FTF FREESCALE TECHNOLOGY FORUM 2014

Advanced Yocto Project™ and How to Use It with the QorIQ Linux SDK

FTF-SDS-F0138

Richard Schmitt

APR.2014

freescale™

External Use

Scope

- This session will first introduce you to Yocto Project so that you learn what it is, why you may want to use it, how it can help your project and how to use it with the QorIQ platform Linux® SDK.
- We will also cover the basic usage, prerequisites, installation, setting up running environments, building default images, external tool chain usage and GUI-hob.
- Lastly, learn about advanced usage, including modifying source code, adding patch and rebuild, configuring static IP address for Ethernet ports, adding prebuilt files into rootfs, integrating new packages and integrating kernel modules.

Agenda

- Welcome
- Intro to Yocto
- Yocto Basics for QorIQ
- QorIQ Build Environment
- BitBake
- GIT
- Configurations
- Yocto Metadata
- Yocto Layers
- Image Customization
- Creating Packages
- Tips and Tricks
- Resources
- Questions and Answers

Introduction

What is the Yocto Project?

www.yoctoproject.org

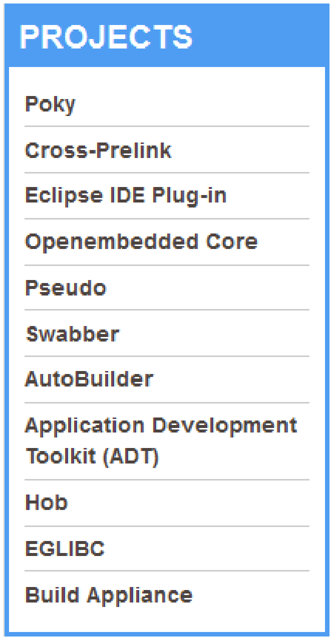
- “An open source collaboration project that provides templates, tools and methods to help you create custom Linux based systems for embedded products regardless of the hardware architecture.”
- Many individuals and companies, including Freescale, Intel, TI, Wind River, Mentor Graphics are contributing to the Yocto project

What is the Yocto Project? (continued)

www.yoctoproject.org

Consists of several separate projects :

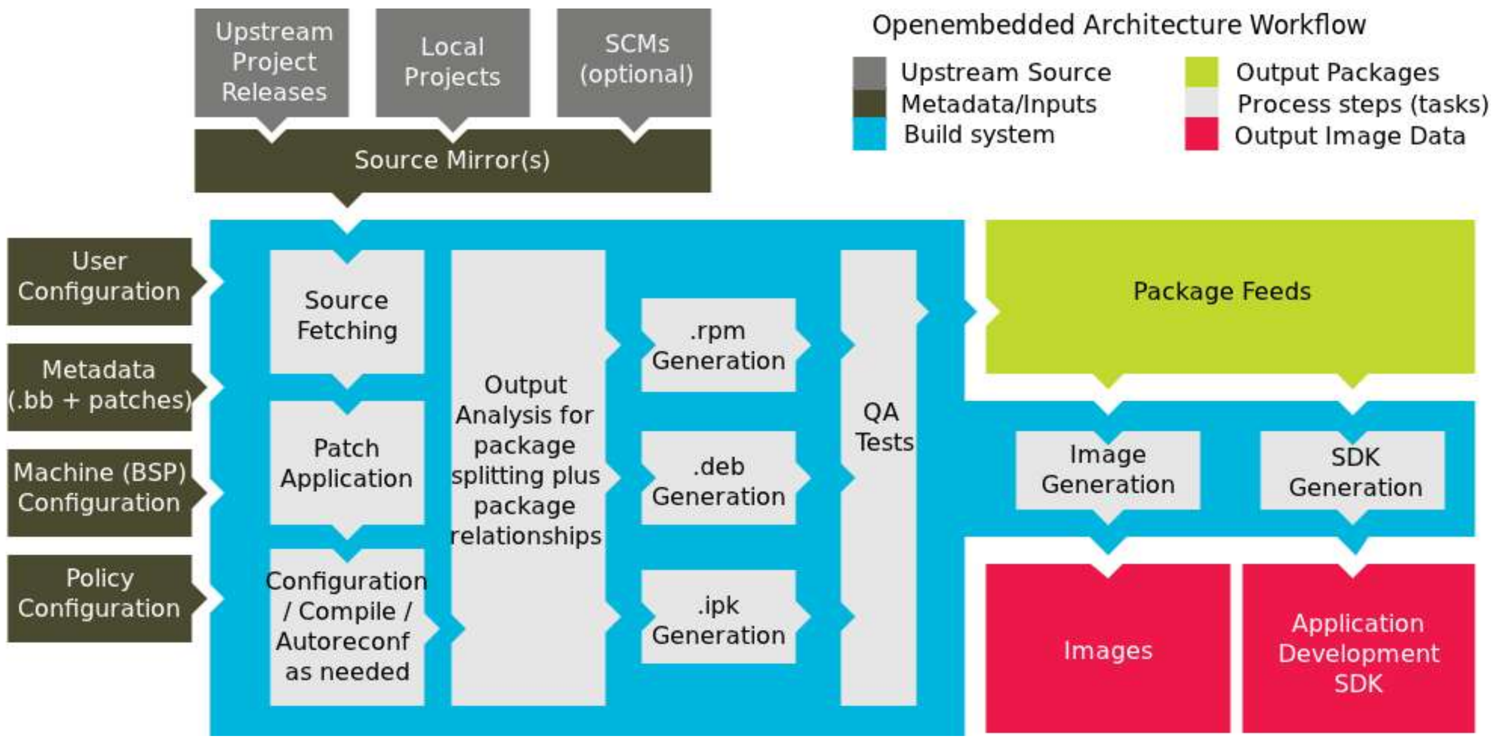
- Bitbake: parses metadata and runs tasks
- OpenEmbedded Core: core metadata and build information to build baseline embedded systems
- Poky: Yocto example distribution which integrates all the required pieces and makes an official release
- Hob: GUI tool to select packages to build and easily create custom image



PROJECTS

- Poky
- Cross-Prelink
- Eclipse IDE Plug-in
- Openembedded Core
- Pseudo
- Swabber
- AutoBuilder
- Application Development Toolkit (ADT)
- Hob
- EGLIBC
- Build Appliance

Yocto Project Development Environment



Yocto and BitBake Documentation

Yocto Project website:

`www.yoctoproject.org`

BitBake User Manual:

`docs.openembedded.org/bitbake/html`

The SDK documentation bundle:

```
[sdk_documentation/pdf/yocto]
├── adt-manual.pdf
├── bsp-guide.pdf
├── dev-manual.pdf
├── kernel-dev.pdf
├── kernel-manual.pdf
├── poky-ref-manual.pdf
└── profile-manual.pdf
```

Yocto Basics for QorIQ SDK

Freescale QorIQ SDK and the Yocto Project

- SDK 1.4 is based off the Yocto 1.4 `"dylan"` release
- SDK 1.5 is based off Yocto 1.5 “dora” release
- Alignment of release numbering just coincidence
- Freescale is an active part of the upstream community and is a full Yocto Project Member and a member of the Advisory Board
- Freescale has created QorIQ specific layers, that can be "plugged" into the Yocto build system, allowing users to build for Freescale target machines

SDK Web Location

- Internal**:**

http://linux.freescale.net/labDownload2/viewDownloads.php?Filter=QorIQ+SDK&field=PL&Action=Filter

- External:

http://compass.freescale.net/livelink/livelink?func=ll&objId=226777046&objAction=browse&viewType=1

Example:

```
wget http://linux.freescale.net/labDownload2/bspnew/QorIQ%20SDK%20v1.5//2013-12-19/QorIQ-SDK-V1.5-SOURCE-20131219-yocto.iso
wget http://linux.freescale.net/labDownload2/bspnew/QorIQ%20SDK%20v1.5//2013-12-19/QorIQ-SDK-V1.5-PPCE6500-CACHE-20131219-yocto.iso
```

QorIQ SDK Installation

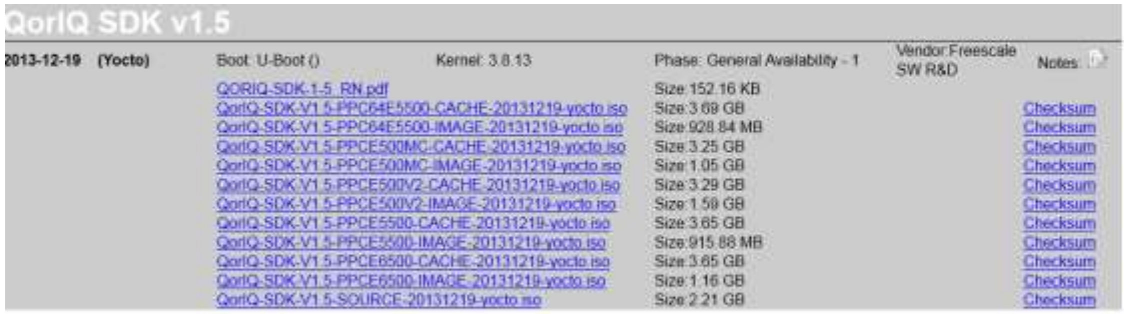
Tested Host Distro's / Release Files

source tar balls and recipes allowing full non-cache builds from source for any core

QorIQ-SDK-V<x.y>-SOURCE-<date>-yocto.iso

`Cache image to avoid having to rebuild all packages`

`QorIQ-SDK-V<x.y>-PPC`*`<core>`*`-CACHE-<date>-yocto.iso` for `E500V2, E500MC, E5500, E5500-64b, E6500 core`



QorIQ SDK v1.5

2013-12-19 (Yocto)	Boot: U-Boot ()	Kernel: 3.8.13	Phase: General Availability - 1	Vendor:Freescale SW R&D	Notes:
	QORIQ-SDK-1-5_RN.pdf		Size:152.16 KB		
	QorIQ-SDK-V1.5-PPC64E5500-CACHE-20131219-yocto.iso		Size:3.69 GB		Checksum
	QorIQ-SDK-V1.5-PPC64E5500-IMAGE-20131219-yocto.iso		Size:928.84 MB		Checksum
	QorIQ-SDK-V1.5-PPCE500MC-CACHE-20131219-yocto.iso		Size:3.25 GB		Checksum
	QorIQ-SDK-V1.5-PPCE500MC-IMAGE-20131219-yocto.iso		Size:1.05 GB		Checksum
	QorIQ-SDK-V1.5-PPCE500V2-CACHE-20131219-yocto.iso		Size:3.29 GB		Checksum
	QorIQ-SDK-V1.5-PPCE500V2-IMAGE-20131219-yocto.iso		Size:1.58 GB		Checksum
	QorIQ-SDK-V1.5-PPCE5500-CACHE-20131219-yocto.iso		Size:3.65 GB		Checksum
	QorIQ-SDK-V1.5-PPCE5500-IMAGE-20131219-yocto.iso		Size:915.88 MB		Checksum
	QorIQ-SDK-V1.5-PPCE6500-CACHE-20131219-yocto.iso		Size:3.65 GB		Checksum
	QorIQ-SDK-V1.5-PPCE6500-IMAGE-20131219-yocto.iso		Size:1.16 GB		Checksum
	QorIQ-SDK-V1.5-SOURCE-20131219-yocto.iso		Size:2.21 GB		Checksum

Supported Build Hosts	
CentOS	Fedora
OpenSUSE	Redhat
Ubuntu	Debian

QorIQ SDK Installation (continued)

- For each ISO image (source and cache) or physical DVD:

```
$ sudo mount -o loop \
QorIQ-SDK-V<x.y>-[SOURCE|<core>]-<date>-yocto.iso \
/mnt/cdrom
```

- As a non-root user [(*)], run the install script:

```
$ /mnt/cdrom/install
```

- In the installation path run the script to prepare the environment (Internet access may be required):

```
$ cd QorIQ-SDK-V<x.y>-<date>-yocto
$ ./scripts/host-prepare.sh
```

(*) Note however you may be required to enable sudo root permission

Freescale QorIQ Specific Layers

The SDK contains three QorIQ specific layers of software components (more on this in the *Layers* section)

`meta-fsl-ppc-toolchain`:
layer for FSL toolchain recipes

- Pushed through `git.freescale.com`

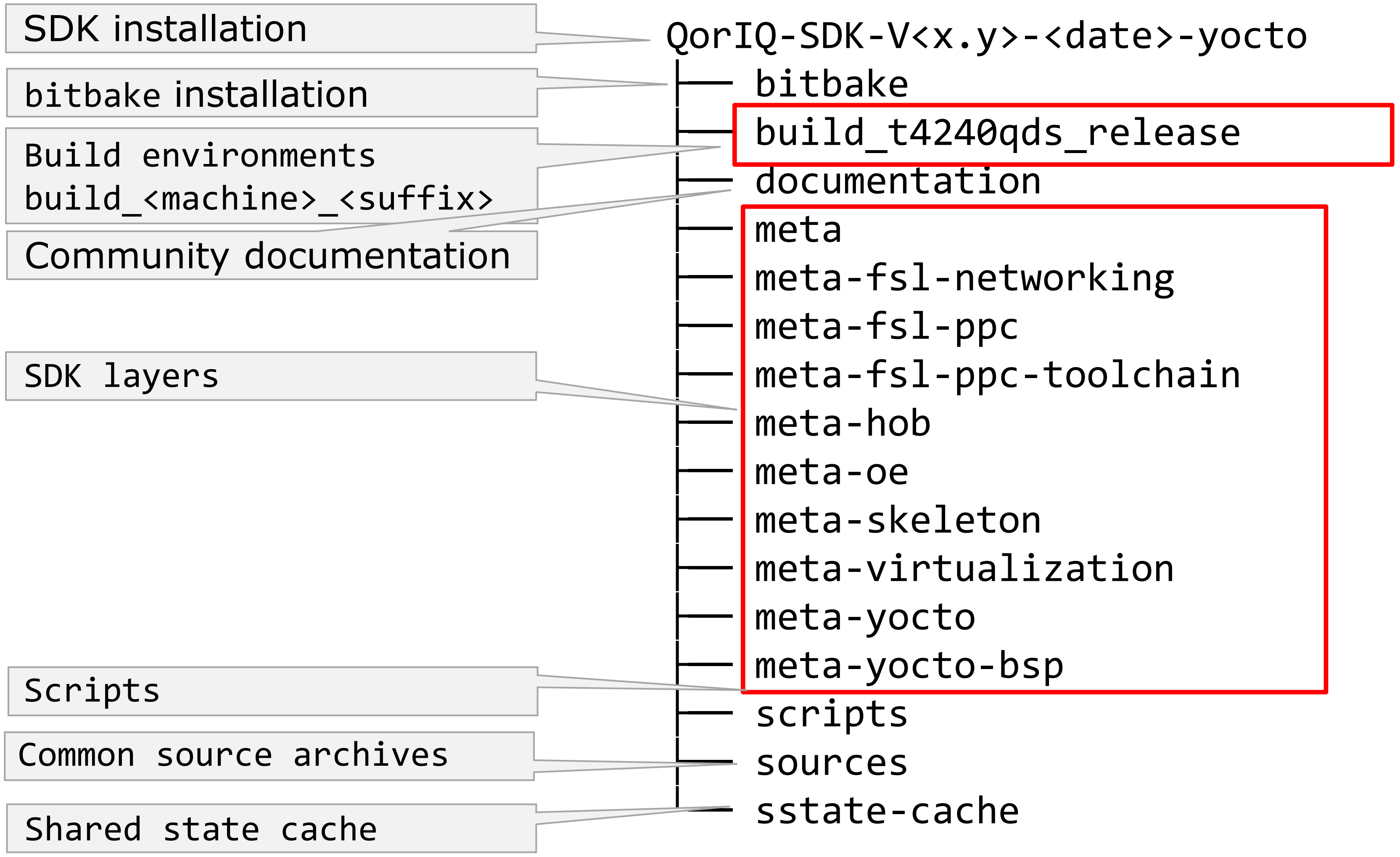
`meta-fsl-ppc`:
public QorIQ software components upstreamed to the community

- a subset is available at `git.yoctoproject.org` for Yocto releases 1.1, 1.2, 1.3, 1.4, …
- Pushed through `git.freescale.com`

`meta-fsl-networking`:
layer for networking-specific recipes

- Pushed through `git.freescale.com`

SDK Installation Structure



```
QorIQ-SDK-V<x.y>-<date>-yocto
├── bitbake
├── build_t4240qds_release
├── documentation
├── meta
├── meta-fsl-networking
├── meta-fsl-ppc
├── meta-fsl-ppc-toolchain
├── meta-hob
├── meta-oe
├── meta-skeleton
├── meta-virtualization
├── meta-yocto
├── meta-yocto-bsp
├── scripts
├── sources
└── sstate-cache
```

SDK Yocto Build Environment

Creating a Build Environment

Usage : `source ./fsl-setup-poky -h`

- The `<sdk_install_dir>/fsl-setup-poky` script sets up a build environment for a chosen target machine:

```
QorIQ-SDK-V<x.y>-yyyymmdd-yocto$ source ./fsl-setup-poky -h

Usage: source ./fsl-setup-poky <-m machine>
    Optional parameters: [-j jobs] [-t tasks] [-s path] [-p] [-l] [-h]
    Supported ppc machines: b4420qds-64b b4420qds b4860qds-64b b4860qds bsc9131rdb
bsc9132qds p1010rdb p1020rdb p1021rdb p1022ds p1023rdb p1025twr p2020ds p2020rdb
p2041rdb p3041ds p4080ds p5020ds-64b p5020ds p5040ds-64b p5040ds t4160qds-64b t4160qds
t4240qds-64b t4240qds

    [-j jobs]:  number of jobs for make to spawn during the compilation stage.
    [-t tasks]: number of BitBake tasks that can be issued in parallel.
    [-d path]:  non-default DL_DIR path (download dir)
    [-c path]:  non-default SSTATE_DIR path (shared state Cache dir)
    [-b path]:  non-default build dir location
    [-s path]:  append an extra path to build_machine_release folder
    [-l]:       lite mode. To help conserve disk space, deletes the building directory
once the package is built.
    [-p]:       append cache and source mirrors (For FSL Internal Use Only)
    [-h]:       help
```

Creating a Build Environment (continued)

Example : `t4240qds`

```
$ source ./fsl-setup-poky -m t4240qds -j 4 -t 4 -l
Configuring for t4240qds board type
Run the following commands to start a build:
    bitbake fsl-image-lsb-sdk
    bitbake fsl-image-minimal
    bitbake fsl-image-kvm
    bitbake fsl-image-full
    bitbake fsl-image-flash
    bitbake fsl-image-core

To return to this build environment later :
    source [path-to]/build_t4240qds_release/SOURCE_THIS
or
    . [path-to]/build_t4240qds_release/SOURCE_THIS
```

Image build commands

- To create multiple build environments for identical machines, extend the default path with `[-s path]`
  E.g. : `build_t4240qds_release_version1`

Selecting an Existing Build Environment

- After creating or returning to an existing build environment, the shell's current working directory is changed to `build_<machine>_release_<suffix>`, from which bitbake must be invoked

From here on, the training material will refer as follows to:

- a build environment folder ➔ `<project>`
- a package name ➔ `<pkg>`

Working with a Build Environment
Local Configuration File :<project>/conf/local.conf

```
# This file is your local configuration file and is where all local user
# settings are placed.

# Package Management configuration
PACKAGE_CLASSES ?= "package_rpm"

# Extra image configuration defaults
# The EXTRA_IMAGE_FEATURES variable allows extra packages to be added to
# the generated images.
EXTRA_IMAGE_FEATURES = "debug-tweaks"

# Additional image features
# The following is a list of additional classes to use when building images
# which enable extra features.
USER_CLASSES ?= "image-mklibs image-prelink"

# CONF_VERSION is increased each time build/conf/ changes incompatibly and
# is used to track the version of this file when it was generated.
CONF_VERSION = "1"
```

SDK uses rpm package management

Working with a Build Environment (continued)

Local Configuration File :`<project>/conf/local.conf (continued)`

```
# Machine Selection
MACHINE = “t4240qds"
```

Set by : -m <machine>

```
# Distro selection
DISTRO = "fsl-networking"

# Parallelism Options
BB_NUMBER_THREADS = “4"
PARALLEL_MAKE = "-j 4"
```

Set by : -t <threads> -j <jobs>

```
# Source download dir
DL_DIR = "/opt/yt_sdks/QorIQ-SDK-V<x.y>-<date>-yocto/\
build_t4240qds_release/../sources"
```

Default. Shared between targets

```
# The sstate-cache dir
SSTATE_DIR = "/opt/yt_sdks/QorIQ-SDK-V<x.y>-<date>-yocto/\
build_t4240qds_release/../sstate-cache"

# use xz instead of gzip for sstate-cache
SSTATE_PKG_SUFFIX ?= "txz"
SSTATE_PKG_TARZIPPROG ?= "xz"

# delete sources after build
INHERIT += "rm_work"
```

Set by : -l (lite mode)

Wo.king with a Build Environment (continued)

Local Configuration File :`<project>/conf/local.conf (continued)`

- Touch `conf/local.conf` to force a reload of the cache

```
$ touch <project>/conf/local.conf
$ bitbake <image_recipe>
```

This will force all configuration files and dependencies to be reparsed

Working with a Build Environment (continued)

Temporary Directory: `<project>/tmp`

- A build environment has a `./tmp` directory, or `${TMPDIR}`, which receives all the build output

```
<project>/tmp
├── buildstats
├── cache
├── deploy
│   ├── images            Generated image files (kernel, u-boot, rcw, rfs, ...)
│   ├── licenses
│   ├── rpm               Produced installable .rpm files of built packages
│   └── sdk
├── log
├── pkgdata
├── sstate-control
├── stamps
├── sysroots              Shared header files and libraries
│   ├── i686-linux
│   ├── t4240qds
│   └── t4240qds-tcbootstrap
└── work
    ├── all-fsl_networking-linux
    ├── i686-linux
    ├── t4240qds-fsl_networking-linux
    └── ppce6500-fsl_networking-linux
```

Working with a Build Environment (continued)

Temporary Directory: `<project>/tmp (continued)`

```
<project>/tmp
├── buildstats
├── cache
├── ccache
├── deploy
│   ├── images
│   ├── licenses
│   └── rpm
├── pkgdata
├── sstate-control
├── stamps
├── sysroots
│   ├── i686-linux
│   ├── t4240qds
│   └── t4240qds-tcbootstrap
├── work
│   ├── all-fsl_networking-linux
│   ├── t4240qds-fsl_networking-linux
│   ├── ppce6500-fsl_networking-linux
│   └── x86_64-linux
└── work-shared
    └── gcc-4.7.2+fsl-r19
```

Package working directories per architecture and board

– A working directory `${WORKDIR}`, is created for each package.
All tasks execute from a work directory.

– Working directories are grouped in sub-folders :

- `<machine>-fsl_networking-linux` : board specific target side packages like rcw, kernel, u-boot, ...
- `<core>-fsl_networking-linux` : non-board specific packages, compiled for the target architecture
- `<host>-*`

Working with a Build Environment (continued)

Image Generation

- Images are generated by invoking `bitbake` for an image recipe, e.g.

```
$ bitbake fsl-image-core
```

- An image recipe can specify multiple image file types to be generated simultaneously, e.g.

```
[meta-fsl-networking/images/fsl-image-flash.bb]
IMAGE_FSTYPES ?= "tar.gz ext2.gz.u-boot jffs2"
```

- A `*.rootfs.tar.gz` image file contains an archive of the file system, suitable for deployment:
  - To external media, like hard drive
  - As an NFS-mounted rootfs

Working with a Build Environment (continued)

Image Generation: Image Recipes

- `fsl-image-minimal` : Basic just packages to boot up a board; suitable as a starting point for a custom image
- `fsl-image-core` : **`fsl-image-minimal`** + FSL-specific packages
- `fsl-image-flash` : To recover **`fsl-image-full`** to SD/USB/HD media
- `fsl-image-full` : All packages + self-hosted tool chain; deploy to mass storage
- `fsl-image-kvm` : **`fsl-image-minimal`** + KVM + QEMU
- `fsl-image-lsb-sdk` : All packages of LSB standard

Working with a Build Environment (continued)

Which Packages in `fsl-image-*` ?

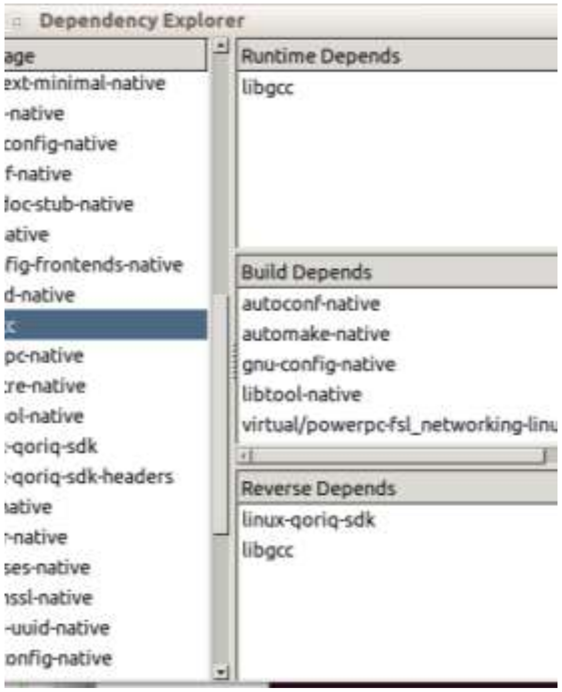
```
$ bitbake -g fsl-image-minimal
Loading cache: 100%
|#################################################################|
ETA:  00:00:00
Loaded 1161 entries from dependency cache.
Parsing recipes: 100%
|#################################################################|
Time: 00:00:00
Parsing of 859 .bb files complete (857 cached, 2 parsed). 1162
targets, 41 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
NOTE: Preparing runqueue
NOTE: PN dependencies saved to 'pn-depends.dot'
NOTE: Package dependencies saved to 'package-depends.dot'
NOTE: Task dependencies saved to 'task-depends.dot'

$ cat pn-depends.dot | grep -v \\-native | \
grep -v digraph | grep -v \\} | grep –v fsl-image | \
awk '{print $1}' | sort | uniq
```

Working with a Build Environment (continued)
Which Packages in `fsl-image-*` ?

- An easier way…
  - Check in tmp/deploy/licenses/fsl-image-core-<target>-<date>/package.manifest
  - Also, look at license.manifest for a list of packages and associated licenses.
  - Or look at individual subdirectories under tmp/deploy/licenses for actual packages and license text.

Dependency Graphs



ext-minimal-native
-native
config-native
f-native
doc-stub-native
ative
fig-frontends-native
d-native
pc-native
re-native
ol-native
-qoriq-sdk
-qoriq-sdk-headers
ative
-native
ses-native
ssl-native
-uuid-native
onfig-native

Runtime Depends: libgcc

Build Depends: autoconf-native, automake-native, gnu-config-native, libtool-native, virtual/powerpc-fsl_networking-linu

Reverse Depends: linux-qoriq-sdk, libgcc

- bitbake -g <target> or bitbake -g -u depexp <target>
  - pn-buildlist
  - pn-depends.dot
  - task-depends.dot
  - package-depends.dot
- dot -Tpng -o pn-depends.png pn-depends.dot
- Graph files not very useful. Use dependency explorer

Working with a Build Environment (continued)

Packages in `fsl-image-minimal`

acl	attr	base-files	base-passwd
bash	binutils	binutils-cross	busybox
bzip2	db	eglibc	eglibc-initial
elfutils	expat	gcc-cross	gcc-cross-initial
gcc-cross-intermediate		gcc-runtime	gdbm
gettext	glib-2.0	initscripts	kbd
keymaps	libffi	libgcc	libtool
libtool-cross	libusb1	libusb-compat	linux-qoriq-sdk
linux-qoriq-sdk-headers		module-init-tools	module-init-tools-
crossmodutils-initscripts		ncurses	netbase
openssl	opkg	opkg-config-base	pciutils
perl	pkgconfig	popt	python
readline	sqlite3	sysvinit	sysvinit-inittab
task-core-boot	tinylogin	u-boot	udev
udev-extraconf	update-modules	update-rc.d	usbutils
zip	zlib		

Working with a Build Environment (continued)

Packages in `fsl-image-flash` and `fsl-image-core`

- `fsl-image-flash = fsl-image-minimal + ...`

dosfstools	dropbear	e2fsprogs	fm-ucode
hv-cfg	hypervisor	lzo	mtd-utils
net-tools	rcw	sysfsutils	sysklogd
sysstat	task-core-ssh-dropbear		util-linux

- `fsl-image-core = fsl-image-flash + ...`

bridge-utils	coreutils	debianutils	eth-config
ethtool	file	flex	flib
fmc	fmlib	fsl-tlu	gawk
gmp	hdparm	i2c-tools	inetutils
iozone3	iperf	iproute2	ipsec-tools
iptables	iputils	libcap	libedit
libhugetlbfs	libpcap	libppc	libxml2
lmbench	mdadm	merge-files	mux-server
netperf	pme-priv	pme-tools	procps
psmisc	qoriq-debug	stat	tcpdump
usdpaa			

Working with a Build Environment (continued)

Image Generation: `<project>/tmp/deploy/images`

```
build_t4240qds_release/tmp/deploy/images/
├── fsl_fman_ucode_t4240qds_106_2_4.bin
├── fsl-image-minimal-t4240qds-20121116160802.rootfs.ext2.gz.u-boot
├── fsl-image-minimal-t4240qds.ext2.gz.u-boot -> fsl-image-minimal-t4240qds-20121116160802.rootfs.ext2.gz.u-boot
├── [...]
├── hv
│   ├── hv
│   ├── hv.bin
│   ├── hv.map
│   ├── hv.uImage
│   └── hypervisor.config
├── hv-cfg
│   ├── README
│   └── R_PPSXX_0xe
│       └── hv-2p-lnx-lnx.dtb
├── modules-3.0.48-rt70-r5-t4240qds.tgz
├── rcw
│   ├── README
│   ├── R_PPSXN_0x10
│   │   ├── rcw_13g_1500mhz_rev2.bin
│   │   ├── [...]
│   │   └── rcw_5g_1500mhz_rev3.bin
│   ├── R_PPSXX_0xe
│   └── R_RRRSS_0x16
├── u-boot-T4240QDS.bin -> u-boot-T4240QDS-git-r26.bin
├── u-boot-T4240QDS-git-r26.bin
├── [...]
├── uImage-3.0.48-r5-t4240qds-20121116160802.bin
├── uImage-3.0.48-r5-t4240qds-20121116160802.dtb
├── uImage-3.0.48-r5-t4240qds-usdpaa-20121116160802.dtb
├── uImage-t4240qds.bin -> uImage-3.0.48-r5-t4240qds-20121116160802.bin
├── uImage-t4240qds.dtb -> uImage-3.0.48-r5-t4240qds-20121116160802.dtb
└── uImage-t4240qds-usdpaa.dtb -> uImage-3.0.48-r5-t4240qds-usdpaa-20121116160802.dtb
```

FMan microcode

RFS image files and symlinks

Embedded hypervisor image

HV configuration device tree

Reset Configuration Word images

u-boot images and symlinks

Kernel image, dtb's and symlinks

Working with a Build Environment (continued)

Image Generation: File System Content

```
<project>/tmp/work/t4240qds-fsl_networking-linux/fsl-image-minimal-1.0-
  r0/rootfs
├── bin
├── boot
├── dev
├── etc
├── home
├── initial
├── install
├── media
├── mnt
├── proc
├── sbin
├── sys
├── tmp
├── usr
└── var
```

The file content of the generated rootfs can be inspected in the `${WORKDIR}/rootfs` of the image

Using BitBake

Usage: BitBake [options] [package ...]
Options

```
--version             show program's version number and exit
-h, --help            show this help message and exit
-b BUILDFILE, --buildfile=BUILDFILE
-k, --continue        continue as much as possible after an error.
-a, --tryaltconfigs
-f, --force           force run of specified cmd, regardless of stamp status
-c CMD, --cmd=CMD     specify task to execute.
-r PREFILE, --read=PREFILE
-R POSTFILE, --postread=POSTFILE
-v, --verbose         output more chit-chat to the terminal
-D, --debug           increase the debug level.
-n, --dry-run         don't execute, just go through the motions
-S, --dump-signatures
-p, --parse-only
-s, --show-versions show current and preferred versions of all packages
-e, --environment     show the global or per-package environment
-g, --graphviz        emit the dependency trees of the specified packages in
                      the dot syntax
-I EXTRA_ASSUME_PROVIDED, --ignore-deps=EXTRA_ASSUME_PROVIDED
-l DEBUG_DOMAINS, --log-domains=DEBUG_DOMAINS
-P, --profile
-u UI, --ui=UI        user interface to use
-t SERVERTYPE, --servertype=SERVERTYPE  --revisions-changed
```

Running BitBake

- BitBake must always be executed from within the `<project>` directory
- When `bitbake` (a Python script) runs:
  - It parses recipes and tasks
  - Determines task queue dependencies
  - Prepares and executes a run queue of tasks, which perform the steps needed to obtain the desired result, e.g. image generation
- Any required earlier tasks will be run first (e.g. source will be installed before compilation)
- To speed up subsequent builds, the generated `<pkg>.rpm`'s are saved to the binary cache folders in:

```
<project>/tmp/deploy/rpm
```

Running Specific BitBake Tasks

- Invoke `bitbake` to run a specific task specified in the recipe of a package or an image, e.g.
  - Generate one of the image types defined in the SDK
  - Build an individual package
  - Build the cross compiler toolchain
  - Optionally with `-c <CMD>` indicate a specific task to perform

```
$ bitbake [-c <CMD>] [options] <recipe>
```

- An initial image built may take a significant amount of time, if many packages are not available in the binary cache

Useful BitBake Tasks to Run Manually

- For most any recipe:

`build` *(default)*	`clean`	`cleansstate`	`compile`	`configure`
`install`	`listtasks`	`patch`	`rm_work`	

- For `fsl-image-*` recipes: `buildall` `rootfs`
- For kernel recipes:

  `buildall` `compile_kernelmodules` `menuconfig`
  `savedefconfig` `sizecheck`
- For non-image recipes: `deploy`
- Most common sequence:

  `fetch` → `unpack` → `patch` → `configure` → `compile`
  → `install` → `package` → `package_write`

Useful BitBake Tasks to Run Manually

- `clean` : remove the work folder of the package
- `cleansstate` : `clean` + delete the cached binary
  - when a known good package fails to build unexpectedly, or an image build fails with `"error: Failed dependencies"`, do `-c cleansstate` first on the failing package, then rebuild
- `patch` : install source including all patches
- `menuconfig` : run kernel `menuconfig`

Not for regular use, so extreme caution is advised:

- `cleanall` : delete the source archive from `../sources`

rce Modifications in the Working Directory: Implications

- Assume:
  - A package’s source files have been installed in its working directory
  - The user has directly modified one or more source files
- In general, if a BitBake build depends on this package:
  - BitBake will not be able to determine if any local changes were made its source files, so a rebuild will not automatically be triggered
  - Always force rebuild the package when any source files have been changed : `bitbake -c compile -f <pkg>`

Useful BitBake Options

Generate debug output:	`-D, -DD, -DDD`
Force rerun of specified task:	`-f, --force`
Dump the environment:	`-e, --environment`
Dump package dependency list:	`-g, --graphviz`
Continue even in case of error:	`-k, --continue`

BitBake Execution Logs

- For each executed BitBake task, log files are written to the package's `temp` folder, e.g. for `u-boot`’s `deploy` task:

```
$ cd <project>/tmp/work/t4240qds-fsl_networking-linux/u-boot/git-
r33/temp
$ ls -la | grep deploy
lrwxrwxrwx 1 peter peter      19 Nov 16 17:34 log.do_deploy -> log.do_deploy.31278
-rw-rw-r-- 1 peter peter    1404 Nov 16 17:34 log.do_deploy.31278
-rwxrwxr-x 1 peter peter    7069 Nov 16 17:34 run.do_deploy.31278
```

- Whenever a BitBake task for a package fails, the path to the log file capturing the failure is displayed
- To log output to console (e.g. from make) during the build, add the `-v, --verbose` option

BitBake Build Output

```
$  bitbake fsl-image-minimal
Parsing recipes: 100% |###############################################################| Time: 00:00:17
Parsing of 1232 .bb files complete (0 cached, 1232 parsed). 1579 targets, 48 skipped, 0 masked, 0 errors.
OE Build Configuration:
BB_VERSION        = "1.18.0"
BUILD_SYS         = "x86_64-linux"
NATIVELSBSTRING   = "Ubuntu-12.04"
TARGET_SYS        = "powerpc-fsl_networking-linux"
MACHINE           = "b4860qds"
DISTRO            = "fsl-networking"
DISTRO_VERSION    = "1.4"
TUNE_FEATURES     = "m32 fpu-hard e6500 altivec"
TARGET_FPU        = "hard"
meta
meta-yocto
meta-yocto-bsp    = "sdk-v1.4.x:5a7532143a49f59a5c85b08d3daf574fb1eccd8d"
meta-fsl-ppc      = "sdk-v1.4.x:f9fd0a617eb6913f87335c551918315ff4ebe18c"
meta-fsl-ppc-toolchain = "sdk-v1.4.x:8ec94cec04527cb971c125b1ddd2c5375034d723"
meta-virtualization = "sdk-v1.4.x:ad6df4f59cd7646f61db29e8fa51f878329d6f93"
meta-fsl-networking = "(nobranch):00f7a535029ca7ef8c96ba8e9916d4742166bab0"
meta-oe
meta-networking   = "sdk-v1.4.x:7c8dd8f096b64a709175d37a08a4fb02ca263616" "

NOTE: Resolving any missing task queue dependencies
NOTE: Preparing runqueue
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: Running task 1598 of 1602 (ID: 8, /opt/yt_sdks/QorIQ-SDK-V1.4-20130625-yocto/meta-fsl-networking/images/fsl-image-minimal.bb,
do_rootfs)
NOTE: package fsl-image-minimal-1.0-r0: task do_rootfs: Started
NOTE: package fsl-image-minimal-1.0-r0: task do_rootfs: Succeeded
NOTE: Running noexec task 1600 of 1602 (ID: 5, /opt/yt_sdks/QorIQ-SDK-V1.4-20130625-yocto/meta-fsl-networking/images/fsl-image-
minimal.bb, do_build)
[...]
NOTE: Tasks Summary: Attempted 1602 tasks of which 1598 didn't need to be rerun and all succeeded.
```

BitBake and GIT

BitBake and git

- Freescale-specific packages, such as kernel and u-boot, are no longer supplied as pristine tar balls plus patches
- Instead, git tar balls are provided, which include the entire patch commit history
- Yocto uses `git` commands when needed, but hides these from the user
- To get started with `git`:
  - `git` project: `git-scm.com`
  - `git` cheat sheet: `git.jk.gs`
  - `gitk` repo browser: `kernel.org/pub/software/scm/git/docs/gitk.html`

Patches

Where are the Patch Files?

- For a package provide as a git repo, patches are identified in the git commit history
  - For non-git packages patches may be contained in one of the package recipe folders
- To extract the patch files from an installed git tree:
  - Install the package source from the git tar ball
  - Enter the installed package's git folder
  - List the commits with `git log`
  - Identify the <*since*>..<*until*> commit range of interest
  - Generate the patch files with `git format-patch`

Patches
Which Are Applied for a Package?

- To query for all applied patches, e.g. for `busybox`:

```
$ bitbake -e busybox | grep ^SRC_URI | tr -s ' ' '\n' | \
tr -s '\t' '\n' | grep patch
file://udhcpscript.patch
file://udhcpc-fix-nfsroot.patch
file://B921600.patch
file://get_header_tar.patch
file://busybox-appletlib-dependency.patch
file://run-parts.in.usr-bin.patch
file://watch.in.usr-bin.patch
file://busybox-udhcpc-no_deconfig.patch
file://busybox-1.19.4-ubi-user-h.patch
```

- Note: for packages installed from a git repo use `git log`

Installing Package Sources
Example : u-boot

- To install the sources of a package:

```
$ bitbake -c patch <pkg>
```

  Any earlier tasks that must be completed before `do_patch` will be implicitly executed first
- Eventually, sources will be installed into:

```
${WORKDIR}/<src folder>
```

Installing Package Sources (continued)

Example : u-boot (continued)

```
$ cd build_t4240qds_release
$ bitbake -c patch u-boot
[...]
$ cd tmp/work/t4240qds-fsl_networking-linux/u-boot-git-r26
$ ls
deploy-rpms     license-destdir  pkgdata  sstate-install-deploy      temp
deploy-u-boot   package          pseudo   sstate-install-deploy-rpm
git             packages-split   shlibs   sstate-install-package


$ ls git
api          config.mk  driver                                es.mk
arch         COPYING    dts       MAINTAINERS     net          snapshot.commit
board        CREDITS    examples  MAKEALL         onenand_ipl  spl
boards.cfg   disk       fs        Makefile        post         tools
common       doc        include   mkconfig        README
```

```
u-boot supplied as a git repo tar ball
➔ source folder is called git
```

Build Environment Configuration

Machine Configuration File
`meta-fsl-ppc/conf/<machine>.conf`

- Machine specific configuration:
  - Shared hardware tuning definitions: `<core>.inc` files
  - Machine specific BSP information: `<machine>.conf` files
- Example: P4080DS

```
meta-fsl-ppc/conf
└── machine
    ├── e500mc.inc
    ├── e500v2.inc
    ├── e6500-64b.inc
    ├── e6500.inc
  [...]
    ├── p4080ds.conf
    ├── p5020ds-64b.conf
    ├── p5020ds.conf
    ├── t4240qds-64b.conf
    └── t4240qds.conf
```

```
[meta-fsl-ppc/conf/machine/p4080ds.conf]
require e500mc.inc

UBOOT_MACHINES = "P4080DS P4080DS_SECURE_BOOT P4080DS_SDCARD P4080DS_SPIFLASH "
KERNEL_DEVICETREE = "${S}/arch/powerpc/boot/dts/p4080ds.dts"
KERNEL_DEFCONFIG = "${S}/arch/powerpc/configs/corenet32_smp_defconfig"

JFFS2_ERASEBLOCK = "0x10000"
```

Machine Configuration File (continued)

meta-fsl-ppc/conf/<machine>.conf (continued)

- UBOOT_MACHINES: enumeration of u-boot configs to build
  - check u-boot `<source tree>/boards.cfg` for available configs:

```
$ grep P4080DS board.cfg
P4080DS                  powerpc mpc85xx corenet_ds freescale
P4080DS_SDCARD           powerpc mpc85xx corenet_ds freescale [...]
P4080DS_SECURE_BOOT      powerpc mpc85xx corenet_ds freescale [...]
P4080DS_SPIFLASH         powerpc mpc85xx corenet_ds freescale [...]
P4080DS_SRIOBOOT_MASTER  powerpc mpc85xx corenet_ds freescale [...]
P4080DS_SRIOBOOT_SLAVE   powerpc mpc85xx corenet_ds freescale [...]
```

  - Add any configs needed to UBOOT_MACHINES
  - `git` source folder will contain a separate source tree instance per config, all of which will be built

Machine Configuration File (continued)

`meta-fsl-ppc/conf/<machine>.conf (continued)`

- `JFFS2_ERASEBLOCK`: the flash JFFS2 erase block size
- `KERNEL_DEFCONFIG`: the default kernel `defconfig`
  - Common `defconfig` each for `corenet32` and `corenet64` machines
- `KERNEL_DEVICETREE`: the default device tree
- For a custom board, create a `<new_machine>.conf` file

Linux Kernel Configuration

Kernel Configuration Hierarchy

```
meta-fsl-ppc
├── conf
│  [...]
│    └── machine
│      [...]
│        ├── p4080ds.conf
│      [...]
└── recipes-kernel
    └── linux
        ├── files
        ├── linux-qoriq-sdk.bb
        ├── linux-qoriq-sdk-headers.bb
        └── linux-qoriq-sdk.inc
```

Default kernel config and device tree

```
KERNEL_DEVICETREE = "${S}/arch/powerpc/boot/dts/p4080ds.dts \
        ${S}/arch/powerpc/boot/dts/p4080ds-usdpaa.dts"
KERNEL_DEFCONFIG = "${S}/arch/powerpc/configs/corenet32_smp_defconfig"
```

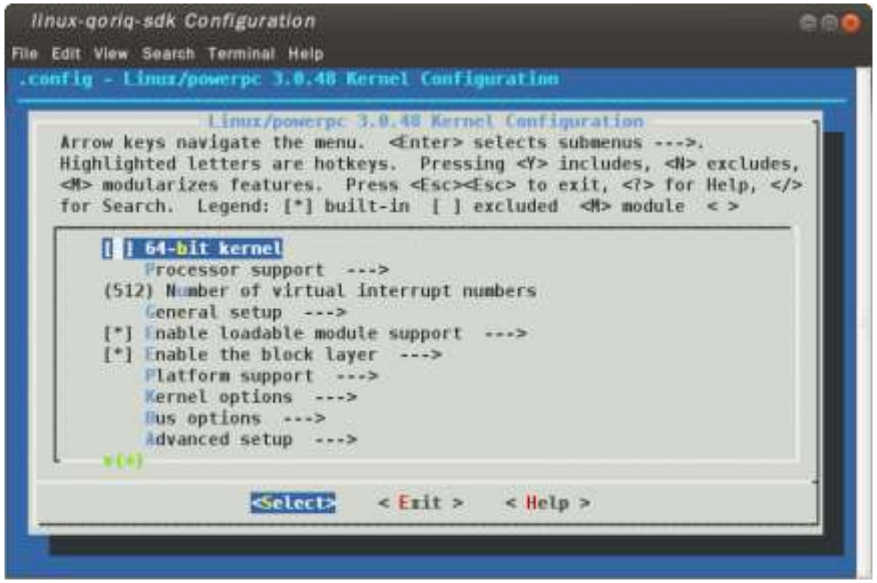
```
do_configure_prepend() {
# copy desired defconfig so we pick it up for the real
kernel_do_configure
cp ${KERNEL_DEFCONFIG} ${B}/.config
```

Linux Kernel Configuration (continued)

- To configure the Linux kernel :

```
$ bitbake -c menuconfig virtual/kernel        or
$ bitbake -c menuconfig linux-qoriq-sdk
```

- The kernel `menuconfig` configuration screen will be shown in a new console window:


linux-qoriq-sdk Configuration
File Edit View Search Terminal Help

```
.config - Linux/powerpc 3.0.48 Kernel Configuration

                Linux/powerpc 3.0.48 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

      [ ] 64-bit kernel
          Processor support  --->
      (512) Number of virtual interrupt numbers
          General setup  --->
      [*] Enable loadable module support  --->
      [*] Enable the block layer  --->
          Platform support  --->
          Kernel options  --->
          Bus options  --->
          Advanced setup  --->
      v(+)

              <Select>    < Exit >    < Help >
```

Linux Kernel Configuration (continued)

- After a configuration change it is recommended to:
  - Force a kernel rebuild:

```
$ bitbake -f -c compile virtual/kernel
$ bitbake virtual/kernel
```

  - The optionally regenerate the rootfs image:

```
$ bitbake -f -c clean custom-image-core
$ bitbake -f custom-image-core
```

Interactive Shell Configuration
General

- BitBake commands are issued from a Linux shell’s command line
- In a graphical X11 desktop environment (Gnome, KDE) this shell is executing in a terminal or console window
- Alternatively, BitBake could be invoked from a shell in a non-graphical environment, e.g. a remote `telnet` or `ssh` session, or a local Linux text screen
- The BitBake `OE_TERMINAL` variable must be appropriately configured to allow correct operation of interactive BitBake shells in any shell environment

....:active Shell Configuration (continued)
X11 Host Environments

- The BitBake variable `OE_TERMINAL` defines the terminal window configuration as one of the following values:

```
auto (default), gnome, xfce, rxvt, screen, konsole
(KDE 3.x only), none
```

  - `auto` is also suitable for a Gnome host environment
  - To configure for a non-Gnome host environment, modify `<project>/conf/local.conf :`
    - Choose a suitable value for `OE_TERMINAL` from one of the other defined display manager types : `xfce, rxvt, screen, konsole`

...active Shell Configuration (continued)

Non-X11 Host Environments

- When working in a non-X11 shell environment, change `<project>/conf/local.conf` as follows:

```
OE_TERMINAL = "screen"
```

- Run the `bitbake` command, e.g.

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

SDK Compiler Tool Chain

- An architecture-specific cross-compiler toolchain and `eglibc` are built and installed per build project and invoked by BitBake tasks as needed
  - An external compiler tool chain can be configured for use by Yocto as described in: www.openembedded.org/wiki/Adding_a_secondary_toolchain
- A cross compiler tool chain for use outside of BitBake can be generated as follows:

```
$ bitbake fsl-toolchain
```

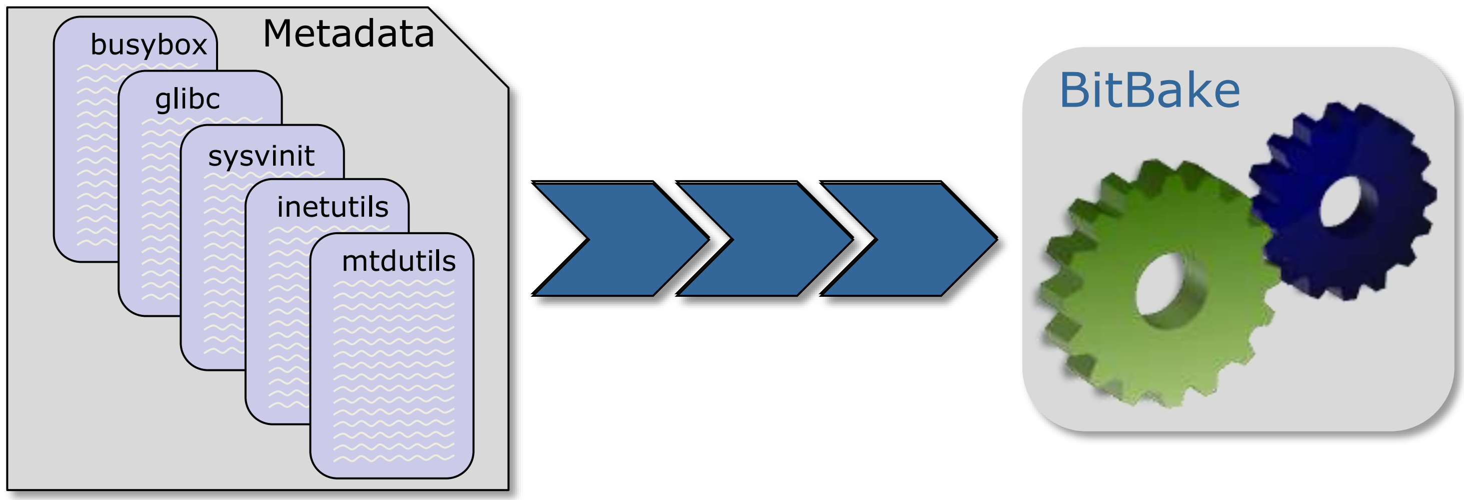
An installable tar file will be installed here:

```
<project>/tmp/deploy/sdk
```

Yocto Metadata Syntax and Semantics

Metadata
[From: http://docs.openembedded.org/bitbake/html]

- `.inc, .bb` and `.bbappend` files : recipes
  - structured collections of instructions which tell BitBake what to build ...



- `.conf` files : configuration files
- `.bbclass` files : classes

Metadata (continued)
Recipes and Tasks

- A `bitbake` recipe is a `.bb` file, that defines all the tasks that apply to building a package or image
- The defined tasks for a recipe can be listed (unsorted), e.g. for `u-boot`:

```
$ bitbake -c listtasks u-boot
[...]
NOTE: package u-boot-git-r26: task do_listtasks: Started
do_fetchall
do_build
do_devshell
do_cleansstate
do_configure
[...]
do_clean
do_package_write_rpm_setscene
do_rm_work
do_package
do_unpack
do_install
do_populate_sysroot_setscene
do_rm_work_all
do_checkuriall
NOTE: package u-boot-git-r26: task do_listtasks: Succeeded
```

Metadata Syntax and Semantics

Variables and Operators

Operator	Operation	Example	Resulting Value
=	Set to a value	VAR1 = "value"	"value"
${VAR}	Expand	VAR2 = "X${VAR1}Y"	"XvalueY"

?=
??=
	Set to a default value	VAR1 ?= "defval"
		VAR1 ??= "defval"
			if VAR1 unassigned : "defval"
			else VAR1 unchanged

:=	Immediate expansion	VAR1 = "value"
		VAR1 := "${VAR1}append
			"value"
			"valueappend"

+=	Append	VAR1 = "value"
		VAR1 += "Y"
			"value"
			"value Y"

=+	Prepend	VAR1 = "value"
		VAR1 =+ "X"
			"value"
			"X value"

.=	Append (no space)	VAR1 = "value" VAR1 .= "Y"	"value" "valueY"

=.	Prepend	VAR1 = "value"
		VAR1 =. "X"
			"value"
			"Xvalue"

N/A	Append/prepend conditional on OVERRIDES	VAR1 = "X Y"
		OVERRIDES = "A:B"
		VAR1_append_A = " C"
			VAR1 set to "X Y C"

Metadata Syntax and Semantics (continued)

append/prepend Conditional on OVERRIDES

- The OVERRIDES variable contains a list of strings ...

```
[meta/conf/bitbake.conf]
OVERRIDES = "${TARGET_OS}:${TARGET_ARCH}:build${BUILD_OS}:\
pn-${PN}:${MACHINEOVERRIDES}:${DISTROOVERRIDES}:\
forcevariable"

[meta/conf/distro/include/tclibc-eglibc.inc]
OVERRIDES .= ":libc-glibc"

$ bitbake -e fsl-image-core  | grep ^OVERRIDES
OVERRIDES="linux:powerpc:build-linux:pn-fsl-image-core:\
p4080ds:e500mc:fsl:forcevariable:libc-glibc"
```

... to match against for conditional append or prepend

```
${VAR}_append_<override string> = "<string to append>"
${VAR}_prepend_<override string> = "<string to prepend>"
```

Metadata Syntax and Semantics (continued)

append/prepend Conditional on OVERRIDES

- If <override string> in OVERRIDES
  after all += and =+ operators have been applied
  apply _append_ or _prepend_

```
[meta-fsl-ppc/recipes-kernel/u-boot/u-boot_git.bb]
TOOLCHAIN_OPTIONS_append_e5500-64b = "/../lib32-${MACHINE}"

[meta-fsl-networking/images/fsl-image-deploy.inc]
IMAGE_INSTALL_append_e500mc = " \
        fm-ucode \
        hv-cfg \
        rcw \
        hypervisor \
        hypervisor-partman \
"
```

Metadata Syntax and Semantics (continued)

`include|require <file>` Directives

- The contents of the specified file will be inserted at that location and parsed by `bitbake`
- The file name convention for an include file: `.inc`
- The path specified is relative and the first one found within BBPATH
- `require` raises a `ParseError` if the file is not found, `include` does not

Metadata Syntax and Semantics (continued)

`include|require <file>` Directives

- Example: P1010RDB machine requires configuration for an e500v2 architecture

```
[meta-fsl-ppc/conf/machine/p1010rdb.conf]
#@TYPE: Machine
#@Name: Freescale P1010RDB
#@DESCRIPTION: Machine configuration for the Freescale P1010RDB

require e500v2.inc

UBOOT_MACHINES = "P1010RDB_NAND P1010RDB_NOR"
KERNEL_DEVICETREE = "${S}/arch/powerpc/boot/dts/p1010rdb.dts"
KERNEL_DEFCONFIG = "${S}/arch/powerpc/configs/mpc85xx_defconfig"

JFFS2_ERASEBLOCK = "0x20000"
```

Metadata Syntax and Semantics (continued)

More Directives

- DEPENDS : build time dependencies between `.bb` files
- RDEPENDS : runtime dependencies
- PROVIDES : specifies the functionality a `.bb` file provides
- PREFERRED_VERSION_ :
  - if multiple `.bb` files exist for a package, `bitbake` defaults to the most recent version
  - to specify in a .conf file a specific package version to use

E.g.: `PREFERRED_VERSION_<pkg> = "x.y"`

`PREFERRED_VERSION_mypackage = "1.3"` will select the recipe `mypackage-1.3.bb` even over any more recent version

Appending Changes to an Existing Recipe

Using a .bbappend File

- Append specific changes to an existing `<pkg>.bb` recipe by creating a `<pkg>.bbappend` file (note the identical basename)

  E.g. used here to add Freescale private configuration info to an upstream (public) package recipe

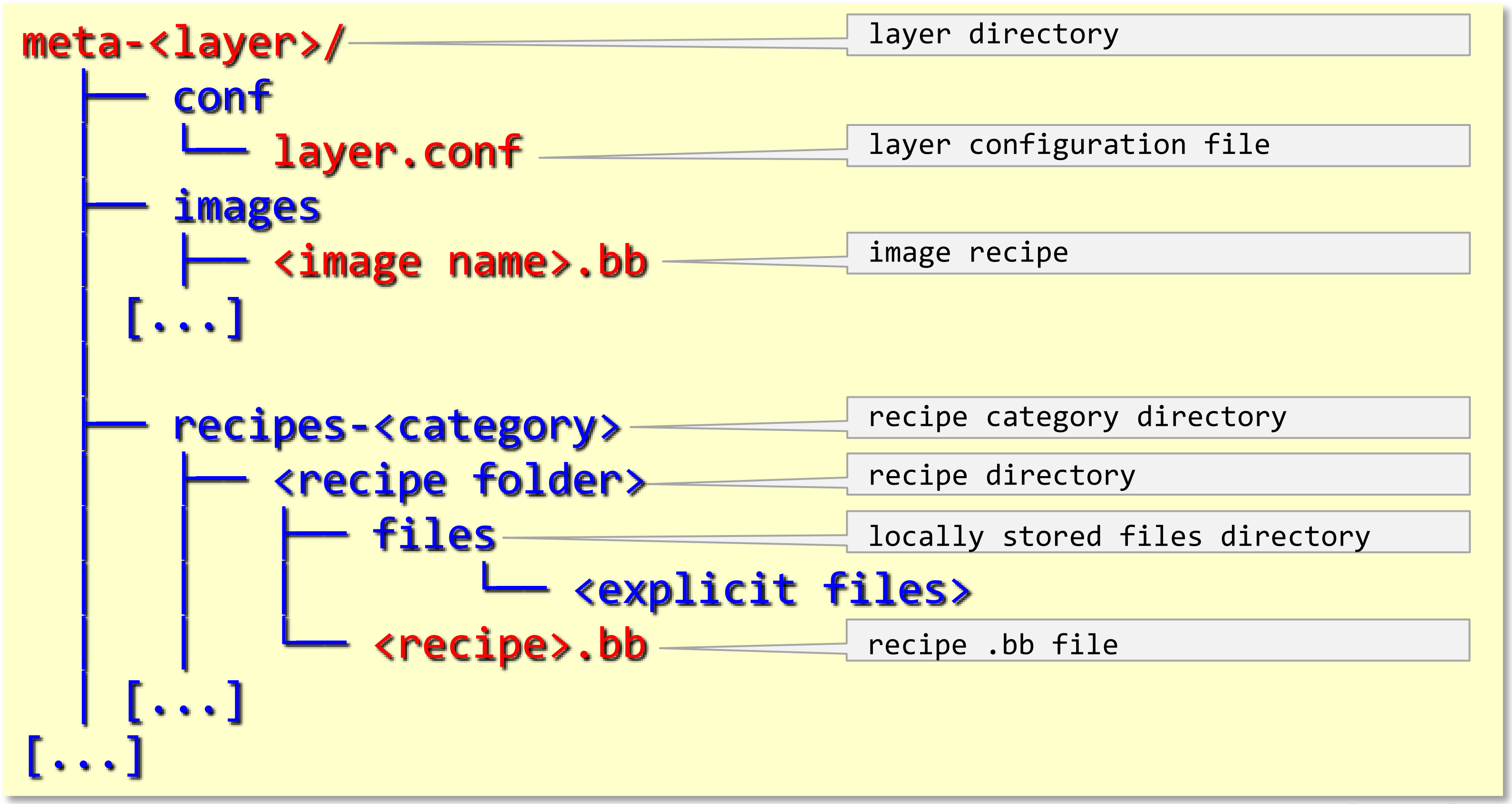
```
meta-fsl-networking/recipes-kernel/
└── dtc
    └── dtc_git.bbappend          changes to append to dtc_git.bb

meta/recipes-kernel/dtc
├── dtc
│   └── make_install.patch
├── dtc_git.bb                    package recipe
└── dtc.inc
```

Yocto Layers

Layers
Intro

- Yocto metadata is organized into multiple layers, so as to allow different customizations to be isolated from each other



```
meta-<layer>/                                  layer directory
  ├── conf
  │    └── layer.conf                          layer configuration file
  ├── images
  │    ├── <image name>.bb                     image recipe
  │   [...]
  │
  ├── recipes-<category>                       recipe category directory
  │    ├── <recipe folder>                     recipe directory
  │    │    ├── files                          locally stored files directory
  │    │    │      └── <explicit files>
  │    │    └── <recipe>.bb                    recipe .bb file
  │   [...]
[...]
```

SDK Layers

- The SDK configuration includes the example Yocto distribution based on the `poky` baseline, plus Freescale specific layers:
  - `meta-oe` (public) ... generic community packages
  - `meta-skeleton` (public)
  - `meta-yocto` (public)
  - `meta-fsl-ppc` (public) ... pushed upstream
  - `meta-fsl-ppc-toolchain` (public)... not upstreamed
  - `meta-fsl-networking` (public) ... not upstreamed
- Public variants live here:
  - `git.freescale.com` ... aligned with SDK releases
  - `git.yoctoproject.org` ... aligned with Yocto releases

SDK Layers (continued)

`meta-fsl-ppc` vs. `meta-fsl-networking`

- `meta-fsl-ppc`:
  - Adds basic support for all supported Freescale boards
  - Recipes for all public Freescale projects : kernel, u-boot, ...
  - Only defines images `fsl-toolchain`, `fsl-image-minimal` (not even those in the future)
  - Hosted on `git.freescale.com`
- `meta-fsl-networking`:
  - Adds definitions for images like `fsl-image-flash` and `fsl-image-full`
  - Focuses on networking, QorIQ-specific technology
  - More and more recipes are moved into `meta-fsl-ppc`
  - Hosted on `git.freescale.com`

SDK Layers (continued)

Dependencies

- Layers can have dependencies on other layers
  - `meta-fsl-ppc` depends on `meta-oe`, `oe-core`
  - `meta-oe` is a secondary repository to `oe-core` that contains a dump of many recipes
- See also:

  http://layers.openembedded.org/layerindex/

SDK Layers (continued)

meta-fsl-ppc Layer (public - upstream)

```
meta-fsl-ppc
├── conf
│   ├── layer.conf
│   └── machine
│       ├── e500mc.inc
│       │   [...]
│       ├── e5500.inc
│       ├── mpc8536ds.conf
│       │   [...]
│  │       └── t4240qds.conf
├── README
└── recipes-*
```

this layer's configuration

hardware tuning and
machine configurations

recipe categories

SDK Layers (continued)
Package Recipes in `meta-fsl-ppc`

- Packages are referred to by recipe:

  `<pkg>[_<version>].bb`

- In BitBake commands just the `<pkg>` is also valid

```
recipes-graphics
   xorg-xserver,xorg-driver

recipes-connectivity
   openssl,samba

recipes-kernel
   oprofile, linux, u-boot

recipes-extended
   ethtool, lm_sensors

recipes-ucode
   fm-ucode, fmc, fmlib

recipes-test
   testfloat

recipes-tools
   embedded-hv, eth-config, boot-
   format, rcw, lio-utils, flib,
   strongswan, hv-cfg, lxc, usdpaa
```

SDK Layers (continued)

meta-fsl-networking Layer (private - not upstream)

```
meta-fsl-networking/
├── conf
│   └── layer.conf
├── images
│   ├── fsl-image-core.bb
│   ├── fsl-image-deploy.inc
│   ├── fsl-image-flash.bb
│   ├── fsl-image-full.bb
│   ├── fsl-image-kvm.bb
│   ├── fsl-image-private.inc
│   └── fsl-toolchain.bbappend
└── recipes-*
```

this layer's configuration

image recipes which depend on Freescale-private packages

recipe categories

SDK Layers (continued)

bitbake-layers Script

- `show-layers`: Shows the currently configured layers

```
$ bitbake-layers show_layers
layer                  path                                                 priority
=====================================================================================
meta                   /bsps/QorIQ-SDK-V1.4-20130625-yocto/meta                    5
meta-yocto             /bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-yocto              5
meta-yocto-bsp         /bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-yocto-bsp          5
meta-fsl-ppc           /bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-fsl-ppc            5
meta-fsl-ppc-toolchain /bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-fsl-ppc-toolchain  5
meta-virtualization    /bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-virtualization     7
meta-fsl-networking    /bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-fsl-networking     5
meta-oe                /bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-oe/meta-oe         1
meta-networking        /bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-oe/meta-networking 5
```

- `flatten`: Takes the current layer configuration and builds a "flattened" directory, containing the contents of all layers, with any overlayed recipes removed and `.bbappend` files appended to the corresponding recipes

```
$ bitbake-layers flatten <directory>
```

SDK Layers (continued)
bitbake-layers Script

- `show_appends` : lists `.bbappend` files and recipes they append to

```
$ bitbake-layers show_appends
Parsing recipes..done.
State of append files:
atk_1.32.0.bb:
  /opt/yt_sdks/QorIQ-SDK-V1.3-20121114-yocto/meta-oe/meta-oe/recipes-support/atk/atk_1.32.0.bbappend
binutils_2.21.1a.bb:
  /opt/yt_sdks/QorIQ-SDK-V1.3-20121114-yocto/meta-fsl-ppc/recipes-devtools/binutils/binutils_2.21.1a.bbappend
[...]
```

- `show_overlayed` : List highest priority recipes with the recipes they overlay as subitems

```
$ bitbake-layers show_overlayed
```

Customizing Images

Customizing Images
Creating a New Layer

- When modifying or developing packages and images, it is advisable to work in a newly created custom layer
  - Avoids having to modify any of the SDK provided layers
- The following customization examples are fully contained within a new `meta-custom` layer
- This custom layer, its recipes and source files are available with this slide deck

Creating a New Layer
Defining a New Custom Layer

- Make a new layer directory: `<install-dir>/meta-custom`
- Create a new `meta-custom/conf/layer.conf` file from a copy of `meta-fsl-ppc/conf/layer.conf` and change as shown:

```
[meta-custom/conf/layer.conf]
# We have a packages directory, add to BBFILES
BBPATH := "${BBPATH}:${LAYERDIR}"

BBFILES += "${LAYERDIR}/recipes-*/*/*.bb*"
BBFILES += "${LAYERDIR}/images/*.bb*"

BBFILE_COLLECTIONS += "custom"
BBFILE_PATTERN_custom := "^${LAYERDIR}/"
BBFILE_PRIORITY_custom = "6"
```

```
Metadata from higher priority layers
overrides same from lower priority layers.
Execute "bitbake-layers show-layers" to see
layer priorities
```

Creating a New Layer (continued)

Enable the Custom Layer

- Edit the `conf/bblayers.conf` file in the build project:

```
[conf/bblayers.conf]
# LAYER_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
LCONF_VERSION = "4"

BBFILES ?= ""
BBLAYERS = " \
  /opt/yt_sdks/QorIQ-SDK-V1.4-20130625-yocto/meta \
  /opt/yt_sdks/QorIQ-SDK-V1.4-20130625-yocto/meta-yocto \
  /opt/yt_sdks/QorIQ-SDK-V1.4-20130625-yocto/meta-fsl-ppc \
  /opt/yt_sdks/QorIQ-SDK-V1.4-20130625-yocto/meta-oe/meta-oe \
  …
  /opt/yt_sdks/QorIQ-SDK-V1.4-20130625-yocto/meta-custom \
  "
```

Customizing the Root File System
Cloning and Modifying a Copy of an Existing Image Recipe

- Copy the original image recipe to your custom layer:

  -
    ```
    $ cp meta-fsl-networking/images/fsl-image-core.bb \
    meta-custom/images/custom-image-core.bb
    ```
- Add/remove packages from the `IMAGE_INSTALL` list in `custom-image-core.bb`
- The `require <file>` lines in the new recipe are no longer sourced from the current layer, but from `meta-fsl-networking`, so add `images/` to their relative paths
  - [custom-image-core.bb]
    ```
    [...]
    IMAGE_INSTALL += " \
        bridge-utils \
        coreutils \
        [...]
        perf \
        psmisc \
        tcpdump \
    "
    ```
- To build :

  -
    ```
    $ bitbake custom-image-core
    ```

Customizing the Root File System (continued)

Using A Custom <image>.bb File

- Create a new image recipe

```
$ touch meta-custom/images/custom-require-image-core.bb
```

- Edit the image recipe to `require` the settings from a pre-existing image recipe and add packages :

```
PR .= ".1"
IMAGE_INSTALL = "bridge-utils"
require images/fsl-image-core.bb
```

- To build:

```
$ bitbake custom-require-image-core
```

- It is not possible to remove packages from the list defined by the required external image recipe

Customizing the Root File System (continued)

In `conf/local.conf`

- Add `CORE_IMAGE_EXTRA_INSTALL = "<pkg> ..."`
  - Specifies the list of packages to be added to the image
- Add `IMAGE_INSTALL_append = " <pkg>"`
  - Note the leading space
  - Check configuration result with:

```
$ bitbake -e fsl-image-core | grep IMAGE_INSTALL
```

Customizing the Root File System (continued)

In `conf/local.conf`

- Add `EXTRA_IMAGE_FEATURES = "<feature>"`
- Available image features:
  - `"dbg-pkgs"` - Adds `-dbg` packages for all installed packages
  - `"dev-pkgs"` - Adds `-dev` packages for all installed packages
  - `"tools-sdk"` - Adds development tools such as `gcc, make, pkgconfig` and so forth.
  - `"tools-debug"` - Adds debugging tools such as `gdb` and `strace`.
  - `"tools-profile"` - Adds profiling tools such as `oprofile, exmap, lttng` and `valgrind` (x86 only).
  - `"tools-testapps"` - Adds useful testing tools such as ts_print, aplay, arecord and so forth.
  - `"debug-tweaks"` - Makes an image suitable for development.

Customizing the Root File System (continued)
Adding a Packagegroup to an Image Recipe

- Add in `<layer>/images/<image>.bb`

```
IMAGE_INSTALL += "... packagegroup-<xxx>“
```

Customizing the Root File System (continued)

ROOTFS_POSTPROCESS_COMMAND Variable

- To modify RFS content after package installation:
  - Add ROOTFS_POSTPROCESS_COMMAND variable to the image recipe, specifying commands to execute before image generation

```
[meta-custom/images/custom-image-core.bb]
require custom-rootfs_post_process.inc

[custom-rootfs_post_process.inc]
ROOTFS_POSTPROCESS_COMMAND += " rm -rf ${IMAGE_ROOTFS}/boot ; \
 rm -rf ${IMAGE_ROOTFS}/usr/include ; \
 rm -rf ${IMAGE_ROOTFS}/usr/share/info ; \
 ( find ${IMAGE_ROOTFS} -type d -name "man"    | xargs rm -rf ) ; \
 ( find ${IMAGE_ROOTFS} -type d -name "src"    | xargs rm -rf ) ; \
 ( find ${IMAGE_ROOTFS} -type d -name "doc"    | xargs rm -rf ) ; \
 ( find ${IMAGE_ROOTFS} -name "*python*"       | xargs rm -rf ) ; \
 ( find ${IMAGE_ROOTFS} -name "elf_*86*"       | xargs rm -rf ) ; \
 ( find ${IMAGE_ROOTFS} -name "elf_*64*"       | xargs rm -rf ) ; \
 ( find ${IMAGE_ROOTFS} -name "*openbios*"     | xargs rm -rf ) ; \
 ( find ${IMAGE_ROOTFS} -name "powerpc-fsl-*" | xargs rm -rf ) ; \
"
```

Customizing the Root File System (continued)
Add Extra Space

- Add in `<layer>/images/<image>.bb`

```
IMAGE_ROOTFS_EXTRA_SPACE = "<size_in_KB>"
```

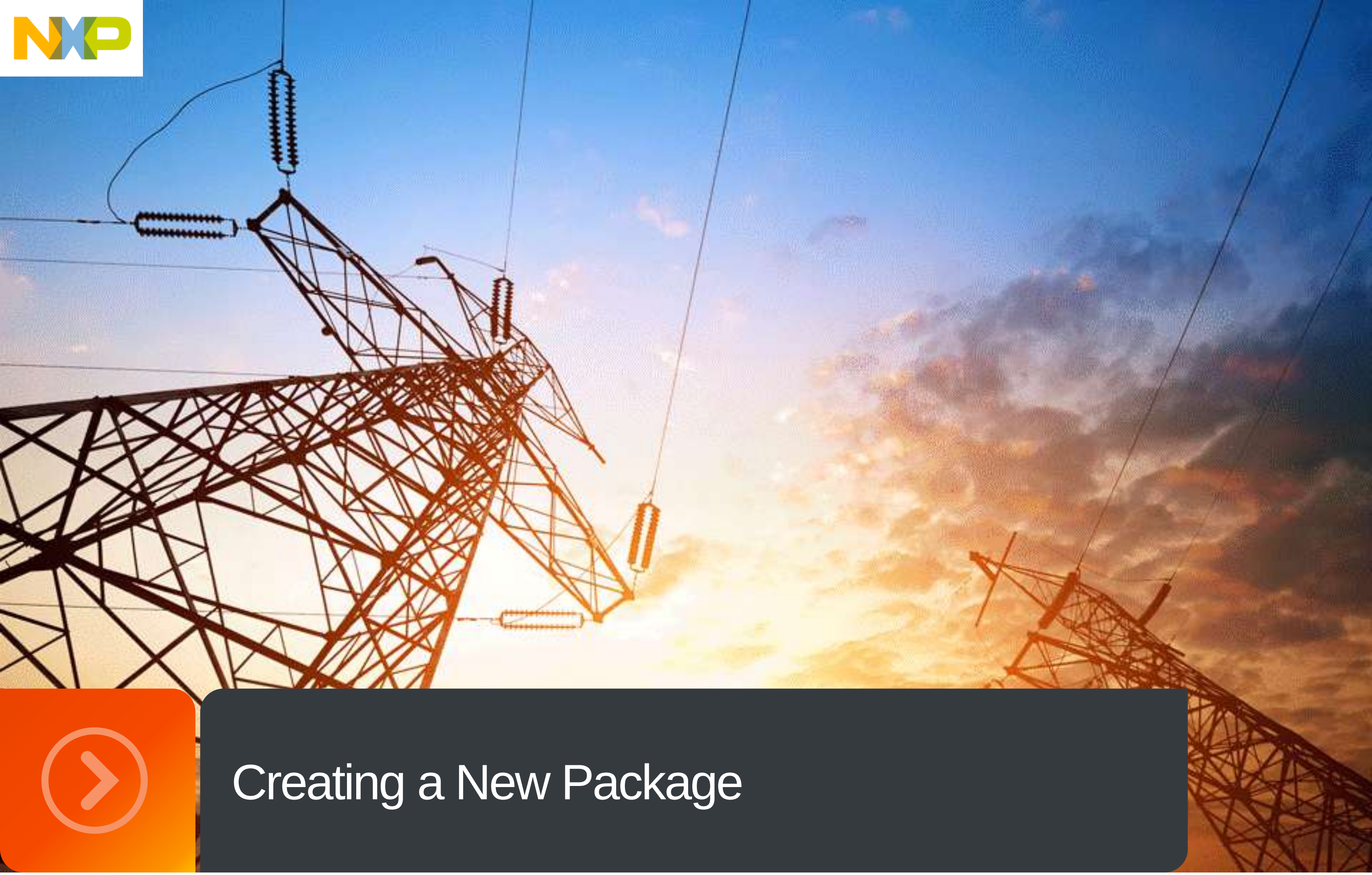
Applying RFS Configuration Changes

- Prior to regenerating the RFS, make sure to clean the `SSTATE` for any contributing package that requires a rebuild because of reconfiguration or source code change:

```
$ bitbake -c cleansstate <pkg>   or
$ bitbake -c cleansstate <image>
```

- Then invoke `bitbake` again:

```
$ bitbake <image>
```



Creating a New Package

Creating a New Package

Generic Recipe Directory Structure

```
meta-<layer>/recipes-<name>/<pkg>
├── <pkg>_<version>.bb              bitbake recipe
└── files                           local files folder
    ├── <pkg>.tar.gz                source archive
    ├── <patch name>.patch          Patch file(s)
    └── [...]                       Other locally stored files
```

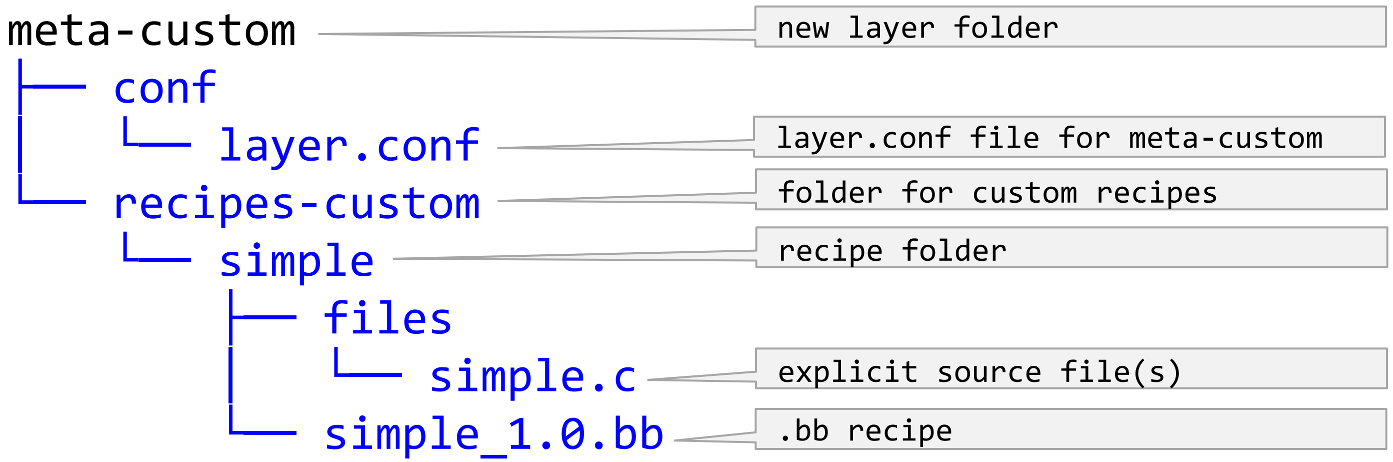
- `_<version>` will set the `${PV}` variable, that may be used in the recipe. Note the leading underscore '`_`'
- Yocto will always use the highest revision number of a package. To use a specific package version:

```
Set PREFERRED_VERSION_<pkg> = “version” in conf file
```

Creating a New Package (continued)

Organize Recipes

- Make one or more directories to group your recipes, e.g. meta-custom/recipes-custom
- Make a sub-directory for each recipe, e.g. `simple`
  - Write a `<pkg>_<version>.bb` recipe, e.g. `simple_1.0.bb`
  - Populate with the required explicit files, tar balls and patches



```
meta-custom
├── conf
│   └── layer.conf
└── recipes-custom
    └── simple
        ├── files
        │   └── simple.c
        └── simple_1.0.bb
```

Creating a New Package (continued)

Recipe Requirements

- A recipe should define:
  - `DESCRIPTION` : package description (*)
  - `LICENSE` : list of package source licenses. For closed source packages, use `LICENSE="CLOSED"`
  - `LIC_FILES_CHKSUM` : checksums of the license text in the recipe source code
  - `SECTION` : section where package should be put (*)
  - `HOMEPAGE` : website with info about package
  - `AUTHOR` : email address used to contact the original author or authors in order to send patches, forward bugs, etc.
  - `SRC_URI` : list of source files - local or remote

(*) used by package managers.

Creating a New Package (continued)

Recipe for a Package with Local Source Files

- SRC_URI variable must list the locally stored source files
- Write a do_compile and do_install task

```
[meta-custom/recipes-custom/simple/simple_1.0.bb]
DESCRIPTION = "Simple application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://simple.c"
S = "${WORKDIR}/simple"

do_compile() {
${CC} ${WORKDIR}/simple.c -o ${S}/simple
}

do_install() {
install -d ${D}${bindir}
install -m 0755 ${S}/simple ${D}${bindir}
}
```

.c source from local files folder

Output folder

Creating a New Package (continued)

Content of ${WORKDIR} After Building

```
ppce500mc-fsl-linux/simple-1.0-r0/ ---------------- ${WORKDIR}
├── debugsources.list
├── deploy-rpms
│   └── ppce500mc
│       ├── simple-1.0-r0.ppce500mc.rpm ----------- Installable binary .rpm
│       ├── simple-dbg-1.0-r0.ppce500mc.rpm
│       └── simple-dev-1.0-r0.ppce500mc.rpm
├── image
│   └── usr
│       └── bin
│           └── simple ---------------------------- Location of binary in rootfs image
├── license-destdir
│   └── simple
├── package
├── packages-split
├── pkgdata
├── pseudo
├── shlibs
├── simple
│   └── simple ------------------------------------ Compiled binary
├── simple.c -------------------------------------- Source file
├── simple.provides
├── simple.requires
├── simple.spec
├── sysroot-destdir
└── temp
```

Creating a New Package (continued)

Recipe for a Package Using `autoconf/automake`

- `SRC_URI` variable must list the source archive
- Use `inherit autotools`
- E.g. for the GNU `hello` package
  (source archive is remotely stored on the GNU mirror):

```
[meta-custom/recipes-custom/hello/hello_2.8.bb]
DESCRIPTION = "GNU Helloworld application"
SECTION = "examples"
LICENSE = "GPLv2+"
LIC_FILES_CHKSUM = "file://COPYING;md5=d32239bcb673463ab874e80d47fae504 "

SRC_URI = "${GNU_MIRROR}/hello/hello-${PV}.tar.gz"
inherit autotools gettext
```

Creating a New Package (continued)
Recipe for a `Makefile`-based Package

- Recipe must:
  - List source archive in the `SRC_URI` variable
  - Store additional make options in the `EXTRA_OEMAKE` variable
  - Provide manually written `do_install` task
- The following example:
  - Pulls `mtd-utils v1.5.0` from upstream git and builds the package using its `Makefile`

Creating a New Package (continued)
Makefile-based

```
[meta-custom/recipes-custom/mtd-utils_1.5.0/mtd-utils_1.5.0.bb]
DESCRIPTION = "Tools for managing memory technology devices."
SECTION = "base"
HOMEPAGE = "http://www.linux-mtd.infradead.org/"

LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=0636e73ff0215e8d672dc4c32c317bb3 \
file://include/common.h;beginline=1;endline=17;md5=ba05b07912a44ea2bf81ce409380049c"

DEPENDS = "zlib lzo e2fsprogs util-linux"
SRC_URI = "git://git.infradead.org/mtd-utils.git;protocol=git;tag=v${PV}"
S = "${WORKDIR}/git/"

EXTRA_OEMAKE = "'CC=${CC}' 'CFLAGS=${CFLAGS} -I${S}/include \
-DWITHOUT_XATTR' 'BUILDDIR=${S}'"

do_install () {
oe_runmake install DESTDIR=${D} SBINDIR=${sbindir} MANDIR=${mandir} \
INCLUDEDIR=${includedir}
install -d ${D}${includedir}/mtd/
for f in ${S}/include/mtd/*.h; do
          install -m 0644 $f ${D}${includedir}/mtd/
done
}
```

Merge Files

Adding Unmanaged Content to a Root File System

- Unmanaged ➔ you have not created a recipe
- By using a `merge-files` recipe your selection of unmanaged files and directories:
  - Will be packaged as an installable `.rpm`
  - Which will be deployed after all other packages and before image files are created
- It is preferable to make a custom copy of the SDK-provided `merge-files` recipe and make modifications there

...ge Files (continued)

Example : Custom `merge-files`

- Make a copy of the `merge-files` recipe provided by the `meta-fsl-networking layer`:

```
$ cp -a meta-fsl-networking/recipes-tools/merge-files \
meta-custom/merge-files-custom
```

- Rename the `.bb` file:

```
$ cd meta-custom/merge-files-custom
$ mv merge-files_1.0.bb merge-files-custom_1.0.bb
```

- Copy the unmanaged file and directory content into `./files/merge`

```
$ cp -a <source location> ./files/merge
```

Merge Files (continued)

Example : Custom `merge-files`

- The recipe folder now has this layout:

```
meta-custom/recipes-custom/merge-files-custom/
├── files
│   └── merge
│       ├── <files...>
│       └── <folder>
│           └── <files...>
└── merge-files-custom_1.0.bb
```

- Rebuild the `merge-files-custom` package:

```
$ bitbake merge-files-custom
```

Merge Files (continued)

Example : Custom `merge-files`

- Add the `merge-files-custom` package to a custom image
- Regenerate the image files
- As an alternative to copying individual files and directories into `files/merge`:
  - Create a tar ball with an `.md5sum` file
  - Change the `SRC_URI` in the `merge-files-custom` recipe so that the archive is sourced
  - Update the tar ball and checksum when needed

Tips and Tricks

BitBake devshell

`bitbake -c devshell` Command

- To work interactively with a package:

```
$ bitbake -c devshell <pkg>
```

- This opens a terminal window with a shell prompt within the SDK environment:
  - with `PATH` variable set to include the cross toolchain
  - `pkgconfig` variables find the correct `.pc` files.
  - `configure` finds all the necessary files
  - working directory is changed to the `${S}` directory
- Within this `devshell` you can manually execute configure commands or compile

BitBake devshell (continued)

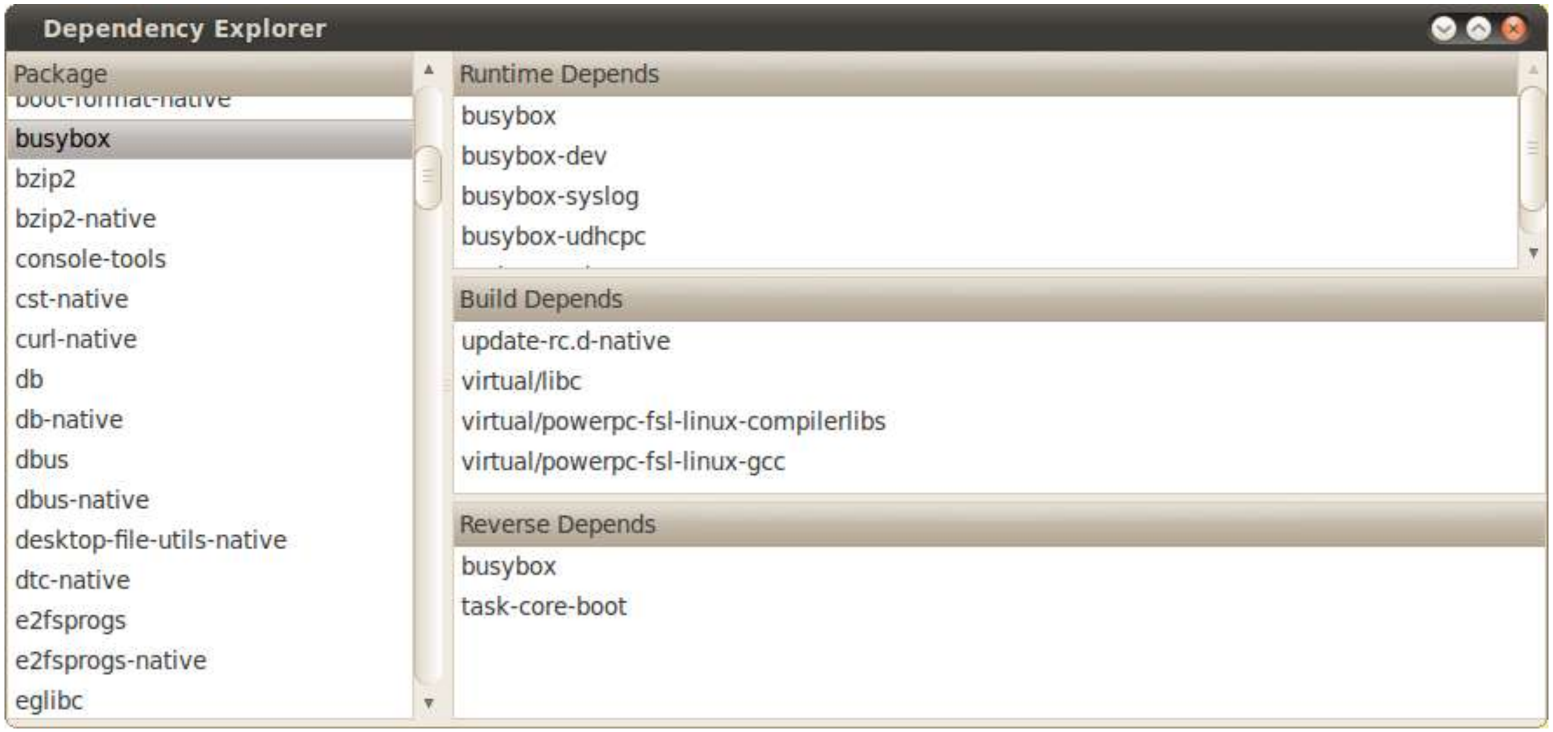
Examples

`$ bitbake -c devshell virtual/kernel` → devshell console starts

```
$ pwd
linux-qoriq-sdk-3.0.48-r5/git
$ make menuconfig     → menuconfig UI opens
scripts/kconfig/mconf Kconfig
*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.
$ export LDFLAGS=""  ← required
$ make modules
[...]
 Building modules, stage 2.
[...]
  LD [M]  net/sctp/sctp.ko
$ make uImage
[...]
  WRAP    arch/powerpc/boot/uImage
Image Name:   Linux-3.0.48-rt70
Created:      Thu Jan 31 10:29:22 2013
Image Type:   PowerPC Linux Kernel Image (gzip compressed)
Data Size:    3859555 Bytes = 3769.10 kB = 3.68 MB
Load Address: 00000000
Entry Point:  00000000
```

BitBake Dependency Explorer

- `$ bitbake -g fsl-image-minimal -u depexp`



Dependency Explorer

Package
busybox
bzip2
bzip2-native
console-tools
cst-native
curl-native
db
db-native
dbus
dbus-native
desktop-file-utils-native
dtc-native
e2fsprogs
e2fsprogs-native
eglibc

Runtime Depends
busybox
busybox-dev
busybox-syslog
busybox-udhcpc

Build Depends
update-rc.d-native
virtual/libc
virtual/powerpc-fsl-linux-compilerlibs
virtual/powerpc-fsl-linux-gcc

Reverse Depends
busybox
task-core-boot

Dump BBFILES and BBPATH Variables

Debugging BitBake environment issues

```
$ bitbake -e | grep -e "^BBFILES" | tr ' ' '\n'
BBFILES="/bsps/QorIQ-SDK-V1.4-20130625-yocto/meta/recipes-*/*/*.bb
/bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-yocto/recipes-*/*/*.bb
/bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-yocto/recipes-*/*/*.bbappend
…
/bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-fsl-ppc/recipes-*/*/*.bb*
/bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-fsl-ppc/images/*.bb*
$ bitbake -e | grep -e "^BBPATH" | tr ':' '\n'
BBPATH="/bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-yocto
/bsps/QorIQ-SDK-V1.4-20130625-yocto/meta
…
/bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-fsl-ppc-toolchain
/bsps/QorIQ-SDK-V1.4-20130625-yocto/meta-fsl-networking
```

`BBFILES`	= List of recipe files used by BitBake to build software
`BBPATH`	= Used by BitBake to locate .bbclass and configuration files, i.e. like a PATH variable

Reducing SDK Disk Footprint

- If the project was not created with `-l` (`lite` mode)
  - Append to `<project_dir>/conf/local.conf` :

```
# delete sources after build
INHERIT += "rm_work"
```

  - When acutely running out of disk space during builds :
  - Delete work directories for non-machine specific packages :

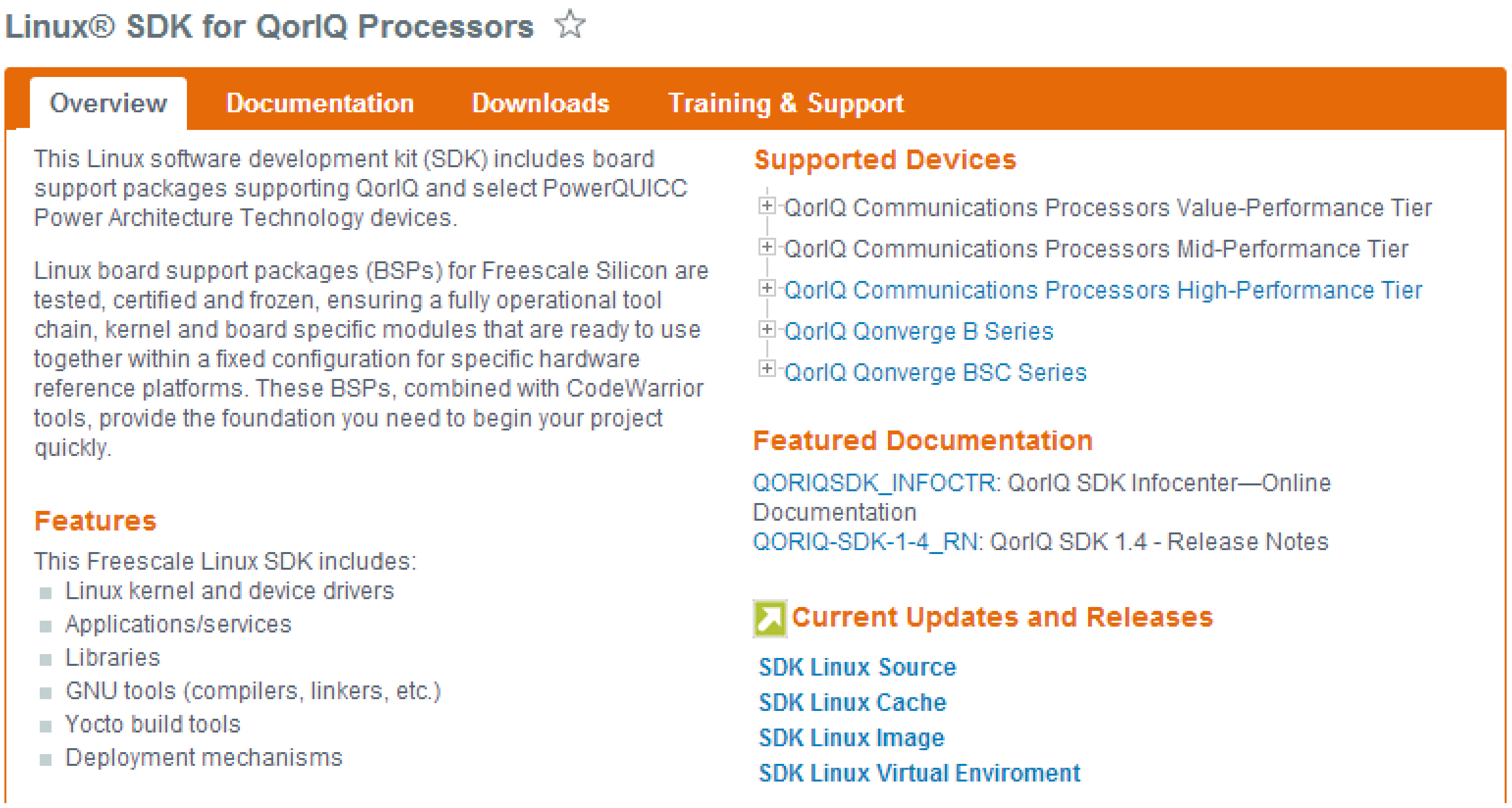
```
$ rm tmp/work/i686-linux/*
$ rm tmp/work/ppc*-fsl-linux/*
```

  - When multiple build projects exist, move their respective `sysroots` host binaries to a common folder, then create symlinks from each build project, e.g.:

```
$ mv -a tmp/sysroots/i686-linux ..
$ ln -s ../../../i686-linux tmp/sysroots/i686-linux
```

Recent QorIQ SDK Release

www.freescale.com/webapp/sps/site/prod_summary.jsp?code=SDKLINUX


Linux® SDK for QorIQ Processors

Overview | Documentation | Downloads | Training & Support

This Linux software development kit (SDK) includes board support packages supporting QorIQ and select PowerQUICC Power Architecture Technology devices.

Linux board support packages (BSPs) for Freescale Silicon are tested, certified and frozen, ensuring a fully operational tool chain, kernel and board specific modules that are ready to use together within a fixed configuration for specific hardware reference platforms. These BSPs, combined with CodeWarrior tools, provide the foundation you need to begin your project quickly.

Features

This Freescale Linux SDK includes:

- Linux kernel and device drivers
- Applications/services
- Libraries
- GNU tools (compilers, linkers, etc.)
- Yocto build tools
- Deployment mechanisms

Supported Devices

- QorIQ Communications Processors Value-Performance Tier
- QorIQ Communications Processors Mid-Performance Tier
- QorIQ Communications Processors High-Performance Tier
- QorIQ Qonverge B Series
- QorIQ Qonverge BSC Series

Featured Documentation

QORIQSDK_INFOCTR: QorIQ SDK Infocenter—Online Documentation
QORIQ-SDK-1-4_RN: QorIQ SDK 1.4 - Release Notes

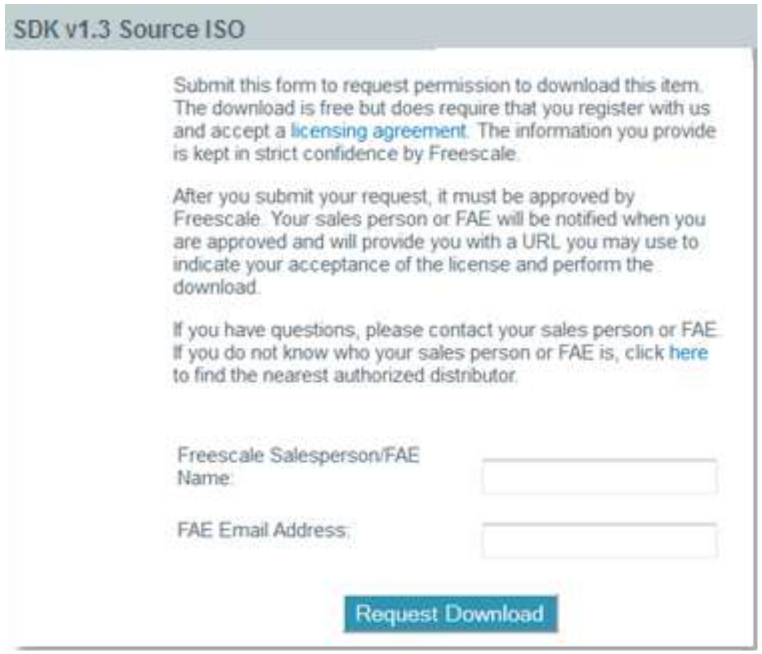
Current Updates and Releases

SDK Linux Source
SDK Linux Cache
SDK Linux Image
SDK Linux Virtual Enviroment

- Log into `freescale.com` to download SDK releases and updates

Recent QorIQ SDK Release (continued)
Moderated Downloads

- Release ISO's and Virtual SDK environments per architecture are available through moderated download
- Submit the download request, specifying Name and Email Address of your Freescale Sales person or FAE



SDK v1.3 Source ISO

Submit this form to request permission to download this item. The download is free but does require that you register with us and accept a licensing agreement. The information you provide is kept in strict confidence by Freescale.

After you submit your request, it must be approved by Freescale. Your sales person or FAE will be notified when you are approved and will provide you with a URL you may use to indicate your acceptance of the license and perform the download.

If you have questions, please contact your sales person or FAE. If you do not know who your sales person or FAE is, click here to find the nearest authorized distributor.

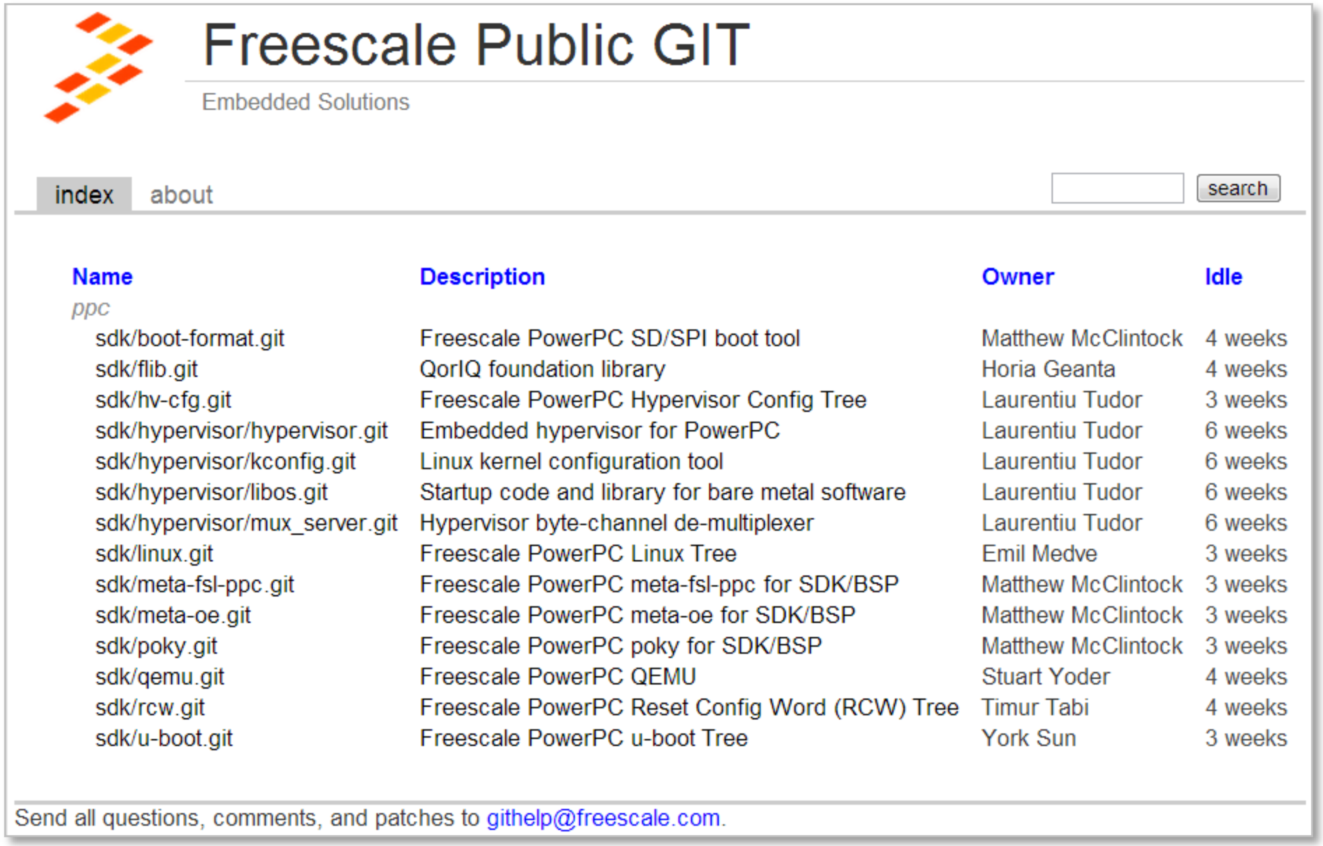
Freescale Salesperson/FAE Name:	
FAE Email Address:	

Recent QorIQ SDK Release (continued)

Freescale Public git Server

- Public source code for the SDK can be also pulled from:

`git.freescale.com/git/cgit.cgi/ppc/sdk`



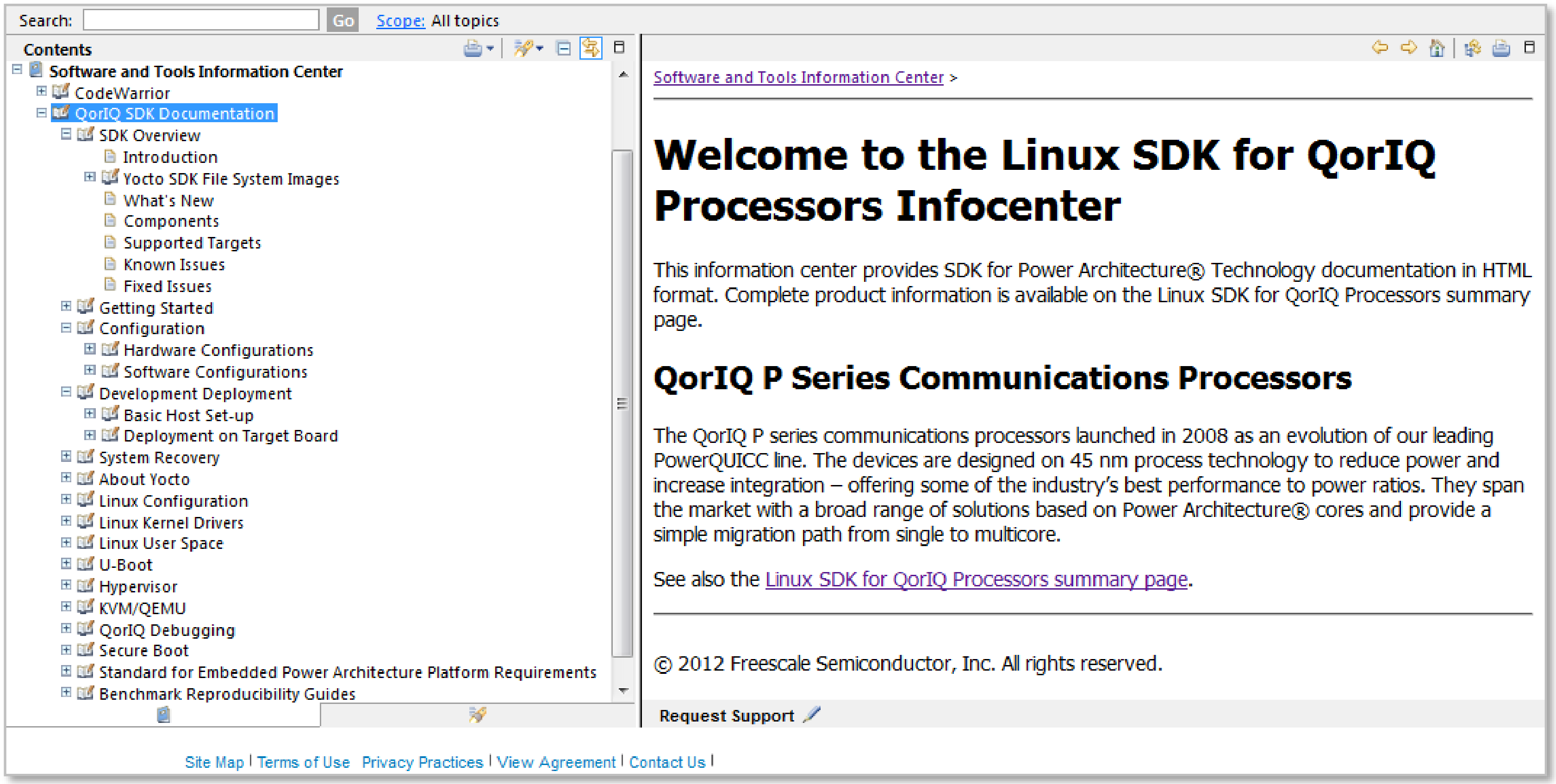
Freescale Public GIT

Embedded Solutions

Name	Description	Owner	Idle
ppc			
sdk/boot-format.git	Freescale PowerPC SD/SPI boot tool	Matthew McClintock	4 weeks
sdk/flib.git	QorIQ foundation library	Horia Geanta	4 weeks
sdk/hv-cfg.git	Freescale PowerPC Hypervisor Config Tree	Laurentiu Tudor	3 weeks
sdk/hypervisor/hypervisor.git	Embedded hypervisor for PowerPC	Laurentiu Tudor	6 weeks
sdk/hypervisor/kconfig.git	Linux kernel configuration tool	Laurentiu Tudor	6 weeks
sdk/hypervisor/libos.git	Startup code and library for bare metal software	Laurentiu Tudor	6 weeks
sdk/hypervisor/mux_server.git	Hypervisor byte-channel de-multiplexer	Laurentiu Tudor	6 weeks
sdk/linux.git	Freescale PowerPC Linux Tree	Emil Medve	3 weeks
sdk/meta-fsl-ppc.git	Freescale PowerPC meta-fsl-ppc for SDK/BSP	Matthew McClintock	3 weeks
sdk/meta-oe.git	Freescale PowerPC meta-oe for SDK/BSP	Matthew McClintock	3 weeks
sdk/poky.git	Freescale PowerPC poky for SDK/BSP	Matthew McClintock	3 weeks
sdk/qemu.git	Freescale PowerPC QEMU	Stuart Yoder	4 weeks
sdk/rcw.git	Freescale PowerPC Reset Config Word (RCW) Tree	Timur Tabi	4 weeks
sdk/u-boot.git	Freescale PowerPC u-boot Tree	York Sun	3 weeks

Send all questions, comments, and patches to githelp@freescale.com.

Recent QorIQ SDK Documentation
On-line : `http://www.freescale.com/infocenter`



Search: Go Scope: All topics

Contents

- Software and Tools Information Center
  - CodeWarrior
  - QorIQ SDK Documentation
    - SDK Overview
      - Introduction
      - Yocto SDK File System Images
      - What's New
      - Components
      - Supported Targets
      - Known Issues
      - Fixed Issues
    - Getting Started
    - Configuration
      - Hardware Configurations
      - Software Configurations
    - Development Deployment
      - Basic Host Set-up
      - Deployment on Target Board
    - System Recovery
    - About Yocto
    - Linux Configuration
    - Linux Kernel Drivers
    - Linux User Space
    - U-Boot
    - Hypervisor
    - KVM/QEMU
    - QorIQ Debugging
    - Secure Boot
    - Standard for Embedded Power Architecture Platform Requirements
    - Benchmark Reproducibility Guides

Software and Tools Information Center >

Welcome to the Linux SDK for QorIQ Processors Infocenter

This information center provides SDK for Power Architecture® Technology documentation in HTML format. Complete product information is available on the Linux SDK for QorIQ Processors summary page.

QorIQ P Series Communications Processors

The QorIQ P series communications processors launched in 2008 as an evolution of our leading PowerQUICC line. The devices are designed on 45 nm process technology to reduce power and increase integration – offering some of the industry's best performance to power ratios. They span the market with a broad range of solutions based on Power Architecture® cores and provide a simple migration path from single to multicore.

See also the Linux SDK for QorIQ Processors summary page.

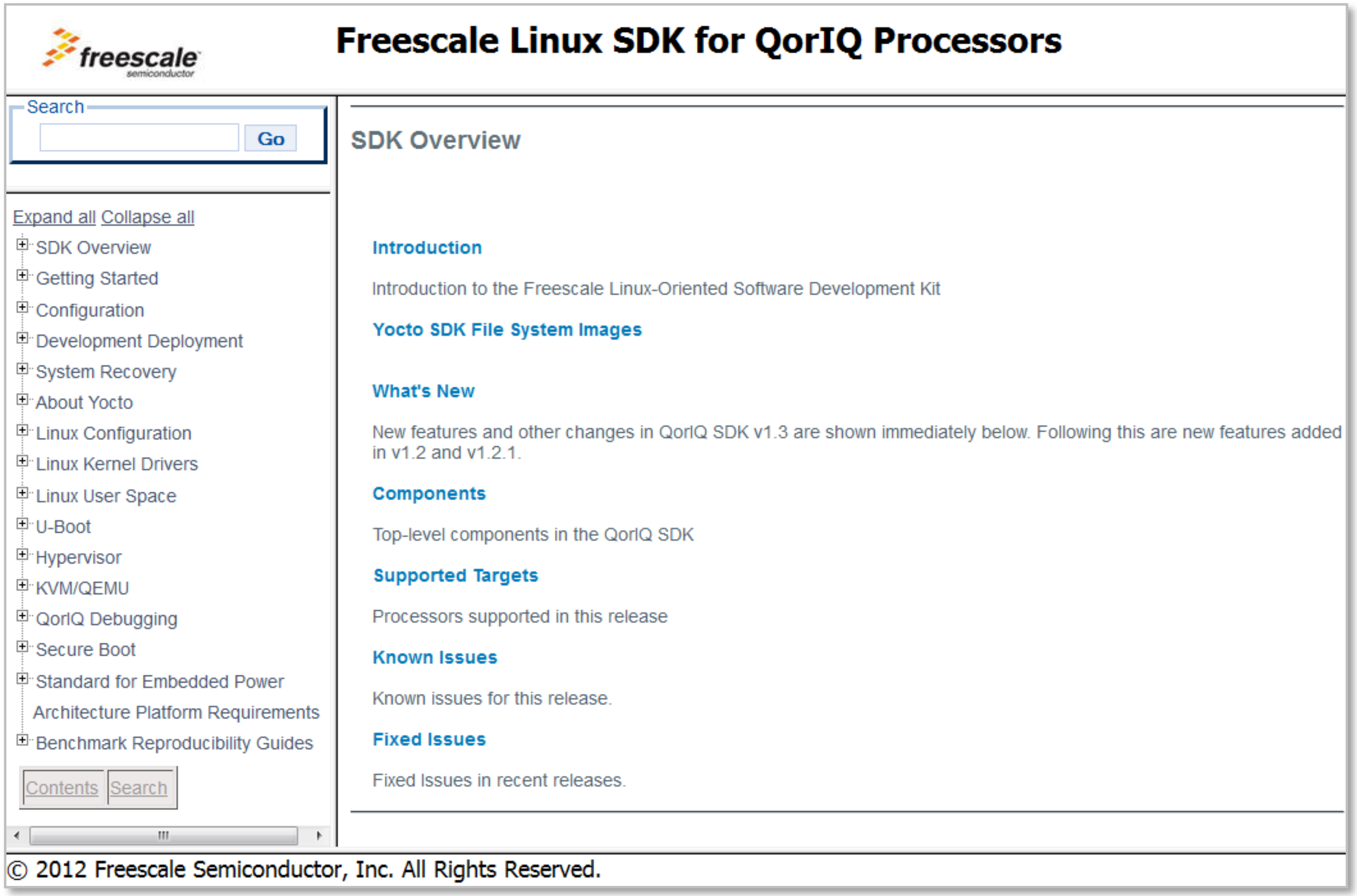
© 2012 Freescale Semiconductor, Inc. All rights reserved.

Request Support

Site Map | Terms of Use Privacy Practices | View Agreement | Contact Us |

Recent QorIQ SDK Documentation (continued)

InfoCenter Lite : `<SOURCE ISO>/documents/START_HERE.htm`



Freescale Linux SDK for QorIQ Processors

- SDK Overview
- Getting Started
- Configuration
- Development Deployment
- System Recovery
- About Yocto
- Linux Configuration
- Linux Kernel Drivers
- Linux User Space
- U-Boot
- Hypervisor
- KVM/QEMU
- QorIQ Debugging
- Secure Boot
- Standard for Embedded Power Architecture Platform Requirements
- Benchmark Reproducibility Guides

SDK Overview

Introduction

Introduction to the Freescale Linux-Oriented Software Development Kit

Yocto SDK File System Images

What's New

New features and other changes in QorIQ SDK v1.3 are shown immediately below. Following this are new features added in v1.2 and v1.2.1.

Components

Top-level components in the QorIQ SDK

Supported Targets

Processors supported in this release

Known Issues

Known issues for this release.

Fixed Issues

Fixed Issues in recent releases.

© 2012 Freescale Semiconductor, Inc. All Rights Reserved.

PDF version : `<SOURCE ISO>/Documentation/sdk_documentation/pdf/QorIQ_SDK_Infocenter.pdf`

Introducing The QorIQ LS2 Family

Breakthrough, software-defined approach to advance the world's new virtualized networks

New, high-performance architecture built with ease-of-use in mind
Groundbreaking, flexible architecture that abstracts hardware complexity and enables customers to focus their resources on innovation at the application level

Optimized for software-defined networking applications
Balanced integration of CPU performance with network I/O and C-programmable datapath acceleration that is right-sized (power/performance/cost) to deliver advanced SoC technology for the SDN era

Extending the industry's broadest portfolio of 64-bit multicore SoCs
Built on the ARM® Cortex®-A57 architecture with integrated L2 switch enabling interconnect and peripherals to provide a complete system-on-chip solution

QorIQ LS2 Family

Key Features

SDN/NFV Switching

Data Center

Wireless Access

Unprecedented performance and ease of use for smarter, more capable networks

High performance cores with leading interconnect and memory bandwidth

- 8x ARM Cortex-A57 cores, 2.0GHz, 4MB L2 cache, w Neon SIMD
- 1MB L3 platform cache w/ECC
- 2x 64b DDR4 up to 2.4GT/s

A high performance datapath designed with software developers in mind

- New datapath hardware and abstracted acceleration that is called via standard Linux objects
- 40 Gbps Packet processing performance with 20Gbps acceleration (crypto, Pattern Match/RegEx, Data Compression)
- Management complex provides all init/setup/teardown tasks

Leading network I/O integration

- 8x1/10GbE + 8x1G, MACSec on up to 4x 1/10GbE
- Integrated L2 switching capability for cost savings
- 4 PCIe Gen3 controllers, 1 with SR-IOV support
- 2 x SATA 3.0, 2 x USB 3.0 with PHY

See the LS2 Family First in the Tech Lab!

4 new demos built on QorIQ LS2 processors:

- Performance Analysis Made Easy
- Leave the Packet Processing To Us
- Combining Ease of Use with Performance
- Tools for Every Step of Your Design

www.Freescale.com