Embedded Linux

by Libero Scarcelli (liberoscarcelli.net)

Table of Contents

Embedded Linux ToolChain

A **toolchain** is a set of packages needed to create software for a specific device. Toolchains can be either downloaded and installed or they can be built using a toolchain generator. A toolchain is the very first thing that is needed when building an embedded Linux machine, as it will build:

1. The bootloader.
2. The kernel.
3. The root filesystem.

Toolchains are usually based on either:

1. GNU's Not Unix! (**GNU**) project & the compiler system GCC (gcc, g++, gfortran and many more).
2. Low Level Virtual Machine (**LLVM**) project and the compiler front end Clang (C, C++, Objective-C and Objective-C++).

Bear in mind that:

1. Clang only compiles C-like languages while relies on other technologies for Ada, Fortran, Java, etc…
2. GCC is a more robust but memory intensive system.
3. However, as progresses are made every day, this might change soon!

Native And Cross ToolChains

Toolchains can be either:

1. **Native**, which is when development and target systems are similar.
2. **Cross**, which is when development and target systems are different. Almost all embedded Linux development is done on cross toolchains.

While the native methodology requires software updates to be strictly controlled, as development and target systems have to be synchronized, cross development requires large amount of work, as all libraries have to be cross compiled.

ToolChains Have To Match The Target Machine Specifications

Toolchains have to be built according to the target machine specifications, such as:

1. CPU type, such as x86_64, MIPS and ARM.
2. Endianness, such as big or little endian although some CPUs can use both.
3. Floating-point hardware support, which specifies whether the system needs additional libraries to work with floating-point numbers.

4. Application Binary Interface (ABI), which specifies the low-level hardware dependent standard that defines functions calling conventions, system calls behaviour, binary format of program libraries, object files and much more.

The Standard GNU ToolChain

The standard GNU toolchain includes:

1. Compilers for C, C++, Assembly and Java that produce assembler code to be passed to as (GNU assembler).
2. Binutils to turn assembly code into binary (as), link objects to create executable files (ld) and much more!
3. A C library implementing POSIX APIs able to talk to the kernel.
4. Debugging tools.

The output of the following command is an example of how the GNU project identifies toolchains:

```
$> gcc -dumpmachine
x86_64-linux-gnu
```

Each of the above fields says something about the platform:

1. Describes the CPU and it is usually equal to ARM, MIPS or **x86_64**.
2. Defines the vendor and it can be omitted.
3. Describes the kernel type and it is always equal to **linux**.
4. Describe the operating system and might contain details about the C library and the ABI. Here it only contains the string **gnu**.

C libraries are wrapper functions for system calls from which they are often named. Programs use C libraries to talk to the kernel:

- glibc is the GNU C library and it is the best implementation of POSIX API, although a large one.
- eglibc was a project forked from glibc, optimized for embedded systems, obsolete and no longer maintained.
- musl libc is a good C standard library for systems with low RAM or not enough storage. It is released under the MIT License.
- uClibc-ng is a project forked from uClibc and it is designed for embedded systems and mobile devices runnning µClinux. Highly compatible with glibc.

How To Choose The Right ToolChain

Those willing to build an embedded system from scratch can choose their toolchain among the following three options:

1. A pre-built toolchain.

2. A toolchain created from scratch before the installation of the system.
3. A toolchain generated using an embedded build tool.

Using a **pre-built** toolchain is the easiest option, although less flexible than the other ones. The following pre-built toolchains are very popular: Debian-based, Yocto Project, Mentor Graphics, TimeSys, MontaVista and Linaro. Always make sure that the chosen toolchain:

1. Comes with the preferred C library.
2. Is easy to update.

Building a toolchain from **scratch** is not an easy task even though several very good projects, such as “Cross Linux From Scratch”, already exist. A simplified approach consists of using crosstool-NG which comes with many useful scripts driven by a front-end. To install **crosstool-NG**:

```
$> git clone https://github.com/crosstool-ng/crosstool-ng.git
$> cd crosstool-ng
$> git checkout crosstool-ng-[latest_version]
#
# Check the dependencies for your build system.
#
$> ./bootstrap
$> ./configure --enable-local
$> make
$> make install
```

If the previous steps succeeded, the reader will have a working installation of crosstool-NG ready on the build machine. The following steps show how to use crosstool-NG to build a toolchain for the emulator **QEMU**:

```
$> ./ct-ng distclean
# List all available configurations
$> ./ct-ng list-samples
# Choose the correct configuration file. Here the one for QEMU is chosen
$> ./ct-ng arm-unknown-linux-gnueabi
# In "Paths and misc options" untick "Render the toolchain read-only"
$> ./ct-ng menuconfig
$> ./ct-ng build
```

Should the build process succeed, the toolchain will be located under:

```
~/x-tools/arm-unknown-linux-gnueabi/bin
```

In this folder, tools such as compiler, debugger, linker, all **renamed** with the **toolchain identity**, are located. For example, the toolchain version of ‘ld’ will be:

```
~/x-tools/arm-unknown-linux-gnueabi/bin/arm-unknown-linux-gnueabi-ld
```

At this stage the user should start familiarising with the new environment. Modifying the **PATH** environmental variable would also be beneficial:

```
PATH=${HOME}/x-tools/arm-unknown-linux-gnueabi/bin/:$PATH
```

Although the procedure above is the one that QEMU users should follow, those who own a physical board can go through the same procedure, making sure to:

1. Choose the correct configuration file.
2. Set the PATH variable according to their toolchain.

How To Use The ToolChain

This chapter will help the reader to familiarize with the toolchain built in the previous chapter. The GNU C and C++ compiler that came with the toolchain can be used to retrieve some system settings:

```
*-gcc -v
```

What Build, Host And Target Are For ToolChains

In the output produced by the line above, the section “Configured with…” can be found, which contains the following details:

1. **−−build=** defines the platform on which build processes run.
2. **−−host=** defines the platform on which the compiled code will be running. If building support libraries for other systems, this parameter will have to be set accordingly: the compiler and those libraries will use different values.
3. **−−target=** is only used when the software being built is itself a building tool targeting different platforms. In fact, to build a cross-compiler, only −−build= and −−host= need to be set.

Native, Cross, Cross Native, Crossback And Canadian Cross Builds

The following relationships clarify the previous concept even further:

- If build == host == target, the build is native.
- If build == host but target is different, the build is cross.
- If host == target but build is different, the build is crossed native.
- If build == target but host is different, the build is crossback.
- If they are all different, the build is Canadian cross.

Therefore, a crossed native build would use a cross-compiler to build native packages for a different system, a crossback build would use a cross-compiler to build packages for the build machine and a

Canadian cross build would use a cross-compiler to build another cross compiler to build packages for a third platform.

More GCC Options For x86_64 Systems

Going back to the output of the previous gcc command, the meaning of the some of the remaining options for a x86_64 system, is the following:

- **--prefix=** specifies the location of the executable gcc.
- **--with-sysroot=** tells GCC which directory contains shared objects for C library, static library archive files for C library, header files for libraries, localization and internationalization information, utility programs for target, the utility ldconfig to optimize the library loading paths.
- **--enable-languages=** which says whether C and C++ are enabled.
- **--with-float=** can be “hard”, generating opcodes for the FPU, or “soft”, using libraries to emulate the FPU.
- **--disable-sjlj-exceptions** which disables SetJump LongJump method of exception handling as this is slower than DWARF-2 (DW2) EH.
- **--enable-__cxa_atexit** that says that the system is using __cxa_atexit, rather than atexit, to register C++ destructors for local statics and global objects.
- **--disable-libmudflap** disables mudflap run time checker, removed in GCC 4.9 as superseded by Address Sanitizer.
- **--disable-libgomp** disables GNU Offloading and Multi Processing Runtime Library.
- **--disable-libssp** disables support for stack smashing protector. Remember that libssp only detects SBO, it does not prevent it.
- **--disable-libquadmath** & **--disable-libquadmath-support** mainly disable Quad-Precision Math Library Application Programming Interface for applications requiring results in higher than double precision.
- **--disable-libsanitizer** disables AddressSanitizer to detect SBO and memory corruption bugs.
- **--disable-libmpx** disables the Intel Memory Protection Extensions for checking pointer references at runtime.
- **--with-gmp=** defines the location of GMP library for arbitrary precision arithmetic, operating on signed integers, rational and floating-point numbers which is very useful for cryptography.
- **--with-mpfr=** defines the location of MPFR library for multiple-precision floating-point computations with correct rounding.
- **--with-mpc=** defines the location of MPC library for arithmetic of complex numbers with arbitrarily high precision and correct rounding.
- **--with-isl=** defines the location of ISL library for manipulating sets and relations of integer points bounded by linear constraints.
- **--enable-lto** enables Link Time Optimisation to reduce code size.
- **--with-cpu=** specifies the target core.
- **--enable-threads=posix** enables POSIX threads.
- **--enable-long-long** enables type “long long”.
- **--enable-target-optspace** optimizes the library for code space instead of code speed.

- **−−enable-plugin** enables GCC plugins.
- **−−enable-gold** builds the Gold linker.
- **−−disable-nls** enables Native Language Support.
- **−−disable-multilib** disables multiple target libraries to support different target variants and calling conventions should not be built.

Analyzing The C Library

The next step should be analyzing the **C library** and its components, in order to familiarize with this important part of the system:

- **libc** is the main library that contains, for example, the POSIX implementations of “printf” and “close”. This library is so important that smart compiler such as GCC can import it automatically when the programmer forgets to do so.
- **libpthread** is the library for POSIX thread functions. It can be linked by using the -lpthread flag.
- **libm** is the library for math functions. It can be linked by using the -lm flag.
- **librt** is the library for POSIX real time extensions, which includes asynchronous Input/Output and shared memory. It can be linked by using the -lrt flag.

Statically And Dynamically Linked Libraries

Libraries can be either statically or dynamically linked to those files that need them. **Static linking**:

1. Static library contents physically exist in the executable files that link them.
2. Executable files that use static libraries increase in size.
3. Removes any compatibility issue as each executable file includes all libraries it needs.
4. Increases execution speed.
5. Tends to create slower build processes.
6. Usually, the keyword -static forces the operating system into statically linking all libraries.

To create **static libraries** and **link** them into an executable file:

```
*-gcc -c mystaticlib1.c
*-gcc -c mystaticlib2.c
*-ar rc libmystaticlib.a mystaticlib1.o mystaticlib2.o
*-gcc myprog.c -lmystaticlib -I../usr/include -L../libs -o myprog
```

Where “-L” appends the directory to the list of directories to be searched for library files and “-I” appends the directory to the list of directories to be searched for header files. On the other hand, **dynamic linking**:

1. Performs on-the-fly linking as programs are executed.
2. Libraries are loaded in memory only once, therefore, linking process might speed up if the code is already present in memory.

3. Might trigger compatibility issues if a library is changed without recompiling the one into memory.
4. Slower execution time.
5. Smaller executable files.

To create **dynamic libraries** and **link** them to an executable file:

```
*-gcc -fPIC -c mydynlib1.c
*-gcc -fPIC -c mydynlib2.c
*-gcc -shared -o libmydynlib.so mydynlib1.o mydynlib2.o
*-gcc myprog.c -lmydynlib -I../usr/include -L../libs -o myprog
```

Where "-L" appends the directory to the list of directories to be searched for library files and "-I" appends directory to the list of directories to be searched for header files. The linker will look for libmydynlib.so into /lib and /usr/lib and alternative directories can be specified modifying the shell variable **LD_LIBRARY_PATH**.

Shared Library Version Number And Interface Number

The shared **library version number** is a string that is appended to the library name to define its version and it is not included in the symbolic link name used to load the library. For example, the library libmylib.so.1.0.10 will have a symbolic link called libmylib.so. When another minor fix is released as libmylib.so.1.0.11, only the symbolic link will need to be modified to point at libmylib.so.1.0.11.

The **interface number** encodes the interface number created when the library was built. Its format is:

```
[library_name].so.[interface_number]
```

This is useful when major changes are deployed. Should the new version libxyz.so.2.0.10 of libxyz.so.1.0.10 be released, backward compatibility would break. However, old programs would not see the new library, as they were linked to libxyz.so.1.*.*

How Libraries Look Like On Linux Systems

The following schema shows how dynamic and static libraries will look like on Linux systems:

```
# Archive for static linking
libxyz.a
# Dynamic linking
libxyz.so → libxyz.so.1.0.10
# Load the library at runtime
libxyz.so.1 → libxyz.so.1.0.10
# Actual library
libxyz.so.1.0.10
```

While the first two are used by build machines, the remaining two are needed by the runtime.

MakeFile, AutoTools And Cmake

Makefiles are text files that tell **make**, a build automation tool, what to do. Makefiles contain rules and have the following layout:

```
CC=gcc
CFLAGS=-I.
HelloW: HelloW.c HelloWFunc.c
    $(CC) -o HelloW HelloW.c HelloWFunc.c
```

The file above tells make to compile a C program and an external function by using gcc as C compiler. CFLAGS is used to tell the compiler that header files will be located into the local directory.

Cross Compiling Using Make And Makefiles

To cross compile software packages using make or a makefile, the variables **CROSS_COMPILE** and **ARCH** usually need to be set:

```
$> export CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
$> export ARCH=arm64
$> make
```

More simply:

```
$> make CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- ARCH=arm64
```

Similarly, the makefile will need to include the following:

```
ARCH=arm64
CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
```

Building Systems With Autotools

Autotools is a collection of tools that are used to build systems. The main aim of autotools is providing users with standardized build procedures. Programmers will need to learn a single tool that is able to compile packages using different versions of compilers while loading different header files and different versions of libraries. Autotools consists of:

- GNU Autoconf which generates the configure script that checks the host system and creates the makefile and header files from templates.

- GNU Automake which generates Makefile.in templates from Makefile.am templates allowing programmers to write makefiles in a higher-level language.
- GNU Libtool which creates portable compiled libraries.
- GNULib which is a portability library.

Moreover, autotools:

- Depends on Bourne shell, therefore it is slow.
- Depends on make.
- Backward and forward compatibility depend on a wrapper script.
- Generated scripts are generally very complex and large.
- Depends on M4 which is unknown to many developers.
- Might not be able to communicate with packages that implement their own configuration and build systems.

Usually, to configure, build and install packages using autotools the following steps are required:

```
$> ./configure
$> make
$> make install
```

It is possible to **analyze** packages configurations by inspecting their .pc file, which are used for tracking packages installations, as follows:

```
$> export XTOOLS=$HOME/x-tools
$> cat $($XTOOLS/arm-unknown-linux-gnueabi/bin/arm-unknown-linux-gnueabi-gcc -print-sysroot)/usr/lib/pkgconfig/expat.pc
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include
Name: expat
Version: 2.2.6
Description: expat XML parser
URL: http://www.libexpat.org
Libs: -L${libdir} -lexpat
Cflags: -I${includedir}
```

The same output can be obtained by using the tool **pck-config**, making sure its configuration directory **PKG_CONFIG_LIBDIR** is properly set:

```
$> export XTOOLS=$HOME/x-tools
$> export PKG_CONFIG_LIBDIR=$($XTOOLS/arm-unknown-linux-gnueabi/bin/arm-unknown-linux-gnueabi-gcc -print-sysroot)/usr/lib/pkgconfig
$> pck-config expat --libs --cflags
```

Building Systems With CMake

CMake does not depend on make and the Unix shell, therefore, it is able to run on Windows too. In fact, it is designed to work with native build environments. In the CMake system, very simple configuration files called CMakeLists.txt are stored into each project (sub)directory and they are used to produce build files. CMake can interact with Microsoft Visual Studio, XCode, Eclipse CDT, MSBuild, Make and many more. The following is an example of the extremely intuitive syntax used to build CMakeLists.txt files:

```
cmake_minimum_required(VERSION 3.9)
project(HWorld)
add_executable(HWorld HWorld.c Utils.c)
install(TARGETS HWorld DESTINATION bin)
```

CPack is the packaging system fully integrated with CMake, although it can work by itslef. CPack can create the following types of archives: deb, rpm, gzip, NSIS and Mac OS X packages. Usually, to configure, build and install a pakcage with CMake the following commands are required:

```
$> ./ccmake .
$> make
$> make install
```

Common Issues That Affect Build Systems

The following are common issues that affect the most commonly used build systems:

- Configuration scripts might use information retrieved by pkg-config disregarding −−host override.
- Some scripts might try to run cross compiled code, therefore, they will fail.
- Target machine floating point format might be incompatible or byte-ordering might be different.
- Some programs might not work in a cross compiled environment.

Boot Process

Booting the Linux kernel on embedded systems is a process that consists of the following four stages, in the given order:

1. ROM.
2. Secondary Program Loader (SPL).
3. Tertiary Program Loader (TPL).
4. Kernel level.

The ROM Step

The **ROM** contains code that is executed right after reset or power-on. This software is shipped with the chip, it is often proprietary and cannot be replaced. The main purpose of this step is inizializing the devices for the next stage. The ROM loads code from specific locations into SRAM as DRAM cannot be used yet. DRAM requires code to initialize the memory controller which cannot be done by the ROM. As the SRAM is usually too small to contain the bootloader, an intermediate loader has to be used: the Secondary Program Loader (SPL) which may be either Open Source or proprietary. SPL is a small binary that fits in SRAM and loads the regular bootloader (such as U-Boot or BareBox) into system RAM. At the end of this first step, the SPL is present in the SRAM and the ROM points at the beginning of this code.

The SPL Step

SPL configures the memory controller and other components so that the Tertiary Program Loader (TPL) can be loaded into DRAM. If SPL comes with file system drivers, it can read files such as "u-boot.img" from the disk. SPL usually does not allow for any user interaction. At the end of this stage DRAM contains TPL and SPL can jump to it.

The TPL Step

When the **TPL** stage is reached, full bootloaders such as U-Boot or BareBox are also running. Users can access a command line interface to select a newer kernel, perform maintenance tasks and much more. At the end of this step the kernel is stored in memory, ready to be started. Embedded bootloaders usually disappear once the kernel is loaded in order to free up memory.

The UEFI Standard And Booting Process

The firmware of many ARM and Intel platforms is based on Universal Extensible Firmware Interface (**UEFI**). Many **bootloaders** compatible with this standard exist, therefore, the reader can choose any of them. For example, either systemd-boot or Barebox would be a very good choice. UEFI booting process consists of similar steps to those already discussed, as it can also be divided into four steps:

1. UEFI initializes the hardware.
2. The firmware does the job of SPL as it initializes the memory controller and other components, so that an EFI boot manager can be loaded from the EFI system partition (ESP) or from the network via PXE boot. ESP must be FAT16 or FAT32, while the boot manager must be located somewhere like /efi/boot/bootx64.efi.
3. TPL has to load the Linux kernel and an optional RAM disk into memory.
4. Kernel level.

The Bootloader And The Kernel

The **bootloader** always passes some information to the kernel:

- Detected hardware such as CPU clock speed and amount of RAM.
- Machine number for those platforms (PowerPC and ARM) that do not support device trees.

- Kernel command line parameters.
- Size and location of device tree (optional) otherwise the system will have to discover platform details at runtime or they have to be hardcoded in the kernel.
- Size and location of initramfs (optional) which is a set of directories (normally found on root filesystems) compressed into an archive. This is mounted as "/" by the kernel to run "/init" which is useful to kernel modules.

Device Trees To Simplify Physical Devices Management

Device trees, as defined in the OpenBoot standard, are tree data structures with nodes that describe physical devices. They can be either loaded by the bootloader which then pass them to the kernel usually through R2 register or embedded into the kernel. Device trees are saved into .dts files and they are compiled using the Device Tree Compiler to produce a Device Tree Blob.

ARM systems did not use device trees, but they would store information inside **ATAGS**, whose address was saved into the R2 register, ready to be passed to the kernel instead. Machine type was also passed to the kernel as integer, stored into the R1 register. PowerPC would simply pass the kernel a pointer containing the address of an information structure.

How To Read And Create Device Trees

The following is an excerpt of a typical **.dts** file used to describe a made up board:

```
{
  model = "MY Board Electronics";
  /*Drivers compatibility*/
  compatible = "ti,am33xx";
  #address-cells = <1>;
  #size-cells = <1>;
  cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    /*dev name + @ + address*/
    cpu@0 (
      /*Drivers compatibility*/
      compatible = "arm,cortex-a8";
      device_type = "cpu";
      reg = <0>;
    };
  };
  memory@0x2000 {
    device_type = "memory";
    reg = <0x2000 0x20 0xDA00 0x10>;
  };
};
```

The snippet above, the **cpus** container node defines all CPUs present on the system. In this instance, as only a single **cpu** node is defined, the board will be equipped with only one CPU. Moreover, as each node is assigned a unique id, the only CPU running on this board will be identified by the cpu@0 label. The properties #address-cells and #size-cells are defined in the snippet, as all addressable devices have to set initialize these in order to interpret the reg property:

- **#address-cells** is the number of 32-bit cells required to encode the address field in reg.
- **#size-cells** is the number of 32-bit cells required to encode the size field in reg.

The property **reg** is the address of the device's resources within the address space defined by its parent. It consist of couple of fields, the first one being the address and the second one being the size. In the eaxmple above, memory@0x2000 would have a 32-byte block at offset 0x2000 and a 16-byte block at offset 0xDA00. Machines that require 64-bit addressing may set #address-cells and #size-cells to two. The following properties are also important:

- **interrupt-controller** set the current node to receive interrupt signals.
- **interrupts** defines the node containing the list of interrupt specifiers the meaning of which depends on the binding for the interrupt controller device.
- **interrupt-parent** defines the link (phandle) that points to the interrupt controller for the current node.
- **#interrupt-cells** defines how many cells are in an interrupt specifier for this interrupt controller.

The following is one more snippet defining **interrupt controller**s and it should be self-explanatory:

```
...
compatible = "comp1,comp2";
#address-cells = <1>;
#size-cells = <1>;
/*The controller is intc*/
interrupt-parent = <&intc>;
...
intc: interrupt-controller@10150000 {
  compatible = "arm,pl190";
  reg = <0x10150000 0x2000>;
  interrupt-controller;
  /*defines how to specify interrupts*/
  #interrupt-cells = <2>;
};
...
serial@1a1f3000 {
  compatible = "arm,pl011";
  reg = <0x1a1f3000 0x2000 >;
  /*defines interrupts for this device*/
  interrupts = < 2 0 >;
};
```

Device trees have to be passed to the kernel in their binary representation as .dtb files obtained using the program **dtc** (device tree compiler). Bear in mind that this utility does not return verbose debug information, therefore, working with custom device tree files is difficult. The device tree compiler can also be used to unpack device tree blob files .dtb., so that they can be reverse engineered and expanded. In conclusion, device trees:

- Only describe the hardware present on the platform and how this works but they do not define how hardware configuration should be used.
- Are platform independent, therefore, they can be considered as stable structures.
- Might make the kernel bigger and might slow the boot process down.
- Should be kept minimal and contain as little details as possible, as large device trees might make the binding process hard.
- May be extended but they should never be modified.

Introduction To U-Boot

U-Boot is a **primary boot loader**, used in embedded devices, originally designed for PowerPC, it is now available for a number of architectures including: x86, ARM, MIPS, 68k, SuperH, PPC, RISC-V, MicroBlaze, Blackfin and Nios.

How U-Boot Works

Initially loaded by the ROM or the BIOS, U-Boot can work on very limited amount of resources as it can be split into stages: a stripped down version of U-Boot (SPL) is loaded first to perform basic hardware configuration, in order to start the bootloader full version. It comes with a command line which users can use to boot a particular kernel, manipulate device trees, download files, work with environmental variables and much more. It requires users to specify the memory locations of all objects it has to access: copying a ramdisk or jumping to a kernel image is done through memory addresses.

Downloading And Building The U-Boot Code

Regardless whether the user is using a physical board or an emulator (QEMU), the very first step is getting U-Boot up and running so that its console can be accessed to launch commands to load the kernel. As U-Boot can be automated, users do not always need to type these commands. The following are the steps required to **build** U-Boot:

```
$> git clone git://git.denx.de/u-boot.git
$> cd u-boot
$> git checkout [your_chosen_branch]
```

As the directory “configs” contains all available **configurations**, the reader should choose the one that matches the platform the software is being built for. Also, the reader should make sure all environment variables are set correctly, as shown in the previous chapters, before proceeding:

```
#Configure U-Boot
$> make CROSS_COMPILE=[board_platform] [config_file]
#Build U-Boot
$> make CROSS_COMPILE=[board_platform]
#Example for QEMU
$> make CROSS_COMPILE=arm-unknown-linux-gnueabi- qemu_arm_defconfig
$> make CROSS_COMPILE=arm-unknown-linux-gnueabi-
```

The steps above are quite straightforward, although certain boards might require the user to set additional variables to have make compiling the code correctly. If everything goes well, the procedure should create the “u-boot” and “u-boot.bin” files in the “u-boot” directory. The following are important files to familiarize with:

- u-boot is the executable file in ELF format, used with debuggers.
- u-boot.bin is the executable that runs on the device.
- u-boot.map is the symbol table.
- u-boot.img is similar to the previous one, but it includes the U-Boot header.
- u-boot.srec is the executable in Motorola S-Record format and is to be used over serial connections.
- MLO is the secondary Program Loader and it is built only if needed.

Test U-Boot On QEMU

To **test** the U-Boot build with the **QEMU** emulator, the reader should move into the u-boot root directory, next:

```
$> qemu-system-arm -M virt -nographic -no-reboot -bios u-boot.bin
```

Bear in mind that, although virtual boards such as **versatilepb** and **vexpress-a9** are still around, they are obsolete and all code should be tested using **virt** instead. Once inside the QEMU prompt, the user may launch some commands as the following:

```
=> printenv
arch=arm
baudrate=115200
board=qemu-arm
board_name=qemu-arm
=> reset
```

Test U-Boot On A Physical Board

Should the reader own a **physical board**, this section will show how to make U-Boot files accessible to the hardware. The reader should choose any support, such as SD cards, USB mass storage or serial interfaces, to store the code which is going to be read by the ROM and create two partitions, as follows:

Min Size	Max size	Type	Mount Point	Content
64 Mb	128 Mb	FAT32	/media/[user]/boot	Bootloader
1 Gb	2 Gb	ext4	/media/[user]/rootfs	Root file system

Should a MLO file required by the board, it should be copied to the boot partition. The same should be done with u-boot.img. The board can now be powered on and accessed using the preferred terminal program, making sure to set the port at 115200 bps with no flow control.

The program **mkimage** is used to create images for U-Boot to contain a Linux kernel, a root file system, a firmware, device tree blob files and much more. It is possible to create legacy images:

```
$> mkimage -A arm -O linux -T kernel -C gzip -a 0x80001000 -e 0x80001000 -n 'Linux' -d zImage uImage
```

Or current Flattened Image Tree (FIT) images, which are designed to be more flexible and safer:

```
$> mkimage -f kernel.its kernel.itb
```

The following command can be launched into a U-Boot prompt in order to **load** files from FAT partitions:

```
=> fatload mmc 0:1 83000000 uImage
```

- mmc is the interface type.
- 0 defines the first device, counting from zero.
- 1 defines the first partition, counting from one.
- 0x80001000 is the chosen RAM area and it has to be empty.
- uImage is the file to load.

The system administrator will need to make sure the chose address ‘0x83000000’ is not overwritten while the kernel is copied into ‘0x80001000’, as defined by the mkimage command.

The process can occur over a network using the **TFTP** protocol: the user will set local and server addresses and then the image will be loaded by specifying RAM address and file name:

```
=> setenv ipaddr 192.168.100.2
=> setenv serverip 192.168.100.1
=> saveenv
=> tftpboot 83000000 uImage
```

Now images are ready to be programmed into **NAND**. Let’s use some Error Correction Coding in order to prevent corruption:

```
=> nandecc hw
```

Let’s clean up NAND:

```
=> nand erase 300000 400000
```

Finally, the following line gets the image at 0x83000000 and **writes** 0x400000 bytes to NAND flash at the address 0x300000:

```
=> nand write 83000000 300000 400000
```

Similarly, written bytes can be retrieved from NAND using the following command which reads 0x400000 bytes from offset 0x300000 from beginning of NAND, storing them into RAM address 0x83000000:

```
=> nand read 83000000 300000 400000
```

As no **kernel** has been built yet, next chapters will have to cover this step again. However, a kernel stored into memory can be **booted** by using the following:

```
=> bootm [kernel] [ramdisk] [dtb]
```

If no initramfs is provided, then “-” can be used followed by the address pointing at the device tree blob:

```
=> bootm 83000000 - 84000000
```

How To Automate U-Boot

U-Boot can be **automated** by creating scripts as follows:

- Store all commands into a text file.
- Convert the text file into U-Boot image using **mkimage**.
- Download the file using TFTP on the target machine.
- Use the **source** command to execute the script.

It is also possible to create a variable with **setenv** to store all commands. Next, the variable is executed using the **run** command.

U-Boot Falcon Mode

So far, the boot procedure has been following these steps: ROM -> SPL -> u-boot.bin -> kernel. U-Boot **Falcon** mode makes the SPL loading the kernel, removing the need for u-boot.bin, making the whole process faster. However, enabling this configuration is not an easy task, as it might require the user to build U-Boot after having modified a large number of properties stored into several configuration files.

Introduction To BareBox

BareBox is a project derived from U-Boot and originally named **U-Boot v2**. The aim of BareBox is combining the U-Boot and Linux technologies as this product does not require the user to work with memory addresses.

Downloading And Building The BareBox Source Code

To retrieve the source code

```
$> git clone git://git.pengutronix.de/git/barebox.git
$> cd barebox
$> git checkout [your_chosen_branch]
```

The directory arch/${ARCH}/configs contains all available configurations, therefore, the reader should choose the one that matches the platform for which this package will be built. If BareBox is being built for a physical board that requires the **MLO**, this will have to be built first. The MLO is also called x-loader and it is used when the SRAM is too small to contain the entire code needed to load the kernel. After having made sure that the environment variables are correctly set:

```
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform] [mlo_config_file]
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform] menuconfig
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform]
#Example for ARM Cortex-A8
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
am335x_mlo_defconfig
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- menuconfig
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
```

MLO configuration files are located into arch/${ARCH}/configs and their name usually contain either the string mlo or xload or anything similar. The reader should make sure to choose the right configuration. Moreover, the menuconfig step can be skipped if no customization is needed.

The **actual bootloader** can now be built following a similar procedure that points at a different configuration file:

```
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform] [config_file]
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform] menuconfig
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform]
#Example for ARM Cortex-A8
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
am335x_defconfig
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- menuconfig
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
```

Again, the user should make sure to choose the right configuration file and skip menuconfig step, should no further customization be needed.

Using BareBox On A Physical Board

To make the BareBox files accessible to the board, one should choose any support such as SD cards, USB mass storage or serial interfaces to store the code which is going to be read by the ROM. Create two partitions, as follows:

Min Size	Max Size	Type	Mount Point	Content
64 Mb	128 Mb	FAT32	/media/[user]/boot	Bootloader
1 Gb	2 Gb	ext4	/media/[user]/rootfs	Root file system

Next, if a **MLO** file was created, it has to be copied to the boot partition while the same should be done with **barebox.bin**.

After having reset the board, you will get a prompt similar to the Linux one: many well known commands such as **ls**, **cp**, **rm** and **mount** will work just fine. For those who choose to load the code from a SD card, the following line of code can be used to mount the first partition:

```
barebox:/ mount /dev/mmc0.0 /mnt
```

Once everything is ready, the following command can be used to **boot** any BareBox image, ARM Linux zImage or U-Boot uImage:

```
barebox:/ bootm [/path/image]
```

The following line specifies a device tree at boot time

```
barebox:/ bootm -o [/path/dtb_file] [/path/image]
```

To pass parameters to the kernel parameters, the following format can be used:

```
barebox:/ global linux.bootargs.root="[parameters]"
barebox:/ bootm [/path/image]
```

A complete sequence of commands to boot a system with BareBox could be:

```
barebox:/ global.bootm.oftree=/mnt/am335x-boneblack.dtb
barebox:/ global linux.bootargs.root="root=/dev/mydev rootwait”
barebox:/ bootm /mnt/zImage
```

Differences Between U-Boot And BareBox

In conclusion, to those who are not sure which bootloader to install, the following section might be helpful:

- As U-Boot is being used by a larger number of installations, it is very well maintained.
- U-Boot is well known for its high level of configurability and flexibility.
- U-Boot requires deep board knowledge.
- U-Boot command line forces users into working with memory addresses rather than file names.
- U-Boot is harder to configure: many files have to be edited in order to change a configuration.
- In BareBox, environment variables and scripts cannot be mixed up.

Choosing A Linux Kernel

This chapter helps the reader understanding the basics of the Linux kernel in order to be able to choose the right one for any device.

What The Linux Kernel Does

The Linux kernel:

1. Manages resources such as devices, memory, file system and processes.
2. Acts as interface to the hardware.
3. Provides users with an API with a useful level of abstraction to user space programs.

The kernel is a trusted element that has complete and unrestricted access to the underlying hardware, therefore, it runs in **kernel space**. Actions that can be performed without any special privilege run in **user space** instead. Transitions from user space to kernel space are triggered by any of the following:

1. A CPU instruction, such as a system call or a breakpoint.
2. A signal sent by a process or the OS.
3. A system call.
4. A software interrupt, such as a page fault or an exception.
5. Hardware interrupt.

However, although each platform executes a different set of steps to transition from user space to kernel space, the basic interactions between the two spaces are similar across all systems and can be summarized by the following:

- Application → User Space
- C Library → User Space
- System Call Handler → Kernel Space
- Generic Services → Kernel Space
- Device Drivers → Kernel Space
- Hardware → Kernel Space

The C Library, The User Space And The Kernel

The **C library** is the primary interface between the user space and the kernel, as it is able to translate user level functions into system calls. The **system call interface** uses architecture-specific technologies, such as traps or software interrupts, to switch the CPU level from user to kernel mode, enabling the program to access CPU registers and all memory addresses. The **system call handler** dispatches each call to the right kernel subsystem such as the memory manager or filesystem code.

A System Is Not Just Its Kernel

The kernel is just one of the essential components of the Linux installation, together with the C Library, basic command tools and much more. Moreover, the Linux kernel can be coupled with:

- The GNU user space to create the GNU/Linux operating system.
- The Android user space to create the Android operating system.
- BusyBox user space to create embedded systems
- Much more.

Operating systems derived from the defunct **BSD**, such as FreeBSD, OpenBSD and NetBSD are structured differently: kernel, toolchain and user space are combined into a single code base. Linux is a kernel while *BSD are complete products.

The Linux Kernel Versioning System

Each kernel is defined by its own version number. Before July 2011, the accepted format was the following:

2	**6**	**39**	**1**
Major version	Minor version	Revision	Stable version

An odd minor version number would indicate a developer release, while an even one, would mean that the kernel was ready to be installed on end users computers. Every now and then a fix would be pushed, which would increase the stable number. As the minor version number was later dropped, the numbering jumped from 2.6.39 to 3.0.

A **full cycle** of kernel development starts with the opening of the merge window: the development community then pushes all code that is deemed to be stable into mainline kernel. The window stays open for approximately **two weeks**. Next, Linus Torvalds closes the window and produces **release candidates** which are labelled appending -rc plus the version number. During this time, users test the kernel and submit bug reports and fixes. When everything is ready, the kernel is **released**. All kernel releases changelogs at available here: http://kernelnewbies.org/LinuxVersions. After the release of a mainline kernel, the code is then pushed to the stable tree, which is managed by Greg Kroah-Hartman. A new development cycle can now begin on mainline kernel, while bug fixes will be stored into the stable tree. Releases that mainly publish bug fixes are called point releases and are marked by a third number, the rightmost one (3.61.**2**). As already explained, before version three,

four numbers were used. Moreover, some kernels are labelled as long term and maintained for 2 years or more.

How To Retrieve The Linux Kernel Source Code

To clone the Linux kernel repository:

```
$> git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
$> cd linux-stable
$> git checkout [linux_version]
```

Configuring The Linux Kernel

Although the previous chapter has already shown the steps required to download the Linux kernel, it would be beneficial to list them again:

```
$> git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
$> cd linux-stable
$> git checkout [linux_version]
```

The Linux Kernel Folder Structure And Content

Should the reader prefer to retrieve the Linux kernel as tarball files, they are available at https://cdn.kernel.org/pub/linux/kernel. Regardless the methodology used to download the code, the kernel folder will contain, at least, the following subfolders:

- arch for architecture-specific files.
- drivers for device drivers.
- fs for code that supports all filesystems supported by the kernel. Additional binary formats, such as the Java one, can be defined here.
- include for kernel header files.
- init for kernel start-up code
- kernel for core functions such as locking, timers, power management, debugging, tracing and scheduling.
- mm for files data structures that are used by the system to manage memory-related issues.
- net for the repository for the socket abstraction and the network protocols.
- scripts for many useful scripts
- tools for useful tools for profiling and tracing.

Introduction To The Linux Kernel With Kconfig

By configuring the kernel the reader can reach maximum flexibility. The configuration system is called **Kconfig** and it is documented in the “Documentation/kbuild” directory. The configuration process can be text based using **make config**, ncurses-based with a pseudo-graphical menu using make **menuconfig**, Qt based using make **xconfig**, GTK based using make **gconfig** and so on. The

command **make help** can be used to explore all possibilities. In this page it is assumed the user will choose the most common option:

```
$> make ARCH=[platform_arch] menuconfig
```

When the chosen configurator starts, it reads the main Kconfig file located in the appropriate **arch** subdirectory. When the user does not set the ARCH variable during the configuration step, the system defaults to the local machine architecture. The main Kconfig file contains references to additional configuration files, declared using the following syntax:

```
source "path/Kconfig"
```

These external files can contain references to external Kconfig files too: they are all processed by Kconfig.

How Kconfig Works And Its Syntax

The **syntax** that Kconfig understands is simple: each line starts with a keyword which may be followed by multiple arguments. The **config** keyword starts a new configuration, while the following lines define the attributes of this configuration:

```
config LCD
  bool "Make all module available"
  depends on SCREEN
  help
     Write something useful to the users...
```

Kconfig files allow the following **data types**:

1. Bool: yes and no.
2. Tristate: yes, no and module.
3. String.
4. Hexadecimal.
5. Integer.

The keyword **prompt** is used to display messages to the user:

```
prompt "Install all drivers"
```

As type definition may also accept user, the following:

```
bool "Install all drivers"
```

Is equivalent to:

```
bool
```

```
prompt "Install all drivers"
```

The **default** keyword assigns default values to a configuration. Dependencies for specific values can be specified using the **if** keyword instead:

```
default y if LCD
```

The code above means that the default value is yes if LCD option is also activated. Default values are always equal to “no” otherwise.

The keyword **depends** defines menu dependencies. Multiple dependencies are defined using two ampersands. This keyword affects all options within the menu:

```
depends on LCD
bool "SCREEN"
default y
```

In the example above, the second and third line of code also inherit the LCD dependency defined by the first line. Standard dependencies restrict the upper limit while reverse ones restrict the lower one by using the keyword **select**.

```
config USE_A_STACK
  bool "Graphics mode on?"
  select B_GRAPHICS_NEEDED
```

The lines above mean that if USE_A_STACK radio button is checked, the system will also need B_GRAPHICS_NEEDED. This type of dependencies can also be manage using the keyword **imply**, which allows the symbol’s value to be changed by a direct dependency or via a visible prompt:

```
config A
  tristate
  imply C
config C
  tristate
  depends on B
```

In the configuration above, if A = “n” and B = “y” then C = “n”, but C can also be changed to “m” or “y” while if A = “m” and B = “y” then C = “m”, but C can also be changed to “y” or “n”.

The keyword **visible** can turn a menu visibility on and off:

```
visible if [expression]
```

The keyword **help** defines help text whose end is determined by the indentation level. The following two are equivalent:

```
help
  Some help text
---help---
  The same help text
```

The position of the **menu** entry in the tree is determined in two ways:

- Using the keywords **menu** and **endmenu**.
- Analyzing all dependencies, as a menu entry that depends on a previous one can be displayed as submenu of it. One of these two conditions must also be true:
  - If “parent = n” the child has to be invisible.
  - The child is visible only if the parent is visible.

The following snippet shows how to use menu and endmenu:

```
#All blocks are submenu of "Network tuning"
menu "Network tuning"
  depends on NET
  ...
#Inherits dependencies of the menu such as “NET”
  config NETTOOLS
  ...
endmenu
```

This is example shows how the position of a menu can change because of its dependencies:

```
config MODS
  bool "Enable loadable module"

#This depends on MODS, visible if MODS!=n
config MODVER
  bool "Version information"
  depends on MODS
#The comment is visible when MODS=n
comment "Module support disabled"
  depends on !MODS
```

Creating The Linux Kernel Configuration File

After having correctly configured all Kconfig files, it is possible to run the configuration procedure, using any of the available tool, like the following:

```
$> make ARCH=[platform_arch] menuconfig
```

At the end of this step, the configuration file required by the kernel build will be saved. By default it will be named .config. How does the **.config** file drive the kernel build? In a few words:

- The main Makefile reads the .config file and performs tasks while analyzing all subdirectories recursively.
- The main Makefile reads the one located in the usual "arch/$ARCH" directory in order to gather architecture-specific information.
- All makefiles present on each subdirectory will carry out commands passed from above.

As configuring the kernel from scratch is a big job, it is possible to use known Kconfig files stored into **arch/$ARCH/configs**. The .config file can then be created using the usual syntax. The following command reads a configuration compatible to many ARMv7-A machines and creates a .config accordingly:

```
$> make ARCH=arm multi_v7_defconfig
```

Working With The Linux Kernel Version Number

The version of a previously downloaded Linux kernel can be printed by launching the following command, which will return the same string than the command **uname**:

```
$> make ARCH=[kernel_arch] kernelversion
```

When the kernel is modified, the **CONFIG_LOCALVERSIO**N variable has to be modified to contain the string that will be appended to the standard Linux version. When this is done, as the command above will still return the same output, the following has to be used instead:

```
$> make ARCH=[kernel_arch] kernelrelease
```

Embedded Devices And Linux Kernel Modules

While Desktop Linux distributions do use **modules** extensively, embedded devices do not, as their hardware and software configurations are more stable and known at the time the kernel is built. Therefore, embedded kernels, should always be built without any modules, unless it is crucial to:

- Reduce the boot time by deferring the loading of non-essential drivers.
- Avoid licensing issues with proprietary modules.
- Reduce the compilation time when it is necessary to support a very large number of drivers.

Building The Linux Kernel

Kbuild is a set of make scripts designed to build the Linux kernel according to the information read from the **.config** file. Kbuild can:

- Work out all dependencies by reading all makefiles in each subdirectory.

- Build a kernel with statically linked components.
- Create a device tree binary.
- Create kernel modules.

An Example Of Kconfig Makefile

The following is an example of **Makefile** stored under “drivers/char”:

```
obj-y += mem.o random.o
obj-$(CONFIG_TTY_PRINTK) += ttyprintk.o
```

While the first line forces the build of mem.c and random.c, the following one simply treats CONFIG_TTY_PRINTK as a variable to be replaced:

- If CONFIG_TTY_PRINTK == y, then ttyprintk.c is compiled as built-in.
- If CONFIG_TTY_PRINTK == m, then ttyprintk.c is built as a module.
- Otherwise, ttyprintk.c is not built at all.

Building The Kernel Creating The Right Output

The list of the files to be built depends on what the bootloader expects. Therefore, while old versions of **U-Boot** might need a **uImage** file, more recent ones can load **zImage** files using the bootz command. Other bootloaders usually do require a **zImage** file. Moreover, when the target is a **x86** system, a **bzImage** file is usually required. The following line of code can be used to build a zImage:

```
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform] zImage
```

Building A uImage For ARM

Building **uImage** files for **ARM** with multi-platform support is tricky, as from Linux 3.7 onward, a single kernel binary can run on multiple platforms. This was done in order to have less kernels targeting more ARM devices. This is a difficult scenario as the bootloader passes the device tree and/or the machine number to the kernel, which then should select the correct platform. But as the memory location and the relocation address code might be different for each platform, this approach does not work. Bear in mind that the relocation address is hardcoded into the **uImage** header by the **mkimage** program. To solve this problem, one must read the memory address stored into **zreladdr-y**, which can be found into the **Makefile.boot**, which then needs to be passed to the **LOADADDR** variable. The “Makefile.boot” will be stored into a location similar to the following one:

- arch/arm/mach-[your_SoC]/Makefile.boot

Therefore, if the “Makefile.boot” shows that the value of “zreladdr-y” is the address 0x80009000, the command line to create an “uImage” file compatible with multi-platform images might be the following:

```
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- LOADADDR=0x80009000 uImage
```

The Kernel Linux Build Output

Each kernel build will generate the following two files, located in the top level directory:

- **vmlinux** which is a statically linked executable file in ELF format, which can be used for debugging, if it was built with the CONFIG_DEBUG_INFO flag on. It has to be made bootable by adding moltiboot header, bootsector and setup routines.
- **System.map** which contains the symbol table in a readable form. It shows associations between symbols names, such as variables names and functions names and memory addresses. This file is very useful for debugging.

As most **bootloaders** cannot handle **ELF** files correctly, the vmlinux file, which contains the Linux kernel, needs to be further processed to create binary files in a format that bootloaders can understand:

- **Image** is a “vmlinux” converted into raw binary format.
- **zImage** is, for most platforms, a combination of the compressed “Image” and some stub code that decompresses and relocates it. In the PPC architecture instead, this is a just a compressed “Image” file which is decompressed by the bootloader.
- **uImage** is a “zImage” file with a 64-byte U-Boot header.

The following command can be used to produce zImage files:

```
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform] zImage
```

Next, all binaries are to be copied to **arch/$ARCH/boot**. For example, for an ARM Cortex-A8 board, this will build the zImage:

```
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- zImage
```

Building The Device Tree File

All **device trees**, as multi-platform builds can define many of them, have to be built now. The make target **dtbs** is to be used to accomplish this task. To do this, make reads the rules listed in **arch/$ARCH/boot/dts/Makefile**. The following creates device trees:

```
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform] dtbs
```

Building The Kernel Modules

If any module is required, it can be built now. As the reader might guess, the make command that creates modules is the following:

```
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform] modules
```

Compiled modules will be created with the **.ko** (kernel object) suffix, as this is the extension of kernel modules loaded by modprobe since kernel version 2.6. In fact, before Linux 2.6, a user space program would read ".o" files to link them to the kernel, in order to create the final binary image. However, as from Linux 2.6 onward the linking is done by the kernel, the extension ".ko" was used to flag these new ELF files that were passing additional information to the kernel. After the build, these ".ko" files will be located all over the place in the directory source, therefore, the following line can be used to place each file into the right place:

```
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform]
INSTALL_MOD_PATH=[staging_directory] modules_install
```

Should the user choose $HOME/rootfs as staging directory, all modules will be located into $HOME/rootfs/**lib**/**modules**.

Make Targets To Clean The Build Environment

Should the reader need to rebuild multiple times using the same source folder, the following make targets can be used to clean the environment to make sure that all artefacts of previous builds are removed:

- **clean** which simply removes object files and most intermediates.
- **mrproper** which removes all intermediate files, including .config.
- **distclean** which is similar to mrproper but it also deletes all backup files created by the editor, patch files and anything else related to the development of the Linux kernel.

Example Of Sequence Of Commands To Build The Linux Kernel

Finally, this is the complete sequence of commands that is required to build the Linux kernel for an **ARM Cortex-A8** board:

```
$> cd linux-stable
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- mrproper
$> make ARCH=arm multi_v7_defconfig
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- zImage
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- modules
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- dtbs
```

This is what is required for **QEMU**:

```
$> cd linux-stable
$> make ARCH=arm CROSS_COMPILE=arm-unknown-linux-gnueabi- mrproper
$> make ARCH=arm versatile_defconfig
$> make ARCH=arm CROSS_COMPILE=arm-unknown-linux-gnueabi- zImage
$> make ARCH=arm CROSS_COMPILE=arm-unknown-linux-gnueabi- modules
```

```
$> make ARCH=arm CROSS_COMPILE=arm-unknown-linux-gnueabi- dtbs
```

Testing The Linux Kernel

Should the reader decide to boot the kernel that was just built, an **error** similar to the following will be generated:

---[end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)]---

The reader should not be surprised, as this is the expected behaviour at this stage. The next chapters will explain what this error means and how to fix it.

Booting The Linux Kernel

As already explained, booting the kernel built in the previous chapter, will produce an error similar to the following:

---[end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)]---

The Boot Process And Root File System

In fact, the kernel needs to mount a **root file system** executing a program into it, in order to transition from kernel initialization to user space. The kernel can mount either the **real file system** on a block device or an **initial ramdisk**, which is a methodology by which a temporary root file system is loaded into memory. The code that controls this process is located into the **init/main.c** file and it starts with the **rest_init** function, according to the following:

start_kernel() –>	rest_init() –>	kernel_init()
PID = 0	PID = 0	PID = 1

The **start_kernel** function opens the door to the architecture-independent section of the kernel startup process. More in particular:

Name	Tasks	Calls
head.S	This is the entry into the kernel. Architecture specific initialization code required, for instance, to create page table entries, initialize virtual memory and memory management unit.	start_kernel()
start_kernel()	This is the kernel entry point executed at the address defined by LOAD_PHYSICAL_ADDR (if not relocatable) while the kernel will be loaded at CONFIG_PHYSICAL_START. It initializes the console and many subsystems.	setup_arch()
setup_arch()	Responsible for initial, machine	rest_init()

	specific initialization procedures.	
rest_init()	Creates a kernel thread passing kernel_init() as the entry point.	schedule(), cpu_idle()
schedule()	Kickstarts the task scheduling.	
kernel_init()		do_basic_setup(), init_post()
do_basic_setup()	At this stage CPU subsystem, memory and process management are all up. It manages the initialization of hardware and software interrupts, timers, kernel subsystems, built-in devices and much more.	
init_post()	This is the last step of the Linux booting process and it tries to run the following, in the given order: /sbin/init, /etc/init, /bin/init and /bin/sh in order to turn the current kernel thread into the user space init process.	

Although platform might differ greatly when it comes down to the kernel initialization process, the schema above explains how the code that set the system up works. If a **ramdisk** is present, **kernel_init** tries to run the program **/init**, which creates and initialize the user space. If the previous step **fails**, the same function tries to mount a file system by calling **prepare_namespace** which is defined in **init/do_mounts.c**. For this step to work, the user has to pass the correct block device to the kernel using the syntax:

root=/dev/[disk_name][partition_n]

Any eMMC and SD card will be identified by:

root=/dev/[disk_name]p[partition_n]

Therefore, the first partition on a SD card will be:

root=/dev/mmcblk0p1

If this mount succeeds, the system tries to run the following ones in the given order, stopping at the first one that works:

- /sbin/init
- /etc/init
- /bin/init
- /bin/sh

Passing Parameters To The Kernel

The initial command to be executed by the kernel can be changed by using the following boot time parameters:

- **init=** to define the init program to run from a mounted file system.

- **rdinit=** to define the init program to run from a ramdisk.

Passing **parameters** (bootargs) to the kernel can be done in many different ways:

- Using the correct bootloader option.
- Modifying CONFIG_CMDLINE to have these arguments hard-coded in the kernel.
- Modifying the device tree to hard-code any bootargs.

The following table shows a list of the most commonly used bootargs, while the complete list is available in **Documentation/kernel-parameters.txt**.

Name	Description
debug	Set console level to 8: all messages shown
quiet	Set console level to 0: only emergency messages shown
panic	Defines what to do when the kernel panics
init=	Init program to run from the mounted file system
rdinit=	Init program to run from ramdisk
ro	Mount the root device as read-only
root=	Device to mount the root file system
rootdelay=	Seconds to wait before mount the root device
rootwait	Wait indefinitely for the root device to be detected
rootfstype=	File system type for the root device
rw	Mounts the root device as read-write

The table above mentions the concept of message level which is important to **debug** and maintain the kernel. In fact, printing messages is always the easiest way to debug software packages. The C function **printk** does for the kernel what **printf** does for the userspace: these messages can be later displayed using the Linux command **dmesg**. The function “printk” works by writing messages on the **__log_buf** ring buffer, therefore, older messages get overwritten once the buffer fills up. The size of this buffer can be changed by modifying **CONFIG_LOG_BUF_SHIFT**, which is usually located in **init/Kconfig** file, while the amount of bytes read by “dmesg” can be changed by using its “-s” option. Moreover, the function “printk” has an optional prefix string that defines the loglevel of the message being logged:

```
printk(LOG_LEVEL "Write something\n");
```

These messages are categorized according to their importance, with zero being the highest, as the table above shows.

Booting The Kernel Using QEMU

Should the reader be curious to load the kernel into a **QEMU**, the following can be useful:

```
$> export QEMU_AUDIO_DRV=none
$> qemu-system-arm -m 256M -nographic -M virt -kernel [zImage] -append
"console=ttyAMA0,115200" -dtb [dtb]
```

Booting The Kernel Using A Physical Board

The following instead is the procedure to boot the kernel with BeagleBone Black:

- Plug the microSD card with the U-Boot installation into the reader.
- Plug the card reader to the build PC.
- Copy the “zImage” and .dtb file to the “boot” partition on the memory device.
- Unmount the card reader and plug it into the BeagleBone Black.
- Start a terminal emulator able to talk to the board.
- Power on the board and be prepared to press space bar when U-Boot messages appear.
- At this stage the user should get the U-Boot prompt.
- Load “zImage” from the boot partition of mmc and place it into the given starting address:

```
=> fatload mmc 0:1 0x80007fc0 zImage
```

- Do the same with the device tree blob file:

```
=> fatload mmc 0:1 0x80F80000 am335x-boneblack.dtb
```

- Tell the system to use the first UART device for the console output:

```
=> setenv bootargs console=ttyO0
```

- Boot the kernel without an initrd image as ‘-‘ is specified

```
=> bootz 0x80007fc0 – 0x80F80000
```

Shells And Command Line Utilities

The kernel now needs a root file system or an initramfs obtained by either:

- Getting a pointer from the bootloader.
- Mounting the block device that was specified on the kernel command line by using the **root=** kernel parameter.

Creating The Root File System

How to create the root file system the kernel needs? The following roadmap will show how this can be achieved:

- Create a minimal file system to get a shell prompt.
- Add scripts able to install additional programs, configure user permissions, configure network adapters and complete the basic setup.

After this the system would be ready to execute the first program which is usually the init executable.

Any root file system will need to contain the following components:

- **daemons** are background programs that are not under control of interactive users. They are created by either a process forking a child process that is then adopted by init or by the init itself starting another process.
- **init** is the daemon that starts everything else and it is the ancestor of all processes as it adopts all orphaned processes.
- **configuration files** usually stored as text file under the **/etc** directory, they control the behaviour of all daemons, including init.
- **shell** is a program that takes input from the user and executes commands.
- **shared libraries** linked by most programs.
- **device nodes** are special files that allow applications to interact with devices by using drivers via system calls. They can be either character special files, therefore unbuffered, or block special files, therefore buffered. Nodes can be created using mknod system call.
- **procfs** is a pseudo file system that presents information about processes and other operating system related information in a hierarchical file-like structure. It is usually mapped as **/proc** and it is used to get and set kernel parameters at runtime.
- **sysfs** is a pseudo file system that presents information about many kernel subsystems, hardware devices and their drivers in a hierarchical file-like structure. It is usually mapped as **/sys** and it is used to get and set kernel parameters at runtime. Since the release of kernel version 2.6, much of the information has been migrating from "/proc" to "/sys".
- **kernel modules** located into **/lib/modules/** if the kernel is configured to use modules.

It is possible to combine many of the previous components into a statically linked program. Assuming this program is called "prog":

1. The monolithic program "/prog" will start first replacing "init".
2. The program will be specified using the kernel command line "init=/prog".

This configuration should be implemented when a high level of security is needed as the OS will not be able to start anything else apart from "/prog".

When designing the **directory layout**, users are free to implement whichever directory structure they prefer. In fact, Android and Linux distributions come with completely different directory layouts. Whichever the chosen structure is, the first step should always be the creation of the staging directory, which for instance may be named "rootfs":

```
$> mkdir ~/rootfs
$> cd ~/rootfs
$> mkdir bin sbin dev lib etc home proc sys usr var tmp
$> mkdir usr/bin usr/sbin usr/lib
$> mkdir -p var/log
$> sudo chown -R root:root *
```

Adding A Command Shell To The System With BusyBox

The next step would be choosing the **shell** that will run on the embedded system. Any of the following would be a valid choice:

- **ash** is based on the Unix Bourne shell and it is much smaller than bash. It is the default shell on FreeBSD, NetBSD, MINIX and many other Linux systems. It used to be the default shell on Android until its version 4.0, when it was replaced by the Korn shell.
- **bash** is a superset of the Unix Bourne shell with many extensions and advanced features unique to this shell, called bashisms.
- **hush** is a very small shell that can be run on devices with very limited memory. This shell comes with no support for I/O redirection or pipes, therefore, many commands have to include additional arguments to make up for this limitation.

How to install the chosen shell onto an embedded systems? This will be explained soon.

Linux shells allow users to launch **programs** with some flow control being able to pass information between programs. As all shells need utilities to be able to work effectively, users would have to face two major issues:

- The amount of disk space that this collection of programs would need to be installed.
- The difficulty of tracking down and cross-compiling the code of each one of the these programs.

Users of embedded systems looked for and found a solution tailored to their unique needs. In fact, **BusyBox** is one of the available solutions to the previous problem, as this package combines stripped down versions of selected UNIX utilities, called applets, into a single executable. For example, BusyBox comes with its own versions of **init**, **ash**, **hush**, **vi**, **dd**, **sed**, **mount** and many more. To use BusyBox, one needs to type the name of the applet after the name of the main application:

```
$> busybox [applet] [arg1] [arg2] ...
```

Therefore, the following line shows the content of a text file:

```
$> busybox cat mytextfile.txt
```

As the standard installation process can create soft links, the users will be able to omit the initial **busybox**. While the build and deployment procedures for this application will be covered shortly, let’s take some time to dig more into the BusyBox architecture:

- Each applet exports its main function following the “[appletname]_main” format. For example, “rmdir” exports “rmdir_main” in “coreutils/rmdir.c”.
- The BusyBox “main” function redirects all calls to the correct applet according to the command line arguments that will be parsed and analyzed by “libbb/appletlib.c”.

Downloading And Building BusyBox

To download BusyBox and build it:

```
#For tarball archives: http://busybox.net/downloads
$> git clone git://busybox.net/busybox.git
$> cd busybox
$> git checkout [bb_version]
$> make distclean
$> make defconfig
$> make menuconfig
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform]
$> make ARCH=[board_arch] CROSS_COMPILE=[board_platform] install
```

For example, to build BareBox for the BeagleBone Black board, the last two command lines will be:

```
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
$> make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- install
```

The **menuconfig** step might be skipped if no fine tuning is required. For example, to change the installation directory just pass the **CONFIG_PREFIX** with the desired value to make when running the installation step.

ToyBox As An Alternative To BusyBox

ToyBox is a very good alternative to BusyBox which implements the BSD license instead of GPL. ToyBox aims to comply with standards such as POSIX-2008 and LSB 4.1 rather than mirroring the GNU project. It has been included with Android since version 6.0. While the procedure to build and install ToyBox is similar to the BusyBox one, the git repository and .tar files are located at the following addresses:

- git -> https://github.com/landley/toybox.git
- tar -> http://landley.net/toybox/downloads

Bear in mind that the default installation directory is “/usr/toybox” and this behaviour can be quickly modified by passing the preferred value to the **PREFIX** variable.

The Root File System And Initramfs

One of the following three methodologies can be used to **transfer** the staging directory created during the previous chapter, into the target system:

- **initramfs** (Initial Random-Access Memory File System) is a file system image that is loaded into RAM by the bootloader. This is a temporary root file system, only present in memory, that can be used to initialize the system.

- **disk image** is a copy of the root file system prepared to be copied onto the target storage device. Commands such as **dd**, **mkfs** and **genext2fs** might be used to create this image.
- **network file system** can be used by implementing a NFS server: the file system is mounted over the network by the target at boot time.

Introduction To Initramfs

The **initramfs** is a compressed cpio archive, which is an older and simpler utility than tarball which can be easily managed by the kernel. It requires the kernel option **CONFIG_BLK_DEV_INITRD** to be enabled. Although many architectures do not neeed an initramfs file, some other systems do. Initramfs files are mainly used to:

- Carry out work that would be too hard for the kernel.
- Load modules necessary to the boot procedure.
- Provide users with a minimalistic rescue shell.

Shifting tasks out of the kernel **simplifies** the work of system programmers and administrators as the kernel will not need to be rebuilt every time the is changed. An initramfs file can be created as any of the following:

- A **standalone** cpio archive. This is the most flexible option, although some boot loaders might not be able to boot two files.
- A cpio archive **embedded** into the kernel.
- A **device table** which the kernel build process.

Building An Initramfs As A Standalone CPIO Archive

Before starting creating an initramfs file as a **standalone** cpio archive the reader should make sure that the staging directory, called “rootfs” contains all required files:

```
$> cd ~/rootfs
$> find . | cpio --format=newc -ov --owner root:root > ../initramfs.cpio
$> cd ..
$> gzip initramfs.cpio
$> mkimage -O linux -A arm -T ramdisk -d initramfs.cpio.gz uRamdisk
```

Should the initramfs file be too big, one can:

- Shrink BusyBox and ToyBox installation by removing all unnecessary applets and libraries.
- Remove all unnecessary drivers and functions from the kernel.
- Statically rebuild packages such as BusyBox.
- Use uClibc-ng or musl libc instead of glibc.

To test this initramfs file with QEMU:

```
$> export QEMU_AUDIO_DRV=none
```

```
$> qemu-system-arm -m 256M -nographic -M versatilepb -kernel zImage -append
"console=ttyAMA0 rdinit=/bin/sh" -dtb versatile-pb.dtb -initrd initramfs.cpio.gz
```

Should anything go wrong during the boot process:

- Make sure the kernel QEMU is trying to load was compiled by the right toolchain with the correct parameters that would match the virtual machine that is being used.
- Use **lsinitramfs -l** to double check the initramfs file: broken links and missing files are quite a common issue.
- Make sure initramfs contains all required libraries and support files.
- Check for QEMU out of memory errors.
- Check the QEMU virtual machine documentation out.

To test this initramfs with BeagleBone Black:

```
=> fatload mmc 0:1 0x80201000 zImage
=> fatload mmc 0:1 0x80F01000 am335x-boneblack.dtb
=> fatload mmc 0:1 0x81001000 uRamdisk
=> setenv bootargs console=ttyO0,115200 rdinit=/bin/sh
=> bootz 0x80201000 0x81001000 0x80F01000
```

Should anything go wrong during the boot process, it would be possible to go through the same troubleshooting steps that were listed for QEMU.

Building An Initramfs As A CPIO Archive Embedded Into The Kernel

To create initramfs files as cpio archives **embedded** into the kernel simply rebuild the kernel setting the value of **CONFIG_INITRAMFS_SOURCE** to the full path of the uncompressed cpio archive:

- Run make menuconfig in the kernel folder.
- Modify the “General setup -> Initramfs source file(s)” option.

To test this initramfs:

- For QEMU, just omit the ramdisk file.
- For BeagleBone Black, do not fatload the ramdisk and replace the second parameter of “bootz” with a minus sign, “-“.

Using Device Tables As Initramfs

Device tables define device nodes, files, directories and links that go into archives or file system images. Linux expands these tables in order to create the above-mentioned system objects. As device tables are simple text files, ordinary users can edit them in order to create objects that will be assigned to root, without having special privileges. A device table will need to be created and then the **CONFIG_INITRAMFS_SOURC**E will need to point at its full path. Device tables are created according to the following format:

- nod [name] [mode] [uid] [gid] [dev_type] [maj] [min]
  - **nod** creates a node into the initramfs cpio.
- file [name] [location] [mode] [uid] [gid]
  - **file** copies source file to the initramfs cpio with the right mode, UID and GID.
- dir [name] [mode] [uid] [gid]
  - **dir** creates a directory into the initramfs cpio.
- slink [name] [target] [mode] [uid] [gid]
  - **slink** creates a link into the initramfs cpio.

As creating these files from scratch might be time consuming, the following script, which has only been tested on bash, can be used to **automate** this process:

```
$> scripts/gen_initramfs_list.sh
```

Why Are Some Commands Still Not Working?

Finally, for those who have noticed that the **ps** command is not working on their installations, it has to be said that this is caused by the fact that procfs has not been mounted yet:

```
root> mount -t proc proc /proc
```

Init And Alternatives For Embedded Systems

As already seen before in this course, many programs can be launched right after the boot procedure:

- A shell
- A script
- An executable file

Init And Its Alternatives

However, the **init** daemon can be run by all systems that need for complex initialisation procedures and for starting and monitoring other programs. This is not the only available solution as the following options are also available:

- **sysvinit**, which is a collection of System V-style init programs that includes packages such as init, telinit, shutdown, poweroff, halt, reboot, killall5, runlevel and fstab-decode. The telinit executable can be used to change the runlevel to the one specified by the user, forcing init to kill all running processes that do not belong to the new state.
- **systemd**, which is a system and service manager commonly used on server and desktop Linux distributions that is also to be used on more complex and advanced embedded systems. Like init, systemd is the very first daemon that starts right after the boot process, therefore it is assigned the PID number 1 and it is always the last to be killed during a

system shutdown. The systemd package is configured via simple text files which replace the per-daemon startup shell scripts. Configuration files are hierarchically organised, therefore, those located into higher level directories, override those with the same name but stored into locations with lower priorities. This software suite also comes with additional functionalities that replace the original utilities and daemons such as cron. The following is a list of some of utilities that come with systemd:

- **logind**, is a daemon which manages users login.
- **networkd**, is a daemon which manages the network interfaces configuration.
- **udevd**, is a daemon which manages devices for the Linux kernel.
- **journald**, is a daemon that manages the event logging.
- Many more…

Configuring The BusyBox Init

Although nowadays many Linux distributions no longer rely upon the traditional init as they use systemd instead, most embedded systems are simple enough to still be managed the old way. In fact, these devices might not need (yet) the advanced functionalities that systemd offers, such as parallelization capabilities or stronger integration with Gnome. As already explained, BusyBox comes with its own implementation of init that takes its configuration from /etc/inittab. As at this stage a very simple configuration would be enough, the inittab might just include the following two lines:

```
::sysinit:/etc/init.d/rcS
::askfirst:-/bin/ash
```

Considering that the **inittab** file format is '**ID**:**RunLevel**:**Action**:**Command**' that has to be read as follows:

- **ID** is for any standard init implementation, the identifier for the process. However, BusyBox uses this field to define the controlling tty for the specified process to run on.
- **RunLevel** is the run levels in which this entry can be processed. This field is ignored by the init implementation offered by BusyBox. Therefore, should the reader need runlevels, alternative options such as sysvinit can be used instead.
- **Action** defines how the process has to be handles by the operating system and it can be set to many different values, for example:
  - **once**, which starts the program once. The process is not restarted when it ends.
  - **respawn**, which has the process restarted every time it terminates.
  - **wait**, which starts the process while init waits for its termination. The process is not restarted when it ends.
  - **ctrlaltdel**, which defines the program to be run when the system detects that CTRL-Alt-Del keys combination has been pressed.
  - **shutdown**, which executes the process when the system is told to reboot.
- **Command** defines the command to be run and its parameters.

Therefore, in the example above, while the first line runs the script **rcS**, the second one starts a login shell that will get the profile information from the following files, before displaying the prompt:

- /etc/profile
- ~/.profile

Moreover, the script rcS is the place where file systems have to be mounted:

```
mount -t sysfs sysfs /sys
mount -t proc proc /proc
```

Otherwise, should the reader need for a **more robust** configuration, one may consider to use the default inittab file that comes with BusyBox instead, as this one would, for example, umount all devices and turn the swap area off before rebooting.

To have **QEMU** executing the init program, the parameter -append has to be modified as follows:

```
… -append "console=ttyAMA0 rdinit=/sbin/init"
```

Similarly, the reader can have **U-Boot** execute init by passing the right value to the boot arguments as follows:

```
=> setenv bootargs console=ttyO0,115200 rdinit=/sbin/init
```

As init is started, **other daemons** might need to be launched too at startup, once again using one of the applet that come with BusyBox or any other similar package. This can be achieved by editing the inittab file as follows:

```
::respawn:[path_to_executable] [options]
```

For example, the next line would start syslogd:

```
::respawn:/sbin/syslogd -n
```

Here the option “-n” means that the process needs to run as a foreground process.

Managing Device Nodes

Device nodes appear to be ordinary files and allow application programs to interact with devices through their drivers:

- They are stored into the /dev directory.
- The kernel knows when these files are created and keeps track of them, which does not happen to ordinary files.

- The kernel reads and writes data to and from these files.
- They can be created and managed in many different ways:
  - MAKEDEV
  - mknod
  - mdev
  - devtmpfs
  - udev

Managing Device Nodes With Makedev

The program **MAKEDEV** is an obsolete, but still working, solution to automate the creation of device files into the /dev directory. As this application might not be aware of all devices the system administrator needs to configure, **mknod** can be invoked to manually create the remaining nodes. The utility mknod works as follows:

```
$> mknod [device-name] [device-type] [major-num] [minor-num]
```

The above mentioned fields mean:

- **device-name** is a name such as /dev/mydev which is used to identify the device.
- **device-type** usually can be ‘b’ for block device (storage), ‘c’ for buffered character devices, ‘u’ for un-buffered character devices or ‘p’ for a FIFO.
- **major-num** identifies the driver associated with the device.
- **minor-num** is necessary to differentiate among devices that share the same major-num as they are controlled by the same driver.

Managing Device Nodes With Udev

Although **udev** is an application that can be mostly found on Desktop systems, it can also be used on embedded devices which need for advanced configuration. This device manager consists of some kernel services that inform the udevd daemon when certain system events occur so that it can trigger the proper responses. In fact, udev relies upon sysfs as when drivers register with sysfs they become available to userspace processes and to udevd itself:

- Drivers that are compiled into the kernel register their objects with sysfs.
- Drivers that are compiled as modules register their objects with sysfs when they are loaded.

More in particular, after the kernel creates the device file into the devtmpfs file system, a uevent message is sent to udevd which will read the following files in order to look for rules to be applied to the device node:

- /run/udev/rules.d
- /etc/udev/rules.d
- /lib/udev/rules.d

Although going into all details of these rules go well beyond the scope of these tutorial, the next example can be used in order to gain a basic understanding of this topic:

```
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", ENV{PRODUCT}=="5ac/12[9a][0-9a-f]/*", ACTION=="remove", RUN+="/usr/sbin/usbmuxd -x"
```

The rule above will have the system execute '/usr/sbin/usbmuxd -x' in order to stop usbmuxd when the last usb device is unplugged and only if the PRODUCT environment variable matches the given regular expression. Using the following command as root, it is possible to display how udev and the kernel communicate to one another:

```
root> udevadm monitor
```

At this present moment, this technology can be retrieved and installed in many different ways:

- The traditional udev is part of systemd and it is installed and configured by default when the latter is installed.
- The package eudev is a Gentoo fork of udev that is not part of systemd. This is the best solution for those who do not need the entire software suite.

Managing Device Nodes With Mdev

BusyBox comes with the **mdev** applet, which is a light-weight version of udev and it can be used to create nodes and execute updates on embedded systems. In order to properly run, this application requires:

- The pseudo file system sysfs to be mounted as /sys.
- Hot-plugging enabled in the kernel.

The following snippet is an example of how to edit the init script to run this applet:

```
mount -t proc proc /proc
mount -t sysfs sysfs /sys
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

Should procfs be unavailable, the following alternative can be used:

```
mount -t sysfs sysfs /sys
sysctl -w kernel.hotplug=/sbin/mdev
mdev -s
```

The mdev applet can be configured by editing the /etc/mdev.conf file, which controls the ownership and the permissions of device nodes. The content of this file will be similar to the following:

```
null 0:0 666
```

The line above will tell mdev to assign /dev/null to the user id zero, the group id zero setting the permissions to 666. As actions to be executed when devices are removed or added can be set, the reader should retrieve the official documentation in order to evaluate the full potential of this tool.

Managing Device Nodes With Devtmpfs

Lastly, this section of the tutorial is an introduction to **devtmpfs**, a pseudo file system onto which the kernel creates nodes for all devices it knows about and for all the new ones that are detected at run-time. This file system is mount over /dev at boot time. In order to use this technology, the kernel has to be built with the CONFIG_DEVTMPFS option enabled, which might not be the default configuration for all platforms. To try devtmpfs, the following command can be launched using the root account:

```
root> mount -t devtmpfs devtmpfs /dev
```

To permanently activate this technology, the line above has to be added to an initialisation script such as the standard rcS.

Conclusions

As already explained, very often Linux systems are configured to have all device nodes automatically created by devtmpfs, while mdev and udev are used just to implement rules and setting ownership and permissions. Moreover, simple static device nodes can be preferred over dynamically created ones when:

- The booting procedure should be as fast as possible and CPU cycle cannot be used to create device nodes.
- The hardware configuration is not supposed to change over time.

Basic Network Configuration

Although countless tools can be used to configure network services on Linux platforms, this tutorial mostly focuses on the easiest of all options: using applets released with the **BusyBox** package, as this solution would be flexible enough for most systems, still delivering good performance. Other valid alternatives are also:

- **Buildroot**, among all options, this tool allows system administrators to setup the network of bootable Linux environments, just by preparing a bunch of configuration files and make settings.
- **nmcli** configures systems on which NetworkManager is up and running, where it can quickly setup and monitor network connections.
- **Netplan** is the default network configuration tool installed on Ubuntu systems. It allows system administrators to easily setup networking interfaces via YAML files. This tool supports NetworkManager and systemd-networkd as backend services.

- **systemd-networkd** can also be used on embedded systems, where it can manually configure network devices and network properties via text files.
- **ConnMan** is a network manager explicitly designed for embedded systems: it can easily configure and manage WiFi, Ethernet, Bluetooth, 2G, 3G and 4G network cards via text files. Choosing connman is good idea for those willing to use systemd without networkd. This tool also allows system administrators to create customised plugins and automation scripts.
- **WICD** is a simple and lightweight alternative to NetworkManager and it can manage WiFi and Ethernet network connections. At this present stage, this technology does not come with any support for advanced configuration such as DSL routing.

Managing The Network With BusyBox

This section of the tutorial explains how to set up basic networking on embedded systems using BusyBox. To do so, it is assumed that the physical board is equipped with an Ethernet interface called eth0, able to communicate on an IPv4 network. In the staging directory 'rootfs' used during the previous tutorials, the following folders will have to be created with the proper permissions:

- var/run
- etc/network/if-pre-up.d
- etc/network/if-up.d
- etc/network

In order to assign a **static** IPv4 address to eth0, 192.168.100.2/24 for example, a text file called 'etc/network/interfaces' has to be created with the following content:

```
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet static
   address 192.168.100.2
   netmask 255.255.255.0
   network 192.168.100.0
```

Should the board need a **dynamic** address, the same file will have to contain the following instead:

```
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet dhcp
```

Next, **udchpcd**, the BusyBox DHCP client, needs to be setup by dropping its configuration file into the '/usr/share/udhcpc/default.script' directory. This file should be modelled according to the default example located into the BusyBox folder, which might be 'examples/udhcp/simple.script'. For instance, the configuration file allows system administrators to modify the following parameters:

- Start and end of IP lease block.
- Interfaces names that will use DHCP.
- Maximum number of leases.
- How long an offered address is reserved.
- The amount of time that an IP will be reserved, should an ARP conflict or DHCP decline message occur.
- Time period at which udhcpd will write out leases file.
- BOOTP specific options such as next server and TFTP server names.
- Static leases map.
- DHCP specific options such as the addresses of important servers (WINS, DNS and gateway), static routes to be used and other network specific options.

As the board needs to correctly locate objects such as protocol numbers, passwords and host addresses, on most glibc-based systems administrators will have to create the '/etc/nsswitch.conf' file, providing the **Name Service Switch** (NSS) with information about name resolution mechanisms and common configuration databases used by the system. The following example of nsswitch.conf would be enough for most systems:

```
# Find groups into /etc/group
group: files
# Find hostnames into /etc/hosts and use DNS
hosts: files dns
# Find networks into /etc/networks
networks: files
# Find passwords into /etc/passwd
passwd: files
# Find protocols into /etc/protocols
protocols: files
# Find services into /etc/services
services: files
# Find shadow passwords into /etc/shadow
shadow: files
```

Finally, the administrator should copy from the toolchain **sysroot** all **libraries** Linux uses to perform the name resolution. As these are neither modules nor dependencies, they would not show up in the output of commands such as ldd or readelf:

```
$> cd $HOME/rootfs
$> cp -a $SYSROOT/lib/libnss* lib
$> cp -a $SYSROOT/lib/libresolv* lib
```

Managing The Network With Nmcli

The tool **nmcli** can be used to configure NetworkManager, a daemon that relies on own configuration files '/etc/NetworkManager/NetworkManager.conf' (usually left as default) and

‘/etc/NetworkManager/conf.d/*’. The following example shows how to connect the board to a wireless network using nmcli:

```
# Turn the antenna on
$> nmcli radio wifi on
# List all available WiFi
$> nmcli device wifi list
# Connect to SSID with a password
$> nmcli device wifi connect [ssid] password [password]
```

The following nmcli command line is to assign a static IP to an Ethernet network card and it has to be run as root:

```
root> nmcli con add type ethernet ifname eno1 con-name static-eno1 ip4 192.168.100.2/24 gw4 192.168.100.1
root> nmcli con mod static-eno1 ipv4.dns "192.168.100.100,192.168.100.101"
root> nmcli con up static-eno1
```

Managing The Network With Netplan

The tool netplan is good at simplifying the management of systems running systemd-networkd. In fact, while the former only requires one configuration file, the latter instead needs up to three of them. The following is an example of a netplan configuration file, saved as ‘/etc/netplan/01-netcfg.yaml’, that can setup an Ethernet network card:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    eth0:
      dhcp4: no
      dhcp6: no
      addresses: [192.168.100.2/24]
      gateway4: 192.168.100.1
      nameservers:
        search: [local.domain]
        addresses: [192.168.100.100, 192.168.100.101]
```

Next, as superuser, the following command has to be run in order to apply the above configuration:

```
root> netplan apply
```

Managing The Network With Systemd-Networkd

Very complex systems can benefit from running **systemd-networkd**, especially those ones that need to spawn isolated environments which can be done using systemd-nspawn, a command able to

fully virtualize file systems, all IPC subsystems, host names and domain names. The following example file called '/etc/systemd/network/100-wired.network' can be used on system running systemd-networkd to set up a static IP on a network adapter:

```
[Match]
Name= eth0
[Network]
Address=192.168.100.2/24
Gateway=192.168.100.1
DNS=192.168.100.100
```

Managing The Network With ConnMan

The following snippet of the file '/var/lib/connman/settings' shows the typical **ConnMan** configuration to assign static IP addresses to an Ethernet network adapter:

```
[my_home_ethernet]
Type = ethernet
IPv4 = 192.168.100.2/255.255.255.0/192.168.100.1
IPv6 = 2001:db8::42/64/2001:db8::1
MAC = 00:A0:C9:09:C7:28
Nameservers = 192.168.100.100,192.168.100.101
SearchDomains = local.domain
Domain = another.domain
```

Using Device Tables To Create File System Images

As explained in the previous chapters, **device tables** can define initramfs, but they can also define other file system image formats, such as ext2, jffs2 and ubifs. After having created a device table, it is possible to generate the actual file system by using **genext2fs**, a program that does not require either of the following:

- Mounting the image file before being able to copy files on it.
- Superuser permissions in order to run.

How To Use Genext2fs To Create File Systems

The device table format that genext2fs is able to understand is the following (a row for each device or group):

- [name] [type] [mode] [UID] [GID] [major] [minor] [start] [inc] [count]

Here's the meaning of each field:

- **name** is the file name (path). Some examples:

    - /dev
    - /dev/loop
- **type** describe the file name and it can be one of the following:
    - b for block devices.
    - c for character devices.
    - d for directories.
    - f for regular files.
    - p for named pipes.
- **UID** is the file UID.
- **GID** is the file GID.
- **major** is major device number. For device special files only
- **minor** is the minor device number. For device special files only.
- **count** is for grouping device nodes and it is the total number.
- **inc** is for grouping device nodes and it sets the increment.
- **start** is for grouping device nodes and it is the starting number.

While the initramfs device tables require the system administrator to specify all files, in order to create other file system image formats, the configuration will only define the staging directory with all the exceptions that have to be applied to obtain the expected file system layout. The following lines will create the '/dev' and '/dev/mem' directories:

```
#name       type    mode  UID   GID   major  minor  start  inc    count
/dev        d       755   0     0     -      -      -      -      -
/dev/mem    c       640   0     0     1      1      0      0      -
```

The following can be used to create four teletype devices, from tty0 to tty3:

```
#name       type    mode  UID   GID   major  minor  start  inc    count
/dev/tty    c       666   0     0     5      0      0      0      -
/dev/tty    c       666   0     0     4      0      0      1      3
```

The following to create the master disk on IDE primary controller named '/dev/hda' and its four partitions, from hda1 to hda4:

```
#name       type    mode  UID   GID   major  minor  start  inc    count
/dev/hda    b       640   0     0     3      0      0      0      -
/dev/hda    b       640   0     0     3      1      1      1      4
```

The following to create the null device:

```
#name       type    mode  UID   GID   major  minor  start  inc    count
/dev/null   c       666   0     0     1      3      0      0      -
```

Once the device table file is ready, genext2fs can be run as follows:

```
$> genext2fs [options] [output_image_file]
```

The most commonly used options are:

- **-b** sets the image size in blocks.
- **-d** adds the given directory at a specific path.
- **-D** specifies the device table file path.
- **-N** defines the maximum allowed number of inodes.
- **-U** changes inodes ownership added with the '-d' option to root:root.

For example, knowing that the staging directory is called 'rootfs':

```
$> genext2fs -d rootfs -b 4096 -D dtable.txt -U rootfs.ext2
```

The next step is copying the 'rootfs.ext2' file to the board using **dd**, which requires root permissions. As seen before in this course, the board could be using a SD card to store boot files and the root file system. Should the second partition on the SD card be used as root file system storage, the 'dd' command line will look like this:

```
root> dd if=rootfs.ext2 of=/dev/mmcblk0p2
```

Testing The File System By Starting The Board

The board could then be started using a sequence of commands similar to the following:

```
=> fatload mmc 0:1 0x80F80000 devicefile.dtb
=> fatload mmc 0:1 0x80007FC0 zImage
=> setenv bootargs console=ttyO0,115200 root=/dev/mmcblk0p2
=> bootz 0x80007FC0 – 0x80F80000
```

Booting Through The Network

During the booting process, the only component that has to be locally stored is the bootloader. In fact, the following can be all loaded through the network using **TFTP**:

- Kernel
- Device tree
- Initramfs
- Root file system

Booting The Board Using TFTP

The TFTP protocol needs a server (which can be configured on any personal computer) and a client (which runs on the embedded system being configured). While the configuration of the server changes according to the chosen operating system, the following is an example of the U-Boot command line sequence that will load all files listed above from the server through the network:

```
=> setenv ipaddr [client_ip]
=> setenv serverip [server_ip]
=> setenv netmask [net_mask]
=> setenv bootargs console=ttyO0,115200 root=[root_file_system] rw rootwait ip={ipaddr}
=> tftpboot 0x80200100 zImage
=> tftpboot 0x80F00100 [device_tree]
=> bootz 0x80200100 – 0x80F00100
```

Booting The Board Using NFS

As device tree, initramfs and root file system can also be loaded from a **NFS** server, should the reader choose to implement this scenario, the U-Boot sequence of commands will need to be modified as follows:

```
=> setenv ipaddr [client_ip]
=> setenv serverip [server_ip]
=> setenv npath [staging_dir_path]
=> setenv bootargs console=ttyO0,115200 root=/dev/nfs rw nfsroot=${serverip}:${npath} ip=$
{ipaddr}
=> fatload mmc 0:1 0x80200100 zImage
=> fatload mmc 0:1 0x80F00100 [device_tree]
=> bootz 0x80200100 – 0x80F00100
```

Making sure that the values passed to ‘root=’ and ‘nfsroot=’ are both correct.

Building Embedded Systems With Buildroot

The process of building a Linux embedded system shown so far can be automated using any of the following:

- Buildroot
- EmbToolkit
- OpenEmbedded
- OpenWrt
- PTXdist
- The Yocto Project

Introduction To Buildroot

This chapter will dive into Buildroot, a collection of Makefiles and patches, designed to simplify the process of building Linux embedded systems, even the complex ones, through cross-compilation. Although the main goal of this product is creating root file system images, Buildroot is also able to build bootloaders, kernel images and custom packages. More in particular, Buildroot can build systems with the following characteristics (including, but not limited to):

- **C library**: uClibc-ng, glibc and musl.
- **Bootloader**: U-Boot, grub2, BareBox, afboot-stm32, s500-bootloader.
- **File system**: ext2, ext3, ext4, squashfs, jffs2, cpio, initial RAM file system.
- **Kernel**: patches installation, DTB build, zImage support, uImage support, vmlinux support.
- **Packages**: BusyBox, OpenSSH, Qt and custom.
- **System configuration**: networking, timezone, init (BusyBox, systemd, OpenRC and sysvinit) and '/dev' management (static, eudev, mdev and devtmpfs).

Downloading The Buildroot Source Code

To retrieve the Buildroot **source code** from the Git repository:

```
$> git clone git://git.buildroot.net/buildroot
$> cd buildroot
$> git checkout [latest_version]
```

To get the list of all available Buildroot versions to choose from, the reader can press 'TAB' after 'git checkout' command. The same code is also available as tar archive at the following address:

- https://buildroot.org/downloads

Next, in the 'docs/manual/prerequisite.txt' file (this path might change) the user will have to find the list of mandatory packages that are expected to be already installed on the build machine.

How To Configure The System With Buildroot

At this stage, Buildroot should be ready to go. To configure a system from scratch and create the '.config' file, any of the following commands can be launched from 'buildroot' directory:

```
$> make menuconfig
$> make nconfig
$> make gconfig
$> make xconfig
```

Otherwise, the system administrator can simply choose any of the configuration files stored into the 'configs' directory, making sure to select the one that matches the board or emulator architecture. A list of the available configurations can be printed on screen by running the following:

```
$> make list-defconfigs
```

Please note that a ***defconfig** file contains only the options for which a non-default value has been chosen. Next, the following will create the .config at top level Buildroot level (CONFIG_DIR=$TOPDIR), according to the directives contained in the chosen configuration file:

```
$> make [config_file]
```

How To Tune The Kernel Using Buildroot

The reader should open the .config that was just created in order to familiarize with its options and their values. If needed, the **kernel** can now be tuned and configured by running any of the following:

```
$> make linux-menuconfig
$> make linux-nconfig
$> make linux-gconfig
$> make linux-xconfig
```

How To Build The System With Buildroot

Next, the following command will build the system:

```
$> make
```

After having run the command above, the Buildroot folder will contain the following directories:

- **arch** contains definitions for all supported architectures.
- **board** contains kernel and bootloader configurations, patches and device tables specific to each board.
- **boot** contains packages for bootloaders
- **configs** contains all configurations files for the command ‘make’.
- **dl** contains upstream projects that Buildroot builds.
- **fs** contains Makefiles and dependencies needed to create the target root file system images in several formats.
- **linux** contains Makefiles and dependencies needed to create the kernel.
- **output** contains intermediate and final resources:
  - **output/build** is where components are built.
  - **output/host** contains tools the Buildroot needs to run, which includes the toolchain for cross-compilation. Also contains sysroot of the target toolchain which contains libraries and headers of all user-space packages useful to build other packages: this is not the root file system for the target!
  - **output/images** contains the output of the build: one or multiple bootloaders, kernel images, root file systems and device tree blobs).
  - **output/staging** contains symbolic links to the toolchain sysroot.
  - **output/target** contains the staging area for the root directory. Almost all the files needed to build the target root file system are stored here, except for the ‘/dev’ device files.
- **package** contains all user-space packages and it is the place to store project-specific packages.
- **support** contains patches that Buildroot needs to apply to scripts and packages and test scripts.
- **system** contains what is needed for system integration such as template of a working init system or a file system.

- **toolchain** contains all scripts and tools needed to cross compile and generate the toolchain itself.
- **utils** contains scripts and utilities that Buildroot needs to run.

Almost all directories above, apart from those ones that store the output, will contain at least the following files:

- ***.mk** used by Makefile to define the software to download, configure, build and install. For example, ‘file1.mk’ will contain information able to drive the compilation of the ’file1’ package. This format is an improvement of the standard ‘make’ files, it is very compatible with it and runs up to 30 times faster.
- **Config.in** written according the Kconfig standard and contains details about the configuration settings of all objects of the build.

Testing The System With QEMU

The following lines allow the reader to test a Buildroot build that targets the verstatilepb board on QEMU (login credentials are ‘root’ with a blank password):

```
$> FILEDIR=output/images
$> qemu-system-arm -machine versatilepb -m 256 -dtb ${FILEDIR}/versatile-pb.dtb -kernel
${FILEDIR}/zImage -drive file=${FILEDIR}/rootfs.ext2,if=scsi,format=raw -append
"root=/dev/sda console=ttyAMA0,115200" -serial stdio -net nic,model=rtl8139 -net user
```

Before starting a build, the reader should double check the following:

- **board/[brand]/[model]** contains all configuration files, patches, binary blobs and extra build steps for the kernel Linux and U-Boot. This also includes ‘genimage.cfg’, an intuitive configuration file read by ‘genimage’ which creates the bootable image for the storage device, usually a SD card (use ‘dd’ to copy this file). Buildroot comes with many working ‘genimage.cfg’ for all supported boards that can be taken as example for custom configurations. Also, bear in mind that ‘genimage’ is usually called by ‘post-image.sh’, a script that runs right after the build and it is to be saved in the same directory. The command ‘make menuconfig’ (‘System configuration’ menu) can be used to make sure the correct post image script is called.
- **configs/[model]_defconfig** contains the board default settings.
- **package/[brand]/[package]** contains all packages that needs installing.

How To Create Custom Systems With Buildroot

This is very important especially when the system administrator is using Buildroot to create custom software for a specific board. In fact, to customize U-Boot one will have to:

- Install the patch into ‘board/[brand]/[model]’.
- Run ‘make menuconfig’ in order to set the path of the patch file, the board name and the U-Boot version.

- Add uEnv.txt (the U-Boot environment variables) to the ‘board/[brand]/[model]’ directory. This file will contain the sequence of commands to be launched to start the system.

Once a new configuration has been tested, it can also be saved using:

```
$> make savedefconfig BR2_DEFCONFIG=configs/[configuration_name]
```

To add **custom packages** that system administrators will be able to select from a **make menu**:

1. Create a subfolder into ‘package’ for the new application. For example, if the new software is called ‘mynewapp’, the relative path will be ‘package/mynewapp’
2. Store the mynewapp **Config.in** in ‘package/mynewapp’.
3. Make the new package **visible** to Buildroot by editing ‘package/Config.in’.
4. Create the **mk** file called ‘mynewapp.mk’ in ‘package/mynewapp’. For simple packages, system administrators should be able to create custom configurations just by looking at any working *.mk.

The **package/mynewapp/Config.in** for the custom application will be something like:

```
config BR2_PACKAGE_MYNEWAPP
    bool "This is my new app"
```

While the following is to be added to **package/Config.in** to make Buildroot aware of the new software:

```
menu "These are my programs"
    source "package/mynewapp/Config.in"
endmenu
```

Alternatively, to deploy the package as an **overlay** the system administrator will:

- Build the package using the Buildroot toolchain located in the ‘output/host/usr/bin’ directory.
- Copy the package to the staging area which would be something like ‘board/[brand]/[model]/overlay/usr/bin’.
- Use ‘make menuconfig’ to modify the value of BR2_ROOTFS_OVERLAY by accessing the menu ‘System configuration >> Root filesystem overlay directories’.

Building Embedded Systems With The Yocto Project

The Yocto Project is a set of tools based on shell scripts and Python designed to create embedded Linux images, kernels, bootloaders, toolchains and root file systems. While Yocto Project currently supports AMD, ARM, Intel, MIPS, PPC and many other architectures, its output can be easily moved across different systems. The Yocto Project consists of many components:

- AutoBuilder, to automate testing of builds and monitoring quality assurance (QA).
- Auto Upgrade Helper, automates the generation of updates for packages created by Yocto.
- BitBake, a parser and a task scheduler.
- Cross Platforms (CROPS), based on Docker containers, allows the user to build software for multiple architectures.
- Cross-Prelink, improves performances by executing prelinking instead of having the dynamic linker executing tasks at runtime.
- Devtool, a command line tool designed for building, testing and packaging software.
- Eclipse IDE plugin, to develop using the Yocto Project within Eclipse.
- OE-Core, the core metadata.
- Pseudo, the Yocto version of fakeroot.
- Poky, the reference distribution.
- Toaster, a web interface to BitBake and its metadata which simplifies execution of tasks.
- Many more…

How To Install The Yocto Project

The source code can be retrieved either via Git:

```
$> git clone git://git.yoctoproject.org/poky.git
$> cd poky
$> git checkout [required_version]
```

Or via HTTP at the address:

- http://downloads.yoctoproject.org/releases/yocto/

After having checked that the host machine matches the system requirements, as outlined in the ‘documentation/ref-manual’ directory, before running any of the Yocto command shown in this tutorial, the following have to be executed:

```
$> cd poky
$> source oe-init-build-env
```

The ‘oe-init-build-env’ script needs to be sourced at the beginning of each work session in order to set up the environment.

BitBake And The Yocto Project Metadata

The main task of BitBake is parsing metadata to find tasks to be executed to produce the expected output. The behaviour of BitBake can be controlled via metadata on a global level through the following files:

- **Recipes** (.bb), contain the information needed to build a single package, such as:
  - Source code location
  - Dependencies information

    - Patches location
    - Compilation information
    - Packaging information
- **Class data** (.bbclass), contain information and settings that recipes need to share
- **Configuration data** (*.conf), define configuration variables such as compiler options, machine configuration options and much more

Or on a build level, through the following technologies:

- **Metadata**, which can be provided by the user as:
    - Recipes (.bb)
    - Recipe append files (.bbappend) to override or extend data defined into recipe
    - Patches (*.patch or *.diff)
- **Board Support Package** (BSP) to define drivers and support packages necessary to a specific platform to work properly
- **Policy Configuration** to define a set of policies that apply to images and SDKs to control their high level behaviour. For example, a policy configuration can dictate the package manager a specific distribution will use

The Yocto Project Layers

Yocto Project uses layers to store and organise its metadata: non-homogeneous metadata should not be mixed up in the same layer. Therefore, for example, the GUI and the middleware metadata should be split into two different layers. All layers names begin with the ‘meta’ prefix. BitBake will try to find layers by traversing the list of directories specified in the ‘[project_home]/conf/bblayers.conf’ file by the BBLAYERS variable, as in the following excerpt:

```
BBPATH = "${TOPDIR}"
BBFILES ?= ""
BBLAYERS ?= " \
  ${TOPDIR}/layer1 \
  ${TOPDIR}/layer2 \
  ${TOPDIR}/layer3 \
  ${TOPDIR}/layer4 \
  "
```

To add support for a specific platform or technology, system administrators can either create a new layer from scratch or simply download one from the internet. To create a layer from scratch, the reader can run the following to generate the basic structure of the new layer:

```
$> bitbake-layers create-layer [layer_name]
```

After having opportunely customised the new layer ‘layer.conf’, the system administrator needs to copy the new layer to the chosen directory by modifying the ‘bblayers.conf’ file accordingly by running the following:

```
$> bitbake-layers add-layer [layer_name]
```

To make sure the new layer has been actually added, the system administrator can run the following:

```
$> bitbake-layers show-layers
```

The Yocto Project Recipes

Recipes are files that contain tasks whose names always begin with the ‘do_’ prefix (‘do_build’ always being the default task), written using Python and shell script, parsed and executed by BitBake. The following example recipe builds and installs a piece of software:

```
SUMMARY = "Some random package"
HOMEPAGE = "https://github.com/someurl/somewhere/somepackage"
LICENSE = "ISC"
LIC_FILES_CHKSUM = "file://LICENSE;md5=6eed01fa0e673c76f5a5715438f65b1d"
SECTION = "libs"
DEPENDS = ""
SRC_URI = "git://github.com/someurl/somewhere/somepackage \
"
S = "${WORKDIR}/git"
DEPENDS = "glib-2.0 libsolv rpm gtk-doc json-c swig-native"
#Set the environment using EXTRA_OEMAKE
CFLAGS += "-Iarch/${ARCH} -Iarch/common"
EXTRA_OEMAKE = "CFLAGS='${CFLAGS}' LIBDIR='${base_libdir}'"
#Never call ‘make’, use ‘oe_runmake’ instead
do_compile() { oe_runmake ARCH=${ARCH}}
do_install() { oe_runmake ARCH="${ARCH}" DESTDIR="${D}" install }
```

As the recipes syntax is very intuitive, readers with a good programming background will be able to start creating their own custom recipes in no time.

How To Build An Image With The Yocto Project

As already explained, the first step is sourcing the ‘oe-init-build-env’ script to prepare the environment and set up the directory ‘build/’ as the working directory. The default working directory can be changed by passing the desired folder as parameter, as follows:

```
$> source oe-init-build-env [my_working_directory]
```

The system administrator will now open the ‘build/conf/local.conf’ file looking for the ‘MACHINE’ variables group, making sure to remove the hashtag symbol from the targeted architecture. The configuration below, targets the ARM architecture for the QEMU emulator:

```
MACHINE ?= "qemuarm"
```

```
#MACHINE ?= "qemuarm64"
#MACHINE ?= "qemumips"
#MACHINE ?= "qemumips64"
#MACHINE ?= "qemuppc"
#MACHINE ?= "qemux86"
#MACHINE ?= "qemux86-64
```

Running the following command inside the ‘poky’ directory shows all available images the system administrator can choose from:

```
$> ls meta*/recipes*/images*/*.bb
```

After having chosen an image, the system administrator can run the build:

```
$> bitbake [image_name]
```

The default settings will place all artefacts in the ‘poky/build/tmp’. Moreover, the administrator might also want to monitor the following folders:

- ‘poky/build/tmp/deploy/images’ to store full images
- ‘poky/build/tmp/deploy/rpm’ to store RPM packages. Useful when systems have to be rebuilt multiple times, as packages will not need to be rebuilt multiple times. The same goes for the ‘poky/build/tmp/deploy/deb’ and ‘poky/build/tmp/deploy/ipk’ directories
- ‘poky/build/downloads’, to store downloaded tarballs that can be reused for any build subsequent to the first one
- ‘poky/meta’ to store the core metadata files (*.bb, *.inc, *.patch, etc…). Usually *.bbappend files are not located here, as they only define custom settings. However, they can be found into directories such as ‘poky/meta-yocto-bsp’, which contains the BSP for Yocto

How To Test The Image With QEMU

To test the image that was just built in the previous step, the system administrator will need to locate the image first. As already explained, this file can be found into the architecture specific folder ‘poky/build/tmp/deploy/images/[arch_specific]’. After having located the image, the system administrator can run:

```
$> runqemu [image_name]
```

When the image is fully loaded, the administrator should be able to login as ‘root’ providing a blank password.

How To Add Recipes To An Image

To quickly add recipes to a test or development image without having to manage layers, the system administrator can simply modify the ‘IMAGE_INSTALL_append’ variable in the ‘local.conf’ file as follows (including a space right before the first package to include):

```
IMAGE_INSTALL_append = " new_pkg_one new_pkg_two"
```

Should any of the test packages be needed on a production image too, the administrator will create a recipe file that includes:

- A ‘require’ directive that points at the desired base image stored into any of the ‘images’ directories
- The ‘IMAGE_INSTALL’ variable to list the additional packages

The following is an example of a recipe that implements the steps above:

```
require [recipe_dir]/images/[image_file].bb
IMAGE_INSTALL += "new_pkg_one new_pkg_two"
```

However, working with any of the ‘IMAGE_INSTALL’ variables is dangerous, as they might cause issues with dependencies.

How To Create And Manage A SDK

Generating a SDK allows programmers, software engineers and software testers to work on images, kernels, applications and QA test cases without having to install the full Yocto Project:

- The system administrator creates the SDK and installs it on the software engineer machine
- The system engineer builds or downloads the target image
- The system engineer starts developing or testing an application

The following command line creates the SDK installation script in the ‘/tmp/deploy/sdk’ directory:

```
$> bitbake -c populate_sdk [image_name]
```

After having installed the SDK, the software engineer will need to source the correct script to initialise the environment: the scrip to run will not be ‘oe-init-build-env’ as this does not work for the SDK.