Making Buildfiles for the QNX Neutrino RTOS

by Akhilesh Mritunjai

Staring at the boot screen, how many times have you wished for the thing to boot faster? How many times have you murmured "Abracadabra I wish this junkie 486 would turn into a slick MP3 player"? All your prayers have been answered now. QNX Neutrino RTOS gives you the powers to do all these things yourself! And the key lies in designing an appropriate image and making the corresponding buildfile for it. So let's dive in and explore... Splash!!!

[**Note**: This article refers to embedding tools which may not be available on publicly available versions of QNX Neutrino. In particular, the **mkxfs**, **mkifs** and **mkefs** are only available through QNX Momentics.]

Understanding Buildfiles

In short, Buildfiles are a set of rules according to which system image is generated. A buildfile has the following parts:

- bootstrap script
- startup script
- file list
- unlink list (optional)

Let's analyze these parts in detail.

Bootstrap script

Different types of CPUs boot in different ways, thus the startup program is different for each CPU. The bootstrap script specifies the startup program to use, including any special flags needed for correct operation, e.g. disabling HLT instructions for some early buggy Intel® Pentium® processors, reserving specific regions of address space for particular drivers, etc. Let's take a look at a typical case:

```
[virtual=x86,bios +compress] .bootstrap = {
startup-bios -s 64k -D 8250.3f8.9600 -A -vv
PATH=/proc/boot LD_LIBRARY_PATH=/proc/boot:/usr/lib:/lib procnto
}
```

In this example, the first line specifies that it is a bootstrap file. The components of this bootstrap script actually boot the system and bring it to stage where it can do useful work. The file's attributes precede the file name. Here "virtual" keyword specifies that this buildfile builds a boot image corresponding to a virtual address space to be resolved at boot time. The "**x86,bios**" keywords specify the processor (x86) and machine type (booting via BIOS). The "**+compress**" keyword specifies that the image is to be compressed. This often results in a much smaller image size. Note that the bootstrap file components, themselves, are never compressed.

The QNX Neutrino RTOS v6.1 supports the booting of multi-megabyte images. In version 6.0 image size was limited to 632K for an x86 booting off a disk.

Analyzing the contents of the bootstrap file you'll find that it closely resembles a shell script. The bootstrap file starts two programs:

startup-bios

Startup-bios is responsible for uncompressing the image (**bios.boot** does that in an image for an x86 equipped with a BIOS, it is included automatically when you specify target as "**x86,bios**"), placing it at a proper address in RAM, doing configurations such as detecting CPUs present, and finally, running the kernel. In this example, the "**-s 64k**" option tells it to copy the first 64K of video BIOS from ROM to RAM for faster performance (since ROM is much slower than RAM) and also to allow the support of multiple video cards. The second option tells it to open a debugging channel on the first serial port at 9600 baud. Thus, the debug output can be captured on another PC connected via a null modem serial cable. This is very handy if you can't get your system to boot. The third option tells it to immediately reboot the system on abnormal kernel termination. This option is quite typical in real-world situations where the controller system should work without any downtime. Finally, the fourth option tells it to be doubly verbose with debug output. For many more options, consult helpviewer.

procnto is the QNX Neutrino microkernel of the QNX Neutrino RTOS (integrated with process manager, hence the name). It provides the basic functionality of creating processes and threads, and performs message passing, memory management, etc. You can pass specific options here, for example, "**-p**" if you like to live dangerously and want to disable preemption. Specifying the "**PATH**" and "**LD_LIBRARY_PATH**" variables makes every spawned process inherit them. Not passing them almost always means trouble. "**PATH**" specifies the program search path and "**LD_LIBRARY_PATH**" specifies the shared library search path.

At this point the bootstrap process is over and the kernel is now ready to spawn processes that will put the system to work.

Startup script

This script is executed after the bootstrap process is over. All resource managers (drivers) and application programs are started here. Let's take a look at a typical script:

```
[+script] .script = {
# symlink this critical library. Some apps search it there
#procmgr_symlink is internal equivalent to "ln -s"
procmgr_symlink ../../proc/boot/libc.so /usr/lib/ldqnx.so.2
#seedres fills kernel data structure with appropriate
#system specific values (IRQs, DMA channels etc)
seedres
# If your system has PCI interface (most
# 100MHz Intel® Pentium® processors) then
# start pci server otherwise don't use it
pci-bios &
# wait till pci-bios probes system
# required only if you've started pci server
waitfor /dev/pci
#start NIC driver with full TCP/IP stack
#with half duplex operation in this case
io-net -d tulip duplex=0 -p tcpip
#wait until io-net starts
waitfor /dev/socket
# configure NIC
ifconfig en0 192.168.4.100 netmask 255.255.255.0
```

```
route add default 192.168.4.1
# start console driver with two virtual consoles
# accessible at ctrl+alt+1 and ctrl+alt+2
devc-con -n2 &
# start shell on the consoles
#open stdin,stdout,stderr directed to con1
reopen /dev/con1
# start shell as session leader
[+session] TERM=qansi uesh &
#open stdin,stdout,stderr directed to con2
reopen /dev/con2
# start shell as session leader
[+session] TERM=qansi uesh &
}
```

Again, you may have noticed that this script closely resembles a shell script. The attribute block "**[+script]**" at the beginning of the filename "**.script**" signifies that it is a script to be executed after the system is ready.

This script is laden with comments, so much of the content should be clear. However, there are some points to be noted. You must not start PCI server "**pci-bios**" if you don't have a PCI bus in your system. This applies to, but is not limited to, 486s (and most Intel® Pentium® processors up to 90MHz) motherboards. If you do have a PCI bus and have a card on it that you want to use (e.g. NIC or video card), you must start PCI server.

The QNX Neutrino RTOS v6.1 doesn't have "**nettrap**", a utility to discover NICs and appropriate drivers. So, either find this information yourself by figuring out the chipset of your NIC, or use "**enum-devices -n**" to see which driver is mounted at io-net and use it.

If you're short of space (e.g., when using a flash disk or booting from boot ROM) you may want to replace the BSD TCP/IP stack with the tiny TCP/IP stack. This has the benefit of saving a lot in size: *ttcpip.so* is only 40% of the size of *tcpip.so*. Additionally you no longer need the utilities "**ifconfig**" and "**route**". By replacing the full stack you gain over 150kB. Unfortunately you'll have to sacrifice some less used TCP/IP features. But don't worry, most applications will work absolutely fine with the tiny stack. The command line, in this case, becomes (the \ indicates continuation of single line):

```
io-net -d tulip duplex=0 -p ttcpip \
if=en0:192.168.4.100:255.255.255.0 default=192.168.4.1
```

One more thing you might have noticed is that this script is generously littered with "**waitfor**" statements. This is a built-in command that blocks execution of the script until the specified path appears. Most of the commands in the QNX Neutrino RTOS are non-blocking in nature. This has both good and bad implications. Good because you can start another simultaneous process and take advantage of multiprocessing if both processes are mutually independent; and bad because it poses problems when a new process depends on the work of a former one. For instance, in the example above, the NIC card is a PCI one so io-net can't start unless the PCI server is up and running. "**ifconfig**" and "route" are similar cases. They can't run unless io-net is finished configuring the NIC. To get around this situation you can wait until the device node of that device driver appears, which signals that the device driver is up and running. The third part of the buildfile tells you which files are necessary in order to get this stuff going.

File list

Here's the file list that's used to make this stuff:

```
#Files to be included
#these will end up in /proc/boot
libc.so
/lib/dll/devn-tulip.so
/lib/dll/npm-tcpip.so
libsocket.so
#This is an example of explicitly specifying path of
# destination and source
/etc/termcap=/etc/termcap
/etc/hosts=/etc/hosts
#Executable programs to be loaded
[code=uip data=copy perms=+r,+x]
/sbin/io-net
/usr/bin/ifconfig
/usr/bin/route
/usr/bin/telnet
/usr/bin/ftp
devc-con
# uesh is a tiny shell, use "ksh" if you have enough space
uesh
pci-bios
```

I have broken the explanation of this File List into three parts.

Part 1: Shared Libraries

The initial lines denote shared libraries. The first one, "*libc.so*" is almost always required since most C applications, including system utilities, are dynamically linked against "*libc.so*". The other libraries are for the NIC driver, the TCP/IP stack, and the socket library for networking. The specified files end up in /proc/boot on the target system upon booting. So what if you want your files to end up in some other specific location? The above example also shows how you explicitly specify the location of source and destination files. Note that directory structure creation is automatic, so you don't have to worry about that. But even this doesn't solve a common problem with shared libraries.

Part 2: Symbolic links

Some programs expect the libraries to be in one location while others might expect them to be somewhere else. Worse, many applications expect that a particular version of a file will be named in a specific way while others expect another way. (e.g., *libexpat.so* might be referred to as libexpat.so.1 if its version 1.0) Making copies of the library will be a sheer waste of space. To overcome this, symbolic link support is added. Instead of creating link at run time, as explained formerly, you can specify the links to be created at the time of making image. This is done using "type=link" attribute as follows:

```
[type=link] /usr/lib/ldqnx.so=/proc/boot/libc.so
```

This would make a symbolic link *"/usr/lib/ldqnx.so*" pointing to *"/proc/boot/libc.so*". I'd suggest you to use **procmgr_symlink** to create symbolic links to files/paths created/mounted at run time, and use the above mentioned approach for creating symbolic links to static files/paths included in buildfile.

Part 3: Executables

The final section is marked by attributes "`[code=uip data=copy perms=+r,+x]`". The first part of the attribute description, "`code=uip`", says that the following lines will specify binary executables which should be available in memory ready to run, when needed. Remember when the system boots, the files in the image are in RAM so their code can be used directly from memory instead of copying it and running it from another portion of RAM. Only the data segment has to be "copied" for each instance. This is called running the program "in place". It is possible to run a single launch program entirely in-place by requesting its data segment to be "**uip**" as well. In this case, the programs should not be run more than once, since they would overwrite the memory of the first running program, resulting in some undesirable side effects! The second part of the attribute description defines permissions given to the files by "**perms=+r,+x**" which indicate read and execute permissions. After the attributes, the files to be loaded are listed.

Unlink list (optional)

This is a QNX Neutrino RTOS v6.1 only feature. The point to note here is that /proc is not on any physical medium. It actually represents the kernel's internal data structures in RAM in a running system. So, /proc/boot, where all your files end up, also exists in RAM. Since those applications are running now, and you won't be running many of them again from same location, keeping a copy of them in RAM is a waste of resources. You can instruct the system that you don't need a particular set of files and that they are safe to delete. Keep in mind that currently running executables have the attribute "`code=uip`" and/or "`data=uip`" and won't be deleted since they're being run straight from image memory space. Deleting would mean killing them! Wildcard characters are allowed in the unlink list. Just make sure that your programs don't require running them from this location after the boot up process is over.

```
unlink_list={ /proc/boot/chkfsys /proc/boot/seedres /proc/boot/ifconfig
/proc/boot/route }
```

So now that you have a buildfile, be sure and read the second half of the article to learn how to use the buildfile to make the image.