initramfs is a root filesystem which is embedded into the kernel and loaded at an early stage of the boot process. It is the successor of *initrd*. It provides *early userspace* which lets you do things that the kernel can't easily do by itself during the boot process.

Using *initramfs* is optional. By default, the kernel initializes hardware using built-in drivers, mounts the specified root partition, loads the init system of the installed Linux distribution. The init system then loads additional modules and starts services until it finally allows you to log in. This is a good default behaviour and sufficient for many users. *initramfs* is for users with advanced requirements, for users who need to do things as early as possible, before the root partition is mounted.

Here are some examples of what you can do with *initramfs*:

- Customize the boot process (e.g. print a welcome message, boot splash, ...)
- Load modules (e.g. a third party driver that can not be integrated into the kernel directly)
- Mount the root partition (for encrypted, logical, and otherwise special partitions)
- Provide a minimalistic rescue shell (if something goes wrong)
- Anything the kernel can't do (as long as you can do it in user space, e.g. by executing commands)

If you don't have advanced requirements, you do not need *initramfs*.

Contents

Prerequisites

There are countless ways to make an *initramfs*. You can choose not to create an *initramfs* at all but let other apps, such as genkernel, do the work for you. If you are lucky, *genkernel* does what you want out of the box, and you don't need to bother with how *initramfs* works and what it does anymore. If you're unlucky, *genkernel* does not do what you want and you have to build an *initramfs* all by yourself.

The *initramfs* usually contains at least one file, /init. This file is executed by the kernel as the main init process (PID 1). It has to do all the work. In addition, there can be any number of additional files and directories that are required by /init. They are usually files you will also find on any other root filesystem, such as /dev for device nodes, /proc for kernel information, /bin for addition binaries, and so on. The structure of the *initramfs* can be simple, or it can be complicated, depending on what you are planning to do.

When the kernel mounts the *initramfs*, your target root partition is not yet mounted, so you can't access any of your files. That means there is nothing but the *initramfs*. So everything you need, everything you want, you have to include it in your *initramfs*. If you want a shell, you have to include it in your *initramfs*. If you want to mount something, you need a mount utility. If you need to load a module, your *initramfs* has to provide both the module, as well as a utility to load it. If the utility depends on libraries in order to work, you have to include the libraries as well. This seems complicated, and it is, because the *initramfs* has to function independently.

Basics

In this section you will learn the *easy and straightforward way to initramfs creation*. You will make a functional - albeit minimalistic - *initramfs* which you then can extend according to your own requirements.

Directory Structure

Create the directory that will later become your *initramfs* root. For consistency. we'll work in /usr/src/initramfs, but any directory would do. Create initramfs root directory and cd into it.

```
mkdir /usr/src/initramfs
cd /usr/src/initramfs
```

Create a basic directory layout.

```
mkdir -p bin lib dev etc mnt/root proc root sbin sys
```

Device Nodes

Most things you do in *initramfs* will require a couple of device nodes to be present, especially the device for your root partition. Throughout this document, /dev/sda1 will be used as example device. Copy basic device nodes. Which devices you need exactly depends entirely on what you are going to use *initramfs* for. Please adapt to your own needs.

```
cp -a /dev/{null,console,tty,sda1} /usr/src/initramfs/dev/
```

Note: More advanced approaches to device nodes are covered later in the Initramfs#Functionality section.

Applications

Any binary you want to execute at boot needs to be copied into your *initramfs* layout. You also need to copy any libraries that your binaries require. To see what libraries any particular binary requires, use the tool ldd. An example examining what libraries app-text/nano requiries:

ldd /usr/bin/nano
```
    linux-gate.so.1 =>  (0xb7f8a000)
    libncursesw.so.5 => /lib/libncursesw.so.5 (0xb7f2e000)
    libc.so.6 => /lib/libc.so.6 (0xb7dbb000)
    /lib/ld-linux.so.2 (0xb7f8b000)
```

Here you see that for app-text/nano to work in our *initramfs*, we not only need to copy /usr/bin/nano to /usr/src/initramfs/bin, but also /lib/libncursesw.so.5, /lib/libc.so.6 and /lib/ld-linux.so.2 to /usr/src/initramfs/lib. Note that you don't need the linux-gate.so.1.

Additionally, some applications may be dependent on other files but libraries to work. app-text/nano for example needs a terminfo file /usr/share/terminfo/l/linux from the ncurses package, that has to be copied to your *initramfs*. To find these dependencies, tools like equery and strace proves to be helpful.

Busybox

Instead of collecting countless utilities and libraries (and never seeing the end of it), you can just use sys-apps/busybox. It's a set of utilities for rescue and embedded systems, it contains a shell, utilities like ls, mkdir, cp, mount, insmod, and many more - all in a single binary called /bin/busybox. For busybox to work properly in a *initramfs* you'll firstly need to emerge sys-apps/busybox with the **static** USE flag enabled, then copy the /bin/busybox binary into your *initramfs* layout as /usr/src/initramfs/bin/busybox:

Note: Without the **static** USE flag, the static busybox binary may be called /bin/bb. Check with ldd that you're copying a static binary. Also take note that enabling **static** will override the pam USE flag and disable PAM support.
cp -a /bin/busybox /usr/src/initramfs/bin/busybox

Init

The file structure of your *initramfs* is almost complete. The only thing that is missing is /init, the executable in the root of the *initramfs* that is executed by the kernel once it is loaded. Because sys-apps/busybox includes a fully functional shell this means you can write your /init binary as a simple shell script (instead of making it a complicated application written in Assembler or C that you have to compile).

The following example realizes this executable as a minimalistic shell script, based on the busybox shell:

File: /usr/src/initramfs/init
```
#!/bin/busybox sh

# Mount the /proc and /sys filesystems.
mount -t proc none /proc
mount -t sysfs none /sys

# Do your stuff here.
echo "This script mounts rootfs and boots it up, nothing more!"

# Mount the root filesystem.
mount -o ro /dev/sda1 /mnt/root
```

```
# Clean up.
umount /proc
umount /sys

# Boot the real thing.
exec switch_root /mnt/root /sbin/init
```

This example needs some device nodes to work, mainly the root block device. Change the script and copy the the corresponding file from /dev/ to fit your needs.

Lastly, make the /init executable:

chmod +x /usr/src/initramfs/init

Packaging Your Initramfs

Your *initramfs* now has to be made available to your kernel at boot time. This is done by packaging it as a compressed cpio archive. This archive is then either embedded directly into your kernel image, or stored as a separate file which can be loaded by grub during the boot process. Both methods perform equally well, simply choose the method that you find most convenient.

Kernel Configuration

With either method, you need to enable support for *Initial RAM filesystem and RAM disk (initramfs/initrd) support* for the *initramfs* functionality.

Linux Kernel Configuration: Enabling the initramfs

```
General setup  --->
    [*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
```

Embedding into the Kernel

If you want the *initramfs* to be embedded into the kernel image, edit your kernel config and set *Initramfs source file(s)* to the root of your *initramfs*, (e.g /usr/src/initramfs):

Linux Kernel Configuration: Enabling the initramfs

```
General setup  --->
    (/usr/src/initramfs) Initramfs source file(s)
```

Now when you compile your kernel it will automatically compress the files into a cpio archive and embed it into the bzImage. You will need to recompile the kernel any time you make any changes to your *initramfs*.

Creating a Separate File

You can create a standalone archive file by running the following commands:

cd /usr/src/initramfs
find . -print0 | cpio --null -ov --format=newc | gzip -9 > /boot/my-initramfs.cpio.gz
This will create a file called *my-initramfs.cpio.gz* in your /boot directory. You now need to instruct grub to load this file at boot time, you do this with the *initrd* line.

File: /boot/grub/grub.conf

```
title=My Linux
root (hd0,0)
kernel /boot/my-kernel
```

```
initrd /boot/my-initramfs.cpio.gz
```

Finalizing

You can now reboot your machine. On boot, the kernel will extract the files from your *initramfs* archive automatically and execute your /init script, which in turn should then take care of mounting your root partition and exec the init of your installed Linux distribution.

Functionality

Now that you've covered the *initramfs* basics, in this section you will learn how to extend your /init script with more advanced functionality.

Rescue Shell

If you want to be dropped to a rescue shell if an error occurs, you can add the following function to your /init and call it when something goes wrong.

File: /usr/src/initramfs/init

```
rescue_shell() {
    echo "Something went wrong. Dropping you to a shell."
    busybox --install -s
    exec /bin/sh
}
```

This can, for example, be used to drop you into a shell if something goes wrong. In the example below we drop to the rescue_shell if the root partition fails to mount:

File: /usr/src/initramfs/init

```
mount -o ro /dev/sda1 /mnt/root || rescue_shell
```

Remote Rescue Shell

If you don't have physical access to the machine that's booting, or if it doesn't have a monitor/keyboard, you can access the rescue shell remotely via the network, using telnetd. For this to work, you will first need to populate the /etc/passwd, /etc/shadow and /etc/group files in your *initramfs*. Then you can bring up the network, and start a telnet server in your /init script.

Warning: Unlike SSH, Telnet is unencrypted and therefore poses a security risk.
In addition to the local rescue shell above, add this function:

File: /usr/src/initramfs/init

```
remote_rescue_shell() {
    # Bring up network interface
    ifconfig eth0 10.0.0.1 up

    # telnetd requires devpts
    mkdir -p /dev/pts
    mount -t devpts none /dev/pts

    # Start the telnet server
    telnetd
    # Continue with the local rescue shell
    rescue_shell
}
```

Note: Change the IP address to one that you can access.
Then, change the original rescue shell example to use remote_rescue_shell instead:

File: /usr/src/initramfs/init
```
mount -o ro /dev/sda1 /root || remote_rescue_shell
```

Dynamic Devices

If you have to work with dynamic devices, you can use either devtmpfs or mdev. Please note that the kernel can take some time detecting devices (such as external USB drives), so you may also have to add a sleep statement to your script.

devtmpfs

This is a recent addition to the Linux kernel, designed to offer device nodes early at bootup. To use it, enable **CONFIG_DEVTMPFS** in your kernel .config.

Linux Kernel Configuration: Enabling devtmpfs
```
Device Drivers  --->
        Generic Driver Options  --->
                [*] Maintain a devtmpfs filesystem to mount at /dev
```
You can include the following snippet in your /init script to have it mount at boot:

File: /usr/src/initramfs/init
```
mount -t devtmpfs none /dev
```

Don't forget to unmount it again in the cleanup phase of the script:

File: /usr/src/initramfs/init
```
umount /dev
```

mdev

Alternatively, you can use mdev, the udev replacement of busybox. For mdev to work, you have to make /sbin/mdev a symlink to /bin/busybox in your *initramfs*.

ln -s ../bin/busybox /usr/src/initramfs/sbin/mdev
Then add the following snippet to your /init, after mounting /proc and /sys:

File: /usr/src/initramfs/init
```
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

LVM

If your root partition is located on a logical volume, you need to include a static LVM binary in your *initramfs*. You can get a static binary by enabling the **static** USE-Flags for sys-fs/lvm2. Copy it to your *initramfs* sbin/ directory.

Note: The static LVM binary may also be called /sbin/lvm.static. Check with ldd that you're copying a static binary.

```
cp -a /sbin/lvm /usr/src/initramfs/sbin/
```

Now you can enable your LVM root partition in /init. This example assumes that your volume group is called **vg**, and your root volume is called **root**. Replace them with the names you chose when creating the volume.

File: /usr/src/initramfs/init

```
lvm vgscan --mknodes
lvm lvchange -aly vg/root
```

Your root partition may then be called /dev/vg/root or /dev/mapper/vg-root.

DM-Crypt

If your root partition is encrypted, you need to include a static cryptsetup binary in your *initramfs*. You can get a static binary by setting the **static** USE-Flag for sys-fs/cryptsetup (versions prior to 1.2.0 need the **-dynamic** USE-Flag). Copy it to your *initramfs* sbin/ directory:

```
cp -a /sbin/cryptsetup /usr/src/initramfs/sbin/
```

If you are using cryptsetup-1.2.0 or later then you will need the random generator device files otherwise you will receive an error (*Fatal error during RNG initialisation. Cannot initialize crypto backend.*) and will be unable to decrypt. For this just add the device files to your *initramfs*:

```
cp -a /dev/{random,urandom} /usr/src/initramfs/dev/
```

Now you can unlock your encrypted root partition in /init:

File: /usr/src/initramfs/init

```
cryptsetup -T 5 luksOpen /dev/sda1 luks
```

Once you entered your passphrase, your root partition will be available as /dev/mapper/luks.

GnuPG

Note: You can avoid GnuPG in *initramfs* altogether by using cryptsetup with Initramfs#Loop_Device for encrypted keyfiles.

If you want to use an encrypted keyfile for unlocking your encrypted partition, you need to include app-crypt/gnupg in your *initramfs*. Unfortunately, version 2 of GnuPG added so many dependencies that it would be quite a challenge to make it work properly in *initramfs*. For the sake of simplicity, this guide uses version 1 of GnuPG which is still available in Portage and works as a single binary. You can get a static binary by setting the **static** USE-Flag for <app-crypt/gnupg-2.

```
USE="static" emerge -1a '<app-crypt/gnupg-2'
cp -a /usr/bin/gpg /usr/src/initramfs/bin/
```

Note: Make sure to go back to the current version of gnupg after you copied the binary.

Use gpg -c ~/key to encrypt your key file symmetrically using a passphrase of your choice. Copy it into the *initramfs* as well.

```
cp ~/key.gpg /usr/src/initramfs/root/
```

Now you can decrypt the key and unlock your partition with it in /init.

File: /usr/src/initramfs/init

```
# gpg tty workaround
cp -a /dev/console /dev/tty

# decrypt the key
i=0

while [ $i -lt 5 ]
do
    i=$(($i+1))
    echo
    echo -n "(try $i) "
    gpg -d -q /root/key.gpg -o /root/key 2> /dev/null

    # Failure? Try again.
    if [ ! -e /root/key ]
    then
        sleep 5
        continue
    fi

    # Success.
    break
done

# Unlock the partition.
cryptsetup -d /root/key luksOpen /dev/sda1 luks
```

Loop Device

You can also use DM-Crypt to encrypt a keyfile via a loop device. Busybox comes with losetup, so you already have all the tools you need in your *initramfs*. First, you'll have to create a file of the desired size using dd. If you intend to use LUKS instead of pure DM-Crypt, add 2056 blocks for metadata. The following example will give you 512 bytes of usable space with LUKS. If you want to use a different size, please adapt accordingly.

dd if=/dev/urandom of=/usr/src/initramfs/root/key.iso count=2057
losetup /dev/loop0 /usr/src/initramfs/root/key.iso
cryptsetup luksFormat /dev/loop0
cryptsetup luksOpen /dev/loop0 lukskey
Afterward, /dev/mapper/lukskey will contain 512 bytes of random data which you can use as key for your real DM-Crypt containers. You can also write a new or existing key to the loop file using dd.

dd if=/path/to/existing/key of=/dev/mapper/lukskey
Unlocking the root device using this key in your /init can then be done like this:

File: /usr/src/initramfs/init
```
# Obtain key through loop device
losetup /dev/loop0 /root/key.iso
cryptsetup -T 5 luksOpen /dev/loop0 lukskey

# Unlock the root partition
cryptsetup --key-file /dev/mapper/lukskey luksOpen /dev/sda1 luks

# Clean up the loop device
cryptsetup luksClose lukskey
losetup -d /dev/loop0
```

As before, your root partition should then be available as /dev/mapper/luks.

Software RAID

Normally the Linux kernel will automatically scan for any "Linux raid autodetect" partitions (type: fd) and start as many software RAIDs as it can find. But if you use an *initramfs*, the kernel will not automatically scan for RAIDs until it is told to. In the following example we instruct the kernel to scan for software RAIDs and start as many as it can find. This will actually start all arrays, not just /dev/md0:

File: /usr/src/initramfs/init
```
raidautorun /dev/md0
```

mdadm

If you are not using "Linux raid autodetect" partitions, or need to do more advanced RAID setup, then you can use mdadm instead. You will need to include a static mdadm binary in your *initramfs*. You can get a static binary by enabling the **static** USE-Flag for sys-fs/mdadm.

Copy the binary(/sbin/mdadm) and your /etc/mdadm.conf into your *initramfs*:

```
cp -a /sbin/mdadm /usr/src/initramfs/sbin/
cp -a /etc/mdadm.conf /usr/src/initramfs/etc/
```
Edit your the mdadm.conf in your *initramfs* to your liking. An example mdadm.conf follows:

File: /usr/src/initramfs/etc/mdadm.conf
```
# mdadm configuration file

DEVICE /dev/sd??*

ARRAY /dev/md0 UUID=57b62805:84968c87:2222d7c7:02df0843
```

This mdadm.conf will scan all /dev/sd* devices and assemble the RAID device fitting the UUID *57b62805:84968c87:2222d7c7:02df0843*. You get the UUID by running:

```
mdadm -D /dev/md0
```
Now you can initialize your software RAIDs in /init:

File: /usr/src/initramfs/init
```
mdadm --assemble --scan /dev/md0
```

Networking

Note: *initramfs* may not be necessary for root on NFS.
If your root partition is only reachable via the network, or if you need to do networking for other things, all required network related drivers have to be built into your kernel, and you'll have to configure the network interfaces in your /init. How exactly this has to be done, depends on your network situation. The following sections cover only the most common cases.

Static IP

If your network situation allows you to use a static network IP, you can set it up using the ifconfig and route commands, which are included in Busybox. This is by far the easiest solution, so if it's at all possible, go for it.

File: /usr/src/initramfs/init
```
ifconfig eth0 10.0.2.15
route add default gw 10.0.2.2
```

DHCP

To obtain a dynamic IP address from your network's DHCP server, you need a DHCP client. Busybox comes with a minimalistic DHCP client called udhcpc, which is sufficient for most users. Unfortunately, udhcpc has a dependency: it requires the help of a separate script to actually configure the network interface. An example for such a script is included in the Busybox distribution, but not installed by Gentoo. You will have to obtain it directly from the Busybox tarball (it's called examples/udhcp/simple.script) or download it from the Busybox project page.

Copy the script to your *initramfs*.

cp simple.script /usr/src/initramfs/bin
Edit the script's first line to read !#/bin/busybox sh or create a symlink for /bin/sh:

ln -s busybox /usr/src/initramfs/bin/sh
Don't forget the executable flag if it's not set already.

chmod +x /usr/src/initramfs/bin/simple.script
Now, you can obtain a dynamic IP address for eth0 using DHCP:

File: /usr/src/initramfs/init
```
ifconfig eth0 up
udhcpc -t 5 -q -s /bin/simple.script
```

DNS

Your network should be up and running now. However, that's only if you know exactly which IPs to talk to. If all you have is a host or domain name, it's a different story entirely. In that case, you need to be able to resolve hostnames. Unfortunately, this is where our luck leaves us. Until now, everything could be done with just the static binary of Busybox - however, this is not the case with DNS.

Note: Additional libraries are required to make DNS work.
This is because sys-libs/glibc itself dynamically includes additional libraries for DNS lookups. As long as you don't need that functionality, you're fine, but if you need it, you have no choice but to include those libraries in your *initramfs*. The only alternative would be building Busybox against another libc such as sys-libs/uclibc, however that would go beyond the scope of this document.

Copy the necessary libraries to your *initramfs*.

cp /lib/libnss_{dns,files}.so.2 /lib/{libresolv,ld-linux}.so.2 /lib/libc.so.6 /usr/src/initramfs/lib/
Create a /etc/resolv.conf with at least one nameserver you can use. Note that this may be done automatically

for you if you use Initramfs#DHCP.

```
echo nameserver 10.0.2.3 > /usr/src/initramfs/etc/resolv.conf
```

With this, nameserver lookups should now work for you.

Kernel Modules

If you want to have your *initramfs* load kernel modules you'll need to check for modules dependencies, copy the required kernel modules to the *initramfs* and have the /init script load them. Start with checking what dependencies the modules has, in this example we want to load the ext4.ko module(EXT4 Filesystem support). This we do with modinfo and by grepping for *depends*:

```
modinfo /lib/modules/$(uname -r)/kernel/fs/ext4/ext4.ko | grep depends
```

Take note of the $(uname -r) here, you can of course substitute this with whatever kernel version intend to use. The output should be similar to:

```
depends:        mbcache,jbd2
```

Create the /usr/src/initramfs/lib/modules directory:

```
mkdir -p /usr/src/initramfs/lib/modules
```

Here we make our lives a little easier by using find to search and then copy the module and its dependencies to the *initramfs*:

```
find /lib/modules/$(uname -r)/ -iname "mbcache.ko" -exec cp {} /usr/src/initramfs/lib/modules/ \; find
/lib/modules/$(uname -r)/ -iname "jbd2.ko" -exec cp {} /usr/src/initramfs/lib/modules/ \; find
/lib/modules/$(uname -r)/ -iname "ext4.ko" -exec cp {} /usr/src/initramfs/lib/modules/ \;
```

Lastly we need to tell the /init script to load the modules:

File: /usr/src/initramfs/init

```
load_modules() {
    MODULES="mbcache.ko jbd2.ko ext4.ko"
    MOD_PATH="/lib/modules"
    for MODULE in ${MODULES} ; do
        insmod -f ${MOD_PATH}/${MODULE}
    done
}
```

Make sure the **MOD_PATH** points to the directory containing your modules and the **MODULES** variable contains the modules you want to load expressed by the modinfo output, **in order**, ending with wanted module. It's often a good idea to invoke bits like this with the rescue_shell to fall back on:

File: /usr/src/initramfs/init

```
load_modules || rescue_shell
```

Multiple initramfs

It is possible to specify multiple *initramfs* to be extracted during boot. This can be useful if you want to create a generic *initramfs* (for example one that does mdadm) and then add modifications in separate files (for example a custom /etc/mdadm.conf for every machine). It is also convenient if you need an initramfs for some reason and also want a bootsplash. You can specify multiple *initramfs* archives in your grub config by using multiple initrd lines.

File: /boot/grub/grub.conf

```
title=Gentoo Linux 2.6.27-r8 with initramfs (hd0,0)
root (hd0,0)
kernel /boot/kernel-2.6.27-gentoo-r8
initrd /boot/first-initramfs.cpio.gz
initrd /boot/second-initramfs.cpio.gz
initrd /boot/third-initramfs.cpio.gz
```

The kernel will extract these files in the order they are specified. The files will all be extracted into the same place, this means files from later archives will overwrite former ones if they have the same filename. Also note that it is possible to have an *initramfs* archive embedded in the kernel as well as extra ones specified in the grub config. The archive in the kernel will be extracted first, followed by the ones in the grub config.

UUID/LABEL Root Mounting

If you want to be able to set root on the kernel command line with either a LABEL or the UUID you'll need to add that parsing functionality to your *initramfs*, note that this depends on Dynamic Devices:

File: /usr/src/initramfs/init

```
uuidlabel_root() {
    for cmd in $(cat /proc/cmdline) ; do
        case $cmd in
        root=*)
            type=$(echo $cmd | cut -d= -f2)
            if [ $type == "LABEL" ] || [ $type == "UUID" ] ; then
                uuid=$(echo $cmd | cut -d= -f3)
                mount -o ro $(findfs "$type"="$uuid") /mnt/root
            else
                mount -o ro $(echo $cmd | cut -d= -f2) /mnt/root
            fi
            ;;
        esac
    done
}
```

It's a good idea to invoke this with the rescue_shell. Also make sure you **don't have any other mount lines set to mount the root filesystem** in your /init:

File: /usr/src/initramfs/init

```
uuidlabel_root || rescue_shell
```

This snippet will now allow for grub.conf entries like:

File: Example grub.conf

```
title My Linux
kernel /my-kernel root=LABEL=Gentoo
```

Or, using UUIDs:

File: Example grub.conf

```
title My Linux
kernel /my-kernel root=UUID=7364896a-ed4d-41ec-a08a-8e010be61446
```

The standard /dev/sd??* arguments will still work.

Bootsplash

fbsplash works by putting the splash theme's files and some helper files common to all splash themes in an initramfs. If your kernel has the fbsplash patch and a splash theme is specified, the kernel will look for the theme's configuration file in the /etc/splash/theme_name folder of the initramfs and try to run that theme. No init file is needed for this to happen, it will instead automatically run /sbin/fbcondecor_helper on your initramfs.

See the Fbsplash article for instructions.

Troubleshooting

Static vs. Dynamic binaries

Any custom binaries you need to use in your *initramfs* before mounting have to be fully functional, independently from any files you may have installed on your root partition. This is much easier to achieve with static binaries (which usually work as single file) than with dynamic binaries (which need any number of additional libraries to work).

Gentoo provides static binaries for some ebuilds. Check if the ebuild for your binary offers a static or -dynamic USE flag. That's by far the easiest method to get a static binary for something, but unfortunately only a select few ebuilds support it.

Many applications also offer static builds with an option in their configure scripts. There is no standard name for the option, it may be --enable-static or similar. When compiling a package manually, check the list of available options with ./configure --help to see if it supports building static binaries for you.

You can check whether or not a binary is static by using the ldd command. For static binaries, it will simply say that it's not a dynamic executable. Otherwise it will show a list of libraries, for example:

```
ldd /bin/ls
        linux-gate.so.1 =>  (0xffffe000)
        librt.so.1 => /lib/librt.so.1 (0xb7ee8000)
        libacl.so.1 => /lib/libacl.so.1 (0xb7ee0000)
        ...
```

Including libraries into your *initramfs* in order to make a dynamic executable work is a last resort only. It makes the *initramfs* much larger and more complicated to maintain than necessary.

Sometimes even static binaries also have dependencies on other files or libraries that are not obvious at first glance. In those cases you can use strace to investigate in more detail why a binary is failing, when it does not give you a good error message.

Kernel panics

When working with *initramfs* and writing custom init scripts for it, you may experience the following kernel panic on boot:

```
 Kernel panic - not syncing: Attempted to kill init!
```

This is not an error in the kernel, but an error in your /init script. This script is executed as the init process with PID 1. Unlike other processes, the PID 1 init process is special. It is the only process that is started by the kernel on boot. It's the process that takes care of starting other processes (boot process, init scripts) which in turn start other processes (daemons, login prompts, X), which in turn start other processes (bash, window manager, browser, ...). The init process is the mother of all other processes, and therefore it mustn't be killed. On shutdown, it's again the init process that takes care of cleaning up by shutting down other processes first, then running processes that will unmount the filesystems, until it is safe to actually do a shutdown without corrupting anything.

If you have some error in your init script, that causes the init process to end, this basically means there are no processes left to run, there is nothing that could take care of cleaning up, and the kernel has no choice but to panic. For this reason there are some things in /init that you can't do like you can do them in a normal shell script, like using return or exit, or letting the script just run a series of commands and then simply end.

If you want /init to end, you have to pass the responsibility of the init process to someone else using *exec*. See the examples above how *exec* is used to either run /sbin/init of the mounted root partition or to run a rescue shell in case something went wrong.

Job Control

While working with *initramfs*, especially the Initramfs#Rescue_Shell, you may come across this message:

```
/bin/sh: can't access tty; job control turned off
```

The lack of Job Control is usually not a problem, since /init is not supposed to be interactive. However, if you want to work with the Busybox shell on a regular basis, being unable to control programs with Ctrl-C or Ctrl-Z can easily become a huge issue. In worst case, if Job Control is not available, and a program refuses to quit, you have to reboot.

The busybox faq [1] offers some help here. You can either use

```
setsid sh -c 'exec sh </dev/tty1 >/dev/tty1 2>&1'
```
or

```
setsid cttyhack sh
```
to start a shell on tty1 with jobcontrol enabled

Salvaging

If for whatever reason you lost your /usr/src/initramfs structure, but you still got either the kernel image with the builtin *initramfs*, or the separate cpio archive, it's possible to salvage it from there. Although it may be easier to just redo it from scratch - if you've done it once, doing it again should be a piece of cake. So this is just in case.

Warning: The following commands will overwrite files, so you should do all your work in a temporary directory like /tmp/initramfs

Dismantling the Kernel

You can skip this step if your *initramfs* is a separate cpio archive already. Otherwise, you'll have to get the built-in cpio archive out of the kernel image. To do that, you have to dismantle it, which isn't easy, since the kernel image is a combination of boot sector and compressed archive itself. It also depends on the compression you are using for your kernel and for your *initramfs*. For simplicity, this example assumes bzip2 - however, the principle is the same for other compression methods.

First, you have to search for the compression signature in the kernel image. For bzip2, this is **BZh**.

```
grep -a -b --only-matching BZh bzImage
```
For me, this prints **12888:BZh**, so the offset is 12888 bytes. Now you can extract the kernel image:

```
dd if=bzImage bs=1 skip=12888 | bunzip2 > Image
```
Now, you have the uncompressed kernel image. Somewhere within this image resides the compressed *initramfs* archive, so just iterate the previous process to find it.

```
grep -a -b --only-matching BZh Image
```
For me, this prints **171424:BZh**, so the offset is 171424 bytes this time. Now you can extract the *initramfs* cpio archive:

```
dd if=Image bs=1 skip=171424 | bunzip2 > initramfs.cpio
```
If you want to verify that you actually got a cpio archive from that, use file:

```
file initramfs.cpio
```

Extracting the cpio archive

If your *initramfs* cpio archive was a separate file, you have to uncompress it first.

```
gunzip initramfs.cpio.gz
```
To extract the uncompressed initramfs.cpio, you can do so with the following command. Please note that this overwrites files in the directory you're currently in:

```
cpio -i -d -H newc --no-absolute-filenames < initramfs.cpio
```
With this, you should have successfully recovered your *initramfs* structure.

Files in initramfs unreachable

Many guides to initramfs mention a filelist fed to cpio or /usr/src/linux/usr/gen_init_cpio instead of building it on the fly with find . -print0. If you follow one of these guides, you need to take extra care of having the correct line order in your filelist. For example, using

File: initramfs.filelist.broken
```
dir /lib/udev 755 0 0
dir /lib 755 0 0
```

will include /lib/udev, but it won't be reachable inside the initramfs. The correct order would be

File: initramfs.filelist
```
dir /lib 755 0 0
dir /lib/udev 755 0 0
```