



Sysdev - Application development and remote debugging

Objective:

Compile and run your own DirectFB application on the target.

Use strace and ltrace to diagnose program issues.

Use gdbserver and a cross-debugger to remotely debug an embedded application

Setup

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

Compile your own application

In this part, we will re-use the system built during the «Buildroot lab» and add to it our own application.

First, instead of using an ext2 image, we will mount the root filesystem over NFS to make it easier to test our application. So, create a `qemu-rootfs/` directory, and inside this directory, uncompress the tarball generated by Buildroot in the previous lab (in the `output/binaries/` directory). Don't forget to extract the archive as root since the archive contains device files.

Then, adapt the `run_qemu` script to your configuration, and verify that the system works as expected.

Now, our application. In the lab directory the file `data/app.c` contains a very simple DirectFB application that displays the `data/background.png` image for five seconds. We will compile and integrate this simple application to our Linux system.

Let's try to compile the application:

```
arm-linux-gcc -o app app.c
```

It complains that it cannot find the `directfb.h` header. This is normal, since we didn't tell the compiler where to find it. So let's use `pkg-config` to query the `pkg-config` database about the location of the header files and the list of libraries needed to build an application against DirectFB:

```
export
PKG_CONFIG_PATH=/home/<user>/felabs/sysdev/buildroot/output/staging/usr/lib/pkgconfig
```

```
export
PKG_CONFIG_SYSROOT_DIR=/home/<user>/felabs/sysdev/buildroot/output/staging/
```

```
arm-linux-gcc -o app app.c $
(/home/<user>/felabs/sysdev/buildroot/output/host/usr/bin/
pkg-config --libs --cflags directfb)
```

Compiling fails again, because the compiler could not find the library. The simplest solution to fix this issue is to use the `sysroot` feature of the cross-compiling toolchain: we can tell the toolchain against which directory the paths like `/usr/lib`, `/lib` or `/usr/include` should be interpreted.



```
arm-linux-gcc --sysroot
/home/<user>/felabs/sysdev/buildroot/output/staging/ -o
app app.c $
(/home/<user>/felabs/sysdev/buildroot/output/host/usr/bin/
pkg-config --libs --cflags directfb)
```

Our application is now compiled! Copy the generated binary and the background.png image to the NFS root filesystem (in the root/ directory for example), start your system, and run your application !

Debugging setup

For the debugging part we don't need an emulated LCD anymore, so we will move back to your ARM board. Boot your ARM board over NFS on the filesystem produced in the «Tiny embedded system» lab, with the same kernel.

Setting up gdbserver, strace and ltrace

`gdbserver`, `strace` and `ltrace` have already been compiled for your target architecture as part of the cross-compiling toolchain. Find them in the installation directory of your toolchain. Copy these binaries to the `/usr/bin/` directory in the root filesystem of your target system.

Enabling job control

In this lab, we are going to run a buggy program that keeps hanging and crashing. Because of this, we are going to need job control, in particular `[Ctrl] [C]` allowing to interrupt a running program.

At boot time, you probably noticed that warning that job control was turned off:

```
/bin/sh: can't access tty; job control turned off
```

This happens when the shell is started in the console. The system console cannot be used as a controlling terminal.

A work around is to start this shell in `ttys0` (the first serial port) by modifying the `/etc/inittab` file:

```
Replace
::askfirst:/bin/sh          (implying /dev/console)
by
ttyS0::askfirst:/bin/sh    (using /dev/ttyS0)
```

Also create `/dev/ttyS0` and reboot. You should no longer see the "Job control turned off" warning, and should be able to use `[Ctrl] [C]`.

Using strace

`strace` allows to trace all the system calls made by a process: opening, reading and writing files, starting other processes, accessing time, etc. When something goes wrong in your application, `strace` is an invaluable tool to see what it actually does, even when you don't have the source code.

With your cross-compiling toolchain, compile the `data/vista-emulator.c` program, and copy the resulting binary to the `/root` directory of the root filesystem (you might need to create this

Caution: if your board uses a serial port device different from `/dev/ttyS0`, don't forget to use the correct name instead of `ttys0` (example: `ttys2` on the Beagle board).



directory if it doesn't exist yet).

```
arm-linux-gcc -o vista-emulator data/vista-emulator.c
cp vista-emulator path/to/root/filesystem/root
```

Back to target system, try to run the `/root/vista-emulator` program. It should hang indefinitely!

Interrupt this program by hitting [Ctrl] [C].

Now, running this program again through the `strace` command and understand why it hangs. You can guess it without reading the source code!

Now add what the program was waiting for, and now see your program proceed to another bug, failing with a segmentation fault.

Using ltrace

Try to run the program through `ltrace`. You will see that another library is required to run this utility. Find this library in the toolchain and add it to your root filesystem again.

Now, `ltrace` should run fine and you should see what the program does: it tries to consume as much system memory as it can!

Using gdbserver

We are now going to use `gdbserver` to understand why the program segfaults.

Compile `vista-emulator.c` again with the `-g` option to include debugging symbols. Keep this binary on your workstation, and make a copy in the `/root` directory of the target root filesystem. Then, strip the binary on the target to remove the debugging symbols. They are only needed on your host, where the cross-debugger will run:

```
arm-linux-strip path/to/root/filesystem/root/vista-emulator
```

Then, on the target side, run `vista-emulator` under `gdbserver`. `gdbserver` will listen on a TCP port for a connection from GDB, and will control the execution of `vista-emulator` according to the GDB commands:

```
gdbserver localhost:2345 vista-emulator
```

On the host side, run `arm-linux-gdb` (also found in your toolchain):

```
arm-linux-gdb vista-emulator
```

You can also start the debugger through the `ddd` interface:

```
ddd --debugger arm-linux-gdb vista-emulator
```

GDB starts and loads the debugging information from the `vista-emulator` binary that has been compiled with `-g`.

Then, we need to tell where to find our libraries, since they are not present in the default `/lib` and `/usr/lib` directories on your workstation. This is done by setting GDB `sysroot` variable:

```
(gdb) set sysroot /usr/local/xtools/arm-unknown-linux-
uclibcgnueabi/arm-unknown-linux-uclibcgnueabi/sys-root/
```

And tell `gdb` to connect to the remote system:

```
(gdb) target remote <target-ip-address>:2345
```



Then, use `gdb` as usual to set breakpoints, look at the source code, run the application step by step, etc. Graphical versions of `gdb`, such as `ddd` can also be used in the same way. In our case, we'll just start the program and wait for it to hit the segmentation fault:

```
(gdb) continue
```

You could then ask for a backtrace to see where this happened:

```
(gdb) backtrace
```

This will tell you that the segmentation fault occurred in a function of the C library, called by our program. This should help you in finding the bug in our application.

What to remember

During this lab, we learned that...

- Compiling an application for the target system is very similar to compiling an application for the host, except that the cross-compilation introduces a few complexities when libraries are used.
- It's easy to study the behavior of programs and diagnose issues without even having the source code, thanks to `strace` and `ltrace`.
- You can leave a small `gdbserver` program (300 KB) on your target that allows to debug target applications, using a standard GDB debugger on the development host.
- It is fine to strip applications and binaries on the target machine, as long as the programs and libraries with debugging symbols are available on the development host.