A toolchain is the set of tools that compiles source code into executables that can run on your target device, and includes a compiler, a linker, and run-time libraries.

A standard GNU toolchain consists of three main components:

1. **Binutils**: A set of binary utilities including the assembler and the linker. It is

available at <http://www.gnu.org/software/binutils>.

1. **GNU Compiler Collection** (**GCC**): These are the compilers for C and

other languages which, depending on the version of GCC, include C++,

Objective-C, Objective-C++, Java, Fortran, Ada, and Go. They all use a

common backend which produces assembler code, which is fed to the GNU

assembler. It is available at <http://gcc.gnu.org/>.

1. **C library**: A standardized **application program interface** (**API**) based on

the POSIX specification, which is the main interface to the operating

system kernel for applications. There are several C libraries to consider,

Most people would consider the **GNU Debugger** (**GDB**) to be part of the toolchain as well

For our purposes, there are two types of toolchain:

* **Native**: This toolchain runs on the same type of system (sometimes the

same actual system) as the programs it generates. This is the usual case for

desktops and servers, and it is becoming popular on certain classes of

embedded devices. The Raspberry Pi running Debian for ARM, for

example, has self-hosted native compilers.

* **Cross**: This toolchain runs on a different type of system than the target,

allowing the development to be done on a fast desktop PC and then loaded

onto the embedded target for testing.

Almost all embedded Linux development is done using a cross development

toolchain, partly because most embedded devices are not well suited to program development since they lack computing power, memory, and storage, but also because it keeps the host and target environments separate. The latter point is especially important when the host and the target are using the same architecture, x86\_64, for example. In this case, it is tempting to compile natively on the host and simply copy the binaries to the target.

The toolchain has to be built according to the capabilities of the target CPU,which includes:

* **CPU architecture**: ARM, MIPS, x86\_64, and so on
* **Big- or little-endian operation**: Some CPUs can operate in both modes,

but the machine code is different for each

* **Floating point support**: Not all versions of embedded processors

implement a hardware floating point unit, in which case the toolchain has to

be configured to call a software floating point library instead

* **Application Binary Interface** (**ABI**): The calling convention used for passing parameters between function calls.

With many architectures, the ABI is constant across the family of processors One notable exception is ARM. The ARM architecture transitioned to the **Extended Application Binary Interface** (**EABI**) in the late 2000s, resulting in the previous ABI being named the **Old Application Binary Interface** (**OABI**). While the OABI is now obsolete, you continue to see references to EABI. Since then, the EABI has split into two, based on the way the floating point parameters are passed. The original EABI uses general purpose (integer) registers, while the newer **Extended Application Binary Interface Hard-Float** (**EABIHF**) uses floating point registers. The EABIHF is significantly faster at floating point operations, since it removes the need for copying between integer and floating point registers, but it is not compatible with CPUs that do not have a floating point unit. The choice, then, is between two incompatible ABIs; you cannot mix and match the two, and so you have to decide at this stage.

The C library is the implementation of that interface; it is the gateway to the kernel for Linux programs, as shown in the following diagram. Even if you are writing programs in another language, maybe Java or Python, the respective run-time support libraries will have to call the **C library** eventually, as shown here:

Whenever the C library needs the services of the kernel, it will use the kernel system call interface to transition between user space and kernel space. It is possible to bypass the C library by making the kernel system calls directly, but that is a lot of trouble and almost never necessary. There are several C libraries to choose from. The main options are as follows:

1 : glibc

2: musl libc

3: uClibc-ng

4: eglibc

**Building a toolchain for BeagleBone** **Black :**

Step 1 :

Before you begin, you will need a working native toolchain and build tools onyour host PC. To work with crosstool-NG on an Ubuntu host, you will need toinstall the packages using the following command:

**$ sudo apt-get install automake bison chrpath flex g++ git gperf gawk libexpat1-dev libncurses5-dev libsdl1.2-dev libtool python2.7-dev texinfo**

Step 2 :

Get the current release from the crosstool-NG Git repository.

**$ git clone** [**https://github.com/crosstool-ng/crosstool-ng.git**](https://github.com/crosstool-ng/crosstool-ng.git)

**$ cd crosstool-ng**

**$ ./bootstrap**

**$ ./configure --enable-local**

**$ make**

**$ make install**

Step 3 :

Crosstool-NG can build many different combinations of toolchains. To make the initial configuration easier, it comes with a set of samples that cover many of the common use-cases. Use **$ ./ct-ng list-samples** to generate the list.

Step 4 :

**$ ./ct-ng show-arm-cortex\_a8-linux-gnueabi**

**$ ./ct-ng arm-cortex\_a8-linux-gnueabi**

**Step 5 :**

There are two configuration changes that I would recommend you make at thispoint:

* In Paths and misc options, disable Render the toolchain read-only (CT\_INSTALL\_DIR\_RO)
* In Target options | Floating point, select hardware (FPU) (CT\_ARCH\_FLOAT\_HW)

Step 6 :

Now you can use crosstool-NG to get, configure, and build the components according to your specification, by typing the following command:

**$ ./ct-ng build**

The build will take about half an hour, after which you will find your toolchain is

present in

~/x-tools/arm-cortex\_a8-linux-gnueabihf.

Step 7 :