LINUX KERNEL PARAMETERS

Dip your toe into the mysterious heart of your Linux machine, with **Andrew Conway** and the magic of Linux kernel parameters.

The dark days when new computer hardware often required compiling your own kernel are now firmly in Linux's past (though those were fun days). But the fact that Linux – meaning the kernel itself – is free software means that you can still delve deep into its innards and tweak it to your heart's content.

In an ideal world, the user would never need to think about the kernel, but there are cases where it's useful. In fact, it becomes a necessity when hardware is faulty, such as with bad memory, or if it is shipped with buggy firmware that tells lies about capabilities such as CPU speed or temperature. In these cases, you'll need more control over the hardware than userspace software allows, and setting kernel parameters lets you do that without the hassle of compiling a kernel.

“In an ideal world the user would never need to think about the kernel, but sometimes we have to.”

Let's start with an example we encountered when bad memory came to plague a shiny new laptop. Sometimes it's not possible to replace the memory (for example, the SOC in a Raspberry Pi), and for some the expense of buying replacement hardware may be prohibitive – think of folk using old hardware in developing countries.

If you're experiencing the symptoms of bad memory – random freezes and crashes – then you should test it using a venerable utility called *memtest86+*. Many distros include it as an option at the boot prompt, or you can put a distro such as GPartedLive on a USB stick and select the **memtest86+** option. The test will tell you if you've got bad memory and exactly where it is bad. In our case, the bad patch was reported as 2056.0MB to 2176.0MB. The solution was to restart the laptop, and when the bootloader began, switch to its command line and set the **memmap** kernel parameter with

```
memmap=256M$2048M
```

This instructs the kernel not to use the 256MB of memory from 2048MB, and once booted with this parameter setting, the laptop became completely stable. The only noticable difference was that it had 256MB less memory than before. Given that it had 8GB of memory in the first place, this loss isn't too much of a problem and saves on the cost and hassle of having it fixed if outside the warranty period.

The arcane lore of the kernel

The **memmap** fix is straightforward enough once you understand it, but first you have to know that such a parameter exists and what its cryptic syntax means. For example, you can replace **$** with **&** or **#** and it'll work completely differently, and if you don't respect

DISCLAIMER

The examples we've picked are unlikely to cause trouble, but altering kernel parameters can cause crashes and data loss. Tread lightly and experiment in a VM or a non-production system.

memory boundaries it won't work at all. The aim of this article is to explore kernel parameters in case you have to use them in real situations involving faulty hardware or custom hobby projects.

To set kernel parameters at boot time you need to get to know your bootloader. You can either set kernel parameters manually each time or edit the bootloader's configuration file so that it gets set automatically on every boot. We'll concentrate on the most popular bootloader *Grub 2*, but the parameters themselves will be the same for any bootloader such as *Lilo* or *SysLinux*.

To boot

You need to pay close attention at startup. If you see a text screen with 'GNU GRUB' at the top, then just press E before the timeout ends and booting begins. If you don't see the *Grub* screen, and by default you won't with a normal Ubuntu install, you'll need to press the Shift key to enter the *Grub* menu. This didn't work for us when booting Ubuntu 14.04 in a *VirtualBox* VM due to a problem with the VM capturing the keyboard, so if you experience it too, or you wish to boot to the *Grub* screen everytime instead of frantically jabbing at the Shift key, let it boot up into the OS, open a terminal window and edit the configuration file:

```
sudo nano /etc/default/grub
```

and put a **#** at the start of the line with **GRUB_HIDDEN_TIMEOUT=0** to comment it out. Then save the file and tell *Grub* to update with:

```
sudo update-grub
```

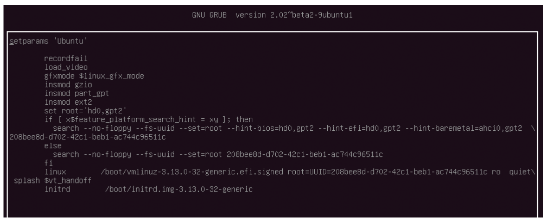
This causes *Grub* to rewrite its files on the disk. These are essential for booting your computer, so be vigilant for errors or warnings, and if you do get any, check online to find out what they mean and take action if needed. If **update-grub** did its job properly, reboot your computer and you should be presented with the GNU Grub screen where you can press E.

You will now see a rather intimidating dozen or so lines, as shown in the screenshot, below. These are commands for *Grub*, but towards the end you should see a line that starts with **linux** – this is the kernel command line. On Ubuntu 14.04 (installed using GPT and EFI) it looks like this:

```
linux        /boot/vmlinuz-3.13.0-32-generic.efi.signed
root=UUID=<long UUID> ro quiet splash
```

Yours may differ in detail, but immediately after **linux** you will see the location of the file containing the kernel that will be used. After that is our first kernel parameter, **root**. This is crucial. It specifies the device that contains the root filesystem; in this case it's specified by a UUID. It's still common to see something like **root=/dev/sda2**, which means the second partition (**2**) on the first disk (**a**). Then we have **ro**, which means that the filesystem should be mounted read-only so that disk checks can be reliably performed – it will be remounted **rw** or read-write later on. Next we have **quiet**, which tells the kernel not to output verbose text to the console, and **splash** enables a pretty graphic screen during booting.

As a safe and informative experiment, try deleting **quiet** and **splash**, then hit F10 to boot the system and you'll see lots of kernel messages spewed onto the console. Ordinarily this isn't all that useful, except perhaps if you twitch your unblinking eyes very fast and say "interesting" to fool an onlooker that you can read as fast as Data from *Star Trek*. Of course, these messages can yield vital clues if the boot process is getting held up, or stops altogether. One common problem that can be spotted this way is if the partition or disk with the root filesystem isn't found where it should be. Correcting the setting of the **root** kernel parameter can fix it, although it could just be that that the drive isn't plugged in, or has an unsupported filesystem. If you've ruled out all those causes and you're dealing with a USB-connected drive, then it's possible it might not have "settled" by the time the kernel starts looking for it. To avoid this problem you can add **rootdelay=10**, which tells the kernel to delay

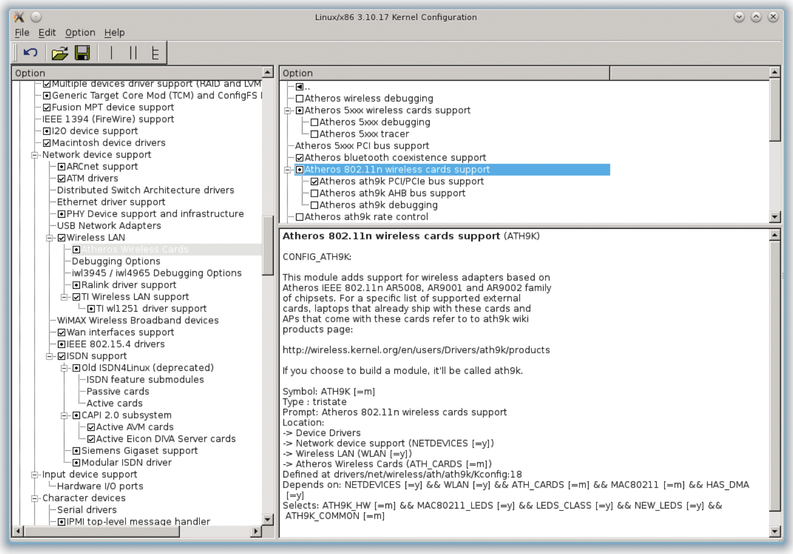


```
                         GNU GRUB  version 2.02~beta2-9ubuntu1

setparams 'Ubuntu'

        recordfail
        load_video
        gfxmode $linux_gfx_mode
        insmod gzio
        insmod part_gpt
        insmod ext2
        set root='hd0,gpt2'
        if [ x$feature_platform_search_hint = xy ]; then
          search --no-floppy --fs-uuid --set=root --hint-bios=hd0,gpt2 --hint-efi=hd0,gpt2 --hint-baremetal=ahci0,gpt2  \
208bee8d-d702-42c1-beb1-ac744c96511c
        else
          search --no-floppy --fs-uuid --set=root 208bee8d-d702-42c1-beb1-ac744c96511c
        fi
        linux        /boot/vmlinuz-3.13.0-32-generic.efi.signed root=UUID=208bee8d-d702-42c1-beb1-ac744c96511c ro  quiet\
 splash $vt_handoff
        initrd        /boot/initrd.img-3.13.0-32-generic
```

Press the E key at boot time to bring up this *Emacs*-like editor, which lets you set kernel parameters in *Grub*.

Graphical kernel configuration

If you're uncomfortable on the command line and just want to learn more about the kernel, there is a powerful yet simple GUI application that lets you explore the kernel and its configuration. To use it, you first need to download the kernel source from kernel.org, or you can use **apt-get** on Ubuntu or **yum** on RPM distributions to grab it and associated tools – details vary, so best consult the documentation for your distro. Then open a terminal window and **cd** to the top level directory of the kernel source (often **/usr/src/linux**, but it's recommended to copy the whole directory tree to somewhere under your home directory). Now type **make xconfig** and the window shown will appear – you can then lose yourself in the graphical tree of kernel parameter goodness.



```
☑ Multiple devices driver support (RAID and LVM
⊡ Generic Target Core Mod (TCM) and ConfigFS I
☑ Fusion MPT device support
IEEE 1394 (FireWire) support
⊡ I2O device support
☑ Macintosh device drivers
Network device support
    ⊡ ARCnet support
    ☑ ATM drivers
    Distributed Switch Architecture drivers
    Ethernet driver support
    ⊡ PHY Device support and infrastructure
    USB Network Adapters
    ☑ Wireless LAN
        ⊡ Atheros Wireless Cards
        Debugging Options
        iwl3945 / iwl4965 Debugging Options
        ⊡ Ralink driver support
        ☑ TI Wireless LAN support
            ⊡ TI wl1251 driver support
    WiMAX Wireless Broadband devices
    ☑ Wan interfaces support
    ⊡ IEEE 802.15.4 drivers
    ☑ ISDN support
        ⊡ Old ISDN4Linux (deprecated)
            ISDN feature submodules
            Passive cards
            Active cards
        ⊡ CAPI 2.0 subsystem
            ☑ Active AVM cards
            ☑ Active Eicon DIVA Server cards
        ⊡ Siemens Gigaset support
        ⊡ Modular ISDN driver
Input device support
    Hardware I/O ports
Character devices
    Serial drivers
    ⊡ IPMI top-level message handler
```

```
..
☐ Atheros wireless debugging
⊡ Atheros 5xxx wireless cards support
    ☐ Atheros 5xxx debugging
    ☐ Atheros 5xxx tracer
Atheros 5xxx PCI bus support
☑ Atheros bluetooth coexistence support
⊡ Atheros 802.11n wireless cards support
    ☑ Atheros ath9k PCI/PCIe bus support
    ☐ Atheros ath9k AHB bus support
    ☐ Atheros ath9k debugging
☐ Atheros ath9k rate control
```

```
Atheros 802.11n wireless cards support (ATH9K)

CONFIG_ATH9K:

This module adds support for wireless adapters based on
Atheros IEEE 802.11n AR5008, AR9001 and AR9002 family
of chipsets. For a specific list of supported external
cards, laptops that already ship with these cards and
APs that come with these cards refer to to ath9k wiki
products page:

http://wireless.kernel.org/en/users/Drivers/ath9k/products

If you choose to build a module, it'll be called ath9k.

Symbol: ATH9K [=m]
Type : tristate
Prompt: Atheros 802.11n wireless cards support
Location:
-> Device Drivers
-> Network device support (NETDEVICES [=y])
-> Wireless LAN (WLAN [=y])
-> Atheros Wireless Cards (ATH_CARDS [=m])
Defined at drivers/net/wireless/ath/ath9k/Kconfig:18
Depends on: NETDEVICES [=y] && WLAN [=y] && ATH_CARDS [=m] && MAC80211 [=m] && HAS_DMA
 [=y]
Selects: ATH9K_HW [=m] && MAC80211_LEDS [=y] && LEDS_CLASS [=y] && NEW_LEDS [=y] &&
 ATH9K_COMMON [=m]
```

Xconfig uses the *Qt* toolkit to give a graphical overview of your Linux kernel.

10 seconds before mounting the root filesystem. This can be especially handy with a Raspberry Pi if you want to use a large external hard drive to contain a root filesystem that won't fit on an SD card.

Kernel, bread and butter

A helpful analogy is to compare the kernel to bread. A good loaf is baked to a precise recipe. The recipe for the kernel is its configuration, which specifies what hardware the kernel supports, eg types of x86 or ARM CPUs, and other things like what filesystems it can work with, such as ext4 or btrfs.

"Setting module parameters can breathe new life into non-functioning hardware."

To see the configuration of the kernel you're currently running, type:

```
zcat /proc/config.gz | less
```

Anything that's set to **=y** means yes, that feature is enabled and/or built into the kernel. For example, the first few lines on the laptop I'm writing this on read are:

```
CONFIG_64BIT=y
CONFIG_X86_64=y
CONFIG_X86=y
```

which tells me I can run both 32- and 64-bit x86 code.

Once the configuration is decided, the next step is to compile it, which is a bit like baking the bread. Both take a while and involve much heat, which for the kernel is because compilation is CPU-intensive.

Setting module parameters can breathe new life into non-functioning hardware, as we saw with **memmap**, but it can also help work around buggy drivers. I was recently dismayed to discover that my shiny new Dell XPS 13, shipped with Ubuntu (reviewed in LV002), kept dropping its wireless connection. It seemed that, although Dell had included the latest drivers for the wireless chipset, they weren't entirely bug-free. But setting just one module parameter fixed the issue, and saved me a good deal of hair loss.

Before going further, let's take another look at kernel configuration. You may have noticed some lines like this that end in **=m**:

```
CONFIG_EXT4_FS=m
```

where **m** doesn't stand for "maybe" (although that'd be accurate), but "module". This means that the feature, in this case support for the ext4 filesystem, is not built into the kernel, but as a module that can be loaded if needed. When the distro maintainer is compiling the kernel, they can't know which filesystems you will use. So instead of building many filesystems into the kernel, bloating it with code that won't be used, support for different filesystems are built into separate modules, which can be loaded as needed.

If you know exactly what your kernel will be used to do and on what hardware it is going to run, such as a smart TV, then you can build just what is needed into it, keeping it small and simple, and not have any modules. The bread analogy is still applicable. You can use the butter "module" with plain bread for your morning toast, and you can use the same loaf with cheese and tomato as "modules" to build a sandwich for lunch. Alternatively, you could choose to bake cheese and tomato into a loaf, but it would then only be useful for certain meals.

Mess with a running system!

Another advantage of modules is that, unlike the **memmap** example, you can still set kernel parameters after the system has booted up. To display information about a module, use the **modinfo** command, as explained in the boxout, left. You can list the modules currently in use with **lsmod** – both commands need to be run as root or with sudo.

Going back to my laptop's issue with dodgy wireless, I did some searching and discovered that the module was buggy when dealing with hardware encryption, and that loading the module with **hwcrypt=0** would solve problems with the wireless dropping out. Before you can do that, you need to find the name of the module that provides the driver for your wireless chipset. For a USB device, this can be done by looking through the output from:

```
usb-devices | less
```

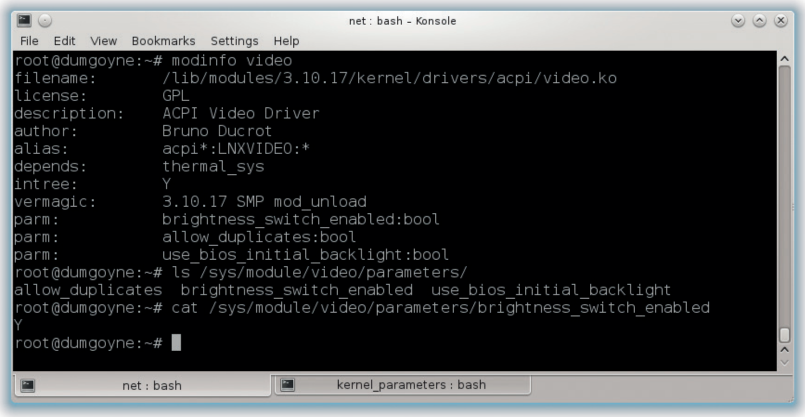
You should see a block of information for your device with some human readable text description like "Wireless networking", and at the end of the last line you will see the kernel module in use after **Driver=**. If your wireless chip is not USB connected, it will be on the PCI bus, which requires two steps to identify it. First locate the wireless chipset with:

```
sudo lspci | grep -i network
```

For me this gave a long line that started with **01:00.0**, which I could then use to display more verbose information with this command:

Module parameters

The command **modinfo video** is a great example of how to show information about a module, in this case the **video** module. Of most interest here are the three parameters in the **parm:** lines. You can discover their current settings by looking at files in the **/sys/module/video** directory, as shown for **brightness_switch_enabled**. We can discover what the parameter does by consulting the list on **kernel.org** (**www.kernel.org/doc/Documentation/kernel-parameters.txt**): "If set to 1 [or Y], on receiving an ACPI notify event generated by hotkey, video driver will adjust brightness level and then send out the event to user space through the allocated input device; If set to 0, video driver will only send out the event without touching backlight brightness level."



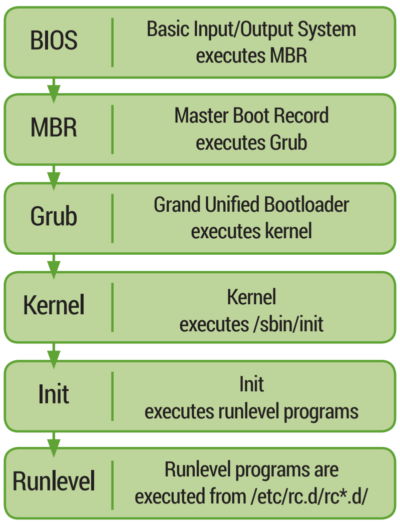
```
root@dumgoyne:~# modinfo video
filename:       /lib/modules/3.10.17/kernel/drivers/acpi/video.ko
license:        GPL
description:    ACPI Video Driver
author:         Bruno Ducrot
alias:          acpi*:LNXVIDEO:*
depends:        thermal_sys
intree:         Y
vermagic:       3.10.17 SMP mod_unload
parm:           brightness_switch_enabled:bool
parm:           allow_duplicates:bool
parm:           use_bios_initial_backlight:bool
root@dumgoyne:~# ls /sys/module/video/parameters/
allow_duplicates  brightness_switch_enabled  use_bios_initial_backlight
root@dumgoyne:~# cat /sys/module/video/parameters/brightness_switch_enabled
Y
root@dumgoyne:~#
```

Bootloaders

After you power up a computer, the onboard firmware will hunt for some code to run. The old-fashioned BIOS will look to the MBR (Master Boot Record) at the start of any disks it can find, and tell the CPU to run any suitable code that is found. These days UEFI has replaced the BIOS and MBR system, and does a similar job but with more features and fewer limitations.

The most common bootloader is *Grub*, but others such as *Lilo*, *Syslinux* and *Gummiboot* all do much the same job in slightly different ways. You can think of a bootloader as a temporary, mini operating system (see MikeOS and its excellent documentation) that is needed only in the early stages of a computer starting up. The final job of any bootloader is to load the kernel with its command line parameters and also to tell it what to run as its first process – these days it will be **systemd**, but some distros still use **init**.

The Raspberry Pi is worth a mention because it doesn't use a normal bootloader. It starts with its ARM CPU disabled; the GPU runs firmware code and looks only at the SD card and runs the file **/boot/bootcode.bin**. If you want to alter kernel parameters at boot time, you can add them to the **cmdline=** line in the **/boot/config.txt** file, or in the file **/boot/cmdline.txt**.


BIOS
Basic Input/Output System executes MBR
MBR
Master Boot Record executes Grub
Grub
Grand Unified Bootloader executes kernel
Kernel
Kernel executes /sbin/init
Init
Init executes runlevel programs
Runlevel
Runlevel programs are executed from /etc/rc.d/rc*.d/

```
sudo lspci -v -s 01:00.0
```

On the last line of my output was the important information **Kernel modules: ath9k**. (If no kernel module is listed, then you might need to load it manually; of which, more later).

To set a module parameter manually, you'll first need to unload the module. Warning: doing this could cause serious problems. It's safe enough to do with the module controlling a wireless chipset (as long as you don't mind losing wireless for a bit), but unloading a module for the root filesystem is asking for trouble! To unload a module, in this case my **ath9k** module, just do:

```
sudo modprobe -r ath9k
```

then to reload it and set the **nohwcrypt** parameter, just enter this line:

```
sudo modprobe ath9k nohwcrypt=1
```

and that's it. This would need to be done every time the laptop is started up, but you can make it permanent by creating a file in **/etc/modprobe.d**. The name of the file is up to you, but something descriptive like **ath9k_myfix.conf** would be appropriate, and it need only contain:

```
options ath9k nohwcrypt=1
```

Remember, this only works if **ath9k** was compiled as a kernel module, which will usually be the case, but if it were compiled into the kernel (**y** instead of **m** in the kernel config), then you can still set it at boot up by adding **ath9k.nohwcrypt=1** to end of the kernel command line.

There's more you can do with the conf files. Sometimes hardware is misidentified and the wrong kernel module is loaded. For example, say you notice your wireless is not working and then you notice that the **ath9k_htc** module is loaded, but you know your hardware is not made by HTC. The solution would be to **blacklist** the offending module, which you can do by creating the file **/etc/modprobe.d/ath9k_myfix.conf**, but this time it has just this line:

```
blacklist ath9k_htc
```

or you might prefer to add that line to an existing **blacklist.conf** file. This will stop that incorrect module from loading, and hopefully you'll find **ath9k** is loaded instead.

Sometimes it's still necessary to force the loading of a module. To do this, you can create a file in **/etc/modules-load.d**. For example, in order for the CUPS printing system to work, Ubuntu 14.04 comes with **cups-filters.conf**, which contains the following lines to load three printing related modules:

```
lp
ppdev
parport_pc
```

It is possible to set some kernel parameters on a running kernel even if they're not part of a module. A useful example is the **swappiness** parameter, which controls how swap space is used. To change it, you edit a file (actually it's not a real file, but a virtual one generated by the kernel) with something like:

```
sudo nano /proc/sys/vm/swappiness
```

The file will contain the current setting, which will probably be the default of 60. Set it to 100 and the kernel will swap from memory to disk aggressively; set it to 0 and you may well notice a speed boost, but at the risk of problems if you run short on memory.

Further reading

The definitive source of information on kernel parameters is **www.kernel.org/doc/Documentation/kernel-parameters.txt**. Ubuntu's documentation on kernel parameters is also worth reading: **https://wiki.ubuntu.com/Kernel/KernelBootParameters**, as is the Arch Linux wiki: **https://wiki.archlinux.org/index.php/Kernel_parameters**. For a more complete overview of the subject, though a little out of date now, is *Linux Kernel in a Nutshell* by kernel developer Greg Kroah-Hartman, available as a free download on his website **www.kroah.com/lkn** or in print from **oreilly.com.**