file */sys/conf/param.c* contains the definitions of almost all data structures sized at compile time. This file is copied into the directory of each configured system to allow configuration-dependent rules and values to be maintained. The rules implied by its contents are summarized below (here MAXUSERS refers to the value defined in the configuration file in the ‘‘maxusers’’ rule).

nproc

The maximum number of processes which may be running at any time. It is defined to be 20 + 8 * MAXUSERS and referred to in other calculations as NPROC.

ntext

The maximum number of active shared text segments. Defined as 24 + MAXUSERS + NETSLOP, where NETSLOP is 20 when the Internet protocols are configured in the system and 0 otherwise. The added size for supporting the network is to take into account the numerous server processes which are likely to exist.

ninode

The maximum number of files in the file system which may be active at any time. This includes files in use by users, as well as directory files being read or written by the system and files associated with bound sockets in the UNIX ipc domain. This is defined as (NPROC + 16 + MAXUSERS) + 32.

nfile

The number of ‘‘file table’’ structures. One file table structure is used for each open, unshared, file descriptor. Multiple file descriptors may reference a single file table entry when they are created through a *dup* call, or as the result of a *fork*. This is defined to be

16 * (NPROC + 16 + MAXUSERS) / 10 + 32 + 2 * NETSLOP

where NETSLOP is defined as for **ntext**.

ncallout

The number of ‘‘callout’’ structures. One callout structure is used per internal system event handled with a timeout. Timeouts are used for terminal delays, watchdog routines in device drivers, protocol timeout processing, etc. This is defined as 16 + NPROC.

nclist

The number of ‘‘c-list’’ structures. C-list structures are used in terminal i/o. This is defined as 100 + 16 * MAXUSERS.

nmbclusters

The maximum number of pages which may be allocated by the network. This is defined as 256 (a quarter megabyte of memory) in /sys/h/mbuf.h. In practice, the network rarely uses this much memory. It starts off by allocating 64 kilobytes of memory, then requesting more as required. This value represents an upper bound.

nquota

The number of ‘‘quota’’ structures allocated. Quota structures are present only when disc quotas are configured in the system. One quota structure is kept per user. This is defined to be (MAXUSERS * 9) / 7 + 3.

ndquot

The number of ‘‘dquot’’ structures allocated. Dquot structures are present only when disc quotas are configured in the system. One dquot structure is required per user, per active file system quota. That is, when a user manipulates a file on a file system on which quotas are enabled, the information regarding the user's quotas on that file system must be in-core. This information is cached, so that not all information must be present in-core all the time. This is defined as (MAXUSERS * NMOUNT) / 4 + NPROC, where NMOUNT is the maximum number of mountable file systems.

In addition to the above values, the system page tables (used to map virtual memory in the kernel's address space) are sized at compile time by the SYSPTSIZE definition in the file /sys/vax/param.h. This is defined to be 20 + MAXUSERS pages of page tables. Its definition affects the size of many data structures allocated at boot time because it constrains the amount of virtual memory which may be addressed by the running system. This is often the limiting factor in the size of the buffer cache.

Run-time calculations

The most important data structures sized at run-time are those used in the buffer cache. Allocation is done by swiping physical memory (and the associated virtual memory) immediately after the system has been started up; look in the file /sys/vax/machdep.c. The amount of physical memory which may be allocated to the buffer cache is constrained by the size of the system page tables, among other things. While the system may calculate a large amount of memory to be allocated to the buffer cache, if the system page table is too small to map this physical memory into the virtual address space of the system, only as much as can be mapped will be used.

The buffer cache is comprised of a number of ''buffer headers'' and a pool of pages attached to these headers. Buffer headers are divided into two categories: those used for swapping and paging, and those used for normal file i/o. The system tries to allocate 10% of available physical memory for the buffer cache (where *available* does not count that space occupied by the system's text and data segments). If this results in fewer than 16 pages of memory allocated, then 16 pages are allocated. This value is kept in the initialized variable *bufpages* so that it may be patched in the binary image (to allow tuning without recompiling the system). A sufficient number of file i/o buffer headers are then allocated to allow each to hold 2 pages each, and half as many swap i/o buffer headers are then allocated. The number of swap i/o buffer headers is constrained to be no more than 256.

System size limitations

As distributed, the sum of the virtual sizes of the core-resident processes is limited to 64M bytes. The size of the text, and data segments of a single process are currently limited to 6M bytes each, and the stack segment size is limited to 512K bytes as a soft, user-changeable limit, and may be increased to 6M with the *setrlimit*(2) system call. If these are insufficient, they can be increased by changing the constants MAXTSIZ, MAXDSIZ and MAXSSIZ in the file /sys/vax/vmparam.h, while changing the definitions in /sys/h/dmap.h and /sys/h/text.h. You must be careful in doing this that you have adequate paging space. As normally configured , the system has only 16M bytes per paging area. The best way to get more space is to provide multiple, thereby interleaved, paging areas.

To increase the amount of resident virtual space possible, you can alter the constant USRPTSIZE (in /sys/vax/vmparam.h). To allow 128 megabytes of resident virtual space one would change the 8 to a 16.

Because the file system block numbers are stored in page table *pg_blkno* entries, the maximum size of a file system is limited to 2ˆ19 1024 byte blocks. Thus no file system can be larger than 512M bytes.

The count of mountable file systems is limited to 15. This should be sufficient. If you have many disks it makes sense to make some of them single file systems, and the paging areas don't count in this total. To increase this it will be necessary to change the core-map /sys/h/cmap.h since there is a 4 bit field used here. The size of the core-map will then expand to 16 bytes per 1024 byte page. (Don't forget to change MSWAPX and NMOUNT in /sys/h/param.h also.)

The maximum value NOFILE (open files per process limit) can be raised to is 30 because of a bit field in the page table entry in /sys/machine/pte.h.