NAME

ddb - interactive kernel debugger

SYNOPSIS

In order to enable kernel debugging facilities include:

options KDB
options DDB

To prevent activation of the debugger on kernel panic(9):

options KDB_UNATTENDED

In order to print a stack trace of the current thread on the console for a panic:

options KDB_TRACE

To print the numerical value of symbols in addition to the symbolic representation, define:

options DDB_NUMSYM

To enable the gdb(4) backend, so that remote debugging with kgdb(1) (*ports/devel/gdb*) is possible, include:

options GDB

DESCRIPTION

The **ddb** kernel debugger is an interactive debugger with a syntax inspired by gdb(1) (*ports/devel/gdb*). If linked into the running kernel, it can be invoked locally with the ‘debug’ keymap(5) action, usually mapped to Ctrl+Alt+Esc, or by setting the *debug.kdb.enter* sysctl to 1. The debugger is also invoked on kernel panic(9) if the *debug.debugger_on_panic* sysctl(8) MIB variable is set non-zero, which is the default unless the KDB_UNATTENDED option is specified. Similarly, if the *debug.debugger_on_recursive_panic* variable is set to 1, then the debugger will be invoked on a recursive kernel panic. This variable has a default value of 0, and has no effect if *debug.debugger_on_panic* is already set non-zero.

The current location is called *dot*. The *dot* is displayed with a hexadecimal format at a prompt. The commands **examine** and **write** update *dot* to the address of the last line examined or the last location modified, and set *next* to the address of the next location to be examined or changed. Other commands do not change *dot*, and set *next* to be the same as *dot*.

The general command syntax is: *command*[/*modifier*] [*addr*][,*count*]

A blank line repeats the previous command from the address *next* with count 1 and no modifiers. Specifying *addr* sets *dot* to the address. Omitting *addr* uses *dot*. A missing *count* is taken to be 1 for printing commands or infinity for stack traces. A *count* of -1 is equivalent to a missing *count*. Options that are supplied but not supported by the given *command* are usually ignored.

The **ddb** debugger has a pager feature (like the more(1) command) for the output. If an output line exceeds the number set in the *lines* variable, it displays "--More--" and waits for a response. The valid responses for it are:

SPC	one more page
RET	one more line
q	abort the current command, and return to the command input mode

Finally, **ddb** provides a small (currently 10 items) command history, and offers simple **emacs**-style command line editing capabilities. In addition to the **emacs** control keys, the usual ANSI arrow keys may be used to browse through the history buffer, and move the cursor within the current line.

COMMANDS

COMMON DEBUGGER COMMANDS

help Print a short summary of the available commands and command abbreviations.

examine[/**AISabcdghilmorsuxz ...**] [*addr*][,*count*]
x[/**AISabcdghilmorsuxz ...**] [*addr*][,*count*]

Display the addressed locations according to the formats in the modifier. Multiple modifier formats display multiple locations. If no format is specified, the last format specified for this command is used.

The format characters are:

b	look at by bytes (8 bits)
h	look at by half words (16 bits)
l	look at by long words (32 bits)
g	look at by quad words (64 bits)
a	print the location being displayed
A	print the location with a line number if possible
x	display in unsigned hex
z	display in signed hex
o	display in unsigned octal

d	display in signed decimal
u	display in unsigned decimal
r	display in current radix, signed
c	display low 8 bits as a character. Non-printing characters are displayed as an octal escape code (e.g., ‘\000’).
s	display the null-terminated string at the location. Non-printing characters are displayed as octal escapes.
m	display in unsigned hex with character dump at the end of each line. The location is also displayed in hex at the beginning of each line.
i	display as a disassembled instruction
I	display as a disassembled instruction with possible alternate formats depending on the machine. On i386, this selects the alternate format for the instruction decoding (16 bits in a 32-bit code segment and vice versa).
S	display a symbol name for the pointer stored at the address

xf Examine forward: execute an **examine** command with the last specified parameters to it except that the next address displayed by it is used as the start address.

xb Examine backward: execute an **examine** command with the last specified parameters to it except that the last start address subtracted by the size displayed by it is used as the start address.

print[/**acdoruxz**]
p[/**acdoruxz**]

Print *addr*s according to the modifier character (as described above for **examine**). Valid formats are: **a**, **x**, **z**, **o**, **d**, **u**, **r**, and **c**. If no modifier is specified, the last one specified to it is used. The argument *addr* can be a string, in which case it is printed as it is. For example:

```
print/x "eax = " $eax "\necx = " $ecx "\n"
```

will print like:

```
eax = xxxxxx
ecx = yyyyyy
```

write[/**bhl**] *addr expr1* [*expr2 ...*]
w[/**bhl**] *addr expr1* [*expr2 ...*]

Write the expressions specified after *addr* on the command line at succeeding locations starting with *addr*. The write unit size can be specified in the modifier with a letter **b** (byte), **h** (half word) or **l** (long word) respectively. If omitted, long word is assumed.

Warning: since there is no delimiter between expressions, strange things may happen. It is best to enclose each expression in parentheses.

set $*variable* [=] *expr*

Set the named variable or register with the value of *expr*. Valid variable names are described below.

break[/**u**] [*addr*][,*count*]
b[/**u**] [*addr*][,*count*]

Set a break point at *addr*. If *count* is supplied, the **continue** command will not stop at this break point on the first *count* - 1 times that it is hit. If the break point is set, a break point number is printed with '#'. This number can be used in deleting the break point or adding conditions to it.

If the **u** modifier is specified, this command sets a break point in user address space. Without the **u** option, the address is considered to be in the kernel space, and a wrong space address is rejected with an error message. This modifier can be used only if it is supported by machine dependent routines.

Warning: If a user text is shadowed by a normal user space debugger, user space break points may not work correctly. Setting a break point at the low-level code paths may also cause strange behavior.

delete [*addr*]
d [*addr*]
delete #*number*
d #*number*

Delete the specified break point. The break point can be specified by a break point number with '#', or by using the same *addr* specified in the original **break** command, or by omitting *addr* to get the default address of *dot*.

halt Halt the system.

watch [*addr*][,*size*]

Set a watchpoint for a region. Execution stops when an attempt to modify the region occurs. The *size* argument defaults to 4. If you specify a wrong space address, the request is rejected with an error message.

Warning: Attempts to watch wired kernel memory may cause unrecoverable error in some systems such as i386. Watchpoints on user addresses work best.

hwatch [*addr*][,*size*]
: Set a hardware watchpoint for a region if supported by the architecture. Execution stops when an attempt to modify the region occurs. The *size* argument defaults to 4.

  Warning: The hardware debug facilities do not have a concept of separate address spaces like the watch command does. Use **hwatch** for setting watchpoints on kernel address locations only, and avoid its use on user mode address spaces.

dhwatch [*addr*][,*size*]
: Delete specified hardware watchpoint.

kill *sig pid*
: Send signal *sig* to process *pid*. The signal is acted on upon returning from the debugger. This command can be used to kill a process causing resource contention in the case of a hung system. See signal(3) for a list of signals. Note that the arguments are reversed relative to kill(2).

step[/**p**][,*count*]
s[/**p**][,*count*]
: Single step *count* times. If the **p** modifier is specified, print each instruction at each step. Otherwise, only print the last instruction.

  Warning: depending on machine type, it may not be possible to single-step through some low-level code paths or user space code. On machines with software-emulated single-stepping (e.g., pmax), stepping through code executed by interrupt handlers will probably do the wrong thing.

continue[/**c**]
c[/**c**] Continue execution until a breakpoint or watchpoint. If the **c** modifier is specified, count instructions while executing. Some machines (e.g., pmax) also count loads and stores.

  Warning: when counting, the debugger is really silently single-stepping. This means that single-stepping on low-level code may cause strange behavior.

until[/**p**]
: Stop at the next call or return instruction. If the **p** modifier is specified, print the call nesting depth and the cumulative instruction count at each call or return. Otherwise, only print when the matching return is hit.

next[/**p**]
match[/**p**]
: Stop at the matching return instruction. If the **p** modifier is specified, print the call nesting depth

and the cumulative instruction count at each call or return. Otherwise, only print when the matching return is hit.

trace[/**u**] [*pid* | *tid*][,*count*]
t[/**u**] [*pid* | *tid*][,*count*]
where[/**u**] [*pid* | *tid*][,*count*]
bt[/**u**] [*pid* | *tid*][,*count*]

Stack trace. The **u** option traces user space; if omitted, **trace** only traces kernel space. The optional argument *count* is the number of frames to be traced. If *count* is omitted, all frames are printed.

Warning: User space stack trace is valid only if the machine dependent code supports it.

search[/**bhl**] *addr value* [*mask*][,*count*]

Search memory for *value*. The optional *count* argument limits the search.

reboot[/**s**] [*seconds*]
reset[/**s**] [*seconds*]

Hard reset the system. If the optional argument *seconds* is given, the debugger will wait for this long, at most a week, before rebooting. When the **s** modifier is given, the command will skip running any registered shutdown handlers and attempt the most basic reset.

thread *addr* | *tid*

Switch the debugger to the thread with ID *tid*, if the argument is a decimal number, or address *addr*, otherwise.

watchdog [*exp*]

Program the watchdog(4) timer to fire in 2^*exp* seconds. If no argument is provided, the watchdog timer is disabled.

SPECIALIZED HELPER COMMANDS

findstack *addr*

Prints the address of the thread whose kernel-mode stack contains *addr*, if any.

show active trace
acttrace

Show a stack trace for every thread running on a CPU.

show all procs[/**a**]
ps[/**a**] Display all process information. The process information may not be shown if it is not supported

in the machine, or the bottom of the stack of the target process is not in the main memory at that time. The **a** modifier will print command line arguments for each process.

show all trace
alltrace
Show a stack trace for every thread in the system.

show all ttys
Show all TTY's within the system. Output is similar to pstat(8), but also includes the address of the TTY structure.

show all vnets
Show the same output as "show vnet" does, but lists all virtualized network stacks within the system.

show allchains
Show the same information like "show lockchain" does, but for every thread in the system.

show alllocks
Show all locks that are currently held. This command is only available if witness(4) is included in the kernel.

show allpcpu
The same as "show pcpu", but for every CPU present in the system.

show allrman
Show information related with resource management, including interrupt request lines, DMA request lines, I/O ports, I/O memory addresses, and Resource IDs.

show apic
Dump data about APIC IDT vector mappings.

show badstacks
Walk the witness(4) graph and print any lock-order violations. This command is only available if witness(4) is included in the kernel.

show breaks
Show breakpoints set with the "break" command.

show bio *addr*

Show information about the bio structure *struct bio* present at *addr*. See the *sys/bio.h* header file and g_bio(9) for more details on the exact meaning of the structure fields.

show buffer *addr*
Show information about the buf structure *struct buf* present at *addr*. See the *sys/buf.h* header file for more details on the exact meaning of the structure fields.

show callout *addr*
Show information about the callout structure *struct callout* present at *addr*.

show cdev [*addr*]
Show the internal devfs state of the cdev structure located at *addr*. If no argument is provided, show the list of all created cdevs, consisting of the devfs node name and the *struct cdev* address.

show conifhk
Lists hooks currently waiting for completion in **run_interrupt_driven_config_hooks**().

show cpusets
Print numbered root and assigned CPU affinity sets. See cpuset(2) for more details.

show cyrixreg
Show registers specific to the Cyrix processor.

show devmap
Prints the contents of the static device mapping table. Currently only available on the ARM architecture.

show domain *addr*
Print protocol domain structure *struct domain* at address *addr*. See the *sys/domain.h* header file for more details on the exact meaning of the structure fields.

show ffs [*addr*]
Show brief information about ffs mount at the address *addr*, if argument is given. Otherwise, provides the summary about each ffs mount.

show file *addr*
Show information about the file structure *struct file* present at address *addr*.

show files
Show information about every file structure in the system.

show freepages
Show the number of physical pages in each of the free lists.

show geom [*addr*]
If the *addr* argument is not given, displays the entire GEOM topology. If *addr* is given, displays details about the given GEOM object (class, geom, provider or consumer).

show idt
Show IDT layout. The first column specifies the IDT vector. The second one is the name of the interrupt/trap handler. Those functions are machine dependent.

show igi_list *addr*
Show information about the IGMP structure *struct igmp_ifsoftc* present at *addr*.

show iosched *addr*
Show information about the I/O scheduler *struct cam_iosched_softc* located at *addr*.

show inodedeps [*addr*]
Show brief information about each inodedep structure. If *addr* is given, only inodedeps belonging to the fs located at the supplied address are shown.

show inpcb *addr*
Show information on IP Control Block *struct in_pcb* present at *addr*.

show intr
Dump information about interrupt handlers.

show intrcnt
Dump the interrupt statistics.

show irqs
Show interrupt lines and their respective kernel threads.

show ktr[/**avV**]
Print the contents of the ktr(4) trace buffer. The **v** modifier will request fully verbose output, causing the file, line number, and timestamp to be printed for each trace entry. The **V** modifier will request only the timestamps to be printed. The **a** modifier will request that the output be unpaginated.

show lapic

Show information from the local APIC registers for this CPU.

show lock *addr*

Show lock structure. The output format is as follows:

class:

Class of the lock. Possible types include mutex(9), rmlock(9), rwlock(9), sx(9).

name:

Name of the lock.

flags:

Flags passed to the lock initialization function. *flags* values are lock class specific.

state:

Current state of a lock. *state* values are lock class specific.

owner:

Lock owner.

show lockchain *addr*

Show all threads a particular thread at address *addr* is waiting on based on non-spin locks.

show lockedbufs

Show the same information as "show buf", but for every locked *struct buf* object.

show lockedvnods

List all locked vnodes in the system.

show locks

Prints all locks that are currently acquired. This command is only available if witness(4) is included in the kernel.

show locktree

show malloc[/**i**]

Prints malloc(9) memory allocator statistics. If the **i** modifier is specified, format output as machine-parseable comma-separated values ("CSV"). The output columns are as follows:

Type Specifies a type of memory. It is the same as a description string used while defining the given memory type with MALLOC_DECLARE(9).

InUse Number of memory allocations of the given type, for which free(9) has not been called yet.

MemUse
Total memory consumed by the given allocation type.

Requests Number of memory allocation requests for the given memory type.

The same information can be gathered in userspace with "**vmstat -m**".

show map[/**f**] *addr*
Prints the VM map at *addr*. If the **f** modifier is specified the complete map is printed.

show msgbuf
Print the system's message buffer. It is the same output as in the "**dmesg**" case. It is useful if you got a kernel panic, attached a serial cable to the machine and want to get the boot messages from before the system hang.

show mount [*addr*]
Displays details about the mount point located at *addr*. If no *addr* is specified, displays short info about all currently mounted file systems.

show object[/**f**] *addr*
Prints the VM object at *addr*. If the **f** option is specified the complete object is printed.

show panic
Print the panic message if set.

show page
Show statistics on VM pages.

show pageq
Show statistics on VM page queues.

show pciregs
Print PCI bus registers. The same information can be gathered in userspace by running "**pciconf -lv**".

show pcpu
Print current processor state. The output format is as follows:

cpuid	Processor identifier.
curthread	Thread pointer, process identifier and the name of the process.
curpcb	Control block pointer.
fpcurthread	FPU thread pointer.
idlethread	Idle thread pointer.
APIC ID	CPU identifier coming from APIC.
currentldt	LDT pointer.
spin locks held	Names of spin locks held.

show pgrpdump
Dump process groups present within the system.

show prison [*addr*]
Show the prison structure located at *addr*. If no *addr* argument is specified, show information about all prisons in the system.

show proc [*addr*]
Show information about the process structure located at address *addr*, or the current process if no argument is specified.

show procvm [*addr*]
Show process virtual memory layout for the process located at *addr*, or the current process if no argument is specified.

show protosw *addr*
Print protocol switch structure *struct protosw* at address *addr*.

show registers[/**u**]
Display the register set. If the **u** modifier is specified, the register contents of the thread's previous trapframe are displayed instead. Usually, this corresponds to the saved state from userspace.

show rman *addr*
Show resource manager object *struct rman* at address *addr*. Addresses of particular pointers can be gathered with "show allrman" command.

show route *addr*
Show route table result for destination *addr*. At this time, INET and INET6 formatted addresses are supported.

show routetable [*af*]
Show full route table or tables. If *af* is specified, show only routes for the given numeric address family. If no argument is specified, dump the route table for all address families.

show rtc
Show real time clock value. Useful for long debugging sessions.

show sleepchain
Deprecated. Now an alias for **show lockchain**.

show sleepq *addr*
show sleepqueue *addr*
Show the sleepqueue(9) structure located at *addr*.

show sockbuf *addr*
Show the socket buffer *struct sockbuf* located at *addr*.

show socket *addr*
Show the socket object *struct socket* located at *addr*.

show sysregs
Show system registers (e.g., cr0-4 on i386.) Not present on some platforms.

show tcpcb *addr*
Print TCP control block *struct tcpcb* lying at address *addr*. For exact interpretation of output, visit *netinet/tcp.h* header file.

show thread [*addr* | *tid*]
If no *addr* or *tid* is specified, show detailed information about current thread. Otherwise, print information about the thread with ID *tid* or kernel address *addr*. (If the argument is a decimal number, it is assumed to be a tid.)

show threads
Show all threads within the system. Output format is as follows:

First column	Thread identifier (TID)
Second column	Thread structure address
Third column	Backtrace.

show tty *addr*

Display the contents of a TTY structure in a readable form.

show turnstile *addr*

Show turnstile *struct turnstile* structure at address *addr*. Turnstiles are structures used within the FreeBSD kernel to implement synchronization primitives which, while holding a specific type of lock, cannot sleep or context switch to another thread. Currently, those are: mutex(9), rwlock(9), rmlock(9).

show uma[/**i**]

Show UMA allocator statistics. If the **i** modifier is specified, format output as machine-parseable comma-separated values ("CSV"). The output contains the following columns:

Zone	Name of the UMA zone. The same string that was passed to uma_zcreate(9) as a first argument.
Size	Size of a given memory object (slab).
Used	Number of slabs being currently used.
Free	Number of free slabs within the UMA zone.
Requests	Number of allocations requests to the given zone.
Total Mem	Total memory in use (either allocated or free) by a zone, in bytes.
XFree	Number of free slabs within the UMA zone that were freed on a different NUMA domain than allocated. (The count in the **Free** column is inclusive of **XFree**.)

The same information might be gathered in the userspace with the help of "**vmstat -z**".

show unpcb *addr*

Shows UNIX domain socket private control block *struct unpcb* present at the address *addr*.

show vmochk

Prints, whether the internal VM objects are in a map somewhere and none have zero ref counts.

show vmopag

Walk the list of VM objects in the system, printing the indices and physical addresses of the VM pages belonging to each object.

show vnet *addr*

Prints virtualized network stack *struct vnet* structure present at the address *addr*.

show vnode *addr*

Prints vnode *struct vnode* structure lying at *addr*. For the exact interpretation of the output, look at the *sys/vnode.h* header file.

show vnodebufs *addr*
Shows clean/dirty buffer lists of the vnode located at *addr*.

show vpath *addr*
Walk the namecache to lookup the pathname of the vnode located at *addr*.

show watches
Displays all watchpoints. Shows watchpoints set with "watch" command.

show witness
Shows information about lock acquisition coming from the witness(4) subsystem.

OFFLINE DEBUGGING COMMANDS

dump Initiate a kernel core dump to the device(s) configured by dumpon(8).

gdb Switches to remote GDB mode. In remote GDB mode, another machine is required that runs gdb(1) (*ports/devel/gdb*) using the remote debug feature, with a connection to the serial console port on the target machine.

netdump **-s** *server* [**-g** *gateway* **-c** *client* **-i** *iface*]
Configure netdump(4) with the provided parameters, and immediately perform a netdump.

There are some known limitations. Principally, netdump(4) only supports IPv4 at this time. The address arguments to the **netdump** command must be dotted decimal IPv4 addresses. (Hostnames are not supported.) At present, the command only works if the machine is in a panic state. Finally, the **ddb netdump** command does not provide any way to configure compression or encryption.

netgdb **-s** *server* [**-g** *gateway* **-c** *client* **-i** *iface*]
Initiate a netgdb(4) session with the provided parameters.

netgdb has identical limitations to **netdump**.

capture on
capture off
capture reset
capture status
ddb supports a basic output capture facility, which can be used to retrieve the results of

debugging commands from userspace using sysctl(3). **capture on** enables output capture; **capture off** disables capture. **capture reset** will clear the capture buffer and disable capture. **capture status** will report current buffer use, buffer size, and disposition of output capture.

Userspace processes may inspect and manage **ddb** capture state using sysctl(8):

debug.ddb.capture.bufsize may be used to query or set the current capture buffer size.

debug.ddb.capture.maxbufsize may be used to query the compile-time limit on the capture buffer size.

debug.ddb.capture.bytes may be used to query the number of bytes of output currently in the capture buffer.

debug.ddb.capture.data returns the contents of the buffer as a string to an appropriately privileged process.

This facility is particularly useful in concert with the scripting and textdump(4) facilities, allowing scripted debugging output to be captured and committed to disk as part of a textdump for later analysis. The contents of the capture buffer may also be inspected in a kernel core dump using kgdb(1) (*ports/devel/gdb*).

run
script
scripts
unscript
Run, define, list, and delete scripts. See the *SCRIPTING* section for more information on the scripting facility.

textdump dump
textdump set
textdump status
textdump unset
Use the **textdump dump** command to immediately perform a textdump. More information may be found in textdump(4). The **textdump set** command may be used to force the next kernel core dump to be a textdump rather than a traditional memory dump or minidump. **textdump status** reports whether a textdump has been scheduled. **textdump unset** cancels a request to perform a textdump as the next kernel core dump.

VARIABLES

The debugger accesses registers and variables as $*name*. Register names are as in the "**show registers**" command. Some variables are suffixed with numbers, and may have some modifier following a colon immediately after the variable name. For example, register variables can have a **u** modifier to indicate user register (e.g., "$eax:u").

Built-in variables currently supported are:

radix	Input and output radix.
maxoff	Addresses are printed as "*symbol*+*offset*" unless *offset* is greater than *maxoff*.
maxwidth	The width of the displayed line.
lines	The number of lines. It is used by the built-in pager. Setting it to 0 disables paging.
tabstops	Tab stop width.
workxx	Work variable; *xx* can take values from 0 to 31.

EXPRESSIONS

Most expression operators in C are supported except '~', '^', and unary '&'. Special rules in **ddb** are:

Identifiers	The name of a symbol is translated to the value of the symbol, which is the address of the corresponding object. '.' and ':' can be used in the identifier. If supported by an object format dependent routine, [*filename*:]*func*:*lineno*, [*filename*:]*variable*, and [*filename*:]*lineno* can be accepted as a symbol.
Numbers	Radix is determined by the first two letters: '0x': hex, '0o': octal, '0t': decimal; otherwise, follow current radix.
.	*dot*
+	*next*
..	address of the start of the last line examined. Unlike *dot* or *next*, this is only changed by **examine** or **write** command.
'	last address explicitly specified.
$*variable*	Translated to the value of the specified variable. It may be followed by a ':' and modifiers as described above.
a#*b*	A binary operator which rounds up the left hand side to the next multiple of right hand side.

**expr*	Indirection. It may be followed by a ‘:’ and modifiers as described above.

SCRIPTING

ddb supports a basic scripting facility to allow automating tasks or responses to specific events. Each script consists of a list of DDB commands to be executed sequentially, and is assigned a unique name. Certain script names have special meaning, and will be automatically run on various **ddb** events if scripts by those names have been defined.

The **script** command may be used to define a script by name. Scripts consist of a series of **ddb** commands separated with the ‘;’ character. For example:

```
script kdb.enter.panic=bt; show pcpu
script lockinfo=show alllocks; show lockedvnods
```

The **scripts** command lists currently defined scripts.

The **run** command execute a script by name. For example:

```
run lockinfo
```

The **unscript** command may be used to delete a script by name. For example:

```
unscript kdb.enter.panic
```

These functions may also be performed from userspace using the ddb(8) command.

Certain scripts are run automatically, if defined, for specific **ddb** events. The follow scripts are run when various events occur:

kdb.enter.acpi	The kernel debugger was entered as a result of an acpi(4) event.
kdb.enter.bootflags	The kernel debugger was entered at boot as a result of the debugger boot flag being set.
kdb.enter.break	The kernel debugger was entered as a result of a serial or console break.
kdb.enter.cam	The kernel debugger was entered as a result of a CAM(4) event.
kdb.enter.mac	The kernel debugger was entered as a result of an assertion failure in the mac_test(4) module of the TrustedBSD MAC Framework.

kdb.enter.netgraph	The kernel debugger was entered as a result of a netgraph(4) event.
kdb.enter.panic	panic(9) was called.
kdb.enter.powerpc	The kernel debugger was entered as a result of an unimplemented interrupt type on the powerpc platform.
kdb.enter.sysctl	The kernel debugger was entered as a result of the *debug.kdb.enter* sysctl being set.
kdb.enter.unionfs	The kernel debugger was entered as a result of an assertion failure in the union file system.
kdb.enter.unknown	The kernel debugger was entered, but no reason has been set.
kdb.enter.vfslock	The kernel debugger was entered as a result of a VFS lock violation.
kdb.enter.watchdog	The kernel debugger was entered as a result of a watchdog firing.
kdb.enter.witness	The kernel debugger was entered as a result of a witness(4) violation.

In the event that none of these scripts is found, **ddb** will attempt to execute a default script:

kdb.enter.default	The kernel debugger was entered, but a script exactly matching the reason for entering was not defined. This can be used as a catch-all to handle cases not specifically of interest; for example, *kdb.enter.witness* might be defined to have special handling, and *kdb.enter.default* might be defined to simply panic and reboot.

HINTS

On machines with an ISA expansion bus, a simple NMI generation card can be constructed by connecting a push button between the A01 and B01 (CHCHK# and GND) card fingers. Momentarily shorting these two fingers together may cause the bridge chipset to generate an NMI, which causes the kernel to pass control to **ddb**. Some bridge chipsets do not generate a NMI on CHCHK#, so your mileage may vary. The NMI allows one to break into the debugger on a wedged machine to diagnose problems. Other bus' bridge chipsets may be able to generate NMI using bus specific methods. There are many PCI and PCIe add-in cards which can generate NMI for debugging. Modern server systems typically use IPMI to generate signals to enter the debugger. The *devel/ipmitool* port can be used to send the **chassis power diag** command which delivers an NMI to the processor. Embedded systems often use JTAG for debugging, but rarely use it in combination with **ddb**.

Serial consoles can break to the debugger by sending a BREAK condition on the serial line. This requires a kernel built with **options BREAK_TO_DEBUGGER** is specified in the kernel. Most terminal emulation programs can send a break sequence with a special key sequence or menu selection. Sending the break can be difficult or even happen spuriously in some setups. An alternative method is to build a kernel with **options ALT_BREAK_TO_DEBUGGER** then the sequence of CR TILDE CTRL-B enters the debugger; CR TILDE CTRL-P causes a panic; and CR TILDE CTRL-R causes an immediate reboot. In all these sequences, CR represents Carriage Return and is usually sent by pressing the Enter or Return key. TILDE is the ASCII tilde character (~). CTRL-x is Control x, sent by pressing the Control key, then x, then releasing both.

The break-to-debugger behavior can be enabled by setting sysctl(8) *debug.kdb.break_to_debugger* to 1. The alt-break-to-debugger behavior can be enabled by setting sysctl(8) *debug.kdb.alt_break_to_debugger* to 1. The debugger can be entered by setting sysctl(8) *debug.kdb.enter* to 1.

Output can be interrupted, paused, and resumed with the control characters CTRL-C, CTRL-S, and CTRL-Q. Because these control characters are received as in-band data from the console, there is an input buffer, and once that buffer fills **ddb** must either stop responding to control characters or drop additional input while continuing to search for control characters. This behavior is controlled by the tunable sysctl(8) *debug.ddb.prioritize_control_input*, which defaults to 1. The input buffer size is 512 bytes.

FILES

Header files mentioned in this manual page can be found below */usr/include* directory.

- *sys/buf.h*
- *sys/domain.h*
- *netinet/in_pcb.h*
- *sys/socket.h*
- *sys/vnode.h*

SEE ALSO

gdb(1) (*ports/devel/gdb*), kgdb(1) (*ports/devel/gdb*), acpi(4), CAM(4), gdb(4), mac_ddb(4), mac_test(4), netgraph(4), textdump(4), witness(4), ddb(8), sysctl(8), panic(9)

HISTORY

The **ddb** debugger was developed for Mach, and ported to 386BSD-0.1. This manual page translated from man(7) macros by Garrett Wollman.

Robert N. M. Watson added support for **ddb** output capture, textdump(4) and scripting in FreeBSD 7.1.