



Sysdev - Tiny embedded system with BusyBox

Objective: making a tiny yet full featured embedded system.

After this lab, you will

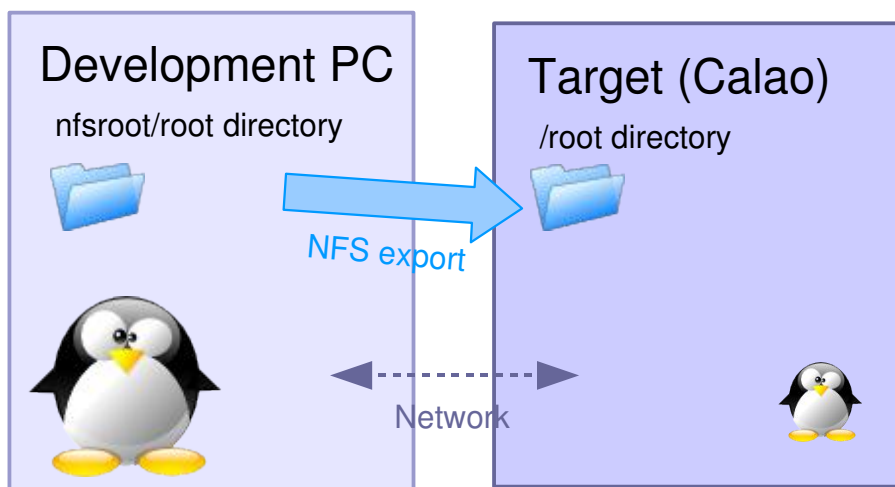
- be able to configure and build a Linux kernel that boots on a directory on your workstation, shared through the network by NFS.
- be able to create and configure a minimalistic root filesystem **from scratch** (ex nihilo, out of nothing, entirely hand made...) for the Calao board
- understand how small and simple an embedded Linux system can be.
- be able to install BusyBox on this filesystem.
- be able to create a simple startup script based on `/sbin/init`.
- be able to set up a simple web interface for the target.
- have an idea of how much RAM a Linux kernel smaller than 1 MB needs.

Lab implementation

While (s)he develops a root filesystem for a device, a developer needs to make frequent changes to the filesystem contents, like modifying scripts or adding newly compiled programs.

It isn't practical at all to reflash the root filesystem on the target every time a change is made. Fortunately, it is possible to set up networking between the development workstation and the target. Then, workstation files can be accessed by the target through the network, using NFS.

Unless you test a boot sequence, you no longer need to reboot the target to test the impact of script or application updates.





Setup

Go to the `/home/<user>/felabs/sysdev/tinysystem/` directory.

Get the sources from the latest 2.6.35.x release and place them in the current directory.

Kernel configuration

Set the `ARCH` and `CROSS_COMPILE` kernel Makefile variables for cross-compiling to the arm instruction set.

Configure the 2.6.35 kernel sources with the configuration file supplied in the `data/` subdirectory.

Add the configuration options that enable booting on a root filesystem sitting on an NFS remote directory.

Compile your kernel and generate the uImage kernel image suitable for U-Boot.

Root filesystem with Busybox

Create an `nfsroot/` directory that will contain the root filesystem for the target.

Download the latest BusyBox 1.17.x release and configure it with the configuration file provided in the `data/` directory.

At least, make sure you build BusyBox statically!

Build BusyBox using the toolchain that you used to build the kernel.

Install BusyBox in the root filesystem by running `make install`.

Setting up the NFS server

Install the NFS server by installing the `nfs-kernel-server` package if you don't have it yet. Once installed, create the `/etc/exports` file as root to add the following line, assuming that the IP address of your board will be 192.168.0.100 (the path and the options must be on the same line!)

```
/home/<user>/felabs/sysdev/tinysystem/nfsroot
192.168.0.100(rw,no_root_squash,no_subtree_check)
```

Then, restart the NFS server:

```
sudo /etc/init.d/nfs-kernel-server restart
```

Booting the system

First, boot the board to the U-Boot prompt.

First temporarily disable automatic booting after `tftp` and `nboot`:

```
setenv autostart no
```

Put the uImage file of your compiled kernel in the directory exported by TFTP. Download and flash your kernel to the board, as done in the previous lab. Now, you can restore autostart:

```
setenv autostart yes
```

Before booting the kernel, we need to tell it that the root filesystem should be mounted over NFS, by setting some kernel parameters.

Other TCP/IP networking configuration settings are already enabled in the provided configuration file.

Compiling Busybox statically in the first place makes it easy to set up the system, because there are no dependencies on libraries. Later on, we will set up shared libraries and recompile Busybox.



First, in Minicom, type [Ctrl] [a] [w] to enable line wrapping. This will allow you to type long lines like the one that follows.

Use the following U-Boot command to do so (in just 1 line):

```
setenv bootargs root=/dev/nfs ip=192.168.0.100  
nfsroot=192.168.0.1:/home/<user>/felabs/sysdev/tinysystem/nfsroot
```

Of course, you need to adapt the IP addresses to your exact network setup. Save the environment variables (with `saveenv`).

Try to boot your new system on the board. If everything goes right, the kernel should confirm that it managed to mount the NFS root filesystem. Then, you should get errors about missing `/dev/ttyX` files. Create them with the `mknod` command (using the same major and minor number as in your GNU/Linux workstation). Try again.

At the end, you will access a console and will be able to issue commands through the default shell.

Virtual filesystems

Run the `ps` command. You can see that it complains that the `/proc` directory does not exist. The `ps` command and other process-related commands use the `proc` virtual filesystem to get their information from the kernel.

From the Linux command line in the target, create the `proc`, `sys` and `etc` directories in your root filesystem.

Now mount the `proc` virtual filesystem.

Now that `/proc` is available, test again the `ps` command.

Note that you can also halt your target in a clean way with the `halt` command, thanks to `proc` being mounted.

System configuration and startup

The first userspace program that gets executed by the kernel is `/sbin/init` and its configuration file is `/etc/inittab`.

In the BusyBox sources, read details about `/etc/inittab` in the `examples/inittab` file.

Then, create a `/etc/inittab` file and a `/etc/init.d/rcS` startup script declared in `/etc/inittab`. In this startup script, mount the `/proc` and `/sys` filesystems.

Any issue after doing this?

Switching to shared libraries

Take the `hello.c` program supplied in the `data` directory. Cross-compile it for ARM, dynamically-linked with the libraries, and run it on the target.

You will first encounter a `not found` error caused by the absence of the `ld-uClibc.so.0` executable, which is the dynamic linker required to execute any program compiled with shared libraries. Using the `find` command (see examples in your command memento sheet), look for this file in the `toolchain` install directory, and copy it to the `lib/` directory on the target.

Then, running the executable again and see that the loader

You can understand our approach to build filesystems from scratch. We're waiting for programs to complain before adding device or configuration files. This is a way of making sure that every file in the filesystem is used.

Actually, you will probably have several instructive surprises when trying to implement this. Don't hesitate to share your questions with your instructor!



executes and finds out which shared libraries are missing. Similarly, find these libraries in the toolchain and copy them to `lib/` on the target.

Once the small test program works, recompile Busybox without the static compilation option, so that Busybox takes advantages of the shared libraries that are now present on the target.

Implement a web interface for your device

Replicate `data/www/` to the `/www` directory in your target root filesystem.

Now, run the BusyBox http server from the command line:
`/usr/sbin/httpd -h /www/ &`

If you use a proxy, configure your host browser so that it doesn't go through the proxy to connect to the target IP address, or simply disable proxy usage. Now, test that your web interface works well by opening `http://192.168.0.100:80` on the host.

See how the dynamic pages are implemented. Very simple, isn't it?

How much RAM does your system need?

Check the `/proc/meminfo` file and see how much RAM is used by your system.

You can try to boot your system with less memory, and see whether it still works properly or not. For example, to test whether 6 MB are enough, boot the kernel with the `mem=6M` parameter. Linux will then use just 6 MB of RAM, and ignore the rest.