Building an Embedded System Based on the Initial RAM Disk

By **Pavel Andris** - March 7, 2014

Here's a rough tutorial on writing an embedded Linux based system, using an initial RAM disk.

The initial RAM disk (initrd) is a tool for loading a temporary root file system into the memory during the Linux boot process. The files of the system perform some initialisation jobs. A boot loader loads the Linux kernel and the initrd into the memory, starts the kernel and informs it about the location of the initrd. The kernel mounts the initrd as its initial root file system at the end of its boot sequence. Then the kernel executes the /linuxrc file as its first process. When the /linuxrc file exits, the kernel assumes that it has mounted the final root file system.
Today, there are more modern methods to achieve this goal-initramfs and cramfs, though I've never tried them.

The idea is to put an embedded application as the */linuxrc* file into the initrd. The system will never go beyond its initrd stage. No Linux daemons will haunt the embedded application.

This works fine for me but there is a disadvantage. The embedded application is resident twice in the embedded computer memory: first as a part of the initrd file system and second, as a running process. My embedded application is a small one but it needs a substantial amount of numerical data. I put the data into files outside the initrd on the embedded computer's disk. When the application starts, it reads the data from the files into the memory. If your embedded computer has enough memory, you can use the presented method directly. If not, you may want to investigate how to run an executable located outside the initrd file system. I've never tried this but I believe it is possible.
So let's get started with the tutorial.

Configuring the kernel for your embedded computer

Keep your kernel small, simple and compact. Include only those features and drivers that your embedded application is going to use. Avoid kernel modules. The kernel must have the support of the initrd (CONFIG_BLK_DEV_INITRD). Have a look at the maximum allowed size for RAM disks (BLK_DEV_RAM_SIZE). You may need to increase it.

Preparing the disk of your embedded computer

You should know whether your embedded application needs to write to the disk. For example, my application has to write some user preference files from time to time. If so, you need two primary partitions on the disk. The first one is bootable and contains some booting files, the kernel, the initrd, and possibly some data that you put outside the initrd. If the embedded application has no data stored outside of the initrd, there is no need to mount the file system located on the first partition. If needed, your application has to mount this file system as a read only one. If your application does not plan to write any data on the disk, one partition on the disk is sufficient.

The second (possibly smaller) partition and its file system contain data the embedded application can rewrite. Again, mount the file system as read-only. Just before you are going to write something, remount it as a read-write file system. Having finished the writing, remount it as a read-only file system again.

The method proposed above is a very rudimentary protection of the file systems on the embedded disk. The embedded application is capable of running even if the second file system is damaged. Of course, you should protect the embedded system from sudden power offs while writing to the file system, if possible.

Linking your embedded application

Try to link your embedded application with static libraries (lib*.a), not with shared libraries (lib*.so). You can do it if you write your application as just one (possibly multi-thread) process. Linking the application with the static libraries provides a smaller memory and disk footprint of the embedded system, because only the really used library functions are added.

The static linking is ineffective or impossible if you have to use functions such as system(), popen(), fork(), and so on. You'll have to copy some shared libraries into your initrd. The ldd command will tell you which ones.

I had started with a statically linked embedded application but later developments forced me to use the shared libraries.

Choosing a bootloader

This depends on the file system you plan to use for your embedded disk. As I have ext file systems on my embedded disk, I've chosen the extlinux branch of the syslinux package by Peter Anvin (www.syslinux.org). It supports several file systems including ext2, ext3 and ext4. Read the documentation and install it. If you don't like looking at Peter Anvin's copyright sign during the boot process, you can display a nice boot picture of your own.

Creating and populating your initrd

There are some special tools for making an initrd, but they are not useful here. Create a file of appropriate size, as follows:

```
dd if=/dev/zero of=initrd.img bs=$SIZE count=1
```

…where $SIZE should be just big enough to accommodate all files and directories of the initrd file system. Let's convert initrd.img into a file system, as follows:

```
mke2fs -F -m 0 -N 100 initrd.img
```

and mount it to a local directory:

```
mkdir ./mnt; mount -t ext2 -o loop initrd.img ./mnt
```

Now, you can make the necessary directories using commands like:

```
mkdir ./mnt/dev
```

You'll probably need other directories. This would depend on the organisation of your application and on the other programs your application is going to run. If you plan to mount your embedded disk partitions, add mount points. Use the mknod command to make device nodes for devices your application needs; for example:

```
mknod ./mnt/dev/sda  b 8 0
mknod ./mnt/dev/ram0 b 1 0
```

Here, /dev/ram0 is the root device. Copy your embedded application ($PROGRAM) as 'linuxrc' to the initrd:

```
cp -p $PROGRAM ./mnt/linuxrc
```

If you use shared libraries and/or other programs, copy them, too. Finally, unmount the new initrd:

```
umount ./mnt
```

Now, you have got your initrd containing all the necessary items in the *initrd.img* file.

Populating your embedded disk

Let us suppose that the embedded disk has the device node
/dev/sda and /dev/sda1 is its first and bootable partition. Create a file system on the partition and mount it, as follows:

```
mke2fs -m 0 -N 200 /dev/sda1
tune2fs -c 0 -i 0 /dev/sda1
mount -t ext2 /dev/sda1 ./mnt
```

Set the master boot record on your embedded disk as shown below:

```
cat /usr/share/syslinux/mbr.bin > /dev/sda
```

The mbr.bin file is a part of the SYSLINUX software package mentioned above. Copy your embedded kernel, initrd and extlinux.conf files:

```
cp bzImage ./mnt/bzImage
cp initrd.img ./mnt/initrd.img
cp extlinux.conf ./mnt/extlinux.conf
```

The extlinux.conf file is a text file and should contain something like:

```
default linux
label linux
display boot_picture
kernel bzImage
append initrd=initrd.img ramdisk_size=40000 root=/dev/ram0 init=/linuxrc
```

You will probably need more append parameters to treat your peripherals. If there are some files belonging to the embedded application that you have decided to put outside the initrd, copy them to the embedded disk, too. Now, run the following command:

```
extlinux -i ./mnt
```

It will install the SYSLINUX boot loader on your embedded disk. Unmount the ./mnt directory. Now, you can boot from the embedded disk. The procedure described above will start your embedded application at the end of the boot process.

Acknowledgement

This publication is the result of the project: "Robust sensory system for industrial environments with high pressures, temperatures and high degree of electromagnetic interference", ITMS code 26240220037, supported by the Research & Development Operational Program funded by the ERDF (European Regional Development Fund) that supports research activities in Slovakia. The project is co-financed by the EU resources.

Pavel Andris

The author has been writing hard real-time embedded software for various machines since 1981. He works at the Institute of Informatics, Bratislava, Slovakia.