Documentation

1、Introduction

crosstool-NG aims at building toolchains. Toolchains are an essential component in a software development project. It will compile, assemble and link the code that is being developed. Some pieces of the toolchain will eventually end up in the resulting binaries: static libraries are but an example.

So, a toolchain is a very sensitive piece of software, as any bug in one of the components, or a poorly configured component, can lead to execution problems, ranging from poor performance, to applications ending unexpectedly, to misbehaving software (which more than often is hard to detect), to hardware damage, or even to human risks (which is more than regrettable).

Toolchains are made of different piece of software, each being quite complex and requiring specially crafted options to build and work seamlessly. This is usually not that easy, even in the not-so-trivial case of native toolchains. The work reaches a higher degree of complexity when it comes to cross-compilation, where it can become quite a nightmare …

Some cross-toolchains exist on the internet, and can be used for general development, but they have a number of limitations:

They can be general purpose, in that they are configured for the majority: no optimisation for your specific target.

They can be prepared for a specific target and thus are not easy to use, nor optimised for, or even supporting your target.

They often are using aging components (compiler, C library, etc.) not supporting special features of your shiny new processor.

On the other side, these toolchain offer some advantages:

They are ready to use and quite easy to install and setup.

They are proven if used by a wide community.

But once you want to get all the juice out of your specific hardware, you will want to build your own toolchain. This is where crosstool-NG comes into play.

There are also a number of tools that build toolchains for specific needs, which are not really scalable. Examples are:

buildroot, whose main purpose is to build root file systems, hence the name. But once you have your toolchain with buildroot, part of it is installed in the root-to-be, so if you want to build a whole new root, you either have to save the existing one as a template and restore it later, or restart again from scratch. This is not convenient.

ptxdist, whose purpose is very similar to buildroot.

Other projects (e.g., openembedded.org), which are again used to build root file systems.

crosstool-NG is really targeted at building toolchains, and only toolchains. It is then up to you to use it the way you want.
History

crosstool was first conceived by Dan Kegel, who offered it to the community as a set of scripts, a repository of patches, and some pre-configured, general purpose setup files to be used to configure crosstool. This is available at kegel.com/crosstool, and the subversion repository is hosted on Google Code.

Yann E. Morin once managed to add support for uClibc-based toolchains, but it did not make it into mainline, mostly because Yann didn’t have time to port the patch forward to the new versions, due in part to the big effort it was taking.

So Yann decided to clean up crosstool in the state it was, re-order the things in place, add appropriate support for what he needed, that is uClibc support and a menu-driven configuration, named the new implementation crosstool-NG, (standing for crosstool Next Generation, as many other community projects do, and as a wink at the TV series “Star Trek: The Next Generation” ;-)) and made it available to the community, in case it was of interest to any one.

In late 2014, Yann became very busy with buildroot and other projects, and so Bryan Hundven opted to become the new maintainer for crosstool-NG.

Referring to crosstool-NG

The long name of the project is crosstool-NG:

no leading uppercase (except as first word in a sentence),

crosstool and NG separated with a hyphen (dash),

NG in uppercase.

Crosstool-NG can also be referred to by its short name CT-NG:

all in uppercase,

CT and NG separated with a hyphen (dash).

The long name is preferred over the short name, except in mail subjects, where the short name is a better fit.

When referring to a specific version of crosstool-NG, append the version number either as:

crosstool-NG X.Y.Z (long name, a space, and the version string)

crosstool-ng-X.Y.Z (long name in lowercase, a hyphen (dash), and the version string) – this is used to name the release tarballs

crosstool-ng-X.Y.Z+git_id (long name in lowercase, a hyphen, the version string, and the Git ID as returned by ct-ng version) – this is used to differentiate between releases and snapshots

The frontend to crosstool-NG is the command ct-ng:

all in lowercase,

ct and ng separated by a hyphen (dash).

2、Installing crosstool-NG

Before installing crosstool-NG, you may need to install additional packages on the host OS. Specific instructions for several supported operating systems and distributions are provided here. Note that not all the dependencies are currently detected by the configure script; missing some of them may later result in failing ct-ng build.

There two ways how you can obtain the crosstool-NG sources:

by downloading a released tarball;

or by cloning the current development repository.

There also are two ways you can use crosstool-NG:

build and install it, then get rid of the sources like you’d do for most programs;

or only build it and run from the source directory.

The typical workflow assumes using a released tarball and installing crosstool-NG. If you intend to do some development on crosstool-NG and/or submit patches, you’d likely want a clone of the repository and running from the source directory.

Downloading a released tarball

First, download the tarball:

```
wget
http://crosstool-ng.org/download/crosstool-ng/crosstool-ng-VERSION.tar.bz2
```

or

```
wget
http://crosstool-ng.org/download/crosstool-ng/crosstool-ng-VERSION.tar.xz
```

Starting with 1.21.0, releases are signed with Bryan Hundven’s PGP key. The fingerprint is:

561E D9B6 2095 88ED 23C6 8329 CAD7 C8FC 35B8 71D1

Starting with 1.23.0, releases are signed with Alexey Neyman’s PGP key. The fingerprint is:

64AA FBF2 1475 8C63 4093 45F9 7848 649B 11D6 18A4

The public keys are found on http://pgp.surfnet.nl/. To validate the release tarball run you need to import the keys from the keyserver and download the signature of the tarball, then verify the tarball with both the tarball and the signature in the same directory:

```
gpg --keyserver http://pgp.surfnet.nl --recv-keys 35B871D1 11D618A4
wget
http://crosstool-ng.org/download/crosstool-ng/crosstool-ng-VERSION.tar.bz2.sig
```

```
gpg --verify crosstool-ng-VERSION.tar.bz2.sig
```

Crosstool-NG releases 1.19.0 and earlier provide MD5/SHA1/SHA512 digests for the tarballs. Use md5sum/sha1sum/sha512sum commands to verify the tarballs:

```
md5sum -c crosstool-ng-VERSION.tar.bz2.md5
sha1sum -c crosstool-ng-VERSION.tar.bz2.sha1
sha512sum -c crosstool-ng-VERSION.tar.bz2.sha512
```

Cloning a repository

If the released version is not recent enough for your purposes, you can try to build using the currently developed version. To do that, clone the Git repository:

```
git clone https://github.com/crosstool-ng/crosstool-ng
```

You’ll need to run the bootstrap script before running configure:

```
./bootstrap
```

Install method

First unpack the tarball and cd into the crosstool-ng-VERSION directory.

Note

Due to a bug in release scripts, version 1.22.0 of the crosstool-ng was packaged without the VERSION suffix.

Then follow the classical configure recipe: ./configure way:

```
./configure --prefix=/some/place
make
make install
export PATH="${PATH}:/some/place/bin"
```

You can then get rid of crosstool-NG source. Next create a directory to serve as a working place, cd in there and run:

```
mkdir work-dir
cd work-dir
ct-ng help
```

Note

If you call ct-ng --help you will get help for make(1). This is because ct-ng is in fact a make(1) script. There is no clean workaround for this.

A man page for the ct-ng utility is also installed. You can get some brief help by typing man ct-ng.

The Hacker’s way

Then, you run ./configure for local execution of crosstool-NG:

```
./configure --enable-local
make
```

Now, do not remove crosstool-NG sources. They are needed to run crosstool-NG! Stay in the directory holding the sources, and run:

```
./ct-ng help
```

Preparing for packaging

If you plan on packaging crosstool-NG, you surely don’t want to install it in your root file system. The install procedure of crosstool-NG honors the DESTDIR variable:

```
./configure --prefix=/usr
make
make DESTDIR=/packaging/place install
```

Shell completion

crosstool-NG comes with a shell script fragment that defines bash-compatible completion. That shell fragment is currently not installed automatically.

To install the shell script fragment, you have two options:

- install system-wide, most probably by copying ct-ng.comp into /etc/bash_completion.d/, or

- install for a single user, by copying ct-ng.comp into ${HOME}/ and sourcing this file from your ${HOME}/.bashrc.

Contributed code

Some people contributed code that couldn’t get merged for various reasons. This code is available as lzma-compressed patches, in the contrib/ sub-directory. These patches are to be applied to the source of crosstool-NG, prior to installing, using something like the following:

```
lzcat contrib/foobar.patch.lzma | patch -p1
```

There is no guarantee that a particular contribution applies to the current version of crosstool-ng, or that it will work at all. Use contributions at your own risk.

3、Configuring crosstool-NG

crosstool-NG is configured with a configurator presenting a menu-structured set of options. These options let you specify the way you want your toolchain built, where you want it installed, what architecture and specific processor it will support, the version of the components you want to use, etc. The value for those options are then stored in a configuration file.

The configurator works the same way you configure your Linux kernel. It is assumed you know how to handle this.

To enter the menu, type:

```
ct-ng menuconfig
```

Almost every config item has a help entry. Read them carefully.

String and number options can refer to environment variables. In such a case, you must use the shell syntax: ${VAR}. You shall neither single- nor double-quote the string/number options.

There are three environment variables that are computed by crosstool-NG, and that you can use:

CT_TARGET: Represents the target tuple you are building for. You can use it for example in the installation/prefix directory, such as: /opt/x-tools/${CT_TARGET}. If needed, parts of CT_TARGET are also available as CT_TARGET_ARCH, CT_TARGET_VENDOR, CT_TARGET_KERNEL and CT_TARGET_SYS.

CT_TOP_DIR: The top directory where crosstool-NG is running. You shouldn’t need it in most cases. One case where you may need it is if you have local patches or config files and you store them in your current working directory, you can refer to them by using CT_TOP_DIR, such as: ${CT_TOP_DIR}/patches.myproject.

CT_VERSION: The version of crosstool-NG you are using. Not much use for you, but it’s there if you need it.

You can also refer to the config variables recursively, but take care to avoid circular dependencies or nesting the references too deep (crosstool-NG currently only follows them to a depth of 10).

Sample configurations

Crosstool-NG ships with several sample configurations (pre-configured toolchains that are known to build and work). Sample names are 1- to 4-part tuples. To get the list of these samples and see more detailed information on any sample, do (replace “arm-unknown-linux-gnueabi” with the sample name you want to view):

```
ct-ng list-samples
ct-ng show-arm-unknown-linux-gnueabi
```

Once you chose one sample as a starting point, load it as a base and fine-tune using ct-ng menuconfig as described above:

```
ct-ng arm-unknown-linux-gnueabi
ct-ng menuconfig
```

Interesting config options

CT_LOCAL_TARBALLS_DIR: If you already have some tarballs in a directory, enter it here. That will speed up the retrieving phase, where crosstool-NG would otherwise download those tarballs.

CT_PREFIX_DIR: This is where the toolchain will be installed in (and for now, where it will run from). Common use is to add the target tuple in the directory path, such as (see above): /opt/x-tools/${CT_TARGET}.

CT_TARGET_VENDOR: An identifier for your toolchain, will take place in the vendor part of the target tuple. It shall not contain spaces or dashes. Usually, keep it to a one-word string, or use underscores to separate words if you need. Avoid dots, commas, and special characters. It can be set to empty, to remove the vendor string from the target tuple.

CT_TARGET_ALIAS: An alias for the toolchain. It will be used as a prefix to the toolchain tools. For example, you will have ${CT_TARGET_ALIAS}-gcc.

Also, if you think you don’t see enough versions, you can try to enable one of those:

CT_OBSOLETE: Show obsolete versions of tools. Most of the time, you don’t want to base your toolchain on too old a version (of gcc, for example). But at times, it can come handy to use such an old version for regression tests or to support some outdated system configuration. Those old versions are hidden behind CT_OBSOLETE. Those versions (or features) are so marked because maintaining support for those in crosstool-NG would be too costly, time-wise, and time is dear. Note that these versions are likely going to disappear in the next crosstool-NG release.

CT_EXPERIMENTAL: Show experimental versions of tools and crosstool-NG features. This may enable using unreleased versions of the tools, or configure the toolchain in a way that is not thoroughly tested. Use with care.

Re-building an existing toolchain

If you have an existing toolchain, you can re-use the options used to build it to create a new toolchain. That needs a very little bit of effort on your side but is quite easy. The options to build a toolchain are saved with the toolchain, and you can retrieve this configuration by running:

```
${CT_TARGET}-ct-ng.config
```

An alternate method is to extract the configuration from a build.log file. This will be necessary if your toolchain was build with crosstool-NG prior to 1.4.0, but can be used with build.log files from any version:

```
ct-ng extractconfig <build.log >.config
```

Or, if your build.log file is compressed (most probably!):

```
bzcat build.log.bz2 | ct-ng extractconfig >.config
```

The above commands will dump the configuration to stdout, so to rebuild a toolchain with this configuration, just redirect the output to the .config file:

```
${CT_TARGET}-ct-ng.config >.config
ct-ng oldconfig
```

Then, you can review and change the configuration by running:

```
ct-ng menuconfig
```

Note

The same procedure applies to rebuilding the toolchain with a newer crosstool-NG version. Perform ct-ng oldconfig prior to building the toolchain! Otherwise, the options introduced in the new release will not be set to their default values, and this will result in errors later on! If you cloned a Git repository, you need to do ct-ng oldconfig each time you do a git pull.

4、Building the Toolchain

To build the toolchain, simply type:

```
ct-ng build
```

This will use the above configuration to retrieve, extract and patch the components, build, install and eventually test your newly built toolchain. If all goes well, you should see something like this:

```
[INFO ]   Performing some trivial sanity checks
[INFO ]   Build started 20170319.002217
[INFO ]   Building environment variables
[EXTRA]   Preparing working directories
[EXTRA]   Installing user-supplied crosstool-NG configuration
[EXTRA]
=================================================================
[EXTRA]   Dumping internal crosstool-NG configuration
[EXTRA]     Building a toolchain for:
[EXTRA]       build   = x86_64-pc-linux-gnu
[EXTRA]       host    = x86_64-pc-linux-gnu
[EXTRA]       target = mipsel-sde-elf
[EXTRA]   Dumping internal crosstool-NG configuration: done in 0.05s (at
00:02)
[INFO                                                              ]
=================================================================
[INFO ]   Retrieving needed toolchain components' tarballs
[EXTRA]     Retrieving 'gmp-6.1.2'
[EXTRA]     Saving 'gmp-6.1.2.tar.xz' to local storage
[EXTRA]     Retrieving 'mpfr-3.1.5'
[EXTRA]     Saving 'mpfr-3.1.5.tar.xz' to local storage
...
[INFO ]   Installing cross-gdb
[EXTRA]     Configuring cross-gdb
[EXTRA]     Building cross-gdb
[EXTRA]     Installing cross-gdb
[EXTRA]     Installing '.gdbinit' template
[INFO ]   Installing cross-gdb: done in 98.55s (at 10:51)
[INFO                                                              ]
=================================================================
[INFO ]   Cleaning-up the toolchain's directory
[INFO ]     Stripping all toolchain executables
[EXTRA]     Creating toolchain aliases
```

```
[EXTRA]      Removing access to the build system tools
[EXTRA]      Removing installed documentation
[INFO ]   Cleaning-up the toolchain's directory: done in 0.42s (at 10:52)
[INFO ]   Build completed at 20170319.003309
[INFO ]   (elapsed: 10:51.42)
[INFO ]   Finishing installation (may take a few seconds)...
```

You are then free to add the toolchain’s /bin directory in your PATH to use it at will.
Stopping and restarting a build

If you want to stop the build after a step you are debugging, you can pass the variable STOP to make:

```
ct-ng build STOP=some_step
```

Conversely, if you want to restart a build at a specific step you are debugging, you can pass the RESTART variable to make:

```
ct-ng build RESTART=some_step
```

Alternatively, you can call make with the name of a step to just do that step:

```
ct-ng libc_headers
```

which is equivalent to:

```
ct-ng build RESTART=libc_headers STOP=libc_headers
```

The shortcuts +step_name and step_name+ allow to respectively stop or restart at that step. Thus

```
ct-ng +libc_headers
```

is equivalent to

```
ct-ng build STOP=libc_headers
```

and

```
ct-ng libc_headers+
```

is equivalent to:

```
ct-ng build RESTART=libc_headers
```

To obtain the list of acceptable steps, please call:

```
ct-ng list-steps
```

Note that in order to restart a build, you'll have to say Y to the config option CT_DEBUG_CT_SAVE_STEPS, and that the previous build effectively went that far.
Overriding the number of jobs

If you want to override the number of jobs to run in (the -j option to make), you can either re-enter the menuconfig, or simply add it on the command line, as such:

```
ct-ng build.4
```

which tells crosstool-NG to override the number of jobs to 4.

You can see the actions that support overriding the number of jobs in the help menu. Those are the ones with [.#] after them (e.g., build[.#] or build-all[.#], and so on).

Note

The crosstool-NG script ct-ng is a Makefile-script. It does not execute in parallel (there is not much to gain). When speaking of jobs, we are refering to the number of jobs when making the components. That is, we speak of the number of jobs used to build gcc, glibc, and so on.

Building all toolchains at once

You can build all samples; simply call:

```
ct-ng build-all
```

Note that it is very time consuming (depending on your machine configuration and host OS, it takes from 24 hours to a full week). By default, this removes each build tree after a successful build, but leaves the unpacked/patched sources so that they can be re-used by the samples that follow). However, even that consumes considerable amount of disk space given the variety of component versions represented in samples.

5、Using the toolchain

Using the toolchain is as simple as adding the toolchain's bin directory in your PATH, such as:

```
export PATH="${PATH}:/your/toolchain/path/bin"
```

Depending on the project being compiled, there may be different ways to specify the toolchain.

If the software uses GNU autotools or a similar configure script, you shoul use the --host tuple to tell the build system to use your toolchain (if the software package uses the autotools system you should also pass --build, for completeness):

```
./configure --host=your-host-tuple --build=your-build-tuple
```

Other systems may need, for example:

```
make CC=your-host-tuple-gcc
make CROSS_COMPILE=your-host-tuple-
make CHOST=your-host-tuple
```

and so on. Please read the documentation of the package being compiled as to how it can be cross-compiled.

Note

in the above example, host refers to the host of your program (i.e. the target of the toolchain), not the host of the toolchain; and build refers to the machine where you build your program, that is the host of the toolchain.

Assembling a root filesystem

Assembling a root filesystem for a target device requires the successive building of a set of software packages for the target architecture. Building a package potentially requires artifacts which were generated as part of an earlier build. Note that not all artifacts which are installed as part of a package are desirable on a target’s root filesystem (e.g., man/info files, include files, etc.). Therefore we must distinguish between a staging directory and a rootfs directory.

A staging directory is a location into which we install all the build artifacts. We can then point future builds to this location so they can find the appropriate header and library files. A rootfs directory is a location into which we place only the files we want to have on our target.

There are four schools of thought here:

Option 1: Install directly into the sysroot of the toolchain.

By default (i.e., if you don’t pass any arguments to the tools which would change this behaviour) the toolchain that is built by crosstool-NG will only look in its toolchain directories for system header and library files:

```
#include "..." search starts here:
#include <...> search starts here:
<ct-ng install path>/lib/gcc/<host tuple>/4.5.2/include
<ct-ng install path>/lib/gcc/<host tuple>/4.5.2/include-fixed
<ct-ng install path>/lib/gcc/<host tuple>/4.5.2/../../../../<host tuple>/include
<ct-ng install path>/<host tuple>/sysroot/usr/include
```

In other words, the compiler will automagically find headers and libraries without extra flags if they are installed under the toolchain’s sysroot directory.

However, this is bad because the toolchain gets poluted, and can not be re-used.

```
$ ./configure --build=<build tuple> --host=<host tuple> \
        --prefix=/usr --enable-foo-bar...
$ make
$ make DESTDIR=/<ct-ng install path>/<host tuple>/sysroot install
```

Option 2: Copy the toolchain’s sysroot to the staging area.

If you start off by copying the toolchain’s sysroot directory to your staging area, you can simply proceed to install all your packages’ artifacts to the same staging area. You then only need to specify a --sysroot=<staging area> option to the compiler of any subsequent builds and all your required header and library files will be found/used.

This is a viable option, but requires the user to always specify CFLAGS in order to include --sysroot=<staging area>, or requires the use of a wrapper to a few select tools (gcc, ld…) to pass this flag.

Instead of polluting the toolchain’s sysroot you are copying its contents to a new location and polluting the contents in that new location. By specifying the --sysroot option you’re effectively abandoning the default sysroot in favour of your own.

Incidentally this is what buildroot does using a wrapper, when using an external toolchain.

```
$ cp -a $(<host tuple>-gcc --your-cflags-except-sysroot -print-sysroot) \
        /path/to/staging
$ ./configure --build=<build tuple> --host=<host tuple>             \
                    --prefix=/usr --enable-foo-bar...                      \
                    CC="<host tuple>-gcc --syroot=/path/to/staging"        \
                    CXX="<host tuple>-g++ --sysroot=/path/to/staging"      \
                    LD="<host tuple>-ld --sysroot=/path/to/staging"        \
                    AND_SO_ON="tuple-andsoon --sysroot=/path/to/staging"
$ make
$ make DESTDIR=/path/to/staging install
```

Option 3: Use separate staging and sysroot directories.

In this scenario you use a staging area to install programs, but you do not pre-fill that staging area with the toolchain’s sysroot. In this case the compiler will find the system includes and libraries in its sysroot area but you have to pass appropriate CPPFLAGS and LDFLAGS to tell it where to find your headers and libraries from your staging area (or use a wrapper).

```
$ ./configure --build=<build tuple> --host=<host tuple>                  \
                    --prefix=/usr --enable-foo-bar...                      \
                    CPPFLAGS="-I/path/to/staging/usr/include"              \
                    LDFLAGS="-L/path/to/staging/lib -L/path/to/staging/usr/lib"
$ make
$ make DESTDIR=/path/to/staging install
```

Option 4: A mix of options 2 and 3, using carefully crafted union mounts.

The staging area is a union mount of:

the sysroot as a read-only branch

the real staging area as a read-write branch

This also requires passing --sysroot to point to the union mount, but has other advantages, such as allowing per-package staging, and a few more obscure pros. It also has its disadvantages, as it potentially requires non-root users to create union mounts. Additionally, union mounts are not yet mainstream in the Linux kernel, so it requires patching. There is a FUSE-based unionfs implementation, but development is almost stalled, and there are a few gotchas.

```
$ (good luck!)
```

It is strongly advised not to use the toolchain sysroot directory as an install directory (i.e., option 1) for your programs/packages. If you do so, you will not be able to use your toolchain for another project. It is even strongly advised that your toolchain is chmod-ed to read-only once successfully install, so that you don’t go polluting your toolchain with your programs’/packages’ files. This can be achieved by selecting the “Render the toolchain read-only” from crosstool-NG’s “Paths and misc options” configuration page.

Thus, when you build a program/package, install it in a separate, staging, directory and let the cross-toolchain continue to use its own, pristine, sysroot directory.

When you are done building and want to assemble your rootfs you could simply take the full contents of your staging directory and use the populate script to add in the necessary files from the sysroot. However, the staging area you have created will include lots of build artifacts that you won’t necessarily want/need on your target. For example: static libraries, header files, linking helper files, man/info pages. You’ll also need to add various configuration files, scripts, and directories to the rootfs so it will boot.

Therefore you’ll probably end up creating a separate rootfs directory which you will populate from the staging area, necessary extras, and then use crosstool-NG’s populate script to add the necessary sysroot libraries.
The populate script

When your root directory is ready, it is still missing some important bits: the toolchain’s libraries. To populate your root directory with those libs, just run:

```
your-target-tuple-populate -s /your/root -d /your/root-populated
```

This will copy /your/root into /your/root-populated, and put the needed and only the needed libraries there. Thus you don’t pollute /your/root with any cruft that would no longer be needed should you have to remove stuff. /your/root always contains only those things you install in it.

You can then use /your/root-populated to build up your file system image, a tarball, or to NFS-mount it from your target, or whatever you need.

The populate script accepts the following options:

-s src_dir: Use src_dir as the un-populated root directory.

-d dst_dir: Put the populated root directory in dst_dir.

-l lib1 [...]: Always add specified libraries.

-L file: Always add libraries listed in file.

-f: Remove dst_dir if it previously existed; continue even if any library specified with -l or -L is missing.

-v: Be verbose, and tell what’s going on (you can see exactly where libs are coming from).

-h: Print the help.

See your-target-tuple-populate -h for more information on the options.

Here is how populate works:

```
Perform some sanity checks:
    src_dir and dst_dir are specified,
    src_dir exists,
    unless forced, dst_dir does not exist,
    src_dir != dst_dir.

Copy src_dir to dst_dir.
Add forced libraries to dst_dir:
    build the list from -l and -L options,
    get forced libraries from the sysroot (see below for heuristics)
        abort on the first missing library, unless -f is specified.
Add all missing libraries to dst_dir:
    scan dst_dir for every ELF files that are executable or shared object
    list the “NEEDED Shared library” fields
        check if the library is already in dst_dir/lib or dst_dir/usr/lib
        if not, get the library from the sysroot
            if it’s in sysroot/lib, copy it to dst_dir/lib
            if it’s in sysroot/usr/lib, copy it to dst_dir/usr/lib
            in both cases, use the SONAME of the library to create the file in dst_dir
            if it was not found in the sysroot, this is an error.
```

The cross-ldd script

There is another script provided by crosstool-NG to work with the sysroot on the host. A dynamically linked application will load certain shared libraries at runtime. These libraries in turn may require some more shared libraries as dependencies. The search paths for each of these dynamic objects differ, and finding the shared libraries required for a given application is not always trivial. Crosstool-NG attempts to solve this task by providing a ${CT_TARGET}-ldd script in the generated toolchain (optionally, if the “Install a cross ldd-like helper” option is selected in the configuration).

This script recursively resolves all the dynamic library dependencies and outputs the list of libraries in a format compatible with that of GNU libc’s ldd script. It needs to have the system root specified, using either the --root option or the CT_XLDD_ROOT environment variable.

For example:

```
PATH=~/x-tools/powerpc64-multilib-linux-gnu/bin:$PATH
powerpc64-multilib-linux-gnu-gcc -o example example.c
powerpc64-multilib-linux-gnu-ldd       --root=`powerpc64-multilib-linux-gnu-gcc
-print-sysroot` example
```

produces an output like this:

```
        libc.so.6 => /lib64/libc.so.6 (0x00000000deadbeef)
        ld64.so.1 => /lib64/ld64.so.1 (0x00000000deadbeef)
```

The load addresses are obviously bogus, as this script does not actually load the libraries.

6、Toolchain Types

There are four kinds of toolchains you could encounter.

First off, you must understand the following: when it comes to compilers there are up to four machines involved:

the machine configuring the toolchain components: the config machine
the machine building the toolchain components: the build machine
the machine running the toolchain: the host machine
the machine the toolchain is generating code for: the target machine

We can most of the time assume that the config machine and the build machine are the same. Most of the time, this will be true. The only time it isn’t is if you’re using distributed compilation (such as distcc). Let’s forget this for the sake of simplicity.

So we’re left with three machines:

build
host
target

Any toolchain will involve those three machines. You can be as pretty sure of this as “2 and 2 are 4”. Here is how they come into play:

build == host == target (“native”)

This is a plain native toolchain, targeting the exact same machine as the one it is built on, and running again on this exact same machine. You have to build such a toolchain when you want to use an updated component, such as a newer gcc for example.

build == host != target (“cross”)

This is a classic cross-toolchain, which is expected to be run on the same machine it is compiled on, and generate code to run on a second machine, the target.

build != host == target (“cross-native”)

Such a toolchain is also a native toolchain, as it targets the same machine as it runs on. But it is build on another machine. You want such a toolchain when porting to a new architecture, or if the build machine is much faster than the host machine.

build != host != target (“canadian”)

This one is called a “Canadian Cross”[1] toolchain, and is tricky. The three machines in play are different. You might want such a toolchain if you have a fast build machine, but the users will use it on another machine, and will produce code to run on a third machine.

The term Canadian Cross was coined because at the time that these issues were all being hashed out, Canada had three national political parties (per Wikipedia).

crosstool-NG can build all these kinds of toolchains, or is aiming at it, anyway. There are a few caveats, though.

While building a “native” toolchain, crosstool-ng will currently still compile new version of libc for the target. There is currently no way to use the system libc and/or system kernel headers as a part of the toolchain. This may work if you choose a compatible version (i.e., the applications compiled with the toolchain will load the system libc).

A “cross-native” toolchain can be built as a trivial case of the “canadian” toolchain. It is suboptimal, as it makes crosstool-NG build the tools targeting the host machine twice (first, as a separate toolchain which is a prerequisite for all canadian builds; and second, as a part of temporary toolchain created as a part of the canadian build itself). This will likely be improved in the future.

To build a “canadian” toolchain, you must build a toolchain that runs on build and targets the host as a prerequisite (i.e., a simple cross). Then, add the /bin directory of this prerequisite to the $PATH environment variable and configure the canadian, specifying the target of the prerequisite toolchain as the host of the new toolchain.

There are a few samples of canadian toolchains shipped with crosstool-NG. The names of the canadian samples consist of two comma-separated parts, i.e. HOST,TARGET. They require HOST sample as a prerequisiste. For example:

```
ct-ng x86_64-w64-mingw32
ct-ng build
PATH=~/x-tools/x86_64-w64-mingw32/bin:$PATH
ct-ng x86_64-w64-mingw32,x86_64-pc-linux-gnu
ct-ng build
```

Note that you will not be able to run the binaries from the canadian toolchain on your build machine! You need to transfer them to a machine running the OS configured as the host.

7、Contributing to crosstool-NG

Sending a bug report

Please file the bug reports on Github.

Crosstool-NG also has a mailing list at crossgcc@sourceware.org. Archive and subscription info can be found here: https://sourceware.org/ml/crossgcc/

Sometimes you may be able to contact a maintainer on IRC at the #crosstool-ng channel on the FreeNode net (irc.freenode.net).

Using a mailing list or IRC to triage an issue is okay. If it looks like a bug, you'll be asked to file an issue at Github. If it looks like a pilot error, you might get an advice over the mailing list or IRC faster.

Sending patches

The preferred method of patch submission is via the Github's pull requests. The status of currently pending pull requests can be checked here.

Patches are also welcome at the mailing list.

Patches should come with the appropriate Signed-off-by line (SOB line). An SOB line is typically something like:

Signed-off-by: John DOE <john.doe@somewhere.net>

Why this line is needed is well described in the Linux kernel's documentation on patch submission.

You can also add any of the following lines if applicable:

Acked-by:
Tested-by:
Reviewed-by:

If submitting patches over the mailing list, please also follow other guidelines described in the Linux kernel’s guide.

We previously used patchwork for development, but it is no longer used. I’d like to see patches that are still applicable turned into Pull Requests on GitHub. Here is the list of pending patches.

8 、rosstool-NG Internals

Internally, crosstool-NG is script-based. To ease usage, the frontend is Makefile-based.

Makefile front-end

The entry point to crosstool-NG is the Makefile script ct-ng. Calling this script with an action will act exactly as if the Makefile was in the current working directory and make was called with the action as rule. Thus:

ct-ng menuconfig

is equivalent to having the Makefile in CWD, and calling:

make menuconfig

Having ct-ng as it is avoids copying the Makefile everywhere, and acts as a traditional command.

ct-ng loads sub- Makefiles from the library directory $(CT_LIB_DIR), as set up at configuration time with ./configure.

ct-ng also searches for config files, sub-tools, samples, scripts and patches in that library directory.

Because of a stupid make behavior/bug I was unable to track down, implicit make rules are disabled: installing with --local would trigger those rules, and mconf was unbuildable.

Kconfig parser

The kconfig language is a hacked version, vampirised from the Linux kernel, and (heavily) adapted to my needs.

The list of the most notable changes (at least the ones I remember) follows:

the CONFIG_ prefix has been replaced with CT_

a leading | in prompts is skipped, and subsequent leading spaces are not trimmed; otherwise leading spaces are silently trimmed

removed the warning about undefined environment variable

The kconfig parsers (conf and mconf) are not installed pre-built, but as source files. Thus you can have the directory where crosstool-NG is installed, exported (via NFS or whatever) and have clients with different architectures use the same crosstool-NG installation, and most notably, the same set of patches.
Architecture-specific

Note

this chapter is not really well written, and might thus be a little bit complex to understand. To get a better grasp of what an architecture is, the reader is kindly encouraged to look at the arch/ sub-directory, and to the existing architectures to see how things are laid out.

An architecture is defined by:

a human-readable name, in lower case letters, with numbers as appropriate; underscore is allowed; space and special characters are not, e.g.:

arm
x86_64

a file in config/arch/, named after the architecture's name, and suffixed with .in, e.g.:

config/arch/arm.in

a file in scripts/build/arch/, named after the architecture's name, and suffixed with .sh, e.g.:

scripts/build/arch/arm.sh

The architecture's .in file API

Note

Here, and in the following, %arch% is to be replaced with the actual architecture name.

The ARCH_%arch% option

This config option must have neither a type, nor a prompt! Also, it can not depend on any other config option.

EXPERIMENTAL is managed as in kernel options ?.

A (terse) help entry must be defined for this architecture, e.g.,

```
config ARCH_arm
  help
    The ARM architecture.
```

Adequate associated config options may be selected, e.g.,

```
config ARCH_arm
  select ARCH_SUPPORTS_BOTH_ENDIAN
  select ARCH_DEFAULT_LE
  help
    The ARM architecture.
```

Note

64-bit architectures shall select ARCH_64.

```
config ARCH_x86_64
   select ARCH_64
   help
     The x86_64 architecture.
```

Other target-specific options

At your discretion. Note however that to avoid name-clashing, such options shall be prefixed with ARCH_%arch%.

Note

Due to historical reasons, and lack of time to clean up the code, I may have left some config options that do not completely conform to this, as the architecture name was written all upper case. However, the prefix is unique among architectures, and does not cause harm).

The architecture’s .sh file API

the function CT_DoArchTupleValues

parameters: none

environment:

all variables from the .config file,

the two variables target_endian_eb and target_endian_el which are the endianness suffixes

return value: 0 upon success, !0 upon failure

provides:

environment variable CT_TARGET_ARCH (mandatory)
contains: the architecture part of the target tuple, e.g. armeb for big endian ARM, or i386 for an i386

environment variable CT_TARGET_SYS (optional)

contains: the system part of the target tuple, e.g., gnu for glibc on most architectures, or gnueabi for glibc on an ARM EABI

defaults to:

gnu for glibc-based toolchain

uclibc for uClibc-based toolchain

environment variables to configure the cross-gcc (defaults) (optional)

gcc ./configure switch	selects	default
CT_ARCH_WITH_ARCH	architecture level	--with-arch=${CT_ARCH_ARCH}
CT_ARCH_WITH_ABI	ABI level	--with-abi=${CT_ARCH_ABI}
CT_ARCH_WITH_CPU	CPU instruction set	--with-cpu=${CT_ARCH_CPU}
CT_ARCH_WITH_TUNE	scheduling	--with-tune=${CT_ARCH_TUNE}
CT_ARCH_WITH_FPU	FPU type	--with-fpu=${CT_ARCH_FPU}
CT_ARCH_WITH_FLOAT	floating point arithm.	--with-float=soft or [empty]

environment variables to pass to the cross-gcc to build target binaries (defaults) (optional)

gcc ./configure switch	selects	default
CT_ARCH_ARCH_CFLAG	architecture level	-march=${CT_ARCH_ARCH}

CT_ARCH_ABI_CFLAG	ABI level	-mabi=${CT_ARCH_ABI}
CT_ARCH_CPU_CFLAG	CPU instruction set	-mcpu=${CT_ARCH_CPU}
CT_ARCH_TUNE_CFLAG	scheduling	-mtune=${CT_ARCH_TUNE}
CT_ARCH_FPU_CFLAG	FPU type	-mfpu=${CT_ARCH_FPU}
CT_ARCH_FLOAT_CFLAG	floating point arithm.	-msoft-float or [empty]
CT_ARCH_ENDIAN_CFLAG	big or little endian	-mbig-endian or -mlittle-endian

the environment variables to configure the core and final compiler, specific to this architecture (optional):

CT_ARCH_CC_CORE_EXTRA_CONFIG: additional, architecture specific core gcc ./configure flags

CT_ARCH_CC_EXTRA_CONFIG: additional, architecture specific final gcc ./configure flags

default to: all empty

the architecture-specific CFLAGS and LDFLAGS (optional):

CT_ARCH_TARGET_CLFAGS

CT_ARCH_TARGET_LDFLAGS

default to: all empty

You can have a look at config/arch/arm.in and scripts/build/arch/arm.sh for a quite complete example of what an actual architecture description looks like.

Kernel specific

A kernel is defined by:

a human-readable name, in lower case letters, with numbers as appropriate; underscore is allowed, space and special characters are not (although they are internally replaced with underscores); e.g.:

linux
bare-metal

a file in config/kernel/, named after the kernel name, and suffixed with .in, e.g.:

config/kernel/linux.in

config/kernel/bare-metal.in

a file in scripts/build/kernel/, named after the kernel name, and suffixed with .sh, e.g.:

scripts/build/kernel/linux.sh
scripts/build/kernel/bare-metal.sh

The kernel's .in file must contain:

an optional line containing exactly # EXPERIMENTAL, starting on the first column, and without any following space or other character.

If this line is present, then this kernel is considered EXPERIMENTAL, and correct dependency on EXPERIMENTAL will be set.

the config option KERNEL_%kernel_name% (where %kernel_name% is to be replaced with the actual kernel name, with all special characters and spaces replaced by underscores), e.g.:

KERNEL_linux
KERNEL_bare_metal

This config option must have neither a type, nor a prompt! Also, it can not depends on EXPERIMENTAL.

A (terse) help entry for this kernel must be defined, e.g.:

```
config KERNEL_bare_metal
  help
    Build a compiler for use without any kernel.
```

Adequate associated config options may be selected, e.g.:

```
config KERNEL_bare_metal
  select BARE_METAL
  help
    Build a compiler for use without any kernel.
```

other kernel specific options, at your discretion. Note however that, to avoid name-clashing, such options should be prefixed with KERNEL_%kernel_name%, where %kernel_name% is again to be replaced with the actual kernel name.

Note

Due to historical reasons, and lack of time to clean up the code, Yann may have left some config options that do not completely conform to this, as the kernel name was written all upper case. However, the prefix is unique among kernels, and does not cause harm).

The kernel’s .sh file API:

- is a bash script fragment
- defines function CT_DoKernelTupleValues
  - see the architecture’s CT_DoArchTupleValues, except for: [FIXME?]
  - set the environment variable CT_TARGET_KERNEL, the kernel part of the target tuple
  - return value: ignored
- defines function do_kernel_get:
  - parameters: none
  - environment:
    all variables from the .config file.
  - return value: 0 for success, !0 for failure.
  - behavior: download the kernel’s sources, and store the tarball into ${CT_TARBALLS_DIR}. To this end, a function is available that abstracts downloading tarballs:

    CT_DoGet <tarball_base_name> <URL1 [URL...]>, e.g.:

    CT_DoGet linux-2.6.26.5 ftp://ftp.kernel.org/pub/linux/kernel/v2.6

    Note

    Retrieving sources from SVN, CVS, git and the likes is not supported by CT_DoGet. You’ll have to do this by hand, as it is done for eglibc in scripts/build/libc/eglibc.sh.
- defines function do_kernel_extract:
  - parameters: none
  - environment:
    all variables from the .config file,

return value: 0 for success, !0 for failure.

behavior: extract the kernel’s tarball into ${CT_SRC_DIR}, and apply required patches. To this end, a function is available that abstracts extracting tarballs:

CT_ExtractAndPatch <tarball_base_name>, e.g.:

CT_ExtractAndPatch linux-2.6.26.5

defines function do_kernel_headers:

parameters: none

environment:
all variables from the .config file,

return value: 0 for success, !0 for failure.

behavior: install the kernel headers (if any) in ${CT_SYSROOT_DIR}/usr/include

defines any kernel-specific helper functions

These functions, if any, must be prefixed with do_kernel_%CT_KERNEL%_, where %CT_KERNEL% is to be replaced with the actual kernel name, to avoid any name-clashing.

You can have a look at config/kernel/linux.in and scripts/build/kernel/linux.sh as an example of what a complex kernel description looks like.
Adding a new version of a component

When a new component, such as the Linux kernel, gcc or any other is released, adding the new version to crosstool-NG is quite easy. There is a script that will do all that for you:

scripts/addToolVersion.sh

Run it with no option to get some help.
Build scripts

[To Be Written later...]

9、How a toolchain is constructed

This is the result of a discussion with Francesco Turco.

Francesco had a nice tutorial for beginners [dead link, Wayback Machine has no archived version], along with a sample, step-by-step procedure to build a toolchain for an ARM target from an x86_64 Debian host.

Thank you Francesco for initiating this!
I want a cross-compiler! What is this toolchain you’re speaking about?

A cross-compiler is in fact a collection of different tools set up to tightly work together. The tools are arranged in a way that they are chained, in a kind of cascade, where the output from one becomes the input to another one, to ultimately produce the actual binary code that runs on a machine. So, we call this arrangement a “toolchain”. When a toolchain is meant to generate code for a machine different from the machine it runs on, this is called a cross-toolchain.
So, what are those components in a toolchain?

The components that play a role in the toolchain are first and foremost the compiler itself. The compiler turns source code (in C, C++, whatever) into assembly code. The compiler of choice is the GNU compiler collection, well known as gcc.

The assembly code is interpreted by the assembler to generate object code. This is done by the binary utilities, such as the GNU binutils.

Once the different object code files have been generated, they got to get aggregated together to form the final executable binary. This is called linking, and is achieved with the use of a linker. The GNU binutils also come with a linker.

So far, we get a complete toolchain that is capable of turning source code into actual executable code. Depending on the Operating System, or the lack thereof, running on the target, we also need the C library. The C library provides a standard abstraction layer that performs basic tasks (such as allocating memory, printing output on a terminal, managing file access…). There are many C libraries, each targeted to different systems. For the Linux desktop, there is glibc or eglibc or even uClibc, for embedded Linux, you have a choice of eglibc or uClibc, while for system without an operating system, you may use newlib, dietlibc, or even none at all. There a few other C libraries, but they are not as widely used, and/or are targeted to very specific needs (e.g., klibc is a very small subset of the C library aimed at building constrained initial ramdisks).

Under Linux, the C library needs to know the API to the kernel to decide what features are present, and if needed, what emulation to include for missing features. That API is provided by the kernel headers. Note: this is Linux-specific

(and potentially a very few others), the C library on other OSes do not need the kernel headers.
And now, how do all these components chained together?

So far, all major components have been covered, but yet there is a specific order they need to be built. Here we see what the dependencies are, starting with the compiler we want to ultimately use. We call that compiler the final compiler.

the final compiler needs the C library, to know how to use it, but:

building the C library requires a compiler

A needs B which needs A. This is the classic chicken'n'egg problem… This is solved by building a stripped-down compiler that does not need the C library, but is capable of building it. We call it a bootstrap, initial, or core compiler. So here is the new dependency list:

the final compiler needs the C library, to know how to use it,

building the C library requires a core compiler but:

the core compiler needs the C library headers and start files, to know how to use the C library

B needs C which needs B. Chicken'n'egg, again. To solve this one, we will need to build a C library that will only install its headers and start files. The start files (also called "C runtime", or CRT) are a very few files that gcc needs to be able to turn on thread local storage (TLS) on an NPTL system. So now we have:

the final compiler needs the C library, to know how to use it,

building the C library requires a core compiler

the core compiler needs the C library headers and start files, to know how to use the C library but:

building the start files require a compiler

Geez… C needs D which needs C, yet again. So we need to build a yet simpler compiler, that does not need the headers and does need the start files. This compiler is also a bootstrap, initial or core compiler. In order to differentiate the two core compilers, let's call that one core pass 1, and the former one core pass 2. The dependency list becomes:

the final compiler needs the C library, to know how to use it, building the C library requires a compiler

the core pass 2 compiler needs the C library headers and start files, to know how to use the C library

building the start files requires a compiler

we need a core pass 1 compiler

And as we said earlier, the C library also requires the kernel headers. There is no requirement for the kernel headers, so end of story in this case:

the final compiler needs the C library, to know how to use it,

building the C library requires a core compiler

the core pass 2 compiler needs the C library headers and start files, to know how to use the C library

building the start files requires a compiler and the kernel headers

we need a core pass 1 compiler

We need to add a few new requirements. The moment we compile code for the target, we need the assembler and the linker. Such code is, of course, built from the C library, so we need to build the binutils before the C library start files, and the complete C library itself. Also, some code in gcc will turn to run on the target as well. Luckily, there is no requirement for the binutils. So, our dependency chain is as follows:

the final compiler needs the C library, to know how to use it, and the binutils

building the C library requires a core pass 2 compiler and the binutils

the core pass 2 compiler needs the C library headers and start files, to know how to use the C library, and the binutils

building the start files requires a compiler, the kernel headers and the binutils

the core pass 1 compiler needs the binutils

Which turns in this order to build the components:

binutils

core pass 1 compiler

kernel headers

C library headers and start files

core pass 2 compiler

complete C library

final compiler

Yes! :-) But are we done yet?

In fact, no, there are still missing dependencies. As far as the tools themselves are involved, we do not need anything else.

But gcc has a few pre-requisites. It relies on a few external libraries to perform some non-trivial tasks (such as handling complex numbers in constants…). There are a few options to build those libraries. First, one may think to rely on a Linux distribution to provide those libraries. Alas, they were not widely available until very, very recently. So, if the distro is not too recent, chances are that we will have to build those libraries (which we do below). The affected libraries are:

the GNU Multiple Precision Arithmetic Library, GMP;

the C library for multiple-precision floating-point computations with correct rounding, MPFR;

the C library for the arithmetic of complex numbers, MPC.

The dependencies for those libraries are:

MPC requires GMP and MPFR

MPFR requires GMP

GMP has no pre-requisite

So, the build order becomes:

GMP

MPFR

MPC

binutils

core pass 1 compiler

kernel headers

C library headers and start files

core pass 2 compiler

complete C library

final compiler

Yes! Or yet some more?

This is now sufficient to build a functional toolchain. So if you’ve had enough for now, you can stop here. Or if you are curious, you can continue reading.

gcc can also make use of a few other external libraries. These additional, optional libraries are used to enable advanced features in gcc, such as loop optimisation (GRAPHITE) and Link Time Optimisation (LTO). If you want to use these, you’ll need three additional libraries:

To enable GRAPHITE, depending on GCC version, it may need one or more of the following:

the Parma Polyhedra Library, PPL;
the Integer Set Library, ISL;
the Chunky Loop Generator, using the PPL backend, CLooG/PPL;
the Chunky Loop Generator, using the ISL backend, CLooG.

To enable LTO: - the ELF object file access library, libelf

The dependencies for those libraries are:

PPL requires GMP;

CLooG/PPL requires GMP and one of PPL or ISL;

ISL has no prerequisites;

libelf has no pre-requisites.

The list now looks like:

GMP

MPFR

MPC

PPL (if needed)

ISL (if needed)

CLooG (if needed)

libelf (if needed)

binutils

core pass 1 compiler

kernel headers

C library headers and start files

core pass 2 compiler

complete C library

final compiler

This list is now complete! Wouhou! Or is it?
But why does crosstool-NG have more steps?

The already thirteen steps are the necessary steps, from a theoretical point of view. In reality, though, there are small differences; there are three different reasons for the additional steps in crosstool-NG.

First, the GNU binutils do not support some kinds of output. It is not possible to generate flat binaries with binutils, so we have to use another component that adds this support: elf2flt. elf2flt also requires the zlib compression library - we may not be able to use the host’s zlib if we’re building a canadian or cross-native toolchain.

Second, localizations of the toolchain require additional libraries on some host OSes: gettext and libiconv.

Third, crosstool-NG can also build some additional debug utilities to run on the target. This is where we build, for example, the cross-gdb, the gdbserver and the native gdb (the last two run on the target, the first runs on the same machine as the toolchain). The others (strace, ltrace, DUMA and dmalloc) are absolutely not related to the toolchain, but are nice-to-have stuff that can greatly help when developing, so are included as goodies (and they are quite easy to build, so it’s OK; more complex stuff is not worth the effort to include in crosstool-NG).

10、Credits

This section comes from crosstool-NG author, Yann Morin. Now that crosstool-NG uses Git (and previously used Mercurial), each contribution is attributed to its author, so this file will probably not be updated anymore.

I would like to thank these fine people for making crosstool-NG possible:

Dan Kegel, the original author of crosstool:

Dan was very helpful and willing to help when I build my first toolchains. I owe him one. Thank you Dan! Some crosstool-NG scripts have code snippets coming almost as-is from the original work by Dan.

And in order of appearance on the mailing list:

Allan Clark:

Allan made extensive tests of the first alpha of crosstool-NG on his MacOS-X, and unveiled some bash-2.05 weirdness.

Enrico Weigelt:

some improvements to the build procedure
cxa_atexit disabling for C libraries not supporting it (old uClibc)
misc suggestions (restartable build, ...)
get rid of some bashisms in ./configure
contributed OpenRISC or32 support

Robert P. J. Day:

some small improvements to the configurator, misc prompting glitches
‘sanitised’ patches for binutils-2.17
patches for glibc-2.5
misc patches, typos and eye candy
too many to list any more!

Al Stone:

initial ia64 support
some cosmetics

Szilveszter Ordog:

a uClibc floating point fix
initial support for ARM EABI

Mark Jonas:

initiated Super-H port

Michael Abbott:

make it build with ancient findutils

Willy Tarreau:

a patch to glibc to build on ‘ancient’ shells
reported mis-use of $CT_CC_NATIVE

Matthias Kaehlcke:

fix building glibc-2.7 (and 2.6.1) with newer kernels

Daniel Dittmann:

PowerPC support

Ioannis E. Venetis:

preliminary Alpha support
intense gcc-4.3 brainstorming

Thomas Jourdan:

intense gcc-4.3 brainstorming
eglibc support

Konrad Eisele:

initial multilib support

Many others have contributed, either in form of patches, suggestions, comments, or testing… Thank you to all of you!

Special dedication to the buildroot people for maintaining a set of patches I happily and shamelessly vampirize from time to time.

11 、Known issues

This files lists the known issues encountered while developing crosstool-NG, but that could not be addressed before the release.

The file has one section for each known issue, each section containing four sub-sections: Symptoms, Explanations, Fix, and Workaround.

Each section is separated from the others with a lines of at least 4 dashes.

The following dummy section explains it all.

Symptoms: A one- or two-liner of what you would observe. Usually, the error message you would see in the build logs.

Explanations: An as much as possible in-depth explanations of the context, why it happens, what has been investigated so far, and possible orientations as how to try to solve this (eg. URLs, code snippets...).

Status: Tells about the status of the issue:

UNCONFIRMED : missing information, or unable, to reproduce, but there is consensus that there is an issue somewhere...
CURRENT : the issue is applicable.
DEPRECATED : the issue used to apply in some cases, but has not been confirmed or reported again lately.
CLOSED : the issue is no longer valid, and a fix has been added either as a patch to this component, and/or as a workaround in the scripts and/or the configuration.

Fix: What you have to do to fix it, if at all possible. The fact that there is a fix, and yet this is a known issue means that time to incorporate the fix in crosstool-NG was missing, or planned for a future release.

Workaround: What you can do to fix it temporarily, if at all possible. A workaround is not a real fix, as it can break other parts of crosstool-NG, but at least makes you going in your particular case.

So now, on for the real issues...

Symptoms: gcc is not found, although I do have gcc installed.

Explanations: This is an issue on at least RHEL systems, where gcc is a symlink to ccache. Because crosstool-NG create links to gcc for the build and host environment, those symlinks are in fact pointing to ccache, which then doesn’t know how to run the compiler.

A possible fix could probably set the environment variable CCACHE_CC to the actual compiler used.

Status: CURRENT

Fix: None known.

Workaround: Uninstall ccache.

Symptoms: Build fails with: unable to detect the exception model

Explanations: On some architectures, proper stack unwinding (C++) requires that setjmp/longjmp (sjlj) be used, while on other architectures do not need sjlj. On some architectures, gcc is unable to determine whether sjlj are needed or not.

Status: CURRENT

Fix: None so far.

Workaround: Trying setting use of sjlj to either ‘Y’ or ‘N’ (instead of the default ‘M’) in the menuconfig, option CT_CC_GCC_SJLJ_EXCEPTIONS labeled “Use sjlj for exceptions”.

Symptoms: On x86_64 hosts with 32bit userspace the GMP build fails with:

```
configure: error: Oops, mp_limb_t is 32 bits, but the assembler code
in this configuration expects 64 bits.
You appear to have set $CFLAGS, perhaps you also need to tell GMP the
intended ABI, see "ABI and ISA" in the manual.
```

Explanations: uname -m detects x86_64 but the build host is really x86.

Status: CURRENT

Fix: None so far. See above issue.

Workaround: use “setarch i686 ct-ng build”

12、Setting up host OS

This section describes the setup needed by various operating systems in order to run crosstool-NG, as well as some OS-specific caveats and limitations. The package lists given in the following subsections cover all the features tested by the sample configurations. You particular configuration may not need all those packages. For example, git is needed if your configuration is for an uClinux-based target which requires elf2flt utilities (which does “rolling releases” and must be checked out from a Git repository).

If on the other hand you encounter a dependency not listed here, please let us know over the mailing list or via a pull request!

Linux: ArchLinux

The following packages are needed for crosstool-NG (assuming pacstrap ... base base-devel and pacman -S grub os-prober were performed during installation): git help2man python gperf. Install them with pacman -S PACKAGES.

Notes

The default color scheme used by Kconfig on ArchLinux makes the active menu selection hard to see. A workaround for this issue is to put the following line into your shell’s profile (e.g. ~/.bash_profile or ~/.bashrc for bash):

```
export MENUCONFIG_COLOR=mono
```

Linux: CentOS

The following packages need to be installed on CentOS 7:

```
yum install autoconf gperf bison flex texinfo help2man gcc-c++ patch \
    ncurses-devel python-devel perl-Thread-Queue bzip2 git
```

Linux: Fedora Core

The following packages need to be installed on Fedora Core 25:

```
dnf install autoconf gperf bison flex texinfo help2man gcc-c++ patch \
    ncurses-devel python-devel perl-Thread-Queue git
```

Linux: Gentoo

The following packages need to be installed after installing a minimal profile:

```
emerge dev-vcs/git
```

Linux: Ubuntu

The following packages need to be installed on Ubuntu 16.04.2 (server):

```
apt-get install gcc gperf bison flex texinfo help2man make libncurses5-dev \
    python-dev
```

macOS (a.k.a Mac OS X, OS X)

Originally contributed by: Titus von Boxberg

The instructions below have been verified on macOS Sierra (10.12). They have been previously reported to work with versions since Mac OS X Snow Leopard (10.6) with Developer Tools 3.2, and with Mac OS X Leopard (10.5) with Developer Tools + GCC 4.3 or newer installed via MacPorts.

You have to use a case sensitive file system for crosstool-NG's build and target directories. Use a disk or disk image with a case sensitive FS that you mount somewhere.

Install required tools via HomeBrew. The following set is sufficient for HomeBrew: autoconf binutils gawk gmp gnu-sed help2man mpfr openssl pcre readline wget xz. Install them using brew install PACKAGE command.

Also, installing homebrew/dupes/grep is recommended. It has been noticed that GNU libc was misconfigured due to a subtle difference between BSD grep (which is used by macOS) and GNU grep. This has since been fixed, but other scripts in various packages may still contain GNUisms.

If you prefer to use MacPorts, refer to the previous version of the instruction below and let us know if it works with current crosstool-NG and macOS releases.

Mac OS X defaults to a fairly low limit on the number of the files that can be opened by a process (256) that is exceeded by the build framework of the GNU C library. Increase this limit to 1024:

```
ulimit -n 1024
```

Notes:

When building on macOS, the following message may be displayed:

```
clang: error: unsupported option '-print-multi-os-directory'
clang: error: no input files
```

It is reported when the host version of libiberty (from GCC) is compiled by macOS default compiler, clang. In absense of any reported multilib information, libiberty is then configured with the default compilation flags. This does not seem to affect the resulting toolchain.

ct-ng menuconfig will not work on Snow Leopard 10.6.3 since libncurses is broken with this release. MacOS <= 10.6.2 and >= 10.6.4 are ok.

Previous version of the installation guidelines

Crosstool-NG has been reported to work with MacPorts as well, using the following set of ports: lzmautils libtool binutils gsed gawk. On Mac OS X Leopard, it is also required to install gcc43 and gcc_select.

On Leopard, make sure that the MacPort's gcc is called with the default commands (gcc, g++,…), via MacPort's gcc_select.

On OSX 10.7 Lion / when using Xcode >= 4 make sure that the default commands gcc, g++, etc.) point to gcc-4.2, NOT llvm-gcc-4.2 by using MacPort's gcc_select feature. With MacPorts >= 1.9.2 the command is: "sudo port select –set gcc gcc42"

This also requires (like written above) that macport's bin directory comes before the standard directories in your PATH environment variable because the gcc symlink is installed in /opt/local/bin and the default /usr/bin/gcc is not removed by the gcc_select command!

Explanation: llvm-gcc-4.2 (with Xcode 4.1 it is on my machine "gcc version 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)") cannot boostrap gcc. See this bug

Apparently, GNU make's builtin variable .LIBPATTERNS is misconfigured under MacOS: It does not include lib%.dylib. This affects build of (at least) GDB 7.1 Put lib%.a lib%.so lib%.dylib as .LIBPATTERNS into your environment before executing ct-ng build. See here for details.

Note however, that GDB 7.1 (and anything earlier than 7.10) are known to fail to build on macOS.

Windows: Cygwin

Originally contributed by: Ray Donnelly

Prerequisites and instructions for using crosstool-NG for building a cross toolchain on Windows (Cygwin) as build and, optionally Windows (hereafter) MinGW-w64 as host.

Use Cygwin64 if you can. DLL base-address problems are lessened that way and if you bought a 64-bit CPU, you may as well use it.

You must enable Case Sensitivity in the Windows Kernel (this is only really necessary for Linux targets, but at present, crosstool-ng refuses to operate on case insensitive filesystems). The registry key for this is: HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\kernel\obcaseinsensitive Read more here.

Since release 1.7, Cygwin no longer supports the ‘managed’ mount option. You must use case sensitive FS.
Using setup.exe or setup-x86_64.exe, install the default packages and also the following ones:

autoconf
automake
bison
diffutils
flex
gawk
gcc-g++
git
gperf
help2man
libncurses-devel
make
patch
python-devel
texinfo
wget
xz Leave “Select required packages (RECOMMENDED)” ticked.

Notes:
The packages marked with * are only needed if your host is MinGW-w64.

Although nativestrict symlinks seem like the best idea, extracting glibc fails when they are enabled, so just don’t set anything here. If your host is MinGW-w64 then these ‘Cygwin-special’ symlinks won’t work, but you can dereference them by using tar options –dereference and –hard-dereference when making a final tarball. I plan to investigate and fix or at least work around the extraction problem. Read more here.
If both BFD and GOLD linkers are enabled in binutils, collect2.exe will attempt to run ld which is a shell script so you need to make sure that a working shell is in your path. Eventually this will be replaced with a native program for MinGW-w64 host.

Notes

Cygwin is slow. No, really, really slow. Expect about approximately 5x to 10x slowdown compared to a Linux system.

FreeBSD (and other BSD)

FreeBSD support is currently experimental in crosstool-NG.

FreeBSD does not provide a gcc command by default. Crosstool-NG and many of the packages used expect this by default. A comprehensive fix for various ways of setting up the OS is planned after the 1.23 release. Until then, setting up the following packages is recommended as a prerequisite for crosstool-NG:

```
archivers/zip
devel/automake
devel/bison
devel/gettext-tools
devel/git
devel/gmake
devel/gperf
devel/libatomic_ops
devel/libtool
devel/patch
lang/gcc6
lang/gawk
misc/help2man
print/texinfo
textproc/asciidoc
textproc/gsed
textproc/xmlto
```

Use any supported method of installation, e.g.:

```
cd /usr/ports/lang/gcc6
make install clean
```

Even with these packages installed, some of the samples are failing to build. YMMV.

Previous version of the installation guidelines

Contributed by: Titus von Boxberg

Prerequisites and instructions for using ct-ng for building a cross toolchain on FreeBSD as host.

Tested on FreeBSD 8.0

Install (at least) the following ports archivers/lzma textproc/gsed devel/gmake devel/patch shells/bash devel/bison lang/gawk devel/automake110 ftp/wget

Of course, you should have /usr/local/bin in your PATH.
run ct-ng’s configure with the following tool configuration:

```
./configure --with-sed=/usr/local/bin/gsed \
    --with-make=/usr/local/bin/gmake \
    --with-patch=/usr/local/bin/gpatch
[...other configure parameters as you like...]
```

proceed as described in general documentation but use gmake instead of make

13、Notes on specific toolchain features

This section describes setup quirks, caveats and limitations specific to particular features in crosstool-NG.
GNU libc locales

GNU libc does not offer a means to cross-compile locale data for the target system. As a workaround, crosstool-NG configures the glibc for the build machine and generates the locales on the build, even though they will be used on the target.

This obviously has caveats:

The build and the target machines must have the same endianness and the same sizes of integer types.
This approach does not work on the build machines not supported by GNU libc. Currently, GNU libc locales are disabled on macOS and Cygwin.

uClibc/uClibc-NG configuration

Before release 1.21.0, it was necessary to supply a configuration file for uClibc/uClibc-ng library. Current crosstool-NG releases can generate this configuration file based on the menu selection. You can still supply your own uClibc configuration file.

Note however that the choices made in crosstool-NG configuration will be applied on top of the uClibc configuration file, to ensure the compatibility of the uClibc with the target components built afterwards. This also allows the user to share the same base configuration file for uClibc and tweak it for different

targets using crosstool-NG configuration.
uClibc with newer GCC

uClibc (not uClibc-ng) produces invalid binaries when compiled with GCC5 or newer (noticed at least on the i686 architecture); applications segfault while starting up when run against the resulting libraries. Use GCC 4.9 or 4.8 when compiling a uClibc based toolchain.

Given that uClibc is essentially unmaintained, this is unlikely to get ever fixed.
Multilib caveats

Crosstool-NG has experimental support for building multilib toolchain. There are a few caveats though:

Buildroot does not accept the toolchains if the produced libraries are scattered across separate subdirectories. For example, on AArch64 the GNU libc installs its dynamic linker into /lib, while the rest of the dynamic libraries are installed into /lib64. Crosstool-NG offers a configuration option to combine the libraries into a single library directory; it is turned on by default for non-multilib builds. For multilib builds, it is not recommended to turn it on unless your multilib configuration uses separate sysroots for all variants. At this time, only the SuperH architecture is known to do that.

Starting with version 2016.02, Buildroot is rejecting the toolchains with /etc/ld.so.conf present in sysroot. Generation of this file is optional in crosstool-NG; however, without this file the cross-ldd helper script will not be able to find the library dependencies residing outside of the default /lib and /usr/lib directories. A fix for this is planned.

On x86, current GNU libc versions share the headers between different multilib variants. Older libc versions had conflicting headers, in particular <ucontext.h>. If you see compilation errors referring to wrong registers for selected multilib variant (e.g., %rip with -m32 flag), the selected version of GNU libc is too old and does not support multilib.

Certain architectures have a fixed set of multilibs in GCC. As a result, if not all of them are supported by the selected C library (glibc, uclibc), the build will fail. This is not a problem with crosstool-NG.

Using crosstool-NG to build Alpha toolchains

The alphaev67-unknown-linux-gnu sample produces errors related to the .eh_frame_hdr section in the C runtime files (crt*.o). The toolchain compiles fine, but may have issues with the generated binaries. If you use this toolchain and encounter any issues, please let us know.
Python scripting in cross GDB

Crosstool-NG offers an option to enable Python scripting in the cross-GDB for the host. This requires the Python headers and libraries for the host to be available. Usually, these come from a python-dev or a similarly named package.

For canadian (and hence, for cross-native) toolchains, this configuration option will result in build failure, unless special steps are taken to place the cross-compiled Python libraries and headers where the compiler for the host will be able to find them. Otherwise, the build will fail as well.
Using crosstool-NG to build Xtensa toolchains

Contributed by: Max Filippov

Xtensa cores are highly configurable: endianness, instruction set, register set of a core is chosen at processor configuration time. New registers and instructions may be added by designers, making each core configuration unique. Toolchain components cannot know about features of each individual core and need to be configured in order to be compatible with particular architecture variant. This configuration includes:

definitions of instruction formats, names and properties for assembler, disassembler and debugger;
definitions of register names and properties for assembler, disassembler and debugger;
selection of predefined features, such as endianness, presence of certain processor options or instructions for compiler, debugger C library and OS kernels;
macros with instruction sequences for saving and restoring special, user or coprocessor registers for OS kernels.

This configuration is provided in form of source files, that must replace corresponding files in binutils, gcc, gdb or newlib source trees or be added to OS kernel source tree. This set of files is usually distributed as archive known as Xtensa configuration overlay.

Tensilica provides such an overlay as part of the processor download, however, it needs to be reformatted to match the specific format required by the crosstool-NG. For a script to convert the overlay file, and additional information, please see this link

The current version of crosstool-NG requires that the overlay file name has the format xtensa_.tar, where CORE_NAME can be any user selected name. To make crosstool-NG use overlay file located at `/xtensa_.tar` select XTENSA_CUSTOM, set config parameter `CT_ARCH_XTENSA_CUSTOM_NAME` to `CORE_NAME` and `CT_ARCH_XTENSA_CUSTOM_OVERLAY_LOCATION` to `PATH`.

The fsf target architecture variant is the configuration provided by toolchain components by default. It is present only for build-testing toolchain components and is in no way special or universal.