



# Linux initial RAM disk (initrd) overview

Learn about its anatomy, creation, and use in the Linux boot process

Level: Intermediate

M. Tim Jones ([mtj@mtjones.com](mailto:mtj@mtjones.com)), Consultant Engineer, Emulex

31 Jul 2006

The Linux® initial RAM disk (initrd) is a temporary root file system that is mounted during system boot to support the two-state boot process. The initrd contains various executables and drivers that permit the real root file system to be mounted, after which the initrd RAM disk is unmounted and its memory freed. In many embedded Linux systems, the initrd is the final root file system. This article explores the initial RAM disk for Linux 2.6, including its creation and use in the Linux kernel.

## What's an initial RAM disk?

The *initial RAM disk (initrd)* is an initial root file system that is mounted prior to when the real root file system is available. The initrd is bound to the kernel and loaded as part of the kernel boot procedure. The kernel then mounts this initrd as part of the two-stage boot process to load the modules to make the real file systems available and get at the real root file system.

The initrd contains a minimal set of directories and executables to achieve this, such as the `insmod` tool to install kernel modules into the kernel.

In the case of desktop or server Linux systems, the initrd is a transient file system. Its lifetime is short, only serving as a bridge to the real root file system. In embedded systems with no mutable storage, the initrd is the permanent root file system. This article explores both of these contexts.

---

## Anatomy of the initrd

The initrd image contains the necessary executables and system files to support the second-stage boot of a Linux system.

Depending on which version of Linux you're running, the method for creating the initial RAM disk can vary. Prior to Fedora Core 3, the initrd is constructed using the *loop device*. The *loop device* is a device driver that allows you to mount a file as a block device and then interpret the file system it represents. The loop device may not be present in your kernel, but you can enable it through the kernel's configuration tool (`make menuconfig`) by selecting **Device Drivers > Block Devices > Loopback Device Support**. You can inspect the loop device as follows (your initrd file name will vary):

### Listing 1. Inspecting the initrd (prior to FC3)

```
# mkdir temp ; cd temp
# cp /boot/initrd.img.gz .
# gunzip initrd.img.gz
# mount -t ext -o loop initrd.img /mnt/initrd
# ls -la /mnt/initrd
#
```

You can now inspect the `/mnt/initrd` subdirectory for the contents of the `initrd`. Note that even if your `initrd` image file does not end with the `.gz` suffix, it's a compressed file, and you can add the `.gz` suffix to `gunzip` it.

Beginning with Fedora Core 3, the default `initrd` image is a compressed `cpio` archive file. Instead of mounting the file as a compressed image using the loop device, you can use a `cpio` archive. To inspect the contents of a `cpio` archive, use the following commands:

### Listing 2. Inspecting the `initrd` (FC3 and later)

```
# mkdir temp ; cd temp
# cp /boot/initrd-2.6.14.2.img initrd-2.6.14.2.img.gz
# gunzip initrd-2.6.14.2.img.gz
# cpio -i --make-directories < initrd-2.6.14.2.img
#
```

The result is a small root file system, as shown in Listing 3. The small, but necessary, set of applications are present in the `./bin` directory, including `nash` (not a shell, a script interpreter), `insmod` for loading kernel modules, and `lvm` (logical volume manager tools).

### Listing 3. Default Linux `initrd` directory structure

```
# ls -la
#
drwxr-xr-x 10 root root 4096 May 7 02:48 .
drwxr-xr-x 15 root root 4096 May 7 00:54 ..
drwxr-xr-x 2 root root 4096 May 7 02:48 bin
drwxr-xr-x 2 root root 4096 May 7 02:48 dev
drwxr-xr-x 4 root root 4096 May 7 02:48 etc
-rwxr-xr-x 1 root root 812 May 7 02:48 init
-rw-r--r-- 1 root root 172392 May 7 02:45 initrd-2.6.14.2.img
drwxr-xr-x 2 root root 4096 May 7 02:48 lib
drwxr-xr-x 2 root root 4096 May 7 02:48 loopfs
drwxr-xr-x 2 root root 4096 May 7 02:48 proc
lrwxrwxrwx 1 root root 3 May 7 02:48/sbin -> bin
drwxr-xr-x 2 root root 4096 May 7 02:48 sys
drwxr-xr-x 2 root root 4096 May 7 02:48 sysroot
#
```

Of interest in Listing 3 is the `init` file at the root. This file, like the traditional Linux boot process, is invoked when the `initrd` image is decompressed into the RAM disk. We'll explore this later in the article.

## Tools for creating an `initrd`

Let's now go back to the beginning to formally understand how the `initrd` image is constructed in the first place. For a traditional Linux system, the `initrd` image is created during the Linux build process. Numerous tools, such as `mkinitrd`, can be used to automatically build an `initrd` with the necessary libraries and modules for bridging to the real root file system. The `mkinitrd` utility is actually a shell script, so you can see exactly how it achieves its result. There's also the `YAIRd` (Yet Another Mkinitrd) utility, which permits customization of every aspect of the `initrd` construction.

### The `cpio` command

Using the `cpio` command, you can manipulate `cpio` files. `Cpio` is also a file format that is simply a concatenation of files with headers. The `cpio` file format permits both ASCII and binary files. For portability, use ASCII. For a reduced file size, use the binary version.

## Manually building a custom initial RAM disk

Because there is no hard drive in many embedded systems based on Linux, the initrd also serves as the permanent root file system. Listing 4 shows how to create an initrd image. I'm using a standard Linux desktop so you can follow along without an embedded target. Other than cross-compilation, the concepts (as they apply to initrd construction) are the same for an embedded target.

### Listing 4. Utility (mkird) to create a custom initrd

```
#!/bin/bash

# Housekeeping...
rm -f /tmp/ramdisk.img
rm -f /tmp/ramdisk.img.gz

# Ramdisk Constants
RDSIZE=4000
BLKSIZE=1024

# Create an empty ramdisk image
dd if=/dev/zero of=/tmp/ramdisk.img bs=$BLKSIZE count=$RDSIZE

# Make it an ext2 mountable file system
/sbin/mke2fs -F -m 0 -b $BLKSIZE /tmp/ramdisk.img $RDSIZE

# Mount it so that we can populate
mount /tmp/ramdisk.img /mnt/initrd -t ext2 -o loop=/dev/loop0

# Populate the filesystem (subdirectories)
mkdir /mnt/initrd/bin
mkdir /mnt/initrd/sys
mkdir /mnt/initrd/dev
mkdir /mnt/initrd/proc

# Grab busybox and create the symbolic links
pushd /mnt/initrd/bin
cp /usr/local/src/busybox-1.1.1/busybox .
ln -s busybox ash
ln -s busybox mount
ln -s busybox echo
ln -s busybox ls
ln -s busybox cat
ln -s busybox ps
ln -s busybox dmesg
ln -s busybox sysctl
popd

# Grab the necessary dev files
cp -a /dev/console /mnt/initrd/dev
cp -a /dev/ramdisk /mnt/initrd/dev
cp -a /dev/ram0 /mnt/initrd/dev
cp -a /dev/null /mnt/initrd/dev
cp -a /dev/tty1 /mnt/initrd/dev
cp -a /dev/tty2 /mnt/initrd/dev

# Equate sbin with bin
pushd /mnt/initrd
ln -s bin sbin
popd

# Create the init file
cat >> /mnt/initrd/linuxrc << EOF
#!/bin/ash
echo
echo "Simple initrd is active"
echo
mount -t proc /proc /proc
mount -t sysfs none /sys
/bin/ash --login
EOF

chmod +x /mnt/initrd/linuxrc

# Finish up...
umount /mnt/initrd
gzip -9 /tmp/ramdisk.img
```

```
cp /tmp/ramdisk.img.gz /boot/ramdisk.img.gz
```

To create an initrd, begin by creating an empty file, using `/dev/zero` (a stream of zeroes) as input writing to the `ramdisk.img` file. The resulting file is 4MB in size (4000 1K blocks). Then use the `mke2fs` command to create an ext2 (second extended) file system using the empty file. Now that this file is an ext2 file system, mount the file to `/mnt/initrd` using the loop device. At the mount point, you now have a directory that represents an ext2 file system that you can populate for your initrd. Much of the rest of the script provides this functionality.

The next step is creating the necessary subdirectories that make up your root file system: `/bin`, `/sys`, `/dev`, and `/proc`. Only a handful are needed (for example, no libraries are present), but they contain quite a bit of functionality.

To make your root file system useful, use BusyBox. This utility is a single image that contains many individual utilities commonly found in Linux systems (such as `ash`, `awk`, `sed`, `insmod`, and so on). The advantage of BusyBox is that it packs many utilities into one while sharing their common elements, resulting in a much smaller image. This is ideal for embedded systems. Copy the BusyBox image from its source directory into your root in the `/bin` directory. A number of symbolic links are then created that all point to the BusyBox utility. BusyBox figures out which utility was invoked and performs that functionality. A small set of links are created in this directory to support your init script (with each command link pointing to BusyBox).

The next step is the creation of a small number of special device files. I copy these directly from my current `/dev` subdirectory, using the `-a` option (archive) to preserve their attributes.

The penultimate step is to generate the `linuxrc` file. After the kernel mounts the RAM disk, it searches for an `init` file to execute. If an `init` file is not found, the kernel invokes the `linuxrc` file as its startup script. You do the basic setup of the environment in this file, such as mounting the `/proc` file system. In addition to `/proc`, I also mount the `/sys` file system and emit a message to the console. Finally, I invoke `ash` (a Bourne Shell clone) so I can interact with the root file system. The `linuxrc` file is then made executable using `chmod`.

Finally, your root file system is complete. It's unmounted and then compressed using `gzip`. The resulting file (`ramdisk.img.gz`) is copied to the `/boot` subdirectory so it can be loaded via GNU GRUB.

To build the initial RAM disk, you simply invoke `mkird`, and the image is automatically created and copied to `/boot`.

### An initrd Linux distribution

An interesting open source project that was designed to be a Linux distribution that fits within an initrd is Minimax. It's 32MB in size and uses BusyBox and uClibc for its ultra small size. Despite its small size, it's a 2.6 Linux kernel with a large array of useful tools.

### Alternative to the ext2 file system

While ext2 is a common Linux file system format, there are alternatives that can reduce the size of the initrd image and the resulting mounted file systems. Examples include `romfs` (ROM file system), `cramfs` (compressed ROM file system), and `squashfs` (highly compressed read-only file system). If you need to transiently write data to the file system, ext2 works fine. Finally, the `e2compr` is an extension to the ext2 file system driver that supports online compression.

## Testing the custom initial RAM disk

Your new initrd image is in `/boot`, so the next step is to test it with your default kernel. You can now restart your Linux system. When GRUB appears, press the `C` key to enable the command-line utility within GRUB. You can now interact with GRUB to define the specific kernel and initrd image to load. The kernel

### Initrd support in the Linux kernel

For the Linux kernel to support the initial RAM disk, the kernel must be compiled with the `CONFIG_BLK_DEV_RAM` and

command allows you to define the kernel file, and the `initrd` command allows you to specify the particular `initrd` image file. When these are defined, use the `boot` command to boot the kernel, as shown in Listing 5.

`CONFIG_BLK_DEV_INITRD` options.

### Listing 5. Manually booting the kernel and `initrd` using GRUB

```
GNU GRUB  version 0.95  (638K lower / 97216K upper memory)

[ Minimal BASH-like line editing is supported. For the first word, TAB
  lists possible command completions. Anywhere else TAB lists the possible
  completions of a device/filename. ESC at any time exits. ]

grub> kernel /bzImage-2.6.1
      [Linux-bzImage, setup=0x1400, size=0x29672e]

grub> initrd /ramdisk.img.gz
      [Linux-initrd @ 0x5f2a000, 0xb5108 bytes]

grub> boot

Uncompressing Linux... OK, booting the kernel.
```

After the kernel starts, it checks to see if an `initrd` image is available (more on this later), and then loads and mounts it as the root file system. You can see the end of this particular Linux startup in Listing 6. When started, the `ash` shell is available to enter commands. In this example, I explore the root file system and interrogate a virtual `proc` file system entry. I also demonstrate that you can write to the file system by touching a file (thus creating it). Note here that the first process created is `linuxrc` (commonly `init`).

### Listing 6. Booting a Linux kernel with your simple `initrd`

```
...
md: Autodetecting RAID arrays
md: autorun
md: ... autorun DONE.
RAMDISK: Compressed image found at block 0
VFS: Mounted root (ext2 file system).
Freeing unused kernel memory: 208k freed
/ $ ls
bin          etc          linuxrc      proc          sys
dev          lib          lost+found /sbin         sbin
/ $ cat /proc/1/cmdline
/bin/ash/linuxrc
/ $ cd bin
/bin $ ls
ash          cat          echo         mount         sysctl
busybox      dmesg        ls           ps            ps
/bin $ touch zfile
/bin $ ls
ash          cat          echo         mount         sysctl
busybox      dmesg        ls           ps            ps
zfile
```

## Booting with an initial RAM disk

Now that you've seen how to build and use a custom initial RAM disk, this section explores how the kernel identifies and mounts the `initrd` as its root file system. I walk through some of the major functions in the boot chain and explain what's happening.

The boot loader, such as GRUB, identifies the kernel that is to be loaded and copies this kernel image and any associated `initrd` into memory. You can find much of this functionality in the `./init` subdirectory under your Linux kernel source directory.

After the kernel and `initrd` images are decompressed and copied into memory, the kernel is invoked. Various

initialization is performed and, eventually, you find yourself in `init/main.c:init()` (subdir/file:function). This function performs a large amount of subsystem initialization. A call is made here to `init/do_mounts.c:prepare_namespace()`, which is used to prepare the namespace (mount the dev file system, RAID, or md, devices, and, finally, the initrd). Loading the initrd is done through a call to `init/do_mounts_initrd.c:initrd_load()`.

The `initrd_load()` function calls `init/do_mounts_rd.c:rd_load_image()`, which determines the RAM disk image to load through a call to `init/do_mounts_rd.c:identify_ramdisk_image()`. This function checks the magic number of the image to determine if it's a minux, etc2, romfs, cramfs, or gzip format. Upon return to `initrd_load_image`, a call is made to `init/do_mounts_rd.c:crd_load()`. This function allocates space for the RAM disk, calculates the cyclic redundancy check (CRC), and then uncompresses and loads the RAM disk image into memory. At this point, you have the initrd image in a block device suitable for mounting.

Mounting the block device now as root begins with a call to `init/do_mounts.c:mount_root()`. The root device is created, and then a call is made to `init/do_mounts.c:mount_block_root()`. From here, `init/do_mounts.c:do_mount_root()` is called, which calls `fs/namespace.c:sys_mount()` to actually mount the root file system and then `chdir` to it. This is where you see the familiar message shown in Listing 6: VFS: Mounted root (ext2 file system).

Finally, you return to the `init` function and call `init/main.c:run_init_process`. This results in a call to `execve` to start the `init` process (in this case `/linuxrc`). The `linuxrc` can be an executable or a script (as long as a script interpreter is available for it).

The hierarchy of functions called is shown in Listing 7. Not all functions that are involved in copying and mounting the initial RAM disk are shown here, but this gives you a rough overview of the overall flow.

#### Listing 7. Hierarchy of major functions in initrd loading and mounting

```

init/main.c: init
  init/do_mounts.c: prepare_namespace
    init/do_mounts_initrd.c: initrd_load
      init/do_mounts_rd.c: rd_load_image
        init/do_mounts_rd.c: identify_ramdisk_image
          init/do_mounts_rd.c: crd_load
            lib/inflate.c: gunzip
          init/do_mounts.c: mount_root
            init/do_mounts.c: mount_block_root
              init/do_mounts.c: do_mount_root
                fs/namespace.c: sys_mount
            init/main.c: run_init_process
          execve

```

## Diskless Boot

Much like embedded booting scenarios, a local disk (floppy or CD-ROM) isn't necessary to boot a kernel and ramdisk root filesystem. The Dynamic Host Configuration Protocol (or DHCP) can be used to identify network parameters such as IP address and subnet mask. The Trivial File Transfer Protocol (or TFTP) can then be used to transfer the kernel image and the initial ramdisk image to the local device. Once transferred, the Linux kernel can be booted and initrd mounted, as is done in a local image boot.

## Shrinking your initrd

When you're building an embedded system and want the smallest initrd image possible, there are a few tips to consider. The first is to use BusyBox (demonstrated in this article). BusyBox takes several megabytes of utilities and shrinks them down to several hundred kilobytes.

In this example, the BusyBox image is statically linked so that no libraries are required. However, if you need the standard C library (for your custom binaries), there are other options beyond the massive glibc. The first small library is uClibc, which is a minimized version of the standard C library for space-constrained systems. Another library that's ideal for space-constrained environments is dietlib. Keep in mind that you'll need to recompile the binaries that you want in your embedded system using these libraries, so some additional work is required (but worth it).

---

## Summary

The initial RAM disk was originally created to support bridging the kernel to the ultimate root file system through a transient root file system. The initrd is also useful as a non-persistent root file system mounted in a RAM disk for embedded Linux systems.

## Resources

### Learn

- "[Inside the Linux boot process](#)" (developerWorks, May 2006) explores the Linux boot process from the initial bootstrap to the start of the first user-space application.
- In "[Boot Linux from a FireWire device](#)" (developerWorks, July 2004), see how you can start Linux (with its initrd) from a multitude of devices on a variety of platforms.
- The [cpio file format](#) is both simple and compact. It's no wonder the Fedora team chose it as a format option for the initrd.
- The [mkinitrd](#) utility is ideal for creating initrd images. In addition to creating an initrd image, it also identifies the modules to load for your particular system and populates them into the image.
- The [loop device](#) is a great driver for mounting image files as file systems.
- The [Network Boot and Exotic Root HOWTO](#) illustrates not only booting Linux from the network, but also other interesting scenarios such as floppy boot, CD-ROM boot, and embedded scenarios.
- In the [developerWorks Linux zone](#), find more resources for Linux developers.
- Stay current with [developerWorks technical events and Webcasts](#).

### Get products and technologies

- The [cpio file format](#) (now supported as a Fedora Core initrd image format) has a long history and operates on a wide range of UNIXes.
- The [ash shell](#) is a Bourne Shell clone (mostly compliant) that is small, but does the job. It's great for use as a script interpreter in space-constrained Linux embedded systems.
- [BusyBox](#) is a great way to shrink the memory requirements for your next embedded Linux project.

- To shrink the size of your initrd even further, consider using a C library alternative to glibc, such as [uClibc](#) or [dietlib](#). If you prefer C++, you can try the alpha version of the [uClibc++](#) library.
- [Minimax](#) is a Linux distribution that fits entirely in an initrd image!
- Order the [SEK for Linux](#), a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With [IBM trial software](#), available for download directly from developerWorks, build your next development project on Linux.

## Discuss

- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

## About the author



M. Tim Jones is an embedded software architect and the author of *GNU/Linux Application Programming*, *AI Application Programming*, and *BSD Sockets Programming from a Multilanguage Perspective*. His engineering background ranges from the development of kernels for geosynchronous spacecraft to embedded systems architecture and networking protocols development. Tim is a Consultant Engineer for Emulex Corp. in Longmont, Colorado.

IBM, DB2, Lotus, Rational, Tivoli, and WebSphere are trademarks of IBM Corporation in the United States, other countries, or both. Linux is a trademark of Linus Torvalds in the United States, other countries, or both. UNIX is a registered trademark of The Open Group in the United States and other countries. Other company, product, or service names may be trademarks or service marks of others.