Building a Super Kernel for Data Forensics

Linux Kernel Customization

by Thomas Rude, CISSP

aka, farmerdude

March 2002

There are two goals for this paper. One, I hope to enlighten you as to how to build a custom kernel for your Linux system (the steps required). Two, I have included in detail options that I believe are required for building a kernel specific for data forensics (Super Kernel). Not all options are covered, nor are all options that I have chosen necessarily the ones you will choose. I have tried to omit the specific hardware options (the hardware your base system is running on), but include the options for general hardware devices (such as firewire, parallel port, and usb devices) that are useful in processing data forensics. I have also chosen to omit other specific-to-your-system-and-environment options such as general networking configurations (firewall/vpn support, logging, etc.) though I have included options for Network File Systems support and both IPX and Appletalk protocols (used in Novell and Apple networks respectively).

I will walk you through creating a boot disk, updating to a newer kernel, and customizing and compiling the new Super Kernel in this paper. If you find an error, or have feedback, please contact me. Here's hoping you find something useful here!

Assumptions

Please note: there are many flavors (distributions) of Linux. For the purpose of this paper I am going to use the following software and hardware:

- -Dell Latitude CPx Laptop (Pentium III 500MHz Processor)
- -Maxtor 60GB External FireWire Hard Drive
- -SIIG PCMCIA 2-Port Cardbus-to-1394 FireWire adapter card
- -MicroSolutions Backpack cd-rewriter
- -Red Hat Linux 7.2, kernel 2.4.7-10

Okay, what's the significance of these assumptions? Well, the only assumption that is really key here is the kernel version (2.4.7-10). Different kernel versions may have different options available during customization. The 2.4.x series kernels should not have much difference, but to be on the safe side I want to state the version for this paper. If you are running a stock Red Hat Linux 7.2 system then the images within this paper will match exactly to what you see.

The devices are listed because the focus of this paper is building a kernel for data forensics. In much of my work I use firewire and parallel port devices. The firewire hard drives are great for dumping images. The Backpack cd-rewriter is great for burning off data. By including support for these devices you should be able to apply the steps to your specific system and devices.

Now, before we go on, we should make sure we have a basic understanding of key elementary components of a Linux system:

kernel = the Linux kernel can be thought of as the brains of the system. It is the core operating system and it is a compressed file.

kernel-headers = the kernel headers are the actual C header files for the kernel.

kernel-source = the source code for the Linux kernel.

kernel-pcmcia-cs = the PCMCIA card services required for laptops.

modules = modules can be anything (device drivers, executables, etc.). They are dynamically loaded and unloaded as needed.

Overview

Okay, now it's time to review what we are going to do and in what order;

- I) Make Emergency Boot Disk
- II) Test Emergency Boot Disk
- III) Query System for Required Packages
- IV) Update Kernel Packages
- V) Customize and Compile the Kernel
- VI) Reboot with New Data Forensics Super Kernel

You ready? Alright, let's do it! Remember, commands are in bold and are indented!

I. Make Emergency Boot Disk

Mission Critical! Before we even begin to dig into the customization we must make an emergency boot disk. Ever build a kernel only to reboot and have the boot process hang? Ah, me too. Not a fun day at the beach, is it? One thing I have learned is to create these boot disks religiously! We'll save ourselves plenty of aggravation in the long run. I know, you've heard it all before. But trust me, make the boot disk and *test* it.

In order to use the 'mkbootdisk' command we must know the kernel version. To find out the running kernel version for our Linux

system we can issue:

```
uname -r
returns '2.4.7-10'
```

Cool. This tells us our kernel version is 2.4.7-10. Now we are ready to make a boot diskette by issuing:

```
mkbootdisk 2.4.7-10
```

II. Test Emergency Boot Disk

Ah . . . the afterthought. Ever make a boot disk but not test it? Ever feel like a schmuck when you *need* that boot disk only to find out it does *not* work? We're going to save some future grief now. Time to shutdown and boot up with our newly created boot disk:

```
poweroff
```

After successfully booting the system with the emergency boot disk we're set to move on. Obviously if we can't boot using the boot disk we need to reboot and create another boot disk (hopefully successfully this time!).

III. Query System for Required Packages

Before we grab any new packages and install them we must know whether or not our current system is ready for recompilation. We can find this out by querying for the following packages; kernel-source*, kernel-headers*, mkinitrd* (for SCSI devices), kernel-pcmcia* (for laptop PCMCIA) and kernel-2.4*.

We'll use the ever-powerful rpm command to query for these packages:

```
rpm -qa | grep mkinitrd*
returns 'mkinitrd-3.2.6-1'
```

Cool. We have the mkinitrd package installed, and this is good. Because we need it. If you're primary master hard drive is of SCSI type then you will need this package as well. 'mkinitrd' creates a RAMDISK to allow for loading of modules that are needed in order to boot.

```
rpm -qa | grep kernel*
returns 'kernel-pcmcia-cs-3.1.27-10
        kernel-2.4.7-10
        kernel-source-2.4.7-10
        kernel-headers-2.4.7-10
        kernel-doc-2.4.7-10'
```

What we have done above is query (q) all (a) of our installed RPMs, and then piping that output to the grep command, specifying anything leading with kernel*. Our output shows we are golden - we have the kernel headers and source already installed. In order to compile a custom kernel both of these packages are required.
NOTE: you do not need either the kernel-headers or kernel-source packages in order to upgrade or install a kernel. They are required for compiling, though.

If we simply wanted to recompile the kernel we could do that now. But, why don't we update our kernel to a newer version first?

IV. Update Kernel Packages

Warning: updating a kernel may be hazardous to your sanity! The beauty of Open Source is that new code is being written on a continuous basis. The darker side of this is that as new code is applied things that were working before may now be broken in the new code. This is the gamble we take when updating kernels. Because of this, it is always wise to *wait* a bit before applying an update. Giving a week or two for the new code to work its way around will allow for feedback to be posted - good and bad. For this reason alone I recommend reviewing the archives over at KERNEL.ORG. Further, aside from waiting, do make sure you update/install the kernel on a non-production system (I.E., do it on your test platform first!) before attempting on your production system.

Okay, we have two options here. We can *install* a new kernel and respective kernel packages or we can *upgrade* to a new kernel. Is there a difference? You bet!

Upgrade will do just that - upgrade. Think of it as overwriting if that helps. When we upgrade there's no turning back. The old version is gone and the new is the one and only. A simple analogy could be the following: let's say we have a cellular phone plan for 400 minutes per month for $29.99. We decide to upgrade our plan to 800 minutes per month for $38.99. At the end of the day after the upgrade we have only *one* plan we can use - the $38.99 800 minute plan.

Install will do just that - install. Think of this as installing alongside of if that helps. With this there is an option to turn back. If we install we'll have both kernels installed. A simple analogy may be: let's say we are using the 'Golden Blue Dream Plan' for our cellular plan. The Golden Blue Dream Plan allows for 100 minutes per month. We think we need more minutes, but are not quite sure. So we decide to buy a second plan, the 'Flying Garlic Clove Plan' which allows for 999 minutes per month. At the end of the day we now have two plans from which we may use. If we decide we do not need the extra minutes we can fall back on our original plan, Golden Blue Dream Plan.

Shifting this to the bit bucket, let's take a gander at our system; we have a stock Red Hat Linux 7.2 system, running the stock 2.4.7-10 kernel. We want to *update* to the 2.4.9-13 kernel update released by Red Hat. Our only decision to make is to upgrade or to install? Anyone? Anyone? Bueller?

Okay, this is just my personal methodology, so your's may differ. But for the sake of this example let us *install* the new kernel updates. This will allow us to fall back on the current, working system if the newly updated one fails for some reason. (Normally I install the kernel package and upgrade the kernel-headers, kernel-source, and kernel-doc packages. (Keeping a backup of these packages handy, of course!)). If disk space is of no concern then perhaps you will simply install each package instead of upgrading (upgrading saves space).

In order to update our packages we must download the updates from a reliable source. Since this is a Red Hat Linux system I would prefer to download the updates from the official Red Hat Errata web site. Now that we have downloaded and verified them we are ready to install them:

```
rpm -ivh kernel-2.4.9-13.rpm
rpm -ivh kernel-headers-2.4.9-13.rpm
rpm -ivh kernel-source-2.4.9-13.rpm
rpm -ivh kernel-doc-2.4.9-13.rpm
```

We see from the Red Hat Errata site that our system is current for both kernel-pcmcia-cs* and mkinitrd* packages (so we do not need to update these two packages).

V. Customize and Compile the Kernel

And the entree is served! We have our emergency boot disk, we've downloaded and installed the necessary packages, and now we are good to go. It's kernel time!

There are a number of steps we must undertake in order to customize, build, and install our new kernel. We will go through each step, in order, and in detail. This might be a good point to take a moment to pause, stretch your legs, let the dog out, and grab another beverage of choice.

Ready to go? If you don't have the X Window System running let's start it up via:

```
startx
```

For the purposes of this paper we're going to use the GUI X-Window System to choose our options. I think there are two methods everyone should become familiar with when choosing options for the kernel; one is this GUI based method and the other the interactive 'menuconfig' method. (There will be a time when you will not have X running on the system for whatever reason! Not to mention this process will be quicker from the command line versus the X-Window System GUI).

Now we will open a terminal and navigate to the required directory to begin our work:

```
cd /usr/src/linux-2.4/
```

I cannot stress how *critical* this step is. We must issue the commands from the /usr/src/linux-2.4/ directory.

Now let's see what is in this directory:

```
ls -la
```

Why do we list out all the files in this directory? Simple; if there is a '.config' file here it might be a wise choice to copy this file to another directory for later reference. If we've not compiled the kernel yet for the first time this '.config' file is a generic configuration file that is a good base kernel. *However*, if we have previously compiled the kernel then this '.config' file will be the most recent configuration we compiled.
As I sit and type this on my laptop I have 3 kernels to choose from upon boot up. Needless to say, when you have a configuration file you like it is good to copy it and save it. So let us pretend this '.config' file is a keeper and so we will copy it:

```
cp -v .config /home/Cust_Kerns/
```

Our next step is to edit the 'Makefile' file. We do this for a very import reason. To keep from overwriting the current kernel! Sounds like a good thing to do, doesn't it? There is a line in this file which we will append to. It is very important to remember what we append as well, as we will need this same exact information later in the kernel recompilation process.
To take a peek at the 'Makefile' file we issue:

```
more Makefile
```

Notice near the beginning of this file there is a line that begins with 'Extraversion='. This is the line we want to modify. Let's say the line currently exists as such:

```
Extraversion = -13custom
```

Let's go ahead and modify this line. I'll use the *pico* editor but any editor will do:

```
pico Makefile
```

Pico is an editor that is rather intuitive to use. (Much more so than Vi) We simply navigate to the line to modify and we'll change it to reflect;

```
Extraversion = -13farmerdude
```

What we have done is changed the kernel we will be customizing and building to '-13farmerdude' from -13custom (or, 2.4.9-13farmerdude to be precise). This way, if there exists a -13custom kernel we will not overwrite it! NOTE: There may not be a -13custom kernel. However, it's good practice to view the 'Extraversion =' line to see what will be built, and to modify it accordingly to avoid overwriting a kernel you wish to keep.

We're almost there. To be anal and on the safe side we will clean out the source tree (and this is when the '.config' file(s) will be removed from the /usr/src/linux-2.4/ directory) by issuing the following:

```
make mrproper
```

NOTE: this will remove ALL '.config' files from this directory! Including those such as '.config_working' and '.configSave'. You see what happens. If the file starts with '.config' it will be wiped out with this command. You have been warned.

Now, what if we had a .config file we liked and saved to another directory (such as we did above), can we copy it back now? The answer is yes. After 'mrproper' has completed it is safe to copy a .config file back to this directory, and in fact, we will do that now as we want to use our saved .config file as a baseline for our new kernel:

```
cp -v /home/Cust_Kerns/.config /usr/src/linux-2.4/
```

Bored yet? Hope not - this is where we actually begin to select the options for our kernel. When choosing how to select the components for our kernel we have a number of options available to us;
'make config'
'make menuconfig'
'make oldconfig'
'make xconfig'

Obviously there may be a time and place for each of these. However, there are only two that I ever really use regularly; 'make menuconfig' and 'make xconfig'. I opt for 'menuconfig' when I either have a shell and no X-Windows available or to save time. And I opt for 'xconfig' when I have the GUI available and speed is not critical. (Though with today's processing power recompiliation via the X-Windows GUI is pretty fast!)

'make config' is text based. However, be careful. You cannot change an option once set!

'make menuconfig' is also text based, but you can go back and change options. I like that.

'make oldconfig' is a generic configuration file. This is not interactive. It is a script that will build a clean .config file such as the one that was created when you first installed the operating system. Perhaps if you get in a jam and really muck things up you can revert to this.

'make xconfig' is X-Windows based, and a nicely laid out GUI menu from which to select options.

Just as I believe it is good practice to perform both text and GUI installations of the operating system I also believe that it is good practice to perform both text and GUI kernel recompilations. If nothing else it just makes you more comfortable with the command line.

Now, having said that, for our example we'll opt for the easy way and issue:

```
make xconfig
```

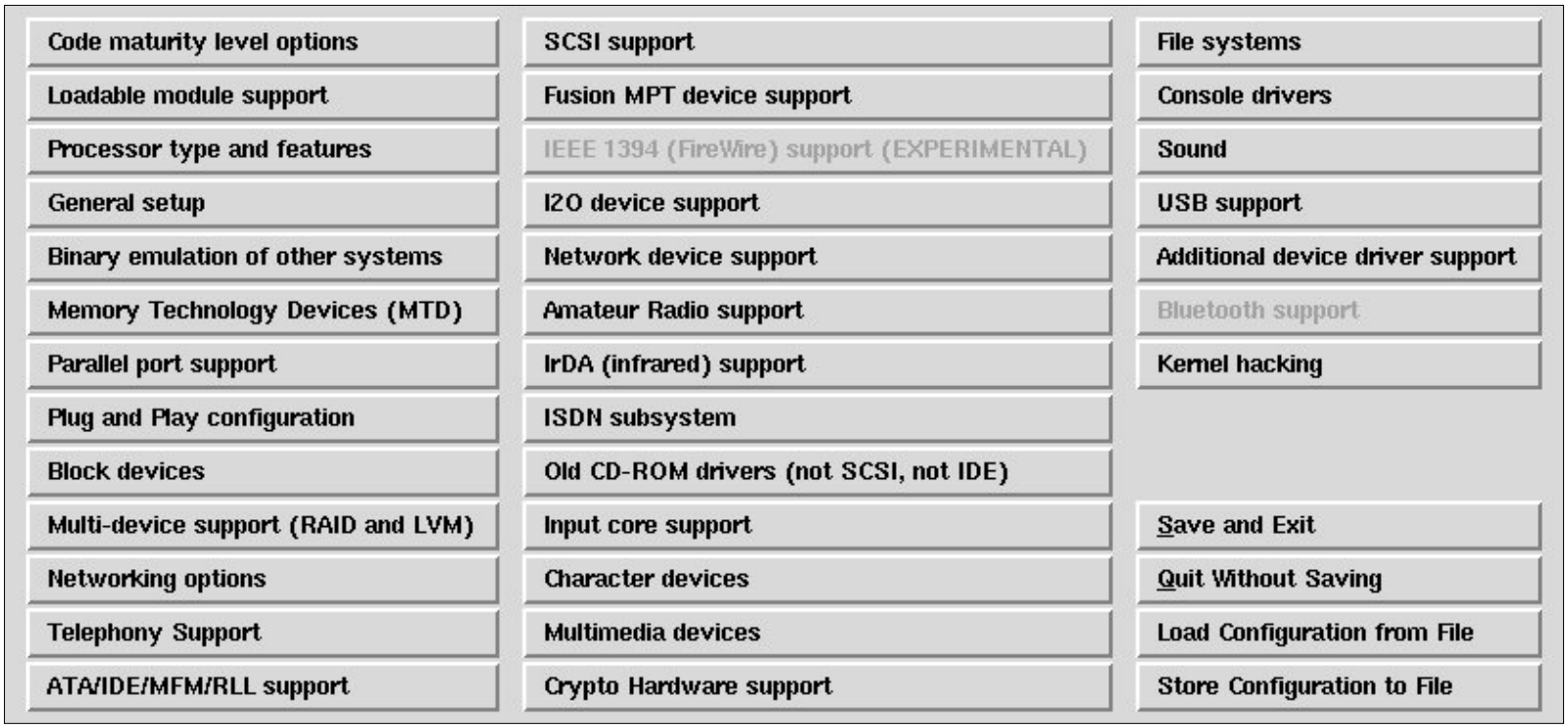

Image 1: Kernel Categories Menu via 'make xconfig'

I think most of the categories are pretty self explanatory. They may seem a bit overwhelming for someone new to Linux (or simply new to compiling a custom kernel). However, like anything, practice and more practice leads to familiarity. Build a custom kernel a dozen times or so and you won't remember when you couldn't do it!

There is something I would like to point out with reference to *Image 1* above; notice how there are two categories dulled/greyed out? There is a reason for this. I will try to keep the explanation simple.
These two are turned off by default and we must recompile a custom kernel to enable them. How's that?!!? (There is actually one key setting we must enable that will allow for these to become active so that we may select and configure them. We'll cover this setting soon enough.)

Notice, too, the four options in the bottom right of the menu; Save and Exit, Quit without Saving, Load Configuration from File, and Store Configuration to File.

'Save and Exit' is what we will choose once we have completed setting our options. This saves the settings to the .config file in the /usr/src/linux-2.4 directory.

'Quit without Saving' is self explanatory. Any changes or settings will be lost. If you need to abort the process for whatever reason you can kill it here and none of the changes to settings will be recorded in the .config file.

'Load Configuration from File' again, probably self explanatory. By clicking here a window pops up wherein we can put the path to the .config file we want to load. Say, for our example that would be the /usr/src/linux-2.4/.config file we saved and then copied back to this directory after 'make mrproper'.

'Store Configuration to File' Okay, so we want to save these settings? We can click here and save what changes we have made to a .config file.

Remember the point of this paper?!?! To build a kernel specifically for data forensics? Good. Well, that being the case, I am not going to go through every category here. I will hit the categories that I use to build a forensics platform. I will work left to right, top to bottom from the categories present in *Image 1*. This is a good way to select the options. A reason being is that many options are determined by selections made previously (I.E., by checking 'y' to Code Maturity we can now select FireWire options). It is in your best interest to sit back, relax, and go through every category and read about each option (by click on 'Help'). This will take some time. But, remember, as you become more familiar you'll know more by default. And, hence, you won't have to hit every category and/or option.

CODE MATURITY LEVEL OPTIONS

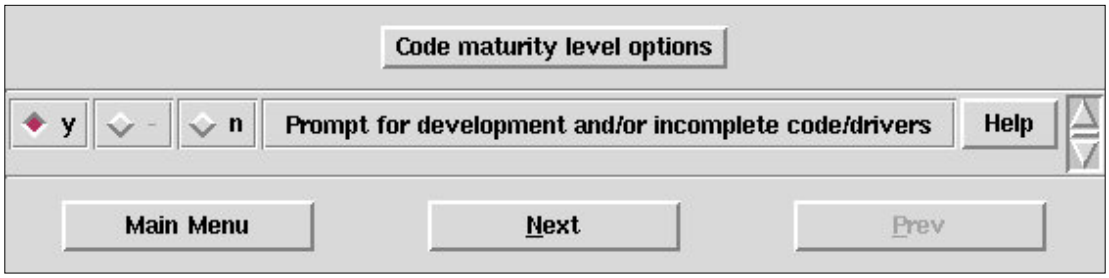

Image 2: farmerdude Code Maturity Level options

By default this is set to no (n), meaning the kernel does not support drivers that are incomplete, super new, etc. This is why both FireWire and Bluetooth are greyed out in *Image 1*. So we want to change this value to:

'y Prompt for development and/or incomplete code/drivers'

By doing this the greyed out categories will become black (active) as well as other options located within each category. This is a very crucial setting to change! Not all drivers are stable, and that is the risk you take, but for data forensics we must enable this and take the risk.

Now is a good time to mention this; for each option that can be set, there may be a total of three (3) possible settings:
y = yes (this will include support directly into the kernel)
m = module (this will include support as a loadable module)
n = no (this will not include support)
Not all three choices will be availabe for each option that we want to configure. Sometimes only a 'y' or 'n' will be available. Other times only a 'm' or 'n' will be the choices.

LOADABLE MODULE SUPPORT

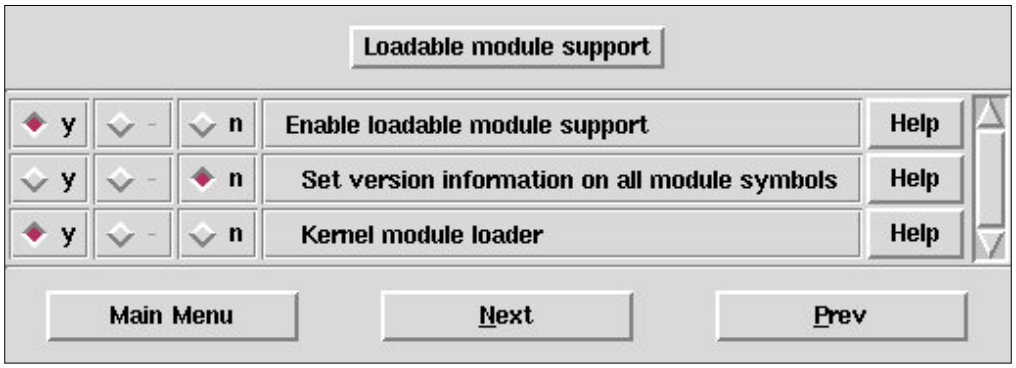

Image 3: farmerdude Loadable Module Support options

What we're looking to add here is support for loadable modules. Building a modular kernel have a few advantages. Namely, the actual kernel will be smaller if there are modules as opposed to a monolithic (no modules) kernel. Further, you may realize you will not always need drivers all the time for every device and/or protocol you use. So let's say we use our Backpack cd-rewriter every other week. We could build that support into the kernel. But, based on the infrequency of use I prefer to include modular support (Which requires me to remember or note module names so that I may insert them when I need them!).

'y Enable loadable module support'
'n Set version information on all module symbols'
'y Kernel module loader'

So, we have enabled modules. Next, we have said *no* to the 'set version' option. If we were to say yes to this we could resuse the modules with new kernels - not a good idea! It is the rule of thumb to compile modules along with your new kernel. Why? To eliminate compatibility options. Lastly, by opting for *yes* for the 'Kernel module loader' we have removed ourselves from having to rely on the somewhat flaky kerneld daemon to load modules.

PARALLEL PORT SUPPORT

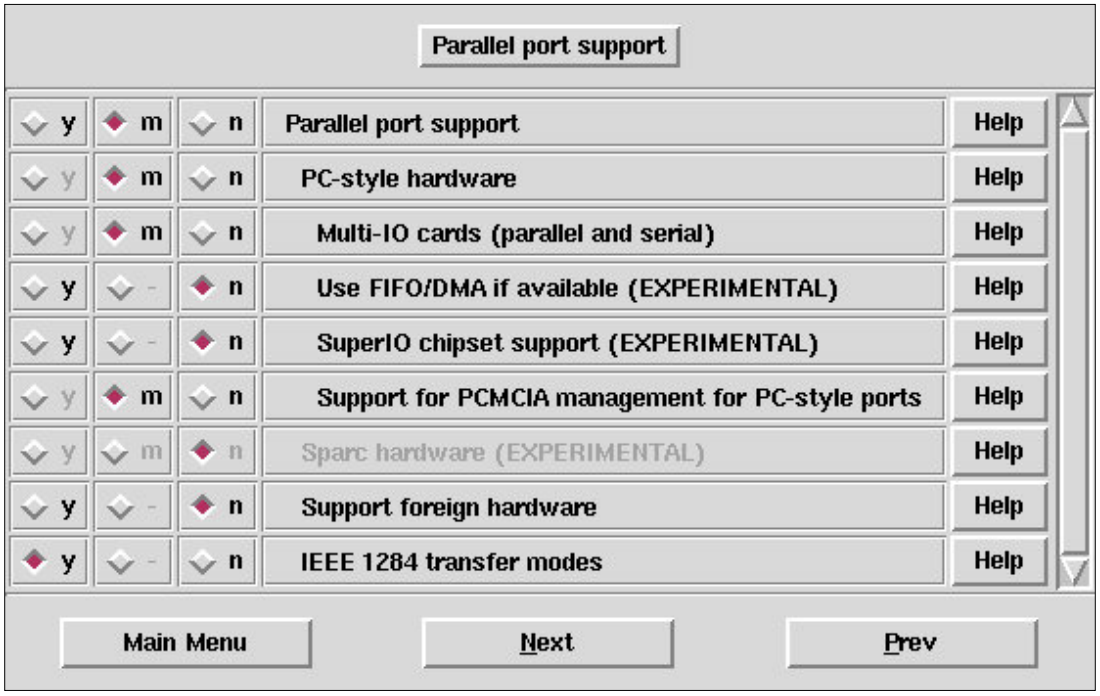

Image 4: farmerdude Parallel Port Support options

I don't use the parallel port a whole lot anymore, but one thing I do use it for is my trusty Backpack cd-rewriter that uses the parallel port connector. As mentioned earlier, the Backpack is not used daily. Therefore we will include parallel port support as a module with options set as:

'm Parallel port support' (parport.o is the module name)
'm PC-style hardware'
'm Multi-IO cards (parallel and serial)'
'm Support for PCMCIA management for PC-style ports'

BLOCK DEVICES

The crem de la crem of our little world! Here is where we can set many options regarding hard drives, cd burners, etc. Remember our Backpack cd-rewriter? Here is where we can add support for this device:

'm Parallel Port IDE device support' (paride.o is the module name)
'm Parallel port IDE disks'
'm Parallel port ATAPI CD-ROMs'
'm Parallel port generic ATAPI devices'
'm MicroSolutions backpack (Series 5) protocol'
'm MicroSolutions backpack (Series 6) protocol' (bpck6.0 is the module name)

The Series 5 protocol is for the older models. Our specific example uses the Series 6 protocol.

There is one more *very critical* option we can set while we are here. All the way down, almost near the bottom, is the Loopback device support. If we intend (and we do!) to mount a file as a block device then here is where we enable support. For this option I prefer to compile support for the Loopback devices directly into the kernel by selecting:

'y Loopback device support'

NETWORKING OPTIONS

I know, I know, you're thinking 'What does networking have to do with data forensics?' Well? Actually, quite a lot, depending, of course. Let's say we need to drop this laptop in a Novell or Macintosh network. Here is where we can add in support to do just this. (And remember, not all data forensics is on stand alone PCs!)

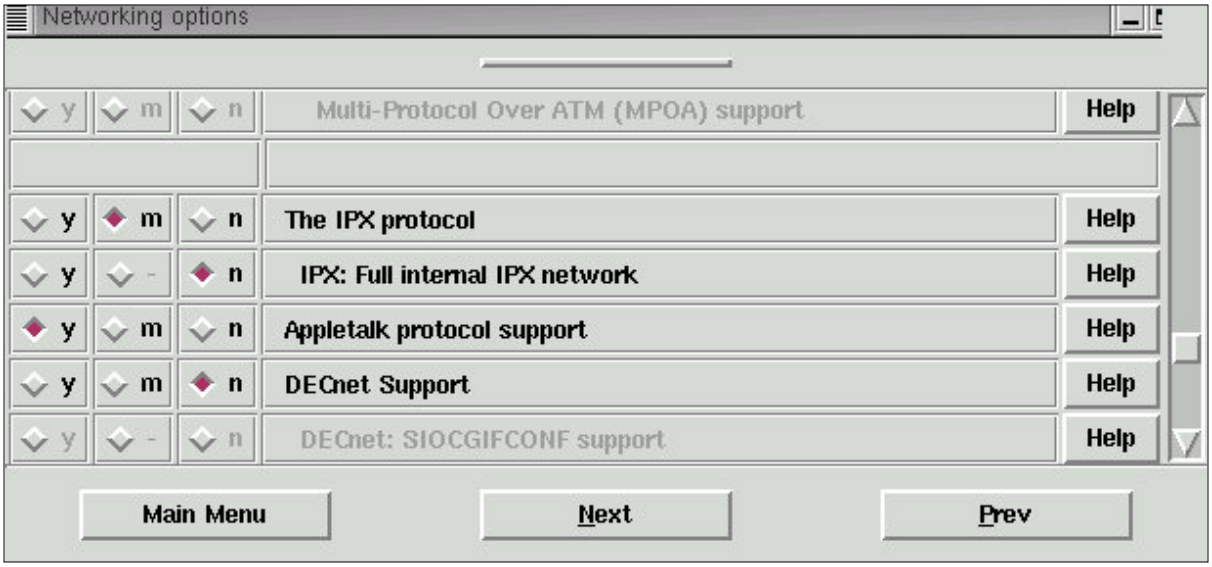

Image 5: farmerdude Networking Options; IPX and Appletalk options

To include support for the native Apple and Novell networking protocols:
'm The IPX protocol' (ipx.o is the module name)
'm Appletalk protocol support' (appletalk.o is the module name)

SCSI SUPPORT

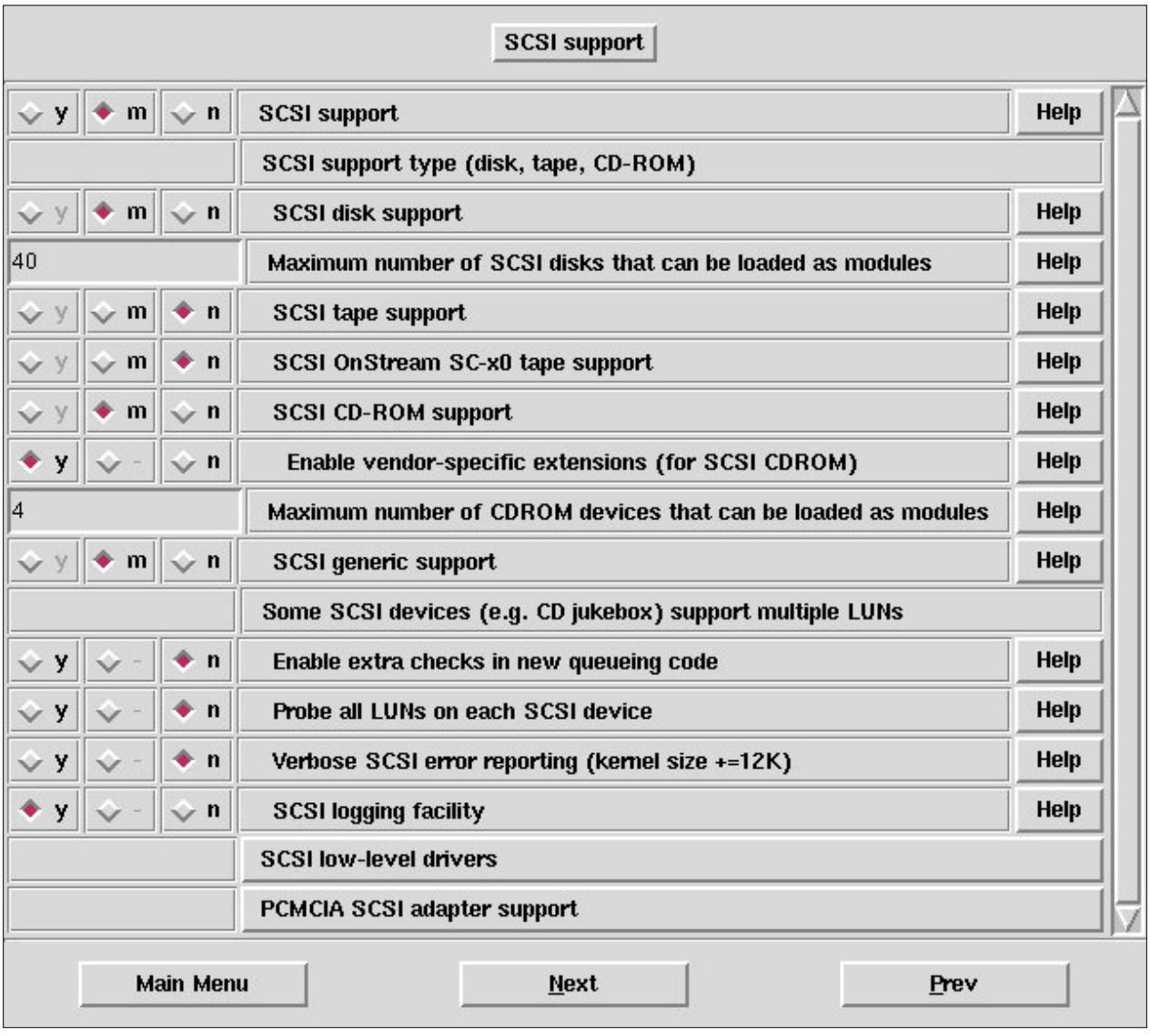

Image 6: farmerdude SCSI support options

SCSI baby. Since this is a laptop with an IDE drive we don't have any SCSI drivers or devices for our hard drive. However, we want to be able to use the external FireWire hard drive, and that is recognized as a SCSI device. Further, perhaps we may acquire future SCSI devices, such as cd burners, etc. Let's take a look at what options we set here:

'm SCSI support' (scsi_mod.o is module name)
'm SCSI disk support' (sd_mod.o is module name)
'm SCSI CD-ROM support'
'y Enable vendor-specific extensions (for SCSI CDROM)' NOTE: I recommend reading the HELP section here.
'm SCSI generic support'
'y SCSI logging facility' You got your eyes open, ghostrider? That is a 'yes' to this selection. It may save you serious hassle later in life. By selecting 'y' here, we have not turned on logging, but merely activated the capability to log. And this is a very wise thing to do. Let's say we attach a SCSI device, only to be faced with some problems (whether it be recognition, data transfer, or whatnot). Since we have enabled SCSI logging we can then *set* the logging and record the error messages for reference in attempting to solve the issue(s).

The last two items are also very important;
SCSI low level drivers - if our primary master hard drive was attached via a SCSI adapter then this is where we would look for compiling support.
PCMCIA SCSI adapter support - selecting 'y' for 'PCMCIA SCSI adapter support' is a good choice.

IEEE 1394 (FireWire) support (EXPERIMENTAL)

FireWire is a great technology enabler for data forensics. Data transfer speeds are quite quick, and the number of firewire devices keeps growing. I personally like the firewire hard drives, though the lack of a cooling fan causes overheating issues at times. To solve this I've placed a rack of firewire hard drives in a mini fridge, running a power cable out the back along with the firewire transfer cable. I've found this to be quite convenient. Not only do the drives not overheat but I've always got a cold beverage nearby as well!

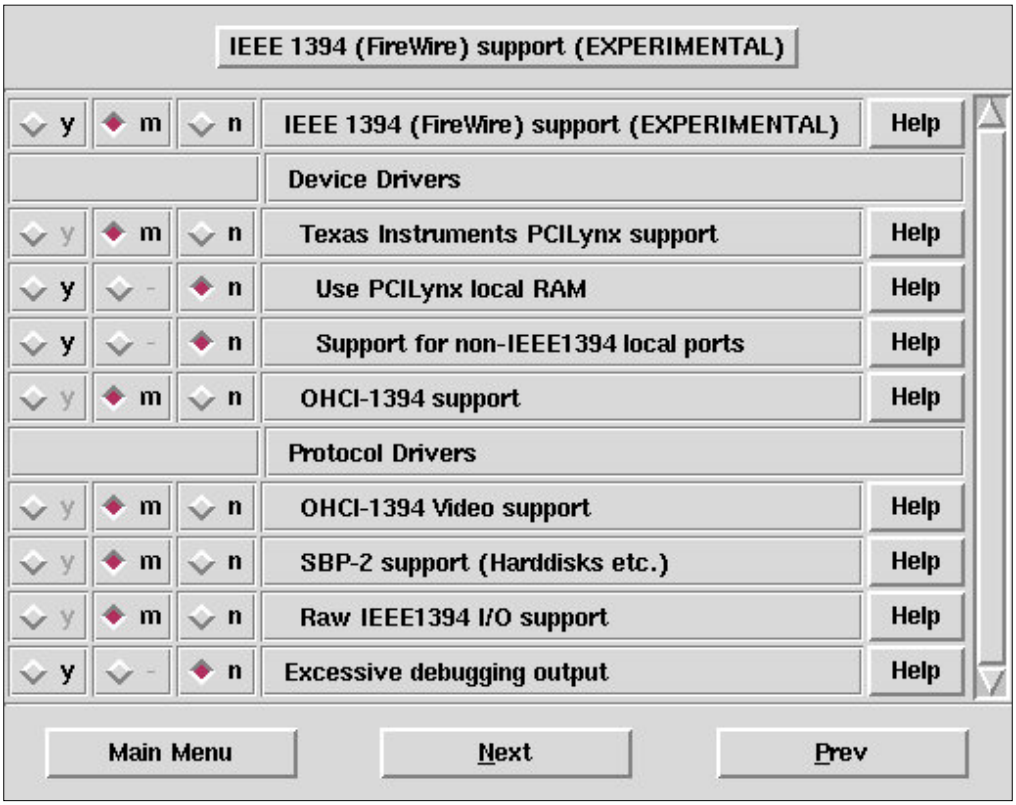

Image 7: farmerdude Firewire support options

Let's take a look at what options we'll set for 1394 support:

'm IEEE 1394 (FireWire) support (EXPERIMENTAL)' (ieee1394.o is module name)
NOTE: I opt to enable support as a module for a couple of reasons; firewire support in Linux is still fairly immature and rapidly improving, and I don't always have a 1394 device attached. Therefore, I opt for modular support here.
'm Texas Instruments PCILynx support' (pcilynx.o is module name)
'm OHCI-1394 support' (ohci1394.o is module name)
NOTE: the two options above are both low level drivers for 1394. Most likely you will have an OHCI type of adapter/card. However, to be on the safe side, let's elect to have modular support for the pcilynx driver as well.
'm SBP-2 support' (sbp2.o is module name) NOTE: we must compile support for sbp2 as a module only! Compiling sbp2 directly into the kernel will cause heartache later on. Modular support for sbp2 is the right way to go.
'm Raw IEEE1394 I/O support'

NETWORK DEVICE SUPPORT

Oh no, not more networking! Yup. Back to more networking options. The main thing here we want to cover is support for the

Appletalk protocol. (Besides choosing options for NIC cards!) So navigating through the Network Device Support options we select 'Appletalk devices'.

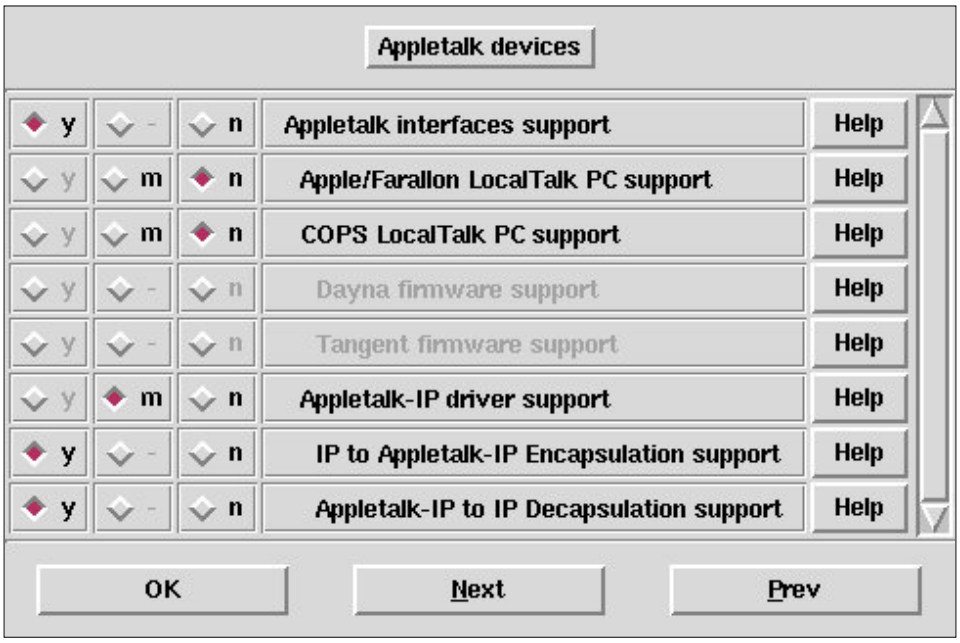

Image 8: farmerdude Appletalk Devices options

And we will select the options as follows:

'y Appletalk interfaces support'
'm Appletalk-IP driver support'
'y IP to Appletalk-IP Encapsulation support'
'y Appletalk-IP to IP Decapsulation support'

FILE SYSTEMS

Show me the money! This is probably the most critical category for building a super kernel for data forensics. After all, having a system that has the ability to recognize and interpret the file system type is the key to being able to successfully analyze an image in the most efficient manner.

Here, like in other categories, please read each HELP section to understand more about the file system in question. I have not included support for every file system type as you will see in the following images. However, you can add in support for those any time you like. The ones I've left out are types that I have not come across yet in my journeys! Lastly, since this is such a large section I have decided to break it down into seven (7) areas. Let's look at the first area:

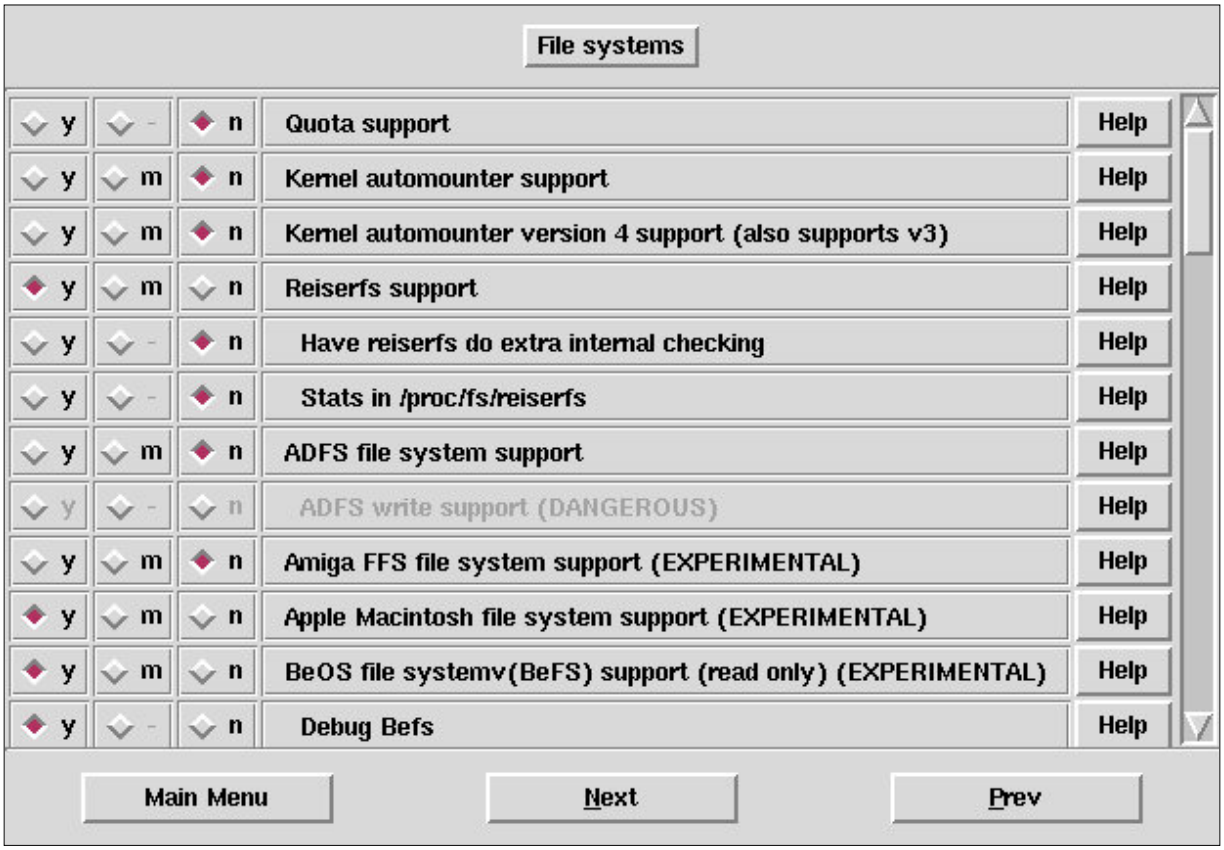

Image 9a: farmerdude File systems options area 1

Notice that I have not included support for the automounter daemon. This is because I want granular control over what I mount and how I mount it. Automounter will attempt to automatically mount remote file systems, such as those typically found in NFS environments.

'y Reiserfs support'

'y Apple Macintosh file system support (EXPERIMENTAL)'

NOTE: In *Image 9a* you see an option for the BeOS file system (BeFS). However, that option will not be here unless you patch the kernel. Since we have not covered that here I will omit this option. If you are interested in mounting drives/volumes from a BeOS platform then this is something you will need to do. However, patching the kernel can be quite cumbersome and is not something I want to cover in this paper. Perhaps in Part 2!

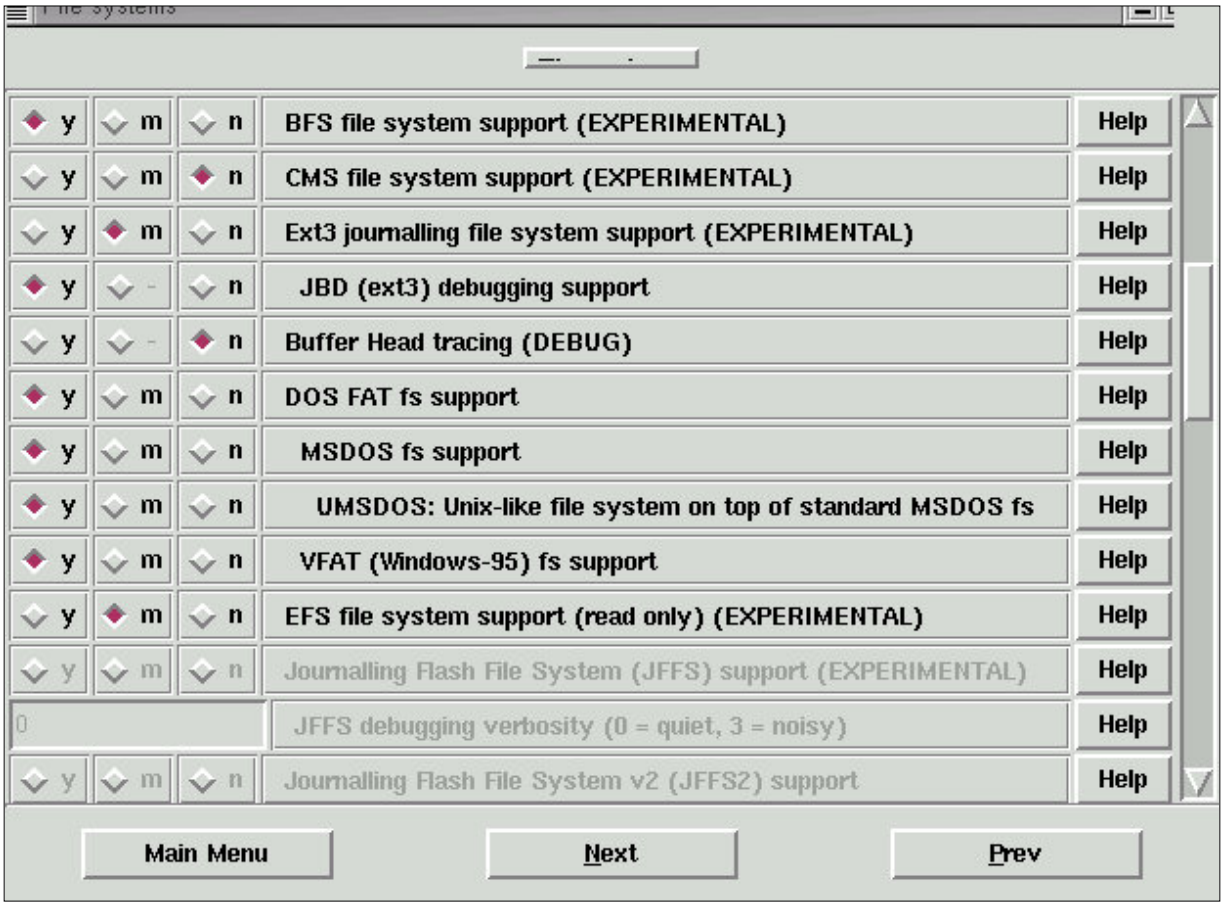

Image 9b: farmerdude File systems options area 2

'y BFS file system support (EXPERIMENTAL)'
'm Ext3 journalling file system support (EXPERIMENTAL)'
'y JBD (ext3) debugging support'
'y DOS FAT fs support'
'y MSDOS fs support'
'y VFAT (Windows-95) fs support'

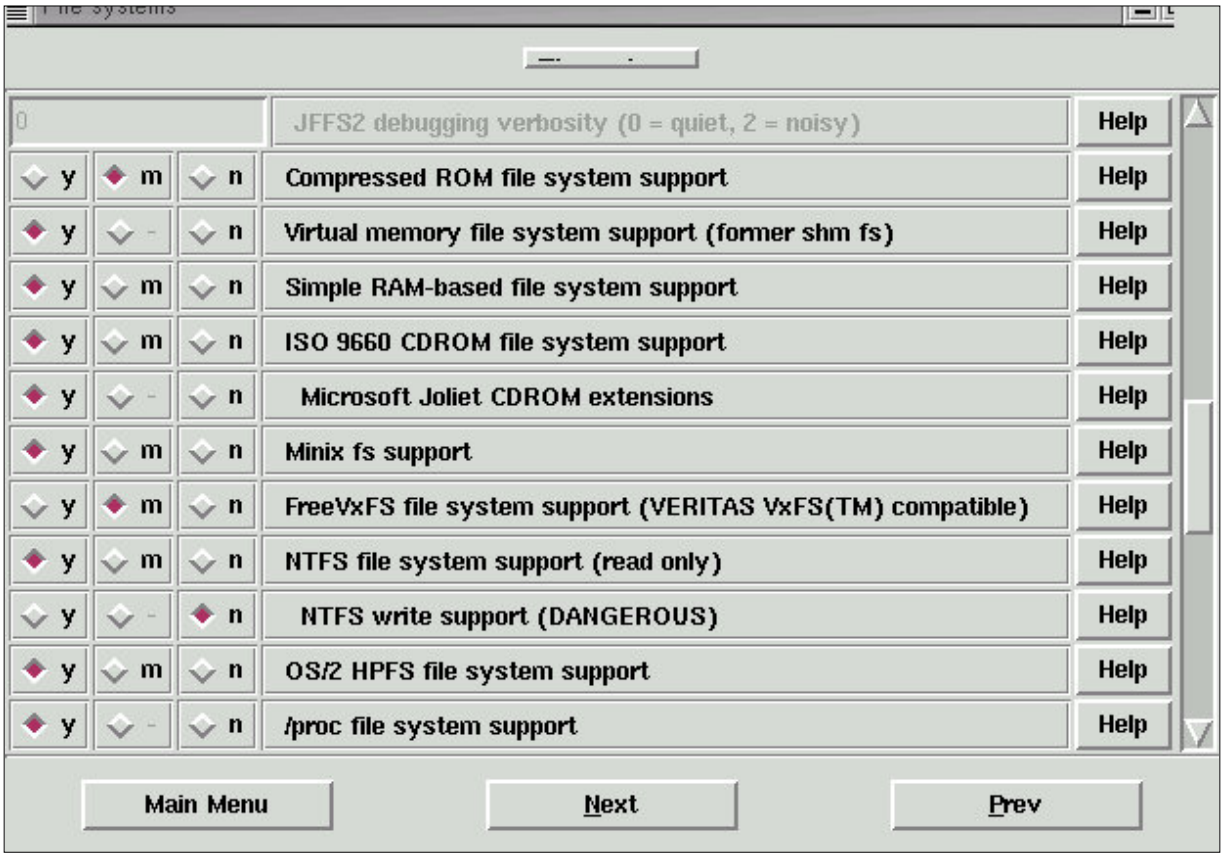

Image 9c: farmerdude File systems options area 3

'y ISO 9660 CDROM file system support'
'y Microsoft Joliet CDROM extensions'
'y Minix fs support'
'y NTFS file system support (read only)'

NOTE: We do not include WRITE support for NTFS. Write support is not very good. Do not include support for NTFS Write. You have been warned!

'y OS/2 HPFS file system support'
'y /proc file system support'

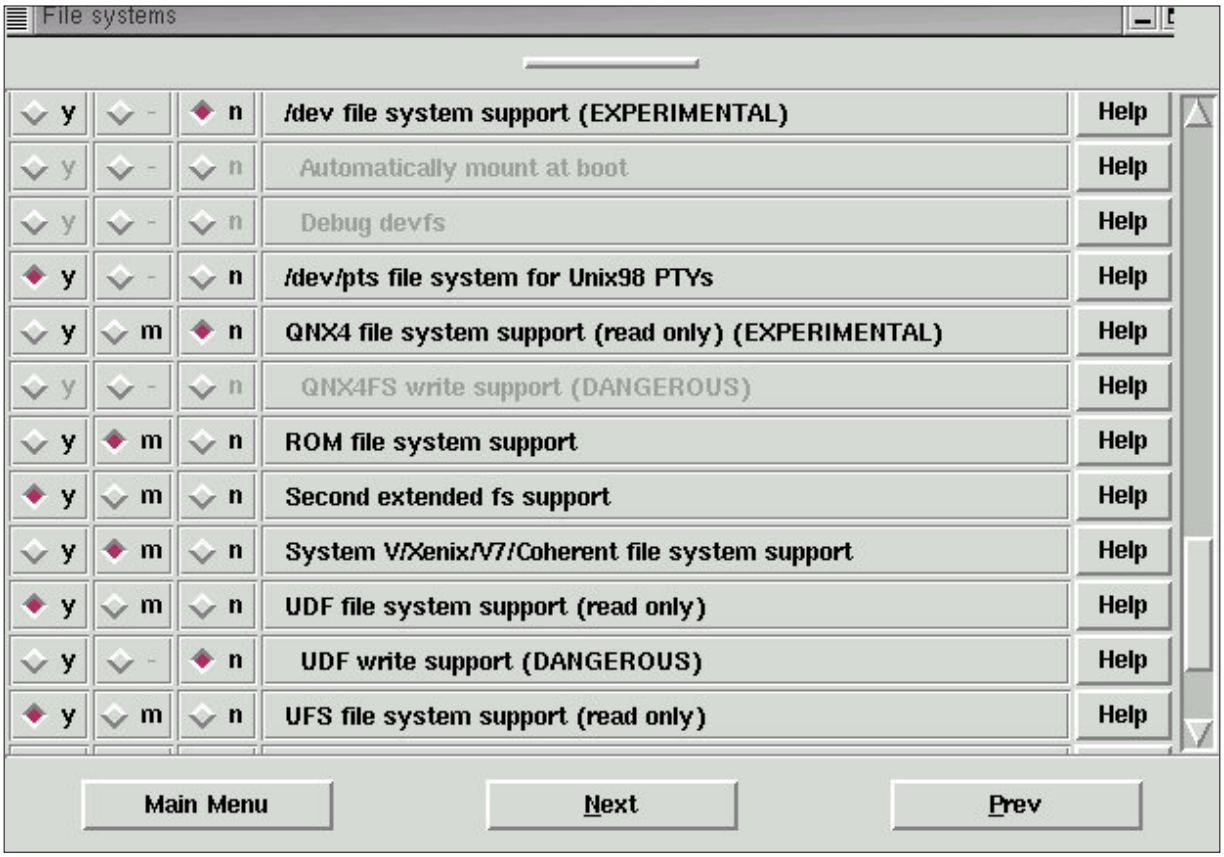

Image 9d: farmerdude File systems options area 4

'y /dev/pts file system for Unix98 PTYs'
'y Second extended fs support'
'y UDF file system support (read only)'
'y UFS file system support (read only)'

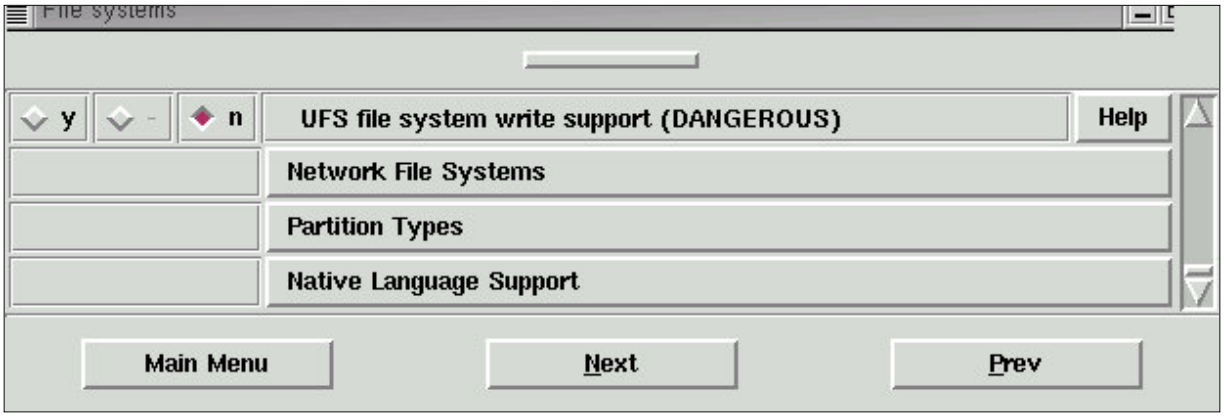

Image 9e: farmerdude File systems options area 5

Now that we have selected each of the file systems we want to include support for we must now include support for Network File Systems, Partition Types, and Native Language Support.
Let's look at the Network File Systems category now.

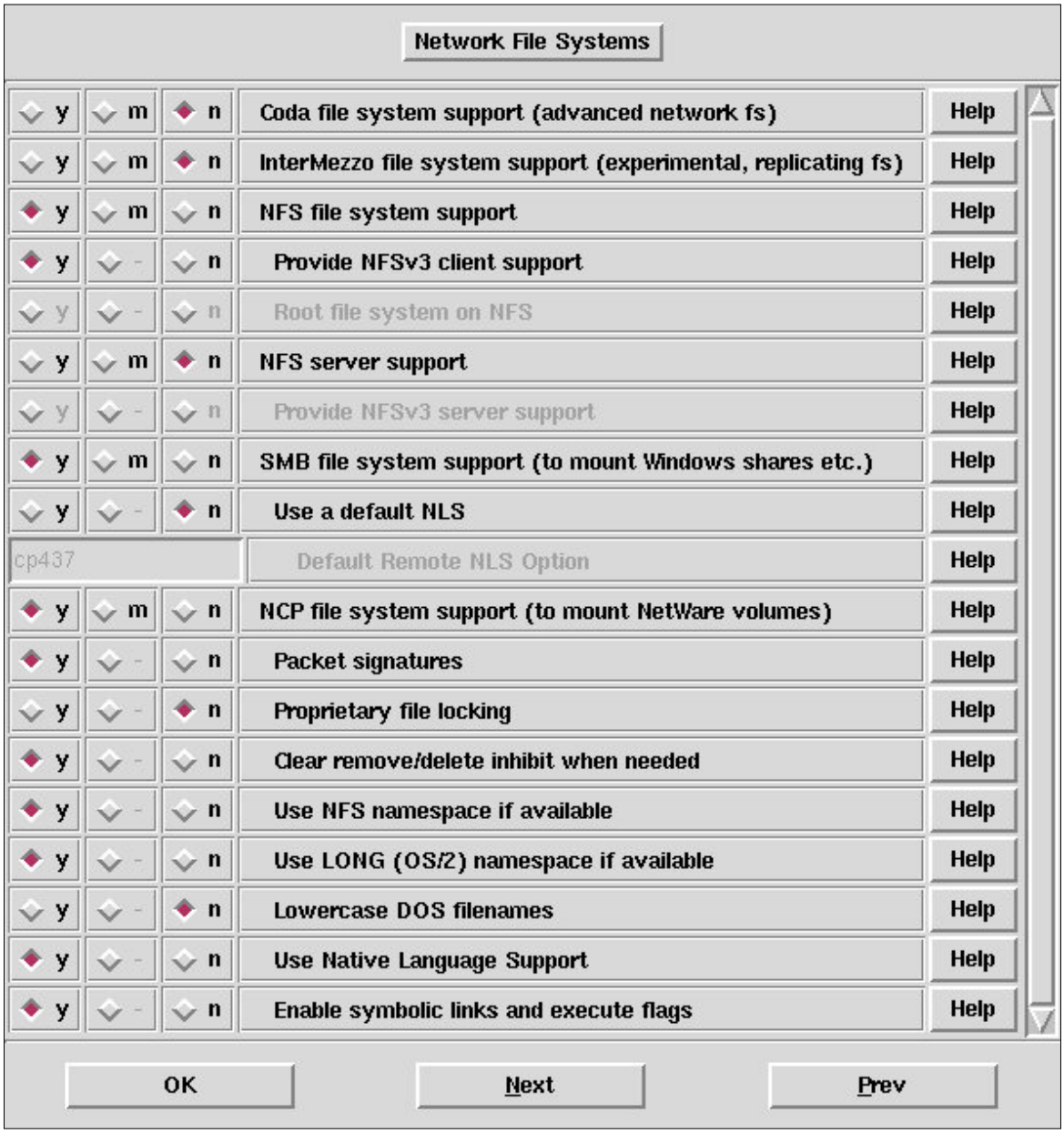

Image 10: farmerdude Network File Systems options

'y NFS file system support'
'y Provide NFSv3 client support'
'y SMB file system support (to mount Windows shares, etc.)'
'y NCP file system support (to mount NetWare volumes)'
'y Packet signatures'
'y Clear/remove/delete inhibit when needed'
'y Use NFS namespace if available'
'y Use LONG (OS/2) namespace if available'
'y Use Native Language Support'
'y Enable symbolic links and execute flags'

Now that we have finished selecting support for Network File Systems we return to the main File Systems category and select the Partition Types category. Here is where we will enable support for partitions created by other non-linux operating systems. We will select the following options:

'y Advanced partition selection'
'y Macintosh partition map support'
'y PC BIOS (MSDOS partition tables)'
'y BSD disklabel (FreeBSD partition tables) support'
'y Minix subpartition support'
'y Solaris (x86) partition table support'
'y Unixware slices support'
'y Windows Logical Disk Manager (Dynamic Disk) support'
'y SGI partition support'
'y Sun partition tables support'

USB SUPPORT

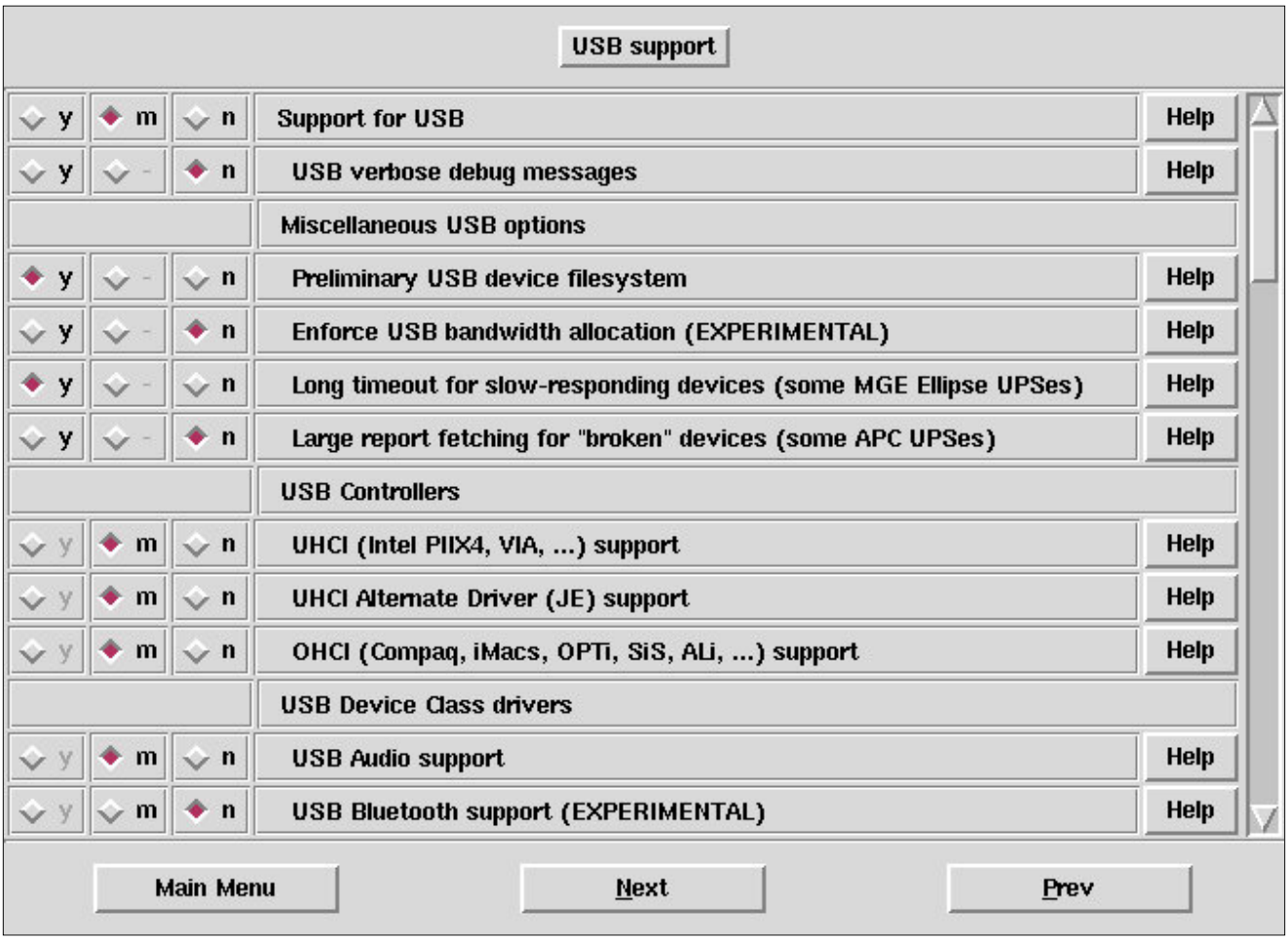

Image 11a: farmerdude USB support options

USB . . . scanners, cd burners, digital cameras, and even hard drives all connected via USB. So let us take a peek at the USB Support options and configure as follows:

'm Support for USB'
'y Preliminary USB device filesystem'
'm UHCI (Intel PIIX4, VIA, ...) support'
'm UHCI Alternate Driver (JE) support'
'm OHCI (Compaq, iMacs, OPTi, SiS, ALi, ...) support'

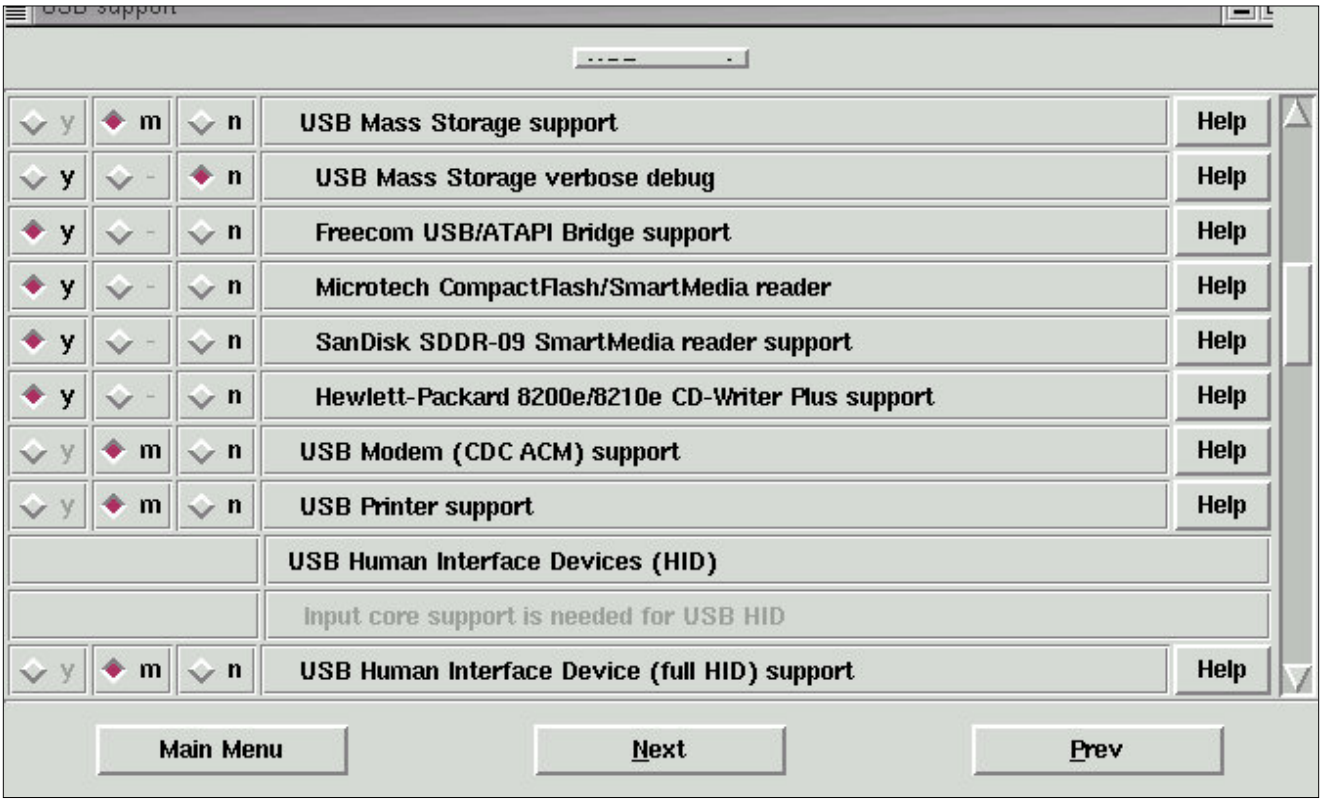

Image 11b: farmerdude USB support options

'm USB Mass Storage support' (usb-storage.o is module name)
'y Microtech CompactFlash/SmartMedia reader'
'y HP CD-Writer 82xx support'
'm USB Modem (CDC ACM) support'
'm USB Printer support' (printer.o is module name)

That's it. We're done! Yes, we have finally finished selecting our options for our super kernel. We are now ready to begin building our kernel and compiling our modules!

Let's get back to our terminal and remember, we must be in the /usr/src/linux-2.4 directory! We will need to set the dependencies and the command to do this is:

```
make dep
```

After setting the dependencies we will prepare the source tree for building our super kernel via:

```
make clean
```

And now we're ready to build the actual kernel. Wow Batman, is it really time? Yup! Let's issue:

```
make bzImage
```

The building of the kernel will take some time. Variables include your processor and what has been configured into the kernel. The smaller the kernel, the faster the building process.

Once the kernel has been built it is time to make the modules we have included support for:

```
make modules
```

Like the kernel building process, making the modules will take some time. Once we have made the modules we must install them. To install the modules issue:

```
make modules_install
```

Now we will copy two files ('bzImage' and 'System.map')to the /boot directory (or partition). If we navigate to the /boot directory and issue 'ls -la' we may find that either one,both or none of these files are already here. But, in the event that they are not, manually copying them will place them here. However, there is a catch.

We do not want to overwrite our previous working kernel files. Therefore, we must remember what we decided to call this custom kernel (remember editing the 'Extraversion =' line in the 'Makefile' way back when?). And when we copy these file we must append that custom name to them.
So, let's copy the first file, and assuming we are still in /usr/src/linux-2.4 we issue:

```
cp -v arch/i386/boot/bzImage /boot/vmlinuz-2.4.9-13farmerdude
```

Peeking into the /boot directory we should see this file was successfully copied to this directory.

Now we will copy the second file:

```
cp -v System.map /boot/System.map-2.4.9-13farmerdude
```

Peeking into the /boot directory we should see the 'System.map-2.4.9-13farmerdude' file.
Notice how our 'Extraversion = -13farmerdude' matches these two files (we appended the -13farmerdude to them during the copying)?

Lastly, let us go ahead and issue one last command before we check, and update if necessary, our boot loader. This command will install, check dependencies, and make the SCSI initrd img file:

```
new-kernel-pkg --install --depmod --mkinitrd 2.4.9-13farmerdude
```

Note that we appended our custom kernel name to the 'mkinitrd' portion of the command.

Before we reboot our system we must check our boot loader to make sure the updated information (newly compiled kernel) is present. If we are using GRUB (and we are!) then all we have to do is navigate to:

```
cd /boot/grub/
```

And view the grub.conf file:

```
more grub.conf
```

We should see our new kernel already here (that is a nice thing about GRUB);

```
# grub.conf generated by anaconda
#
# Note that you do not have to rerun grub after making changes to this file
# NOTICE: You have a /boot partition. This means that
# all kernel and initrd paths are relative to /boot/, eg.
# root (hd0,0)
# kernel /vmlinuz-version ro root=/dev/hda2
# initrd /initrd-version.img
#boot=/dev/hda
default=2
timeout=20
splashimage=(hd0,0)/grub/splash.xpm.gz
title Red Hat Linux (2.4.9-13farmerdude)
    root (hd0,0)
    kernel /vmlinuz-2.4.9-13farmerdude ro root=/dev/hda2
    initrd /initrd-2.4.9-13farmerdude.img
title Red Hat Linux (2.4.9-13)
    root (hd0,0)
    kernel /vmlinuz-2.4.9-13 ro root=/dev/hda2
    initrd /initrd-2.4.9-13.img
title Red Hat Linux (2.4.7-10custom)
    root (hd0,0)
    kernel /vmlinuz-2.4.7-10custom ro root=/dev/hda2
    initrd /initrd-2.4.7-10custom.img
title FreeBSD 4.5
    root (hd0,3,a)
    kernel /boot/loader
    boot
```

We see in the 'grub.conf' file above that our new kernel, 2.4.9-13farmerdude, was successfully added to the file. We only have to reboot next and test it to see if we can boot with it.

NOTE: If we were using LILO as our boot loader we would have to manually edit the 'lilo.conf' file found under /etc/ directory (/etc/lilo.conf) to reflect the new kernel and kernel files, followed by issuing the '/sbin/lilo -v -v' command. The simplest way to do this is to copy the old running kernel information and paste it to the end of the file. Then change that pasted section information to reflect the name of the new custom kernel.

VI. Reboot with New Data Forensics Super Kernel

Okay, so unmount any media (floppies, etc.) and reboot the system. All the while crossing your fingers! At the boot prompt select the '2.4.9-13farmerdude' option and hold on. Here's to a successful boot!

I hope I have enlightened you with the steps necessary to building a custom kernel. If you have any questions or comments, please forward them to me at info@crazytrain.com

farmerdude

Back to www.crazytrain.com/papers.html

copyright © 2002 info@crazytrain.com