Smith Firmware Deployment Toolkit

Jason Lefley

Revision 2 – 09/19/2014

Introduction

The Smith Firmware Deployment Toolkit is a collection of scripts and other files used to generate a packaged form of a particular version of the smith firmware suitable for distribution. A firmware package includes not only the core software itself but also all dependencies and supporting configuration. The toolkit also includes provisions for generating an image that contains the development dependencies. This document describes important aspects of the toolkit along with a procedure outlining how the toolkit can be used to generate a firmware package and a development/build system image. The toolkit is entirely contained in the deploy directory within the //depot/ACG/Smith Perforce directory.

Definitions

This section defines important terms used throughout this document.

* **Core component**: A program that when executed completes domain specific tasks, such as controlling the 3D printing process. The smith binary (smith), local web server (smith-server), web client (smith-client), and configuration utility (smith-config) are core components.
* **Firmware**: One or more software applications designed to run on the target system along with the supporting drivers, libraries, and configuration files. In the case of smith, the firmware consists of the core components, Linux drivers, configuration files, and required libraries and system-level applications.
* **Firmware package**: An archive file used to distribute a particular firmware version to the target system(s).
* **Target system**: The computer that firmware will be deployed to and run on. It could be a BeagleBone, integrated electronics board, or other similar device.
* **Target file system**: A collection of directories and files that make up the majority of the file system of the target system. It contains all drivers, static configuration files, libraries, and applications needed to support execution of the Linux kernel and applications contained in the firmware. Once deployed, the target file system is entirely read-only. These directories and files are generated and reside on the build system and are built into an image during the packaging process.
* **Main storage**: A storage partition on the target system where firmware images, dynamic configuration files, log files, user data, and other artifacts of a running system are stored. This part of storage is writable and is created when a new target system is setup. While a firmware image contains the majority of the target system’s file system, the main storage contains the remainder.
* **Firmware image**: A single file containing an entire compressed target file system. A firmware image is located in main storage and loaded by the target system on boot.
* **Development system**: The computer on which firmware and deploy toolkit source code is authored, managed, and edited. This system typically has the source code under some sort of version control and hosts IDEs and/or other development tools.
* **Build system**: The computer on which the core components and firmware packages/images are built. The build system has the same architecture as the target system or has the ability to cross compile or otherwise build software compatible with the target system  
    
  Note: The target/development/build system definitions describe roles one or more computer may play in the process. It is possible, for instance, for a single physical device to play the role of development and build system. Although for convenience, it usually makes more sense to have an environment set up for managing and editing source code on one machine (development) and then another set up to build/package (build).

Directory Overview

This section describes key directories and files located under the deploy directory.

* build\_package\_scripts contains scripts used by build\_package.rb.
  + install.sh copies core components of firmware from the build host to the target file system when a package is built. This script is invoked by build\_package.rb and subsequently runs on the build system.
* configs contains configuration files specifying parameters used by omap-image-builder to build a target file system as a collection of directories and files on the build system.
  + smith-development.conf is used to generate the file system used in development and for building images/packages and core components.
  + smith-release.conf is used to generate the file system used to construct release images.
  + smith-common.conf contains configuration options that are common to both the release and development configurations.
* initramfs contains scripts and configuration files for building the initial RAM file system (initramfs). The custom initramfs image is installed on a target system when it is first set up and provides the mechanism for locating and loading a firmware image as part of the boot process.
  + build\_initramfs.sh is used to rebuild the initramfs image (initrd.img). This script is intended to run on the build system.
* omap-image-builder is a clone of the omap-image-builder project hosted on GitHub (<https://github.com/RobertCNelson/omap-image-builder>). The BeagleBone community uses this project to generate the default image included with an off-the-shelf BeagleBone.
  + RootStock-NG.sh is the primary entry point of omap-image-builder and takes a configuration file from the configs directory as an argument. This script is invoked by build\_package.rb and subsequently runs on the build system.
* setup contains files that populate a target system during initial setup along with a script to complete the setup process in an automated fashion.
  + boot contains files that are copied to the boot partition when a target device is set up. The contents are broken down into development, release, or common to both groups and versioned by subdirectory corresponding to the kernel version.
    - dtbs contains the device tree binary file for the am335x BeagleBone black architecture.
    - initrd.img is the initial RAM file system image.
    - MLO is the first stage boot loader image.
    - u-boot.img is the second stage boot loader image.
    - uEnv.txt is the configuration file for the u-boot environment.
    - zImage is the Linux kernel image.
  + eeprom contains an EEPROM dump that has the identifier for the BeagleBone Black.
  + main contains files that are copied to the main partition when a target device is set up. It includes the var directory skeleton as a tar file. This directory is also populated with a firmware image when build\_package.rb is run.
  + setup\_eeprom.sh flashes the EEPROM with the contents of bbb-eeprom.dump (contained in the eeprom directory). It is intended to run on the target system.
  + setup\_emmc.sh formats, partitions, and populates the boot and main storage regions of a target system. It is intended to run on the target system.
* target contains scripts and configuration used by omap-image-builder to build a target file system
  + hooks contains scripts called before and after the chroot tasks are completed. They facilitate installation/copying of files between the build system and the target file system. These scripts are invoked by omap-image-builder. Hook scripts specific to the development and release build configurations are present in this directory.
  + config contains configuration files, drivers, libraries, and applications that are copied into the target file system by a before hook script. The files are categorized by build configuration. smith-common contains files that are copied to both the development and release images. During the build process, the root of the appropriate directory is mapped to the root of the target file system such that config/smith-\*/etc/foo.conf is copied to /etc/foo.conf with respect to the target file system.
  + chroot contains scripts that are run within a chroot jail such that the root of the file system available in the context of a chroot script is pointing to the directory containing the target file system. This scenario allows tasks to be completed as if running on the target system even though the execution is taking place on the build system thereby giving operations that require write-access to the target file system an opportunity to run. These scripts are invoked by omap-image-builder. Release and development specific scripts are present.
* u-boot contains a patch for the u-boot source code that enables booting with an empty EEPROM
  + hardcode-bbb.patch is the source code patch. The patch hardcodes the identifier for the BeagleBone Black rather than attempting to read it from the EEPROM.
  + readme.txt explains how to apply the patch and build u-boot
* build\_package.rb is used to build a firmware package containing a particular version of the firmware. Files specific to the smith-release configuration are used in this process. The packaging process consists of building a target file system, copying the core components from the build system to the target file system, and generating a firmware image and package. Once a target file system has been built by calling build\_package.rb, build\_package.rb can be run again to update the previously built target file system with new versions of the core components without the need to rebuild the target file system again. When called without any arguments, the script will read the version string to use in the resulting image and package names from the smith Ruby gem currently installed on the build system. This value can be overridden by specifying a version string as the first argument when calling build\_package.rb.
* build\_development\_image.sh is used to build a development image using files specific to the smith-development configuration. This script outputs an .img file that can be flashed to a micro-SD card and then the micro-SD card can be used to boot into a build system where core components and firmware packages can be built. Unlike the build\_package.rb script, this script will always generate a new root file system when invoked. Since the resulting file system allows configuration changes and installation of additional packages, it is important to make relevant configuration changes within the toolkit so the development environment can be tracked.
* deploy.sh is used to synchronize the contents of the deploy directory residing on the development system to the build system using rsync if it is available. This functionality can also be achieved with scp, etc. In practice this script is more useful than scp as it allows for a “mirror” copy operation, which only updates out of date or non-existent files on the build system side along with deleting files from the build system side that have been deleted from the development system side.

Packaging Process Overview

A firmware package is composed of a squashfs file system image file (firmware image) and an additional file containing the MD5 checksum of the image. A firmware package can be loaded on a target system through the upgrade mechanism to install the contained applications, libraries, drivers, and configuration files for that particular firmware version on the target system. A firmware package for a particular version can be built using the following procedure:

1. Install/build the core components on the build system. The respective build method (NetBeans, make, rake, etc.) of a specific component is used to complete this step.
   1. For the C++ component this means building through NetBeans running on the development system with the build system configured as a remote build host.
   2. For the Ruby component this means executing rake deploy on the development system with the build system connected.
2. Copy the deploy directory from the development system to the build system. This can be accomplished with an SCP utility or the deploy.sh script if rsync is available on the development system. The rest of the procedure will assume that the deploy directory has been copied to the /root directory (i.e. /root/deploy exists and contains the deploy toolkit).
3. Invoke build\_package.rb on the build system:  
     
   cd /root  
   ./deploy/build\_package.rb
4. Respond appropriately to the prompts presented by build\_package.rb. Answer y when asked to build a target file system if one does not yet exist, changes have been made to the smith-release.conf configuration file, one or more scripts in target/chroot or target/hooks have been modified, or one or more files within target/config have been added, removed, or modified. Otherwise, it is not necessary to build another target file system. Note: building a target file system from scratch takes approximately 45 minutes and requires the build system to have internet connectivity.
5. When build\_package.rb completes, the new firmware package will be contained in the /root/deploy/deploy directory with a name of smith-<VERSION>.tar. This file can be distributed as a firmware upgrade to target systems that are already setup. At this point, the newly built firmware image will also have been copied to setup/main/firmware for use in new target system setup.
6. Optionally, a new target system can be setup by booting the target system in a way that allows access to the contents of the setup directory and the block device, which will hold the boot and main partitions. Then setup/setup\_emmc.sh is executed. Typically this arrangement can be achieved by booting the new target system from an SD card containing the built firmware package and then writing to the on-board eMMC storage using setup\_emmc.sh. If the EEPROM is empty, it is necessary to execute setup/setup\_eeprom.sh and reboot before setting up the eMMC storage.
7. Troubleshooting note: If there are issues in the build process check that the line endings were not changed by a version control system and also try deleting the entire deploy directory from the build system and re-copying it to the build system.

Development/Build System Image Building Overview

1. Copy the deploy directory from the development system to the system where the development image will be built. The only requirement of this system is that it has the same architecture as the system the image will be used on. This can be accomplished with an SCP utility or the deploy.sh script if rsync is available on the development system. The rest of the procedure will assume that the deploy directory has been copied to the /root directory (i.e. /root/deploy exists and contains the deploy toolkit).
2. Invoke build\_development\_image.sh:  
     
   cd /root  
   ./deploy/build\_development\_image.sh
3. When build\_development\_image.sh completes, the new image will be located in the /root/deploy/deploy. The image can then be flashed to an SD card using dd or another flashing/copy utility.
4. Troubleshooting note: If there are issues in the build process check that the line endings were not changed by a version control system and also try deleting the entire deploy directory from the build system and re-copying it to the build system.