



Third party libraries and applications

Objective: Learn how to leverage existing libraries and applications: how to configure, compile and install them.

To illustrate how to use existing libraries and applications, we will extend the small root filesystem built in the “A tiny embedded system” lab to add the DirectFB graphic library and sample applications using this library. Because the Calao board doesn't provide a display, we will test the result of this lab with Qemu.

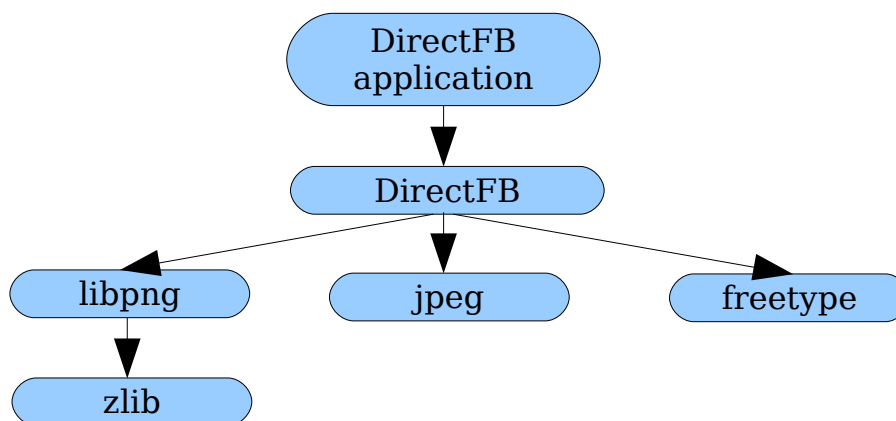
We'll see that manually re-using existing libraries is quite tedious, so that more automated procedures are necessary to make it easier. However, learning how to perform these operations manually will significantly help you when you'll face issues with more automated tools.

Figuring out library dependencies

As most libraries, DirectFB depends on other libraries, and these dependencies are different depending on the configuration chosen for DirectFB. In our case, we will enable support for:

- PNG image loading;
- JPEG image loading;
- Font rendering using a font engine.

The PNG image loading feature will be provided by the `libpng` library, the JPEG image loading feature by the `jpeg` library and the font engine will be implemented by the `freetype` library. The `libpng` library itself depends on the `zlib` compression/decompression library. So, we end up with the following dependency tree:



Of course, all these libraries rely on the C library, which is not mentioned here, because it is already part of the root filesystem built in the “A tiny embedded system” lab. You might wonder how to figure out this dependency tree by yourself. Basically, there are several ways, that can be combined:

- Read the library documentation, which often mentions the dependencies;
- Read the help message of the configure script (by running `./configure --help`).



- By running the configure script, compiling and looking at the errors.

To configure, compile and install all the components of our system, we're going to start from the bottom of the tree with `zlib`, then continue with `libpng`, `jpeg` and `freetype`, to finally compile `DirectFB` and the `DirectFB` sample applications.

Preparation

For our cross-compilation work, we will need to separate spaces:

- A «staging» space in which we will directly install all the packages: non-stripped versions of the libraries, headers, documentation and other files needed for the compilation. This «staging» space can be quite big, but will not be used on our target, only for compiling libraries or applications;
- A «target» space, in which we will copy only the required files from the «staging» space: binaries and libraries, after stripping, configuration files needed at runtime, etc. This target space will take a lot less space than the «staging» space, and it will contain only the files that are really needed to make the system work on the target.

To sum up, the «staging» space will contain everything that's needed for compilation, while the «target» space will contain only what's needed for execution.

So, in `/home/<user>/felabs/sysdev/thirdparty`, create two directories: `staging` and `target`.

For the target, we need a basic system with `BusyBox`, device nodes and initialization scripts. We will re-use the system built in the “A tiny embedded system” lab, so copy this system in the target directory:

```
sudo cp -a /home/<user>/felabs/sysdev/tinysystem/nfsroot/* target/
```

The copy must be done as root, because the root filesystem of the “A tiny embedded system” lab contains a few device nodes.

Testing

Make sure the `target/` directory is exported by your NFS server by adding the following line to `/etc/exports`:

```
/home/<user>/felabs/sysdev/thirdparty/rootfs 172.20.0.2(rw, no_root_squash, no_subtree_check)
```

And restart your NFS server. Then, run Qemu with the provided script:

```
./run_qemu
```

The system should boot and give you a prompt.

zlib

Zlib is a compression/decompression library available at <http://www.zlib.net/>. Download version 1.2.3, and extract it in `/home/<user>/felabs/sysdev/thirdparty/`.

By looking at the configure script, we see that this configure script



has not been generated by autoconf (otherwise it would contain a sentence like « Generated by GNU Autoconf 2.62 »). Moreover, the project doesn't use automake since there are no Makefile.am files. So zlib uses a custom build system, not a build system based on the classical autotools.

Let's try to configure and build zlib:

```
./configure  
make
```

You can see that the files are getting compiled with gcc, which generates code for x86 and not for the target platform. This is obviously not what we want, so we tell the configure script to use the ARM cross-compiler:

```
CC=arm-linux-gcc ./configure
```

Of course, the arm-linux-gcc cross-compiler must be in your PATH prior to running the configure script. The CC environment variable is the classical name for specifying the compiler to use. Moreover, the beginning of the configure script tells us about this:

```
# To impose specific compiler or flags or  
# install directory, use for example:  
# prefix=$HOME CC=cc CFLAGS="-O4" ./configure
```

Now when you compile with make, the cross-compiler is used. Look at the result of compiling: a set of object files, and a file libz.a, generated by the following command:

```
ar rc libz.a adler32.o compress.o crc32.o gzio.o uncompr.o  
deflate.o trees.o zutil.o inflate.o inffast.o  
inftrees.o
```

This libz.a file is the static version of the zlib library. It can be used to compile applications linked statically with the zlib library, as shown by the compilation of the example program:

```
arm-linux-gcc -O3 -DUSE_MMAP -o example example.o -L.  
libz.a
```

As we want to use dynamic libraries as much as possible, let's tell the zlib configure script to generate a shared library. You can look at help information from configure script by running ./configure --help to discover that it provides a --shared option. Let's use it:

```
CC=arm-linux-gcc ./configure --shared
```

Now, the compilation process also generates libz.so.1.2.3:

```
arm-linux-gcc -shared -Wl,-soname,libz.so.1 -o  
libz.so.1.2.3 adler32.o compress.o crc32.o gzio.o  
uncompr.o deflate.o trees.o zutil.o inflate.o inffast.o  
inftrees.o
```

And creates symbolic links libz.so and libz.so.1:

```
ln -s libz.so.1.2.3 libz.so  
ln -s libz.so.1.2.3 libz.so.1
```

These symlinks are needed for two different reasons:

- libz.so is used at compile time when you want to compile an application that is dynamically linked against the library. To do so, you pass the -llibname option to the compiler, which



will look for a file named `lib<LIBNAME>.so`. In our case, the compilation option is `-lz` and the name of the library file is `libz.so`. So, the `libz.so` symlink is needed at compile time;

- `libz.so.1` is needed because it is the SONAME of the library. SONAME stands for « *Shared Object Name* ». It is the name of the library as it will be stored in applications linked against this library. It means that at runtime, the dynamic loader will look for exactly this name when looking for the shared library. So this symbolic link is needed at runtime.

To know what's the SONAME of a library, you can use:

```
arm-linux-readelf -d libz.so.1.2.3
```

and look at the (SONAME) line. You'll also see that this library needs the C library, because of the (NEEDED) line on `libc.so.0`.

The mechanism of SONAME allows to change the library without recompiling the applications linked with this library. Let's say that a security problem is found in `zlib 1.2.3`, and fixed in the next release `1.2.4`. You can recompile the library, install it on your target system, change the link `libz.so.1` so that it points to `libz.so.1.2.4` and restart your applications. And it will work, because your applications don't look specifically for `libz.so.1.2.3`, but for the SONAME `libz.so.1`. However, it also means that as a library developer, if you break the ABI of the library, you must change the SONAME: change from `libz.so.1` to `libz.so.2`.

Finally, the last step is to tell the configure script where the library is going to be installed. Most configure scripts consider that the installation prefix is `/usr/local/` (so that the library is installed in `/usr/local/lib`, the headers in `/usr/local/include`, etc.). But in our system, we simply want the libraries to be installed in the `/usr` prefix, so let's tell the configure script about this:

```
CC=arm-linux-gcc ./configure --shared --prefix=/usr  
make
```

For the `zlib` library, this option may not change anything to the resulting binaries, but for safety, it is always recommended to make sure that the prefix matches where your library will be running on the target system.

Do not confuse the prefix (where the application or library will be running on the target system) from the location where the application or library will be installed on your host while building the root filesystem. For example, `zlib` will be installed in `/home/<user>/felabs/sysdev/thirdparty/target/usr/lib/` because this is the directory where we are building the root filesystem, but once our target system will be running, it will see `zlib` in `/usr/lib`. The prefix corresponds to the path in the target system and **never** on the host. So, one should **never** pass a prefix like `/home/<user>/felabs/sysdev/thirdparty/target/usr`, otherwise at runtime, the application or library may look for files inside this directory on the target system, which obviously doesn't exist! By default, most build systems will install the application or library in the given prefix (`/usr` or `/usr/local`), but with most build systems (including autotools), the installation prefix can be overridden, and be different from the configuration prefix.

Unfortunately, the custom build system used by `zlib` doesn't provide



such a feature, so we must install the library manually. Fortunately, the number of files to install is limited.

First, let's make the installation in the «staging» space:

1. Create the `staging/usr/lib` and `staging/usr/include` directories. They will contain the binaries of all the libraries and the header files;
2. Copy both the static and dynamic versions of the `zlib` library, by copying `libz.a` and all `libz.so` files in `staging/usr/lib`:
`cp -a libz.a libz.so* ../staging/usr/lib`
The `-a` option of `cp` will preserve symbolic links.
3. Copy the headers needed to compile applications against `zlib`:
`cp zconf.h zlib.h ../staging/usr/include/`

Now, let's install the library in the «target» space:

1. Create the `target/usr/lib` directory, it will contain the stripped version of the library
2. Copy the dynamic version of the library. Only `libz.so.1` and `libz.so.1.2.3` are needed, since `libz.so.1` is the SONAME of the library and `libz.so.1.2.3` is the real binary:
`cp -a libz.so.1* ../target/usr/lib`
3. Strip the library:
`arm-linux-strip ../target/usr/lib/libz.so.1.2.3`

Ok, finally we're done with `zlib`!

Libpng

Download `libpng` from its official website at <http://www.libpng.org/pub/png/libpng.html>. We tested the lab with version 1.2.35.

Once uncompressed, we quickly discover that the `libpng` build system is based on the `autotools`, so we will work with a regular `configure` script.

As we've seen previously, if we just run `./configure`, the build system will use the native compiler to build the library, which is not what we want. So let's tell the build system to use the cross-compiler:

```
CC=arm-linux-gcc ./configure
```

Quickly, you should get an error saying:

```
configure: error: cannot run C compiled programs.  
If you meant to cross compile, use `--host'.  
See `config.log' for more details.
```

If you look at `config.log`, you quickly understand what's going on:

```
configure:2942: checking for C compiler default output file name  
configure:2964: arm-linux-gcc conftest.c >&5  
configure:2968: $? = 0  
configure:3006: result: a.out  
configure:3023: checking whether the C compiler works  
configure:3033: ./a.out  
./configure: line 3035: ./a.out: cannot execute binary file
```

The `configure` script compiles a binary with the cross-compiler and then tries to run it on the development workstation. Obviously, it cannot work, and the system says that it «cannot execute binary



file». The job of the configure script is to test the configuration of the system. To do so, it tries to compile and run a few sample applications to test if this library is available, if this compiler option is supported, etc. But in our case, running the test examples is definitely not possible. We need to tell the configure script that we are cross-compiling, and this can be done using the `--build` and `--host` options, as described in the help of the configure script:

System types:

```
--build=BUILD  configure for building on BUILD
                [guessed]
--host=HOST    cross-compile to build programs to run
                on HOST [BUILD]
```

The `--build` option allows to specify on which system the package is built, while the `--host` option allows to specify on which system the package will run. By default, the value of the `--build` option is guessed and the value of `--host` is the same as the value of the `--build` option. The value is guessed using the `./config.guess` script, which on your system should return `i686-pc-linux-gnu`. See http://www.gnu.org/software/autoconf/manual/html_node/Specifying-Names.html for more details on these options.

So, let's override the value of the `--host` option:

```
CC=arm-linux-gcc ./configure --host=arm-linux
```

Now, we go a little bit further in the execution of the configure script, until we reach:

```
checking for zlibVersion in -lz... no
configure: error: zlib not installed
```

Again, we can check in `config.log` what the configure script is trying to do:

```
configure:12452: checking for zlibVersion in -lz
configure:12487: arm-linux-gcc -o conftest -g -O2
conftest.c -lz -lm >&5
/usr/local/uclibc-0.9.30/arm/usr/bin/./lib/gcc/arm-linux-
uclibcgnueabi/4.3.3/./././././arm-linux-
uclibcgnueabi/bin/ld: cannot find -lz
collect2: ld returned 1 exit status
```

The configure script tries to compile an application against `zlib` (as can be seen from the `-lz` option) : `libpng` uses the `zlib` library, so the configure script wants to make sure this library is already installed. Unfortunately, the `ld` linker doesn't find this library. So, let's tell the linker where to look for libraries using the `-L` option followed by the directory where our libraries are (in `staging/usr/lib`). This `-L` option can be passed to the linker by using the `LDFLAGS` at configure time, as told by the help text of the configure script:

```
LDFLAGS  linker flags, e.g. -L<lib dir> if you have
         libraries in a nonstandard directory <lib dir>
```

Let's use this `LDFLAGS` variable:

```
LDFLAGS=-L/home/<user>/felabs/sysdev/thirdparty/staging/
usr/lib \
CC=arm-linux-gcc ./configure --host=arm-linux
```

Let's also specify the prefix, so that the library is compiled to be installed in `/usr` and not `/usr/local`:



```
LDFLAGS=-L/home/<user>/felabs/sysdev/thirdparty/staging/  
usr/lib \  
CC=arm-linux-gcc ./configure --host=arm-linux \  
--prefix=/usr
```

Then, run the compilation using make. Quickly, you should get a pile of error messages, starting with:

```
In file included from png.c:13:  
png.h:470:18: error: zlib.h: No such file or directory
```

Of course, since libpng uses the zlib library, it includes its header file! So we need to tell the C compiler where the headers can be found: there are not in the default directory `/usr/include/`, but in the `/usr/include` directory of our «staging» space. The help text of the configure script says:

```
CPPFLAGS          C/C++/Objective C preprocessor flags,  
                  e.g. -I<include dir> if you have headers  
                  in a nonstandard directory <include dir>
```

Let's use it:

```
LDFLAGS=-  
L/home/<user>/felabs/sysdev/thirdparty/staging/usr/lib \  
CPPFLAGS=-  
I/home/<user>/felabs/sysdev/thirdparty/staging/usr/include \  
CC=arm-linux-gcc ./configure --host=arm-linux \  
--prefix=/usr
```

Then, run the compilation with make. Hopefully, it works!

Let's now begin the installation process. Before really installing in the staging directory, let's install in a dummy directory, to see what's going to be installed (this dummy directory will not be used afterwards, it is only to verify what will be installed before polluting the staging space):

```
make DESTDIR=/tmp/libpng/ install
```

The `DESTDIR` variable can be used with all Makefiles based on automake. It allows to override the installation directory: instead of being installed in the `configuration-prefix`, the files will be installed in `DESTDIR/configuration-prefix`.

Now, let's see what has been installed in `/tmp/libpng/`:

```
./usr/lib/libpng.la          → libpng12.la  
./usr/lib/libpng12.a        → libpng12.a  
./usr/lib/libpng12.so.0     → libpng12.so.0.35.0  
./usr/lib/libpng.so.3.35.0 → libpng12.so.3.35.0  
./usr/lib/libpng12.so.0.35.0 → libpng12.so.0.35.0  
./usr/lib/libpng.a         → libpng12.a  
./usr/lib/libpng.so        → libpng12.so  
./usr/lib/libpng12.la      → libpng12.la  
./usr/lib/libpng12.so      → libpng12.so.0.35.0  
./usr/lib/libpng.so.3     → libpng12.so.3.35.0  
./usr/lib/pkgconfig/libpng.pc → libpng12.pc  
./usr/lib/pkgconfig/libpng12.pc → libpng12.pc  
./usr/share/man/man5/png.5 → libpng12/png.5  
./usr/share/man/man3/libpngpf.3 → libpng12/libpngpf.3  
./usr/share/man/man3/libpng.3 → libpng12/libpng.3  
./usr/include/pngconf.h    → libpng12/pngconf.h  
./usr/include/png.h        → libpng12/png.h
```



```
./usr/include/libpng12/pngconf.h
./usr/include/libpng12/png.h
./usr/bin/libpng-config          → libpng12-config
./usr/bin/libpng12-config
```

So, we have:

- The library itself. In the case of libpng, it is quite complicated since there are in fact two different versions of the libraries: libpng12.so.0.35.0 and libpng.so.3.35.0. The first is the current library, while the second one is the obsolete version, that still exists for old applications. In the end we have:
 - libpng12.so.0.35.0, the binary of the current version of library
 - libpng.so.3.35.0, the binary of the obsolete version of the library
 - libpng.so.3, a symbolic link to libpng.so.3.35.0, so that applications using libpng.so.3 as the SONAME of the library will still find and use the obsolete version against which they were compiled for
 - libpng12.so.0, a symbolic link to libpng12.so.0.35.0, so that applications using libpng12.so.0 as the SONAME of the library will find it and use the current version
 - libpng12.so is a symbolic link to libpng12.so.0.35.0. So it points to the current version of the library, so that new applications linked with -lpng12 will use the current version of the library
 - libpng.so is a symbolic link to libpng12.so. So applications linked with -lpng will be linked with the current version of the library (and not the obsolete one since we don't want anymore to link applications against the obsolete version!)
 - libpng12.a is a static version of the library
 - libpng.a is a symbolic link to libpng12.a, so that applications statically linked with libpng.a will in fact use the current version of the library
 - libpng12.la is a configuration file generated by libtool which gives configuration details for the library. It will be used to compile applications and libraries that rely on libpng.
 - libpng.la is a symbolic link to libpng12.la: we want to use the current version for new applications, once again.
- The pkg-config files, in /usr/lib/pkgconfig/. These configuration files are used by the pkg-config tool that we will cover later. They describe how to link new applications against the library.
- The manual pages in /usr/share/man/, explaining how to use the library.
- The header files, in /usr/include/, needed to compile new applications or libraries against libpng. They define the interface to libpng. There are symbolic links so that one can choose between the following solutions:



- Use `#include <png.h>` in the source code and compile with the default compiler flags
- Use `#include <png.h>` in the source code and compile with `-I/usr/include/libpng12`
- Use `#include <libpng12/png.h>` in the source and compile with the default compiler flags
- The `/usr/bin/libpng12-config` tool and its symbolic link `/usr/bin/libpng-config`. This tool is a small shell script that gives configuration informations about the libraries, needed to know how to compile applications/libraries against libpng. This mechanism based on shell scripts is now being superseded by `pkg-config`, but as old applications or libraries may rely on it, it is kept for compatibility.

Because the obsolete version of libpng is not necessary, we will skip its compilation. The help text of the configure script says:

Optional Packages:

```
--with-PACKAGE[=ARG]  use PACKAGE [ARG=yes]
--without-PACKAGE     do not use PACKAGE
                     (same as --with-PACKAGE=no)
```

[...]

```
--with-libpng-compat  Generate the obsolete libpng.so
                     library [default=yes]
```

So, the default is to compile the obsolete library (as we've seen), and to disable it, we need to pass `--without-libpng-compat`. Let's do this:

```
LDFLAGS=-
L/home/<user>/felabs/sysdev/thirdparty/staging/usr/lib \
CPPFLAGS=-
I/home/<user>/felabs/sysdev/thirdparty/staging/usr/include \
CC=arm-linux-gcc ./configure --host=arm-linux \
--prefix=/usr --without-libpng-compat
```

And rebuild with `make`. Remove `/tmp/libpng` and try again the installation:

```
make DESTDIR=/tmp/libpng install
```

Then, check in `/tmp/libpng/usr/lib`, you shouldn't have the obsolete version of the library anymore. So, let's make the installation in the «staging» space:

```
make
DESTDIR=/home/<user>/felabs/sysdev/thirdparty/staging/ install
```

Then, let's install only the necessary files in the «target» space, manually:

```
cp -a staging/usr/lib/libpng12.so.0* target/usr/lib
arm-linux-strip target/usr/lib/libpng12.so.0.35.0
```

And we're finally done with libpng!

libjpeg

Now, let's work on libjpeg. Download it from <http://www.ijg.org/files/jpegsrc.v6b.tar.gz> and extract it. Unfortunately, the build system of libjpeg is severely broken in terms of cross-compilation, so we need to apply a few patches to it:

```
cp data/libjpeg/config.sub jpeg-6b/
```



```
cp data/libjpeg/config.guess jpeg-6b/
```

The `config.guess` script is used to guess the machine type of the host on which we're building software:

```
$ ./config.guess
i386-pc-linux-gnu
```

The `config.sub` script is used to canonicalize a machine type:

```
$ ./config.sub arm-linux
arm-unknown-linux-gnu
```

Then, apply two patches to the libjpeg build system:

```
cd jpeg-6b
cat ../data/libjpeg/jpeg-build.patch | patch -p1
cat ../data/libjpeg/jpeg-libtool.patch | patch -p1
```

The first patch, `jpeg-build.patch`, fixes several issues in the Makefile:

- Respect the options coming from the configure script in terms of `bindir`, `libdir`, `includedir` and `mandir`;
- Use the `AR` environment variable instead of hardcoding the usage of `ar`;
- Fix the installation steps to respect `$(DESTDIR)`
- Install the missing `jpegint.h` header

The second patch, `jpeg-libtool.patch`, fixes the `configure` script and the `ltconfig` scripts, to support cross-compilation and systems based on `uClibc`.

These fixes have been borrowed from the Buildroot build system, available at <http://buildroot.uclibc.net>.

Then, the configuration is now very similar to the configuration of the previous libraries, except for the `--enable-shared` and `--enable-static` options that are needed to enable the compilation of both the shared and static variants of the library:

```
CC=arm-linux-gcc ./configure --host=arm-linux \
--enable-shared --enable-static --prefix=/usr
```

Of course, compile the library:

```
make
```

Installation to the «staging» space can be done using the classical `DESTDIR` mechanism, thanks to the patch applied previously:

```
make
DESTDIR=/home/<user>/felabs/sysdev/thirdparty/staging/ install
```

And finally, install manually the only needed files at runtime in the «target» space:

```
cp -a staging/usr/lib/libjpeg.so.62* target/usr/lib/
arm-linux-strip target/usr/lib/libjpeg.so.62.0.0
```

Done with libjpeg!

freetype

The Freetype library is the next step. Grab the tarball from <http://www.freetype.org>. We tested the lab with version 2.3.9 but



more other versions may also work. Uncompress the tarball.

The Freetype build system is a nice example of what can be done with a good usage of the autotools. Cross-compiling Freetype is very easy. First, the configure step:

```
CC=arm-linux-gcc ./configure --host=arm-linux \
--prefix=/usr
```

Then, compile the library:

```
make
```

Install it in the «staging» space:

```
make
DESTDIR=/home/<user>/felabs/sysdev/thirdparty/staging/ install
```

And install only the required files in the «target» space:

```
cp -a staging/usr/lib/libfreetype.so.6* target/usr/lib/
arm-linux-strip target/usr/lib/libfreetype.so.6.3.20
```

Done with Freetype!

DirectFB

Finally, with zlib, libpng, jpeg and freetype, all the dependencies of DirectFB are ready. We can now build the DirectFB library itself. Download it from the official website, at <http://www.directfb.org/>. We tested version 1.2.7 of the library. As usual, extract the tarball.

Before configuring DirectFB, let's have a look at the available options by running `./configure --help`. A lot of options are available. We see that:

- Support for Fbdev (the Linux framebuffer) is automatically detected, so that's fine;
- Support for PNG, JPEG and Freetype is enabled by default, so that's fine;
- We should specify a value for `--with-gfxdrivers`. The hardware emulated by Qemu doesn't have any accelerated driver in DirectFB, so we'll pass `--with-gfxdrivers=none`;
- We should specify a value for `--with-inputdrivers`. We'll need keyboard (for the keyboard) and linuxinput to support the Linux Input subsystem. So we'll pass `--with-inputdrivers=keyboard,linuxinput`

So, let's begin with a configure line like:

```
CC=arm-linux-gcc ./configure --host=arm-linux \
--prefix=/usr --with-gfxdrivers=none \
--with-inputdrivers=keyboard,linuxinput
```

In the output, we see:

```
*** JPEG library not found. JPEG image provider will not be
built.
[...]
configure: WARNING: *** libz not found. PNG image provider will
not be built.
[...]
checking for libpng-config... no
configure: WARNING:
*** libpng-config not found
```




ng-config

In fact, this step was not really necessary, as DirectFB configure script falls back to a working default value when the `libpng-config` executable doesn't exist. However, as this mechanism of `libxxx-config` executable is quite common, we thought it was worth spending a little extra time describing how it works.

Let's continue the analysis of the configure script output. We can also read:

```
FreeType2                yes
-I/usr/include/freetype2  -lfreetype -lz
```

Which obviously doesn't look good: we shouldn't be looking at headers files in `/usr/include/`, but only in the `/usr/include` directory of our «staging» space.

In fact, the DirectFB configure script uses the `pkg-config` system to get the configuration parameters to link the library against Freetype. By default, `pkg-config` looks in `/usr/lib/pkgconfig/` for `.pc` files, and because the `libfreetype6-dev` package is already installed in your system (it was installed in a previous lab as a dependency of another package), then the configure script of DirectFB found the Freetype library of your host! This is one of the biggest issue with cross-compilation : mixing host and target libraries, because build systems have a tendency to look for libraries in the default paths. In our case, if `libfreetype6-dev` was not installed, then the `/usr/lib/pkgconfig/freetype2.pc` file wouldn't exist, and the configure script of DirectFB would have said «Sorry, can't find Freetype».

So, now, we must tell `pkg-config` to look inside the `/usr/lib/pkgconfig/` directory of our «staging» space. This is done through the `PKG_CONFIG_PATH` environment variable, as explained in the manual page of `pkg-config`.

Moreover, the `.pc` files contain references to paths. For example, in `/home/<user>/felabs/sysdev/thirdparty/staging/usr/lib/pkgconfig/freetype2.pc`, we can see:

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include
[...]
Libs: -L${libdir} -lfreetype
Cflags: -I${includedir}/freetype2 -I${includedir}
```

So we must tell `pkg-config` that these paths are not absolute, but relative to our «staging» space. This can be done using the `PKG_CONFIG_SYSROOT_DIR` environment variable.

Unfortunately, This is only possible with `pkg-config >= 0.23`, which is not yet available in the Ubuntu distribution. So start by installing the `pkg-config` Ubuntu package available in the `data/` directory of the lab.

Then, let's run the configuration of DirectFB again, passing the `PKG_CONFIG_PATH` and `PKG_CONFIG_SYSROOT_DIR` environment variables:

```
LDFLAGS=-
L/home/<user>/felabs/sysdev/thirdparty/staging/usr/lib \
```



```
CPPFLAGS=-
I/home/<user>/felabs/sysdev/thirdparty/staging/usr/include \
PKG_CONFIG_PATH=/home/<user>/felabs/sysdev/thirdparty/staging/us
r/lib/pkgconfig/ \
PKG_CONFIG_SYSROOT_DIR=/home/<user>/felabs/sysdev/thirdparty/sta
ging/ \
PATH=/home/<user>/felabs/sysdev/thirdparty/staging/usr/bin:$PATH
\
CC=arm-linux-gcc ./configure --host=arm-linux \
--prefix=/usr --with-gfxdrivers=none \
--with-inputdrivers=keyboard,linuxinput
```

Ok, now, the line related to Freetype 2 looks much better:

```
FreeType2          yes
-I/home/<user>/felabs/sysdev/thirdparty/staging/usr/include/freet
ype2          -lfreetype
```

Let's build DirectFB with make, and then install it to the «staging» space using:

```
make
DESTDIR=/home/<user>/felabs/sysdev/thirdparty/staging/ install
```

And so the installation in the «target» space:

- First, the libraries:

```
cp -a staging/usr/lib/libdirect-1.2.so.0*
target/usr/lib

cp -a staging/usr/lib/libdirectfb-1.2.so.0*
target/usr/lib

cp -a staging/usr/lib/libfusion-1.2.so.0*
target/usr/lib

arm-linux-strip target/usr/lib/libdirect-
1.2.so.0.7.0

arm-linux-strip target/usr/lib/libdirectfb-
1.2.so.0.7.0

arm-linux-strip target/usr/lib/libfusion-
1.2.so.0.7.0
```
- Then, the plugins that are dynamically loaded by DirectFB. We first copy the whole /usr/lib/directfb-1.2-0/ directory, then remove the useless files (.o, .a and .la) and finally strip the .so files:

```
cp -a staging/usr/lib/directfb-1.2-0/ target/usr/lib

find target/usr/lib/directfb-1.2-0/ -name '*.o' -o -
name '*.a' -o -name '*.la' -exec rm {} \;

find target/usr/lib/directfb-1.2-0/ -name '*.so'
-exec arm-linux-strip {} \;
```



DirectFB examples

To test that our DirectFB installation works, we will use the example applications provided by the DirectFB project. Start by downloading the tarball at <http://www.directfb.org/downloads/Extras/DirectFB-examples-1.2.0.tar.gz> and extract it.

Then, we configure it just as we configured DirectFB:

```
LDFLAGS=-
L/home/<user>/felabs/sysdev/thirdparty/staging/usr/lib \
CPPFLAGS=-
I/home/<user>/felabs/sysdev/thirdparty/staging/usr/include \
PKG_CONFIG_PATH=/home/<user>/felabs/sysdev/thirdparty/staging/us
r/lib/pkgconfig/ \
PKG_CONFIG_SYSROOT_DIR=/home/<user>/felabs/sysdev/thirdparty/sta
ging/ \
PATH=/home/<user>/felabs/sysdev/thirdparty/staging/usr/bin:$PATH \
CC=arm-linux-gcc ./configure --host=arm-linux \
--prefix=/usr
```

Then, compile it with make. Soon a compilation error will occur because “bzero” is not defined. The “bzero” function is a deprecated BSD function, and `memset` should be used instead. The GNU C library still defines “bzero”, but by default, the uClibc library doesn't provide “bzero” (to save space). So, let's modify the source code in `src/df_knuckles/matrix.c` to change the line:

```
#define M_CLEAR(m) bzero(m, MATRIX_SIZE)
```

to

```
#define M_CLEAR(m) memset(m, 0, MATRIX_SIZE)
```

Run the compilation again, it should succeed.

For the installation, as DirectFB examples are only applications and not libraries, we don't really need them in the «staging» space, but only in the «target» space. So we'll directly install in the «target» space using the `install-strip` make target. This make target is always available with autotools based build systems. In addition to the destination directory (`DESTDIR` variable), we must also tell which strip program should be used, since stripping is an architecture-dependent operation (`STRIP` variable):

```
make STRIP=arm-linux-strip \
DESTDIR=/home/<user>/felabs/sysdev/thirdparty/target/
install-strip
```

Final setup

Start the system in Qemu using the `run_qemu` script, and try to run the `df_andi` program, which is one of the DirectFB examples.

The application will fail to run, because the pthread library (which is a component of the C library) is missing. This library is available inside the toolchain. So let's add it to the target:

```
cp -a /usr/local/uclibc-0.9.30/arm/lib/libpthread* \
target/lib/
```

Then, try to run `df_andi` again. It will complain about `libdl`, which is used to dynamically load libraries during application execution. So let's add this library to the target:



```
cp -a /usr/local/uclibc-0.9.30/arm/lib/libdl* \  
target/lib
```

When running `df_andi` again, it will complain about `libgcc_s`, so let's copy this library to the target:

```
cp -a /usr/local/uclibc-0.9.30/arm/usr/arm-linux-  
uclibcgnueabi/lib/libgcc_s* target/lib
```

Now, the application should no longer complain about missing library. But when started, should complain about inexistent `/dev/fb0` device file. So let's create this device file:

```
sudo mknod target/dev/fb0 c 29 0
```

Next execution of the application will complain about missing `/dev/tty0` device file, so let's create it:

```
sudo mknod target/dev/tty0 c 4 0
```

And then will complain again about missing `/dev/tty6` device file, so let's do:

```
sudo mknod target/dev/tty6 c 4 6
```

Finally, when running `df_andi`, another error message shows up:

```
Unable to dlopen '/usr/lib/directfb-1.2-  
0/interfaces/IDirectFBImageProvider/libidirectfbimageprovi  
der_png.so' !  
→ File not found
```

DirectFB is trying to load the PNG plugin using the `dlopen()` function, which is part of the `libdl` library we added to the target system before. Unfortunately, loading the plugin fails with the «File not found» error. However, the plugin is properly present, so the problem is not the plugin itself. What happens is that the plugin depends on the `libpng` library, which itself depends on the `mathematic` library. And the `mathematic` library `libm` (part of the C library) has not yet been added to our system. So let's do it:

```
cp -a /usr/local/uclibc-0.9.30/arm/lib/libm* target/lib
```

Now, you can try and run the `df_andi` application!