

# Embedded Linux Kickstart Session

Der Herr Hofrat

OpenTech EDV-Research GmbH Austria  
Lichtenstein Str 31, A-2130 Mistelbach, Austria  
<http://www.opentech.at>

## Abstract

This embedded Kickstart session introduces embedded Linux starting at the installation of a Slackware 9.1 system. Next a minimum filesystem is built and a ramdisk image is generated. Together with a kernel suitable for such a ramdisk-based environment a system is booted using this minimum embedded system. This kick-start manual is Licensed under FDL V1.2 <http://www.gnu.org/copyleft/fdl.html>

## 1 Introduction

This manual is not a in depth introduction to installing and running embedded GNU/Linux but, as the name says, a kick-start. It should guide you step-by-step to get you up and running on embedded GNU/linux quickly. Although this document describes the steps for a RTLinux/GPL based embedded environment built from a Slackware 9.1 system, steps will more or less be the same on a different distribution, and for other flavors of RT-enhanced Linux. For further kickstar sessions on other topics see <http://www.opentech.at/documents.html>. Feedback, especially on trying this for other platforms, is always appreciated.

## 2 Slackware 9.1 Install

- Boot from CD: < ENTER >

The boot prompt is only intended for passing additional kernel parameters - norm ally necessary if you have some non-standard hardware, also if the default `bare.i` kernel does not work, press [F2] at the boot prompt for a list of possible kernels to boot.

- boot: < ENTER >
- Enter 1 to select a keyboard:
- Keyboard map selection:

```
qwertz/de-latin1-noddeadkeys.map < OK >
```

- Keyboard test

```
1 < OK >
```

Not a very intuitive interface that requires to type in 1 to the text field before hitting < OK > - but thats Slackware...

You may now login as 'root'

Slackware login:

At this point Slackware is running a minimum system in a ramdisk so you actually are login into the Linux box as root at this point. So type in root and hit < ENTER >

## 2.1 Partitioning

As noted above Slackware boots into a minimum system loaded into a ramdisk - so you have the 'standard' GNU/Linux tools available for system setup. Slackware does not bother providing a 'User Friendly' wrapper to these functions, you simply use them on the command line and that ensures that you actually know what you are doing.

```
# fdisk /dev/hda
```

The number of cylinders for this disk is set to 15017.

There is nothing wrong with that, but this is larger than 1024, and could in certain setups cause problems with:

- 1) software that runs at boot time (e.g., old versions of LIL0)
- 2) booting and partitioning software from other OSs (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help):

To check the existing partition table use the 'p' command

Command (m for help): p

```
Disk /dev/hda: 123.5 GB, 123522416640 bytes
255 heads, 63 sectors/track, 15017 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
```

Device	Boot	Start	End	Blocks	Id	System
/dev/hda1		1	103	827316	82	Linux swap
/dev/hda2	*	104	15016	119788672+	83	Linux

First we delete all partitions as this is going to be a pure Linux box. If there are partitions defined make sure you delete them in reverse order - so start with the highest numbered partition and delete one by one (in my case this was 2).

```
Command (m for help): d
Partition number (1-4): 2
```

```
Command (m for help): d
Selected partition 1
```

```
Command (m for help): 1
```

Next we create two new partitions one as are Linux filesystem (we will simply put it all in one big chunk for now) and one swap partition.

```
Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
```

We respond with 'p' for a primary partition and then get the prompt for the partition number.

```

P
Partition number (1-4): 1
First cylinder (1-15017, default 1): <ENTER>

```

As our first partition should start at the first cylinder we simply hit enter.

```

Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-15017, default 15017): +512M

```

On the first partition we are going to put the swap partition, so we request 512MB for the first partition, the '+' tells fdisk to increment 512MB starting at the current cylinder position, which is 1 in our case. We could give it a cylinder number too but then you must calculate the size your self..

```

Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)

```

```

P
Partition number (1-4): 2
First cylinder (64-15017, default 64):
Using default value 64
Last cylinder or +size or +sizeM or +sizeK (64-15017, default 15017):
Using default value 15017

```

The second partition is again a primary partition and will simply be the full remaining disk, which is offered by default.

If we now print the current partition table we see the two desired partitions, but they are both marked as Linux, we need one to be a swap partition.

```

ommand (m for help): p

```

```

Disk /dev/hda: 123.5 GB, 123522416640 bytes
255 heads, 63 sectors/track, 15017 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

```

Device	Boot	Start	End	Blocks	Id	System
/dev/hda1		1	63	506016	83	Linux
/dev/hda2		64	15017	120118005	83	Linux

So the next step is to change our first partition to Linux swap with the 't' command.

```

Command (m for help): t
Partition number (1-4): 1
Hex code (type L to list codes): L

```

0	Empty	1c	Hidden W95 FAT3	70	DiskSecure Mult	bb	Boot Wizard hid
1	FAT12	1e	Hidden W95 FAT1	75	PC/IX	be	Solaris boot
2	XENIX root	24	NEC DOS	80	Old Minix	c1	DRDOS/sec (FAT-
3	XENIX usr	39	Plan 9	81	Minix / old Lin	c4	DRDOS/sec (FAT-
4	FAT16 <32M	3c	PartitionMagic	82	Linux swap	c6	DRDOS/sec (FAT-
5	Extended	40	Venix 80286	83	Linux	c7	Syrinx
6	FAT16	41	PPC PReP Boot	84	OS/2 hidden C:	da	Non-FS data
7	HPFS/NTFS	42	SFS	85	Linux extended	db	CP/M / CTOS / .
8	AIX	4d	QNX4.x	86	NTFS volume set	de	Dell Utility
9	AIX bootable	4e	QNX4.x 2nd part	87	NTFS volume set	df	BootIt
a	OS/2 Boot Manag	4f	QNX4.x 3rd part	8e	Linux LVM	e1	DOS access
b	W95 FAT32	50	OnTrack DM	93	Amoeba	e3	DOS R/0

```

c  W95 FAT32 (LBA) 51  OnTrack DM6 Aux 94  Amoeba BBT      e4  SpeedStor
e  W95 FAT16 (LBA) 52  CP/M              9f  BSD/OS         eb  BeOS fs
f  W95 Ext'd (LBA) 53  OnTrack DM6 Aux a0  IBM Thinkpad hi ee  EFI GPT
10 OPUS           54  OnTrackDM6       a5  FreeBSD        ef  EFI (FAT-12/16/
11 Hidden FAT12   55  EZ-Drive         a6  OpenBSD        f0  Linux/PA-RISC b
12 Compaq diagnost 56  Golden Bow      a7  NeXTSTEP       f1  SpeedStor
14 Hidden FAT16 <3 5c  Priam Edisk     a8  Darwin UFS     f4  SpeedStor
16 Hidden FAT16   61  SpeedStor       a9  NetBSD         f2  DOS secondary
17 Hidden HPFS/NTF 63  GNU HURD or Sys ab  Darwin boot    fd  Linux raid auto
18 AST SmartSleep 64  Novell Netware b7  BSDI fs        fe  LANstep
1b Hidden W95 FAT3 65  Novell Netware b8  BSDI swap      ff  BBT
Hex code (type L to list codes): 82
Changed system type of partition 1 to 82 (Linux swap)

```

You should then check that the partition table is correct and then call the write command to actually write the new partition table to disk. Until you type 'w' for write nothing on the disk was changed - so you can quit any time by pressing < CNTRL > <C> or 'q' at the menu.

## 2.2 Starting setup

Slackware has a setup program on the ramdisk that you invoke by simply typing in

```
# setup
```

we then select the key-map again - see above - and proceed on to setting up our swap disk, the partition is all ready created and the setup script will find it, so we just need to activate it

- SWAP SPACE DETECTED

```
    /dev/hda1
```

- FORMATTING SWAP PARTITION...
- SWAP SPACE CONFIGURED

```
    /dev/hda1
```

The partition we set up for the Linux filesystem needs to be formatted next, first we select the partition from the presented possibilities, which is only /dev/hda2 in our case, and hit < ENTER >, next we are ask for the method of formating, Format is OK for almost all systems, if the hard-disk is some old disk (and not too large...) you might want to select 'Check' which will actually check each block and update the bad-blocks list if necessary.

- Select Linux installation partition:

```
    /dev/hda2
```

- FORMAT PARTITION /dev/hda2

```

NOTE: This will erase all data on it.
Format < OK >
Check
No

```

We suggest using ext3 filesystem, it will not save your data if you crash the kernel with a buggy kernel module, but it is a good protection against power-failure or reset button induced problems... The inode density can be left at the suggested default value.

- SELECT FILESYSTEM FOR /dev/hda2

```
ext2
ext3 < OK >
reiserfs
```

- SELECT INODE DENSITY FOR /dev/hda2

```
4096 1 inode per 4096 bytes. < OK >
2048 1 inode per 2048 bytes.
1024 1 inode per 1024 bytes.
```

- FORMATTING...
- DONE ADDING LINUX PARTITIONS TO /etc/fstab

So now the systems partitions are set up and formatted - we are ready to fill the disk up with content. But before that we have to select a installation media, which is the CD we booted from in our case, this menu question makes sense because Slackware can also be installed starting with a floppy disk and if you have a really fast university network (does something like this actually exist ?) then NFS install may be an option. In case you booted from the CD the auto-detecton will work fine, so we select [auto] and hit < OK >.

- SOURCE MEDIA SELECTION

```
1 Install from a Slackware CD or DVD < OK >
2 Install from a hard drive partition
3 Install from NFS (Network File System)
4 Install from a pre-mounted directory
```

- SCANNING FOR CD or DVD DRIVE

```
auto < OK >
manual
```

- SCANNING...

If this does not kick your CD-ROM drive up then you should try manual selection.

Once you have your source media set we go to the package selection. Slackware allows you to select each package individually, which can take a very long time, so unless you really want a minimum system using the prepackaged selections is fine and will result in a system with all tools we need for real-time. The only thing we deselect here is KDE and GNOME, simply to reduce install time and because we are not concerned with X-setup for this session. We want to show you the power of the command-line, you can learn how to play with X-Windows later :)

After de-selecting KDE and GNOME we can simply install everything and hit < OK >.

- PACKAGE SERIES SELECTION

```
NOTE: If you install without KDE and GNOME, you will only need disc 1.
```

- SELECT PROMPTING MODE

```
full < OK >
```

- Installing...

If you did not de-select KDE and GNOME then you will be prompted for the second disk, in our case this does not happen, so we would go right to the kernel selection.

- INSERT NEXT DISC

Continue < OK >

- Installing...

You should not necessarily select the hottest and most optimized kernel here, you should select the safest kernel for the system, for IDE based systems the `bare.i` from the cdrom is what you want.

- INSTALL LINUX KERNEL

```
bootdisk
cdrom < OK >
floppy
skip
```

- CHOOSE LINUX KERNEL

/cdrom/kernels/bare.i/bzImage < OK >

It is a wise thing to create a boot-disk for a development system, sooner or later you might damage the system with your first (buggy) kernel modules, and as we never make backups... a boot-disk is helpful. In this kickstart session we will skip this step though.

We are not going to bother with the modem, and for desk-top systems you probably will not need the hot-plug subsystem, but it does not hurt to enable it.

- MAKE BOOTDISK

```
Create
Skip < SKIP >
```

- MODEM CONFIGURATION

no modem < OK >

- ENABLE HOTPLUG SUBSYSTEMS AT BOOT?

< Yes >

We need a boot-loader to actually start the system on power-on, so next we configure the LInux LOader - LILO. If you know lilo and want some special options set, select 'expert' if you are a new-bee, take the 'simple' option, it will work in more or less all cases where you have a IDE based system.

Selecting frame-buffer console is important or you will not get the penguin logo in the top left hand corner of your screen...

- INSTALL LILO

```
simple < OK >
expert
skip
```

- CONFIGURE LILO TO USE FRAME BUFFER CONSOLE?

1024x768x256 < OK >

If you have a CD-burner in your system, which is quite common, then you want to set up that CD-ROM as a SCSI device via the ide-scsi emulation, so we pass the device specific module to LILO telling it to use scsi emulation for /dev/hdc in this case.

LILO is put on the Master Boot Record (MBR). After this step the Linux loader LILO is installed and the system could boot. Note that you only should put Lilo on the MBR if it is a stand-alone instalation, if you want to share the hardware emong different Linux and non-Linux installs refere to `Multiboot-with-LILO` in the Linux HOWTO collection.

- OPTIONAL LILO append="<kernel parameters>" LINE

```
hdc=ide-scsi < OK >
```

- SELECT LILO DESTINATION

```
Root
Floppy
MBR < OK >
```

The rest of the configuration is not that general and may be different in your case. First we set up the mouse and configure General Purpose Mouse-support - GPM which allows using the mouse in text mode.

- MOUSE CONFIGURATION

```
ps2 < OK >
bare 2 button serial mouse
ms 3 button serial mouse
```

- GPM CONFIGURATION < Yes >

The network configuration is site specific, so you need to get the infos from your network admin. The infos you will need are:

- Host name
- Domain name
- IP address
- Netmask
- Default gateway
- Domain Name Server (DNS)

With these informations ready we can proceed with the network configuration.

- CONFIGURE NETWORK? < Yes >
- ENTER HOSTNAME

```
rtl15 < OK >
```

- ENTER DOMAINNAME for 'rtl15'

```
hofr.at < OK >
```

- SETUP IP ADDRESS FOR 'rtl15.hofr.at'

```
static IP < OK >
DHCP
loopback
```

- Fill in the
  - IP address
  - Netmask
  - Default gateway
  - Domain Name Server (DNS)
- CONFIRM SETUP COMPLETE < Yes >
- CONFIRM STARTUP SERVICES TO RUN < OK >
- CONSOLE FONT CONFIGURATION < No >

Setting up the clock: assume the clock is UTC and select your time-zone from the list.

- HARDWARE CLOCK SET TO UTC?

```
No
Yes < OK >
```

- TIMEZONE CONFIGURATION

```
Europe/Vienna < OK >
```

If you did not select the KDE and GNOME packages during installation, you should not select KDE or gnome here... but there are a number of interesting and light weight window managers around that are worth giving a look.

- SELECT DEFAULT WINDOW MANAGER FOR X

```
xinitrc.kde < OK >
xinitrc.gnome
```

## 2.3 Final steps before reboot

Last thing the system needs before we can reboot is a root password, for this session you should set it to 'nopasswd', but in your company network or at home, make sure you have a reasonable root-password that will not be guessed easily.

- WARNING: NO ROOT PASSWORD DETECTED Would you like to set a root password? < Yes >

```
New password: nopasswd
Re-enter new password: nopasswd
```

- SETUP COMPLETE < OK >
- EXIT < OK >

This terminates the setup program of Slackware, and we can reboot the system. Just to make sure the filesystems are cleared properly we do:

```
# umount -a
# <CTRL>-<ALT>-<DELETE>
```

...and don't forget to remove the CD-ROM or you will fall into an endless loop.

## 2.4 System Boot and user account

At the LILO prompt you can add boot-command-line parameters for the kernel. This is helpful for instance, if you set up X (which is in init 4) and your screen just flickers, then you simply type in 'linux init 3', and boot the system to text-mode only. for more info on available settings check the BootPrompt-HOWTO locate in /usr/doc/Linux-HOWTOs/BootPrompt-HOWTO on your Slackware 9.1 distribution.

```
boot: linux < ENTER >
```

After the boot messages scrolled by you get the login prompt:

```
rtl15 login: root
Password: nopasswd
```

We hope that the messages produced by Slackware after login - the so called fortunes - are politically correct, but we take no responsibility for these messages....

We need to add a regular user-account, if you want to use the GNU/Linux box via remote logins or send e-mail etc. you should not work as root, so lets ad a regular user account and then we are done.

```
root@rtl15:~ # adduser
  Login name for new user []: georgs
  User ID: 500
  Initial group [users]: < ENTER >
  Additional groups []: < ENTER >
  Home directory [/home/georg]: < ENTER >
  Shell [/bin/bash]: < ENTER >
  Expiry date []: < ENTER >
press ENTER to go ahead and make the account. < ENTER >
  Full Name []: Georg S
  Room Number []: < ENTER >
  Work Phone []: +43-12345
  Home Phone []: +12345
  Other []: < ENTER >
New password: nopasswd
Re-enter new password: nopasswd
Account setup complete.
```

A bit rough in style but it should get you up and running quickly :)

## 3 RTLinux kernel install

Mount the proceedings CD

As the default location to attache the cdrom is /mnt/cdrom (see /etc/fstab for the default on your system), you can use the command

```
root@rtl15:~ # mount /mnt/cdrom
```

and all necessary informations would be extracted from /etc/fstab - as we want to show you as much of what is happening beneath the covers - we will use the explicit mount command and create a mount point first.

```
root@rtl15:~ # mkdir /cdrom
root@rtl15:~ # mount -t iso9660 /dev/hdb /cdrom
```

In the system used for this HOWTO the cdrom was the secondary slave device on the IDE subsystem so /dev/hdb in this case - you must replace any references to /dev/hdb by what is given by your system configuration to find the device quickly - type in the following:

```
root@rtl15:~# dmesg | grep hd
```

```
hda: IC35L120AVV207-0, ATA DISK drive
hdb: LITE-ON COMBO LTC-48161H, ATAPI CD/DVD-ROM drive
hda: attached ide-disk driver.
hda: host protected area => 1
hda: 241254720 sectors (123522 MB) w/1821KiB Cache, CHS=15017/255/63
  hda: hda1 hda2
...
```

This tells us that the CDROM is attached as hdb.

Now copy and unpack the vanilla kernel linux-2.4.21 from Proceedings CD

```
root@rtl15:~ # cp /cdrom/kernel/linux-2.4.21.tar.bz2 /usr/src/
root@rtl15:~ # cd /usr/src/
root@rtl15:/usr/src # tar -xjf linux-2.4.21.tar.bz2
root@rtl15:/usr/src # mv linux linux-2.4.21-rtl3.2
root@rtl15:/usr/src # ln -s linux-2.4.21-rtl3.2 linux
```

Note that older tar versions use -tIf and -xIf.

Copy RTLinux from the Proceedings CD and unpack it

```
root@rtl15:/usr/src # cp /crom/rtlinux-3.2-pre3.tar.bz2 ./
root@rtl15:/usr/src # tar -xjf rtlinux-3.2-pre3.tar.bz2
root@rtl15:/usr/src # ln -s rtlinux-3.2-pre3 rtlinux
```

### 3.1 Patch kernel

Decompress the kernel patch

```
root@rtl15:/usr/src # cd rtlinux/patches
root@rtl15:/usr/src/rtlinux/patches # bunzip2 kernel_patch-2.4.21-rtl3.2-pre3.bz2
```

First test the kernel patch to see if it applies properly

```
root@rtl15:/usr/src/rtlinux/patches # cd /usr/src/linux
root@rtl15:/usr/src/linux # patch -p1 --dry-run \
  < /usr/src/rtlinux/patches/kernel_patch-2.4.21-rtl3.2-pre3
```

If all goes well (sure it does...) patch the kernel now

```
root@rtl15:/usr/src/linux # patch -p1 \
  < /usr/src/rtlinux/patches/kernel_patch-2.4.21-rtl3.2-pre3
```

### 3.2 Configure RTLinux/GPL kernel

check with `lsmod` what essential kernel modules we need in the SuSE install (forget sound modules...) check network modules and peripherals that are essential. you can also get the config file SuSE used from the `/proc` directory (`/proc/config.gz`), but you need to check this config simply copying it may lead to problems (i.e. APM enabled...) .

```
root@rtl15:/usr/src/linux # make menuconfig
```

Code Maturity Level Options ---->

[\*] Prompt for development and/or Incomplete code/drivers

Loadable Module Support ---->

[\*] Enable loadable module support

[ ] Set Version Information on all module symbols

[\*] Kernel module loader

Processor Type and feature ---->

Select **\*EXACTLY\*** your CPU or a generic low-end CPU (check with `cat /proc/cpuinfo`)

Block devices ---->

```
...
<*> RAM disk support
(16384) Default RAM disk size (NEW)
[*] Initial RAM disk (initrd) support
```

Memory Technology Devices (MTD) --->

```
<M> Memory Technology Device (MTD) support
[*] Debugging
(3) Debugging verbosity (0 = quiet, 3 = noisy) (NEW)
```

```
...
<M> Caching block device access to MTD devices
```

Self-contained MTD device drivers --->

```
...
<M> Test driver using RAM (NEW)
(8192) MTD RAM device size in KiB (NEW)
(128) MTD RAM erase block size in KiB (NEW)
```

Filesystem ---->

```
...
<*> Reiserfs support x x
<*> Ext3 journalling file system support x x
...
<*> Minix fs support
...
[*] /dev file system support (EXPERIMENTAL)
[*] /proc file system support
[*] /dev/pts file system for Unix98 PTYs
< > QNX4 file system support (read only)
< > ROM file system support
<*> Second extended fs support
...
```

save and exit - Note the reiserfs is needed because SuSE-8.0 installed the basic system on a reiserfs partition - other distributions prefer other filesystems, check in `/etc/fstab` what filesystems you will need on the system.

```
root@rtl15:/usr/src/linux # cp .config myconfig
```

this saves the config in a way that will not be deleted by `make mrproper`.

### 3.3 compile and install kernel

```
root@rtl15:/usr/src/linux # make dep
root@rtl15:/usr/src/linux # make modules
root@rtl15:/usr/src/linux # make modules_install
root@rtl15:/usr/src/linux # make bzlilo
...
a half a million lines of confusing output later...
cp /usr/src/linux-2.4.21-rtl3.2/System.map /
if [ -x /sbin/lilo ]; then /sbin/lilo; else /etc/lilo/install; fi
Added Linux *
make[1]: Leaving directory '/usr/src/linux-2.4.18-rtl3.2/arch/i386/boot'
root@rtl15:/usr/src/linux #
```

The kernel was copied to `/vmlinuz` and the modules are in `/lib/modules/2.4.21-rtl3.2-pre3`. We now need to edit `/etc/lilo.conf` add entry for `rtlinux`. To boot your new kernel edit `/etc/lilo.conf` to add the new kernel entry. note that it depends on the kernels `INSTALL_PATH=` where it is put so in the case of the vanilla kernel the new kernel ends up in `/vmlinuz` not `/boot/vmlinuz` (like with `adeos`).

```
root@rtl15:/usr/src/linux # cd /etc
root@rtl15:/etc # vi lilo.conf
```

At the beginning of the `lilo.conf` in Slackware 9.1 you can find the lines

```
# VESA framebuffer console @ 1024x768x256
vga = 773
# Normal VGA console
# vga = normal
```

These should be changed to:

```
# VESA framebuffer console @ 1024x768x256
# vga = 773
# Normal VGA console
vga = normal
```

Note that the exact appearance may vary in other distributions - but the changes required are the same - these changes are necessary unless you want to configure frame-buffer support into the kernel - as this leads to some problems, especially with embedded boards, we recommend you set `vga = normal` unless you know exactly what this is about.

```
# End LILO global section
# Linux bootable partition config begins
image = /boot/vmlinuz
  root = /dev/hda2
  label = Linux
  read-only
```

Copy these 4 last lines and edit them so you end up with, as `make bzlilo` will put the new kernel into `/boot/vmlinuz` we boot the original Slackware kernel by putting the `image=/boot/vmlinuz-ide-2.4.22` line into the first boot selection item.

```
image = /boot/vmlinuz-ide-2.4.22
  root = /dev/hda2
  label = Linux
  read-only
image = /boot/vmlinuz
  root = /dev/hda2
  label = rtlinux
  append = "ide=nodma apm=off acpi=off"
  read-only
```

This will leave the default kernel set to the original distribution kernel and allow you to boot the patched RTLinux kernel at the `lilo` prompt. Note that the naming of the kernels is distribution specific and some distributions put the new kernel in `/vmlinuz` not `/boot/vmlinuz`. Note the `append` line inserted - this turns off DMA for the ide discs and disable power management, generally this is a good idea for real-time systems. Note also that the very careful setting of `ide=nodma` is not a requirement of RTLinux, where as `apm=off` and `acpi=off` is a requirement if you want to guarantee hard-realtime performance.

Next we need to install the new boot-loader configuration to the disk by running `lilo`.

```
root@rtl15:/etc # lilo
Added Linux *
Added rtlinux
```

this should run without any errors and show you the rtlinux image.

Now we can tell the system to boot rtlinux on the next reboot - this will not permanently change the boot kernel - so by default non-rt Linux will be booted and only if we select rtlinux at the boot-prompt or by running `lilo -R rtlinux` will rtlinux boot.

```
root@rtl15:/etc # lilo -R rtlinux
root@rtl15:/etc # reboot
```

After the system comes up again - login as root. check what we have running

```
root@rtl15:~ # uname -a
Linux linux 2.4.21-rtl3.2-pre3 #5 Sun Nov 2 23:12:18 PST 2003 i686 unknown
```

## 4 RTLinux

RTLinux is installed from sources on the Proceedings CD, no rpm's for RTLinux around. The procedure here applies not only to the rtlinux-3.2-pre3 version but is more or less identical for other versions. If you ever run into a problem of a module or a system behaving very strange, then please verify the strange behavior on a clean installation as described here, often strange behavior is due to accumulating changes and build procedures for custom modules not being clean... For question pertaining to the basic setup of RTLinux you can also contact the community via the rtlinux mailing list at [www2.fsmallbs.com/mailman/listinfo.cgi/rtl](http://www2.fsmallbs.com/mailman/listinfo.cgi/rtl). For the latest developments check the RTLinux/GPL developers site at <http://www.rtlinux-gpl.org>.

### 4.1 configure/compile rtlinux

```
root@rtl15:/etc # cd /usr/src/rtlinux
root@rtl15:/usr/src/rtlinux # make menuconfig
```

Support option --->

```
[*] Posix standard I/O
[ ] POSIX Priority protection
[*] Dev mem support
[*] Enable debugging
[*] rtl_printf uses printk
[ ] Nolinux support
[*] POSIX Signals
[*] POSIX Timers (NEW)
Message queue constants --->
[ ] RTLinux tracer support (experimental)
[ ] Userspace Real Time
[*] Floating Point Support
[*] RTLinux V1 API support
[*] RTLinux Debugger
[ ] Synchronized clock support
```

lets leave it all defaults (shown above) for now, also the driver section can be left as it is - Save and exit menuconfig.

```
root@rtl15:/usr/src/rtlinux # make dep
root@rtl15:/usr/src/rtlinux # make 2>&1 | tee build.log
root@rtl15:/usr/src/rtlinux # make devices
```

The command `make 2>&1 | tee build.log`— records the entire compiler output into the file `build.log`, so if anything goes wrong this can help you, and also help when you report errors to the mailing list.

The `make devices` is only necessary for the first installation - this creates the rtlinux specific device files in `/dev/`.

## 4.2 Check your instalation

The regression test performs a number of sanity checks - it will not tell you if the setup is suited for hard realtime applications, it will basically tell you that the instalation worked and that rlinux will not crash your box ;)

```
root@rtl15:/usr/src/rtlinux # ./scripts/regression.sh
```

the regressions script should ONLY return [ OK ], if you get anything else pleas let the community know . After the script terminated all rlinux modules are unloaded.  
now launch rlinux

```
root@rtl15:/usr/src/rtlinux # ./scripts/insrtl  
root@rtl15:/usr/src/rtlinux # lsmod
```

Check if the modules are loaded - it should return something like:

```
root@rtl15:/usr/src/rtlinux # dmesg -c
```

Clear the kernel message ring buffer so that we can see what messages popped up after we launched rlinux examples.

Lets start with a very simple example - "hello World" in hard-realtime.

## 4.3 hello.o

```
root@rtl15:/usr/src/rtlinux # cd examples/hello  
root@rtl15:/usr/src/rtlinux/examples/hello # sync ; insmod hello.o
```

Why do we do `sync ; insmod hello.o`. If you load a module that you are playing with and you made a mistake your system can crash fairly easily, as the filesystem may be in a inconsistent state at this point it could loose data or even be damaged (depending on how wildly you crash your system) so it is a good habit to sync your disk before loading a kernel module as this reduces the probability of loosing data considerably.

```
root@rtl15:/usr/src/rtlinux/examples/hello # dmesg
```

check the messages that the `hello.o` module is generating with `dmesg`. To stop our realtime "hello World" we remove the `hello.o` module and clear the kernel message buffer.

```
root@rtl15:/usr/src/rtlinux/examples/hello # rmmod hello  
root@rtl15:/usr/src/rtlinux/examples/hello # dmesg -c
```

## 4.4 rt\_process.o

Now to a more usable example - `rt_process.o`. This kernel module measure the scheduling jitter of the hardware. It sets up a thread to run periodically for `ntests` times in a loop and report the minimum and maximum deviation of the time it actually ran to the time it should have run by writing the data to a realtime fifo (`rtf`). The data can be retrieved from the fifo with the `monitor` program. The `monitor` will, by default, retrieve 10000 data samples and the terminate, by passing it the `-s#para` meter you can tell it to grab exactly `#` samples.

```
root@rtl15:/usr/src/rtlinux/examples/hello # cd ../measurements  
root@rtl15:/usr/src/rtlinux/examples/measurements #  
root@rtl15:/usr/src/rtlinux/examples/measurements # sync ; insmod rt_process.o bperiod=0
```

load the measurement module with a `sync` again, if you don't want to do it, then simply `insmod rt_process.o` and find out the hard way why to sync your disk before loading a module ;) The `bperiod=0` module parameter instructs `rt_process.o` to launch only one thread and not launch a background thread that would compete for the CPU.

```
root@rtl15:/usr/src/rtlinux/examples/measurements # ./monitor -s 1000 | tee data
```

We only collect 1000 samples and put them in the file data, while at the same time displaying them on the screen.

```
root@rtl15:/usr/src/rtlinux/examples/measurements # ./gist data | tee data.out
```

Next we make a "histogram" of this data - Note that this is not a real histogram as we don't have access to the samples but only the min/max values of every ntests-large sample (default 500) so you can't interpret this data statistically. It does give you a fairly good overview of the systems rt-performance though - especially if you produce a high-load and high interrupt situation while running this test. If you launch `rt_process` without the `bperiod=0` parameter then you run two rt-threads and you can see what influence two threads completing at the same priority will have on the worst case scheduling-jitter of this specific system-hardware. So now lets clean up - removing the module and clearing the kernel message buffer again.

```
root@rtl15:/usr/src/rtlinux/examples/measurements # rmmod rt_process
root@rtl15:/usr/src/rtlinux/examples/measurements # dmesg -c
```

## 4.5 shut down rtlinux

```
root@rtl15:/usr/src/rtlinux/examples/measurements # cd /usr/src/rtlinux
root@rtl15:/usr/src/linux # ./scripts/rmrtl
root@rtl15:/usr/src/linux # dmesg -c
```

just to check if there were any problems unloading the rtlinux modules !  
Thats it - you now are rtlinux experts.... almost.

## 5 Debugging

To use gdb for debugging we need to reconfigure rtlinux and recompile it. To do this we first clean up and then launch menuconfig again.

```
root@rtl15:~ # cd /usr/src/rtlinux
root@rtl15:~ # make distclean
root@rtl15:~ # make menuconfig
```

Support option --->

```
[*] Posix standard I/O
[*] POSIX Priority protection
[*] Dev mem support
[*] Enable debugging
[*] rtl_printf uses printk
[ ] Nollinux support
[*] RTLinux tracer support (experimental)
[*] Userspace Real Time
[*] Floating Point Support
[ ] RTLinux V1 API support
[ ] RTLinux Debugger
[*] Synchronized clock support
```

save and exit.

```
root@rtl15:/usr/src/rtlinux # make dep
root@rtl15:/usr/src/rtlinux # make
```

RTLinux is now rebuilt with debugging flags, and with the RTLinux debugger module in the debugger subdirectory (`rtl_debug.o`)

## 5.1 RTLinux Debugger

Before we can launch the debugger we need to reload the modified rtlinux modules, we can use the scripts in the top-level rtlinux directory again.

```
root@rtl15:/usr/src/rtlinux # ./scripts/insrtl
root@rtl15:/usr/src/rtlinux # cd debugger
root@rtl15:/usr/src/rtlinux/debugger # insmod rtl_debug.o
```

RTLinux is now ready for debugging, before we insert the actual module to be debugged lets give it a look. If you look `hello.c` and compare it with `examples/hello/hello.c` you will find some debugging specific differences.

```
#include <rtl_debug.h>
...
void * start_routine(void *arg)
{
    ...
    if (((int) arg) == 1) {
        breakpoint();
    }
    ...
}
```

This `breakpoint();` instruction is the point where gdb will halt and you can continue from there on, if your module has not breakpoint and no bug that causes an exception then gdb can't connect, so either code a `segfault` or use the `breakpoint(); function ;`

```
root@rtl15:/usr/src/rtlinux/debugger # insmod hello.o
root@rtl15:/usr/src/rtlinux/debugger # gdb hello.o
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-suse-linux"...
(gdb) target remote /dev/rtf10
Remote debugging using /dev/rtf10
[New Thread -1012596736]
[Switching to Thread -1012596736]
start_routine (arg=0x1) at hello.c:37
37 for (i = 0; i < 20; i ++ ) {
warning: shared library handler failed to enable breakpoint
(gdb) l
32
33 if (((int) arg) == 1) {
34 breakpoint();
35 }
36
37 for (i = 0; i < 20; i ++ ) {
38 pthread_wait_np ();
39 rtl_printf("I'm here; my arg is %x\n", (unsigned) arg);
40 }
41 return 0;
(gdb) break rtl_printf
Function "rtl_printf" not defined.
(gdb) modaddsymb ../modules/rtl.o
```

```

add symbol table from file "../modules/rtl.o" at
.text_addr = 0xc88da060
(gdb) modaddsym ../modules/rtl_time.o
add symbol table from file "../modules/rtl_time.o" at
.text_addr = 0xc88e0060
(gdb) modaddsym ../modules/rtl_sched.o
add symbol table from file "../modules/rtl_sched.o" at
.text_addr = 0xc88ea060
(gdb) break rtl_printf
Breakpoint 1 at 0xc88db189: file rtl_printf.c, line 39.
(gdb) c
Continuing.

```

```

Breakpoint 1, rtl_printf (fmt=0xc8905140 "I'm here; my arg is %x\n")
  at rtl_printf.c:39

```

```

39 {
(gdb) l
34 static char initial_printkbuf [MAX_PRINTKBUF];
/* need to protect in_printkbuf from overflowing */

35 static char in_printkbuf[MAX_PRINTKBUF]; /* please don't put this on my stack*/
36 static char *printkptr = &in_printkbuf[0];
37 static spinlock_t rtl_cprintf_lock = SPIN_LOCK_UNLOCKED;
38 int rtl_printf(const char * fmt, ...)
39 {
40 rtl_irqstate_t flags;
41 int i;
42 va_list args;
43
(gdb) quit
The program is running.  Exit anyway? (y or n)

```

What we did was connect to gdb via /dev/rtf10 (that the 'target remote /dev/rtf10' line), then gdb stopped at the breakpoint instruction, and we tried to set a breakpoint at `rtl_printf`, but that was not known because the symbols were not loaded, so next we use a convenient `modaddsym` macro found in the `.gdbinit` file of the debugger directory, to load the symbol information from the `rtlinux` core modules. Now setting of the breakpoint works fine, and we can continue (command 'c' in gdb), stopping at the first `rtl_printf`. Next we list (command 'l' in gdb) the `rtl_printf` function. There really is not much to do in this example so we quite (command 'q' in gdb).

As we were debugging the module, it was of course not running in realtime, and if you now type `dmesg`, you can see that the execution of the individual threads (`hello.o` spawns two threads! ) gets confused.

```

RTLinux Extensions Loaded (http://www.fsmlabs.com/)
RTLinux Debugger Loaded (http://www.fsmlabs.com/)
rtl_debug: exception 0x3 in hello (EIP=0xc89050a8),
  thread id 0xc3a50000; (re)start GDB to debug
I'm here; my arg is 1
I'm here; my arg is 1
I'm here; my arg is 1
I'm here; my arg is 2
I'm here; my arg is 1
I'm here; my arg is 2
I'm here; my arg is 1
I'm here; my arg is 2

```

So gdb is good to find a segfault or some other coding error, but it will not allow to debug race-conditions or to locate temporal misbehavior of your realtime module, and in fact the behavior of multi-threaded apps

can be quite strange in gdb, even for those that work fine when run free !  
We are done with our debugging intro so lets clean up.

```
root@rtl15:/usr/src/rtlinux/debugger # dmesg -c
root@rtl15:/usr/src/rtlinux/debugger # rmmod hello
root@rtl15:/usr/src/rtlinux/debugger # rmmod rtl_debug
```

## 5.2 RTLinux Tracer

When we reconfigured RTLinux, with menuconfig, above, we enabled the RTLinux Tracer. The problem with GDB was that it did not allow temporal debugging, and this is because gdb is taking control of the rt-thread, so this is a rt-thread under control of a non-rt executable. To allow temporal debugging the RTLinux tracer was designed that we will briefly introduce here.

```
root@rtl15:/usr/src/rtlinux/debugger # cd ../
root@rtl15:/usr/src/rtlinux # insmod modules/mbuff.o
```

The mbuff module, contributed by thomas motylewsky, is a shared memory module, allowing to share memory between rt-threads and user-space applications. The RTLinux Tracer records important events with timestamps (system events are hard-coded, like entering and exiting the scheduler routine) and user-defined events can be included in your application that trigger writes of the buffered data into shared memory whenever a user-specified condition is met. This way you can back-trace what events lead to the event that you are watching.

```
root@rtl15:/usr/src/rtlinux # cd tracer
```

If we look into `rt_process.c` in the tracer directory and compare it with `rt_process` in `examples / measurements` then we can find the following difference. The recording of the maximum in the inner loop changes from:

```
if(diff > max_diff){
    max_diff = diff;
}
```

to include the `RTL_TRACE_USER` event recording the new absolute jitter maximum, and the trace buffer is flushed to shared memory where the user-space application can read it. (there are a few other minor changes like `#include <rtl_trace.h>` and the variable `abs_max_diff` not shown here - but those should be quite self explaining).

```
if(diff > max_diff){
    max_diff=diff;
    if(max_diff > abs_max_diff){
        abs_max_diff=max_diff;
        rtl_trace2(RTL_TRACE_USER,(long) abs_max_diff);
        rtl_trace2(RTL_TRACE_FINALIZE,0);
    }
}
```

Now every time a new maximum jitter value is encountered the buffer will be flushed, this way we can trace the path of events that leads to a jitter maximum and thus (hopefully) locate the hot-spot in the code. Now lets load the tracer module and the modified `rt_process.o` to watch it.

```
root@rtl15:/usr/src/rtlinux/tracer # insmod rtl_tracer.o
root@rtl15:/usr/src/rtlinux/tracer # insmod rt_process.o ; ./tracer | tee trace.log
P0 131828416 rtl_restore_interrupts      0x46 <c88e09cd>
P0  576 scheduler out                   0xc88ecd64 <c88ea95d>
P0  480 rtl_restore_interrupts          0x96 <c88ea96b>
P0  512 hard sti                         0 <c88e0865>
```

```

P0 576 rtl_no_interrupts          0x286 <c88e0707>
P0 480 rtl_spin_unlock           0xc88e15f8 <c88e0212>
P0 416 rtl_restore_interrupts    0x203 <c88e021d>
P0 41184 rtl_no_interrupts       0x203 <c88e01c7>
P0 448 rtl_spin_lock             0xc88e15f8 <c88e01d6>
...
---snip---
...
P0 448 rtl_restore_interrupts    0x46 <c88e09cd>
P0 640 rtl_switch_to            0xc3ab8000 <c88ea7a0>
P0 1952 scheduler out           0xc3ab8000 <c88ea95d>
P0 640 rtl_restore_interrupts    0x92 <c88ea96b>
P0 992 rtl_restore_interrupts    0x297 <c88eaea4>
P0 3488 user                    0x1cc0 <c894f325>
That was trace # 1

```

The commands for insmod and starting the tracer are concatenated to a command sequence to make sure we launch the tracer fast enough as it happen quite frequently that the maximum is reached right at the beginning and then we don't see anything. Furthermore we redirect the tracer output to a log file (`trace.log`), to terminate the tracer type `<CNTRL>-<C>`. The tracer output will scroll by on the screen as the new absolute maximum is frequently encountered at the beginning and then output will stall, if you want to produce a new worst-case maximum, then switch to a different console (`<CNTRL>-<ALT>-<F2>`) login as root again and start something like.

```
root@rtl15:~ # ls -lR /
```

Note that the RTLinux Tracer does introduce a slight distortion on the systems realtime behavior so you will not be able to find everything this way, but its about as close to temporal debugging that you can get. After we terminated the tracer we do the usual cleanup.

```

root@rtl15:/usr/src/rtlinux/tracer # rmmod rt_process
root@rtl15:/usr/src/rtlinux/tracer # rmmod rtl_tracer
root@rtl15:/usr/src/rtlinux/tracer # rmmod mbuff

```

## 6 Setting up a small filesystem from scratch

To download the latest MTD utils use the following cvs commands. MTD is an integrated part of the linux kernel, but it is mainained as an external tree aswell with additional documentation, mailing list on its home-page .

### 6.1 MTD cvs

```

cvs -d :pserver:anoncvs@cvs.infradead.org:/home/cvs login
password: anoncvs
cvs -d :pserver:anoncvs@cvs.infradead.org:/home/cvs co mtd

```

This will download linux-mtd and put it in the directory mtd. Next we patch this lates version into the kernel tree.

```

root@rtl14:/usr/src/mtd/patches# sh patchin.sh
usage: patchin.sh [-c] [-j] kernelpath
  -c -- copy files to kernel tree instead of building links
  -j -- include jffs2 filesystem
root@rtl14:/usr/src/mtd/patches# sh patchin.sh /usr/src/linux
Patching /usr/src/linux

```

```
Include Filesystems: no
Zlib-Patch needed: no
Method: Link
Can we start now ? [y/N]y
```

```
Patching MTD
drivers/mtd
drivers/mtd/chips
drivers/mtd/devices
drivers/mtd/maps
drivers/mtd/nand
include/linux/mtd
Patching done
Please update Documentation/Configure.help from
/usr/src/dev_kit/src/mtd/Documentation/Configure.help
root@rtl14:/usr/src/mtd/patches # cd ../util
root@rtl14:/usr/src/mtd/util # make
...
root@rtl14:/usr/src/mtd/util# ls -l erase
-rwxr-xr-x  1 root  root  13131 Dec 11 09:09 erase*
```

For now we will only need the flash erase tool from mtd - but mtd provides a number of utilities to manage different types of flash devices, DiskOnChip, etc. In this section we will set up a embedded filesystem on a MTD device step-by-step. It assumes that the kernel we configured in the last section for RTLlinux was configured and compiled successfully with the MTD devices as described and that the system was reboot (if necessary) to make these modules available - if this is not the case then go back first and make sure the MTD setup of your RTLlinux kernel is available before continuing.

When talking about filesystems normally there are two distinct levels of understanding - the one is 'something containing files and directories' the other is 'a structured storage area with inodes and superblock(s)'. In most cases the distinction is fairly irrelevant to users or to people installing a Linux system, in the case of an embedded device, with scarce resources, the distinction is essential as optimization takes place in both areas independently.

## 7 Using MTD flash emulation device

First we will create a filesystem on a device typically for embedded systems, now it would be hard to give a step-by-step guide for a specific flash device as these vary in the way they are accessed, fortunately MTD provides a flash-disk emulation that runs in RAM, this we will use to learn details of handling flash-devices without actually needing one. Anybody that sets up a filesystem on a specific flash that would like to contribute a step-by-step guide for that specific device is encouraged to mail it to <der.herrhoffr.at> and it then can be included in updated versions of this tutorial.

Clear the kernel's ring buffer first, so we see when messages pertaining to the setup we are doing come in. Initialize mtdram a Flash-emulation to 8MB and make it behave like a NAND with a 128K erase block size. If you want to build a filesystem for a specific device, check its erase size first.

```
rtl14:~ # dmesg -c
rtl14:~ # modprobe mtdram total_size=8192 erase_size=128
rtl14:~ # lsmod
Module                Size  Used by    Not tainted
mtdram                 1596   0 (unused)
mtdcore                2500   1 [mtdram]
...
```

if the modules are not in the system's default location where modprobe finds them then manually install them

```
rtl14:/usr/src/mtd # insmod mtdcore.o
```

```
rtl14:/usr/src/mtd # insmod mtdram.o total_size=8192 erase_size=128
rtl14:/usr/src/mtd # dmesg
mtd: Giving out device 0 to mtdram test device
```

if you did not install the mtd utils in a system wide location add the path to you \$PATH variable before the default path so that we find the right tool version.

```
rtl14:/usr/src/mtd # export PATH="/usr/src/mtd/util:$PATH"
```

Next we need to erase the flash device - so check first if the MTD specific device files exist in /dev, if they don't then look up the device major/minor in the kernel documentation linux/Documentation/devices.txt or use the MAKEDEV shell script found in mtd's util directory, if you did not yet create them during the util install step (see chapter 4). So let's erase the 'flash' device with mtd's erase function.

```
rtl14:/usr/src/mtd # cd util
rtl14:/usr/src/mtd/util # erase /dev/mtd0
```

load the block emulation layer so that linear devices like flash/RAM behave like a block-device.

```
rtl14:~ # modprobe mtblock
```

now create a filesystem on the first mtblock device

```
rtl14:~ # mkfs.minix /dev/mtblock0
1376 inodes
4096 blocks
Firstdatazone=47 (47)
Zonesize=1024
Maxsize=268966912
```

this shows a few interesting things - you see the limit in the available inodes and the maximum filesystem size that this filesystem can support (that's the old 2GB limit) mount it like you would mount a regular block device.

```
rtl14:~ # mount -t minix /dev/mtblock0 /mnt/
```

after mounting the new filesystem df should show you something like

```
rtl14:~ # df
Filesystem          1k-blocks      Used Available Use% Mounted on
/dev/hda6            18627528    4128740  13921896  23% /
/dev/hda5             52627         3550    46269    8% /boot
shmfs                47288          0    47288    0% /dev/shm
/dev/mtblock0        4049           1     4048    1% /mnt
```

this shows you that the filesystem is mounted and it also shows you the filesystem overhead - minix used up 48 1k blocks to setup the filesystem structure, so from the 4096K that mtblock0 occupies only 4048 are available - this is not relevant for a hard-disk based system but for a flash or RAM based system this is relevant.

## 7.1 filesystem in a file

First we only are interested in the filesystem issues themselves so we will eliminate the hardware level by simply using a file as the 'byte-bucket' for our filesystem. The procedure described here can later be used to build your self a ramdisk image, but the primary purpose here is to see a few specifics of filesystems for embedded devices.

The first thing we need is a 'bit-bucket' that we can play with - so let's create a file of 8MB size, with

```
rtl14:/tmp # dd if=/dev/zero of=myfs bs=512 count=16384
16384+0 records in
16384+0 records out
rtl14:/tmp # ls -l myfs
-rw-r--r--  1 root    root      8388608 Dec 10 21:19 myfs
```

This is just 8MB of 0 - which Linux recognizes as ascii-text.

```
rtl14:/tmp # file myfs
myfs: ASCII text
```

To make this into a filesystem we must format it - that is structure the bit-bucket - to do this we use the regular Linux filesystem tools for ext2 (that should be available on really any system).

```
rtl14:/tmp # mkfs.ext2 myfs
mke2fs 1.32 (09-Nov-2002)
myfs is not a block special device.
Proceed anyway? (y,n) y
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
2048 inodes, 8192 blocks
409 blocks (4.99%) reserved for the super user
First data block=1
1 block group
8192 blocks per group, 8192 fragments per group
2048 inodes per group
```

```
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```

This filesystem will be automatically checked every 30 mounts or 180 days, whichever comes first. Use tune2fs -c or -i to override.

The first thing we get is a warning as mkfs.ext2 was expecting a block-device and not a file - its safe to say y here. If we look at what the file is now.

```
rtl14:/tmp # file myfs
myfs: Linux rev 1.0 ext2 filesystem data (mounted or unclean)
```

Now this IS a filesystem - it basically contains no files and it actually only contains two directory that is the root-directory of this filesystem or the directory /. and the ext2 specific /lost+found, thats all - other than that it's empty.

Note that the output is quite dependant on the ext2fs tools in use - on SuSE 7.0 systems file shows myfs: Linux/i386 ext2 filesystem

For the next steps you need to be root on your system as we are now going to mount the filesystem. So become root now (use su -, so you inherit the proper root environment, not only su).

```
rtl14:/tmp # mkdir /myfs
rtl14:/tmp # mount -t ext2 /tmp/myfs /myfs
mount: /tmp/myfs is not a block device (maybe try '-o loop'?)
```

Linux is very user frindly so it tels us that mount wants a block device or we need to tell it that we are loop mounting something, so if you don't have the loop device statically compiled into your kernel then load the module first. To see if the loop device is available do a ksyms -a — grep loop which should return something similar to what you see here.

```

rtl14:/tmp # ksyms -a | grep loop
c0271aa8 loops_per_sec
c0189cec dev_loopback_xmit
c0289320 loopback_dev
c01c2364 loop_register_transfer
c01c238c loop_unregister_transfer

```

The loopback\_dev is what shows us that the loop device is available. if not then use insmod loop to load it - if that fails then go back to the kernel configuration step... Note that if you have versioning turned on in your kernel then the symbols all look something like loopback\_dev\_Rf5a6ec6d, but thats the same thing - just the checksum added.

Now we can mount it, and check its size with df.

```

rtl14:/tmp # mount -t ext2 -o loop /tmp/myfs /myfs
rtl14:/tmp # df
Filesystem          1k-blocks      Used Available Use% Mounted on
....
/tmp/myfs            7931           13      7509    1% /myfs

```

Here we see the first surprise - the 8MB is only 7931 k not 8192. The 13 used are from the lost+found directory - so lets remove that fist as it is kind of useless on a ramdisk-filesystem (don't do that on your desk-top system though). The lost+found is used by ext2 to put files that it finds after a filesystem crash that don't have an associated directory any more - so thats where its name comes from.

```

rtl14:/tmp # cd /myfs
rtl14:/myfs # ls -l
total 14
drwxr-xr-x  3 root    root          1024 Jan  8 20:20 .
drwxr-xr-x 25 root    root           604 Jan  8 07:30 ..
drwxr-xr-x  2 root    root        12288 Jan  8 20:20 lost+found
rtl14:/myfs # rmdir lost+found
rtl14:/myfs # df /myfs
Filesystem          1k-blocks      Used Available Use% Mounted on
rtl14:/myfs # ls -l
/tmp/myfs            7931           1      7521    1% /fs
total 2
drwxr-xr-x  2 root    root          1024 Jan  8 20:51 .
drwxr-xr-x 25 root    root           604 Jan  8 07:30 ..

```

Note that the directory . is occupying 1024 bytes and is part of myfs, .. is part of the root-fs of the filesystem that contains the mount-point /myfs and happens to be reiserfs in this case - so thats where the 604 bytes come from, but thats not part of our filesystem.

So this is really an empty filesystem now - we still only have 7931 blocks available - one is in use for . but where is the rest ? - fist we are missing 132 blocks from the displayed size to the creation size of 8MB and then we are missing some blocks from the displayed available and used that don't fill up to the total size of 7931 blocks. Now on your desk-top system this is also the case, but we are talking about a few kilo byte here which you simply don't care about on a 10GB system, for embedded systems this is relevant so we will first try to 'optimize' the low level filesystem before we bother with optimizing the filesystem contents size.

The first set of missing blocks, that is 8192-7931 is the overhead of the filesystem itself, that is the inodes and the superblock(s), the second missing bits from 7931-7521, this second part is the area reserved for the root-user, that is if the user floods the filesystem then it will flood at 7521 and the 410 blocks 'missing' will only be granted to root, this ensures that even on a flooded filesystem the root user can still operate. So these 410 blocks are what mkfs.ext2 reported with

```
409 blocks (4.99%) reserved for the super user
```

This is the closest mkfs.ext2 can get to the default 5% blocks to reserve for root.

Before proceeding with the possibilities for filesystem creation we scan the basics of the ext2 filesystem.

## 7.2 Ext2 filesystems basics

You can skip this section if you like, this section should just scan some basic concepts for the ext2 fs and give you a feeling for filesystems in general and how you might go about optimizing the filesystem layer - in many cases reading the manual page to the `mkfs.FILESYSTEM` will be sufficient to tweek most of it. For a detailed coverage of the ext2 filesystem see [and](#) .

On to ext2.

The ext2 filesystem is more or less the native Linux fs, atleast its supported by every distribution and actually it is quite efficient even for embedded systems if you use it the right way. First I'll list some of the optimization strategies that ext2 employs and then we'll go through the possible consequences for embedded filesystem layout.

- the amount of space reserved for the root-user can be set.
- block sizes can be set at filesystem creation time (1024-4096)
- the number of inodes can be set
- symbolic links with short path names can be stored in the inode and don't require a data-block.
- support for immutable files
- reduction of fragmentation. By preallocating adjacent blocks for regular files these can grow without immediately introducing fragmentation.
- disk block optimization for harddisks by splitting disk-blocks into groups that contain inodes in adjacent tracks.

There is more to ext2, especially if you include some of the ext2-patches around for compression/encryption which unfortunately seems to be available for 2.2.X/2.3.X kernels only (anybody have a pointer to 2.4.X kernel patches ??) , ACL etc. but here we will stick to the standard ext2 features.

The first five items are of interest to embedded filesystems.

- this first option is quite obvious - it does need close consideration though.
- The block size needs to be adjusted to the average filesize of the system, this reduces fragmentation, so this means in most cases that 1024 byte data blocks will be used, if the system is fairly large or contains large data files it may well be better to use larger block sizes (lets say 90
- We noted above that the first chunk missing in the filesystem was for the inodes and the superblock(s), so if, as is quite common on embedded systems, the filesystem content is static then we know how many inodes we need, and can adjust the filesystem to provide these only, furthermore on an embedded system we often don't need the backup superblocks or atleast not in the default extent
- Especially on embedded filesystems the number of symbolic links is greater than on a desk-top system - just look at a system containing busybox, tynilgin and maby a multical-shell-script, on some filesystems there are almost ten times as many symbolic links than regular files. So offering an efficient way of storing symbolik links is an essential option for an embedded filesystem.
- The immutable flag is a real security issue - most script kiddies hacking will fail to remove there fingerprints of a filesystem that has immutable flags set for log-files :). There is naturally a risk to this too - that is that there is no simply way to reduce filesystem footprint in the running system.

## 7.3 creating an embedded Ext2 filesystems

Next we will look at these five selected items from a practical standpoint - how to actually make use of these features.

During filesystem creation we got the following line from `mkfs.ext2`:

```
410 blocks (4.99%) reserved for the super user
```

We can eliminate this by using the `-m 0` option to `mkfs.ext2`, this needs close consideration though - on some embedded devices this is fine, especially if the ext2 filesystem is not the root-filesystem and does not contain any directories where root would need to create files (i.e. `/var/lock` for lock files or `/tmp` for strange sockets or temporary files). In many cases it is better not to set it to 0 but to a very small number so that a lock-file creation or a socket is still possible for root even on a flooded filesystem. setting this to 0 we get the filesystem size equal the available size minus one block for the `.` directory in `/` of the filesystem.

```
/tmp/myfs          7931          1      7930    1% /fs
```

For the block size it's really hard to give any reasonable advice - this depends strongly on the actual systems setup, as noted above if its a data-logging system that will use 90

The third item was the number of inodes to be created - now if you create too few then you may have plenty of space left on the filesystem but no inodes and thus your half-full filesystem is unusable, on the other hand if you simply don't need that many inodes then you are allocating many of them that are totally unused. so this is again something that needs some brute force testing in the specific case. The default `mkfs.ext2` on our 8MB filesystem gave us

```
2048 inodes, 8192 blocks
```

```
resulting in 3.19
```

```
rtl14:/tmp # df /myfs
/tmp/myfs          7931          1      7930    1% /myfs
```

Now applying the options for 0 reserved block and inodes per group, mounting the filesystem and removing the lost+found directory:

```
rtl14:/tmp # umount /myfs
rtl14:/tmp # mkfs.ext2 -m 0 -N 512 /tmp/myfs
...
Fragment size=1024 (log=0)
512 inodes, 8192 blocks
0 blocks (0.00%) reserved for the super user
...
512 inodes per group
...
rtl14:/tmp # mount -t ext2 -o loop /tmp/myfs /myfs
rtl14:/tmp # rmdir /myfs/lost+found
rtl14:/tmp # df /myfs
/tmp/myfs          8123          1      8122    1% /myfs
```

we get a filesystem that is providing 99.14% for data blocks, so that give us an extra 64K of disk-space which is half the size of the busybox-1.0pre3 executable ! And comparing with the results from `mke2fs` with no options at all and no cleanup of the lost+found we improved from 91.2% to 99.14% which means we gained an additional 614k which is clearly relevant for a small ramdisk system.

```
rtl14:/tmp # df /myfs
Filesystem          1k-blocks      Used Available Use% Mounted on
/tmp/myfs            8123           1      8122    1% /myfs
```

The conclusion from this section I hope is that it is clear that not only the filesystem needs careful selection but the filesystem creation process is equally important to consider for an embedded filesystem.

## 8 Minimum Filesystem setup

now lets cd into the new filesystem location and start creating a miimum filesystem

```
rtl14:~ # cd /mnt
rtl14:/mnt # mkdir {dev,lib,bin}
rtl14:/mnt # ls
.  ..  bin  dev  lib
```

You should have the three directories dev, lib and bin. First we will copy a few device files to play with.

```
rtl14:/mnt # cp -pR /dev/null dev/
rtl14:/mnt # cp -pR /dev/zero dev/
```

We couls have used mknod aswell - see the man 1 mknod for details.

Next we will need at least libc on the filesystem, but as we are not going to do any application debugging on the embedded filesystem we can use striped libs !

```
rtl14:/mnt # cd lib/
rtl14:/mnt/lib # cp /lib/libc.so.6 ./
rtl14:/mnt/lib # ls -l
-rwxr-xr-x  1 root  root  1549556 Dec 10 22:01 libc.so.6
rtl14:/mnt/lib # strip libc.so.6
-rwxr-xr-x  1 root  root  1325040 Dec 10 22:02 libc.so.6
```

still larg but better... to make shure all apps find libc we need to add some symlinks .

```
rtl14:/mnt/lib # ln -s libc.so.6 libc
rtl14:/mnt/lib # ln -s libc.so.6 libc.so
rtl14:/mnt/lib # ls -l
lrwxrwxrwx  1 root  root  9 Dec 10 22:02 libc -> libc.so.6
lrwxrwxrwx  1 root  root  9 Dec 10 22:02 libc.so -> libc.so.6
-rwxr-xr-x  1 root  root  1325040 Dec 10 22:02 libc.so.6
```

libc should now be accessible for every app that we will put into our embedded filesystem. But what other libs do we need ?? the ldd command from the binutils package will show us what libs an application is linked to - as we need at least a shell we will start with bash.

```
rtl14:/mnt/lib # ldd /bin/bash
        libtermcap.so.2 => /lib/libtermcap.so.2 (0x40026000)
        libdl.so.2 => /lib/libdl.so.2 (0x4002a000)
        libc.so.6 => /lib/tls/libc.so.6 (0x42000000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

so bash requires quite a lot of additional libs ! copy and link them into the embedded filesystems lib directory.

```
rtl14:/mnt/lib # cp /lib/libdl.so.2 ./
rtl14:/mnt/lib # cp /lib/ld-linux.so.2 ./
rtl14:/mnt/lib # cp /lib/libtermcap.so.2 ./
rtl14:/mnt/lib # ln -s libdl.so.2 libdl.so
rtl14:/mnt/lib # ln -s libtermcap.so.2 libtermcap.so
rtl14:/mnt/lib # ls -l
-rwxr-xr-x  1 root  root  103044 Dec 10 22:04 ld-linux.so.2
lrwxrwxrwx  1 root  root  9 Dec 10 22:02 libc -> libc.so.6
lrwxrwxrwx  1 root  root  9 Dec 10 22:02 libc.so -> libc.so.6
-rwxr-xr-x  1 root  root  1325040 Dec 10 22:02 libc.so.6
lrwxrwxrwx  1 root  root  10 Dec 10 22:06 libdl.so -> libdl.so.2
-rwxr-xr-x  1 root  root  15084 Dec 10 22:04 libdl.so.2
lrwxrwxrwx  1 root  root  15 Dec 10 22:06 libtermcap.so -> libtermcap.so.2
-rwxr-xr-x  1 root  root  11784 Dec 10 22:04 libtermcap.so.2
```

that looks ok now - but it's large, so lets at least strip it all.

```
rtl14:/mnt/lib # strip *
```

will be a bit better (still 1412k ...) .

```
rtl14:/mnt/lib # cd ../bin
rtl14:/mnt/bin # ls
rtl14:/mnt/bin # cp /bin/bash ./
rtl14:/mnt/bin # ls -l
-rwxr-xr-x    1 root    root        626028 Dec 10 22:07 bash
rtl14:/mnt/bin # file bash
bash: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
  for GNU/Linux 2.2.5, dynamically linked (uses shared libs),
  stripped
```

bash is always striped (on all distributions I know at least ), so we can't do anything here.

```
rtl14:/mnt/bin # cd ..
rtl14:/mnt # du -s .
2031
```

thats bad - an embedded filesystem that only contains bash and its 2031 K ! bad - this is no good for an embedded filesystem.

## 9 Testing your filesystem

lets test this filesystem befor we go on, root has his home in /root so lets creat a /root in the embedded filesystem. And chroot into it . This means we are running off our embedded filesystem now - not using any of the system libs or executables that are installed on the host-pc.

```
rtl14:/mnt # mkdir root
rtl14:/mnt # chroot /mnt
```

```
bash-2.05b# echo $USER
root
bash-2.05b# cd /
bash-2.05b# echo $PWD
/
bash-2.05b# pwd
/
bash-2.05b# exit
rtl14:/mnt #
```

Within the embedded filesystem we can't realy do much , but we can do some things althoug there are no other executables than bash on our system - why that ?? Bash is a shell that provides a large list of built-in commands - this is to accelerate the shell and not require calling the most common shell-related commands all the time

Also note that the shell prompt reverted to the compiled in default - so this bash is running without any limits or site specific configuration .

Even though we had some built in commands - it was not enough to do anything serious - so next lets add a few standard commands.

```
rtl14:/mnt # cd bin
rtl14:/mnt/bin # cp /bin/ls ./
rtl14:/mnt/bin # ls -l ls
-rwxr-xr-x    1 root    root          67668 Dec 10 22:11 ls
```

```
rtl14:/mnt/bin # file ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1,
dynamically linked (uses shared libs), stripped
rtl14:/mnt/bin # chroot /mnt
```

back in the embedded filesystem now:

```
bash-2.05b# ls
bin dev lib root
bash-2.05# exit
rtl14:/mnt/bin #
```

ls will actually fail for some distros (like SuSE 7.0) because of missing libs - so lets copy librt.so.1 into /lib of the embedded filesystem and link it.

```
rtl14:/mnt/bin # cd ../lib
rtl14:/mnt/lib # cp /lib/librt.so.1 ./
rtl14:/mnt/lib # ln -s librt.so.1 librt.so
```

Then we should check with ldd /bin/ls if anything else is missing ? - now this is important to note - you will not always get such a nice error message, so ldd is the better way of figuring out the mandatory library than chroot'ing and failing. and don't forget to strip the new libs .

```
rtl14:/mnt/lib # ldd /bin/ls
      libc.so.6 => /lib/libc.so.6 (0x40039000)
      libpthread.so.0 => /lib/libpthread.so.0 (0x4015f000)
      /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
rtl14:/mnt/lib # cp /lib/libpthread.so.0 ./
rtl14:/mnt/lib # ln -s libpthread.so.0 libpthread.so
rtl14:/mnt/lib # strip *
rtl14:/mnt/lib # cd ..
rtl14:/mnt # du -s .
```

even less impressive now - the filesystem has bash and ls on it and thats not a wild functionality we get for 2381 K (on SuSE 7.0)! - On Slackware 9.1 ls unfortunately (for a tutorial) does not need any additional libs so it works

```
rtl14:/mnt # chroot /mnt/
bash-2.05b# ls -l
drwxr-xr-x   2 0      0          1024 Dec 10 21:11 bin
drwxr-xr-x   2 0      0          1024 Dec 10 21:01 dev
drwxr-xr-x   2 0      0          1024 Dec 10 21:07 lib
drwxr-xr-x   2 0      0          1024 Dec 10 21:10 root
```

ls reports the UID/GID as numeric values - thats not very user-frindly so we need to add some system config files so that ls can resolve the UID/GID corectly - this is where we need to create /etc in the embedded filesystem.

```
bash-2.05# exit
exit
rtl14:/mnt #
rtl14:/mnt # mkdir etc
rtl14:/mnt # cd etc
rtl14:/mnt/etc # cp /etc/passwd ./
rtl14:/mnt/etc # cp /etc/shadow ./
rtl14:/mnt/etc # cp /etc/group ./
rtl14:/mnt/etc # cp /etc/gshadow ./
```

And because we are using bash we copy `/etc/profile` as well - we don't want unlimited users on an embedded system !

```
rtl14:/mnt/etc # cp /etc/profile ./
rtl14:/mnt/etc # cp /etc/bashrc ./
```

`/etc/bashrc` or `/etc/bash.bashrc` (names vary a bit between different distributions) is specific to the shell and not everyone will need it but lets drop it in as we are using bash.

```
rtl14:/mnt/etc # chroot /mnt
bash-2.05# ls -l
total 3
drwxr-xr-x    2 0          0          1024 Mar 18 10:54 bin
```

`ls` is still showing UID/GID in numeric form - why that ?? `ldd /bin/ls` showd us the libs `ls` needs but it will not really show all dependancies - what we are missing is `libnss_files` . This is the limitation of `ldd` - `ldd` will only ensure you that the command you moved to the embedded filesystem works but possible system dependencies are not covered.

```
bash-2.05# exit
exit
rtl14:/mnt/etc #
rtl14:/mnt # cd ../lib
rtl14:/mnt/lib # cp /lib/libnss_files.so.2 ./
rtl14:/mnt/lib # chroot /mnt
bash-2.05# ls -l
total 6
drwxr-xr-x    2 root      root      1024 Dec 10 21:26 bin
...
```

Success ! so the dependancies of a simple command like `ls` can be quite hard to figure out - note that `libnss_files` is not really a requirement of `ls` but the system needs it to resolve UID/GID corectly - you might be using `nis` ? so to make it really clean we need to add `/etc/nsswitch.conf` and then it's clean.

## 10 System database issues

Various functions in the C Library need to be configured to work correctly in the local environment. Traditional unix simply had the files `/etc/passwd` `/etc/groups` etc. other resources like NIS were compiled into `libc` (with static paths...), GNU C Library 2.x (`libc.so.6`) is based on a clean solution, derived from Sun Microsystems design, this scheme called "Name Service Switch" (NSS) has one configuration file that sets their lookup order and what databases to use in `/etc/nsswitch.conf`.

```
rtl14:/mnt/lib # cd ../etc
rtl14:/mnt/etc # cp /etc/nsswitch.conf ./
```

On embedded systems you might concider removing some of these standard features to reduce filesystem usage - but don't forget that this way of sving space can break compatibility to the desk-top system . you also need to check the config files to make shure they don't contain more than you want - the minimum `nsswitch.conf` to only use local databasefiles (`/etc/passwd` `/etc/groups` etc.) is:

```
# minimum /etc/nsswitch.conf
passwd:      files
shadow:     files
group:      files

hosts:      files dns
services:  files
```

```
protocols:    files
rpc:          files
ethers:       files
publickey:    files

aliases:      files
```

To check other problems we will need a simple tool `/usr/bin/env` - we could output all variables by calling `echo $...` but `/usr/bin/env` is kind of handy so we will add it. The usual procedure, create directory - check for libs copy and strip.

```
rtl14:/mnt # mkdir -p usr/bin
rtl14:/mnt # ldd /usr/bin/env
                libc.so.6 => /lib/libc.so.6 (0x40027000)
                /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
rtl14:/mnt # cp /usr/bin/env usr/bin/
rtl14:/mnt # strip usr/bin/env
```

`env` is ok - we don't need any additional libs here.

Now we can start building a system - we have a shell and `ls` to check the system setup a bit. so we need something to create files and to dump their output to the terminal.

```
rtl14:/mnt # cd bin
rtl14:/mnt/bin # cp /bin/touch ./
rtl14:/mnt/bin # cp /bin/cat ./
rtl14:/mnt/bin # file *
...
```

They are all striped already (they should be!). and if you run `ldd` on them it will show that we don't need any additional libs yet. add `rm` to remove files we can now create with `cat` and `touch`.

```
rtl14:/mnt/bin # chroot /mnt
rtl14:/mnt/bin # cp /bin/rm ./
rtl14:/mnt/bin # ldd rm
                libc.so.6 => /lib/libc.so.6 (0x40027000)
                /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

So `rm` will work - file would show us that it already is striped.

This filesystem is large and very limited, some embedded GNU/Linux developers would be really disgusted at this point... but it has one clear advantage it is fully compatible with your desk-top system, everything behaves as you would expect - shell-expansion - all options to `ls` are there. For some embedded systems this is ok if you have the resources stick to this compatible solution, it will make life a lot easier not only during development but also during end-user operation as any body acquainted with a GNU/Linux system will have no problem handling this device.

## 11 first cleanup

The prompt is not nice and the hostname is missing and the system can't beep yet ... a few more things to add.

```
rtl14:/mnt # cp -p /etc/hosts etc/
rtl14:/mnt # chroot /mnt
bash-2.05b#
```

This does it on SuSE 7.0 but not on Slackware 9.1 - we could start guessing what files to copy into the filesystem until we fix this - but the better way is to use `strace` to find out what files were being searched for.

```

rtl14:/mnt # strace -f chroot /fs 2>&1 | grep open
...
open("/etc/nsswitch.conf", 0_RDONLY) = 3
open("/lib/libnss_files.so.2", 0_RDONLY) = 3
open("/etc/passwd", 0_RDONLY) = 3
open("/root/.bashrc", 0_RDONLY|0_LARGEFILE) = -1 ENOENT (No such file or directory)
...

```

This shows a horribly long list of dependencies - after playing around we find that it is `/root/.bashrc` that references `/etc/bashrc` which then executed `/usr/bin/hostname` to set the prompt. `strace` helps a lot here, but it is not able to resolve any commands from within shell-scripts - so one has to follow the list of scripts that show up to locate such problems - which can be quite a mess.

Now we can easily get completely confused on what system we actually are on... Next a console so we can beep. The beep itself is an escape sequence that is defined in `/etc/profile` on SuSE 7.0 as.

```
alias beep='echo -en "\x07"'
```

to beep we need a console device and the alias.

```

rtl14:/mnt # cp -pR /dev/console dev/
rtl14:/mnt # chroot /mnt
rtl14:/# alias beep='echo -en "\x07" '
rtl14:/# beep > /dev/console

```

By now this should be recognizable as a GNU/Linux system even if not much is available - but the concept of adding executables is hopefully clear by now so you can now play with building an embedded system and learn to understand the dependencies hands-on - make yourself a larger file system for this though !

```
rtl14:/# exit
```

## 12 second cleanup

To call this system UNIX we need a few more config files:

- `fstab` - file system layout
- `group` - list of groups
- `gshadow` - group passwords
- `passwd` - user data base
- `shadow` - encrypted passwords for the users
- `inittab` - processes that are started via `init` and that need to be relaunched on exit (respawned)
- `ld.so.conf` - so the linux loader knows where to find its shared libs
- `resolv.conf` - name server resolution
- `rpc` - nfs port numbers
- `securetty` - limit root logins to these terminals
- `shells` - list of allowed shells
- `issue` - a very important file
- `fleystems` - list of supported filesystems
- `init.d/rcS` - this shell script is called after boot up and can be used to configure any system resources needed, i.e. network, launching servers etc. services

A good package to look at for some example files, as well as some useful system utilities to tune the system is `util-linux-2.12pre`, pre version at time of writing. Most of the files can simply be grabbed from the Slackware 9.1 `/etc` directory.

## 13 booting the new filesystem

To boot our minimum filesystem we prepared the kernel with ramdisk support and initrd support, what needs to be done is add a few device files that were not required for the change root environment but for standalone operation, and to set up the boot-loader properly.

Before we can pack up the filesystem we need to set up the root device file `/dev/ram` and at least one terminal device `/dev/tty0`.

```
rtl14:/mnt # cp -pR /dev/ram dev/
rtl14:/mnt # cp -pR /dev/ram0 dev/
rtl14:/mnt # cp -pR /dev/tty0 dev/
rtl14:/mnt # cp -pR /dev/tty1 dev/
```

We need to create an appropriate `/etc/fstab` for our embedded system. Our root filesystem is the `initrd` that will go into `/dev/ram0`. Other than that we only need `/proc` mounted so we can see a few things on the embedded system.

```
/dev/ram0      /                ext2  defaults
/proc          /proc           proc  defaults

rtl14:/mnt # cd
rtl14:~ # umount /mnt
rtl14:~ # cd /tmp
rtl14:/tmp # gzip -9 myfs
rtl14:/tmp # ls -l myfs.gz
-rw-r--r--    1 root    root      2019679 Dec 10 21:53 myfs.gz
```

2 MB ramdisk - that's acceptable even if this does not do much. The kernel is configured for `RAMDISK` support and `initrd`, all that needs to be done is put the image in the right location `/boot/myfs.gz`, and add some entries to `/etc/lilo.conf`

```
image = /vmlinuz
  root = /dev/ram0
  initrd = /boot/myfs.gz
  append="init=/bin/bash"
  label = myfs
  read-only
```

This tells `lilo` to boot `/vmlinuz` as the kernel, put the ramdisk located at `/boot/myfs.gz` into `/dev/ram0` and use that as root filesystem. The only somewhat strange thing is the `init=/bin/bash` kernel commandline parameter - that is needed because we have no real `init` process on our filesystem so we will spawn a single shell without any authentication... ;)

After the system comes up we see the kernel boot output and then ...

```
bash-2.05b#
```

What's that? In the change root environment we had the hostname? - the problem is that we don't have a `/proc` filesystem available because we have no `mount` command and we don't have any `init-script` that is setting all the mount points up properly.

### EXERCISE 1

Add `init`, `mount`, and the mount points that are needed. Set up a minimum `/etc/inittab` to execute a `init` script called `rc.S` located in `/etc/init.d/`.

`inittab` should spawn at least one shell on `/dev/tty0`. `rc.S` should set up the `/proc` filesystem

## 14 Busybox and Tinylogin

Busybox and Tinylogin, are multical binaries intended specifically for embedded systems. They provide a large set of functionally stripped down standard UNIX tools in a single binary. The individual functions are invoked via symbolic links that are then used internally to decide which applet to call. This way the filesystem can be minimized as instead of adding special purpose libraries to the system, any exotic functions that might be needed can simply be added to the busybox binary, providing a "built-in-lib" functionality. busybox provides the most important file and find utilities along with a shell (ash based) and some basic system utilities (modutils, init, etc). tinylogin provides the minimum set of login and authentication tools needed to build a UNIX like system. With these two packages together a fairly complete UNIX like system can be built for embedded systems, that is compatible with a normal GNU/Linux desk-top system to a point where no special training is necessary to manage such a system.

### 14.1 busybox-1.0-pre3

First we need to unzip myfs.gz in /tmp again, mount it and clean away the large binaries we put in.

```
rtl14:/tmp # gunzip myfs.gz
rtl14:/tmp # mount -t ext2 -o loop myfs /myfs
rtl14:/tmp # cd /myfs
rtl14:/tmp # rm -rf bin
rtl14:/tmp # rm -rf usr
rtl14:/tmp # mkdir -p {bin,usr/bin,usr/sbin,sbin}
\end{verbatim}
```

```
{\sf
\begin{verbatim}
[root@rtl13 tmp]# tar -tzf busybox-1.00-pre3.tar.gz
...
busybox-1.00-pre3/coreutils/mv.c
busybox-1.00-pre3/coreutils/od.c
busybox-1.00-pre3/coreutils/printf.c
...
\end{verbatim}
```

Just a reminder... before unpacking a tar archive - check where it will end up

Now unpack and configure it, basically we can leave all settings as they are by default except for the

```
{\sf
\begin{verbatim}
[root@rtl13 tmp]# tar -xzf busybox-1.00-pre3.tar.gz
[root@rtl13 tmp]# cd busybox-1.00-pre3
[root@rtl13 busybox-1.00-pre3]# make menuconfig
```

```
Installation Options --->
(/myfs) BusyBox installation prefix
```

```
*** End of BusyBox configuration.
*** Check the top-level Makefile for additional configuration options.
\end{verbatim}
```

Save your settings and exit the configuration menu, next step is just `make` followed by `make`

```
{\sf
\begin{verbatim}
```

```

[root@rtl13 busybox-1.00-pre3]# make

...
docs/busybox.net/BusyBox.html
mkdir -p docs
[root@rtl13 busybox-1.00-pre3]# make install
/bin/sh applets/install.sh "/fs/"
/myfs/bin/ash -> busybox
...
/myfs/usr/bin/uptime -> ../../bin/busybox
/myfs/usr/bin/which -> ../../bin/busybox
/myfs/usr/bin/whoami -> ../../bin/busybox
/myfs/usr/bin/yes -> ../../bin/busybox
/myfs/usr/sbin/chroot -> ../../bin/busybox
[root@rtl13 busybox-1.00-pre3]# chroot /fs
chroot: /bin/bash: No such file or directory

```

That's because busybox provides /bin/ash and not /bin/bash - simply adding a link will not do because there is no busybox applet called bash, to resolve this we simply need to change our current environment so that a valid shell can be found.

```

[root@rtl13 busybox-1.00-pre3]# export SHELL=/bin/ash
[root@rtl13 busybox-1.00-pre3]# chroot /fs

```

```

BusyBox v1.00-pre3 (2003.12.11-10:27+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

```

```
#
```

This filesystem now has some config files on it and busybox - a check of busybox with `ldd libdl.so` so we can remove that. The filesystem is now about the same size as it was with `bash,rm,ls`, but `ls -R` shows us that this time the filesystem is really providing most of the things an embedded system will ever need, aside from login related utilities - that's where `lynlogin` comes in.

```

/tmp/myfs          8123      2365      5758  30% /myfs
rtl14:/tmp # ls -R /myfs
/myfs:
bin dev etc lib root sbin usr

/myfs/bin:
ash      chown  dmesg  gunzip  ls      mv      rmdir  tar      zcat
busybox  cp     echo   gzip    mkdir  ping    sed    touch
cat      date   false  hostname mknod  ps      sh      true
chgrp   dd     fgrep  kill    more   pwd     sleep  umount
chmod   df     grep   ln      mount  rm      sync   uname

/myfs/dev:
console null ram0 tty0 tty1 zero

/myfs/etc:
bashrc  group  mtab          passwd  shadow
fstab   gshadow nsswitch.conf profile  termcap

/myfs/lib:
ld-linux.so.2 libc.so  libnss_files.so  libtermcap.so

```

```

libc          libc.so.6  libnss_files.so.2  libtermcap.so.2

/myfs/root:

/myfs/sbin:
halt ifconfig init klogd poweroff reboot route syslogd

/myfs/usr:
bin lib sbin

/myfs/usr/bin:
[      cut      env  head      logger  reset  test  uptime  yes
basename  dirname  find  id      mesg    sort  tty  which
clear     du      free  killall nslookup tail  uniq  whoami

/fs/usr/lib:
locale

/myfs/usr/lib/locale:

/myfs/usr/sbin:
chroot

```

## 14.2 tinylogin-1.4

Tinylogin provides basic user-management and login/authentication applications. This is again a multical binary, just like busybox was. The configuration is not (yet) as elegant as for busybox, to change the configuration we need to edit the top level `Config.h` file.

```

...
#define CONFIG_ADDUSER
#define CONFIG_ADDGROUP
#define CONFIG_DELUSER
...
// Enable using shadow passwords
#define CONFIG_FEATURE_SHADOWPASSWDS
//
...
#ifdef CONFIG_FEATURE_SHA1_PASSWORDS
#define CONFIG_SHA1
#endif

```

It is divided in applets to be included, features to be used, and some fine tuning of the features at the end - generally you can leave it as it is. Next are the steps to unack, make and install tinylogin.

```

rtl14:/tmp # tar -xzf tinylogin-1.4.tar.gz
rtl14:/tmp # cd tinylogin-1.4
rtl14:/tmp/tinylogin-1.4 # make
...
rtl14:/tmp/tinylogin-1.4 # make PREFIX=/myfs install
/myfs/bin/addgroup -> tinylogin
/myfs/bin/adduser -> tinylogin
/myfs/bin/delgroup -> tinylogin
/myfs/bin/deluser -> tinylogin
/myfs/bin/login -> tinylogin
/myfs/bin/su -> tinylogin

```

```

/myfs/sbin/getty -> ../bin/tinylogin
/myfs/sbin/sulogin -> ../bin/tinylogin
/myfs/usr/bin/passwd -> ../../bin/tinylogin
/myfs/usr/bin/vlock -> ../../bin/tinylogin
rtl14:/tmp/tinylogin-1.4 # export SHELL=/bin/ash
rtl14:/tmp/tinylogin-1.4 # chroot /myfs
# login
login: error while loading shared libraries: libcrypt.so.1: cannot
open shared object file: No such file or directory
# exit
rtl14:/tmp/tinylogin-1.4 #

```

Checking with ldd shows that libcrypt.so is missing so add this and then all dependencies should be fine. If you chroot to the filesystem and try to run login you might get an error due to the missing tty but this is no problem when you boot the filesystem and login on tty0, on the embedded fs one will probably not need to create or copy the pty's you need to mount the .

```

rtl14:/myfs/lib # cp /lib/libcrypt.so.1 ./
rtl14:/myfs/lib # strip libcrypt.so.1
rtl14:/myfs/lib # ln -s libcrypt.so.1 libcrypt.so
rtl14:/myfs/lib # chroot /myfs

```

BusyBox v1.00-pre3 (2003.12.11-10:27+0000) Built-in shell (ash)  
Enter 'help' for a list of built-in commands.

```

# login
Unable to determine your tty name.
# mkdir /proc
# mount -t proc proc /proc
# login

```

rtl13.hofr.at login:

This more or less complete filesystem now is a bit over 2MB - playing with the busybox and tinylogin configuration a reasonable filesystem can be created that is no more than 2MB + additional kernel modules and your embedded applications !

```

rtl14:/myfs/lib # df /myfs
/tmp/myfs          8123          2423          5700  30% /myfs

```

We need to update our /etc/fstab so that /proc is properly mounted , check /etc/inittab so that it fits busybox - busybox has its on inittab format. You can find a fairly complete busybox compatible inittab in examples/inittab in the busybox-1.00pre3 distribution.

Last thing left to do is copy the rtlinux modules to your embedded filesystem and maby other kenel modules for your entwork card add any applications you might need on your embedded system, like the monitor application for rtlinux' examples/measurements/rt\_process.o - then load these after booting with the initrd and you have a usable embedded real-time linux system up and running.

## EXERCISE 2

- add the rtlinux core modules to the embedded filesystem
- add examples/measurements/rt\_process.o and monitor
- add the rtlinux specific devices files /dev/rtf\* and /dev/mbuff
- pack it up and reboot - run the rtlinux scheduling test on your embedded filesystem.

have fun !

## References

- [1] [RTLinux/GPL on the web] *RTLinux/GPL Download Site*, <http://www.rtlinux-gpl.org>
- [2] [RTLinux Thesis] Michael Barabanov, *Rtlinux*, 1996, New Mexico Tech.
- [3] [RT-Synchronisation] V. Yodaiken: *Temporal inventory and real-time synchronisation in RTLinux/Pro*, FSMLabs Inc., 2003
- [4] J. Vidal, F. Gonzalves, I. Ripoll: *POSIX TIMERS implementation in RTLinux, RTLinux-3.2-pre3*, <http://www.rtlinux-gpl.org>
- [5] V. Yodaiken: *Priority inheritance is a non-solution to the wrong problem*, Technical report, FSMLabs Inc., 2002
- [6] [Multiboot howto] G. Schiesser, F. Bruckner, A. Staub, N Mc Guire, *Multiboot Howto*, OpenTech EDV-Research GmbH, 2003, <http://www.opentech.at/howtos/multiboot-howto/html/index.html>
- [7] Linus Torvalds , 2002, *Linux kernel Home-Page*, <ftp://ftp.kernel.org/pub/linux/kernel/v2.4/>,
- [8] David Woodhouse , 2002, *Memory Technology Devices*, <http://linux-mtd.infradead.org>,
- [9] Daniel P. Bovet and Marco Cesati, 2001, *Understanding the Linux Kernel*, O'Reilly, ISBN 0-596-00002-2.
- [10] Dave Poirier , 2002, *Second Extended File System*, <http://www.nongnu.org/ext2-doc/>,
- [11] ?? , 2002 , *Linux AE/ACL*, <http://www.bestbits.at>,
- [12] Antoine Dumesnil de Maricourt / Peter Reiter , 2002, *Transparent Compression for ext2 Filesystem*, [http://www.netspace.net.au/reiter/e2compr\\_intro.html](http://www.netspace.net.au/reiter/e2compr_intro.html).
- [13] David Anderson , 2002, *Busybox*, <http://www.lineo.com>, <ftp://ftp.lineo.com>.
- [14] Tom Fawcett , 2002, *BootDisk-HOWTO*, <http://http://www.tldp.org/HOWTO/Bootdisk-HOWTO/>.
- [15] [tinylogin] Erik Andersen , 2003, *tinylogin*, <http://tinylogin.busybox.net>.
- [16] [Linux Router project] ??? , 2002, *Linux Router Project*, <http://www.linuxrouter.org>, <ftp://ftp.linuxrouter.org>.
- [17] [minirtl] Der Herr Hofrat , 2002, *MiniRTL*, <http://www.thinkingnerds.com>, <ftp://ftp.thinkingnerds.com>.
- [18] [util-linux] Andries Brouwer (Maintainer) util-linux@math.uio.no, 2003, *util-linux-2.12pre*, <ftp://ftp.kernel.org>.
- [19] [Memory technology devices] David Woodhouse (Maintainer), 2003, *linux-mtd*, <http://linux-mtd.infradead.org/>.