Changes in the Kernel in 4.1bsd

May 10, 1981
Revised: September 1, 1981

Bill Joy

Computer Systems Research Group
University of California, Berkeley

This document summarizes the changes in the kernel between the November 1980 4.0bsd release and the and April 1981 4.1bsd distribution. The information is presented in both overall terms (e.g. organizational changes), and as specific comments about individual changed files. See the source code itself for more details.

The major changes fall in five categories:

1) Changes in the VAX 11/780 specific portions of the code, so that VAX 11/750's are supported also.

2) Changes in the organization of the code, so that more than one configuration of the system may be built from a single set of sources. Each system is described by a single file which includes parameters such as system size, devices on the machine, etc. All ''magic numbers'' such as device register addresses are collected in this single file.

3) Extensive changes in the device subsystem to allow multiple UNIBUS and MASSBUS adapters to be used, multiple instances of device controllers to exist without duplicating driver code, and to provide the capability of system configuration at boot time. The configuration capability is used to produce a generic system which runs on all supported hardware, and is used for distributions. Pattern matching in the configuration capability also allows hardware redundancy to be used to good effect.

4) Diagnostics of the system have been reworked to be in a standard and readable format; file system diagnostic refer to the file systems by name rather than device number. Device diagnostics refer to the devices by name, and print error messages including device registers decoded symbolically rather than simply in octal or hexadecimal. DEC standard bad sector forwarding has been added to the drivers for DEC disks.

5) Performance improvements, noticeably in the paging subsystem.

A number of enhancements and bug fixes have also been made.

Carrying over local software

The majority of local changes should carry over to the new system quite easily. It it necessary to create a configuration file for each machine from which a system will be built, but this is quite easy, and such files are designed to be usable without change in future releases of the system.

Locally written UNIBUS device drivers will need to be converted to work in the new system. The new functions needed of the device drivers are:

1) Forcing a device interrupt at bootstrap time, given a proposed device register address. This is used by the configuration program to decide if the device really exists.

2) If buffered data paths are to be used, the driver must use routines in the UNIBUS adapter subsystem which arranges for i/o requests to be queued when there are no resources available.

3) Drivers must not assume that only one instance of a device exists in the system, but must rather be parameterized and use the information provided by the bootstrap procedure to drive all available

devices.

Of course, it is not necessary to make a driver ‘‘fully supported’’ for it to be used. It suffices to handle 1) by pretending that the interrupt occurred, returning the (for a single system) known UNIBUS vector information, and assuming that the device exists on specific UNIBUS adapters. Drivers which use UNIBUS resources only statically or not all all need not be concerned with 2), and drivers can assume that there is only one instance of the supported device on the system, and just not work if more than one such device is really present.

In any case, more information about device driver changes is given in the last section of this document; also see *autoconf*(4) for information about the messages printed out by the configuration code at bootstrap time. Looking at the provided standard supported drivers for examples of code is also a good idea.

There is also a new interface for MASSBUS devices. Since all MASSBUS devices are already supported, there is no external documentation for writing new MASSBUS drivers at the present. If you have questions or intend to write a driver for a home-brew interface, you should read the MASSBUS and MASSBUS device driver code, which is amply commented. In any case, the MASSBUS interface is more stylized than the UNIBUS interface, and you may have to extend the functionality of the MASSBUS driver to handle radically different devices.

Organizational changes

On RK07 systems the source for the system lives in the root directory, since there is so little space. The system otherwise lives where it used to: the subdirectories of /usr/src/sys, with copies of the header files for the installed system in /usr/include/sys.

The system compilation procedure has been changes so that more than one set of binaries may be kept conveniently with a single copy of the source code. The system sources are kept in the directories **sys/sys** and **sys/dev** with the header files in **sys/h**. Source files which were previously kept in **sys/conf** are now in **sys/dev**, and no binaries are kept in any of these directories.

The directory **sys/conf** contains a number of files related to system configuration. For each machine to be configured, a single file is created in the **sys/conf** directory; thus files **ERNIE** and **BERT** might exist there. Each such file describes all the parameters of the machine to be used: the devices which are to be configured into the system, optional parts of the system to be included, as well as the timezone in which the machine lives and the maximum number of simultaneous active users; the last is used to scale system tables. The format of the configuration files is described in *config*(8).

Corresponding to each system to be configured there is a directory of **sys**, thus **sys/BERT** and **sys/ERNIE**. These directories are made with *mkdir* and then the **/etc/config** command is run, from the **sys/conf** directory, specifying **BERT** or **ERNIE** as argument. The configuration program processes the information in the configuration files, and produces, in the directory **../BERT** or **../ERNIE** respectively:

1) A set of header files, e.g. **dz.h**, which contain the number of devices and controllers to be available in the target system. These definitions force conditional compilation of drivers resulting in the inclusion or exclusion of driver code and the sizing of driver tables. This technique, based on compilation, is more powerful than a loader-based technique, since small sections of code may be easily conditionalized. Similarly, dynamic loading of device drivers is not needed, as only drivers which are needed are included in the resulting system.

2) A small assembly language vector interface, which turns the hardware generated UNIBUS interrupt sequences into C calls on the driver interrupt routines. This **ubglue.s** file glues the hardware interrupt sequence into the UNIX interface.

3) A table file **ioconf.c** which initializes tables to be used at bootstrap time by the system configuration routines. The configuration routines interpret the contents of the table and determine which devices are available on the system. They determine the vector addresses of UNIBUS devices by forcing the devices to interrupt. Pattern matching in the tables may be used to take advantage of hardware redundancy: the specifications need not completely constrain device placement, so the system can be built to bootstrap in several different configurations, locating the same devices on different interconnects by the fact that their unit numbers have not changes (for example). Thus two RP06 disks could

be specified as:

disk hp0 at mba? drive 0
disk hp1 at mba? drive 1

and then the disks could be cabled to any available MASSBUS adapter; the pattern matching in the configuration procedure would locate the drives. Similarly, a tape formatter on the same system could be specified:

master ht0 at mba? drive ?

and then placed anywhere on any MBA. Contrast this flexible specification with

disk hp0 at mba0 drive 0
disk hp1 at mba0 drive 1
master ht0 at mba1 drive 0

which is not reconfigurable. This latter specification corresponds to the previous UNIX capabilities, which did not allow tapes and disks on the same MASSBUS adapter.

4) Finally the *config* program constructs a *makefile* for the system which builds the drivers needed in the specified configuration, and includes system loading sequences for the different root and swap device configurations desired.

It is now easy to include ‘‘subsystems’’ optionally. This is done through the same mechanisms which causes conditional inclusion of device drivers. The file **conf/files** contains a palate of files which builds the system. Each line is either of the form:

filename standard

or

filename optional *xx*

where *xx* is the name of a device which requires the file, or a *pseudo-device*. To define a subsystem to be added to the kernel it suffices to add specifications to the **conf/files** file for the newly optional files and to then place a specification

pseudo-device *xx*

in the system configuration file. A line

options *XX*

may also be added to the configuration to have the symbol XX defined during compilation, for use in conditional compilations in the standard part of the system. Such conditional compilation is typically used to provide hooks in the standard part of the kernel to switch out to subsystem functions.

This completes the general description of organizational changes. We now describe the changes in the system, file by file.

Header files: sys/h and /usr/include/sys

General changes: device drivers now have header files in these directories, thus the ‘‘up’’ driver has a header file ‘‘upreg.h’’. This so the standalone code and the mainline UNIX code can share the common definitions.

The ‘‘.m’’ files of the previous distribution have been eliminated (with the sole exception of **pcb.m**); the magic numbers which were manually entered in these files are instead generated by a program from the definitions in the corresponding **.h** files; a number of header files thus no longer warn about correspondences that must be maintained.

The system tables are now described by pointers to their beginning and end and a count, rather than compiled in constants. This allows table sizes to be chosen at boot time (although the system currently does this only for the file system buffer cache), and makes programs such as **ps** and **w** not compile in these constants. Note, especially, that the symbols such as *proc* and *inode* are now memory locations containing the addresses of these structures rather than the base of the structures themselves. Programs which access these structures have been changed and use the variables *nproc* and *ninode* in core rather than the (now defunct) constants NPROC and NINODE.

buf.h Now declares three headers on which the in-core buffers are placed. Buffers which are locked in the buffer cache are placed on the first queue. Currently, only file system super blocks are locked in core, and to good effect: it is now possible to rebuild the super-block of the root file system with the system quiescent (without rebooting) if the block device is used. It is no longer necessary to take a buffer for the super-block of a file system and also make a copy of it at each sync; the same buffer can be reused and simply released: since it is locked it will remain in the buffer cache.

The other two queues implement the lru cache and the list of blocks which have been discarded. By having queues for both of these rather than using the end of a single queue, we achieve true fifo behavior for blocks which we consider ‘‘discarded’’; previously rather strange behavior resulted from pushing these blocks backwards on the front of the single queue. (In particular pipes would behave badly on idle systems under some circumstances.)

The number of pages paged is counted at pageout completion, as well as the pageout event count. A bug in the **physio** routine which caused physical transfers of more than 60000 bytes to sometimes fail to return an error indication has been fixed.

A flag has been added that marks a buffer as consisting only of a header and also one which marks a buffer being used for bad-sector processing.

callo.h Is now called **callout.h**, and the name of the structure is similarly changed to make it consistent with the other structures in the kernel. The structures are now linked together in linked lists, to prevent arcane situations previously possible where only half of the structures would be used, but the table space would be exhausted.

clock.h A botch in handling of leap years has been fixed. A macro is defined here to queue a software interrupt for handling most of the clock processing at an IPL lower than the clock IPL.

cmap.h This file, like a number of others, no longer warns that the size of the structure is known elsewhere; such dependencies are the concern of a C program and automated through makefile dependencies.

conf.h A *d_dump* entry has been added to the block device table, and is used as the system now normally does automatic dumps of core memory to disk after a crash. The field *d_tab* is now called *d_flags* and set to B_TAPE for tapes. For reasons not worth explaining here, there are no ‘‘tab’’ structures to sensibly use in initializing this field now, and in any case the only use of it was to tell which block devices were tapes.

dkbad.h Is a new file which defines the format of the bad sector forwarding information according to DEC standard 144.

dmap.h The constant DMMIN has been increased to 32 to allow upto 16k bytes to be paged to the paging devices in a clustered pageout.

filsys.h The two fields *s_fname* and *s_fpack* which were not implemented before were merged together (into a single 12 character field) which is called *s_fsmnt*. The system puts the ASCII path name where a file system is mounted (e.g. /usr) in this field and uses it in printing error messages on the console; (e.g. ‘‘/usr: file system full’’ rather than ‘‘no space on dev 0/6’’).

inline.h In order to reduce the number of conditional flags defined when compiling the system, the conditional flag FASTVAX, which was always defined, has been deleted. A conditional flag UNFAST, which is never defined, has been added to take its converse’s place.

inode.h The constant NINDEX has been reduced from 15 to 6. This limits the number of files which may be join()’ed into a multiplexor (*mpx*(2)) tree. You may have to increase this if you use the multiplexor extensively, but it saves a large amount of space in the kernel if you can use the smaller value, since NINDEX of 15 causes 40 bytes of extra unused space to be allocated to every inode.

map.h The **malloc.c** routines have been rewritten to check for table overflow and renamed map.c.

mba.h Is now split into **mbareg.h** and **mbavar.h**, the former contains the definitions of device register and is usable, e.g., in the standalone version of the system. The latter contains system variable related to the MASSBUS adapters.

mem.h Is a new file which contains information on the memory controller registers in the form of macros which make the several VAX processors seem very similar to the UNIX code. Note also that the system now uses interrupts from the memory system to force error logging since the previous technique (polling) works only on the 11/780.

mscp.h A new file which defines the DEC *Mass Storage Control Protocol* used by the UDA50 disk controller.

mtpr.h The register numbers are now given in hex, as in the DEC manuals; registers for all VAX processors are included.

msgbuf.h Defines the structure of the error message buffer, which is now kept in the last 1024 bytes of memory. This allows it to be preserved across system crashes and lets messages such as machine check reports be written conveniently into the error log.

nexus.h A new header file which defines the registers and constants related to the interconnect architecture of VAXen.

param.h No longer defines the large number of constants related to system sizing; a smaller number of rarely changed constants are given here. In particular, constants which were typically changed to affect the maximum number of supportable users are now controlled by the value given the **maxusers** keyword in the machine specification (as described in *config* (8)). The *config* program turns this specification into parameters to the **param.c** file which uses formulae to compute the values for the size of the process table, inode table, etc.

This file now includes the standard file <sys/types.h> to get system types rather than replicating the definitions from that file. It also defines a DELAY(n) macro which is used in device drivers to provide rougly *n* microseconds of delay. Finally the definition of UPAGES, the number of system control pages per-process has been increased from 6 to 8. This is partially due to the fact that there is now a red-zone page between the kernel stack and the kernel critical data in the *u.* area, but also because the kernel stack was precariously close to being too small before.

pcb.h Now includes definitions related to the use of AST's to implement user program profiling and rescheduling. Because AST's are now used, it is no longer necessary to take clock interrupts on the kernel stack; they now run on the interrupt stack where they belong. Also rescheduling processing is much cleaner, since the reschedule interrupts only go off when returning to user mode, not in the kernel where they have to be ignored (because UNIX cannot reschedule when running normally in the kernel.)

proc.h Now defines SOWEUPC, a new flag used to indicate that a profiling count should be generated when the (already posted) AST for this process goes off. Another new flag SSEQL indicates that the process has declared sequential paging behavior for its data space. Finally the field **p_maxrss** has been added, specifying the declared ‘‘memoryuse’’ limit in pages.

psl.h Has a bug fixed in the definitions of PSL_USERCLR. Now also declares PSL_USERSET and PSL_MBZ (must-be-zero).

system.h Defines the variables *hz*, *timezone* and *dstflag* replacing the old compile-time constants. No longer declares *msgbuf* as a variable (see **msgbuf.h**). Defines the *dumpdev* and *dumplo* variables which specify where dumps are to take place. No longer defines the debugging variables *printsw* and *coresw* which have been removed in favor of more local debugging variables. No longer defines the field *sy_nrarg* for the system call entry structures, since system calls never take register arguments on the VAX.

A variable **wantin** has been added which is set each time a process is woken up which wants to be swapped in. This is used so that the code in **swapoutR in vmsched.c** does not run with elevated priority.

trap.h Rearranges some codes previously used only internally so they would be contigous numerically. These are the finer machine traps which result in SIGILL and are made available to a signal handling process and defined in <signal.h>. Defines ASTFLT rather than RESCHED, since the VMS software interrupt which is used for VMS rescheduling never was appropriate for UNIX and is no longer used.

uba.h Has been split into **ubareg.h** and **ubavar.h**; see the description of device driver changes below.

user.h Contains definitions related to the new **#!** exec facility. The field *u.u_cfcode* has been renamed *u.u_code* since it is now used for purposes other than compatibility mode (presenting the more precise hardware reason for SIGILL and SIGFPE signals.)

vlimit.h Now defined LIM_MAXRSS for the ‘‘limit memoryuse’’ feature.

vm.h The **vm*.h** headers have been compressed into a more sensible set of files; the macros are all in **vmmac.h** (absorbing **vmclust.h** and **vmklust.h**), metering stuff is all in **vmmeter.h** (absorbing **vmmon.h** and **vmtotal.h**) and the parameters are all in **vmparam.h** (absorbing **vmtune.h**, most of the parameters of which are now adjusted at boot time in *setupclock* in **vmsched.c**.)

vmmeter.h The structure **vmmeter** now computes the number of pages paged in *v_pgpgin* and pages paged out *v_pgpgout*, as well as the number of pages freed because of the behavior of programs which have told the system they are sequential *v_seqfree*.

vmparam.h The values of MAXDSIZ and MAXSSIZ have increased due to the increase to DMMIN in **dmap.h**. The klustering constants have been changed: in-clustering is now in 4 page (4k byte) chunks, and out-clustering is up to 16k bytes. Sequential programs kluster in 8k bytes, and text segments kluster in 2k bytes. The gap for the window into sequential programs is currently (primitively) defined as a constant kere in KLSDIST.

vmsystm.h Defines a new variable *avefree30*, which computes the average memory like *avefree*, but averaged over a longer period of time. This is used to put more hysteresis into swapping, and keep the system from swapping immediately when memory drops low.

System files: sys/sys

A number of files in the system have had minor changes made to them to reduce the length of time the system runs with the interrupt priority level raised; in particular, the times when the IPL is high enough to block the clock have been severely limited, in hopes of providing better real-time response (eventually) and possibly being able to drive the 11/750 console cassette (soon) which has severe interrupt latency constraints due to poor hardware interface design.

acct.c The code was tightened by using a register variable. The *sysphys* routine was moved to **sys4.c** since it had no business being here.

alloc.c Prints error messages relating to file system problems using the name of the file system rather than the major/minor device number of the device. Some code which attempted to prevent ‘‘dups in free’’ after a reboot, but could not prevent this completely, has just been removed; the condition is not harmful in any case, as it is normal and fixed by *fsck*(8). The system now prints directly on a user’s terminal if that user causes a file system to run out of free space. The routines here also know how to deal with the fact that the superblocks are now kept locked in the buffer cache.

asm.sed No longer defines *spl1* which is now defunct; **spl7** is now VAX IPL 0x1f rather than 0x18, blocking most processor aborts device interrupts from the console storage device, and a number of other processor dependent interrupts. Deals with a strange feature of the optimizer which converts ‘‘$0’’ into a register which contains 0. Implements the *ffs* routine of **sig.c** in a much more efficient way (in just a couple of VAX instructions.) Beware, however: UNIX’s notion of *ffs* returns 1 for the low bit of a word, while the hardware *ffs* would return 0.

clock.c Now runs only that code which is absolutely necessary when the processor priority is very high, queueing a software interrupt at which priority the rest of the clock processing is done. The conditional (old and long unused) code which profiled the kernel in a static buffer has been removed. The option of fishing characters out of the *dz* and *dh* silo’s less often than every clock tick (1/hz) has been removed. Instead the silos are processed every clock tick if the system includes the berknet (bk) line discipline, or not at all (i.e. we take input interrupts) if ‘‘bk’’ is not included in the system.

The processing and watching of hung UNIBUS adapters has been moved from here to the UNIBUS routines. Automatic niceing of long-running (more than 10 minutes of user-state time) processes is now the default here, rather than being based on ‘‘#if ERNIE’’. A bug in the check for timeout table overflow which would cause the table to overrun without overflow being detected has been fixed. The timeout table is now implemented as a linked list, so that the entries can be conveniently discarded before calling the timeout routines. This prevents the anomalous case where only half the entries are used but the table fills up.

fio.c Has been changed to do the correct thing when special files or mounted file systems are closed: a flush is done at the last close and all blocks are invalidated. The standard ‘‘table full’’ routines are called when the file related tables fill up. These routines no longer pass **struct chan *** pointers down to called routines, passing, instead, the more universal **struct file *** pointers from which the **chan** pointers are easily derived.

iget.c Now uses the standard *tablefull* routine.

ioctl.c The last argument to *d_ioctl* routines when called is now always 0.

locore.s Has been extensively changed to accomodate the new configurable system, and to handle multiple UNIBUS and MASSBUS adapters. The code is now written using macros and the C preprocessor, improving readability. Complicated logic (such as the code to handle UNIBUS adapter errors) has been migrated to C code.

MASSBUS and UNIBUS adapters are no longer initialized or mapped here; this is the job of the configuration code in the system. The locore code distinguishes, in handling UNIBUS interrupts, from the machine being *cold* and not; when cold UNIBUS interrupts are handled so as to be suitable for determined device vectoring via probing. Device interrupts on the UNIBUS are now vectored through the code in a file **ubglue.s** produced by the configuration program. To mask as much as possible the differences between the different VAX processors, the 11/780 uses the same **ubglue.s** as the other processors which directly vector UNIBUS interrupts.

Many more of the exceptional conditions in the machine are caught now; only ‘‘SBI alert’’ and ‘‘SBI fault’’ remain uncaught by UNIX. The system control block is now defined in a file **scb.s** so that some symbols derived from C language header files by a program (and printed into a format suitable for inclusion in an assembly) may be stuck in after the system control block and before the mainline **locore.s**.

The primitive routines *copyseg* and *clearseg* are no longer run with the IPL raised very high. Further minor bugs have been fixed in the primitives, notably *addupc* (a bug which caused 1/8 of the profiling ticks to be lost), and *kernacc* (a bug which allowed a strange command to a certain program to crash the system).

machdep.c

Now sets up the error message buffer (in the last 1024 bytes of core) and the system data structures (such as the file and process table) at boot time. Currently only the file system buffer cache is sized at boot time, but all data structures are easily sized here. The startup routine also calls the routine *configure* to configure the system for the current hardware, locating available devices.

The *sendsig* routine passes a code back when a SIGFPE or SIGILL arrives, letting the signal handler determine which of the several conditions mapped to these two signals actually occurred. It uses the <frame.h> header file rather than redefining it.

The routines which monitor memory errors are now driven by interrupts (since the previous polling technique works only on 11/780). Extensive use of macros is made to make the various VAXen look similar. Instead of printing the raw contents of the memory controller registers, a array address and a syndrome is printed. Multiple memory controllers are supported.

The routines related to UNIBUS monitoring have been put with the rest of the UNIBUS routines in **../dev/uba.c**. The reboot interface has been improved, adding an automatic crash dump to a dump device (normally a disk aimed at the back end of a paging area). The system no longer ‘‘halts’’ when you ask it to (since this can cause a reboot to occur); rather it raises the IPL as high as it can and goes into a tight loop. Routines have been added to handle machine checks and print out the stack frame in a format which is readable by one who grok’s what the fields mean.

main.c Now establishes a red zone between the stack and *u.* area in process 0; further processes also have red zones, protecting the *u.* from too-large stacks. The main routines also setup the super-blocks which are locked into the file system buffer cache, and copy the name of the root file system (/) into its super-block so that the name will be available if, e.g., the root file system becomes full.

malloc.c Has been renamed **rmap.c**.

nami.c Now respects the notion of **..** in a directory which is a virtual root directory after a *chroot*(2) call.

prf.c No longer implements the ascii in-core event tracing facility, which proved to be too slow to be useful; a binary facility replaces it, and is also conditionalized on TRACE, but implemented in **vmmon.c**. Implements the output of numbers non-recursively, since the recursive method occasionally caused the kernel stacks to overflow. Implements a new kernel routine *uprintf* which prints directly on a user’s terminal for informing him/her of situations such as file systems which are full (because his/her program wrote to the file system when it was full.) Implements a new format ‘‘%b’’ which takes two arguments, a number and a second pattern. The pattern specifies a base to print the number in, and then a set of short strings separated by bit numbers (origin 1, escaped in octal into the string in the C compiler). The format prints the symbolic names for the bits which are in the string and set in the number within <>’s and separated by commas. This is extensively used to produce readable system error diagnostic messages on the console, decoding the bits of device registers symbolically.

The routines *prdev* and *deverr*, which printed diagnostics which were difficult to interpreter, are deleted. There are two new routines: *tablefull* which balks that a table is full, and *harderr* which begins a message about a hard (unrecoverable) error on a device.

prim.c Now maintains a count of free *clist* space. The code here now runs at **spl5** rather than **spl6** since there is no longer any need to block the clock.

rdwri.c Sees the change FASTVAX to not UNFAST. Also always clears the set-user-id bit when a file with the bit set is written on; previously this was done only ‘‘#if UCB’’. If you ‘‘#define INSECURITY’’ you get the code the old way.

scb.s Is a new file defining the system control block (as described above).

sig.c Has a bug fixed which caused processes to occasionally stop when the shell thought they were running. Processes are now given signals immediately when they are sent if the process is running.

slp.c A clumsiness which forced the swapout code to run with the IPL raised has been fixed by adding a variable *wantin* with which *wakeup* can communicate to the swapper that a swapped out process now wants to return to memory. The routine *setpri* has been modified so that processes which are over their declared (soft) memory size limitation are assigned lower CPU priority when the system is very tight on memory.

sys.c No longer allows detached jobs to access /dev/tty; this was a security glitch.

sys1.c Implements the ‘‘#!’’ executable shell script scheme. No longer lets executable files be read by users using *ptrace* unless the user has read access. Operates *exec* much more efficiently by avoiding copying argument lists unless the *exec* is going to succeed.

sys2.c The *openi* routine passes both the FREAD and FWRITE flags to its callees; this is needed by the magnetic tape open routines. The *maknode* routine sticks the whole argument value in the ‘‘rdev’’ field of the inode. This is used by the *badblock* (8) program to store block numbers corresponding to bad sectors on the disk in otherwise apparently empty files.

sys3.c The *mount* and *umount* calls have been changed to deal correctly with buffer flushing and with simulateous access by other programs to the file system block devices. The *mount* call also copies into the super-block of the file system the name of the device on which the file system is mounted (e.g. /usr).

sys4.c The *syslock* routine has been moved here from **acct.c**. The mechanisms for sending signals to all processes, which is used in shutting down the system, has been changed so that the process which is broadcasting the signal does not receive it itself. This allows the *halt* and *shutdown* programs to be written in a straightforward way.

trap.c Prints out the **pc** when an unexpected trap occurs. Handles AST’s to implement profiling ticks and for rescheduling rather than the (older style) use of reschedule interrupts. Allows process reschedules after page faults.

vmmon.c Contains the internal routine **trace1** which implements kernel event tracing in a circular buffer.

vmpage.c Tracing code, which is normally not compiled in, has been added. A extra case was added to an *if* statement to allow implementation of the vlimit(LIM_MAXRSS) feature of the system, for limiting processes which consume more than a process specific amount of physical memory. A botch was fixed in the virtual memory pre-paging which put prepaged pages in the clock loop rather than at the end of the free list. Code has been added to implement a additonal replacement algorithm for processes which are declared sequential: when a hard page fault occurs, the pages sequentially preceding the faulted page are returned to the free list.

vmproc.c Contains a number of small changes related to AST processing.

vmpt.c Also contains changes for handling AST’s as well as the initialization of the red-zone separating the stack and **u.** area of newly created processes. A bug in translation buffer flushing which caused rare and mysterious kernel crashes with the kernel stack not valid has been fixed.

vmsched.c Code has been added here which initializes the parameters of the clock page replacement algorithm based on the size of the machine. The *swapout* routine has been changed so that it no longer runs entirely at a high interrupt priority level (see **slp.c** above). The algorithm for the choice of processes to swap in and out and the hysteresis in the swap algorithm has been adjusted to work reasonably in extreme conditions when there are very large and or very few processes active in the system.

vmsubr.c Contains the *setredzone* routine definition.

vmsys.c Contains the user interface to the kernel tracing routines. Code has been added to *vadvise* to setup VA_SEQL.

Device support: sys/dev

The major change to the device subsystem is the support of multiple MASSBUS and UNIBUS adapters, the support for multiple instances of each particular controller, and the support of system configuration at bootstrap time, investigating the interconnects, devices, and controllers available on the machine. These changes will be discussed in detail in the next section, which describes how to change existing drivers to work in the new system and gives pointers on style for writing new drivers.

Other changes in the device drivers affecting more than one driver:

* The input silos for DH-11's and DZ-11's are no longer serviced at clock IPL. Rather the clock interrupt queues a software interrupt during to service the silos. This means that the device interrupt routines are called from IPL 0x15, the IPL at which they normally interrupt. Thus it is no longer necessary to define **spl5** to be **spl6** (blocking the clock) in routines which handle asynchronous line input/output.
* The internal interface to the line discipline routines has been changed slightly by reordering parameters to make the arguments to the various *ioctl* interfaces more similar; in particular *ttioctl* routine call has been changed. If you have locally written line disciplines or asynchronous device drivers you should check the interfaces.
* The tty interface now provides full 8-bit output when the terminal is in LLITOUT mode; this requires support from the *xx* param routines in the device drivers (e.g. from *dhparam* and *dzparam*.)
* The UNIBUS adapter support routines have changed substantially, to allow for queueing of requests when resources are short and for support of multiple UNIBUS adapters. The interface now also allows devices which cannot function when other DMA is active on the UNIBUS to obtain exclusive transient use of UNIBUS resources; this is needed to successfully run RK07 disk controllers in the presence of other buffered data path DMA. In addition, it is used by 6250bpi tape drives supported on the UNIBUS. See the section on configuration and UNIBUS device drivers below for more information.
* DEC standard bad sector forwarding is provided for all standard DEC devices using the DEC formatters; the code which implements this is easily ported to the storage module drivers in the system, and this is planned soon.*

bio.c The hashing of buffers has been changed to use the existing device chain two way links. This means that unhashing is much easier, saves space, and uses the pointers which were otherwise little used. The buffers are now kept on one of three lists when not busy: a list of super-blocks which are locked in core, a list of good data blocks, which is kept fifo and used to implement the LRU buffer cache, and a list of data blocks for which further usage is not anticipated; this is also kept fifo.

Calls to some new tracing routines are conditionally included in **bio.c**; we are using them to do some performance measurement. The **d_tab** field of the block device table has been changed to a **d_flags** field, and that change is known here, where old field was checked before (to see if it was non-zero). Better messages are printed now when swap space is exhausted, and a user is told on his/her terminal that a process was killed before it started because there was no space. A subroutine has been added to purge the blocks from a specific device from the cache; this is used to fix some long standing buffer cache flushing problems which prevented removable media from being used reliably.

bk.c The definition of **spl5** as **spl6** has been removed from here. The line discipline is included only if the specification

pseudo-device bk

is included in the system configuration. The input silos on **dh** and **dz** devices are used only when this line discipline is included in the system. The comment about future

* The hard thing in providing bad sector handling for non-DEC drives is providing a formatter which produces the bad block information and flags the bad sectors appropriately.

implementation of 8-bit paths with this discipline has been deleted, since there is no longer any intention of doing this.

conf.c Has been moved to this directory from the directory **../conf**. **This file should be changed only if you are adding support for a device not included on the standard distribution tape.**

ct.c Is a new driver, for a C/A/T phototypesetter interface.

dh.c No longer has to define **spl5** to be **spl6**. Incorporates the DM-11 driver standardly. A method is provided for specifying that lines are to be operated even though the hardware does not indicate that they are ready (using the flags word in the configuration specification, see *dh* (4)). A reasonable messages is printed when a **dh** silo overflows, replacing the old style of just printing a sequence of letter **o**'s on the console.

dhfdm.c Has been incorporated into **dh.c**.

dn.c A driver for the DEC DN-11 autodialer interface.

dsort.c Has been rewritten to correct a bug which caused the elevators to be sorted incorrectly.

dz.c No longer has to define **spl5** to be **spl5**. Has been changed to allow lines to be specified as not properly wired and brought up without the ready signals showing in the interface; see *dz* (4) for details. Prints reasonable diagnostics when the input silo overflows.

flp.c Knows that there is no floppy on an 11/750.

hp.c Is now a sub-driver to **mba.c**, which probes nexus space for the MASSBUS adaptors and device space on the MASSBUS's for disks, setting up the driver for each device which is in the configuration. A number of minor bugs and enhancements have been made to the driver: The driver handles the new RM80 drive and its SSE (skip-sector-error) facility for bad sector handling, as well as the DEC standard bad block forwarding. Due to the bad block forwarding, the last three tracks of each disk are normally reserved to the system and available only through the use of a special file system partition. A further bug has been fixed in the initialization of the tables for RM05 sectoring. The driver no longer (baroquely) turns on and off interrupts on the MASSBUS adapter. Basic dual-port drive handling code has been added to the driver.

The remaining remarks apply to all three supported disk drivers: the **hp** driver for MASSBUS disks, the **up** driver for UNIBUS storage modules, and the **hk** driver for RK07's: The drivers do not SEARCH or SEEK if there is only one drive on the MASSBUS. On a UNIBUS no SEARCHing or SEEKing is done if one drive is on the controller. The offset positions and recalibration of error recovery is now done with interrupts rather than by waiting for the operations to complete. This prevents the system from being tied up during the many recoveries of a disk operation, and is necessary in any case in at least one of the disk drivers (RK07). The iostat numbers for each MASSBUS and UNIBUS drive are calculated by the auto-configurator at boot time, not compiled into the drivers. Much cleaner handling of errors is done: the drivers realize which errors are not even potentially recoverable, handle drives spinning up and down with readable diagnostics, and print reasonable, legible error messages when hard errors and soft ecc's occur. Each driver includes a low-level non-interrupt driver used to take crash dumps at the end of a paging area on the device. The drivers include a raw i/o buffer per drive so that raw operations on separate devices can be overlapped (both seeks and transfers); previously only one raw device operation could be pending per device type.

ht.c The tape drive is now a sub-drive of the MASSBUS driver. The following remarks apply to all supported tape drivers: **ht** and **mt** for MASSBUS tapes, **ts** for the UNIBUS ts-11, and **tm** for the UNIBUS TM-11 emulations.

Each driver implements a set of tape ioctl operations on raw tapes providing access to the functionality of the hardware such as skipping forward and backward records and files and writing end-of-file marks on the tape. Better error diagnostics are also given on tape errors. Multiple tape controllers and transports are supported. A dump routine is provided with each driver for taking a post-mortem crash dump on tape, although dumps are normally made to the paging area on the disk.

With the exception of the **ts** driver, the drivers detect and reject attempts to switch tape density while writing a tape.

lp.c Is a fully supported driver for one or more line printer interfaces. It has been improved from the previous drivers (which were not supported) to take a small fraction of the number of interrupts that the previous drivers took. The user-level code driving the printers has been arranged to work on 1200 baud DECWRITER III terminals or true printers.

mba.c Has been rewritten. Now allows mixing of disks and tapes on the same and across multiple mba's, with the devices being driven from the routines here calling routines defined in the individual device drivers.

mem.c Has been fixed to not allow any access to nexus space, even by the super-users, since such access inevitably results in a machine check and a system crash.

mt.c A driver for the DEC TU78 tape drive.

mx?.c A bug has been fixed which, caused by a missing call to *chdrain* caused multiplexor files to become clogged under certain circumstances.

rk.c Is a new driver for RK07 disks. It uses the same logic as the storage module drive driver **up.c** whenever possible. It also makes use of the interlocking facilities of the UNIBUS device support because the **rk** controller cannot tolerate concurrent UNIBUS dma when it is operating due to a design flaw.

swap.c Now places only half of the first piece of the *swapmap* in the *argmap*.

swap??*.c Are the files for different swap configurations. Thus **swaphp.c** defines the root and swap devices for a UNIX based on a **hp** disk. The files such as **swaphphp.c** are for interleaved paging configurations, placing the swapping and paging activity on two disk arms. You can make additional such files and include them in your configuration files.

tdump.c Has been deleted, replaced by the dump routines in individual drivers.

tm.c Is a driver for UNIBUS tape drives on controllers such as the EMULEX TC-11. It has the same functionality as **ht** (see **ht.c** above.)

ts.c Is a driver for the UNIBUS TS-11 tape drive. It has full functionality except the transport itself only supports 1600 bpi.

tty.c No longer raises its IPL to **spl6** internally to block the clock. Has its internal interface to **ioctl** entries changed slightly to be globally consistent (see, e.g. *ttioctl*). The DIOC* ioctl entries have been deleted since they are not used in any standard UNIX line disciplines.

ttynew.c A bug is fixed which prevented echoing from occurring in raw mode. The dec-compatible method of ^S/^Q processing needed to support VT-100s in smooth scroll mode is implemented when the local mode ‘‘decctlq’’ is specified.

ttyold.c Implements ‘‘decctlq’’ mode.

tu.c A driver for the 11/750 TU58 console cassette interface. **Note: this driver provides reliable service only on a quiescent system.**

uba.c Has a much more structured interface. All the basic routines for dealing with the UNIBUS specify a UNIBUS adapter number to use, since there are potentially several on a machine. When requesting allocation of UNIBUS map entries, the caller specifies whether he is willing to block in the allocation routines waiting for resources to come available. If he is not, and there are no resources available, a value of 0 is returned, and the caller must deal with this. The routine which frees UNIBUS resources now takes the address of the variable describing the resources to be freed rather than the value of this variable to eliminate a race condition (where the routine is called, a UNIBUS interrupt occurs causing a UNIBUS reset, and the resources are freed twice, causing a *panic*).

The normal interface for DMA operation is now to pass a pointer to a UNIBUS related structure to a routine *ubago*, which allocates UNIBUS resources. If resources are not available, the structure is queued on a request queue, and processed when resources are available. When the requested resources are allocated, a driver specific *xxgo* routine is called, and can stuff the device registers with the address into which the operation is mapped and start the operation. The use of this interface is described in the next section.

Finally, we note that the error handling code which was written in assembly language is now written in C.

uda.c A driver for the UDA50 disk controller with RA80 Winchester storage modules.

up.c The UNIBUS storage module disk driver has been fixed up in the same way that the **hp** driver was, giving better error diagnostics and using interrupts during error recovery, etc. See **hp.c** above for details. The driver uses a feature of the EMULEX SC-21 to determine the size of the disks in use, so that it can adapt to both 300M storage modules and the Fujitsu 160M drives which are popular. Other drive sizes can be added easily.

va.c The **varian** printer-plotter driver has been modified so that it can support more than one device, probes the devices so they can be placed on differrent UNIBUS'es, and prints an error diagnostic when device errors are detected.

vaxcpu.c Is a new file which contains initializations of various CPU-type dependent structures.

vp.c Has been modified to handle multiple devices, and adapted to the auto-configuration code.

Configuration and UNIBUS device drivers

Someday this section will be a separate document. This section explains how to interface an existing UNIX device driver to the VAX system, especially to the UNIBUS routines and the autoconfiguration code.

A PDP-11, UNIX/32V or 3BSD or 4.0BSD driver on the VAX UNIBUS will need to be modified to run under 4.1BSD. There are three reasons why such a driver will need to be changed:

1) 4.1bsd supports multiple UNIBUS adapters.

2) 4.1bsd supports system configuration at boot time.

3) 4.1bsd manages the UNIBUS resources and does not crash when resources are not available; the resource allocation protocol must be honored. 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.

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) 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 **h/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 (1);
}
```

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 cancelled before the interrupt can be posted. The return of ‘‘1’’ here indicates that the probe routine is satisfied that the device is present. 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 TS-11 driver does something similar to this because the device is so difficult to force an interrupt on that it hardly seems worthwhile. (Besides, TS-11’s are usually present on small 11/750’s which have only one UNIBUS, and TS-11’s can have only exactly one transport per-controller so little probing is needed.)

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-existant 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++;
		upaddr = (struct updevice *)ui->ui_addr;
		upaddr->upcs1 = 0;
		upaddr->upcs2 = ui->ui_slave;
		upaddr->uphr = UPHR_MAXTRAK;
		if (upaddr->uphr == 9)
			ui->ui_type = 1;		/* fujitsu hack */
		upaddr->upcs2 = UPCS2_CLR;
	}
```

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 uses a hardware feature of the EMULEX SC-21 to ask if the number of tracks on the device is 9. If it is, then the driver assumes that the type is ‘‘1’’, which corresponds to a FUJITSU 160M drive. The alternative is the only other currently supported device, a 300 Megabyte CDC or AMPEX drive, which has **ui_type** 0. Note that if the controller is not an SC-21 then attempting to find out the maximum track in the device will yield an error, and a 300 Megabyte device will be assumed. In any case, any errors resulting from the attempt to type the drive are cleared by a controller clear before the routine returns.

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 not used by the system in this release, but may be used in future releases to look for instances of the device supported by the driver. In the current system, the configuration file specifies the control-status register addresses of all configured devices.

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 adaptor 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 the *config* (8) program 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 structure exists 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 the **config** (8) program 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. Thus the **up.c** driver uses a **ui_type** of 0 to mean a 300 Megabyte drive and a type of 1 to mean a 160 Megabyte FUJITSU drive.

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.

Changing drivers

If you driver does not do buffered data path DMA, conversion to the new system should be straightforward; if it uses buffered data paths more work will be required, but the task is really mostly cosmetic.

In any case, first add a line to the file **conf/files** of the form

    dev/zz.c optional zz device-driver

so that your driver will be included when you specify it in a configuration. Change the **dev/conf.c** file to include a block or character device entry for your device. Note that the block device entries now include a **d_dump** entry; if you are a block device but don’t have a dump entry point, just make one in your driver that returns the value ENODEV.

Then build a system configuration including your driver so that you have a compilation environment for your driver. You will have to add a **struct uba_driver** declaration for your driver, and change its calls to UNIBUS routines to correspond to these routines in the new system. Trouble spots will show up here. In particular, notice that you must specify flags to **uballoc** if you call it:

NEEDBDP if you need a buffered data path

CANTWAIT if you are calling (potentially) from interrupt level

You may discover that your driver ‘‘cantwait’’ but that you are calling from interrupt level. This botch existed in most previous VAX UNIX drivers, since there were no mechanisms for dealing with this. We will describe some options shortly.

First, suppose your driver doesn't do buffered data path dma. What else is there for you to do? Very little really. You should change your driver to print messages on the console in the format now used by all device drivers; see section 4 of the revised programmers manual for details. To make more certain that your driver is ready for the new system environment, look at some of the simple existing drivers and mimic the style to create the portions of the driver which are needed to interface with the configuration part of the system. Useful drivers to look at may be:

ct.c Very simple drive which does programmed i/o to C/A/T phototypesetter.

dh.c Communications line driver which uses non-buffered UNIBUS dma for output.

dz.c Communications line driver which does programmed i/o.

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 **ct.c** are nearly minimal. Order is important here, don't be suprised 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 or 750).

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.

Adapting devices which do buffered data path dma

These devices fall into two categories: those which are controllers to which devices are attached, and those which are just single devices. The interface for the former is very stylized and we recommend that you simply mimic one of the existing tape or disk drivers in adapting to the system. You will find that the existing tape and disk drivers are all **very** similar; this is deliberate so that it isn't necessary to rewrite the whole driver for each device, since the available devices are typically very similar.

Other devices which do buffered data path DMA can be adapted to the new system in one of two ways:

* They can do their own data path allocation, calling the UNIBUS allocation routines from the ''top-half'' (non-interrupt) code, sleeping in the UNIBUS code when resources are not available. See for

an example the code in the **vp.c** driver.

* They can set up a two-level structure like the tape and disk drivers do, and call the *ubago* routine and use the **ud_dgo** interface to start DMA operations. See for an example the code in the **up.c** driver.

Either way works acceptably well; the second (*ubago*) interface is preferable because it does not force a context switch per i/o operation (to the routine driving the i/o from the ‘‘top-half’’).

If you have questions about converting drivers, feel free to call us and ask or to send us mail. We hope (eventually) to write a more complete paper for driver writers, but don’t have the manpower to do this just now.