# Introduction to NanoBSD

Abstract

This document provides information about the NanoBSD tools, which can be used to create FreeBSD system images for embedded applications, suitable for use on a USB key, memory card or other mass storage media.

---

# Table of Contents

1. Introduction to NanoBSD . . . . . 1
2. NanoBSD Howto . . . . . 1

# 1. Introduction to NanoBSD

NanoBSD is a tool developed by Poul-Henning Kamp <phk@FreeBSD.org> and now maintained by Warner Losh <imp@FreeBSD.org>. It creates a FreeBSD system image for embedded applications, suitable for use on a USB key, memory card or other mass storage media.

It can be used to build specialized install images, designed for easy installation and maintenance of systems commonly called "computer appliances". Computer appliances have their hardware and software bundled in the product, which means all applications are pre-installed. The appliance is plugged into an existing network and can begin working (almost) immediately.

The features of NanoBSD include:

- Ports and packages work as in FreeBSD - Every single application can be installed and used in a NanoBSD image, the same way as in FreeBSD.
- No missing functionality - If it is possible to do something with FreeBSD, it is possible to do the same thing with NanoBSD, unless the specific feature or features were explicitly removed from the NanoBSD image when it was created.
- Everything is read-only at run-time - It is safe to pull the power-plug. There is no necessity to run fsck(8) after a non-graceful shutdown of the system.
- Easy to build and customize - Making use of just one shell script and one configuration file it is possible to build reduced and customized images satisfying any arbitrary set of requirements.

# 2. NanoBSD Howto

## 2.1. The Design of NanoBSD

Once the image is present on the medium, it is possible to boot NanoBSD. The mass storage medium

is divided into three parts by default:

- Two image partitions: `code#1` and `code#2`.
- The configuration file partition, which can be mounted under the /cfg directory at run time.

These partitions are normally mounted read-only.

The /etc and /var directories are md(4) (malloc) disks.

The configuration file partition persists under the /cfg directory. It contains files for /etc directory and is briefly mounted read-only right after the system boot, therefore it is required to copy modified files from /etc back to the /cfg directory if changes are expected to persist after the system restarts.

*Example 1. Making Persistent Changes to /etc/resolv.conf*

```
# vi /etc/resolv.conf
[...]
# mount /cfg
# cp /etc/resolv.conf /cfg
# umount /cfg
```



The partition containing /cfg should be mounted only at boot time and while overriding the configuration files.

Keeping /cfg mounted at all times is not a good idea, especially if the NanoBSD system runs off a mass storage medium that may be adversely affected by a large number of writes to the partition (like when the filesystem syncer flushes data to the system disks).

## 2.2. Building a NanoBSD Image

The source code of FreeBSD is required to build NanoBSD. To obtain the source code:

```
# git clone https://git.FreeBSD.org/src.git /usr/src
```

For more details, follow the steps here.

A NanoBSD image is built using a simple nanobsd.sh shell script, which can be found in the /usr/src/tools/tools/nanobsd directory. This script creates an image, which can be copied on the storage medium using the dd(1) utility.

The necessary commands to build a NanoBSD image are:

```
# cd /usr/src/tools/tools/nanobsd ①
# sh nanobsd.sh ②
```

```
# cd /usr/obj/nanobsd.full ③
# dd if=_.disk.full of=/dev/da0 bs=64k ④
```

① Change the current directory to the base directory of the NanoBSD build script.

② Start the build process.

③ Change the current directory to the place where the built images are located.

④ Install NanoBSD onto the storage medium.

### 2.2.1. Options When Building a NanoBSD Image

When building a NanoBSD image, several build options can be passed to nanobsd.sh on the command line. These options can have a significant impact on the build process.

Some options are for verbosity purposes:

- `-h`: prints the help summary page.
- `-q`: makes output quieter.
- `-v`: makes output more verbose.

Some other options can be used to restrict the building process. Sometimes it is not necessary to rebuild everything from sources, especially if an image has already been built, and only little change is made.

- `-k`: do not build the kernel.
- `-w`: do not build world.
- `-b`: do not build either kernel and world.
- `-i`: do not build a disk image at all. As a file will not be created, it will not be possible to dd(1) it to a storage media.
- `-f`: do not build a disk image of the first partition (which is useful for upgrade purposes).
- `-p`: do not prepare the image. Skip running customization and early-customization scripts for incremental image refinement from world, kernel, or packages.
- `-n`: add `-DNO_CLEAN` to `buildworld`, `buildkernel`. Also, all the files that have already been built in a previous run are kept.

A configuration file can be used to tweak as many elements as desired. Load it with `-c`

The last options are:

- `-K`: do not install a kernel. A disk image without a kernel will not be able to achieve a normal boot sequence.
- `-W`: do not install world.
- `-B`: do not install either kernel and world.
- `-I`: build a disk image from an existing build or install. Do not build or install kernel, world, and the etc configuration file, just create the disk image.

### 2.2.2. The Complete Image Building Process

The complete image building process is going through a lot of steps. The exact steps taken will depend on the chosen options when starting the script. Assuming the script is run with no particular options, this is what will happen.

1. `run_early_customize`: Commands that are defined in a supplied configuration file.
2. `clean_build`: Just cleans the build environment by deleting the previously built files.
3. `make_conf_build`: Assemble make.conf from the `CONF_WORLD` and `CONF_BUILD` variables.
4. `build_world`: Build world.
5. `build_kernel`: Build the kernel files.
6. `clean_world`: Clean the destination directory.
7. `make_conf_install`: Assemble make.conf from the `CONF_WORLD` and `CONF_INSTALL` variables.
8. `install_world`: Install all files built during `buildworld`.
9. `install_etc`: Install the necessary files in the /etc directory, based on the `make distribution` command.
10. `setup_nanobsd_etc`: The first configuration specific to NanoBSD takes place at this stage. The /etc/diskless is created and the root filesystem is defined as read-only.
11. `install_kernel`: The kernel and modules files are installed.
12. `run_customize`: All the customizing routines defined by the user will be called.
13. `setup_nanobsd`: A special configuration directory layout is setup. The /usr/local/etc gets moved to /etc/local and a symbolic link is created back from /etc/local to /usr/local/etc.
14. `prune_usr`: The empty directories from /usr are removed.
15. `run_late_customize`: The very last custom scripts can be run at this point.
16. `fixup_before_diskimage`: List all installed files in a metalog.
17. `create_diskimage`: Creates the actual disk image, based on the disk geometries provides parameters.
18. `last_orders`: Does nothing for now.

## 2.3. Customizing a NanoBSD Image

This is probably the most important and most interesting feature of NanoBSD. This is also where you will be spending most of the time when developing with NanoBSD.

Invocation of the following command will force the nanobsd.sh to read its configuration from myconf.nano located in the current directory:

```
# sh nanobsd.sh -c myconf.nano
```

Customization is done in two ways:

- Configuration options
- Custom functions

### 2.3.1. Configuration Options

With configuration settings, it is possible to configure options passed to both the `buildworld` and `installworld` stages of the NanoBSD build process, as well as internal options passed to the main build process of NanoBSD. Through these options it is possible to cut the system down, so it will fit on as little as 64MB. You can use the configuration options to trim down FreeBSD even more, until it will consists of just the kernel and two or three files in the userland.

The configuration file consists of configuration options, which override the default values. The most important directives are:

- `NANO_NAME` - Name of build (used to construct the workdir names).
- `NANO_SRC` - Path to the source tree used to build the image.
- `NANO_KERNEL` - Name of kernel configuration file used to build kernel.
- `CONF_BUILD` - Options passed to the `buildworld` stage of the build.
- `CONF_INSTALL` - Options passed to the `installworld` stage of the build.
- `CONF_WORLD` - Options passed to both the `buildworld` and the `installworld` stage of the build.
- `FlashDevice` - Defines what type of media to use. Check FlashDevice.sub for more details.

There are many more configuration options that could be relevant depending upon the kind of NanoBSD that is desired.

#### General Customization

There are three stages, by design, at which it is possible to make changes that affect the building process, just by setting up a variable in the provided configuration file:

- `run_early_customize`: before anything else happens.
- `run_customize`: after all the standard files have been laid out.
- `run_late_customize`: at the very end of the process, just before the actual NanoBSD image is built.

To customize a NanoBSD image, at any of these steps, it is best to add a specific value to one of the corresponding variables.

The `NANO_EARLY_CUSTOMIZE` variable is used at the first step of the building process. At this point, there is no example as to what can be done using that variable, but it may change in the future.

The `NANO_CUSTOMIZE` variable is used after the kernel, world and etc configuration files have been installed, and the etc files have been setup as to be a NanoBSD installation. So it is the correct step in the building process to tweak configuration options and add packages, like in the cust_nobeastie example.

The `NANO_LATE_CUSTOMIZE` variable is used just before the disk image is created, so it is the very last moment to change anything. Remember that the `setup_nanobsd` routine already executed and that

the etc, conf and cfg directories and subdirectories are already modified, so it is not time to change them at this point. Rather, it is possible to add or remove specific files.

### Booting Options

There are also variables that can change the way the NanoBSD image boots. Two options are passed to boot0cfg(8) to initialize the boot sector of the disk image:

- `NANO_BOOT0CFG`
- `NANO_BOOTLOADER`

With `NANO_BOOTLOADER` a bootloader file can be chosen. The most common possible options are between boot0sio and boot0 depending on whether the appliance has a serial port or not. It is best to avoid supplying a different bootloader, but it is possible. To do so, it is best to have checked the FreeBSD Handbook chapter on the boot process.

With `NANO_BOOT0CFG`, the booting process can be tweaked, like selecting on which partition the NanoBSD image will actually boot. It is best to check the boot0cfg(8) page before changing the default value of this variable. One option that could be interesting to change is the timeout of the booting procedure. To do so, the `NANO_BOOT0CFG` variable can be changed to `"-o packet -s 1 -m 3 -t 36"`. That way the booting process would start after approximately 2 seconds; because it is rare that waiting 10 seconds before actually booting is desired.

Good to know: the `NANO_BOOT2CFG` variable is only used in the `cust_comconsole` routine that can be called at the `NANO_CUSTOMIZE` step if the appliance has a serial port and all console input and output has to take place through it. Be sure to check the relevant parameters of the serial port, as setting a bad parameter value can make it useless.

### Disk Image Creation

In the end of the boot process is the disk image creation. With this step, the NanoBSD script provides a file that can simply be copied onto a disk for the appliance, and that will make it boot and start.

There are many variables that need to be set just right for the script to produce a usable disk image.

- The `NANO_DRIVE` variable must be set to the drive name of the media at runtime. Usually, the default value `ada0`, which represents the first `IDE`/`ATA`/`SATA` device on the appliance is expected to be the correct one, but a different type of storage could also be used - like a USB key, in which case, it would rather be `da0`.
- The `NANO_MEDIASIZE` variable must be set to the size (in 512-byte sectors) of the storage media that will be used. If you set it wrong, it is possible that the NanoBSD image will not boot at all, and a message at boot time will be warning about incorrect disk geometry.
- The /etc, /var, and /tmp directories are allocated as md(4) (malloc) disks at boot time; so their sizes can be tailored to suit the appliance needs. The `NANO_RAM_ETCSIZE` variable sets the size of the /etc; and the `NANO_RAM_TMPVARSIZE` variable sets the size of both the /var and /tmp directory, as /tmp is symbolically linked to /var/tmp. By default, both malloc disks sizes are set at 20MB each. They can always be changed, but usually the /etc does not grow too much in size, so 20MB is a good starting point, whereas the /var and especially /tmp can grow much larger if not careful

about it. For memory constrained systems, smaller filesystem sizes may be chosen.

- As NanoBSD is mainly designed to build a system image for an appliance, it is assumed that the storage media used will be relatively small. For that reason, the filesystem that is laid out is configured to have a small block size (4Kb) and a small fragment size (512b). The configuration options of the filesystem can be modified through the `NANO_NEWFS` variable, but the syntax must respect the newfs(8) command format. Also, by default, the filesystem has Soft Updates enabled. The FreeBSD Handbook can be checked about this.
- The different partition sizes can be set through the use of `NANO_CODESIZE`, `NANO_CONFSIZE`, and `NANO_DATASIZE` as a multiple of 512-byte sectors. `NANO_CODESIZE` defines the size of the first two image partitions: `code#1` and `code#2`. They have to be big enough to hold all the files that will be produced as a result of the `buildworld` and `buildkernel` processes. `NANO_CONFSIZE` defines the size of the configuration file partition, so it does not need to be very big; but do not make it so small that it will not hold all configuration files. Finally, `NANO_DATASIZE` defines the size of an optional partition, that can be used on the appliance. The last partition can be used, for example, to keep files created on the fly on disk.

### 2.3.2. Custom Functions

It is possible to fine-tune NanoBSD using shell functions in the configuration file. The following example illustrates the basic model of custom functions:

```
cust_foo () (
    echo "bar=baz" > \
        ${NANO_WORLDDIR}/etc/foo
)
customize_cmd cust_foo
```

A more useful example of a customization function is the following, which changes the default size of the /etc directory from 5MB to 30MB:

```
cust_etc_size () (
    cd ${NANO_WORLDDIR}/conf
    echo 30000 > default/etc/md_size
)
customize_cmd cust_etc_size
```

There are a few default pre-defined customization functions ready for use:

- `cust_comconsole` - Disables getty(8) on the VGA devices (the /dev/ttyv* device nodes) and enables the use of the COM1 serial port as the system console.
- `cust_allow_ssh_root` - Allow `root` to login via sshd(8).
- `cust_install_files` - Installs files from the nanobsd/Files directory, which contains some useful scripts for system administration.
- `cust_pkgng` - Installs packages from the nanobsd/Pkg directory (needs also pkg-* package to bootstrap).

### 2.3.3. Adding Packages

Packages can be added to a NanoBSD image, to provide specific functionalities on the appliance. To do so, either:

- Add the `cust_pkgng` to the `NANO_CUSTOMIZE` variable, or
- Add a `'customize_cmd cust_pkgng'` command in a customized configuration file.

Both methods achieve the same result: launching the `cust_pkgng` routine. This routine will go through `NANO_PACKAGE_DIR` directory to find either all packages or just the list of packages in the `NANO_PACKAGE_LIST` variable.

It is common, when installing applications through pkg on a standard FreeBSD environment, that the install process puts configuration files, in the usr/local/etc directory, and startup scripts in the /usr/local/etc/rc.d directory. So, after the required packages have been installed, they need to be configured in order for them to start right out of the box. To do so, the necessary configuration files have to be installed in the correct directories. This can be achieved by writing dedicated routines or the generic `cust_install_files` routine can be used to lay out files properly from the /usr/src/tools/tools/nanobsd/Files directory. Usually a statement, sometimes multiple statements, in the /etc/rc.conf also needs to be added for each package.

### 2.3.4. Configuration File Example

A complete example of a configuration file for building a custom NanoBSD image can be:

```
NANO_NAME=custom
NANO_SRC=/usr/src
NANO_KERNEL=MYKERNEL
NANO_IMAGES=2

CONF_BUILD='
WITHOUT_KLDLOAD=YES
WITHOUT_NETGRAPH=YES
WITHOUT_PAM=YES
'

CONF_INSTALL='
WITHOUT_ACPI=YES
WITHOUT_BLUETOOTH=YES
WITHOUT_FORTRAN=YES
WITHOUT_HTML=YES
WITHOUT_LPR=YES
WITHOUT_MAN=YES
WITHOUT_SENDMAIL=YES
WITHOUT_SHAREDOCS=YES
WITHOUT_EXAMPLES=YES
WITHOUT_INSTALLLIB=YES
WITHOUT_CALENDAR=YES
WITHOUT_MISC=YES
```

```
WITHOUT_SHARE=YES
'

CONF_WORLD='
WITHOUT_BIND=YES
WITHOUT_MODULES=YES
WITHOUT_KERBEROS=YES
WITHOUT_GAMES=YES
WITHOUT_RESCUE=YES
WITHOUT_LOCALES=YES
WITHOUT_SYSCONS=YES
WITHOUT_INFO=YES
'

FlashDevice SanDisk 1G

cust_nobeastie() (
    touch ${NANO_WORLDDIR}/boot/loader.conf
    echo "beastie_disable=\"YES\"" >> ${NANO_WORLDDIR}/boot/loader.conf
)

customize_cmd cust_comconsole
customize_cmd cust_install_files
customize_cmd cust_allow_ssh_root
customize_cmd cust_nobeastie
```

All the build and install compilation options can be found in the src.conf(5) man page, but not all options can or should be used when building a NanoBSD image. The build and install options should be defined according to the needs of the image being built.

For example, the ftp client and server might not be needed. Adding `WITHOUT_FTP=TRUE` to a configuration file in the `CONF_BUILD` section will avoid having them built. Also, if the NanoBSD appliance will not be used to build programs then it is possible to add the `WITHOUT_BINUTILS=TRUE` in the `CONF_INSTALL` section; but not in the `CONF_BUILD` section as they will be used to build the NanoBSD image.

Not building a particular set of programs - through a compilation option - shortens the overall building time and lowers the required size for the disk image, whereas not installing the same specific set of programs does not lower the overall building time.

## 2.4. Updating NanoBSD

The update process of NanoBSD is relatively simple:

1. Build a new NanoBSD image, as usual.
2. Upload the new image into an unused partition of a running NanoBSD appliance.

   The most important difference of this step from the initial NanoBSD installation is that now

instead of using _.disk.full (which contains an image of the entire disk), the _.disk.image image is installed (which contains an image of a single system partition).

3. Reboot, and start the system from the newly installed partition.
4. If all goes well, the upgrade is finished.
5. If anything goes wrong, reboot back into the previous partition (which contains the old, working image), to restore system functionality as fast as possible. Fix any problems of the new build, and repeat the process.

To install new image onto the running NanoBSD system, it is possible to use either the updatep1 or updatep2 script located in the /root directory, depending from which partition is running the current system.

According to which services are available on the host serving the new NanoBSD image and what type of transfer is preferred, it is possible to use one of these three methods:

### 2.4.1. Using ftp(1)

If the transfer speed is most important, use this example:

```
# ftp myhost
get _.disk.image "| sh updatep1"
```

### 2.4.2. Using ssh(1)

If a secure transfer is preferred, consider using this example:

```
# ssh myhost cat _.disk.image.gz | zcat | sh updatep1
```

### 2.4.3. Using nc(1)

Try this example if the remote host is not running the ftpd(8) or sshd(8) service:

1. First, open a TCP listener on the host serving the image, and make it send the image to the client:

   ```
   myhost# nc -l 2222 < _.disk.image
   ```

   

   Make sure that the chosen port is not blocked by the firewall to receive incoming connections from the NanoBSD host.

2. Connect to the host serving the new image, and execute the updatep1 script:

```
# nc myhost 2222 | sh updatep1
```