Making Buildfiles for the QNX Neutrino RTOS, Part 2

by Akhilesh Mritunjai

At this point you should have read the article on Making Buildfiles for the QNX Neutrino RTOS. If you've got a buildfile and prepared an image, you're ready to move on.

Using buildfiles

Q. So I have a buildfile. Now what?

A. When you have a buildfile, you just need to make the image file system using the **mkifs** utility.

```
mkifs -v my.build my.ifs
```

This takes the buildfile *my.build*, creates an image file *my.ifs*, and spits out the details due to the **-v** option.

Q. OK! I've got the so-called *my.ifs*. Now what?

A. Well, now it's time to get to work. In order to boot using your image, you should understand how things work to get them to work for you! Here's a brief overview of the boot process. When you switch on a computer it first performs a Power On Self Test (POST) if equipped with a BIOS (in the case of a PC and most other computers). Then it initializes the system components present. If it detects some special components, it calls their extended BIOS routines to initialize them as well. Finally, when everything is done, it looks to see if there is someone to take control. It checks the boot sequence and decides where to look for the boot loader, and in what order. Let's assume it checks the hard disk first. A hard disk is divided into several partitions, each acting as an independent disk. The Track0-Sector0-Head0 of a bootable hard disk (and perhaps a flash disk too) is supposed to contain a Master Boot Record (MBR). On seeing the MBR signature on this sector, the BIOS loads MBR into memory and hands over control to it. MBR has a very small generic program called Master Boot Loader that actually doesn't know how to boot an OS. In the same sector, there is a partition table which contains information about all partitions on the disk. The bootable partition (one which contains an OS) is marked "active" in this partition table. The loader then loads the contents of sector0 of the active partition in memory. This sector contains the "Boot Record."

Note: This type of Master Boot Loader doesn't know anything about multi-booting. The one shipped with the QNX Neutrino RTOS is more flexible and allows you to select the disk/partition to load the OS from at boot time.

In case the floppy of the MBR is absent, and the Boot Record itself, is located at Track0-Sector0- Head0 of the disk. The boot record is an intelligent program specific to the OS on the partition and knows how to boot that OS.

If you had chosen to install the QNX Master Boot Loader, or if you have any other multi-boot master boot loader (like XOSL or GRUB), then when BIOS hands over control to it, it asks for the partition to boot from. Given the choice of a partition to boot, it then loads the boot record of that partition and transfers control to it. If it is a QNX partition, the QNX boot record takes control and presents the familiar message "Press **ESC** for *.altboot*". Then it proceeds to load the contents of "*.boot*" or "*.altboot*" depending upon the user's choice. The "*.boot*" (or "*.altboot*") is actually the QNX Neutrino RTOS image. It consults the header of ".boot" or "*.altboot*" and copies the image into memory at the specified location, then it fills in some critical data structures with the information it has, and hands over control to the startup routine. Note that, this startup code is the "**startup-***" of "*.bootstrap*" of the buildfile.

Note: Strictly speaking, on an x86 with BIOS, "**startup-bios**" doesn't directly get control. There is a small program sitting in front of "**startup-bios**" called "**bios.boot**" which gets the control, pushes the processor in protected mode, copies the image to the proper location, and finally hands over control to startup-bios.

Now, "**startup-***" are more intelligent programs. They knows how to probe for hardware present, get the CPU/machine-specific parameters, uncompress the image (depending upon the architecture), and start the kernel. At this point the system is ready and the kernel proceeds to run the startup script of the buildfile.

If you're still fuzzy about the booting process, I would recommend you go over all this again. The bottom line is that somehow, by hook or by crook you have to get your QNX Neutrino RTOS image into ".boot" (or "*.altboot*") so that boot loader can load it. This exercise is relatively easy and only requires knowledge of an easy program called "**dinit**". A typical use is as follows:

```
dinit -f my.ifs /dev/fd0
```

In this case, **dinit** formats the floppy disk, puts boot record, and fills "*.boot*" with *my.ifs*. Or, if you have a disk formatted as **qnx4fs** (e.g. partition install), just mount it (if it isn't already mounted) and replace */.altboot* (or /.boot if you know it works and want it to be your primary boot image) with *my.ifs* by simply using:

```
cp -f my.ifs /.altboot
```

But be warned! You should be dead sure that either */.boot* or */.altboot* is sane and lets you boot into your system, or you'll regret it later. The recovery procedure in this case is not difficult but its sure long and tedious.

Q. After booting, at some point the system will say it "can't start something." What's going on? A. This is a common and simple to solve problem. Be sure that you've included that "something" application in image. Also, see the "perms" attribute and make sure the application is mode "**`+r,+x`**" in the image and "**PATH**" and "**LD_LIBRARY_PATH**", and contain proper path to the application and libraries respectively. Check for all typos! The output of "**`mkifs -v`**" tells the address of every library and application included. If there's some problem, instead of address you'll see "**--**", which is normally shown only for symbolic links and other stuff that doesn't occupy any space. This should give some clue as to where you went wrong. Watch out for "file not found" warnings. "**mkifs**" doesn't consider it as an error and proceeds without that file only to make your life miserable later. And lastly, check if you've included all shared libraries required by the application.

Note on shared libraries:

When making a system image you need to ensure that you aren't mixing stuff from the QNX Neutrino RTOS version 6.0 with version 6.1. The reason is, programs compiled under v6.0 require v6.0 libraries and the ones compiled in v6.1 require 6.1 libraries. The new libraries are distinguished from the older ones by a "*.2*" suffix, e.g. *libc.so.2*, *libsocket.so.2*, *libm.so.2*, and so on. The corresponding libraries in v6.0 end in a ".1" suffix, e.g. libc.so.1, *libsocket.so.1*, *libm.so.1*, *libstdc++.so.2.10.0* (note the names of old and new C++ libraries!). If for some unavoidable reason you're mixing applications from both releases, be sure to add libraries (and the symbolic links) for both versions in the buildfile.

Here is how you can determine libraries required by a particular application. This would enable you to find out whether the application was compiled for the QNX Neutrino RTOS v6.0 or v6.1. Just use:

```
objdump -x myapplication | grep "NEEDED"
```

Now you're ready to take off!

Buildfile #1

Aim

To make a system image from which you can explore the environment after booting from the image.

Target

```
x86/bios
```

Approach

To explore the environment, you need a shell and some commands to look around, e.g."**ls**" and "**less**." "**Ksh**" should do a good job as a shell in this case since it is feature-rich. It's bulky but you have lots of space.

Buildfile

```
[virtual=x86,bios +compress] .bootstrap={
startup-bios -s 64k
PATH=/proc/boot LD_LIBRARY_PATH=/proc/boot:/usr/lib procnto
}
[+script] .script={
seedres
display_msg "My QNX Explorer Image..."
display_msg "Hello World"
# two virtual consoles
# at Ctrl+Alt+1 and Ctrl+alt+2
devc-con -n2 &
reopen /dev/con2
[+session] ksh &
reopen /dev/con1
[+session] ksh &
}
libc.so
# here's another way of creating a symbolic link
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
[code=uip data=copy perms=+r,+x]
seedres
kill
cat
ls
ksh
devc-con
less
ps
sin
pidin

#Here goes our favorite editor...
vi
```

Buildfile #2

Aim

To make a network-enabled system image that you can telnet and ftp to other machines.

Target

```
x86,bios
```

Approach

To make this type of image you'll need an NIC driver with a TCP/IP stack. Telnet/ftp can easily work with the tiny stack. I'll assume that you have an NE2000-compatible PCI NIC. If the PC doesn't have a PCI bus and card is ISA, you have to exclude "**pci-bios**." Select the appropriate driver for your card. If you have **ssh** for the QNX Neutrino RTOS v6.1, you can place it here as well, but take care of including **openssh** and other required libraries!

Note: Refer to QNX Neutrino RTOS notes for details on using version 6.0 applications in version 6.1.

Buildfile

```
[virtual=x86,bios +compress] .bootstrap={
startup-bios -s 64k
PATH=/proc/boot LD_LIBRARY_PATH=/proc/boot:/usr/lib procnto
}
[+script] .script={
display_msg "My QNX Network image"
seedres
pipe &
pci-bios &
waitfor /dev/pci
#put in your actual IP,netmask and gateway
io-net -dne2000 -pttcpip \
if=en0:YOURIP:YOURNETMASK default=GATEWAY
#If you're using ssh, start random daemon
random -t -p &
devc-con -n2 &
reopen /dev/con2
[+session] HOME=/ TERM=qansi-m ksh &
reopen /dev/con1
[+session] HOME=/ TERM=qansi-m ksh &
}
# the following is an example of inline file
# you are doing this because non-QNX machines
# won't recognize our "qansi-m" terminal. "ansi" terminal
# is found on most machines
[perms=+r,+x] /.kshrc = {
alias telnet="TERM=ansi /proc/boot/telnet"
}
libc.so
```

```
libsocket.so
# required for ssh
libz.so
npm-ttcpip.so
devn-ne200.so
/etc/hosts=/etc/hosts
/etc/termcap=/etc/termcap
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
[code=uip data=copy perms=+r,+x]
seedres
pipe
ksh
devc-con
io-net
/usr/bin/ftp
/usr/bin/telnet
/usr/bin/ssh
/usr/bin/ping
```

Buildfile #3

Aim

To make an almost full-blown networked system image that should boot off a floppy disk and should remote mount its file system through NFS/CIFS, thus acting like a console-based desktop system.

Target

```
x86,bios
```

Approach

To make this type of image you'll need a setup similar to the previous one. In addition, you'll have to prepare a server. If you can get a QNX machine as a server it'll be easy as only / will have to be exported read-only through SAMBA or NFS. Or you can copy the entire /lib, */usr*, */sbin* and **/bin** to any other *nix or Windows machine and export the base directory (say */home/me/qnx*) via NFS or CIFS (Samba/Windows sharing). On the client side, you'll need either "**fs-nfs2**" or "**fs-cifs**."

Buildfile

```
[search=/bin:/usr/bin:/lib:/usr/lib:/sbin:/lib/dll]
[virtual=x86,bios +compress] .bootstrap={
startup-bios -s 64k
PATH=/proc/boot LD_LIBRARY_PATH=/proc/boot:/usr/lib procnto
}
[+script] .script={
HOME=/
TERM=qansi-m
display_msg "Full Network image"
seedres
pipe &
pci-bios &
waitfor /dev/pci
io-net -dne2000 -p tcpip
ifconfig en0 192.168.4.100 netmask 255.255.255.0
route add default 192.168.4.1
fs-nfs2 192.168.4.19:/home/mine/qnx /
# or
# fs-cifs //WinPC:192.168.4.19:/qnxdir / "user" "pass"
devc-con -n2 &
reopen /dev/con2
[+session] uesh &
reopen /dev/con1
[+session] uesh &
}
libc.so
libsocket.so
npm-tcpip.so
devn-ne200.so
/etc/hosts=/etc/hosts
/etc/termcap=/etc/termcap
/etc/passwd=/etc/passwd
/etc/group=/etc/group
[type=link] /usr/lib/ldqnx.so.2=/proc/boot/libc.so
[code=uip data=copy perms=+r,+x]
seedres
pipe
ksh
devc-con
io-net
ifconfig
route
fs-nfs2
#fs-cifs
```

Buildfile #4

Aim

I don't want to wait 30 seconds to boot my PC. I want to boot fast!! And running "**chkfsys**" after the system boot, messes with open files. To ensure the sanity of the system, I want to run "**chkfsys**" just after mounting the file systems.

Approach

Designing such an image is tricky. First, we have to know how the QNX Neutrino RTOS boots. Only then we'll be able to accomplish this task. We begin by looking in /boot/build where the buildfiles of default /.boot are kept. We found that they only contain file system drivers and a utility "**diskboot**". Nothing much is revealed by finding out about "**diskboot**". So, we'll have to change our course of action. We know that all work "**diskboot**" does (we still don't know what!), we will have to do ourselves. Also we know that we have to somehow mount the hard disk and mount it on "/". Thus, we make a buildfile that runs "**devb-eide**" and mounts it on /.

```
devb-eide cam quite blk automount=hd0t79:/:qnx4
```

So, what we find is that after mounting is that most of the usual directories are missing! What went wrong? Wait, there's something called "package file system" in the QNX Neutrino RTOS, right? So, we go ahead and type "**fs-pkg**". Oops, it said "No such command or file". What now? Back in the system we can find "**fs-pkg**" residing in */sbin*, but this directory wasn't there when we mounted the hard disk. This means only one thing, that even /sbin was created by "**fs-pkg**". But then if **fs-pkg** is itself in */sbin*, then where does the system itself run it from? The answer lies in a simple inspection.

Typing "mount" shows all mounted file systems. On my partition install, it shows hd0t79 mounted on /, and whoops what's that..."/boot/fs/qnxbase.qfs on /pkgs/base type qnx". Who mounted that? A "`find /pkgs/base -name fs-pkg`" reveals that **fs-pkg** actually resides in "*/pkgs/base/qnx/os/core-2.1.2/x86/sbin/fs-pkg*". We had already noticed that /boot was available after mounting the hard disk. So we add mount to the buildfile and also add command to mount "*/boot/fs/qnxbase.qfs*" on /pkgs/base, and also to run "**fs-pkg**". Voila! After booting, all directories are there. Congratulations!!

The only work is to get the script run that system runs after booting and configuring the system. This script probes for hardware devices like graphics cards, audio devices, and network devices, and starts appropriate drivers and, finally, the QNX Photon microGUI. It's an obvious guess that this script should be in /etc, so we start hunting there. Wow! We find something that looks like it does a lot of things... */etc/system/sysinit* has the all the commands that start the **enum-*** family of programs that probe for hardware. So we put in a final command in buildfile to execute this file. Now we're in business! But wait! After doing all this, our buildfile is still no faster. Now what? Taking a look at the parameters of **devb-eide** and the time it takes to execute, we find that if we provide the parameters of our EIDE controller, it might be able boot faster. Parameters to note include **nopci**, slave, **ioport** ,and **irq**. Since in this case we don't have a slave hard disk, we should tell it not to probe for one, using the slave flag (Don't use this option if you have a slave disk attached). Also, since we don't want it to scan PCI devices, **nopci** flag is a sensible choice. Now, for **irq** and **ioport** values we use the "**pci -v**" command to see the details of the "mass storage (IDE) device." There we find the "PCI IO address" and, if assigned, the IRQ values also. We'll also need to mount the partitions using the **automount** option.

Tip: If you have plenty of RAM (>= 128M), increasing the cache size of "**devb-eide**" is an attractive option because it improves file system performance. Don't increase it too much because it may mean losing more data in case system fails while the data is still in buffer. Also, running "**chkfsys**" just after mounting the file system is a good idea since it increases reliability (though at cost of increase in boot time by few seconds). Play with various options until you get a combination that suits you. Good candidates are "**alloc**," "**cache**," "**delwri**," "**wipe**," and "**thread**". See "helpviewer" for their descriptions.

The buildfile optimized for my system (a 533MHz Intel® Celeron®, 256M) is as follows:

```
[search=/sbin:/usr/sbin:/bin:/usr/bin:/lib:/lib/dll:/boot/sys:.]
[virtual=x86,bios +compress] boot = {
startup-bios -s 64k
# A \ indicated continuation of line! Type it as ONE single line!!
PATH=/proc/boot:/bin:/usr/bin:/usr/photon/bin \
LD_LIBRARY_PATH=/proc/boot:/usr/lib:/lib \
procnto
}
```

```
[+script] startup-script = {
procmgr_symlink ../../proc/boot/libc.so.2 /usr/lib/ldqnx.so.2
display_msg ""
display_msg "Welcome to QNX Neutrino 6.1"
display_msg "Neutrino image built by Akhilesh Mritunjai"
seedres
pci-bios -v &
slogger
waitfor /dev/pci
display_msg "Starting devb-eide..."
#following is one long line,
# \ denotes continuation as usual
devb-eide blk \
alloc=40m,cache=60m,wipe=25%,delwri=10,\
thread=24:8:12,auto=partition,
automount=hd0t79:/:qnx4,\
automount=hd0t12:/fs/hd0-dos:dos,\
automount=cd0:/fs/cd0:cd dos exec=all,fsi=update \
cam quite eide \
nopci,ioport=f000
waitfor /pkgs 30
display_msg ""
display_msg "Done."
display_msg "/dev/hd0t79 mounted on /"
# This will sure delay booting up, leave it out if you wish
# but data safety is more important to me
display_msg "Checking / for inconsistencies..."
chkfsys -qPr /
display_msg "Done."
display_msg "Mounting base packages"
mount -t qnx4 /boot/fs/qnxbase.qfs /pkgs/base
display_msg "Starting fs-pkg"
/pkgs/base/qnx/os/core-2.1.2/x86/sbin/fs-pkg
#start console driver
/pkgs/base/qnx/os/drivers-2.1.2/x86/sbin/devc-con -n4
display_msg "Starting sysinit..."
sh -c /etc/system/sysinit
}
libc.so
libcam.so
```

```
io-blk.so
cam-disk.so
cam-cdrom.so
fs-qnx4.so
fs-dos.so
fs-cd.so
[code=uip data=copy perms=+r,+x]
seedres
pci-bios
devb-eide
slogger
devc-con
sh
mount
chkfsys
# optionally you can provide a list of things to be unlinked
# from image after booting. This saves RAM. But be sure
# that they aren't used later in the image script!
unlink_list = {
/proc/boot/devb-*
/proc/boot/chkfsys
/proc/boot/mount
/proc/boot/seedres
/proc/boot/pci-bios
/proc/boot/cam-*
/proc/boot/fs-*
/proc/boot/libcam.so
/proc/boot/io-blk.so
/proc/boot/sh
}
```

Enhanced waitfor

When all is going well, why not add a pinch of spice? Write our own **waitfor** that doesn't just sit there, but tells us the time elapsed while waiting. Just compile, put it in the buildfile and replace all "**waitfor**" with "**mywaitfor**." Here is the source code for "**mywaitfor**" in plain C.

```
#include
#include
#include
int main(int argc, char **argv)
{
int secs = -1, nsec = 0;
```

```
if(argc < 2)
{
printf("Use: %s [time_in_seconds]\n", argv[0]);
return -1;
}
if(argc == 3)
{
secs = atoi(argv[2]);
}
while(secs == -1 || secs-- > 0)
{
if(access(argv[1], F_OK) == 0)
{
break;
}
if(secs >= 0)
{
printf("\r%d", nsec++);
fflush(stdout);
}
else
{
printf("\r");
}
delay(200);
printf(" . ");
fflush(stdout);
delay(200);
printf(" . ");
fflush(stdout);
delay(200);
printf(" . ");
fflush(stdout);
delay(200);
printf(" . ");
fflush(stdout);
delay(200);
printf("\r ");
fflush(stdout);
if(secs < 0)
{
secs = -1;
}
}
if(access(argv[1], F_OK) == -1)
{
/* following is one single line
\ denotes continuation */
printf("\rTIMEOUT: Could not access \'%s\' in \
%s seconds.\n", argv[1], argv[2]);
fflush(stdout);
return 1;
}
printf("\r\n");
return 0;
}
```

Epilogue

Hope you enjoyed the journey. Have fun with QNX, and if you do make some interesting stuff from it, do let me know.

Keep Smiling,

- Mritunjai