Tom’s Networking

How To: Hacking the Linksys NSLU2 – Part 4 Customizing the Flash

By Jim Buzbee – September 9 2004

Introduction

I hope you've been enjoying the extensions you've made to your NSLU2 based on my previous articles [Part 1] [Part 2] [Part 3]. What we've accomplished is powerful, but there's something missing. Any time your box reboots, all of our new processes such as telnet, NFS and mt-daapd need to be manually started again. Surely we can do something to automatically start these processes at boot time like a standard computer.

If this were a normal computer we would just put our startup scripts into the boot sequence. On Linux this would mean tracing the startup sequence beginning with the first process, init, and moving through the rc script directory. We would position our script to be called somewhere after networking was initialized. But on this box it's not quite so easy. The entire startup sequence is stored on a RAM disk. Any changes we make on this disk are lost when the box reboots.

I spent quite a bit of time looking for a "hook", a file on the disk where we could put a reference to a custom script. One example of this would be crontab, a file on Unix-like systems that defines processes to be run at specific times. If we could put a reference to a custom script in crontab, Linux would run it for us. Unfortunately on the NSLU2, crontab is also on RAM disk.

The only possible file I found was the Samba configuration file. Samba had some capabilities to run external processes specified in the configuration file, but ultimately this led to a dead-end as well. My original goal was to add functionality without modifying the standard Linksys firmware, but it became apparent that there would be no easy way to do this.

Luckily, while I was spinning my wheels looking for a boot up hook, other members of the NSLU2 development community were taking a different approach. They were exploring the possibility of modifying the Linksys firmware itself. If you can modify the code in the Flash, you can change the boot up sequence to do any number of things including starting up new processes. It may seem an extreme approach, but since this is a Linux system, the layout of the Flash image is fairly standardized and well understood.

Disclaimer: It goes without saying, but I'll say it anyway. Exploring the NSLU2 by looking at its internal file structures using any method that doesn't modify its code should leave your warranty intact. But modifying the NSLU2 in any way will void your warranty.

TomsNetworking, Tom's Guides Publishing and I are not responsible for any damage that the information in this article may cause to your NSLU2 or any data it manages.

So download a copy of the current firmware before you start, and don't go trying to get help from Linksys if you break it.

Plan of Attack

In the case of the NSLU2, the Flash is split into four parts :

- Redboot
- System Configuration
- Kernel
- RAM disk

The Redboot portion is a bit like a PC's BIOS; it initializes the hardware and starts everything from power up. The System Configuration is where persistent variables such as the box's IP address are stored. The Kernel is the Linux operating system image, and the RAM disk is where all of the libraries, configuration files, scripts and executables such as the web server are stored . The RAM disk is what we'll need to modify if we want to automatically start our new processes.

The basic idea is that we'll take a firmware image, split it into its component parts, change the RAM disk portion, put it back together, and reinstall. Easily said, but not so easily done. Any missteps will create what is known as a brick, i.e. a dead piece of equipment that's not much more useful than masonry.

Warning: If you want to follow along with this article, you'll have to accept the very real possibility of destroying your box and most certainly your warranty. If you destroy your box, please don't take it back to the store. We don't want to discourage Linksys from creating devices that people can tinker with!

NOTE: A method to add a serial port to the NSLU2 has been documented. With a serial port, a box with a bad Flash can be recovered. Adding a serial port requires soldering and is beyond the skill of this author and the scope of this article.

Breaking news: The existence of a method to telnet into the RedBoot loader has also been uncovered. This too could assist in the recovery of a bad Flash without having to break out the soldering iron. For information on recovering from a bad Flash using either a serial port or telnet, see the NSLU2 developers mailing list.

Still here? OK, let's dig in. On my Linux system I created a work directory called myFlash. Create your own, and then inside your work directory, download a Flash image from Linksys. After unzipping the file, you should have a release notes file and a Flash image: NSLU2_V23R25.bin.

Now we'll need a tool to split the file into its parts. The tool we'll use, called splitnslu, was developed by Brian Lantz, but I cached a copy on my website since it may be undergoing changes. Fetch it from here. After untarring it ,you should have a README file, a .c file and a Makefile. The Makefile has several useful targets, but for our uses, we're going to do all of our operations by hand so it's clear what is going on.

Unpacking and Modification

First let's build the tool :

```
gcc -o  splitnslu splitnslu.c
```

This tool has two capabilities: unpacking a Flash and packing a Flash. In order to get to the RAM disk, unpack the Flash:

```
./splitnslu unpack  NSLU2_V23R25.bin
```

This should give you a series of messages as it unpacks the Flash into its component parts. The output file that we are interested in is the ramdisk.gz file. As the file extension indicates, the file is gzipped. We'll have to unzip it in order to work with it:

```
gzip -d ramdisk.gz
```

Now we have a file that can be directly mounted as a loop-back filesystem by Linux for manipulation. Create a mount point in your work directory like so:

```
mkdir nslu2
```

Then mount the RAM disk with the following command which will likely require root privilege:

```
mount -o loop ramdisk nslu2
```

Now what we have is a copy of the NSLU2 RAM disk mounted on the nslu2 directory. You can explore the disk by cd'ing into the directory, but be very careful. Any modifications that you purposely or inadvertently make will be permanent when we are finished. If you mess something up, you may destroy your box.

Our approach will be to make as few changes as possible in order to minimize the chances of doing something wrong. We just want to add a little hook and do the real work in a script that will live on the hard drive. First we need to determine where to place our hook.

The path of execution inside the RAM disk is fairly standard for a Linux system and can be easily traced. Almost always on a UNIX-like system the first process to run is called "init" and it is the same for the NSLU2. In our case, some sleuthing shows that init calls etc/rc which calls etc/rc.sysinit which calls etc/rc.d/rc.1. In the rc.1 file we can see where all of the various daemons such as the web server are started. This looks like a reasonable place to add our hook since our new processes are similar to the others being started in this script.

So, take a deep breath, steady your hand, cd into the directory and edit the rc.1 file with a simple text editor. I added the hook just before the web server was started. Here's what my new line looks like followed by the web server line:

```
/bin/echo  "Starting Local Hook:"; . /etc/rc.d/rc.hook;check_status
/bin/echo  "Starting WEB Server:"; . /etc/rc.d/rc.thttpd;check_status
```

Double check and then triple check your new line. I made the change by duplicating the web server line and modifying the "echo" portion and the rc.hook reference. Everything else should be the same.

Now we'll need to create the new script that we referenced in the rc.1 file. The script will live in the rc.d directory and will be called rc.hook. The idea of our new script will be to look for a hard-drive based script and if it exists, execute it. If it doesn't exist, we'll do nothing. No harm, no foul. The system will boot as it always did. Here's the script as I wrote it:

```
#!/bin/sh

if [ -x /share/hdd/conf/rc.d/rc.custom ]
then
  /share/hdd/conf/rc.d/rc.custom
fi
```

The first line of the file is superfluous since we are "dotting" the file from rc.1, not executing it. I added the line to keep it consistent with the rest of the scripts in this directory. The next line checks for an executable script called rc.custom in our conf/rc.d directory on the hard drive. If this file exists, it is executed.

We'll have to be careful that actions we take in the rc.custom script do not block causing the rest of the boot sequence to hang. Once again, double check rc.hook very, very carefully. As a sanity check, execute the script on your development system just like the rc.1 script will, to verify that it contains no errors:

```
. rc.hook
```

In the case of our development system it should not find the rc.custom script and will exit silently. When you are satisfied with the script and your change to rc.1 unmount the RAM disk like so:

```
cd ../../../
umount nslu2
```

We are done with our firmware modifications. If you want to do a complete double-check, you can mount the RAM disk file again, verify your changes are in it and then unmount it.

Repacking and Check Out

Now we'll need to put everything back together. First we need to compress the RAM disk file :

```
gzip -9 ramdisk
```

You should end up with a ramdisk.gz file. Next, we need to build the final Flash image using the pack parameter to splitnslu:

```
./splitnslu pack newflash.bin
```

This will give a few lines of status and then complete, creating a new Flash called newflash.bin. For assurance sake, compare the size of your new image with the size of the original image :

```
ls -l *.bin
```

You should see that both the original image and the new image are the same size.

We are ready to burn our new image to the NSLU2. This is the most critical part of the process. Up to now we haven't modified anything on the box itself. If you take this step, you take the risk of killing the box if the edits were made incorrectly.

We'll use the standard Linksys menus to install the Flash. First, using a browser, go the Administration web page of the NSLU2. Under Administration select Advanced, then Upgrade. Select the "Choose File" button and find your new Flash: newflash.bin. Then on the bottom of the page, select "Start Upgrade".

When a menu appears, take a deep breath and select "OK". At this point you should see the LEDs flickering on the box as the burn occurs. Eventually you'll get a pop-up indicating that the system is going to reboot. Select "OK". After a nervous minute or so, you should hear a single "beep" indicating that the box has booted normally. Whew! We've done it.

Now, enable telnet by executing the standard Management/telnet.cgi script discussed in my previous articles. Telnet in and take a look at the /etc/rc.d directory. You should see your new rc.hook script. Let's try it out. It's set up to execute our script /share/hdd/conf/rc.d/rc.custom which doesn't yet exist, so we'll create it. For starters, just put a test in it to verify that it works. I put the following in mine :

```
#!/bin/sh

/usr/bin/Set_Led beep1
```

Mark it executable, and then execute it through the rc.hook script to make sure it works:

```
chmod +x /share/hdd/conf/rc.d/rc.custom
/etc/rc.d/rc.hook
```

You should hear a "beep". Now reboot :

```
reboot
```

After a minute or so you should hear the standard first beep followed a few seconds later by our second beep. It works! We have a permanent hook on disk to start up our own processes.

Adding the Custom Script

Now it is time to make our new rc.custom file real. No longer will we have to execute the telnet.cgi program to turn on telnet, and we won't have to run our NFS and mt-daapd scripts by hand either. One last time, enable telnet and log in. Find your rc.custom script and update it to call your scripts that we created in the last two articles.

To make telnet permanent, we'll have to put a telnet entry in the /etc/inittab file. Since I don't use SMB filesystems anymore, my script deletes the daemons from the RAM disk. This is not a permanent deletion, it just deletes the executables off of the RAM disk to save space. If I decide to use them again, I'll just comment out my deletions. Remember that any commands you put in this file must either run in the background or finish quickly, otherwise the boot process will hang up.

Here's my rc.custom file :

```
#!/bin/sh

/usr/bin/Set_Led beep1

# setup inetd for telnet
echo "telnet   stream  tcp    nowait  root   /usr/sbin/telnetd" >>/etc/inetd.conf

# delete these daemons to save some space
# if you use smb filesystems, comment out these lines
```

```
rm -f /usr/sbin/smb*
rm -f /usr/sbin/nmbd

# start NFS
rm -f /etc/exports
ln -s /share/hdd/conf/etc/exports /etc/exports
/share/hdd/conf/rc.d/rc.portmap
/share/hdd/conf/rc.d/rc.nfsd

# start up the iTunes server
/share/hdd/conf/rc.d/rc.mt-daapd
```

If you mess something up in this file, it may still have an effect on the system, but it can be fixed by editing the file. You won't have to modify Flash. If worse comes to worst and your changes cause the box not to boot properly, you'll need to mount the disk in an external system to fix it. If you're careful, you shouldn't have any problems.

After you make the above changes, reboot one more time. This time when the box comes back up, you should be able to telnet in without having to run the telnet.cgi program. Your NFS server should be running and mt-daapd will be busy scanning the hard drive for music files.

Now that we've show how to install new software and create a custom Flash, what's next? Lots. We've created our own custom Flash, but the larger development community is way ahead of us. A worldwide community of NSLU2 developers has sprung up and they are working on a custom Flash that doesn't require a RAM disk. The advantage of this is that it frees up 10 megabytes of RAM. That's a lot on a 32 Megabyte system. Freeing up that memory will allow us to run even more sophisticated programs.