Android Build System

From eLinux.org

The Android Build system are described at:
https://android.googlesource.com/platform/build/+/master/core/build-system.html

You use build/envsetup.sh to set up a "convenience environment" for working on the Android source code. This file should be source'ed into your current shell environment. After doing so you can type 'help' (or 'hmm') for a list of defined functions which are helpful for interacting with the source.

Contents

Overview

The build system uses some pre-set environment variables and a series of 'make' files in order to build an Android system and prepare it for deployment to a platform.

Android make files end in the extension '.mk' by convention, with the *main* make file in any particular source directory being named 'Android.mk'.

There is only one official file named 'Makefile', at the top of the source tree for the whole repository. You set some environment variables, then type 'make' to build stuff. You can add some options to the make command line (other targets) to turn on verbose output, or perform different actions.

The build output is placed in 'out/host' and 'out/target' Stuff under 'out/host' are things compiled for your host platform (your desktop machine). Stuff under 'out/target/product/<platform-name>' eventually makes it's way to a target device (or emulator).

The directory 'out/target/product/<platform-name>/obj' is used for staging "object" files, which are intermediate binary images used for building the final programs. Stuff that actually lands in the file system of the target is stored in the directories root, system, and data, under 'out/target/product/<platform-name>'. Usually, these are bundled up into image files called system.img, ramdisk.img, and userdata.img.

This matches the separate file system partitions used on most Android devices.

Some Details

What tools are used

During the build you will be using 'make' to control the build steps themselves. A host toolchain (compiler, linker and other tools) and libraries will be used to build programs and tools that will run on the host. A different toolchain is used to compile the C and C++ code that will wind up on the target (which is an embedded board, device or the emulator). This is usually a "cross" toolchain that runs on an X86 platform, but produces code for some other platform (most commonly ARM). The kernel is compiled as a standalone binary (it does not use a program loader or link to any outside libraries). Other items, like native programs (e.g. init or toolbox), daemons or libraries will link against bionic or other system libraries.

You will be using a Java compiler and a bunch of java-related tools to build most of the application framework, system services and Android applications themselves. Finally, tools are used to package the applications and resource files, and to create the filesystem images that can be installed on a device or used with the simulator.

Telling the system where the Java toolchain is

Before you build anything, you have to tell the Android build system where your Java SDK is. (Installing a Java SDK is a pre-requisite for building).

Do this by setting a JAVA_HOME environment variable.

Specifying what to build

In order to decide what to build, and how to build it, the build system requires that some variables be set. Different products, with different packages and options can be built from the same source tree. The variables to control this can be set via a file with declarations of 'make' variables, or can be specified in the environment.

A device vendor can create definition files that describe what is to be included on a particular board or for a particular product. The definition file is called: buildspec.mk, and it is located in the top-level source directory. You can edit this manually to hardcode your selections.

If you have a buildspec.mk file, it sets all the make variables needed for a build, and you don't have to mess with options.

Another method of specifying options is to set environment variables. The build system has a rather ornate method of managing these options for you.

To set up your build environment, you need to load the variables and functions in build/envsetup.sh. Do this by 'source-ing' the file into your shell environment, like this:

```
$ . build/envsetup.sh
```

You can type 'help' (or 'hmm') at this point to see some utility functions that are available to make it easier to work with the source.

To select the set of things you want to build, and what items to build for, you use either the 'choosecombo' function or the 'lunch' function. 'choosecombo' will walk you through the different items you have to select, one-by-one, while 'lunch' allows you select some pre-set combinations.

The items that have to be defined for a build are:

- the product ('generic' or some specific board or platform name)
- the build variant ('user', 'userdebug', or 'eng')
- whether you're running on a simulator ('true' or 'false')
- the build type ('release' or 'debug')

Descriptions of these different build variants are at http://source.android.com/porting/build_system.html#androidBuildVariants

The build process from a user perspective is described pretty well in this blog post: First Android platform build (http://blog.codepainters.com/2009/12/18/first-android-platform-build/) by CodePainters, December 2009

Actually building the system

Once you have things set up, you actually build the system with the 'make' command.

To build the whole thing, run 'make' in the top directory. The build will take a long time, if you are building everything (for example, the first time you do it).

Build tricks

Seeing the actual commands used to build the software

Use the "showcommands" target on your 'make' line:

```
$ make -j4 showcommands
```

This can be used in conjunction with another make target, to see the commands for that build. That is, 'showcommands' is not a target itself, but just a modifier for the specified build.

In the example above, the -j4 is unrelated to the showcommands option, and is used to execute 4 make sessions that run in parallel.

Make targets

Here is a list of different make targets you can use to build different parts of the system:

- make sdk - build the tools that are part of an SDK (adb, fastboot, etc.)
- make snod - build the system image from the current software binaries
- make services
- make runtime
- make droid - make droid is the normal build.
- make all - make everything, whether it is included in the product definition or not
- make clean - remove all built files (prepare for a new build). Same as rm -rf out/<configuration>/
- make modules - shows a list of submodules that can be built (List of all LOCAL_MODULE definitions)
- make <local_module> - make a specific module (note that this is not the same as directory name. It is the LOCAL_MODULE definition in the Android.mk file)
- make clean-<local_module> - clean a specific module
- make bootimage TARGET_PREBUILT_KERNEL=/path/to/bzImage - create a new boot image with custom bzImage

Helper macros and functions

There are some helper macros and functions that are installed when you source envsetup.sh. They are documented at the top of envesetup.sh, but here is information about a few of them:

- croot - change directory to the top of the tree
- m - execute 'make' from the top of the tree (even if your current directory is somewhere else)
- mm - builds all of the modules in the current directory
- mmm <dir1> ... - build all of the modules in the supplied directories
- cgrep <pattern> - grep on all local C/C++ files
- jgrep <pattern> - grep on all local Java files
- resgrep <pattern> - grep on all local res/*.xml files
- godir <filename> - go to the directory containing a file

Speeding up the build

You can use the '-j' option with make, to start multiple threads of make execution concurrently.

In my experience, you should specify about 2 more threads than you have processors on your machine. If you have 2 processors, use 'make -j4', If they are hyperthreaded (meaning you have 4 virtual processors), try 'make -j6.

You can also specify to use the 'ccache' compiler cache, which will speed up things once you have built things a first time. To do this, specify 'export USE_CCACHE=1' at your shell command line. (Note that ccache is included in the prebuilt section of the repository, and does not have to be installed on your host separately.)

Building only an individual program or module

If you use build/envsetup.sh, you can use some of the defined functions to build only a part of the tree. Use the 'mm' or 'mmm' commands to do this.

The 'mm' command makes stuff in the current directory (and sub-directories, I believe). With the 'mmm' command, you specify a directory or list of directories, and it builds those.

To install your changes, do 'make snod' from the top of tree. 'make snod' builds a new system image from current binaries.

Setting module-specific build parameters

Some code in Android system can be customized in the way they are built (separate from the build variant and release vs. debug options). You can set variables that control individual build options, either by setting them in the environment or by passing them directly to 'make' (or the 'm...' functions which call 'make'.)

For example, the 'init' program can be built with support for bootchart logging by setting the INIT_BOOTCHART variable. (See Using Bootchart on Android for why you might want to do this.)

You can accomplish either with:

```
$ touch system/init/init.c
$ export INIT_BOOTCHART=true
$ make
```

or

```
$ touch system/init/init.c
$ m INIT_BOOTCHART=true
```

Makefile tricks

These are some tips for things you can use in your own Android.mk files.

build helper functions

A whole bunch of build helper functions are defined in the file build/core/definitions.mk

Try grep define build/core/definitions.mk for an exhaustive list.

Here are some possibly interesting functions:

- print-vars - shall all Makefile variables, for debugging
- emit-line - output a line during building, to a file
- dump-words-to-file - output a list of words to a file
- copy-one-file - copy a file from one place to another (dest on target?)

Add a file directly to the output area

You can copy a file directly to the output area, without building anything, using the add-prebuilt-files function.

The following line, extracted from prebuilt/android-arm/gdbserver/Android.mk copies a list of files to the EXECUTABLES directory in the output area:

```
$(call add-prebuilt-files, EXECUTABLES, $(prebuilt_files))
```

Adding a new program to build

Steps for adding a new program to the Android source tree

- make a directory under 'external'
  - e.g. ANDROID/external/myprogram
- create your C/cpp files.
- create Android.mk as clone of external/ping/Android.mk
- Change the names ping.c and ping to match your C/cpp files and program name
- add the directory name in ANDROID/build/core/main.mk after external/zlib as external/myprogram
- make from the root of the source tree
- your files will show up in the build output area, and in system images.
  - You can copy your file from the build output area, under out/target/product/..., if you want to copy it individually to the target (not do a whole install)

See http://www.aton.com/android-native-development-using-the-android-open-source-project/ for a lot more detail.

Building the kernel

The kernel is "outside" of the normal Android build system (indeed, the kernel is not included by default in the Android Open Source Project). However, there are tools in AOSP for building a kernel. If you are building the kernel, start on this page: http://source.android.com/source/building-kernels.html

If you are building the kernel for the emulator, you may also want to look at: http://stackoverflow.com/questions/1809774/android-kernel-compile-and-test-with-android-emulator

And, Ron M wrote (on the android-kernel mailing list, on May 21, 2012):

This post is very old - but nothing has changes as far as AOSP is concerned, so in case anyone is interested and runs into this problem when building for QEMU:

There is actually a nice and shorter way to build the kernel for your QEMU target provided by the AOSP:

1. cd to your kernel source dir (Only goldfish 2.6.29 works out of the box for the emulator)

2. ${ANDROID_BUILD_TOP}/external/qemu/distrib/build-kernel.sh -j=64 --arch=x86 --out=$YourOutDir

3. emulator -kernel ${YourOutDir}/kernel-qemu # run emulator:

Step #2 calls the toolbox.sh wrapper scripts which works around the SSE disabling gcc warning - which happens for GCC < 4.5 (as in the AOSP prebuilt X86 toolchain).

This script adds the " -mfpmath=387 -fno-pic" in case it is an X86 and that in turn eliminates the compilation errors seen above.

To have finer control over the build process, you can use the "toolbox.sh" wrapper and set some other stuff without modifying the script files.

An example for building the same emulator is below:

```
# Set arch
export ARCH=x86
# Have make refer to the QEMU wrapper script for building android over x86
(eliminates the errors listed above)
export
CROSS_COMPILE=${ANDROID_BUILD_TOP}/external/qemu/distrib/kernel-toolchain/android-kernel-toolchain-
# Put your cross compiler here. I am using the AOSP prebuilt one in this example
export
REAL_CROSS_COMPILE=${ANDROID_BUILD_TOP}/prebuilt/linux-x86/toolchain/i686-android-linux-4.4.3/bin/i686-android-
# Configure your kernel - here I am taking the default goldfish_defconfig
make goldfish_defconfig
# build
make -j64
# Run emulator:
emulator -kernel arch/x86/boot/bzImage -show-kernel
```

This works for the 2.6.29 goldfish branch. If anyone is using the emulator with a 3+ kernel I would be like to hear about it.

Retrieved from "http://elinux.org/index.php?title=Android_Build_System&oldid=388096"

Category: Android
