Building 4.4BSD Kernels with Config

Samuel J. Leffler and Michael J. Karels

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

ABSTRACT

This document describes the use of *config* (8) to configure and create bootable 4.4BSD 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). Other configuration options are also listed.

Revised July 5, 1993

1. INTRODUCTION

Config is a tool used in building 4.4BSD system images (the UNIX kernel). 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.

2.1. 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 4.4BSD 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.

2.2. 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 8600, VAX-11/780, VAX-11/750, VAX-11/730 or MicroVAX II. (Other VAX cpu types, including the 8650, 785 and 725, are configured using the cpu designation for compatible machines introduced earlier.) Specifying more than one cpu type implies that the system should be configured to run on any of the cpu's specified. For some types of machines this is not possible and *config* will print a diagnostic indicating such.

2.3. 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.

2.4. 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.

2.5. 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.

2.6. 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.

---

† DEC, VAX, UNIBUS, MASSBUS and MicroVAX are trademarks of Digital Equipment Corporation.

2.7. 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.

2.8. Pseudo devices

Several system facilities are configured in a manner like that used for hardware devices although they are not associated with specific hardware. These system options are configured as *pseudo-devices*. Some pseudo devices allow an optional parameter that sets the limit on the number of instances of the device that are active simultaneously.

2.9. System options

Other than the mandatory pieces of information described above, it is also possible to include various optional system facilities or to modify system behavior and/or limits. For example, 4.4BSD can be configured to support binary compatibility for programs built under 4.3BSD. 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 with *make* depend using *make*(1).
5) Compile and load the system with *make*.

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 4.4BSD distribution contains a number of configuration files for machines at Berkeley; one may be suitable or, in worst case, a copy of the generic configuration file may be edited.

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". Although it is not required that the system sources and configuration files reside in "/sys," the configuration and compilation procedure depends on the relative locations of directories within that hierarchy, as most of the system code and the files created by *config* use pathnames of the form "../". If the system files are not located in "/sys," it is desirable to make a symbolic link there for use in installation of other parts of the system that share files with the kernel.

When building the 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, *Makefile*.
- One file for each possible system image for this machine, *swapxxx.c*, where *xxx* is the name of the system image, 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, *ioconf.c*.
- An assembly language file of interrupt vectors which connect interrupts from the machine's external buses to the main system path for handling interrupts, and a file that contains counters and names for the interrupt vectors.

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 the modification of a system header file. The dependency rules are generated by a pass of the C preprocessor and reflect the global system options. This step must be repeated when the configuration file is changed and *config* is used to regenerate the system makefile.

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. As the system will generally use the disk from which it is loaded as the root filesystem, separate system images are only required to support different swap configurations.

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 *netstat*(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 modules 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’’ in the system compilation directory. 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 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* 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 attributed.

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 **VAX8600**, **VAX780**, **VAX750**, **VAX730** and **VAX630** (MicroVAX II). The 8650 is listed as an 8600, the 785 as a 780, and a 725 as a 730.

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. The following are major options are currently in use: COMPAT (include code for compatibility with 4.1BSD binaries), INET (Internet communication protocols), NS (Xerox NS communication protocols), and QUOTA (enable disk quotas). Other kernel options controlling system sizes and limits are listed in Appendix D; options for the network are found in Appendix E. 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.

makeoptions *optionlist*

Options that are used within the system makefile and evaluated by *make* are listed as *makeoptions*. Options are listed with their values with the form ‘‘makeoptions name=value,name2=value2.’’ The values must be enclosed in double quotes if they include numerals or begin with a dash.

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

Specifies the timezone used by the system. 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). This value is defined for use in conditional compilation, and is also used to locate an optional list of source files specific to this system.

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 and the number and location of paging devices. 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 device.

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 size is to be used for one or more swap areas (less than the full partition), 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 that 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:

* 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.

- 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 adapter),
- 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 ...
disk	hp0	**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.

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. *Drive* numbers may be wildcarded on MASSBUS devices, but not on disks on a UNIBUS controller. For controllers, the control status register must be given explicitly, as well the number of interrupt vectors 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	
disk	hp0	**at** mba0 **drive** ?	
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 **pty**, the pseudo terminal driver (where the optional *howmany* value indicates the number of pseudo terminals to configure, 32 default), and **loop**, the software loopback network pseudo-interface. Other pseudo devices for the network include **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
options         NS
```

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. The “INET” indicates that we plan to use the DARPA standard Internet protocols on this machine, and “NS” also includes support for Xerox NS protocols. Note that unlike 4.2BSD configuration files, the network protocol options do not require corresponding pseudo devices.

The system images and disks are configured 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 devices. Pseudo terminals are needed to allow users to log in across the network (remember the only hardwired terminal is the console). The software loopback device is used for on-machine communications. 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   pty
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 non-standard system facilities to 4.4BSD,
- how to add a device driver to 4.4BSD, and

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’’ (or other machine, as appropriate). 4.4BSD underwent 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.

6.2. Adding non-standard system facilities

This section considers the work needed to augment *config*’s data base files for non-standard system facilities. *Config* uses a set of files that list the source modules that may be required when building a system. The data bases are taken from the directory in which *config* is run, normally /sys/conf. Three such files may be used: *files*, *files*.machine, and *files*.ident. The first is common to all systems, the second contains files unique to a single machine type, and the third is an optional list of modules for use on a specific machine. This last file may override specifications in the first two. The format of the *files* file has grown somewhat complex over time. Entries are normally of the form

dir/source.c type option-list modifiers

for example,

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

The *type* is one of **standard** or **optional**. Files marked as standard are included in all system configurations. Optional file specifications include a list of one or more system options that together require the inclusion of this module. The options in the list may be either names of devices that may be in the configuration file, or the names of system options that may be defined. An optional file may be listed multiple times with different options; if all of the options for any of the entries are satisfied, the module is included.

If a file is specified as a *device-driver*, any special compilation options for device drivers will be invoked. On the VAX this results in the use of the **−i** option for the C optimizer. This is required when pointer references are made to memory locations in the VAX I/O address space.

Two other optional keywords modify the usage of the file. *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* not to 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.

6.3. Adding device drivers to 4.4BSD

The I/O system and *config* have been designed to easily allow new device support to be added. 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-protocol-independent, but network-related code
/sys/netinet	DARPA Internet code
/sys/netimp	IMP support code
/sys/netns	Xerox NS 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 C 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).

In addition to 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 device table specifies the major device number. The block major number 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 4.4BSD''.

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

Mkopt_list ::= Mkoption [**,** Mkoption]*

Mkoption ::= ID **=** Opt_value

/* 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 at 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 − *memsize*

where *dumpdev-size* is the size of the disk partition where system dumps are to be placed, and *memsize* is the size of physical memory. If the disk partition is not large enough to hold a full dump, *dumplo* is set to 0 (the start of the partition).

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
options         NS

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   pty
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. (Each copy normally depends on the copy in /sys/conf, and global modifications cause the file to be recopied unless the makefile is modified.) The rules implied by its contents are summarized below (here MAXUSERS refers to the value defined in the configuration file in the ‘‘maxusers’’ rule). Most limits are computed at compile time and stored in global variables for use by other modules; they may generally be patched in the system binary image before rebooting to test new values.

nproc

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

20 + 8 * MAXUSERS

ntext

The maximum number of active shared text segments. The constant is intended to allow for network servers and common commands that remain in the table. It is defined as

36 + MAXUSERS.

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. It 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

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, and currently each holds 60 characters. Their number is defined as

60 + 12 * 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 8 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

NINODE + (MAXUSERS * NMOUNT) / 4

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/vmparam.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, in which case a message is printed when the system configures at boot time.

Run-time calculations

The most important data structures sized at run-time are those used in the buffer cache. Allocation is done by allocating physical memory (and system 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 the first two megabytes and 5% of the remaining 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), or the default may be overridden with a configuration-file option. For example, the option **options BUFPAGES="3200"** causes 3200 pages (3.2M bytes) to be used by the buffer cache. A sufficient number of file I/O buffer headers are then allocated to allow each to hold 2 pages each. Each buffer maps 8K bytes. If the number of buffer pages is larger than can be mapped by the buffer headers, the number of pages is reduced. The number of buffer headers allocated is stored in the global variable *nbuf*, which may be patched before the system is booted. The system option **options NBUF="1000"** forces the allocation of 1000 buffer headers. Half as many swap I/O buffer headers as file I/O buffers are allocated, but no more than 256.

System size limitations

As distributed, the sum of the virtual sizes of the core-resident processes is limited to 256M bytes. The size of the text segment of a single process is currently limited to 6M bytes. It may be increased to no greater than the data segment size limit (see below) by redefining MAXTSIZ. This may be done with a configuration file option, e.g. **options MAXTSIZ="(10*1024*1024)"** to set the limit to 10 million bytes.

Other per-process limits discussed here may be changed with similar options with names given in parentheses. Soft, user-changeable limits are set to 512K bytes for stack (DFLSSIZ) and 6M bytes for the data segment (DFLDSIZ) by default; these may be increased up to the hard limit with the *setrlimit*(2) system call. The data and stack segment size hard limits are set by a system configuration option to one of 17M, 33M or 64M bytes. One of these sizes is chosen based on the definition of MAXDSIZ; with no option, the limit is 17M bytes; with an option **options MAXDSIZ="(32*1024*1024)"** (or any value between 17M and 33M), the limit is increased to 33M bytes, and values larger than 33M result in a limit of 64M bytes. You must be careful in doing this that you have adequate paging space. As normally configured , the system has 16M or 32M bytes per paging area, depending on disk size. The best way to get more space is to provide multiple, thereby interleaved, paging areas. Increasing the virtual memory limits results in interleaving of swap space in larger sections (from 500K bytes to 1M or 2M bytes).

By default, the virtual memory system allocates enough memory for system page tables mapping user page tables to allow 256 megabytes of simultaneous active virtual memory. That is, the sum of the virtual memory sizes of all (completely- or partially-) resident processes can not exceed this limit. If the limit is exceeded, some process(es) must be swapped out. To increase the amount of resident virtual space possible, you can alter the constant USRPTSIZE (in /sys/vax/vmparam.h). Each page of system page tables allows 8 megabytes of user virtual memory.

Because the file system block numbers are stored in page table *pg_blkno* entries, the maximum size of a file system is limited to 2ˆ24 1024 byte blocks. Thus no file system can be larger than 8 gigabytes.

The number of mountable file systems is set at 20 by the definition of NMOUNT in /sys/h/param.h. This should be sufficient; if not, the value can be increased up to 255. 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.

The limit to the number of files that a process may have open simultaneously is set to 64. This limit is set by the NOFILE definition in /sys/h/param.h. It may be increased arbitrarily, with the caveat that the user structure expands by 5 bytes for each file, and thus UPAGES (/sys/vax/machparam.h) must be increased accordingly.

The amount of physical memory is currently limited to 64 Mb by the size of the index fields in the core-map (/sys/h/cmap.h). The limit may be increased by following instructions in that file to enlarge those fields.

APPENDIX E. NETWORK CONFIGURATION OPTIONS

The network support in the kernel is self-configuring according to the protocol support options (INET and NS) and the network hardware discovered during autoconfiguration. There are several changes that may be made to customize network behavior due to local restrictions. Within the Internet protocol routines, the following options set in the system configuration file are supported:

GATEWAY
The machine is to be used as a gateway. This option currently makes only minor changes. First, the size of the network routing hash table is increased. Secondly, machines that have only a single hardware network interface will not forward IP packets; without this option, they will also refrain from sending any error indication to the source of unforwardable packets. Gateways with only a single interface are assumed to have missing or broken interfaces, and will return ICMP unreachable errors to hosts sending them packets to be forwarded.

TCP_COMPAT_42
This option forces the system to limit its initial TCP sequence numbers to positive numbers. Without this option, 4.4BSD systems may have problems with TCP connections to 4.2BSD systems that connect but never transfer data. The problem is a bug in the 4.2BSD TCP.

IPFORWARDING
Normally, 4.4BSD machines with multiple network interfaces will forward IP packets received that should be resent to another host. If the line ‘‘options IPFORWARDING="0"’’ is in the system configuration file, IP packet forwarding will be disabled.

IPSENDREDIRECTS
When forwarding IP packets, 4.4BSD IP will note when a packet is forwarded using the same interface on which it arrived. When this is noted, if the source machine is on the directly-attached network, an ICMP redirect is sent to the source host. If the packet was forwarded using a route to a host or to a subnet, a host redirect is sent, otherwise a network redirect is sent. The generation of redirects may be inhibited with the configuration option ‘‘options IPSENDREDIRECTS="0".’’

SUBNETSARELOCAL
TCP calculates a maximum segment size to use for each connection, and sends no datagrams larger than that size. This size will be no larger than that supported on the outgoing interface. Furthermore, if the destination is not on the local network, the size will be no larger than 576 bytes. For this test, other subnets of a directly-connected subnetted network are considered to be local unless the line ‘‘options SUBNETSARELOCAL="0"’’ is used in the system configuration file.

The following options are supported by the Xerox NS protocols:

NSIP
This option allows NS IDP datagrams to be encapsulated in Internet IP packets for transmission to a collaborating NSIP host. This may be used to pass IDP packets through IP-only link layer networks. See *nsip*(4P) for details.

THREEWAYSHAKE
The NS Sequenced Packet Protocol does not require a three-way handshake before considering a connection to be in the established state. (A three-way handshake consists of a connection request, an acknowledgement of the request along with a symmetrical opening indication, and then an acknowledgement of the reciprocal opening packet.) This option forces a three-way handshake before data may be transmitted on Sequenced Packet sockets.