Description

Setting up an environment with the appropriate tools can help developers in the process of debugging and overall analyzing a system's behavior. Yocto supports several debugging capabilities which are aimed to produce information useful for the development process. These Yocto features available by adding a few additional packages and features as discussed in the following sections.

General Environment Setup

In order to create an appropriate development environment, Yocto allows to include useful packages and generate specific information required by the development tools. Additionally, it is well known that for the process of debugging, it is also necessary to disable optimizations and enable debug symbols. The following is a list of additional configuration to be added to the **build/conf/local.conf** file. You can choose which ones to include depending on your application and desired environment. The following example enables **ALL** of the recommended development/debugging tools.

$YOCTO_BUILD/conf/local.conf:

```
EXTRA_IMAGE_FEATURES += "\
    dbg-pkgs \       # adds -dbg packages for all installed packages and symbol information for debugging and profiling.
    tools-debug \    # adds debugging tools like gdb and strace.
    tools-profile \  # add profiling tools (oprofile, exmap, lttng valgrind (x86 only))
    tools-testapps \ # add useful testing tools (ts_print, aplay, arecord etc.)
    debug-tweaks \   # make image for suitable of development, like setting an empty root password
    tools-sdk \      # OPTIONAL: adds development tools (gcc, make, pkgconfig, etc)
    dev-pkgs"        # OPTIONAL: adds -dev packages for all installed packages

# Specifies to build packages with debugging information
DEBUG_BUILD = "1"

# Do not remove debug symbols
INHIBIT_PACKAGE_STRIP = "1"

# OPTIONAL: Do not split debug symbols in a separate file
INHIBIT_PACKAGE_DEBUG_SPLIT= "1"
```

EXTRA_IMAGE_FEATURES
: By editing this variable, we can add the desired packages to be included in our generated image. Bear in mind, however, that adding these will increase the size and time needed for building the image. In the following sections, there is a set list of necessary packages for the recommended tools.

DEBUG_BUILD
: Enables full debug and backtrace capabilities for all programs and libraries in the image, by modifying the SELECTED_OPTIMIZATION variable, setting it to "DEBUG_OPTIMIZATION".

INHIBIT_PACKAGE_STRIP
: This is needed to ensure that in the process of building, debugging symbols aren't stripped by Yocto from the binaries it packages. **Note:** If this is only needed for a specific recipe, it can be added by using OVERRIDES.

INHIBIT_PACKAGE_DEBUG_SPLIT
: Adds the symbols and debug info files onto the filesystem as separate files instead of having them embedded with the executables.

Finally, for the general environment setup, depending on where and what we are debugging, it might be useful to have bitbake emit debugging output, which can be achieved by adding the BBDEBUG option to the local.conf as follows. You can also create a build history git repository that stores meta-data about the current project, including package dependencies, size and generated sub-packages, stored in the $PROJECT_DIR/build/buildhistory.

$YOCTO_BUILD/conf/local.conf:

```
BBDEBUG = "yes"

INHERIT += "buildhistory"
BUILDHISTORY_COMMIT = "1"
```

Recommended debugging tools

Once the environment is prepared to generate the appropriate information, most additional tools can be included by adding the tool's recipe to the respective meta-layer and adding it to IMAGE_INSTALL on the desired image recipe. **Note:** If you are not using your own recipe, you can append it to the already existing one by creating a $IMAGE_NAME.bbappend on your meta-layer. Most of these tools' recipes are already available on a very generic format, and can be modified if necessary to adjust to a specific setup or hardware. You can check your current Yocto build and its meta-layers to see if any of them have the desired recipe, or if it is added to your layer path by running.

```
bitbake -e $PACKAGE_NAME
```

For some tools, like GDB and Strace, the process is different, which will be presented in the following sections, since they are already easily included by Yocto.

Tracing and profiling

Strace

Strace allows to observe one or multiple running processes at the level of system (kernel) calls. You can obtain information about your program while it is executing, or you can save your output and analyze it afterwards. It is mostly used for problems outside of the binary itself; for example, knowing which configuration files were read, which files or libraries were ready before a program crash, input data, kernel interfaces, frequency and time consumed by system cals, and so on.

For setting up the environment to use it, remember that the package "tools-debug" already includes it, so you should add to your local.conf file:

$YOCTO_BUILD/conf/local.conf:

```
EXTRA_IMAGE_FEATURES += "tools-debug"
```

If you want to only add strace, make sure the following are added to your image recipe file:

On image recipe:

```
IMAGE_INSTALL += "strace procps"
```

Adding "procps" includes pgrep, a process utility that will make our debugging easier by looking up process IDs by name.

Perf

Tracing and profiling are fundamental on developing, and it is the developer's responsibility to write code that provides the highest possible performance, so it becomes necessary to analyze programs to find bottlenecks and optimize those regions of code. This analysis can be achieved by using Perf. More specifically, it can be used to solve performance and troubleshooting functions, and answer a variety of questions about code-paths, allocated memory, TCP retransmits, calls to kernel functions,

The following options are the minimum needed for running perf.

$YOCTO_BUILD/conf/local.conf:

```
EXTRA_IMAGE_FEATURES = "debug-tweaks tools-profile dbg-pkgs"
INHIBIT_PACKAGE_STRIP = "1"
PACKAGE_DEBUG_SPLIT_STYLE = 'debug-file-directory'
```

Perf is included in the 'tools-profile' package, but it can also be manually included by modifying the image recipe.

On local image:

```
IMAGE_INSTALL += "perf"
```

Additionally, you might need to modify your kernel configuration in order to get useful information. You can modify it with the command:

```
$ bitbake -c menuconfig virtual/kernel
```

Make sure your debug information is available **(CONFIG_DEBUG_INFO=y)**, which you can configure under 'Kernel hacking -> Compile the kernel with debug info' or, on newer versions, 'Kernel hacking -> Compile-time checks and compiler options-> Compile the kernel with debug info'. Also, make sure perf_events are enabled, which you can check under 'General setup -> Kernel Performance Events and Counters' and 'Kernel features -> Enable hardware performance counter support for perf events'. Finally, you can enable trace calls and kprobes/uprobes for dynamic events under 'Kernel hacking -> tracers'

Memory Analysis

Valgrind

Valgrind is a very powerful tool whenever we are dealing with memory leaks and overall trying to find errors of work with memory, but it can also be used for performance profiling, finding synchronization errors in multi-threading programs and analysis of memory consumption.

As you would do for any other package that you want to include in your image and is not part of the default configuration, you need to add the recipe to build it. The source will depend on what board you a using, and you must add the recipe to your meta-layer. Valgrind also needs some of the previously defined features on your **local.conf**, specifically:

$YOCTO_BUILD/conf/local.conf:

EXTRA_IMAGE_FEATURES += " dbg-pkgs tools-debug "

Also, remember to add it to your image recipe, as follows:

On your image recipe:

IMAGE_INSTALL += "valgrind"

Network analysis

Tcpdump

Tcpdump is a powerful command-line packet analyzer; and libpcap, a portable C/C++ library for network traffic capture. Is used to display TCP\IP and other network packets being transmitted over the network attached to the system, as well as to display available interfaces and capture packets.

You can add it to your target image by including the corresponding recipe on your meta-layer. To add it to your image, you need to add to your image recipe:

On your image recipe:

IMAGE_INSTALL += "tcpdump"

Kernel debugging

Kdump

Kdump is very useful for analyzing kernel crashes. It uses kexec to quickly boot to a dump-capture kernel whenever a dump of the system kernel's memory needs to be taken (for example, on system panics). The system kernel's memory image is preserved across the reboot and is accessible to the dump-capture kernel. The crash dump is captured from the context of a freshly booted kernel and not from the context of the crashed kernel. For Yocto, it is available in the kexec-tools package, and it can be included in your image by adding the recipe to your meta-layer, and modifying your recipe by adding the line:

On your image recipe:

IMAGE_INSTALL += "kexec-tools"

KGDB

KGDB is intended to be used for kernel debugging, as a source level debugger along with GDB. It also allows to place breakpoints in kernel code and perform limited execution stepping.

In order to include KGDB on our image, it is necessary to modify the kernel configuration by running the following command:

bitbake -c menuconfig virtual/kernel

The settings are as follows:

Kernel hacking -> KGDB: Kernel debugger (Check it and Enter):
Check: KGDB: use kgdb over the serial console
Check: KGDB_KDB: include kdb frontend for kgdb
Enable: Kernel debugging and Magic SysRq key

Enable: Compile the kernel with debug info

Finally, it is necessary to add SSH to the image. This can be done by adding to the **$YOCTO_BUILD/conf/local.conf**:

$YOCTO_BUILD/conf/local.conf

```
CORE_IMAGE_EXTRA_INSTALL += "openssh"
```

Debugging

GDB

GDB is a powerful debugging tool that allows us to see what is going on "inside" another running program, or what it was doing at the moment it crashed. When building with Yocto, it can either used either on the target, or remotely by using gdbserver, which runs on a remote computer (the host GDB), allowing to run or stop a program and read memory regions. Due to all of these processing running on the host, it becomes quite agile on the target in terms of speed and space, given that the binaries on the target don't need their debugging symbols and information.

Setting up environment for debugging on target

As previously mentioned, GDB is included in the package "tools-debug", so commonly the EXTRA_IMAGE_FEATURES would be set as follows:

$YOCTO_BUILD/conf/local.conf:

```
EXTRA_IMAGE_FEATURES = "tools-debug debug-tweaks dbg-pkgs"
```

Notice also that the package "dbg-pkgs" will help us include the debug symbols for all the packages. To enable debugging symbols for a specific package, the -dbg package must be included. This can be done by modifying the recipe, or the local.conf file, by adding the following line:

On your image recipe:

```
IMAGE_INSTALL += " $PACKAGE_NAME $PACKAGE_NAME-dbg"
```

Setting up environment for remote debugging (GDBServer)

The host GDB must have access to all debugging information and unstripped binaries, libraries and target must be compiled with no optimizations, all of which were mentioned on the general environment setup. For more information, refer to the official documentation. The steps are listed here for convenience:

⑩ Configure your build system to construct the companion debug filesystem by editing the local.conf file:

$YOCTO_BUILD/conf/local.conf:

```
IMAGE_GEN_DEBUGFS = "1"
IMAGE_FSTYPES_DEBUGFS = "tar.bz2"
```

These options cause the OpenEmbedded build system to generate a special companion filesystem fragment, which contains the matching source and debug symbols to your deployable filesystem. The build system does this by looking at what is in the deployed filesystem, and pulling the corresponding -dbg packages.

The companion debug filesystem is not a complete filesystem, but only contains the debug fragments. This filesystem must be combined with the full filesystem for debugging. Subsequent steps in this procedure show how to combine the partial filesystem with the full filesystem.

⑩ Configure the system to include gdbserver in the target filesystem by adding gdbserver to either the local.conf or the recipe:

On your image recipe:

IMAGE_INSTALL += “gdbserver"

⑩ Construct the image and companion filesystem

bitbake $IMAGE_NAME

⑩ Build the cross GDB component and make it available for debugging. Build the SDK that matches the image. Building the SDK is best for a production build that can be used later for debugging, especially during long term maintenance:

bitbake -c populate_sdk $IMAGE_NAME

Alternatively, you can build the minimal toolchain components that match the target. Doing so creates a smaller than typical SDK and only contains a minimal set of components with which to build simple test applications, as well as run the debugger:

bitbake meta-toolchain

A final method (the quickest, but no recommended for long-term maintenance) is to build Gdb itself within the build system, producing a temporary copy of cross-gdb you can use for debugging during development.

bitbake gdb-cross-architecture

⑩ Set up the debugfs by running the following set of commands:

mkdir debugfs
cd debugfs
tar xvfj build-dir/tmp-glibc/deploy/images/machine/image.rootfs.tar.bz2
tar xvfj build-dir/tmp-glibc/deploy/images/machine/image-dbg.rootfs.tar.bz2

Kernel tree root directory:

Documentation/: kernel source documentation.

LICENSES/: It contains the licenses to be applied to the kernel source and individual source files which may have different licenses. Inside this directory you will find 3 subdirectories: preferred

(GNU GPL), exceptions listed in the COPYING file and other.

arch/: source codes for specific architectures (e.g powerpc, x86,etc). Inside this directory you'll find subdirectories belonging to each supported architecture lie i386, sparc, arm, etc.

block/: block I/O layer, contains code for the management of block devices (such as hard disks, dvd, floppy disks, etc.) and their requests.

Certs/: certificates and sign files to enable module signature to make the kernel load signed modules, this can be useful to prevent malicious code from running with kernel modules like rootkits.

Crypto/: Crypto API. Contains cryptographic ciphers which handles cryptographic and compression tasks.

drivers/: Hardware device drivers. Contains code of device drivers to support hardware, inside this directory you'll find subdirectories for each hardware such as video, bluetooth,etc (any hardware supported).

fs/: code for the Virtual File System and additional filesystems. This directory contains the code to support, read and write filesystems.

include/: kernel headers. This directory contains C headers for kernel files such as functions to compile code.

init/: kernel boot. Contains source code related to the initialization of the kernel. The source code is stored in a file called mainc.c within the directory /init. The code initializes the kernel and some initial processes.

ipc/: Inter-Process Communication such as signals and pipes.

kernel/: Core subsystems, such as the scheduler signal handling code, etc.

lib/: library routines common string operations, hardware dependent operations, debugging routines and command line parsing code.

mm/: Memory management and virtual memory. The kernel manages both the hardware and virtual memory (swap). This directory stores code for the memory management.

net/: Network stack. Contains code related to communication protocols such as IP, TCP,UP, etc.

samples/: sample code and configuration files.

scripts/: scripts to build the kernel.

security/: Linux Security Module (LSM) is a framework to enable security policies to access modules control.

sound/: The sound subsystem, here you'll find sound drivers and code related to sound such as ALSA,

tools/: tools for compressed kernel development such as ACPI, cgroup, USB testing tools, vhost test module, GPIO, IIO and spi tools, Inter energy policy tool among more.

usr/: initramfs which roots the filesystem and init in the kernel memory cache.

virt/: Virtualization, this directory contains the KVM (Kernel Virtual Machine) module for hypervisor.

These are the main subdirectories of the kernel tree and their function, hope its helpful for you.

How to Cross Compile the Linux Kernel with Device Tree Support

his article is intended for those who would like to experiment with the many embedded boards in the market but do not have access to them for one reason or the other. With the QEMU emulator, DIY enthusiasts can experiment to their heart's content

You may have heard of the many embedded target boards available today, like the BeagleBoard, Raspberry Pi, BeagleBone, PandaBoard, Cubieboard, Wandboard, etc. But once you decide to start development for them, the right hardware with all the peripherals may not be available. The solution to starting development on embedded Linux for ARM is by emulating hardware with QEMU, which can be done easily without the need for any hardware. There are no risks involved, too.

QEMU is an open source emulator that can emulate the execution of a whole machine with a full-fledged OS running. QEMU supports various architectures, CPUs and target boards. To start with, let's emulate the Versatile Express Board as a reference, since it is simple and well supported by recent kernel versions. This board comes with the Cortex-A9 (ARMv7) based CPU.

In this article, I would like to mention the process of cross compiling the Linux kernel for ARM architecture with device tree support. It is focused on covering the entire process of working—from boot loader to file system with SD card support. As this process is almost similar to working with most target boards, you can apply these techniques on other boards too.

Device tree

Flattened Device Tree (FDT) is a data structure that describes hardware initiatives from open firmware. The device tree perspective kernel no longer contains the hardware description, which is located in a separate binary called the device tree blob (dtb) file. So, one compiled kernel can support various hardware configurations within a wider architecture family. For example, the same kernel built for the OMAP family can work with various targets like the BeagleBoard, BeagleBone, PandaBoard, etc, with dtb files. The boot loader should be customised to support this as two binaries-kernel image and the dtb file – are to be loaded in memory. The boot loader passes hardware descriptions to the kernel in the form of dtb files. Recent kernel versions come with a built-in device tree compiler, which can generate all dtb files related to the selected architecture

family from device tree source (dts) files. Using the device tree for ARM has become mandatory for all new SOCs, with support from recent kernel versions.

Building QEMU from sources

You may obtain pre-built QEMU binaries from your distro repositories or build QEMU from sources, as follows. Download the recent stable version of QEMU, say qemu-2.0.tar.bz2, extract and build it:


```
tar -zxvf qemu-2.0.tar.bz2
cd qemu-2.0
./configure --target-list=arm-softmmu, arm-linux-user --prefix=/opt/qemu-arm
make
make install
```

You will observe commands like qemu-arm, qemu-system-arm, qemu-img under /opt/qemu-arm/bin.

Among these, qemu-system-arm is useful to emulate the whole system with OS support.

Preparing an image for the SD card

QEMU can emulate an image file as storage media in the form of the SD card, flash memory, hard disk or CD drive. Let’ s create an image file using qemu-img in raw format and create a FAT file system in that, as follows. This image file acts like a physical SD card for the actual target board:

```
qemu-img create -f raw sdcard.img 128M
#optionally you may create partition table in this image #using tools like sfdisk, parted
mkfs.vfat sdcard.img
#mount this image under some directory and copy required files
mkdir /mnt/sdcard
mount -o loop,rw,sync sdcard.img /mnt/sdcard
```

Setting up the toolchain

We need a toolchain, which is a collection of various cross development tools to build components for the target platform. Getting a toolchain for your Linux kernel is always tricky, so until you are comfortable with the process please use tested versions only. I have tested with pre-built toolchains from the Linaro organisation, which can be got from the following link http://releases.linaro.org/14.0.4/components/toolchain/binaries/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux.tar.xz or any latest stable version. Next, set the path for cross tools under this toolchain, as follows:

```
tar -xvf gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux.tar.xz -C /opt
export PATH=/opt/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux/bin:$PATH
```

You will notice various tools like gcc, ld, etc, under /opt/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux/bin with the prefix arm-linux-gnueabihf-

Building mkimage

The mkimage command is used to create images for use with the u-boot boot loader.
Here, we'll use this tool to transform the kernel image to be used with u-boot. Since this tool is available only through u-boot, we need to go for a quick build of this boot loader to generate mkimage. Download a recent stable version of u-boot (tested on u-boot-2014.04.tar.bz2) from ftp.denx.de/pub/u-boot:

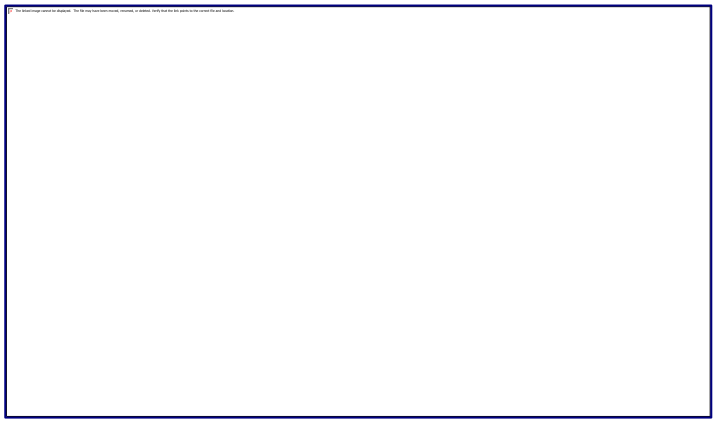
```
tar -jxvf u-boot-2014.04.tar.bz2
cd u-boot-2014.04
make tools-only
```

Now, copy mkimage from the tools directory to any directory under the standard path (like /usr/local/bin) as a super user, or set the path to the tools directory each time, before the kernel build.

Building the Linux kernel

Download the most recent stable version of the kernel source from kernel.org (tested with linux-3.14.10.tar.xz):

```
tar -xvf linux-3.14.10.tar.gz
cd linux-3.14.10
make mrproper #clean all built files and configuration files
make ARCH=arm vexpress_defconfig #default configuration for given board
make ARCH=arm menuconfig #customize the configuration
```

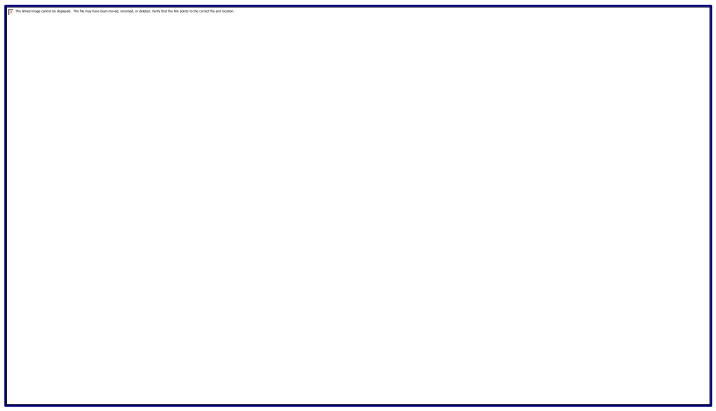


Then, to customise kernel configuration (Figure 1), follow the steps listed below:

1) Set a personalised string, say ' -osfy-fdt', as the local version of the kernel under general setup.
2) Ensure that ARM EABI and old ABI compatibility are enabled under kernel features.
3) Under device drivers–> block devices, enable RAM disk support for initrd usage as static module, and increase default size to 65536 (64MB).
You can use arrow keys to navigate between various options and space bar to select among various states (blank, m or *)
4) Make sure devtmpfs is enabled under the Device Drivers and Generic Driver options.
Now, let's go ahead with building the kernel, as follows:

```
#generate kernel image as zImage and necessary dtb files
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage dtbs
#transform zImage to use with u-boot
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- uImage \ LOADADDR=0x60008000
#copy necessary files to sdcard
cp arch/arm/boot/zImage /mnt/sdcard
cp arch/arm/boot/uImage /mnt/sdcard
cp arch/arm/boot/dts/*.dtb /mnt/sdcard
#Build dynamic modules and copy to suitable destination
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- modules
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- modules_install \
INSTALL_MODPATH=<mount point of rootfs>
```

You may skip the last two steps for the moment, as the given configuration steps avoid dynamic modules. All the necessary modules are configured as static.



Getting rootfs

We require a file system to work with the kernel we’ ve built. Download the pre-built rootfs image to test with QEMU from the following link: http://downloads.yoctoproject.org/releases/yocto/yocto-1.5.2/machines/qemu/qemuarm/core-image-minimal-qemuarm.ext3 and copy it to the SD card (/mnt/image) by renaming it as rootfs.img for easy usage. You may obtain the rootfs image from some other repository or build it from sources using Busybox.

Your first try

Let’ s boot this kernel image (zImage) directly without u-boot, as follows:

```
export PATH=/opt/qemu-arm/bin:$PATH
qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio \
-kernel /mnt/sdcard/zImage \
-dtb /mnt/sdcard/vexpress-v2p-ca9.dtb \
-initrd /mnt/sdcard/rootfs.img -append “ root=/dev/ram0 console=ttyAMA0”
```

In the above command, we are treating rootfs as ‘ initrd image’ , which is fine when rootfs is of a small size. You can connect larger file systems in the form of a hard disk or SD card. Let’ s try out rootfs through an SD card:

```
qemu-system-arm -M vexpress-a9 -m 1024 -serial stdio \
-kernel /mnt/sdcard/zImage \
-dtb /mnt/sdcard/vexpress-v2p-ca9.dtb \
-sd /mnt/sdcard/rootfs.img -append “ root=/dev/mmcblk0 console=ttyAMA0”
```

In case the sdcard/image file holds a valid partition table, we need to refer to the individual partitions like /dev/mmcblk0p1, /dev/mmcblk0p2, etc. Since the current image file is not partitioned, we can refer to it by the device file name /dev/mmcblk0.

Building u-boot

Switch back to the u-boot directory (u-boot-2014.04), build u-boot as follows and copy it to the SD card:

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- vexpress_ca9x4_config
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
cp u-boot /mnt/image
# you can go for a quick test of generated u-boot as follows
qemu-system-arm -M vexpress-a9 -kernel /mnt/sdcard/u-boot -serial stdio
```

Let’s ignore errors such as ‘ u-boot couldn’t locate kernel image’ or any other suitable files.

The final steps

Let’s boot the system with u-boot using an image file such as SD card, and make sure the QEMU PATH is not disturbed.

Unmount the SD card image and then boot using QEMU.

```
umount /mnt/sdcard
qemu-system-arm -M vexpress-a9 -sd sdcard.img -m 1024 -serial stdio -kernel u-boot
```

You can stop autoboot by hitting any key within the time limit and enter the following commands at the u-boot prompt to load rootfs.img, uimage, dtb files from the SD card to suitable memory locations without overlapping. Also, set the kernel boot parameters using setenv as shown below (here, 0x82000000 stands for the location of the loaded rootfs image and 8388608 is the size of the rootfs image).

Note: The following commands are internal to u-boot and must be entered within the u-boot prompt.

```
fatls mmc 0:0 #list out partition contents
fatload mmc 0:0 0x82000000 rootfs.img # note down the size of image being loaded
fatload mmc 0:0 0x80200000 uImage
fatload mmc 0:0 0x80100000 vexpress-v2p-ca9.dtb
setenv bootargs 'console=ttyAMA0 root=/dev/ram0 rw initrd=0x82000000,8388608'
bootm 0x80200000 - 0x80100000
```

Ensure a space before and after the '–'' – ' symbol in the above command.

Log in using ' root' as the username and a blank password to play around with the system.
I hope this article proves useful for bootstrapping with embedded Linux and for teaching the concepts when there is no hardware available.

What is Linux Kernel Porting?

One of the aspects of hacking a Linux kernel is to port it. While this might sound difficult, it won' t be once you read this article. The author explains porting techniques in a simplified manner.
With the evolution of embedded systems, porting has become extremely important. Whenever you have new hardware at hand, the first and the most critical thing to be done is porting. For hobbyists, what has made this even more interesting is the open source nature of the Linux kernel. So, let' s dive into porting and understand the nitty-gritty of it.

Porting means making something work on an environment it is not designed for. Embedded Linux porting means making Linux work on an embedded platform, for which it was not designed. Porting is a broader term and when I say ' embedded Linux porting' , it not only involves Linux kernel porting, but also porting a first stage bootloader, a second stage bootloader and, last but not the least, the applications. Porting differs from development. Usually, porting doesn't involve as much of coding as in development. This means that there is already some code available and it only needs to be fine-tuned to the desired target. There may be a need to change a few lines here and there, before it is up and running. But, the key thing to know is, what needs to be changed and where.

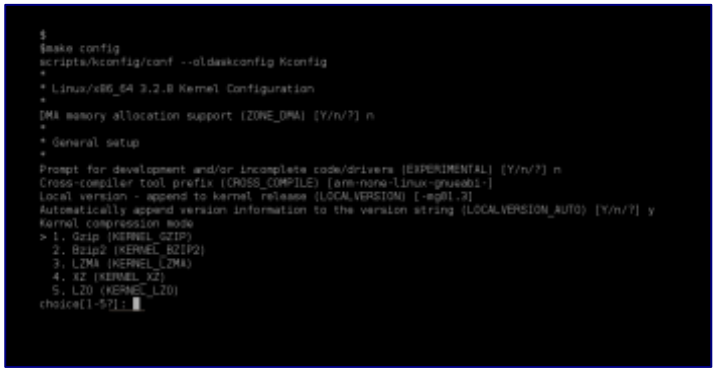
What Linux kernel porting involves
Linux kernel porting involves two things at a higher level: architecture porting and board porting. Architecture, in Linux terminology, refers to CPU. So, architecture porting means adapting the Linux kernel to the target CPU, which may be ARM, Power PC, MIPS, and so on. In addition to this, SOC porting can also be considered as part of architecture porting. As far as the Linux kernel is concerned, most of the times, you don't need to port it for architecture as this would already be supported in Linux. However, you still need to port Linux for the board and this is where the major focus lies. Architecture porting entails porting of initial start-up code, interrupt service routines, dispatcher routine, timer routine, memory management, and so on. Whereas board porting involves writing custom drivers and initialisation code for devices specific to the board.

Building a Linux kernel for the target platform

Kernel building is a two-step process: first, the kernel needs to be configured for the target platform. There are many ways to configure the kernel, based on the preferred configuration interface. Given below are some of the common methods.

To run the text-based configuration, execute the following command:

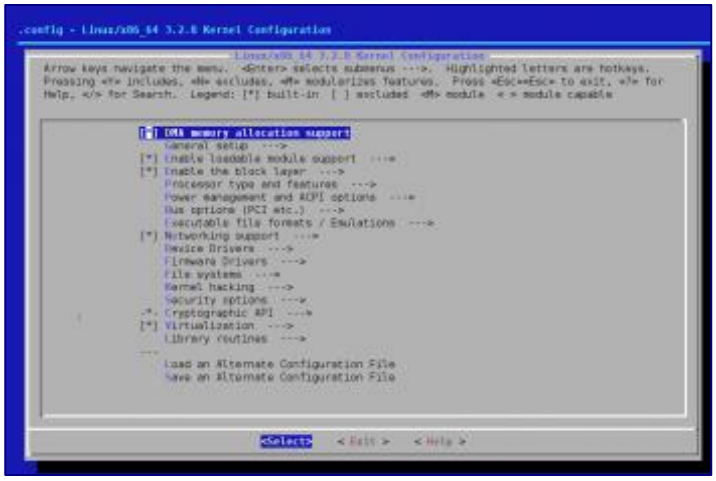
$ make config



This will show the configuration options on the console as seen in Figure 1. It is a little cumbersome to configure the kernel with this, as it prompts every configuration option, in order, and doesn't allow the reversion of changes.

To run the menu-driven configuration, execute the following command:

$ make menuconfig



This will show the menu options for configuring the kernel, as seen in Figure 2. This requires the ncurses library to be installed on the system. This is the most popular interface used to configure the kernel.

To run the window-based configuration, execute the following command:

$ make xconfig

This allows configuration using the mouse. It requires QT to be installed on the system.
For details on other options, execute the following command in the kernel top directory:

$ make help

Once the kernel is configured, the next step is to build the kernel with the make command. A few commonly used commands are given below:

$ make vmlinux - Builds the bare kernel
$ make modules - Builds the modules
$ make modules_prepare – Sets up the kernel for building the modules external to kernel.

If the above commands are executed as stated, the kernel will be configured and compiled for the host system, which is generally the x86 platform. But, for porting, the intention is to configure and build the kernel for the target platform, which in turn, requires configuration of makefile. Two things that need to be changed in the makefile are given below:

ARCH=<architecture>
CROSS-COMPILE = <toolchain prefix>

The first line defines the architecture the kernel needs to be built for, and the second line defines the cross compilation toolchain prefix. So, if the architecture is ARM and the toolchain is say, from CodeSourcery, then it would be:

ARCH=arm
CROSS_COMPILE=arm-none-linux-gnueabi-

Optionally, make can be invoked as shown below:

$ make ARCH=arm menuconfig - For configuring the kernel
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- - For compiling the kernel

The kernel image generated after the compilation is usually vmlinux, which is in ELF format. This image can’t be used directly with embedded system bootloaders such as u-boot. So convert it into the format suitable for a second stage bootloader. Conversion is a two-step process and is done with the following commands:

arm-none-linux-gnueabi-objcopy -O binary vmlinux vmlinux.bin
mkimage -A arm -O linux -T kernel -C none -a 0x80008000 -e 0x80008000 -n linux-3.2.8 -d vmlinux.bin uImage
-A ==> set architecture
-O ==> set operating system
-T ==> set image type
-C ==> set compression type
-a ==> set load address (hex)
-e ==> set entry point (hex)
-n ==> set image name
-d ==> use image data from file

The first command converts the ELF into a raw binary. This binary is then passed to mkimage, which is a utility to generate the u-boot specific kernel image. mkimage is the utility provided by u-boot. The generated kernel image is named uImage.

The Linux kernel build system

One of the beautiful things about the Linux kernel is that it is highly configurable and the same code base can be used for a variety of applications, ranging from high end servers to tiny embedded devices. And the infrastructure, which plays an important role in achieving this in an efficient manner, is the kernel build system, also known as kbuild. The kernel build system has two main components – makefile and Kconfig.

Makefile: Every sub-directory has its own makefile, which is used to compile the files in that directory and generate the object code out of that. The top level makefile percolates recursively into its sub-directories and invokes the corresponding makefile to build the modules and finally, the Linux kernel image. The makefile builds only the files for which the configuration option is enabled through the configuration tool.

Kconfig: As with the makefile, every sub-directory has a Kconfig file. Kconfig is in configuration language and Kconfig files located inside each sub-directory are the programs. Kconfig contains the entries, which are read by configuration targets such as make menuconfig to show a menu-like structure.

So we have covered makefile and Kconfig and at present they seem to be pretty much disconnected. For kbuild to work properly, there has to be some link between the Kconfig and makefile. And that link is nothing but the configuration symbols, which generally have a prefix CONFIG_. These symbols are generated by a configuration target such as menuconfig, based on entries defined in the Kconfig file. And based on what the user has selected in the menu, these symbols can have the values □y', □n', or □m'.

Now, as most of us are aware, Linux supports hot plugging of the drivers, which means, we can dynamically add and remove the drivers from the running kernel. The drivers which can be added/removed dynamically are known as modules. However, drivers that are part of the kernel image can't be removed dynamically. So, there are two ways to have a driver in the kernel. One is to build it as a part of the kernel, and the other is to build it separately as a module for hot-plugging. The value □y' for CONFIG_, means the corresponding driver will be part of the kernel image; the value □m' means it will be built as a module and value □n' means it won't be built at all. Where are these values stored? There is a file called .config in the top level directory, which holds these values. So, the .config file is the output of the configuration target such as menuconfig.

Where are these symbols used? In makefile, as shown below:

```
obj-$(CONFIG_MYDRIVER) += my_driver.o
```

So, if CONFIG_MYDRIVER is set to value □y', the driver my_driver.c will be built as part of the kernel image and if set to value □m', it will be built as a module with the extension .ko. And, for value □n', it won't be compiled at all.

As you now know a little more about kbuild, let' s consider adding a simple character driver to the kernel tree.

The first step is to write a driver and place it at the correct location. I have a file named my_driver.c. Since it’ s a character driver, I will prefer adding it at the drivers/char/ sub-directory. So copy this at the location drivers/char in the kernel.
The next step is to add a configuration entry in the drivers/char/Kconfig file. Each entry can be of type bool, tristate, int, string or hex. bool means that the configuration symbol can have the values ‘ y’ or □n’, while tristate means it can have values □y’, □m’ or □n’. And □int’, □string’ and ‘ hex’ mean that the value can be an integer, string or hexadecimal, respectively. Given below is the segment of code added in drivers/char/Kconfig:

```
config MY_DRIVER
tristate "Demo for My Driver"
default m
help
Adding this small driver to kernel for
demonstrating the kbuild
```

The first line defines the configuration symbol. The second specifies the type for the symbol and the text which will be shown as the menu. The third specifies the default value for this symbol and the last two lines are for the help message. Another thing that you will generally find in a Kconfig file is ‘ depends on’. This is very useful when you want to select the particular feature, only if its dependency is selected. For example, if we are writing a driver for i2c EEPROM, then the menu option for the driver should appear only if the 2c driver is selected. This can be achieved with the ‘ depends on’ entry.

After saving the above changes in Kconfig, execute the following command:

```
$ make menuconfig
```

Now, navigate to Device Drivers->Character devices and you will see an entry for My Driver.
By default, it is supposed to be built as a module. Once you are done with configuration, exit the menu and save the configuration. This saves the configuration in .config file. Now, open the .config file, and there will be an entry as shown below:

```
CONFIG_MY_DRIVER=
m
```

Here, the driver is configured to be built as a module. Also, one thing worth noting is that the symbol □MY_DRIVER’ in Kconfig is prefixed with CONFIG_.
Now, just adding an entry in the Kconfig file and configuration alone won’t compile the driver. There has to be the corresponding change in makefile as well. So, add the following line to makefile:

```
obj-$(CONFIG_MYDRIVER) += my_driver.o
```

After the kernel is compiled, the module my_driver.ko will be placed at drivers/char/. This module can be inserted in the kernel with the following command:

```
$ insmod my_driver.ko
```

Aren’t these configuration symbols needed in the C code? Yes, or else how will the conditional compilation be taken care of? How are these symbols included in C code? During the kernel compilation, the Kconfig and .config files are read, and are used to generate the C header file named autoconf.h. This is placed at include/generated and contains the #defines for the configuration symbols. These symbols are used by the C code to conditionally compile the required code.

Now, let’ s suppose I have configured the kernel and that it works fine with this configuration. And, if I make some new changes in the kernel configuration, the earlier ones will be overwritten. In order to avoid this from happening, we can save .config file in the arch/arm/configs directory with a name like my_config, for instance. And next time, we can execute the following command to configure the kernel with older options:

$ make my_config_defconfig

Linux Support Packages (LSP)/Board Support Packages (BSP)

One of the most important and probably the most challenging thing in porting is the development of Board Support Packages (BSP). BSP development is a one-time effort during the product development lifecycle and, obviously, the most critical. As we have discussed, porting involves architecture porting and board porting. Board porting involves board-specific initialisation code that includes initialisation of the various interfaces such as memory, peripherals such as serial, and i2c, which in turn, involves the driver porting.

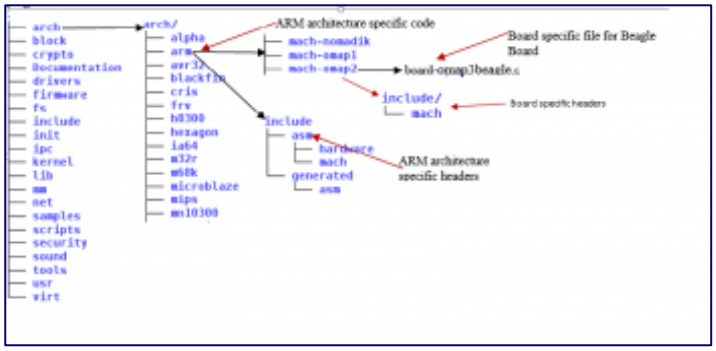
My Driver

There are two categories of drivers. One is the standard device driver such as the i2c driver and block driver located at the standard directory location. Another is the custom interface or device driver, which includes the board-specific custom code and needs to be specifically brought in with the kernel. And this collection of board-specific initialisation and custom code is referred to as a Board Support Package or, in Linux terminology, a LSP. In simple words, whatever software code you require (which is specific to the target platform) to boot up the target with the operating system can be called LSP.

Components of LSP

As the name itself suggests, BSP is dependent on the things that are specific to the target board. So, it consists of the code which is specific to that particular board, and it applies only to that board. The usual list includes Interrupt Request Numbers (IRQ), which are dependent on how the various devices are connected on the board. Also, some boards have an audio codec and you need to have a

driver for that codec. Likewise, there would be switch interfaces, a matrix keypad, external eeprom, and so on.



LSP placement

LSP is placed under a specific <arch> folder of the kernel's arch folder. For example, architecture-specific code for ARM resides in the arch/arm directory. This is about the code, but you also need the headers which are placed under arch/arm/include/asm. However, board-specific code is placed at arch/arm/mach-<board_name> and corresponding headers are placed at arch/arm/mach-<soc architecture>/include. For example, LSP for Beagle Board is placed at arch/arm/mach-omap2/board-omap3beagle.c and corresponding headers are placed at arch/arm/mach-omap2/include/mach/. This is shown in figure 4.

Machine ID

Every board in the kernel is identified by a machine ID. This helps the kernel maintainers to manage the boards based on ARM architecture in the source tree. This ID is passed to the kernel from the second stage bootloader such as u-boot. For the kernel to boot properly, there has to be a match between the kernel and the second stage boot loader. This information is available in arch/arm/tools/mach-types and is used to generate the file linux/include/generated/mach-types.h. The macros defined by mach-types.h are used by the rest of the kernel code. For example, the machine ID for Beagle Board is 1546, and this is the number which the second stage bootloader passes to the kernel. For registering the new board for ARM, provide the board details at http://www.arm.linux.org.uk/developer/machines/?action=new.

Note: The porting concepts described over here are specific to boards based on the ARM platform and may differ for other architectures.

MACHINE_START macro

One of the steps involved in kernel porting is to define the initialisation functions for the various interfaces on the board, such as serial, Ethernet, Gpio, etc. Once these functions are defined, they need to be linked with the kernel so that it can invoke them during boot-up. For this, the kernel provides the macro MACHINE_START. Typically, a MACHINE_START macro looks like what's shown below:

```
MACHINE_START(MY_BOARD, "My Board for Demo")
.atag_offset = 0x100,
.init_early = my_board_early,
.init_irq = my_board_irq,
.init_machine = my_board_init,
MACHINE_END
```

Let’s understand this macro. MY_BOARD is machine ID defined in arch/arm/tools/mach-types. The second parameter to the macro is a string describing the board. The next few lines specify the various initialisation functions, which the kernel has to invoke during boot-up. These include the following:

.atag_offset: Defines the offset in RAM, where the boot parameters will be placed. These parameters are passed from the second stage bootloader, such as u-boot.

my_board_early: Calls the SOC initialisation functions. This function will be defined by the SOC vendor, if the kernel is ported for it.

my_board_irq: Intialisation related to interrupts is done over here.

my_board_init: All the board-specific initialisation is done here. This function should be defined during the board porting. This includes things such as setting up the pin multiplexing, initialisation of the serial console, initialisation of RAM, initialisation of Ethernet, USB and so on.

MACHINE_END ends the macro. This macro is defined in arch/arm/include/asm/mach/arch.h.

How to begin with porting

The most common and recommended way to begin with porting is to start with some reference board, which closely resembles yours. So, if you are porting for a board based on OMAP3 architecture, take Beagle Board as a reference. Also, for porting, you should understand the system very well. Depending on the features available on your board, configure the kernel accordingly. To start with, just enable the minimal set of features required to boot the kernel. This may include but not be limited to initialisation of RAM, Gpio subsystems, serial interfaces, and filesystems drivers for mounting the root filesystem. Once the kernel boots up with the minimal configuration, start adding the new features, as required.

So, let’ s summarise the steps involved in porting:

1. The first step is to register the machine with the kernel maintainer and get the unique ID for your board. While this is not necessary to begin with porting, it needs to be done eventually, if patches are to be submitted to the mainline. Place the machine ID in arch/arm/tools/mach-types.

2. Create the board-specific file ‘ board-<board_name>’ at arch/arm/mach-<soc> and define the MACHINE_START for the new board. For example, the board-specific file for the Panda Board resides at arch/arm/mach-omap2/board-omap4panda.c.

3. Update the Kconfig file at arch/arm/mach_<soc> to add an entry for the new board as shown below:

```
config MACH_MY_BOARD
bool “ My Board for Demo”
depends on ARCH_OMAP3
default y
```

4. Update the corresponding makefile, so that the board-specific file gets compiled. This is shown below:

```
obj-$(CONFIG_MACH_MY_BOARD) += board-my_board.o
```

5. Create a default configuration file for the new board. To begin with, take any .config file as a starting point and customise it for the new board. Place the working .config file at arch/arm/configs/my_board_defconfig.

Writing a Basic Framebuffer Driver

This article deals with the basic structure of a framebuffer and will interest those who know how to write a character device driver. The author has tried to simplify the topic as much as possible so as to make it accessible to more readers.
A framebuffer driver is an intermediate layer in Linux, which hides the complexities of the underlying video device from the user space applications. From the point of view of the user space, if the display device needs to be accessed for reading or writing, then only the framebuffer device such as /dev/fb0 has to be accessed. If you have more than one video card in your computer, then each will be assigned separate device nodes like /dev/fb0.. /dev/fb1.. /dev/fbX (X being the minor number of the device).

We also have many different ways to access a framebuffer. To get a snapshot of your screen, you can just use the cp command, as follows:

```
cp /dev/fb0 myscreen
```

From a programmer’ s point of view, you can read and write to this device, the main use being mmap. You can map the video memory in the user space memory, and then you can control each and every pixel of your screen. We also have a set of ioctl calls to get and set different parameters of the video hardware. A few ioctl commands include FBIOGET_VSCREENINFO and FBIOPUT_VSCREENINFO. As you may have guessed, the FBIOGET_VSCREENINFO call will ask the hardware about the screen information, and FBIOPUT_VSCREENINFO will set the screen information. Some other ioctl commands include FBIOGETCMAP, FBIOPAN_DISPLAY, etc. You can find the list of commands in the uapi/linux/fb.h file in the kernel source.

Now, let’ s look at how to write a module that will work as a very minimal framebuffer driver, with only the required components.
We need a few header files in our module, and with the basic structure of a module our starting point becomes…

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/string.h>
#include <linux/mm.h>
```

```
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/fb.h>
#include <linux/init.h>
#include <linux/pci.h>

static int __init ourfb_init(void)
{
return pci_register_driver(&ourfb_driver);
}
static void __exit ourfb_exit(void)
{
pci_unregister_driver(&ourfb_driver);
}
module_init(ourfb_init);
module_exit(ourfb_exit);
MODULE_LICENSE(“ GPL” );
```

So now we have a very basic module with the required init and exit function, but we have not yet defined ‘ ourfb_driver’ which we are using in the code to register in the PCI subsystem.

```
static struct pci_driver ourfb_driver = {
.name = “ ourfb” ,
.id_table = ourfb_pci_tbl,
.probe = ourfb_pci_init,
.remove = ourfb_pci_remove,
};
```

In the driver structure, we have defined only the required part. If you want to use power management in your driver code and provide Suspend and Resume functionality to your hardware, then you can define that in the driver. But here we are leaving them out as they are optional.
In ourfb_pci_tbl, we have to mention the vendor ID and the device ID of the hardware for which you are writing the driver.

```
static struct pci_device_id ourfb_pci_tbl[] = {
{ PCI_VENDOR_ID_XXX, PCI_DEVICE_ID_XXX, PCI_ANY_ID, PCI_ANY_ID },
{ 0 }
};
MODULE_DEVICE_TABLE(pci, ourfb_pci_tbl);
```

We are assuming that PCI_VENDOR_ID_XXX and PCI_DEVICE_ID_XXX are defined elsewhere with the correct vendor and device ID. This table should always be null terminated. To explain MODULE_DEVICE_TABLE in a very simple way – - it informs the kernel that this driver is to be used if it finds any PCI device with the mentioned vendor and device IDs. When the kernel finds the device, it calls the probe callback function; and if it sees that the device is removed, then it will call the remove callback function. A detailed discussion on PCI drivers is outside the scope of this article.

```
static int ourfb_pci_init(struct pci_dev *dev, const struct pci_device_id *ent)
{
struct fb_info *info;
```

```
struct ourfb_par *par;
struct device *device = &dev->dev;
int cmap_len, retval;
info = framebuffer_alloc(sizeof(struct ourfb_par), device);
if (!info) {
return -ENOMEM;
}
par = info->par;
info->screen_base = framebuffer_virtual_memory;
info->fbops = &ourfb_ops;
info->fix = ourfb_fix;
info->pseudo_palette = pseudo_palette;
info->flags = FBINFO_DEFAULT;
if (fb_alloc_cmap(&info->cmap, cmap_len, 0))
return -ENOMEM;
if (register_framebuffer(info) < 0) {
fb_dealloc_cmap(&info->cmap);
return -EINVAL;
}
pci_set_drvdata(dev, info);
return 0;
}
static void ourfb_pci_remove(struct pci_dev *dp)
{
struct fb_info *p = pci_get_drvdata(dp);
unregister_framebuffer(p);
fb_dealloc_cmap(&p->cmap);
iounmap(p->screen_base);
framebuffer_release(p);
}
```

The above code is an example of a very simple probe function. Now, let's talk a little about what we have used in our probe function.

framebuffer_alloc: This creates a new framebuffer information structure. It also reserves the size for the driver's private data (info->par). You might have guessed by now that ourfb_par is our private data. This function will return a pointer to struct fb_info or it returns NULL if it fails.

ourfb_par: As already said, this is the private data of our driver. It is not a required component but it is recommended that you have it in your driver. At any point of time, this structure will have information about the hardware state of the graphics card.

framebuffer_virtual_memory: This is not defined in our code, but it is actually the virtual memory address for the framebuffer. We get this address after remapping the resource address. The following code will show you how we get this address:

```
unsigned long addr, size;
addr = pci_resource_start(dev, 0);
size = pci_resource_len(dev, 0);
if (addr == 0)
return -ENODEV;
framebuffer_virtual_memory = ioremap(addr, 0x800000);
```

ourfb_ops: This structure will define the operations that we will allow on our framebuffer. We can have many operations here which are similar to any character device driver, like open, read, write, release, ioctl, etc. But the operations that we have to provide are fb_fillrect, fb_copyarea and fb_imageblit. Regarding the other operations, even if we do not mention them in our ops structure, they will be taken care of automatically by the framebuffer layer by default. If we need any customised functions for our driver, then we have to provide for them also.

Fortunately, the framebuffer layer has three functions: cfb_fillrect, cfb_imageblit and cfb_copyarea. We can use these functions in our driver instead of writing our own fb_fillrect, fb_copyarea and fb_imageblit functions. But if you are writing a driver for hardware that supports hardware acceleration, then you have to write your own functions.

```
static struct fb_ops ourfb_ops = {
.owner = THIS_MODULE,
.fb_fillrect = cfb_fillrect,
.fb_imageblit = cfb_imageblit,
.fb_copyarea = cfb_copyarea,
};
```

ourfb_fix: This structure will contain fixed information about our hardware. The following is the structure as defined in the header file:

```
struct fb_fix_screeninfo {
char id[16]; /* identification string */
unsigned long smem_start; /* Start of frame buffer mem (physical address) */
__u32 smem_len; /* Length of frame buffer mem */
__u32 type; /* FB_TYPE_* */
__u32 type_aux; /* Interleave for interleaved Planes */
__u32 visual; /* FB_VISUAL_* */
__u16 xpanstep; /* zero if no hardware panning */
__u16 ypanstep; /* zero if no hardware panning */
__u16 ywrapstep; /* zero if no hardware ywrap */
__u32 line_length; /* length of a line in bytes */
unsigned long mmio_start; /* Start of Memory Mapped I/O (physical address) */
__u32 mmio_len; /* Length of Memory Mapped I/O */
__u32 accel; /* Indicate to driver which specific chip/card we have */
__u16 capabilities; /* FB_CAP_* */
__u16 reserved[2]; /* Reserved for future compatibility */
};
```

flags: This will indicate the type of hardware acceleration our driver is capable of.
fb_alloc_cmap: This allocates memory for the colour map, which our driver is going to use.
register_framebuffer: This is the most important function that will actually register our framebuffer with the framebuffer layer. This function is responsible for creating the device nodes /dev/fb*.

Note: In the probe function, at any stage, if the initialisation fails, we have to undo whatever we have done before that, and then need to return an appropriate error value.

In the ourfb_remove function, we are unwinding what we have done in the probe function in a reverse manner.

We have just looked at how to create a very simple framebuffer device, though we have not touched upon any hardware related topics here. But when we work with an actual framebuffer driver that is responsible for video hardware, we need to give all the hardware initialisation routines in the probe function. If you want to see the code of any framebuffer device driver, you can find it in the drivers/video/fbdev/ folder of the Linux source tree.

Talking to the Kernel through Sysfs

The Linux kernel provides a virtual file system called sysfs. By providing virtual files, sysfs is able to export information about various kernel sub-systems, hardware devices and associated device drivers from the kernel’s device model to user space. To further explore sysfs, dive deep into this article.

There is a need to provide information related to each process, to the user space, which can then be used by programs such as ps. The /proc file system was created for this purpose. Through the proc file system, each process has its directory in the /proc folder. It was originally designed to provide process related information to user space. Adding directories and files to the /proc file system is easier; so, many kernel sub-systems started using this file system for displaying information to the user space. It is also used to take inputs from the user space to control settings inside the kernel modules. But the /proc file system is getting cluttered with lots of non-process related information. From the Linux 2.5 development cycle, a new interface called the /sys file system has been introduced. Sysfs is a RAM based file system. It is designed to export the kernel data structures and their attributes from the kernel to the user space, which then avoids cluttering the /proc file system. The advantages of sysfs over procfs are as follows:

- A cleaner, well-documented programming interface
- Automatic clean-up of directories and files, when the device is removed from the system
- The enforced one item per file rule, which makes for a cleaner user interface

The ‘one item per file’ rule mandates that in each file of sysfs, there will be only one value that can be put in or read from it. This feature really makes it a cleaner interface. Through sysfs, user space programs can get information from the kernel sub-system like device drivers. Programs can also send values to the kernel sub-system and can control the internal settings. This is how programs talk to the Linux kernel.

Sysfs mounting

By default, sysfs is compiled in the Linux kernel. It is dependent on CONFIG_SYSFS being enabled in the kernel configuration. If sysfs is not already mounted, then you can do so by using the following command:

mount -t sysfs sysfs /sys

Linux Kernel objects

Folders in each sysfs directory are represented by kernel objects (kobjects) in Linux. These are represented in the kernel through the struct kobject (defined in include/linux/kobject.h). The important members of the struct kobject are given below.

char *name: This is the name of the kobject. Folders corresponding to the current kobject are created with this name in sysfs.

Struct kobject *parent: This is the parent of the current kobject being created. When a folder is created in sysfs, if this field is present, then the current kobject folder is created inside the parent kobject folder.

Struct kref: This is the reference counting mechanism for kobject. Whenever any kernel module refers to any kobject, its reference count is incremented, and whenever any kobject reference is released by any kernel module, then the reference count is decremented. When the reference count decrements to zero, memory related to the kobject is released.

Directory operations in sysfs

The following are the interfaces that are used for directory related operations in sysfs:

int sysfs_create_dir(struct kobject *kobj);

void sysfs_remove_dir(struct kobject *kobj);

int sysfs_rename_dir(struct kobject *kobj. const char *new_name);

Sysfs directory creation

The sysfs_create_dir interface is used to create a directory in sysfs. The struct kobject parameter passed to it has two fields in it.

kobj->name: This is the name of the directory that is to be created in sysfs.

kobj->parent: This is the kobject pointer of the parent directory in which the current directory is to be created.

Since each directory in sysfs has a struct kobject associated with it, when the new directory is to be created in sysfs, the struct kobject pointer that is passed to sysfs_create_dir should have the parent kobject pointer pointing to the kobject of the directory in which the current directory is to be created. Then the name parameter of the kobject should be filled, with the proper name representing the functionality for which the directory is being created.

But the above interfaces are not directly used to create directories in sysfs when writing kernel modules; instead, the following interface is used:

struct kobject * kobject_create_and_add(const char *name, struct kobject *parent);

…where,

name: is the name of the directory to be created in sysfs, and…

parent: is the kobject of the parent directory in which the current directory is to be created.

The return value of the kobject_create_and_add is the kobject pointer of the created directory, and if it fails to create a directory then a NULL pointer is returned.

Kobject parent pointer: Since each folder in the sysfs file system is represented using a struct

kobject, to create a new directory in sysfs, the parent pointer is to be initialised with the kobject of the directory in which the current directory is to be created. For example, to create a directory named example_dev in /sys/kernel, the following function call is used:

struct kobject *example_dev_kob;

example_dev_kobj = kobject_create_and_add(“ example_dev” , kernel_kobj);

…where, kernel_kobj is the kobject pointer of the kernel directory in sysfs. So in order to create a directory in /sys/firmware, the firmware_kobj pointer will be used.
In order to create the directory named example_dev directly in /sys, the following function call is used:

kobject_create_and_add(“ example_dev” , NULL);

…so if the parent pointer is kept NULL, then the example_dev directory will be created in the /sys folder.

Sysfs directory removal
The sysfs_remove_dir interface is used to remove the directory from the sysfs file system. The parameter that is passed to the sysfs_remove_dir is the kobject pointer of the directory which is to be removed. The kobject pointer should be used to create the directory. Here, too, the same concept applies. Instead of directly calling this interface to remove the directory, the following kernel kobject interface is used, which internally calls the sysfs remove interface:

void kobject_del(struct kobject *kobj);

This interface will delete the directory of sysfs represented by the kobject pointer passed to it.
For example, to delete the example_dev directory that was created as an example, the following function call is used:

kobject_del(example_dev_kobj);

Sysfs directory renaming
In case there is a need to rename the directory created in sysfs, the following functions are available:

int sysfs_rename_dir(struct kobject *kobj, const char *new_name)

…where, kobj is the kobject pointer corresponding to the directory that has to be renamed, and new_name is to be given to the directory. Since the sysfs interface is normally not directly called, the following kobject interface is called for the renaming purpose:

int kobject_rename(struct kobject *kobj, const char *new_name)

…where,
kobject: This is the kobject pointer of the directory to be renamed.
new_name: This is the new name to be given to the directory.

Kobject reference counting
Each kobject has a member struct kref, which is a reference counter. Whenever any module wishes

to use the kobject already created, this reference count is incremented, and whenever that module wishes to stop using kobject, the reference count is decremented to maintain the number of modules using the kobject or the sysfs directory represented by the kobject. Whenever the reference count reaches 0, the memory related to the kobject which was allotted by kobject_create_and_add is released, and whenever a kobject is created using kobject_create_and_add, it internally initialises the kref reference counter by one. This indicates the kobject is being used by the current module, and then kobject_create_and_add internally creates the directory for the kobject. Whenever any new module wishes to use the directory represented by that kobject, the reference counter has to be incremented.
The reference count is maintained in the kobject using the following functions:

```
struct kobject* kobject_get(struct kobject *kobj);

struct kobject* kobject_put(struct kobject *kobj);
```

kobject_get increments the reference count of the kobject passed to it, whereas kobject_put decrements the reference count of the kobject passed to it. So while writing drivers or any kernel module, whenever any kobject of the other module is directly used, kobject_get should be called to increment the reference count of the kobject. When the use of the kobject is finished, kobject_put should be called to decrement its reference count.
Whenever kobject_put is called, passing the kobject pointer to it, it decrements the reference count. When the reference count reaches zero, it frees the memory of the kobject pointer, and the kobject pointer can no longer be used.

Files in sysfs
In the above example, we have created the example_dev directory in the /sys/kernel folder. Suppose the example had some attribute called example_info which we want to expose to the user space using sysfs, then the feature used is called sysfs attributes.
Attributes are represented as regular files in sysfs with one value per file. Attributes are of the type struct attribute (defined in include/linux/sysfs.h). Struct attribute has the following two important pieces of information.
name: This is the name of the attribute(file) to be created.
mode: This is the permission with which the file is to be created. A simple attribute has no means by which it can be read or written; it needs wrapper routines for reading and writing. For this purpose, kobject defines a special structure called struct kobj_attribute (defined in include/linux/kobject.h) as follows:

```
struct kobj_attribute {

struct attribute attr;

ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);

ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count);
};
```

…where, attr is the attribute representing the file to be created, show is the pointer to the function that will be called when the file is read in sysfs, and store is the pointer to the function which will be

called when the file is written in sysfs.
Suppose we need to create our example_info file in the /sysfs/kernel/example_dev directory that was created as an example; then the following method is the way to define the attribute:

```
struct kobj_attribute example_info_attr =
__ATTR(example_info, 0666, example_show, example_store);
```

…where __ATTR is the macro, as shown below:

```
__ATTR(name, permission, show_ptr, store_ptr)
```

…where, name is the name with which the attribute is to be created, permission is file permission where 0666 is read and write permission, show_ptr is the function pointer called when the file is read, and store_ptr is the function pointer called when the file is written.
The example_attr defined above is then grouped in the struct attribute group as follows:

```
struct attribute *example_attrs[] = {
&example_attr.attr,
NULL,
};
```

Note: In the struct attribute, multiple attributes can be grouped together and then be created in one single API call. Also, NULL termination is compulsory for the group.

The above attributes are then given to the attribute group as follows:

```
struct attribute_group example_attr_group = {
.attrs = example_attrs,
};
```

To create attributes, the following sysfs API is used:

```
int sysfs_create_group(struct kobject *kobj, const struct attribute_group *grp);
```

…where,
kobj: This is the pointer to the kobject of the directory in which the file is to be created.
grp: This is the group of attributes to be created, which will be created as files in sysfs.
To create the example_info file in our /sys/kernel/example_dev directory, the following sysfs API is called:

```
sysfs_create_group(example_dev_kobj,&example_attr_group);
```

After this function, a file named example_info will be created in the sys/kernel/example_dev directory.
Now, whenever the file is read, the example_show function will be called and whenever the file is written, the example_store function will be called, which needs to be defined. The function prototype for the show and store function is as follows:

```
ssize_t (*show)(struct kobject * kobj, struct attribute * attr, char * buff);
```

…where,
kobj: This is the pointer to the kobject of the directory which was passed when the file was created.
attr: This is the attribute pointer of the file which was passed when file was created.
buff: This is the buffer in which the output should be written, which will be displayed as the result in user space. The value written should not exceed the size defined by the PAGE_SIZE macro.

```
ssize_t (*store)(struct kobject * kobj, struct attribute * attr, const char * buff, size_t size);
```

…where, kobject and attr are as described above.
buff: This buffer contains the value that was written to the file from user space.
size: This is the size of the data written to the file through user space.
So our example_show function can be defined as follows:

```
ssize_t example_show(struct kobject *kobj, struct kobj_attribute *attr, char *buf)
{
return sprintf(buf, " example show" );
/*Here any value of variable can be written and exposed to user space through this interface
also return value is the size of data written to the buffer*/
}
```

…and an example store can be written as follows:

```
static ssize_t example_store(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t
count)
{
//here the value from the buff can be read and //respective action can be taken by module

return count;

//return the number of bytes read, if all the //bytes are read the count parameter which //was
received by store routine
}
```

Driver sysfs interface

Drivers are modules written to interface user space with hardware devices. Hardware devices have information, which the user needs to know during their operation. So drivers generally create sysfs files for providing their information to user space.
Drivers don' t explicitly create directories in sysfs. Whenever drivers register with their sub-system, directories are internally created for the driver. Each driver has access to the struct device pointer for the device for which the driver is being written. This device pointer is used to create files in the driver interface.
Drivers create one file for each piece of device information that is to be exposed to the user space through sysfs. Drivers use a special API for file creation in the user space, and this is as follows:

```
int device_create_file(struct device *dev, const struct device_attribute *attr);
```

…where,
dev: This is the pointer to the device structure of the driver module.
attr: This is the attribute which is to be created.
For removal of the file from the sysfs directory, the following interface can be used:

void device_remove_file (struct device *dev, const struct device_attribute *attr)

Sysfs **object relationship**

Let’ s suppose there is the directory /sys/kernel/example1, the kobject pointer of which is ex1_kobj and there is a need to create a link file in the /sys/kernel/example2 directory named ‘ ex2’ , which has to show the relationship with the example1 directory. This can be accomplished by symbolic links in sysfs. Also, the kobject pointer of the example2 directory is ex2_kobj.
Symbolic links are created in sysfs through the following interface:

int sysfs_create_link(struct kobject *kobj, struct kobject *target, char *name);

…where,
kobj: This is the pointer to the kobject that represents the directory in which the link is to be created.
target: This is the target directory to which the link is to be pointed.
name: This is the name with which the link is to be created.
For our example, the link is created as follows:

sysfs_create_link(ex1_kobj, ex2_kobj, “ ex2”);

To delete symbolic links, the following interface is available:

void sysfs_remove_link(struct kobject *kobj, char *name);

In our example, to delete the link, the following function is called:

sysfs_remove_link(ex2_kobj, “ ex2”);