Cross-Compiling tutorial with Scratchbox

Veli Mankinen

veli.mankinen@movial.fi

Valtteri Rahkonen

valtteri.rahkonen@movial.fi

Cross-Compiling tutorial with Scratchbox

by Veli Mankinen

by Valtteri Rahkonen

Revision history

Version:	Author:	Description:
2005-04-18	Rahkonen	Exported to docbook, modified to use Scratchbox 1.0
2004-12-08	Mankinen	Original article

Table of Contents

1. Introduction **1**

1.1. What is cross-compiling? 1

1.2. Portability of software 2

2. Tools and devices **4**

2.1. Installing the Scratchbox 4

2.2. Down to business 5

2.3. Cross-compiling in Scratchbox 7

2.4. A more complicated example 9

3. Conclusion **12**

References **13**

Chapter 1. Introduction

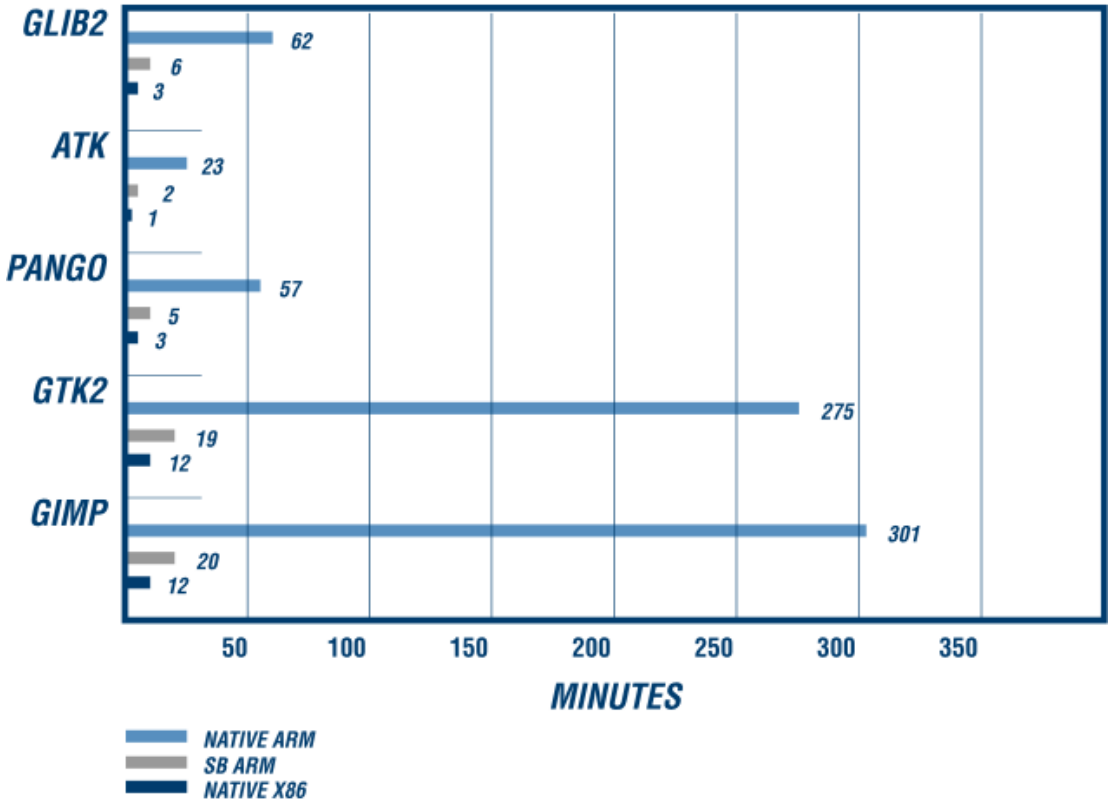
Cross-compiling is a relatively new thing if you compare it to the history of computers and software. In the early years of computing cross-compiling was not really needed as the software was written for a specific machine and purpose. Using the same code again and again was not so much of a thing.

After realization of the facts that some processors have a lot more computing power than others and we most certainly want to use the same code in many machines and for various purposes, it was easy to see that "in many cases compiling the application on some different machine would take a lot less time than compiling it on a machine where it is to be used". Idea of cross-compiling was born.

1.1. What is cross-compiling?

The basic idea of cross-compiling is to use some processor (HOST) to compile software for some other processor (TARGET) that uses different architecture. This means that the machine on which you are compiling the software cannot natively run the software it compiles. The software is compiled for another processor. This is one of the first challenges of cross-compiling. Many build environments want to run some programs during the compiling process which then of course crash the build process.

To avoid the challenges of cross-compiling some people prefer compiling natively. Anyhow in many cases this is very very slow and for that reason cannot be even considered as an option. Below is a graph that shows the compiling time differences between cross-compiling and native compiling.



The native ARM machine used in comparing was:

- Intel SA110 CPU 233MHz
- 40GB IDE HDD
- 256MB RAM
- Debian Sarge with Linux kernel 2.4.25

and the X86 machine used was:

- Intel Xeon CPU 2.80GHz
- 80GB IDE HDD
- 2GB RAM
- Debian Sarge with Linux kernel 2.6.8-1-686
- Scratchbox 1.0.1

1.2. Portability of software

Usually many coders tend to keep portability in mind when they start making software and in general

that is a very good thing. There are even lot of tools that help in making programs more portable. Many of those tools are used in the compile environment to find out information about the machine you are compiling the software on in order to use the information to set up variables. These tools make it possible to compile the program in the current environment more easily. This is again a good thing, unless you are cross-compiling; in that case the tools find out information about the system you are building the software on, and not about the system you are compiling the software for!

Chapter 2. Tools and devices

The normal use case of cross-compiling is when someone wants to compile programs for a PDA or some other small machine that has a relatively slow main processor. In our article we concentrate on building software for an IPAQ on a fast X86 machine.

To get most of this article you should most certainly have an IPAQ or other machine with ARM processor, which runs some Linux distribution. If you have an IPAQ and you want to start using Linux on that and develop or just compile some programs for that, it is suggested that you use Familiar Linux distribution (see [1]). They have very nice installing manuals on their website.

In our article we use cross-compiling tools from a project called Scratchbox [2]. Scratchbox is a configuration and compilation environment for building Linux software and entire Linux distributions. The basic idea of Scratchbox is to offer developers an environment that works and looks like the target environment before the target environment is available.

The Scratchbox is an environment that you log into like you would log into some machine. However, the Scratchbox tools can be used either inside or outside of Scratchbox. Using the tools outside is the same as using any tools that your host machine already has. That is also the normal, or even old fashioned way of cross-compiling.

2.1. Installing the Scratchbox

The current stable release of Scratchbox is 1.0.1. You can get the needed packages from. The following packages are needed from the download page: scratchbox-core, scratchbox-libs and scratchbox-toolchain-arm-glibc. All of those packages are provided as binary tar.gz, deb and rpm form. In this article we use .tar.gz files. The Scratchbox installation we are going to use takes about 522 mb of hard disk space and it will install under ’/scratchbox/’. You will also need some extra space to work in. If you don’t have enough space on your root partition then you can create a symbolic link to some other partition before continuing, e.g.:

```
> mkdir /work/scratchbox
> ln -s /work/scratchbox /scratchbox
```

Scratchbox needs to be installed as root:

```
> cd /tmp/
> wget http://www.scratchbox.org/download/files/sbox-releases/
```

```
1.0/tarball/scratchbox-core-1.0.1-i386.tar.gz
> wget http://www.scratchbox.org/download/files/sbox-releases/
1.0/tarball/scratchbox-libs-1.0.1-i386.tar.gz
> wget http://www.scratchbox.org/download/files/sbox-releases/
1.0/tarball/scratchbox-toolchain-arm-gcc3.3-glibc2.3-1.0.1-i386.tar.gz
> tar -xzf scratchbox-core-1.0.1-i386.tar.gz -C /
> tar -xzf scratchbox-libs-1.0.1-i386.tar.gz -C /
> tar -xzf scratchbox-toolchain-arm-gcc3.3-glibc2.3-1.0.1-i386.tar.gz
-C /
> /scratchbox/run_me_first.sh
```

’run_me_first.sh’ asks you some questions, just use defaults. After this we need to add user for Scratchbox. User must be some user that you have in your system. Do NOT add root user! e.g.:

```
> /scratchbox/sbin/sbox_adduser vmankine
```

Scratcbox installation document [3] chapter number two helps you in installing the packages if you want to use for example .deb packages.

2.2. Down to business

Let us first use the old fashioned way of cross-compiling and use the Scratchbox tools from the host system like any other tools. First we need a small piece of software to compile. "Hello World" will do fine:

```
#include <stdio.h>

int main(void) {
printf("Hello World!\n");

return 0;
}
```

First let us compile the "Hello World" natively for X86 to see it works and to see some information about the binary produced. We assume you wrote the "Hello World" to a file named ’hello.c’. Now in the same directory where you have the ’hello.c’ execute a command:

```
> gcc -Wall -o hello hello.c
```

It should not output anything and then you should have a ’hello’ binary that you can run to get output of "Hello World!".

```
> ./hello
Hello World!
```

Now for running a ’file’ command to see some information about the ’hello’ binary:

```
> file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.2.0, dynamically linked (uses shared libs), not stripped
```

Any of this does not yet have anything to do with Scratchbox or cross-compiling for that matter. For cross-compiling we need to use the cross-compiler provided in the Scratchbox packages. Run command:

```
> /scratchbox/compilers/arm-gcc-3.3.4-glibc-2.3.2/bin/arm-linux-gcc
-Wall -o arm-hello hello.c
```

Running that command should not give you any output but now you should have an ’arm-hello’ binary. You cannot run that as it is for ARM architecture and you are on a X86 machine. Anyhow we can run the ’file’ command to see that it really is an ARM binary:

```
> file arm-hello
arm-hello: ELF 32-bit LSB executable, ARM, version 1 (ARM), for
GNU/Linux 2.0.0, dynamically linked (uses shared libs), not stripped
```

Now we have cross-compiled our first application! If you have your IPAQ or equivalent ARM device running Linux you can of course try the application on that one. Just transfer the ’arm-hello’ to IPAQ with e.g. scp and run it there normally.

If you are compiling small pieces of software like this you really don’t need anything else than the cross-compiler to get everything working. However, many programs that you can find from the open source pool use many libraries and complicated building systems. This means that you have to compile those libraries also and not just the programs. You also most certainly have to tweak the building environments to be able to cross-compile. The tweaking of the build systems is often a very hard and time-consuming task. For that reason we use the Scratchbox environment in the following examples as the Scratchbox environment renders most of the build system tweaking unnecessary.

2.3. Cross-compiling in Scratchbox

Scratchbox is a chrooted cross-compilation environment. This means one has to log into the Scratchbox like one would log into a some real machine. Anyhow normal users must be added as Scratchbox users before they can login to Scratchbox. You should already have user added for Scratchbox as we did it during the installation of Scratchbox. Please note that from now on you should use that user account that use added to Scratchbox.

You can log into the Scratchbox system with a command: ’/scratchbox/login’. You should get some welcome messages and your prompt should change to something like this: ’[sbox-HOST: ~] > ’.

```
[vmankine@host ~] > /scratchbox/login

Welcome to Scratchbox, the cross-compilation toolkit!
...

[sbox-HOST: ~] >
```

Please note that If you were logged into the Scratchbox machine before you were added as a Scratchbox user, you may need to re-login to your machine, so that you get ’sbox’ group privileges needed for running Scratchbox. You can check this by running the following command:

```
> groups
```

The Scratchbox prompt ’[sbox-HOST: ~] > ’ tells you that you are in Scratchbox environment and that you are using target called HOST. In Scratchbox you can have multiple targets which means that you can compile programs for many different architectures and settings. The HOST target is used for compiling programs that are used inside Scratchbox, something we don’t need to worry in this article. Please notice that inside Scratchbox your root directory (/) is not the same as outside and you even have a separate home directory.

After logging into Scratchbox we need to create a target for our cross-compilation needs. Chapter 2.4 in the Scratchbox Installation document explains more deeply how to create a targets in Scratchbox. Now we just create and activate a target by typing in commands:

```
[sbox-HOST: ~] > sb-conf setup MYTARGET -c arm-gcc-3.3.4-glibc-2.3.2
-t qemu-arm
[sbox-HOST: ~] > sb-conf select MYTARGET
[sbox-MYTARGET: ~] >
```

In the Scratchbox environment in a new target the root directory (/) contains a lot of the normal directories (/bin, /sbin, /usr, /var etc.) which however are completely empty. This makes it possible for us to install any given software to the place where we want it on the target without worrying about over writing some host machine programs. As all the directories are empty it means that we don’t even have a c-library in our target which is basically needed by every piece of software. We also have to somehow install or compile everything we want to use inside the target. Luckily Scratchbox toolchains come with a c-library which we can just install to the target by typing in a command:

```
[sbox-MYTARGET: ~] > sb-conf install -c
```

Now we have a target ready for compiling for ARM. Scratchbox comes with ARM emulator QEMU (see [4]) that we use with our target in these examples. Scratchbox needs a device or emulator that can run the target architecture programs. By being able to run the target architecture binaries we can have a cross-compilation environment where we don’t have to know everything about the target system and somehow tweak that information into the build environment but we can let the tools find out the information like in the native compilation.

If we now write the "Hello World" program again and then compile it with command:

```
[sbox-MYTARGET: ~] > gcc -Wall -o sb-arm-hello hello.c
```

it again should compile without errors and as a result we should have a ’sb-arm-hello’ binary. If we run the ’file’ command for the ’sb-arm-hello’ we will get output that shows we have a real ARM binary in our hands again:

```
[sbox-MYTARGET: ~] > file sb-arm-hello
sb-arm-hello: ELF 32-bit LSB executable, ARM, version 1 (ARM), for
```

```
GNU/Linux 2.0.0, dynamically linked (uses shared libs), not stripped
```

But as we are inside the Scratchbox environment we can also run the program although it is an ARM binary and we are on X86 machine. This was because we have the emulator:

```
[sbox-MYTARGET: ~] > ./sb-arm-hello
Hello World!
```

2.4. A more complicated example

For our more complicated program we need a glib library from the GTK+ family. First let us just download the glib package and compile it for the target with following commands:

```
[sbox-MYTARGET: ~] > wget ftp://ftp.gtk.org/pub/gtk/v2.2/glib-2.2.3.tar.bz2
[sbox-MYTARGET: ~] > tar -xjvf glib-2.2.3.tar.bz2
[sbox-MYTARGET: ~] > cd glib-2.2.3
[sbox-MYTARGET: ~/glib-2.2.3] > ./configure
[sbox-MYTARGET: ~/glib-2.2.3] > make
[sbox-MYTARGET: ~/glib-2.2.3] > make install
```

As you can see this already was a lot more complicated example so far. We used ’configure’ script which compiles and runs a lot of different kinds of programs to find out things from the environment. If you would have been cross-compiling outside Scratchbox the ’configure’ script would have crashed as it would not have been able to run the binaries it compiles. If you now take a look at a file ’/tmp/cputransp_$USER.log’ you can see quite some lines which show you commands run with the emulator. Of course glib can be compiled by tweaking the environment also. It is a bit more complicated than what we just did. A document about cross-compiling glib can be found at [5].

Now we have c-library and glib library installed in our target and we can move on to compiling our new "Hello World" that uses glib print function ’glib-hello.c’:

```
#include <glib.h>
#include <glib/gprintf.h>

int main(void)
```

```
{
g_printf("Hello World!\n");

return 0;
}
```

Scratchbox comes with ’pkg-config’ tool which is used to get all the include and library paths for the compilations without worrying too much. By default glib installs itself to ’/usr/local/’ and it also installs the ’pkg-config’ configuration files under that directory. As in this example we want to use ’pkg-config’ we have to set the ’PKG_CONFIG_PATH’ environment variable so that the tool will find the glib files. After that you can also run: ’pkg-config --list-all’ to see which libraries have been installed to your system:

> **Note:** If we would have set the default install prefix for glib to ’/usr’ with --prefix configure option then we would not need to set the PKG_CONFIG_PATH environment variable.

```
[sbox-MYTARGET: ~] > export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig/
[sbox-MYTARGET: ~] > pkg-config --list-all
gmodule-2.0 GModule - Dynamic module loader for GLib
glib-2.0    GLib - C Utility Library
gobject-2.0 GObject - GLib Type, Object, Parameter and Signal Library
gthread-2.0 GThread - Thread support for GLib
```

For compiling the new glib "Hello World" we need to type in following command:

```
[sbox-MYTARGET: ~] > gcc -Wall ‘pkg-config --cflags --libs glib-2.0‘
-o sb-arm-glib-hello glib-hello.c
```

The compilation should again finish without outputting anything and as a result you should have a ’sb-arm-glib-hello’ binary. You can now run the command because we have the QEMU emulator and it should output: "Hello World!". If we now run ’file’ command the output should look the same as with the previous "Hello World". Anyhow now if we run ’ldd’ command which shows the libraries the binary is using we can see that this binary is really using the glib library which we just installed:

```
[sbox-MYTARGET: ~] > ldd sb-arm-glib-hello
```

```
libglib-2.0.so.0 => /usr/local/lib/libglib-2.0.so.0 (0x00000000)
libc.so.6 => /lib/libc.so.6 (0x00000000)
libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x00000000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x00000000)
```

After this you could make binary packages of your new fancy software and libraries or just transfer those on an ARM device by some other means. Anyway all this is basically just scratching the surface. There are so many things you can do and study. Scratchbox also contains a lot of additional features and tools which can help you in developing and building software for example for Debian ARM. Head on to Scratchbox website for more information and more advanced use cases.

Chapter 3. Conclusion

There are a lot of fancy pieces of hardware out there that are already running Linux or which you can install Linux on. It can seem quite hard to make it all work or just to compile your favorite programs for the device. Anyhow with right tools and the help of the many articles and websites you can find from the Internet and magazines it is not too hard. At least it is a very interesting world to play in.

References

[1] *Handhelds (http://www.handhelds.org/)*.

[2] *Scratchbox website (http://scratchbox.org/)*.

[3] *Installing Scratchbox (http://www.scratchbox.org/documentation/docbook/installdoc.html)*.

[4] *QEMU (http://fabrice.bellard.free.fr/qemu/)*.

[5] *Cross-compiling the GLib package (http://developer.gnome.org/doc/API/2.0/glib/glib-cross-compiling.html)*.