Building a Boot Image

Introduction

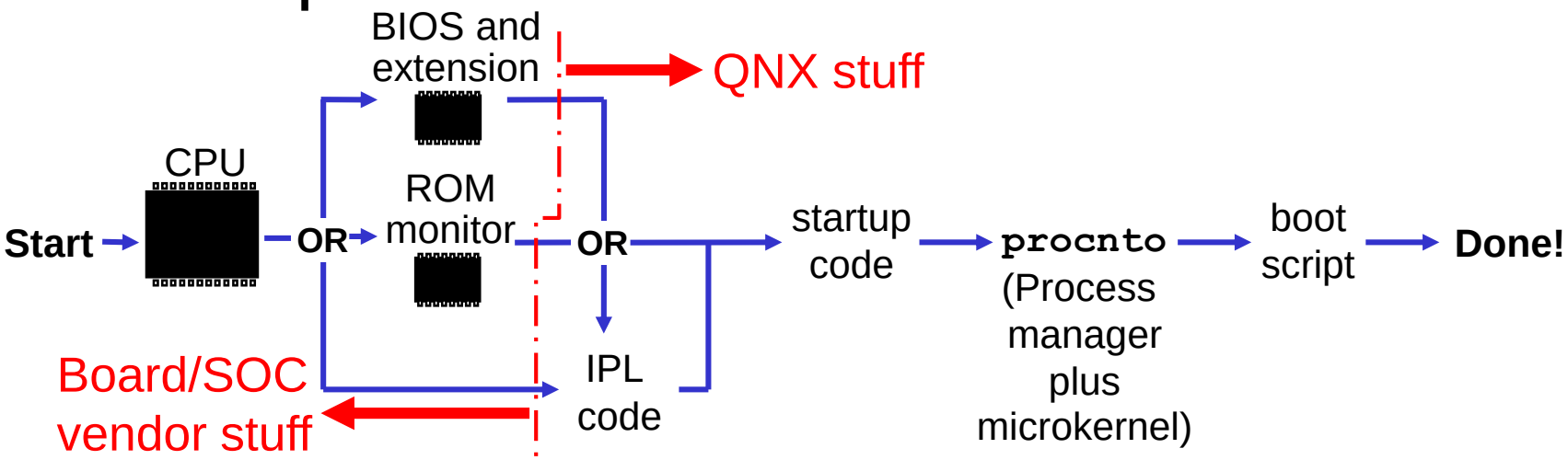
You will learn:

- the QNX boot sequence
- the format of QNX buildfiles
- creating a bootable image

Topics:

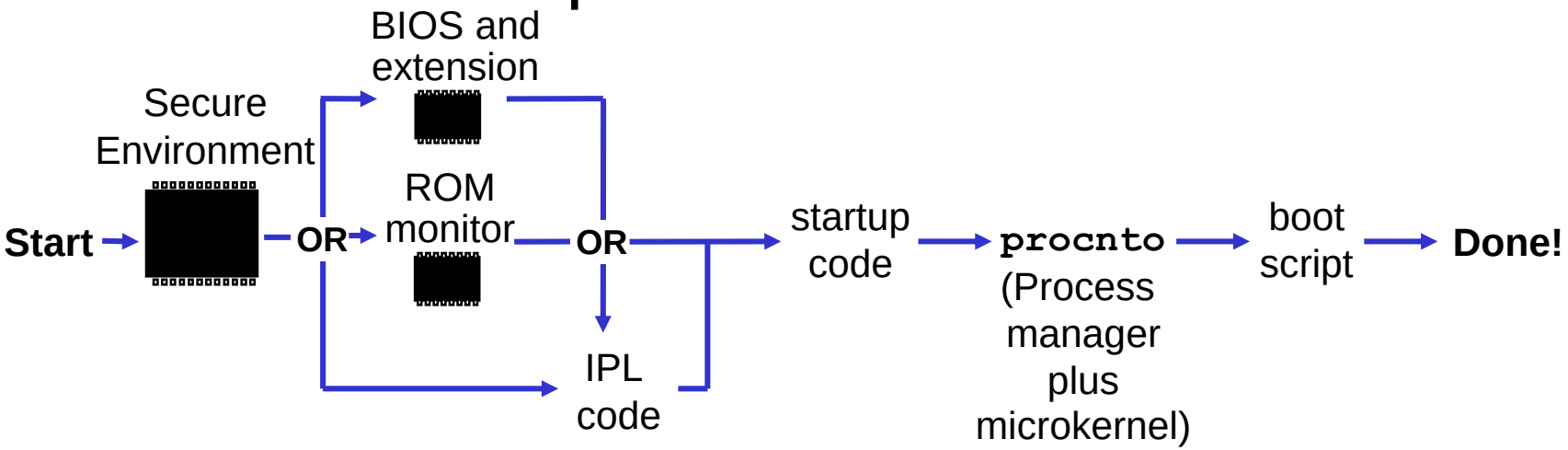
- **Images & Buildfiles**
- **Loading**
- **Exercise**
- **Conclusion**

Boot sequence:



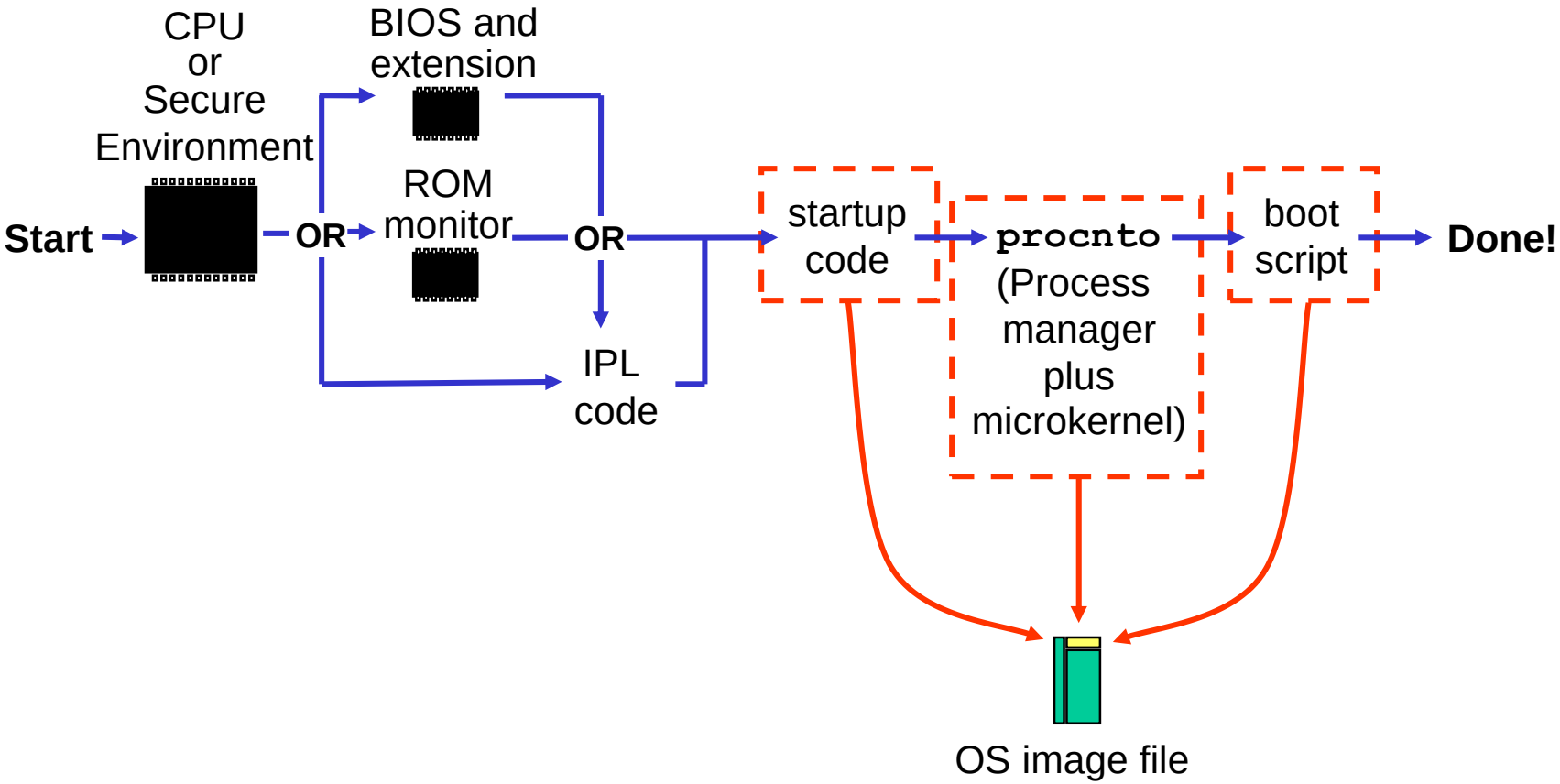
- IPL code:
  - does chip selects and sets up RAM, then jumps to startup code
- startup code:
  - sets up some hardware and prepares environment for **`procnto`**
- **`procnto`**:
  - sets up kernel and runs boot script
- the boot script contains:
  - drivers and other processes, including yours

Secure Boot sequence:



- Secure Environment
  - validates next stage (usually hash and signature)
  - if validation passes, jumps to next stage
- some variability:
  - usually entire image validated by secure environment
  - sometimes stages through IPL
- **`fs-qtd.so`**:
  - hashed filesystem extends chain-of-trust past boot to a filesystem

Much of this is in an OS image file:



What is an image?

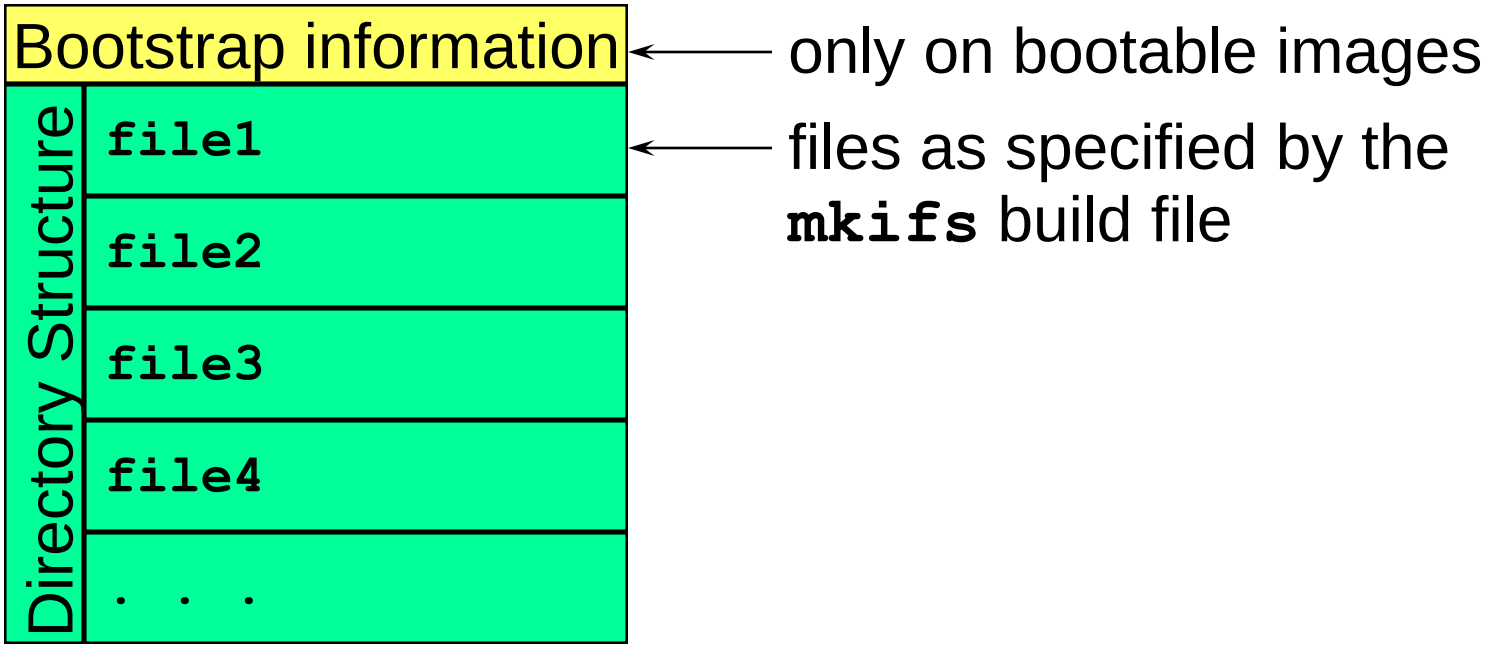
- a file
- contains executables, and/or data files
- can be bootable

After boot, contents presented as a filesystem:

- default: **`/proc/boot`**
- simple
- read-only
- memory-based

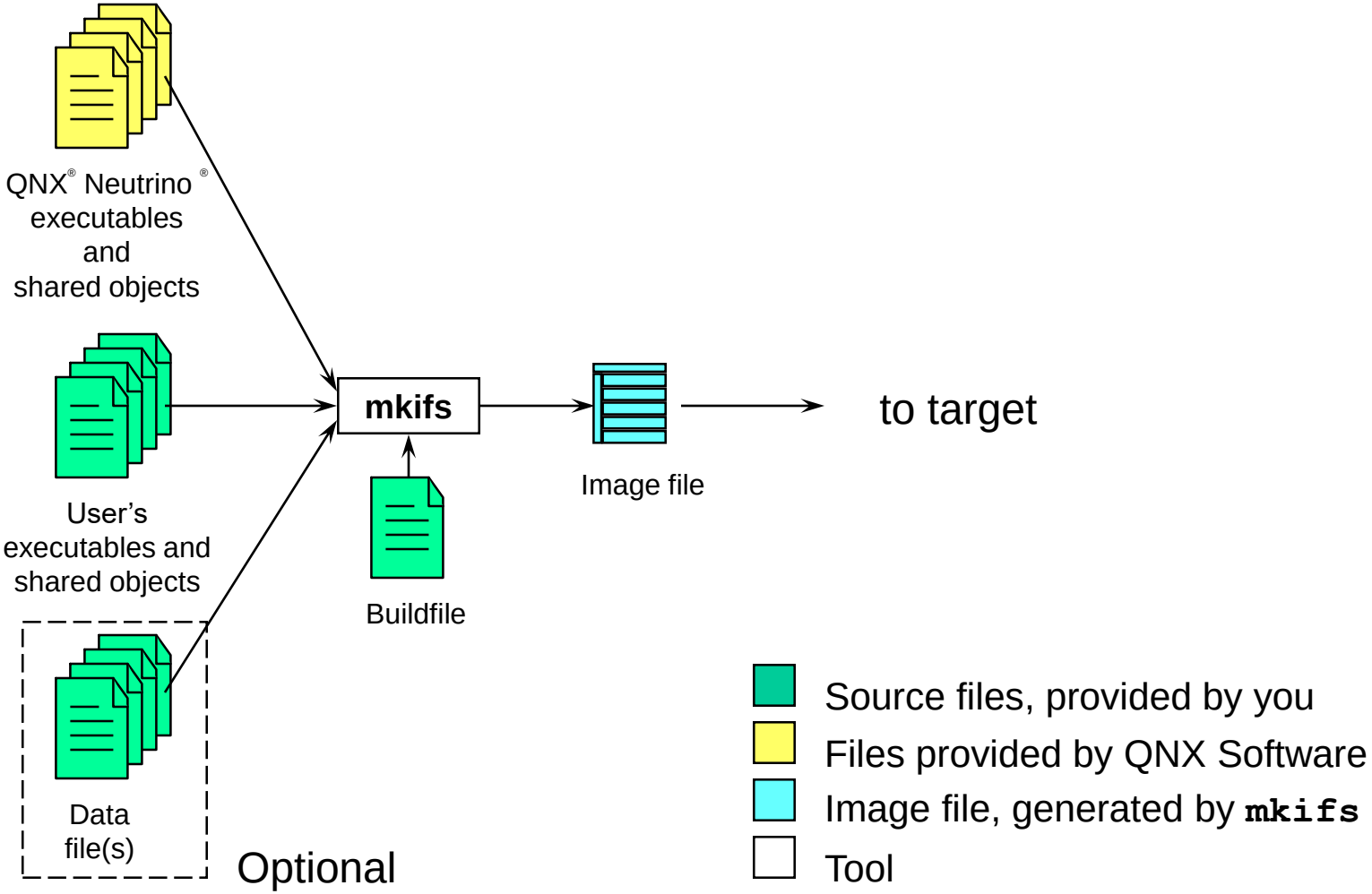
What is an image?

Image:



The image tool chain

Images are created by the following:



What is a buildfile?

- a text file that specifies:
  - files / commands that will be included in the image
  - loading options for files & executables.
  - options for creating the image (e.g. compression)
  - for bootable images, a boot script that says what to run after OS initialization

Buildfile Contents

A buildfile for a bootable image ***must*** contain:

- bootstrap loader:
  - **`startup-`*`boardname`***
- and operating system:
  - **`procnto-smp-instr`**
- boot script
- executables and shared libraries
  - executables aren't strictly required, but then the system wouldn't actually ***do*** anything without them!
  - and, in most cases, to run any executable, you need at least **`libc.so`**, a shared library

A sample build file:

```
#   This is “hello.build”
[virtual=x86_64,multiboot] .bootstrap = {
    startup-x86
    PATH=/proc/boot procnto-smp-instr
}
[+script] .script = {
    procmgr_symlink /proc/boot/ldqnx-64.so.2 /usr/lib/ldqnx-64.so.2
    devc-ser8250 -e -b115200 &
    reopen /dev/ser1
    hello
}
libc.so
libgcc_s.so.1
ldqnx-64.so.2
devc-ser8250
hello
```

To make an image from this, do:

```
mkifs hello.build hello.ifs
```

Buildfile format

General format of a buildfile:

```
[attribute] filename = contents
[attribute] filename = contents
```

...

– Can include blank lines and comments as well (comments begin with the pound sign, “#”)

– All components are optional

  • cannot have contents without a filename

Attributes

There are two types of attributes:

- Boolean
  - **`[+attribute]`**
    - turns on the specified attribute (e.g. **`[+script]`**)
  - **`[-attribute]`**
    - turns off the specified attribute (e.g. **`[-optional]`**)
- Value
  - **`[attribute=`*`value`*`]`**
    - assigns a value to an attribute type (e.g. **`[uid=0]`**)

Attributes

Attributes can apply to single files:

- as in the following:

```
[uid=7] file1_owned_by_user7
[uid=6 gid=5] file2_owned_by_user6
```

Or to all subsequent files:

- as in the following:

```
[-optional]
file1_owned_by_user7
file2_owned_by_user7
```

Adding Files

To add files to your image, list them in your buildfile:

```
devc-ser8250
/etc/hosts = /project_files/target/etc/hosts
```

- find **devc-ser8250** and put it in **/proc/boot**
- explicitly supply path for target (**/etc/hosts**) and host (**/project_files/**…)

Or a file can be given in line:

```
readme = {
   This is a handy way to get a file into the image
without actually having a file.  The file, readme, will be
accessible as /proc/boot/readme.
}
```

Attributes for making sure you got all files:

- **`[-optional]`**
  - tells **`mkifs`** to stop and display an error if a file is not found
  - we recommend this one, put it by itself at the start of the build file so that it applies to all files
- **`[autoso=list]`**
  - says to display a list of any shared libraries (**`.so`** files) which you've missed
- **`[autoso=add]`**
  - says to add any shared libaries (**`.so`** files) which you've missed

Scripts

Using the [+script] attribute, a file is treated as a script:

- It will be executed after the process manager has completed its startup.
- Multiple scripts will be concatenated into one and be interpreted in the order given.
- There are modifiers that can be placed before commands to run:

  Example: **`[pri=27f] esh`**
- There are also some builtin commands:

  Example: **`reopen /dev/con1`**

Example script:

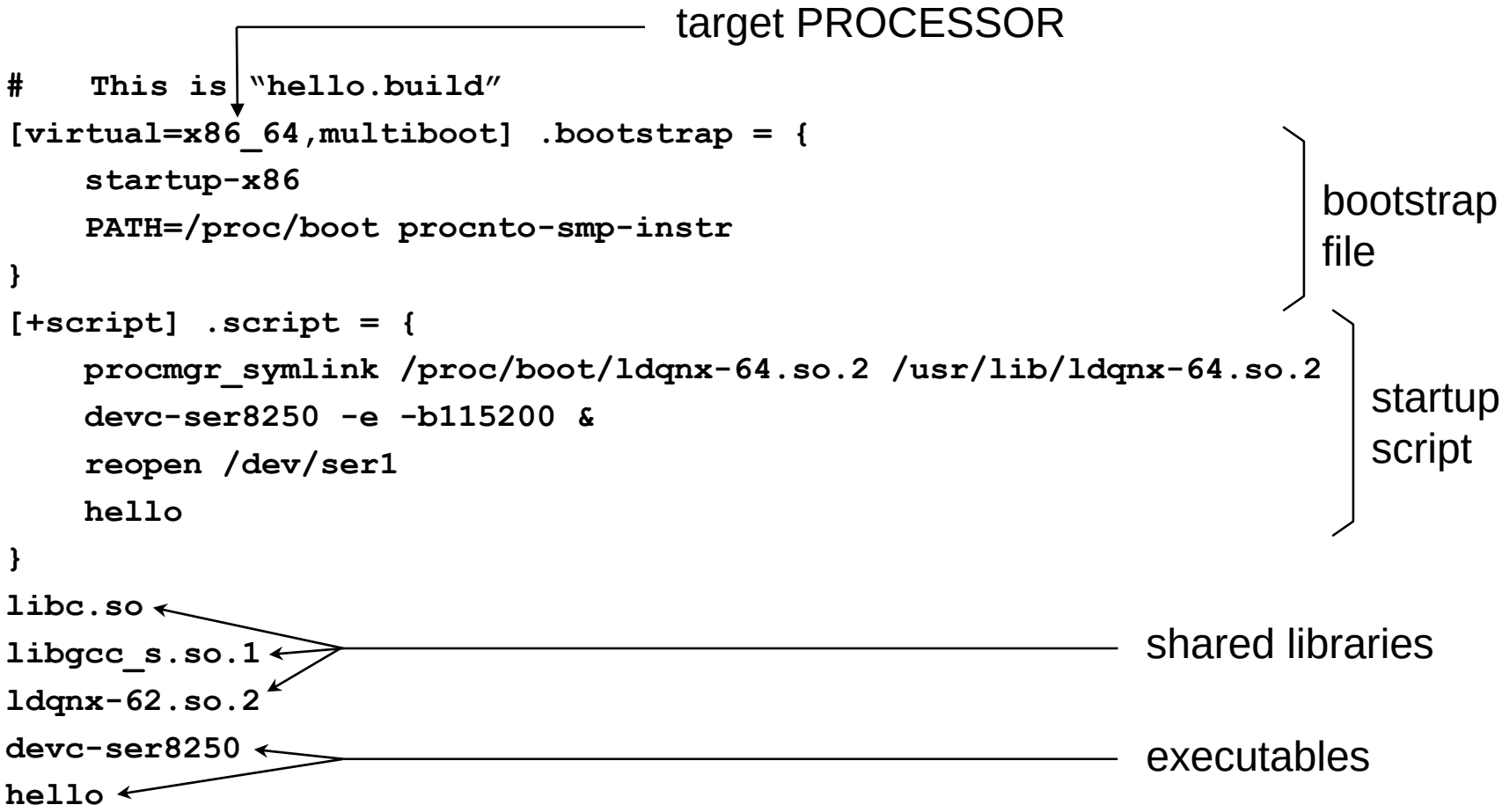
```
[+script] .script = {
    procmgr_symlink /proc/boot/ldqnx-64.so.2 /usr/lib/ldqnx-64.so.2

    display_msg Starting serial driver
    devc-ser8250 -e -b115200 &
    waitfor /dev/ser1 # don't continue until /dev/ser1 exists

    display_msg Starting pseudo-tty driver
    devc-pty &

    display_msg Setting up consoles
    devc-con &
    reopen /dev/con2  # set stdin, stdout and stderr to /dev/con2
    [+session pri=27r] PATH=/proc/boot esh &
    reopen /dev/con1  # set stdin, stdout and stderr to /dev/con1
    [+session pri=10r] PATH=/proc/boot esh &
}
```

Internal Commands

Internal commands are:

- ones that **`mkifs`** recognizes and are not loaded from the host's filesystem:
  - **`display_msg`** outputs the given text
  - **`procmgr_symlink`** is the equivalent of **`ln -P`**, except that you don't have to have **`ln`** present
  - **`reopen`** causes stdin, stdout, and stderr to be redirected to the given filename
  - **`waitfor`** waits until a *stat()* on the given pathname succeeds
- there are no testing, branching, or looping constructs
  - more complicated initialization should be done through scripts or programs

The pieces of our **hello.build** example we saw earlier:



To find files, mkifs, looks in:

`${QNX_TARGET}/${PROCESSOR}/bin`, `../usr/bin`, `../sbin`, `../usr/sbin`	binaries (**`esh`**, **`ls`**, etc)
`${QNX_TARGET}/${PROCESSOR}/boot/sys`	OSes (**`procnto`**, etc)
`${QNX_TARGET}/${PROCESSOR}/lib`, `../usr/lib`	libraries and shared objects
`${QNX_TARGET}/${PROCESSOR}/lib/dll`	shared objects

- The above can be overridden:
  - using the **`MKIFS_PATH`** environment variable:

```
MKIFS_PATH=/usr/nto/x86/bin:
  /usr/nto/x86/sys:/usr/nto/x86/dll:
  /usr/nto/x86/lib:/project/bin
```

    - most BSPs modify this
  - using the **`search`** attribute for a particular file:

```
[search={MKIFS_PATH}:/project/bin] myexec
```

Topics:

- Images & Buildfiles
- → **Loading**
- Exercise
- Conclusion

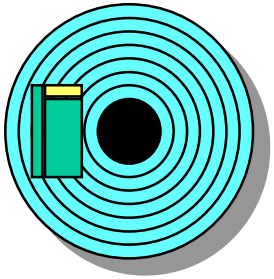
Disk

The image can be loaded from a disk containing a QNX 6 filesystem:

- build image (using **`mkifs`**)

```
cp image /.boot/image.ifs
```

- /.boot directory can contain multiple images
  - by default, most recent image is booted
  - alternate image can be selected from list at boot time



QNX 6 filesystem bootloader copies image to RAM

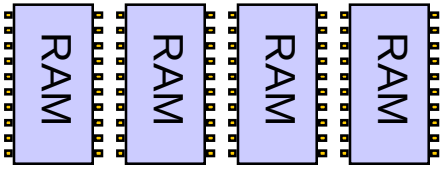


Image stored on pseudo-disk (Flash):

- custom (or manufacturer supplied) tools required

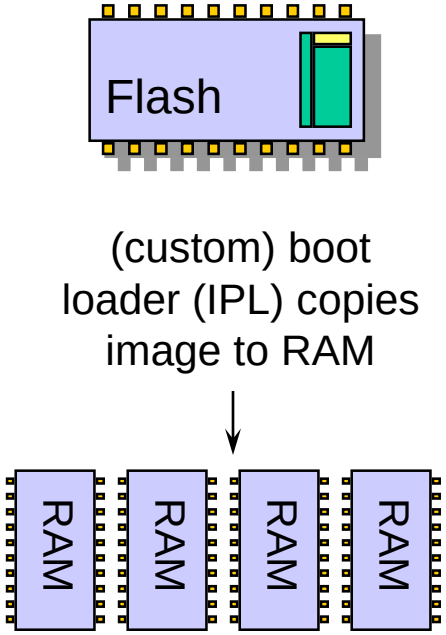
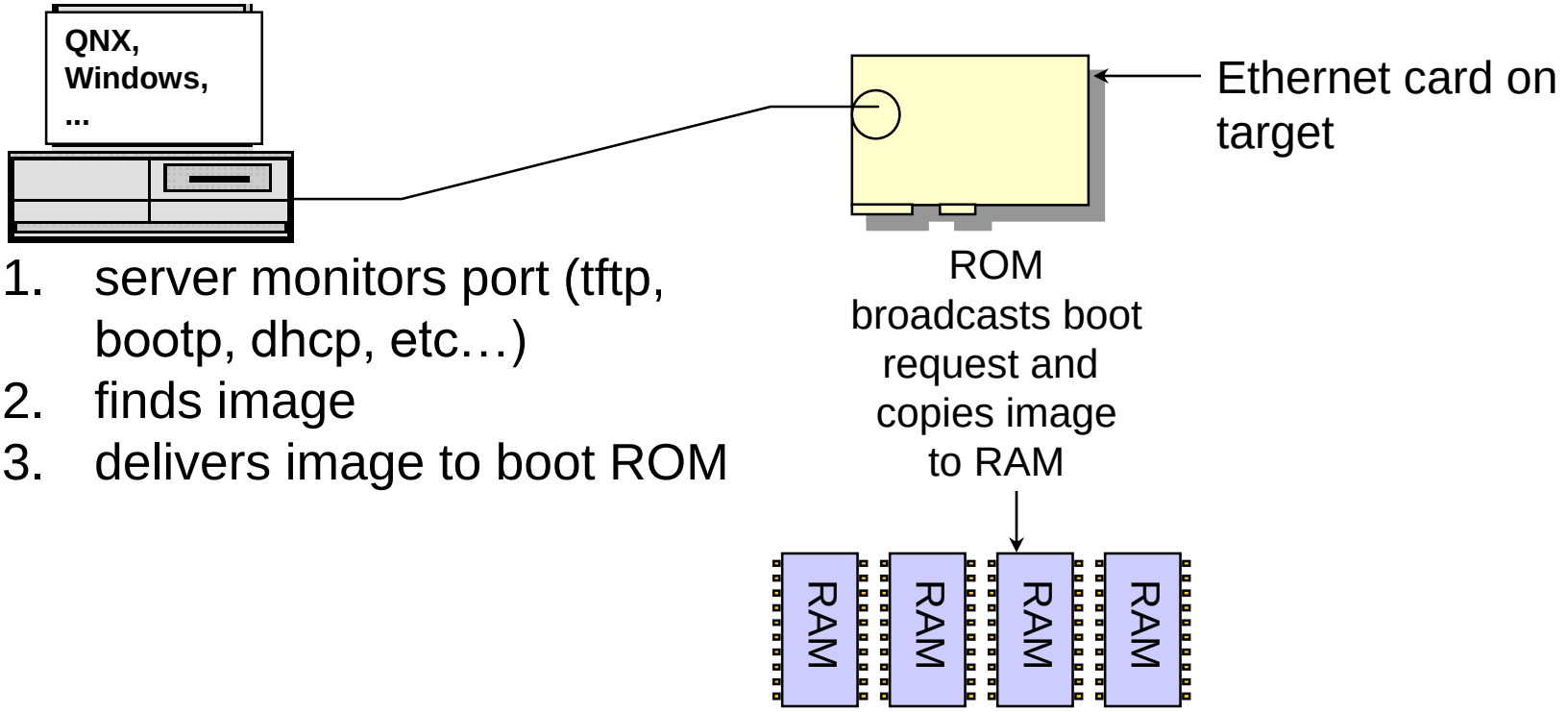


Image can be transferred across network:



1. server monitors port (tftp, bootp, dhcp, etc…)
2. finds image
3. delivers image to boot ROM

Topics:

- **Images & Buildfiles**
- **Loading**
- → **Exercise**
- **Conclusion**

Exercise (x86_64):

- start and connect to the VM supplied by the instructor for this exercise
- this uses your **`images`** project
- modify the buildfile for your target to:
  - display a “hello” message
  - add an existing executable from an existing project
    - e.g. **`hello`**
  - run that executable
- build the image
- copy the new image to **`/bootdisk/.boot`** directory on your target
- reboot the target and select your image from the list
  - use **`shutdown`** to reboot, so the filesystem has a chance to write its journal

Exercise (BSP):

- using the QNX Software Center, download a BSP (e.g. an x86_64 one)
- import it into the IDE:
  - File → Import → QNX → QNX Source Package and BSP
  - choose Browse for ZIP Archive…
  - browse to the **`bsp`** folder under wherever you installed the QNX SDP
  - select the zip file for the BSP
  - click Next a few times and then Finish
- open the project that the IDE just created
  - in the **`images`** folder, open one of the buildfiles
- review the various sections of the build and the attributes and commands
- clean the project and rebuild the images

Topics:

- **Images & Buildfiles**
- **Loading**
- **Exercise**
- → **Conclusion**

Conclusion

You learned:

- what happens at boot time
- how to modify a buildfile
- how to create a bootable image