Linux BSP 36.0_cd User Manual for S32G3 platforms

07-Feb-2023

Contents

1 About this Document

1.1 References

Reference	File Name	Version
S32G3 Reference Manual	S32G3_RM.pdf	Rev. 0.5
S32G3 GMAC Subsystem Reference Manual	RM724301-S32G3 GMAC Subsystem Reference Manual(0.1).pdf	Rev. 0.1
S32G3 SerDes Subsystem Reference Manual	RM724401-S32G3 SerDes Subsystem Reference Manual(0.1).pdf	0.1
PFE Firmware User Manual	PFE_Firmware_S32G_UserManual.pdf	1.5.0
PFE Driver User Manual	PFE_S32G_A53_LNX_UserManual.pdf	1.2.0
LLCE Firmware User Guide	LLCE_firmware_user_guide.pdf	1.0.5
LLCE Firmware Getting Started Guide	LLCE_getting_started_guide.pdf	1.0.5
HSE Service API Reference Manual	S32G3XX_Service_API_Reference_Manual.pdf	0.2.16.1
RDB3 User Guide	UG729203-RDB3 User Guide(0.3).pdf	0.3
RDB3 Reference Manual	RM729303-RDB3 Reference Manual(0.3).pdf	0.3
RDB3 Ethernet Enablement Guide (Secure File)	UG777201-RDB3 Ethernet Enablement Guide(0.1).pdf	0.1
EVB3 User Guide	UG717501-EVB3 User Guide(0.1).pdf	0.1
EVB3 Quick Start Guide	UG717601-EVB3 Quick Start Guide(0.1).pdf	0.1

1.2 Terms and abbreviations

Acronym	Definition
ATF, TF-A	ARM Trusted Firmware for Cortex-A cores
BSP	Board Support Package
EL0,1,2,3	Exception Level 0,1,2,3 for A53 core
GCC	GNU C Compiler Suite
MTD	Memory Technology Device
NFS	Network File System
SD	Secure Digital
TZ	ARM TrustZone
S32CC	Sub-group of S32 processor family containing S32G and S32R processors

2 Introduction

The Linux BSP is a collection of source code that can be used to create U-Boot boot loader, Linux kernel image, a root filesystem and an ARM Trusted Firmware (TF-A) images for the supported boards.

2.1 ARM Trusted Firmware (TF-A)

The Linux BSP delivery package contains the following ARM Trusted Firmware binary: *auto_linux_bsp36.0_cd/<board>-/fip.s32-<sdcard/qspi>*.

This ATF firmware supports SD/MMC/eMMC (*fip.s32-sdcard*) and QSPI (*fip.s32-qspi*).

2.2 Linux Kernel Image

This Linux BSP contains a pre-built kernel image based on the v5.15.85 of the Linux kernel. The kernel image and dtb file is located at the following path: auto_linux_bsp36.0_cd_<soc>/<board>

2.3 Root File System

The package contains the following rootfs file system: *binaries_auto_linux_36.0_cd_<soc>/<board>/fsl-image-[auto/base/flash]-<board>.tar.gz*. The *rootfs.tar.gz* file system includes NXP specific libraries. Its contents can be mounted as a NFS share or can be stored on a boot media such as Secure Digital (SD) card.

3 Build the Linux BSP

This Linux BSP has been built and tested using the GCC 10.2.0 for ARM64 cross-compiler included in the Yocto 4.0.5 "Kirkstone" distribution.

All the steps described below have been run and validated on Ubuntu-20.04 LTS (native or through a virtual machine). It is then recommended to install Ubuntu-20.04 LTS before going through the following sections.

3.1 Building the Linux BSP using YOCTO

3.1.1 Building default YOCTO

Update the package manager:

```
sudo <pkg-mgr> update
```

Install dependencies:

- python 2.x - 2.6 or newer:

  ```
  sudo <pkg-mgr> install python
  ```

- git 1.8.3 or newer:

  ```
  sudo <pkg-mgr> install git
  ```

- curl:

  ```
  sudo <pkg-mgr> install curl
  ```

  where <pkg-mgr> is the package manager for your distribution (apt-get or apt for Debian/Ubuntu, yum or dnf for CentOS/Fedora, zypper for SUSE).

To get the BSP you need to have **repo** installed. Please install it using the following commands (this only needs to be done once):

```
mkdir ~/bin
curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
chmod a+x ~/bin/repo
PATH=${PATH}:~/bin
```

Configure your git environment (you may skip this option if you have git already configured):

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

Next, download the Yocto Project Environment into your directory:

```
mkdir fsl-auto-yocto-bsp
cd fsl-auto-yocto-bsp
repo init -u https://github.com/nxp-auto-linux/auto_yocto_bsp -b release/bsp36.0_cd
repo sync
```

This will download the sources for the latest NXP Auto Linux BSP (from the branch release/bsp36.0_cd), structured on top of the Yocto Kirkstone release and upstream NXP QorIQ SDK.

Optional manifests options:

The repository provides more manifest files, particularized for different use cases. The desired manifest file is selected by specifying it in the repo init command, using parameter `-m <manifest file>`, e.g:

```
repo init -u https://github.com/nxp-auto-linux/auto_yocto_bsp -b release/bsp36.0_cd
-m <manifest file>.xml
```

The manifests are delivered in two formats: *<manifest>.xml* and *<manifest>-bitbucket.xml*. The latter would allow fetching internal bitbucket development repositories.

Enter the directory fsl-auto-yocto-bsp and follow below instructions (available in the *README* file).

Note:

A Yocto build needs at least 50GB of free space and takes a lot of time (2 to 10 hours, depending on the system configuration). It is recommended to use a powerful system with many cores and a fast storage media (SSD is recommended). The recommended RAM size is 8 GB.

3.1.2 BSP Yocto Build

To build the Linux BSP, please follow the steps:

1. First time setup (you need to have rights to execute *sudo apt-get <cmd>* without password).

```
sudo apt update
```

```
./sources/meta-alb/scripts/host-prepare.sh
```

2. Creating Build Directories and Testing the Installation. Now you can create a build directory in the SDK root with:

```
source nxp-setup-alb.sh -m <machine>
```

For example, if targeting S32G3 EVB, machine is `s32g399aevb3`:

```
source nxp-setup-alb.sh -m s32g399aevb3
```

3. When this is done, a `bitbake <imagename>`, e.g.

```
bitbake fsl-image-base
```

would be enough to completely build U-Boot, kernel, modules, the TF-A and a rootfs ready to be deployed. Look for a build result in *<builddirectory>/tmp/deploy/images/*.

If the machine chosen was `s32g399aevb3`, after the successful finalization of the build, the results will be placed in the *build_s32g399aevb3/tmp/deploy/images/s32g399aevb3* directory. Results for other machines will be placed in similar directories, but named according to the machine name. This release includes support for:

- Machines: `s32g399aevb3`, `s32g398aevb3`, `s32g379aevb3`, `s32g378aevb3`, `s32g399ardb3`;
- Images: fsl-image-base, fsl-image-auto.

After deploying the Yocto image and booting the platform, please use the following credentials to log in:

- The user root with no password

3.1.3 BSP Ubuntu Build

In case of building Ubuntu target images, this release includes support for versions 22.04.1 LTS and 20.04.1 LTS. To build the Linux BSP, please follow the steps:

1. First time setup (you need to have rights to execute *sudo apt-get <cmd>* without password).

```
sudo apt update
```

```
./sources/meta-alb/scripts/host-prepare.sh
```

2. Creating Build Directories and Testing the Installation. Now you can create a build directory in the SDK root with:

```
source nxp-setup-alb.sh -m <machine>ubuntu
```

For example, if targeting S32G3 EVB, machine is `s32g399aevb3ubuntu`:

```
source nxp-setup-alb.sh -m s32g399aevb3ubuntu
```

3. When this is done, a `bitbake <imagename>`, e.g.

```
bitbake fsl-image-ubuntu-base
```

would be enough to completely build U-Boot, kernel, modules, the TF-A and a rootfs ready to be deployed. Look for a build result in *<builddirectory>/tmp/deploy/images/*.

This release includes support for:

- Machines: `s32g399aevb3ubuntu, s32g398aevb3ubuntu, s32g379aevb3ubuntu, s32g378aevb3ubuntu, s32g399ardb3ubuntu`;
- Images: fsl-image-ubuntu-base, fsl-image-ubuntu, fsl-image-ubuntu-ros.

In case of Ubuntu images, by default Ubuntu target version is 22.04.1.
If targeting Ubuntu-20.04 images, add the following line in *<builddirectory>/conf/local.conf*.

```
UBUNTU_TARGET_VERSION = "20.04.1"
```

Then run `bitbake <imagename>`, where <imagename> is any of the supported Ubuntu images specified above. After deploying a Ubuntu image and booting the platform, please use the following credentials to log in:

- user: *bluebox*
- password: *bluebox*

3.1.4 Building OP-TEE-Enabled Image

OP-TEE can be built and deployed by editing *<builddirectory>/conf/local.conf* and appending the following line:

```
DISTRO_FEATURES:append = " optee"
```

3.1.5 Building Xen-Hypervisor-Enabled Image

Xen Hypervisor is an Open-Source Type-1 (Bare-metal) hypervisor. The support for Xen is an optional feature that can be enabled by appending the following line to *<builddirectory>/conf/local.conf*:

```
DISTRO_FEATURES:append = " xen"
```

When Xen support is enabled, the BSP components are built with Xen support (where needed): U-Boot loads Xen at EL2 with Xen-specific booting commands, Linux Kernel image is built with support for paravirtualization, and Xen tools and libraries are installed into the root filesystem.

Please refer to Chapter "Building Custom Xen Setups" for more information about how to build and use different Xen configurations.

Note 1: Currently, enabling both Xen and OP-TEE in the same image is not supported.

3.1.6 Building Images with M7 as Boot Target

Cortex-M7 booting flow is provided as an example for enabling lockstep operation mode. It is enabled by editing *<builddirectory>/conf/local.conf* and appending the following line:

```
DISTRO_FEATURES:append = " m7boot"
```

When Cortex-M7 booting flow is enabled, the generated SD card or flash images can be used in the same way as for default Cortex-A53 booting flow. These images include all the needed changes: Image Vector Table (IVT) updates and addition of Cortex-M7 software. In case IVT files needs to be deployed manually, when Cortex-M7 booting flow is enabled the files with **.m7* (e.g. *fip*.m7*) extensions should be used.

3.1.7 Additional instructions for developers

This chapter contains instructions for developers and assumes that the reader is familiar with the Yocto build process as well as the Kbuild framework used by U-Boot and the Linux kernel.

3.1.7.1 How to locate U-Boot, Linux and other sources in the Yocto tree

The following paragraphs assume you are in *<Yocto folder>/build_<machine>* build directory after you have sourced `nxp-setup-alb.sh` (so that `bitbake` is found in $PATH).

Locate the source for any packages, use the command below run from the build directory:

```
bitbake -e <package> | grep "^S="
```

This command will go through the package environment and find out information about the location of the source. Example:

U-Boot source:

```
bitbake -e u-boot | grep "^S="
S="<yocto root>/build_<machine>/tmp/work/<machine>-fsl-linux/u-boot-s32/2020.04-r0/git"
```

Linux source:

```
bitbake -e virtual/kernel | grep  "^S="
S="<yocto root>/build_<machine>/tmp/work/<machine>-fsl-linux/linux-s32/<kernel_version>-r0/git"
```

Where *<machine>* is the name used as Yocto machine. E.g. *s32g399aevb3*, and *<kernel_version>* is the major version of your built kernel, E.g. *5.15.85*.

3.1.7.2 Customizing, building and deploying kernel using YOCTO

3.1.7.2.1 Patch the kernel

In this example, we create a patch for Linux SAR-ADC driver. We should do the steps as below:

1. Go to the SAR-ADC driver folder:

```
cd tmp/work/<machine>-fsl-linux/linux-s32/<kernel_version>-r0/git
```

   and edit the source file drivers/iio/adc/s32cc_adc.c

2. Add source that you changed before to git

```
git add drivers/iio/adc/s32cc_adc.c
git commit -s -m <title of your commit>
```

3. Create the patch

```
git format-patch -1
```

   Check in *<builddirectory>/tmp/work/<machine>-fsl-linux/linux-s32/<kernel_version>-r0/git* folder, there is a patch file created, e.g. *0001-test-sar-adc-repackage.patch*

4. Copy the patch file to recipes-kernel/linux/linux-s32 folder
5. Edit *linux-s32_<kernel_version>.bb* file in recipes-kernel/linux to include the patch file, e.g.

```
SRC_URI += "file://0001-test-sar-adc-repackage.patch"
```

6. Return to the build directory and run command as below to clean the previous sstate:

```
source ./SOURCE_THIS
bitbake virtual/kernel -c cleansstate
```

7. Re-build and deploy linux

```
bitbake virtual/kernel
```

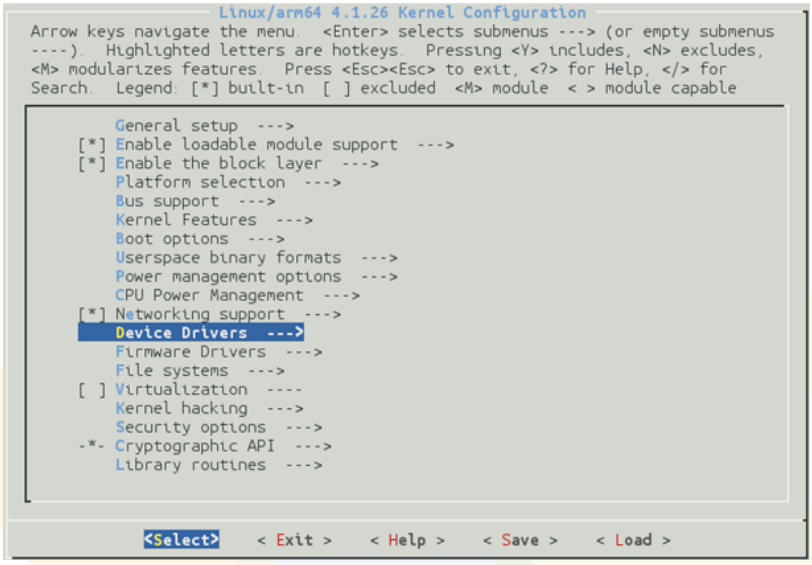
Where *<kernel_version>* is the major version of your built kernel, E.g. *5.15*.

3.1.7.2.2 Invoke menuconfig to set up a specific configuration for Linux

This chapter contains instructions for modifying a particular kernel compile-time option, using the Yocto build framework.

1. From the build folder, run the following command:

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



The user can then navigate the configuration options exposed in the `menuconfig` interface by using the arrow keys, and select/unselect individual configs using the 'y' or 'n' keys, or Space to toggle selection. The user can also search for a specific config option using the slash '/' key from anywhere in `menuconfig`. This opens a search dialog and the user can enter a (sub)string or regexp and jump to the listed findings by pressing the associated numbers: (1), (2),... The user has the option to save or discard the changes before exiting the `menuconfig` interface.

2. Rebuild kernel

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

3. Deploy the images

```
bitbake virtual/kernel
```

After the deploy process is finished, check in the folder, the latest Linux Image is deployed.

3.1.7.2.3 Build standalone kernel modules

To build a standalone kernel module, the following environment variables should be set in order to use the toolchain and the kernel build folder:

```
export ARCH=arm64
export CROSS_COMPILE=<builddir>/tmp/work/<machine>-fsl-linux/linux-s32/<kernel_version>-r0/\
recipe-sysroot-native/usr/bin/aarch64-fsl-linux/aarch64-fsl-linux-
export <Your_kernel_build_dir>=<builddir>/tmp/work/<machine>-fsl-linux/linux-s32/<kernel_version>-r0/build
```

Note 1: <Your_kernel_build_dir> should be the variable used in the module's Makefile, passed as argument to the make invocation via the "-C" parameter. For example, if the module's Makefile invokes the kernel's top Makefile using a similar command:

```
make -C ${KDIR} M=${PWD} modules
```

then, <Your_kernel_build_dir> should be "KDIR", and should contain the path to the kernel build folder, as mentioned above.
Note 2: <machine> is one of the supported machines listed above.
Note 3: *<kernel_version>* is the major version of your built kernel, E.g. *5.15*.

Then build the kernel module invoking "make".

To simplify the procedure to export the toolchain and kernel build folders, the following script can be used to compute these variables automatically:

```
#!/bin/sh

if test $# -lt 1 ;then
  YOCTO_BUILD=`pwd`
else
  YOCTO_BUILD=$1
fi

BB_SOURCE="$YOCTO_BUILD/SOURCE_THIS"

if test -f $BB_SOURCE
then
  . ${BB_SOURCE}
  KERNEL_SETTINGS=$(bitbake -e virtual/kernel | \
  grep "^STAGING_BINDIR_TOOLCHAIN\|^TARGET_PREFIX=\|^KBUILD_OUTPUT=\|export ARCH=")
else
  echo "Usage : $0 <yocto_build_dir>"
  echo "\n\tyocto_build_dir:  The path of Yocto build"
```

```
  echo "\tExample:/build_<machine>"
  echo "\nERROR:$YOCTO_BUILD isn't an yocto build folder !"
  exit 1
fi

get_env_var()
{
  echo $KERNEL_SETTINGS | sed -E "s/.*$1=\"([^\"\]+)\".*/\1/"
}

STAGING_BINDIR_TOOLCHAIN=$(get_env_var "STAGING_BINDIR_TOOLCHAIN")
TARGET_PREFIX=$(get_env_var "TARGET_PREFIX")
KDIR=$(get_env_var "KBUILD_OUTPUT")
ARCH=$(get_env_var "ARCH")

echo KDIR=$KDIR
echo ARCH=$ARCH
echo CROSS_COMPILE=$STAGING_BINDIR_TOOLCHAIN/$TARGET_PREFIX
```

3.1.7.2.4 Building with a 5.10.158 kernel

From BSP 36.0_cd, the default version of kernel is v5.15.85. In case of need, the kernel can be changed to 5.10.158 by adding the following line into the *conf/local.conf* file of your 'build_<machine>' directory:

```
PREFERRED_VERSION_linux-s32 = "5.10.158"
```

Afterwards, any build that uses linux-s32 will use the 5.10.158 one.

3.1.7.3 Customizing, building and deploying U-Boot using YOCTO

3.1.7.3.1 Invoke menuconfig to set up a specific configuration for U-Boot

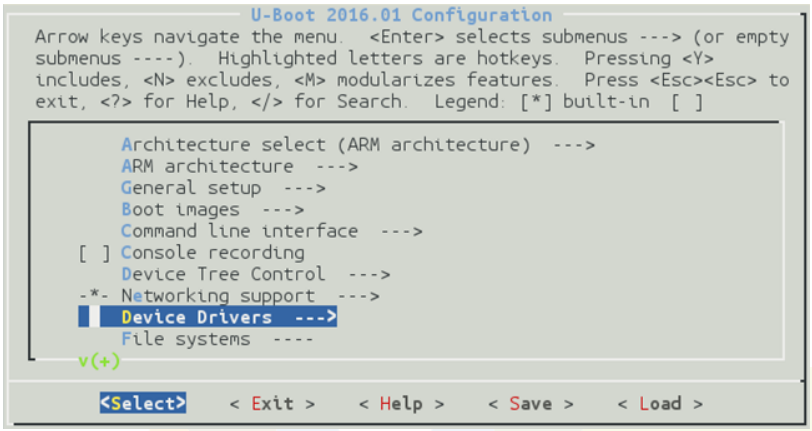
1. From the build folder, run the following command:

```
bitbake u-boot -c configure
bitbake u-boot -c devshell
```

The following commands must be executed in the newly opened window:

```
cd ../build/<defconfig>
make menuconfig
exit
```

where <*defconfig*> is the selected configuration, e.g. *s32g3xxaevb3_defconfig*. For the complete list of configurations for each board refer to section"Building the U-Boot bootloader"



The `menuconfig` interface for U-Boot is the same as the one for the Linux kernel (see Invoke menuconfig to set up a specific configuration for Linux).

2. Rebuild U-Boot

```
bitbake -c compile -f u-boot
```

3. Deploy U-Boot

```
bitbake u-boot
```

After the deploy process is finished, check in the folder, the U-Boot binaries are deployed. For example: `u-boot-s32g399aevb3.s32` `u-boot.s32` `u-boot-s32g399aevb3-2020.04-r0.s32`

3.1.7.4 Applying custom patches in Yocto

There are certain patches not included by default for various reasons, but which may be useful in certain use-cases. Here are the steps required for including such a patch into a Yocto build.

1. Create a .bbappend file for the appropriate recipe:

```
mkdir -p sources/meta-alb/recipes-patched/<package>
touch sources/meta-alb/recipes-patched/<package>/<package_recipe_name>.bbappend
```

More information about .bbappend files can be found at: https://www.yoctoproject.org/docs/4.0.5/dev-manual/dev-manual.html#using-bbappend-files

2. Identify the required .patch file, and add a reference to it in the .bbappend:

```
SRC_URI += "<remote_package_location>/<desired-patch-file>.patch"
```

More information about patching recipes can be found at: https://www.yoctoproject.org/docs/4.0.5/dev-manual/dev-manual.html#new-recipe-patching-code.

3. Rebuild and redeploy

Here is an example:

```
mkdir -p sources/meta-alb/recipes-patched/<package>
touch sources/meta-alb/recipes-patched/<package>/<package>_git.bbappend
echo "SRC_URI += \" <remote_package_location>/<package_patch>.patch\"" >> sources/meta-alb/\
recipes-patched/<package>/<package>_git.bbappend
```

3.1.7.5 Configuration files, logs and other useful items from the Yocto file structure

S32 configuration files are stored in the conf/machine folder, you can find details here.

Yocto saves the output log from bitbake in *<builddirectory>/tmp/log/cooker*, kept under *<builddirectory>/tmp/*. When an error occurs, the exact information listed in the log file is also printed on the console output.

At the following address you can find some useful commands when working with Yocto: https://community.nxp.com/docs/DOC-94953

3.1.7.6 Customize Image Size of Yocto Build

This chapter contains information related to customizing the image size of a Yocto build.

1. How to change the size of boot partition (FAT)

   In `conf/local.conf` set the `BOOT_SPACE` to desired value.

   For example: Set boot partition size to 64MB

```
BOOT_SPACE = "65536"
```

2. How to change the size of the extra free space in the root file system partition (EXT3)

   The IMAGE_ROOTFS_EXTRA_SPACE variable should be set to the desired amount of free space in Kbytes in the root file system partition.

   Example: Allow 1GB of free space.

```
IMAGE_ROOTFS_EXTRA_SPACE = "1048576"
```

   The variable can be set in conf/local.conf or directly in the recipe of the image.

3.1.7.7 Building Linux BSP Yocto images offline

This chapter contains information relating to building Linux BSP images offline, following the steps below:

1. Fetch all necessary packages for the image target e.g: fsl-image-auto. The following operation should be executed with network access.

```
bitbake fsl-image-auto --runall=do_fetch
```

2. Add the following lines in conf/local.conf file from the build directory.

```
BB_NO_NETWORK = "1"
BB_SRCREV_POLICY = "cache"
```

   BB_SRCREV_POLICY variable is used to instruct the fetcher to retain the value the system previously obtained rather than querying the source control system each time.

3. The fsl-image-auto Yocto image can be built without network access.

```
bitbake fsl-image-auto -c clean && bitbake fsl-image-auto
```

3.2 Manually building Linux BSP components

3.2.1 Obtaining a GCC toolchain for ARM 64-bit

This Linux BSP has been built and tested using GCC 10.2 toolchain.
The link for GCC 10.2.0 toolchain, as delivered by Arm at:

https://developer.arm.com/-/media/Files/downloads/gnu-a/10.2-2020.11/binrel/gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu.tar.xz?revision=972019b5-912f-4ae6-864a-f61f570e2e7e&la=en&hash=B8618949E6095C

87E4C9FFA1648CAA67D4997D88

Once you have downloaded the toolchain package, in order to install it, you just need to untar it in a directory of your choice.

3.2.2 Setting up and building U-Boot bootloader

3.2.2.1 Downloading the U-Boot bootloader source code

There are two ways of obtaining the source for this component, each described below. Choose the one which is appropriate for your situation.

1. Cloning the GIT repository:
   In a Linux terminal window, type in the following commands

   ```
   git clone https://github.com/nxp-auto-linux/u-boot
   cd u-boot
   git checkout bsp36.0_cd-2020.04
   ls
   ```

   The contents of the U-Boot source code should appear here.

2. If you already have a clone of the repository, you can run the following commands in the root directory of the existing repository:

   ```
   git fetch origin bsp36.0_cd-2020.04
   git checkout bsp36.0_cd-2020.04
   ls
   ```

   The contents of the U-Boot source code should appear here.

3.2.2.2 Building the U-Boot bootloader

In the same Linux terminal window as above, type in the following commands:

```
make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- <board>_defconfig
make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu-
```

These commands should generate the U-Boot image with Program data (*u-boot-nodtb.bin*).

Note:
The values of <board>are given as in the following.

- `s32g399aevb3` using Memory Card, board is `s32g3xxaevb3`
- `s32g399aevb3` using QSPI, board is `s32g3xxaevb3_qspi`
- `s32g398aevb3` using Memory Card, board is `s32g3xxaevb3`

- s32g398aevb3 using QSPI, board is s32g3xxaevb3_qspi
- s32g379aevb3 using Memory Card, board is s32g3xxaevb3
- s32g379aevb3 using QSPI, board is s32g3xxaevb3_qspi
- s32g378aevb3 using Memory Card, board is s32g3xxaevb3
- s32g378aevb3 using QSPI, board is s32g3xxaevb3_qspi
- s32g399ardb3 using Memory Card, board is s32g399ardb3
- s32g399ardb3 using QSPI, board is s32g399ardb3_qspi

3.2.3 Setting up and building the Linux kernel

3.2.3.1 Downloading the Linux kernel source code

There are two ways of obtaining the source for this component, each described below. Choose the one which is appropriate for your situation.

1. Cloning the GIT repository:
   In a Linux terminal window, type in the following commands

   ```
   git clone https://github.com/nxp-auto-linux/linux
   cd linux
   git checkout bsp36.0_cd-5.15.85-rt
   ls
   ```

   The contents of the linux kernel source code should appear here.

2. If you already have a clone of the repository, you can run the following commands in the root directory of the existing repository:

   ```
   git fetch origin bsp36.0_cd-5.15.85-rt
   git checkout bsp36.0_cd-5.15.85-rt
   ls
   ```

   The contents of the Linux kernel source code should appear here.

3.2.3.2 Building the Linux Kernel

In the same Linux terminal window as above, type in the following commands:

```
make ARCH=arm64 CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- \
    <soc_name>_defconfig
make ARCH=arm64 CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu-
```

For s32g399aevb3, s32g398aevb3, s32g379aevb3, s32g378aevb3, s32g399ardb3: the defconfig is s32cc_defconfig.

This command should generate the kernel binary (*Image*) in arch/arm64/boot and the board device tree blobs (e.g. *s32g3xxa-evb3.dtb*...) in arch/arm64/boot/dts/freescale/.

3.2.4 Building the ARM Trusted Firmware

Prerequisites:

- (Optional) If choosing to include OP-TEE in the TF-A boot flow, one must also build optee_os according to Building the optee_os component of OP-TEE. The optee_os build will be excuted as part of a boot stage in TF-A.
- Install libssl-dev and openssl for headers required by *fiptool*. This is a one-time operation. On Ubuntu-based hosts, you should:

```
sudo apt-get install libssl-dev openssl
```

  (note that you might need to *sudo apt-get update* first, in order to update the host's package list).
- Check your host's dtc version. If you have a dtc version at least 1.4.6, you can skip the apt-get install step following this one:

```
dtc --version
```

- If your dtc is older or you do not have it installed, install/upgrade to dtc version 1.4.6 or above - this is a one-time operation:

```
sudo apt-get install device-tree-compiler
```

Before building the TF-A, please note: The TF-A blob being built is a FIP image that contains both the TF-A boot stages and the U-Boot binary. Because of that, the TF-A build process depends on U-Boot being available and optionally optee_os. To that end, the **BL33** parameter to the make command-line must be the path to the *u-boot-nodtb.bin* located in the directory where U-Boot has been built as part of the prerequisites. **BL32** must be the path to the *tee-header_v2.bin* and **BL32_EXTRA1** must be the path to the *tee-pager_v2.bin*, both located in the directory where optee_os has been built as part of the prerequisites.

Follow these steps to build the TF-A:

1. Obtain the arm-trusted-firmware:

```
git clone https://github.com/nxp-auto-linux/arm-trusted-firmware
cd arm-trusted-firmware
git checkout bsp36.0_cd-2.5
```

   If you already have a clone of the repository, you can run the following commands in the root directory of the existing repository:

```
git fetch origin bsp36.0_cd-2.5
git checkout bsp36.0_cd-2.5
```

2. Build the arm-trusted-firmware FIP image. Choose **only one** of the following two options:

(a) Build the default TF-A FIP image:

```
make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- \
    ARCH=aarch64 PLAT=<plat> BL33=<path-to-u-boot-nodtb.bin>
```

(b) Build the OP-TEE-enabled TF-A FIP image:

```
make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- \
    ARCH=aarch64 PLAT=<plat> BL33=<path-to-u-boot-nodtb.bin> \
    BL32=<path-to-tee-header_v2.bin>  \
    BL32_EXTRA1=<path-to-tee-pager_v2.bin> SPD=opteed
```

3. Deploy *build/<plat>/release/fip.s32* to the sdcard. See also this section for indications on partitioning the SD card.

```
# Skip the partition table from the SD card and copy the rest of the fip image
# (TF-A and U-Boot).
# Presumably, if you are doing these steps, you are using a pre-partitioned SD card
# on which you are only replacing the TF-A and/or U-Boot image which you have manually built.
sudo dd if=<path/to/fip.s32> of=/dev/<sdcard_dev> skip=512 seek=512 \
        iflag=skip_bytes oflag=seek_bytes conv=fsync,notrunc
```

Depending on the target board, <plat> must be replaced with:

- `s32g399aevb3`: <plat> is s32g3xxaevb
- `s32g398aevb3`: <plat> is s32g3xxaevb
- `s32g379aevb3`: <plat> is s32g3xxaevb
- `s32g378aevb3`: <plat> is s32g3xxaevb
- `s32g399ardb3`: <plat> is s32g3xxaevb

Note: The above build steps assume that the boot medium is an SD card. However, the same TF-A FIP image can boot from eMMC as well. For QSPI boot, the BL33 parameter should contain the path to a U-Boot binary built specifically for QSPI compatibility. Please consult the Setting up and building U-Boot bootloader section for instructions on building an U-Boot binary for QSPI boot. For more information on how images can be written on any boot medium, please see the How to boot section.

3.2.4.1 Custom Build Parameters

TF-A provides build time parameters that allow TF-A customization for different booting flow scenarios. For example, if the boot target is M7, TF-A can be configured for a different memory layout. TF-A can read FIP image from a configurable location.

- For QSPI boot, configure TF-A to read the FIP image from a defined QSPI offset. Add **FIP_QSPI_OFFSET=<QSPI offset>** make parameter, as in the following example:

```
make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- \
    ARCH=aarch64 PLAT=<plat> BL33=<path-to-u-boot-nodtb.bin> \
    FIP_QSPI_OFFSET=0x5400
```

- For SD/eMMC boot, configure TF-A to read the FIP image from a defined SD/eMMC offset. Add **FIP_MMC_OFFSET=<MMC offset>** make parameter, as in the following example:

```
make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- \
    ARCH=aarch64 PLAT=<plat> BL33=<path-to-u-boot-nodtb.bin> \
    FIP_MMC_OFFSET=0x5400
```

- Configure TF-A to read the FIP image from a defined memory address instead of reading it from the boot source storage (QSPI or MMC).
  It is assumed the FIP image is copied to the specified memory address from outside TF-A. For example, the FIP image can be copied to specified SRAM address from the M7 bootloader configured as boot target, before starting BL2. Add **FIP_MEMORY_OFFSET=<memory address>** make parameter, as in the following example:

```
make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- \
    ARCH=aarch64 PLAT=<plat> BL33=<path-to-u-boot-nodtb.bin> \
    FIP_MEMORY_OFFSET=0x34520000
```

  It is important to note that the resulting ATF binary will be larger due to the padding added to the BL2 stage to allow in-place execution.

  Additional details on the BL2 internal layout can be obtained by customizing and running the tools/memory/print_memory_map.py helper script from the TF-A source directory.

Depending on the target board, <plat> must be replaced with:

- `s32g399aevb3`: <plat> is s32g3xxaevb
- `s32g398aevb3`: <plat> is s32g3xxaevb
- `s32g379aevb3`: <plat> is s32g3xxaevb
- `s32g378aevb3`: <plat> is s32g3xxaevb
- `s32g399ardb3`: <plat> is s32g3xxaevb

3.2.5 Building the optee_os component of OP-TEE

This section is detailing the required steps for successfully building the optee_os component of the OP-TEE project. The other three components - optee_client, optee_examples and optee_test - are automatically built and deployed through the Yocto build (see Building OP-TEE-Enabled Image) because they should not be modified. For more details on each component, please refer to the OP-TEE chapter.

Prerequisites:

- Certain configurations must be enabled for both U-Boot and Linux so that OP-TEE can communicate with Linux and allow user space applications to call for TAs execution.
  While **building U-Boot**, one must make sure that **CONFIG_OPTEE=y** is set. **Building Linux** requires both **CONFIG_OPTEE=y** and **CONFIG_OPTEE_SHM_NUM_PRIV_PAGES=256** to be set. For more information about obtaining and building any of the two, please see Setting up and building U-Boot bootloader and Setting up and building the Linux kernel.

- Two Python3 packages are mandatory for building OP-TEE. For easier package management and installation, the pip3 package management system for Python3 software should be installed. On Ubuntu-based hosts it is not present by default, so, if it has not been previously installed, run the following command:

  ```
  sudo apt-get install python3-pip
  ```

  Then, the required Python3 packages - pyelftools and pycryptodomex - can be installed using pip3:

  ```
  pip3 install pyelftools pycryptodomex
  ```

Follow these steps to build optee_os:

1. Obtain optee_os:

   ```
   git clone https://github.com/nxp-auto-linux/optee_os
   cd optee_os
   git checkout bsp36.0_cd-3.18
   ```

2. Build optee_os:

   ```
   make CROSS_COMPILE64=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- ARCH=arm  \
       PLATFORM=s32 PLATFORM_FLAVOR=s32g3
   ```

The above steps should create the following two binary images: *out/arm-plat-s32/core/tee-header_v2.bin* and *out/arm-plat-s32/core/tee-pager_v2.bin*. Since OP-TEE can only boot with TF-A support, these two binaries **must be included in the FIP image generated when building the TF-A**. Please refer to the Building the ARM Trusted Firmware section for instructions on building an OP-TEE-enabled FIP image: all the prerequisites must be met and the build steps followed thoroughly.

4 How to boot

This chapter describes the steps to prepare an SD/MMC card and boot up for S32G3 EVB and S32G399A RDB3 boards. The boot modes of above boards are controlled by the boot configuration DIP switches and jumpers on the board. The following sections list basic boot setup configurations only.

The space between 0x0 and 0x1d_3000 is occupied by some or all of the following components: IVT, QSPI Parameters, DCD, HSE_FW, SYS_IMG, Application Boot Code Header, TF-A FIP image. The actual layout is determined at boot time and can be obtained from the arm-trusted-firmware.

Example output:

IVT:	Offset: 0x1000	Size: 0x100
AppBootCode Header:	Offset: 0x1200	Size: 0x40
U-Boot/FIP:	Offset: 0x1240	Size: 0x3d400
U-Boot Environment:	Offset: 0x1e0000	Size: 0x2000

Note 1: For SD/eMMC the partitioned space begins at 0x1d_3000. For more information see the support in classes/image_types_fsl_sdcard.bbclass. For QSPI, the region after 0x1d_3000 is organized as follows:

Kernel	[0x01f_0000 : 0x0ef_ffff]
FDT	[0x0ff_0000 : 0x10e_ffff]
Ramdisk	[0x10f_0000 : 0x2ff_ffff]
PFE Firmware	[0x300_0000 :]

For more information see the support in classes/image_types_fsl_flashimage.bbclass and include/configs/s32-cc.h

4.1 Writing images

4.1.1 Setup an SD/MMC card

Information found here describes the steps to prepare an SD/MMC card to boot up for S32G3 EVB and S32G399A RDB3 boards. The supported SD classes for SD are 4 and 10.

4.1.1.1 Requirements

An SD/MMC card reader is required. It will be used to transfer the boot loader and kernel images to initialize the partition table and copy the root file system. To simplify the instructions, it is assumed that a 4GB SD/MMC card is used.

Any Linux distribution can be used for the following procedure.

The Linux kernel running on the Linux host will assign a device node to the SD/MMC card reader. The kernel might decide the device node name or udev rules might be used. In the following instructions, it is assumed that udev is not used.

To identify the device node assigned to the SD/MMC card, enter the command:

```
cat /proc/partitions
major minor #blocks name

   8    0   85647168   sda
   8    1   82628608   sda1
   8    2          1   sda2
   8    5    3015680   sda5
  11    0      58258   sr0
   8   16    7707648   sdb
   8   17    7703552   sdb1
```

Note: The “lsblk“ command may also be useful for finding the device node.
In this example it is assumed that the device assigned is /dev/sdb. Set DEVSD accordingly, e.g.

```
export DEVSD=/dev/sdb
```

Note:
Make sure the device node is correct for the SD/MMC card! Otherwise, it may damage your operating system or data on your computer!

There are two ways to prepare a bootable SD Card:

1. To use Comprehensive Yocto Image
   After successfully building Yocto, look for build result in
   <builddirectory>/tmp/deploy/images/<board_name>

   The .sdcard format creates an image with all necessary partitions and loads the bootloader, kernel and rootfs to this image.

   Ensure that any partitions on the card are properly unmounted before writing the card image, or you may end up with a corrupted card image. Also ensure to properly `sync` the filesystem before ejecting the card to ensure all data has been written.

   You can also low-level copy the data on this file to the SD card device using `dd` as in the following command example:

   ```
   sudo dd if=./fsl-image-base-s32g399aevb3.sdcard of=${DEVSD} bs=1M && sync
   ```

   If you want to verify the SD card content against the original image, remove the SD card from the slot, insert the SD card once more and use cmp command in the following way:

   ```
   sudo cmp ./fsl-image-base-s32g399aevb3.sdcard ${DEVSD}
   ```

   Note:

   Yocto supports the generation of both uImage and Image kernel binary formats, but only the ‘Image’ format works with the current configuration of U-Boot. It is not expected that uImage support will be restored in the future.

2. To manually copy BSP binaries into the SD Card, please follow the below chapters.

4.1.1.2 Creation of the partitions on the SD card

Our SD Card will be split into two main partitions:

- One for the kernel and board device tree blob;
- Another for the root file system that will run on the board.

An additional, unpartitioned/unformatted space will also be created at the beginning of the SD card to hold the U-Boot and TF-A bootloaders.

To create the partitions, enter the following command:

```
sudo fdisk ${DEVSD}
```

Then type the following parameters to the fdisk console (each followed by <ENTER>):

```
d [repeat this until no partition is reported by the 'p' command]
n [create a new partition]
p [create a primary partition]
1 [the first partition]
8192 [create a partition that starts at offset 8192; this leaves 4MB of unpartitioned
space for the bootloader]
+255M [size of the actual partition = 255 MB]

[It may be that fdisk detects a previous filesystem signature and asks for confirmation
of erase: "Partition #1 contains a vfat signature. Do you want to remove the signature?
[Y]es/[N]o:". Answer Y in this case.]

p [list the parameters of the previously created partition, look at the number of the End
sector: 530431]
n [create a new partition]
p [create a primary partition]
2 [the second partition]
530432 [Start sector of the new partition; partition will be adjacent to the previous
one]
<enter>[using the default value will create a partition that uses the remaining space on
the card]

t [set partition type]
1 [partition #1]
c [FAT32]

t [set partition type]
2 [partition #2]
83 [Linux]
```

```
w [ this writes the partition table to the medium and fdisk exits]
```

Note: The fdisk version used for the above commands was 2.31.1 from Ubuntu package util-linux.

These newly created partitions need to be formatted to be usable. Firstly, remove and reinsert SD card into its slot, then run the following commands to format the partitions:

```
sudo mkfs.vfat -n boot ${DEVSD}1
sudo mkfs.ext3 -L rootfs ${DEVSD}2
```

4.1.1.3 Copying the BSP binaries onto the SD Card

Note: Before proceeding, ensure you have downloaded the SD card binary from flexnet account.

If you remove the SD card from its slot after it has been formatted and you reinsert it, you should see two partitions in your file explorer program (in this example it is assumed that your user is 'public'):

- boot (in */media/public/boot*)
- rootfs (in */media/public/rootfs*)

If the partitions are not mounted automatically, use the following steps to mount them. If you do not know the device node assigned to the SD, see the "Requirements" section.

```
export SD_MOUNT_POINT=/media/public
sudo mount ${DEVSD}1 ${SD_MOUNT_POINT}/boot
sudo mount ${DEVSD}2 ${SD_MOUNT_POINT}/rootfs
```

Run the following commands to copy the necessary binaries onto SD Card:

```
export SD_MOUNT_POINT=/media/public
cd <builddirectory>/tmp/deploy/images/<board_name>
sudo dd if=fip.s32 of=${DEVSD} conv=notrunc,fsync seek=512 skip=512 oflag=seek_bytes iflag=skip_bytes
sudo cp Image ${SD_MOUNT_POINT}/boot/
sudo cp <dtb_file> ${SD_MOUNT_POINT}/boot/<dtb_file>
sudo tar xf <rootfs> -C ${SD_MOUNT_POINT}/rootfs
sync
```

Unmount the file systems and eject cleanly the SD card.

Note: If targeting the S32G3 EVB board, copy *s32g3xxa-evb3.dtb* to *${SD_MOUNT_POINT}/s32g3xxa-evb3.dtb*.
Note: For manual component building, the component names and paths should be updated accordingly to the corresponding build path. For example, for Linux use dtb file arch/arm64/boot/dts/freescale/s32g3xxa-evb3.dtb and image file arch/arm64/boot/Image

4.1.2 Setup QSPI memory

4.1.2.1 Generating the .flashimage file for booting from QSPI

The simplest method of generating a flash bootable image from yocto is with

```
bitbake fsl-image-flash
```

which will produce in *build_*/tmp/deploy/images/<machine>* a file named *fsl-image-flash-<machine>.flashimage*, an equivalent in terms of packages with fsl-image-base, but ready to be written to QSPI memory (see chapter "Booting from QSPI" about burning a file into flash).

By default, `fsl-image-flash` contains the root file system packages defined in `fsl-image-base`. To use root file system packages from a different, custom image, the following changes should be done: in file conf/machine/include/s32-gen1-board.inc, update variable `FLASHIMAGE_ROOTFS` to the desired custom image. For example:

```
        FLASHIMAGE_ROOTFS = "<custom image name>"
```

All that is left is to write the .flashimage file to the beginning of QSPI memory. More details can be found in chapter "Booting from QSPI".

Note:

- The custom image should be small enough to fit into the allocated memory map. If the size of the image is too high, an error is thrown when building *fsl-image-flash*.
- By default, *fsl-image-flash* will boot using a RAM-based filesystem, without a backing store. Thus, all altered or generated files will return to their original state after a board reset.

4.1.3 Setup eMMC

Information found here describes the steps to prepare the eMMC memory to be used for Linux BSP

Build U-Boot

```
export CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu-
make -j9
```

Method 1: Write using Flash Tool

Connect the board to your workstation using an USB cable.

Load Flash Tool on chip

```
./bin/S32FlashTool -t ./targets/S32G3xxx.bin -a ./flash/EMMC.bin -i uart -p /dev/ttyUSB0
```

Write U-Boot image

```
./bin/S32FlashTool -t ./targets/S32G3xxx.bin  -fwrite -addr 0 -f ./fip.s32 -i uart -p /dev/ttyUSB0
```

Method 2: Write from QSPI/SD

Load the desired image into DDR (from QSPI/SD or any other sources). It can be fip.s32, containing ATF and U-Boot images, or SD card image. The below examples will load fip.s32 binary or an eMMC image from FAT32 partition and write it to eMMC.

ATF and U-Boot update

```
=> setenv image fip.s32
=> run loadimage
```

SD card image update

```
=> setenv image fsl-image-base-s32g399aevb3.sdcard
=> setenv loadaddr 0x86000000
=> run loadimage
```

Jump over the MBR during ATF and U-Boot update

```
=> setexpr loadaddr ${loadaddr} + 0x1000
```

Calculate number of eMMC blocks needed for the loaded image.

```
=> setexpr n_blocks ${filesize} / 0x200
=> setexpr n_blocks ${n_blocks} - 7
```

```
=> setexpr n_blocks ${filesize} / 0x200
=> setexpr n_blocks ${n_blocks} + 1
```

Make eMMC available by adapting board jumpers / switches
On S32G3 EVB board this can be done by placing J50 on position 2-3
On S32G399A RDB3 board this can be done by setting SW3 to OFF

Write the image from DDR to eMMC starting with IVT.

```
=> mmc rescan
=> mmc write ${loadaddr} 8 ${n_blocks}
```

Write the image from DDR to eMMC starting with block 0x0.

```
=> mmc rescan
=> mmc write ${loadaddr} 0 ${n_blocks}
```

Power off the board and adapt switches configuration for eMMC boot. For more information on this please refer to section “Booting from eMMC”

4.2 Board set-up and boot

After boot set-up is complete using one of the methods presented in previous section, follow these steps to set-up the target board:

- Power off the Target board system if the power is already on.
- Connect the Target board to the host machine via the serial port with an RS-232 cable
- Connect the Target board to the network via an Ethernet port on the board, if needed.
- Start the serial console tool (such as Minicom or TeraTerm) on the host system.
- Verify that all switches and jumpers are setup correctly as described in section "Boards".
- Power on the board.
- The sections below describe steps to boot Linux with corresponding deployment method.

4.2.1 Booting from SD

The procedure for running Linux is the following:

1. Prepare the SD card setup using one of the methods presented in section "Setup a SD/MMC card".
2. Insert the SD Card into SD Card socket.
3. Verify that all switches and jumpers are setup correctly for booting from SD as described in section "Boards".
4. Reset the board.
5. The U-Boot should start to print on the console a count down, and then it will start the Linux kernel. If any key is pressed during the U-Boot countdown, boot process will stop at U-Boot prompt. Run the following commands to boot Linux:
   - Load the image:

     ```
     => run loadimage
     ```

   - Load the dtb:

     ```
     => run loadfdt
     ```

   - Set the bootargs in order to use the rootfs from SD:

     ```
     => run mmcargs
     ```

   - Start the booting process:

     ```
     => ${boot_mtd} ${loadaddr} - ${fdt_addr}
     ```

6. After that you should be prompted with a Linux console.

How to reset U-Boot environment

When reusing an SD card with a new version of U-Boot, if `saveenv` was previously run in the U-Boot console, the new default environment might not be visible. To clear environment variables previously saved with U-Boot, the following steps should be performed:

1. Print one, several or all variables of the U-Boot environment

```
printenv <variable>
printenv <variable_1> <variable_2> ...
printenv
```

2. Reset the environment to the default values of the build

```
env default -fa
```

3. Save the currently loaded environment to persistent storage (SD/HyperFlash)

```
saveenv
```

4.2.2 Booting from eMMC

The procedure for running Linux is the following:

1. Prepare the eMMC setup using one of the methods presented in section “eMMC setup”..
2. Verify that all switches and jumpers are setup correctly for booting from eMMC as described in section “Boards”.
3. Reset the board.
4. The U-Boot should start to print on the console a count down, and then it will start the Linux kernel.
5. After that you should be prompted with a Linux console.

Note: Sometimes characters are lost when copy-paste. To fix that, a transmit delay of 1 ms/char should be set in the terminal client.

4.2.3 Booting from TFTP

To load Image and dtb files using TFTP it required to setup a TFTP server and to do the TFTP client settings in U-Boot.

TFTP server

During testing phase, we validated the TFTP booting using an Ubuntu-20.04 LTS workstation which had installed the xinet, tftp and tftpd services:

```
apt-get install xinetd tftp tftpd
```

The instructions for the configuration of TFTP server are outside the scope of this document. For more details please consult the tutorials which are available on the Internet.

U-Boot setup

- Power-on the board and stop at U-Boot prompt
- Setup the interface IP and TFTP server IP:

```
=> set ipaddr <yourIPv4>
=> set serverip <serverIPv4>
```

- Test the connectivity with server:

```
=> ping <serverIPv4>
```

Note: If the antivirus or firewall settings are too restrictive, it's likely to drop the U-Boot ARP packets considering them an ARP injection attack.

- Download Image and dtb files

```
=> tftp ${loadaddr} Image
=> tftp ${fdt_addr} <dtb_file>
```

Note: The *Image* and <*dtb_file*> files should be available in the root directory of TFTP server.

- Set the bootargs in order to use the rootfs from SD.

```
=> run mmcargs
```

- Start the booting process

```
=> booti ${loadaddr} - ${fdt_addr}
```

4.2.4 Booting from NFS (root filesystem)

After U-Boot has been loaded (either from QSPI or from SD), the boot process can be continued with files obtained via Ethernet, but the U-Boot configuration must be changed in advance to prepare for this boot flow.

Note: The Ubuntu images on all supported machine does not currently support NFS booting.
Note: This boot flow is currently validated on:

- S32G3 EVB

The default U-Boot configuration for the S32G3 EVB board contains some predefined commands which can be used to continue the boot process using Ethernet. The default configuration will attempt to access the Linux kernel, dtb and root filesystem from the SD card, so it is necessary to change the configuration to prevent the default boot flow. The following steps should be performed:

- Configure another system to be both a TFTP and NFS server which can serve files for S32G3 EVB. The TFTP server software will serve the Image and the dtb files, while the NFS server software will serve the S32G3 EVB root filesystem built with Yocto.
  - make sure the NFS share is exported from */tftpboot/rfs* on the server and it can be mounted, accessed and modified from another linux system
- Stop the U-Boot counting down in order to have access to console
- Change the `image`, `fdt_file` variables to match the files on server
- If necessary, change the `ipaddr`, `serverip` and/or `netmask` configurations to match the used set up.
- After all configurations are correct, from the U-Boot prompt run `run nfsboot` to continue the boot process.

4.2.5 Booting from QSPI

There are at least two ways of writing images into the QSPI:

- Using **S32FlashTool** from **S32 Design Studio**
- Through U-Boot or Linux by loading all the binaries over an Ethernet connection or from SD

The following steps will only cover the second option. Please refer to the **S32 Design Studio** documentation for more details about **S32FlashTool** programmer.

Perform the following steps in the U-Boot or Linux console after you have prepared all the binaries involved in a Linux boot (FIP image, Linux Kernel image, DTB and filesystem) on SD card or on a TFTP server.

NOTE: MX25UW51245G supports full flash erase command which is faster than erasing the flash memory sector by sector. One could use the following command to erase the entire memory before writing the images instead of erasing individual blocks, by using the QSPI partition which maps to the entire flash memory:

In U-Boot

```
=> run flashbootargs
=> sf probe 6:0
=> sf erase 0 ${flashsize}
```

In Linux

```
root@s32g399aevb3:~# flash_erase /dev/mtd0 0 0
```

For the following steps, in case of using Linux, it is assumed that all binaries are in the current working directory. Furthermore, under '/opt' there is 's32cc_qspi_write.sh' bash script available which can be used to write files in the QSPI Flash Memory.

1. Optional: Inspect QSPI Flash Memory partitions
   In U-Boot

```
=> mtd list
SF: Detected mx25uw51245g with page size 256 Bytes, erase size 64 KiB, total 64 MiB
List of MTD devices:
* nor0
  - device: mx25uw51245g@0
  - parent: spi@40134000
  - driver: spi_flash_std
  - type: NOR flash
  - block size: 0x10000 bytes
  - min I/O: 0x1 bytes
  - 0x000000000000-0x000004000000 : "nor0"
          - 0x000000000000-0x000004000000 : "Flash-Image"
          - 0x000000000000-0x0000001e0000 : "FIP"
          - 0x0000001e0000-0x0000001f0000 : "U-Boot-Env"
          - 0x0000001f0000-0x000000ff0000 : "Kernel"
          - 0x000000ff0000-0x0000010f0000 : "DTB"
          - 0x0000010f0000-0x000003000000 : "Rootfs"
          - 0x000003000000-0x000004000000 : "PFE-Firmware"
```

   In Linux

```
root@s32g399aevb3:~# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 04000000 00010000 "Flash-Image"
mtd1: 001e0000 00010000 "FIP"
mtd2: 00010000 00010000 "U-Boot-Env"
mtd3: 00e00000 00010000 "Kernel"
mtd4: 00100000 00010000 "DTB"
mtd5: 01f10000 00010000 "Rootfs"
mtd6: 01000000 00010000 "PFE-Firmware"
```

2. Write complete Flash Image generated by Yocto

   When writing the entire fsl-image-flash image in U-Boot, we change the RAM load address in order to not overlap with the reserved memory ranges.
   Also, the U-Boot steps below assume that all binaries are stored on SD, on the first partition (FAT). The `loadimage` command has to be adapted if the images are placed on a different partition or it can be replaced with `loadtftpimage` in case the TFTP transfer is preferred.

   In U-Boot

```
=> run flashbootargs
=> sf probe 6:0
=> setenv image fsl-image-flash-<machine>.flashimage
=> setenv loadaddr 0x86000000
=> run loadimage
=> sf erase 0 ${flashsize}
=> mtd write Flash-Image ${loadaddr}
```

In Linux

```
root@s32g399aevb3:~# /opt/s32cc_qspi_write.sh Flash-Image \
fsl-image-flash-<machine>.flashimage
```

3. Power off the board and set the jumpers for QSPI boot. These settings are covered by "Boards" section.

   NOTE: Instead of following steps 2 and 3, another option is to individually write each image as exemplified below. Each image such as: FIP, Linux Kernel, DTB, Ramdisk is stored in QSPI using a MTD partition and the following steps will update the partitions content one by one.

4. Update FIP partition containing ATF and U-Boot images

In U-Boot

```
=> setenv image fip.s32-qspi
=> run loadimage
=> mtd erase FIP
=> mtd write FIP ${loadaddr}
```

In Linux

```
root@s32g399aevb3:~# /opt/s32cc_qspi_write.sh \
FIP fip.s32-qspi
```

5. Update partition containing Linux Kernel image

In U-Boot

```
=> setenv image Image
=> run loadimage
=> mtd erase Kernel
=> mtd write Kernel ${loadaddr}
```

In Linux

```
root@s32g399aevb3:~# /opt/s32cc_qspi_write.sh \
Kernel Image
```

6. Update DTB partition containing Linux device tree

In U-Boot

```
=> setenv image ${fdt_file}
=> run loadimage
=> mtd erase DTB
=> mtd write DTB ${loadaddr}
```

In Linux

```
root@s32g399aevb3:~# /opt/s32cc_qspi_write.sh \
DTB <fdt_file>
```

7. Update Rootfs partition containing Linux filesystem

In U-Boot

```
=> setenv image fsl-image-base-<machine>.cpio.gz.u-boot
=> run loadimage
=> mtd erase Rootfs
=> mtd write Rootfs ${loadaddr}
```

In Linux

```
root@s32g399aevb3:~# /opt/s32cc_qspi_write.sh \
Rootfs fsl-image-base-<machine>.rootfs.cpio.gz.u-boot
```

8. Update PFE-Firmware partition containing PFE FW

In U-Boot

```
=> setenv image s32g_pfe_class.fw
=> run loadimage
=> mtd erase PFE-Firmware
=> mtd write PFE-Firmware ${loadaddr}
```

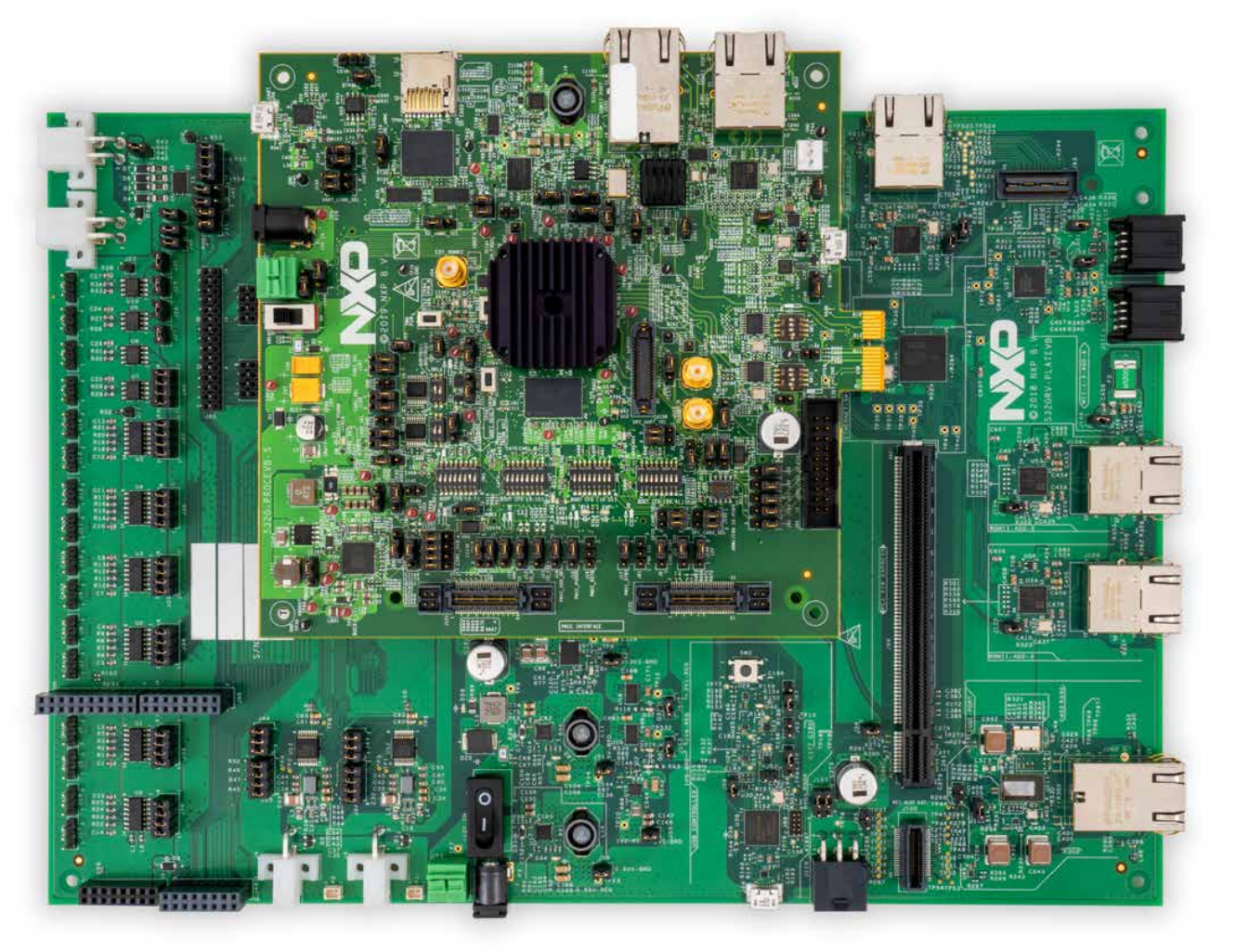
In Linux

```
root@s32g399aevb3:~# /opt/s32cc_qspi_write.sh \
PFE-Firmware s32g_pfe_class.fw
```

9. Power off the board and set the jumpers for QSPI boot. These settings are covered by “Boards” section.

4.3 Boards

4.3.1 S32G3 EVB



This chapter describes the steps to configure the boot jumpers and switch configuration on the board S32G3 EVB SCH-32170 REV B1.

With respect to power source selection, in case S32G-PROCEVB-S is used without S32G-PLATEVB-S the default position of J96 (2-3) is enough. However, shall the user use S32G-PROCEVB-S in conjunction with S32G-PLATEVB-S, J96 needs to be set to position 1-2. The effect of this jumper position change is that the power supply of the S32G-PROCEVB-S is now taken from S32G-PLATEVB-S (instead from the daughter card, as in the default position).

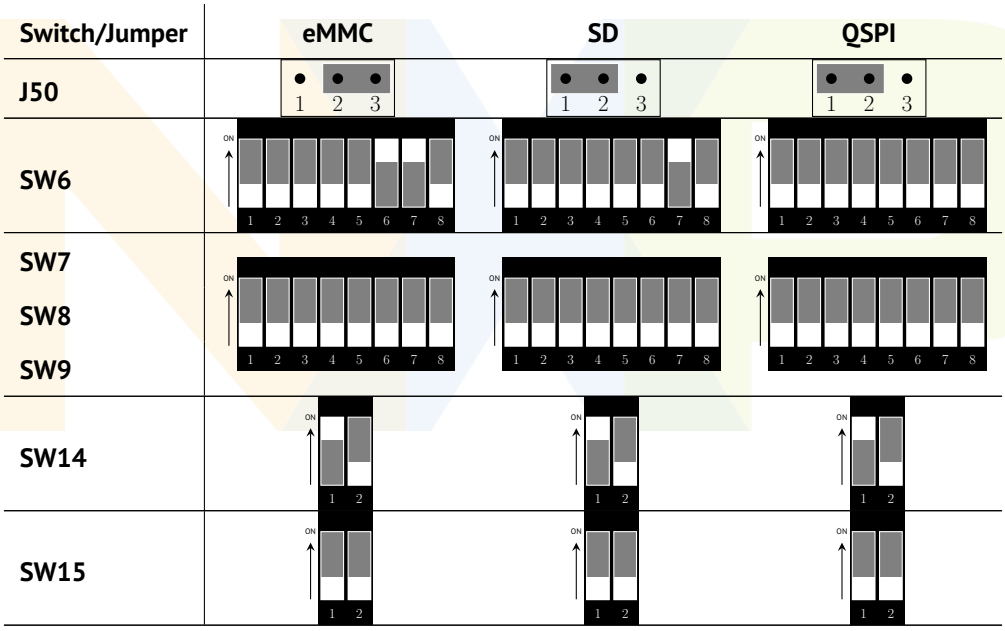
The current pinmuxing configuration for UART using LF0 requires that jumpers J124 on the S32G-PROCEVB-S board be in the 1-3, 2-4 position, which is their default position but is different from BSP22.5.

Jumpers J159 should be in the 1-3, 2-4 position (default configuration).

The S32G3 EVB board supports booting from parallel RCON, as well as serial RCON. In the parallel RCON boot scenario, switches SW6 - SW9, which are connected to RCON's 32 general-purpose I/O pins (RCON[0] to RCON[31]), select the boot configuration. When booting from serial RCON, the boot configuration is read from an external serial EEPROM that communicates with the chip via I^2C. Selecting between serial/parallel RCON boot modes is done by toggling RCON[8] i.e. RCON[8] = 0b when booting from parallel RCON, whereas RCON[8] = 1b when booting from serial RCON. Once the RCON boot mode is selected, one should choose the boot medium (SD, eMMC or QSPI) by setting up RCON[0:7].

4.3.1.1 Boot from parallel RCON

In this mode switches SW6 - SW9 select the values of RCON[0:31], each switch representing 1 byte in the RCON[0:31], i.e. SW6 represents RCON[0:7], SW7 represents RCON[8:15] and so on. SW6 selects the boot medium. These are the settings of the jumpers and switches which select SD, eMMC or QSPI booting:



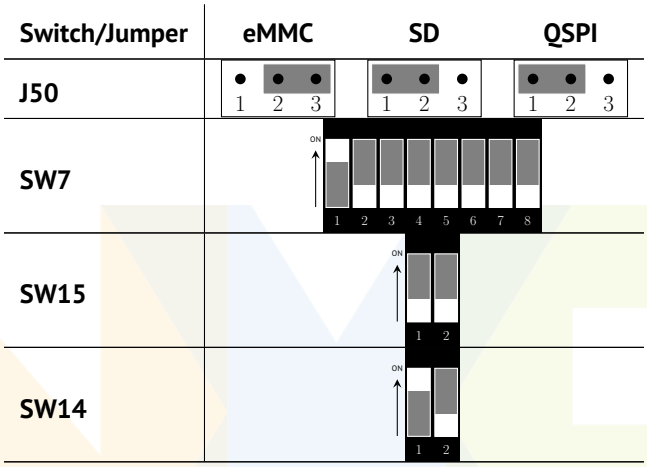
The rest of the switches and jumpers are set to default position.

4.3.1.2 Boot from serial RCON

In this mode, the first 4 bytes residing in EEPROM represent the boot configuration. The first byte represents RCON[0:7], the second byte represents RCON[8:15] and so on. The first byte selects the boot medium. The EEPROM memory should have the following layout, according to the desired boot medium (SD, eMMC or QSPI):

EEPROM Offset	eMMC	SD	QSPI
0h	0x60	0x40	0x0
1h	0x1		
2h	0x7		
3h	0x0		

Jumpers and switches should have the following values:



The values of SW6, SW8 and SW9 are irrelevant. The rest of the switches and jumpers are set to default position.

One possible way of populating the first 4 bytes of the I^{2}C-connected EEPROM with an SD boot configuration, for example, is by issuing the following commands in U-Boot:

```
=> i2c dev 0
=> i2c mw 0x50 0.1 0x40
=> i2c mw 0x50 1.1 0x01
=> i2c mw 0x50 2.1 0x07
=> i2c mw 0x50 3.1 0x00
```

Power off the board, make sure SW7 is set to 0x1 and then power on the board. The newly written boot configuration is fetched from EEPROM no matter what the values of switches SW6, SW8 and SW9 are. Please note that the second command in the above sequence sets the boot medium according to the first table in the paragraph "Boot from serial RCON".

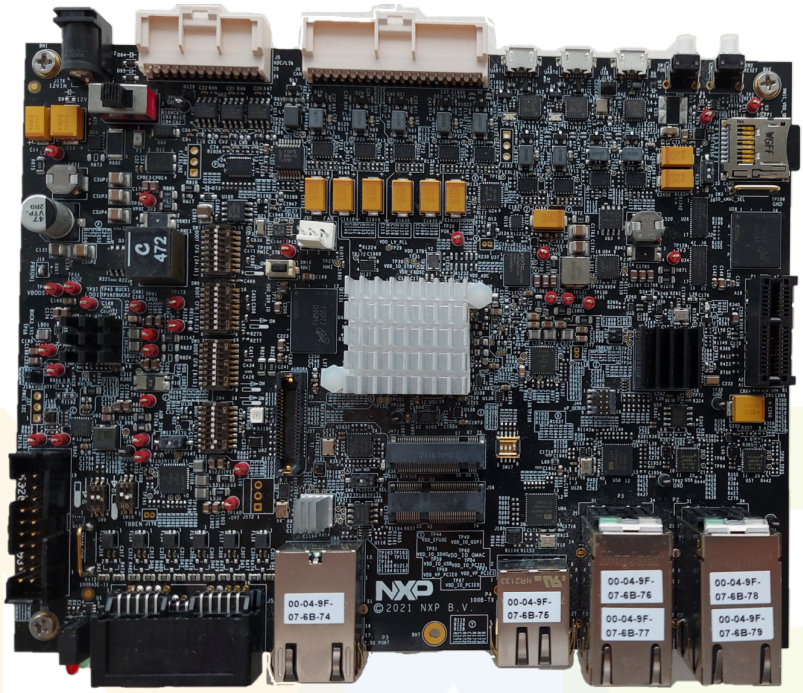
4.3.2 S32G3 EVB3

In case of S32G3 EVB3 SCH-50784 REV A, the same steps to configure the boot jumpers and switch configuration apply as on the board S32G3 EVB SCH-32170 REV B1. Same applies to power source selection, current pinmuxing

configuration for UART using LF0 and booting from parallel RCON.

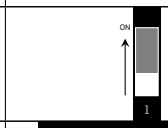
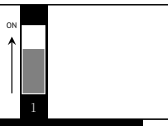
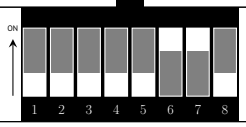
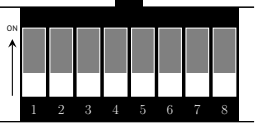
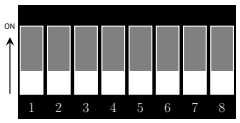
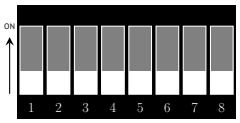
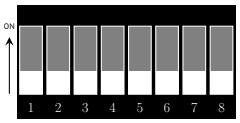
In addition, S32G3 EVB3 SCH-32170 REV B1 provides UART1 serial interface.

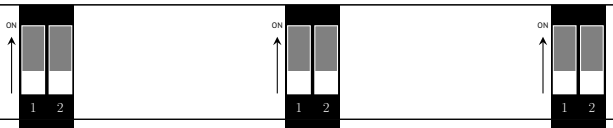
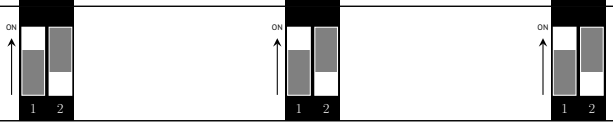
4.3.3 S32G399A RDB3



This chapter describes the switch configuration on the board S32G399A RDB3 SCH-53060 REV E.

These are the settings of the jumpers and DIP switches which select SD, eMMC or QSPI booting:

Switch/Jumper	eMMC	SD	QSPI
SW3			
SW4			
SW5 SW6			

SW7	
SW9	
SW10	

The rest of the switches and jumpers are set to default position.

5 SMP Boot

The S32G3 BootROM can start either the M7_0 or the A53_0 core. This document assumes the boot core is A53. The A53_0 core is thus considered the "primary" (or "master") core (or CPU), while cores 1, 2, 3, 4, 5, 6 and 7 are considered "secondary" (or "slave") cores (or CPUs).

Secondary cores with Linux SMP boot on ARMv8 are managed through PSCI calls [1].

An ARM Trusted Firmware (ATF, or TF-A) [2] implementation is responsible for implementing PSCI functions related to CPU management, in particular SMP boot.

5.1 TF-A SMP Boot

Secondary cores are managed by the Trusted Firmware. At cold boot, the secondary cores are in reset. During kernel boot, Linux uses `PSCI_CPU_ON` calls to enable individual cores. PSCI calls trap at Exception Level 3 (EL3) and are executed by the Trusted Firmware's Secure Monitor component.

Note: U-Boot does not touch the secondary cores. It is the TF-A Secure Monitor that hands them over to Linux - see section "TF-A SMP Boot".

5.2 Linux Kernel SMP

In the Linux device-tree source file, the `cpus` parent node contains children for each `cpu<n>`. The bindings for the `cpu` nodes are specified in the kernel documentation: Documentation/arm64/booting.rst. The device-tree included in the Linux BSP sources specifies the `psci` as `enable-method` by default.

The following is an excerpt of the Linux device-tree `cpus` node:

```
cpus {
        #address-cells = <2>;
        #size-cells = <0>;

        cpu0: cpu@0 {
                device_type = "cpu";
                compatible = "arm,cortex-a53";
                reg = <0x0 0x0>;
                enable-method = "psci";
        };
        [...]
};
```

[1] More details about PSCI can be found in ARM documentation

[2] Please see chapter 25 for more details about ARM Trusted Firmware

6 Lockstep Operation

Lockstep mode is an operation mode of the chip that runs the same set of operations on two cores or clusters. The chip supports lockstep operation for both Cortex-M7 and Cortex-A53 cores. In this document, lockstep operation refers to Cortex-A53 lockstep operation – as Cortex-A53 is the core where Linux BSP runs.

The chip supports two modes of operation for Cortex-A53 clusters.

- Decoupled mode (or performance mode) allows the two Cortex-A53 clusters to operate in an independent manner. All the eight Cortex-A53 are available for the software applications.
- Lockstep mode: the two Cortex-A53 clusters operate in delayed lockstep configuration, that is:
  - core0 of cluster0 is in lockstep with core0 of cluster1
  - core1 of cluster0 is in lockstep with core1 of cluster1
  - core2 of cluster0 is in lockstep with core2 of cluster1
  - core3 of cluster0 is in lockstep with core3 of cluster1

  Only cluster0 is available in the system for software applications. Cluster1 is not visible but works as a checker or redundant module.

The default mode of operation for the Cortex-A53 clusters is the decoupled mode. The operation mode (decoupled or lockstep mode) can be changed only when A53 reset domain is in the reset state. This means that the operation mode cannot be set from software running on Cortex-A53. This can only be done from software running on Cortex-M7 while A53 reset domain is in the reset state.

In the Linux BSP default booting flow, the boot target is Cortex-A53, as set in the Image Vector Table (IVT) Boot configuration word. In this case, Cortex-A53 core0 in cluster0 is the first enabled core that can further start any other core. This flow enables the default mode of operation, that is, decoupled mode.

Lockstep mode of operation can be enabled by changing the default booting flow. In this case, the IVT Boot configuration word should be changed to set the Cortex-M7 as the boot target. The software running on Cortex-M7 enables the Cortex-A53 lockstep mode by setting the `A53_GPR06[CA53_LOCKSTEP_EN]` register bit while the Cortex-A53 is still in reset state. After lockstep is enabled, the Cortex-A53 core is also enabled and it starts in lockstep mode.

Linux BSP software running on Cortex-A53 can operate in lockstep mode with no supplementary configuration or software changes. However, the Cortex-M7 software and IVT Boot configuration that enable lockstep mode need to be updated. To demonstrate lockstep operation mode, Linux BSP provides a booting flow example with Cortex-M7 set as boot target. The M7 booting flow can be enabled as described in section Building Images with M7 as Boot Target

6.1 Enablement flow

Once Cortex-A53 core0 (core0 from cluster0) has started, the software (TF-A or U-Boot) running on Cortex-A53 core0 dynamically handles both decoupled and lockstep modes of operation. The mode of operation is detected at run-time such that there is no configuration for Cortex-A53 software to enable or disable the lockstep mode. The same binary images (for TF-A, U-Boot and Linux) handle lockstep or decoupled modes. The Cortex-A53 software performs the following actions specific to lockstep mode:

- Reset AArch64 Generic Timer registers. The complete list of Generic Timer registers is specified in the SoC's Reference Manual. Reset means that, for each of the mentioned registers, the value 0x0 is written to the register before performing any read access to it. The software running on a Cortex-A53 core can only reset its core registers. For example, software running on Cortex-A53 core0 can only reset core0 registers, but not core1 registers. Therefore, the reset operation is performed executing the same procedure from each of the available Cortex-A53 cores. The purpose of the reset is to guarantee the same registers values for the lockstep cores. In lockstep mode only cluster0 is visible for the Cortex-A53 software such that the software performs the reset only for core0 and core1 of the cluster0. It is the SoC's lockstep mechanisms that automatically performs the reset for the redundant cores. Therefore, a software reset to core0 cluster0 registers means that hardware performs the same write operation to the core0 cluster1 registers. Similarly, a software reset to core1 cluster0 registers triggers the hardware to perform the reset for core1 cluster1 registers.
- Reset general purpose registers. The same procedure described for AArch64 Generic Timer but applied to the generic purpose registers: X0, X1, …., X30. For general purpose registers, the reset value may be different than 0x0 as needed in the assembly code. However, the goal is to always write a value before performing a read access to these registers.
- For lockstep mode only cluster0 is visible to Cortex-A53 software, therefore cluster1 should not be accessed. The following actions are performed:
  - NCore is configured only for cluster0. Refer to the SoC Reference Manual for more information about NCore.
  - Secondary cores are enabled as described in section SMP Boot but only core1 from cluster0 is available as secondary core.

6.2 Errors handling

In lockstep mode, the equivalent operation of the redundant resources (GIC, cluster) is supervised by dedicated comparators, part of the Redundancy Control and Checker Unit (RCCU). Any operation deviations between the active and the redundant resource causes the Fault Collection and Control Unit (FCCU) to be notified.

The Linux FCCU driver allows updating the list of Non-Critical Faults (NCF) handled in FCCU and selecting one of the supported NCF actions using device tree. The NCF lockstep errors and their actions can be configured as part of the FCCU driver configuration in device tree. One example for device tree configuration for lockstep error handling in FCC is the following:

```
        fccu: fccu@4030C000 {
                [...]
                /* A53 NCF fault list */
                nxp,ncf_fault_list = <0 10 35 36 37 38>;
                nxp,ncf_actions = <S32_FCCU_REACTION_ALARM
                    S32_FCCU_REACTION_ALARM
                    S32_FCCU_REACTION_NONE
                    S32_FCCU_REACTION_NONE
                    S32_FCCU_REACTION_NONE
                    S32_FCCU_REACTION_NONE>;
                status = "disabled";
        };
```

In this example, lockstep NCF handling is enabled by adding 0 (A53 lock-step RCCUs combination of all redundant

signals from A53 RCCUs) and 10 (A53 RCCU lock-step error A53 clusters) to FCCU's NCF list. The NCF's action can be selected from the following options:

- No action on detecting the NCF. The status bits are cleared at driver initialization
- Alarm: an alarm is generated and handled in the FCCU's driver interrupt handler. The alarm information is printed in the kernel's log (dmesg).
- Functional reset is triggered when a NCF error is detected
- Non Maskable Interrupt is triggered when a NCF error is detected
- EOUT signal is asserted when a NCF error is detected

7 Ethernet Information

7.1 Prerequisites

PFE Linux driver relies on PFE firmware, which is a separate NXP product available through https://www.nxp.com/.

The compatible version number can be found in PFE Linux driver Release Notes. The required firmware has to be downloaded from NXP Software Center.

For more information, and to find out how to obtain PFE Firmware, please contact NXP marketing department.

7.2 Ethernet Hardware Support on S32G3

7.2.1 Ethernet support on S32G3 SoC

The SoC contains two Ethernet controllers:

1. **GMAC**
2. **PFE** - an Ethernet accelerator with three EMACs (PFE_EMAC_0, PFE_EMAC_1 and PFE_EMAC_2)

The controllers can be programmed to use MII/RMII/RGMII and SGMII interface modes supporting 10/100/1000/2500 Mbit/s speeds.

NOTE: The SoC contains three RGMII blocks which can be connected to the GMAC/PFE EMAC ports in a limited number of combinations.

NOTE: The SGMII block shares the SerDes subsystem with PCIe. S32G3 contains two SerDes blocks with two lanes each. By default, SerDes_0 is used by PCIe and SerDes_1 by SGMII.

7.2.2 SerDes configuration for SGMII in U-Boot

To use SGMII interface it is necessary to configure SerDes. S32G supports two SerDes instances and each instance has multiple configurations available. As the SerDes configured with SGMII can be also shared with PCIe additional care is necessary with configuration. The configuration itself is performed via ATF device tree and U-Boot `hwconfig` environment variable.

To enable SerDes, the `serdes` nodes in the ATF device tree have to be enabled (one per SerDes) in fdts/s32cc.dtsi.

```
    serdes0: serdes@40480000 {
                #phy-cells = <3>;
                compatible = "nxp,s32cc-serdes";
                reg = <0x00 0x40480000 0x0 0x04000  /* serdes registers */
                      0x00 0x40400000 0x0 0x80000>; /* dbi registers */
                reg-names = "ss", "dbi";
                #address-cells = <3>;
                #size-cells = <2>;
                device_type = "pci-generic";
                device_id = <0>;
                num-lanes = <2>; /* supports max 2 lanes */
                clocks = <&clks S32GEN1_SCMI_CLK_SERDES_REF>;
        };
```

The `hwconfig` can be displayed via `printenv hwconfig`.
The format for this variable is (all on one row):

```
=> setenv hwconfig "serdes0:<ss_settings_0>[;pcie0:<pci_settings_0>][;xpcs0_j:<xpcs_settings_0_j>];
                    serdes1:<ss_settings_1>[;pcie1:<pci_settings_1>][;xpcs1_j:<xpcs_settings_1_j>]"
```

Where:

- `<ss_settings_i>` is :

  `mode=<ss_mode_i>,clock=<clock_i>[,skip=<value_i>][,fmhz=<fmhz_i>]`

  `i` is the index of the SerDes module (0 or 1), and the valid values for SerDes options `mode`, `clock`, `fmhz` are defined below:

mode	clock	fmhz	description
'pcie'	'ext' (Gen3) or 'int' (Gen2)	(unset)	Mode 0, Serdes 0 or 1
'pcie&xpcs0' (1G)	'ext' (Gen3) or 'int' (Gen2)	'100'	Mode 1, Serdes 1 only (PFE0) Note: GMAC0 as SGMII is not supported
'pcie&xpcs1' (1G)	'ext' (Gen3) or 'int' (Gen2)	'100'	Mode 2, Serdes 0 or 1
'pcie&xpcs1' (Gen2/2.5G)	'ext' or 'int'	'100'	Mode 2, Serdes 0 or 1
'xpcs0&xpcs1' (1G)	'ext' or 'int'	'100' or '125'	Mode 3, SerDes 0 or 1
'xpcs0&xpcs1' (2.5G)	'ext' or 'int'	'125'	Mode 4, SerDes 1 only

Table 9: `serdesi` options values

- `<xpcs_settings_i_j>` is :

  `speed=<10M|100M|1G|2G5>`

  `i` is the index of the SerDes module (0 or 1), `j` is the index of the XPCS interface (0 or 1).

Note: XPCS0 is GMAC0 on SerDes0 and PFE0 on SerDes1.
XPCS1 is PFE2 on SerDes0 and PFE1 on SerDes1.
GMAC0 can't be used as SGMII for now.

Note: Options `xpcsi_j` are required only if the string `xpcsj` can be found in the value of the option `mode` for the corresponding SerDes*i* module.

Please refer to the "PCIe in U-Boot" for more detailed information on SerDes configuration.

7.2.2.1 Example configurations

SerDes modules can operate in several modes. Below are some configurations showing some of these modes.
Note: The boards have to support the configuration to test it.

Board Name	Configuration	Command

S32G399A RDB3	SerDes0 (Mode 0): PCIe Gen3 X2
	SerDes1 (Mode 2):
	PCIe Gen3 X1 on lane0,
	PFE_EMAC_1 1G on lane1
	Switches: SW17 all ON, SW8 OFF
	Note: SJA1110 with 1G FW
		setenv hwconfig "serdes0:mode=pcie,clock=ext;pcie0:mode=rc;
		serdes1:mode=pcie&xpcs1,clock=ext,fmhz=100;
		pcie1:mode=rc;xpcs1_1:speed=1G"
		setenv pfeng_mode enable,none,sgmii,rgmii

S32G399A RDB3	SerDes0 (Mode 0): PCIe Gen3 X2
	SerDes1 (Mode 2):
	PCIe **Gen2** X1 on lane0,
	PFE_EMAC_1 2.5G on lane1
	Switches: SW17 all ON, SW8 OFF
		setenv hwconfig "serdes0:mode=pcie,clock=ext;pcie0:mode=rc;
		serdes1:mode=pcie&xpcs1,clock=int,fmhz=100;
		pcie1:mode=rc;xpcs1_1:speed=2G5"
		setenv pfeng_mode enable,none,sgmii,rgmii

S32G3 EVB
S32G3 EVB3
	SerDes0 (Mode 0): PCIe Gen3 X2
	SerDes1 (Mode 3):
	PFE_EMAC_0 1G on lane0,
	PFE_EMAC_1 1G on lane1
		setenv hwconfig "serdes0:mode=pcie,clock=ext;pcie0:mode=rc;
		serdes1:mode=xpcs0&xpcs1,clock=ext,fmhz=125;
		xpcs1_0:speed=1G;xpcs1_1:speed=1G"
		setenv pfeng_mode enable,sgmii,sgmii,rgmii

S32G399A RDB3	SerDes0 (Mode 0): PCIe Gen3 X2 SerDes1 (Mode 4): PFE_EMAC_0 2.5G on lane0, PFE_EMAC_1 2.5G on lane1 Switches: SW17 all ON, SW8 ON	setenv hwconfig "serdes0:mode=pcie,clock=ext;pcie0:mode=rc serdes1:mode=xpcs0&xpcs1,clock=ext,fmhz=125; xpcs1_0:speed=2G5;xpcs1_1:speed=2G5" setenv pfeng_mode enable,sgmii,sgmii,rgmii

S32G399A RDB3	SerDes0 (Mode 2):
	PCIe Gen3 X1 on lane0,
	PFE_EMAC_2 1G on lane1
	SerDes1 (Mode 1):
	PCIe Gen3 X1 on lane0,
	PFE_EMAC_1 1G on lane1
	Switches: SW17 all OFF, SW8 OFF
	Note: SJA1110 with 1G FW
		setenv hwconfig "serdes0:mode=pcie&xpcs1,clock=ext,fmhz=100;
		pcie0:mode=rc;xpcs0_1:speed=1G;
		serdes1:mode=pcie&xpcs1,clock=ext,fmhz=100;pcie:mode=rc;
		xpcs1_1:speed=1G"
		setenv pfeng_mode enable,none,sgmii,sgmii

S32G399A RDB3	SerDes0 (Mode 2): lane0 N/A,
	PFE_EMAC_2 2.5G on lane1
	SerDes1 (Mode 4): lane0 N/A,
	PFE_EMAC_1 2.5G on lane1
	Switches: SW17 all OFF, SW8 ON
		setenv hwconfig "serdes0:mode=pcie&xpcs1,clock=ext,fmhz=100;
		pcie0:mode=rc;xpcs0_1:speed=2G5;
		serdes1:mode=xpcs0&xpcs1,clock=ext,fmhz=125;
		xpcs1_0:speed=2G5;xpcs1_1:speed=2G5"
		setenv pfeng_mode enable,none,sgmii,sgmii

S32G399A RDB3	SerDes0 (Mode 3):
	GMAC0 1G on lane0,
	PFE_EMAC_2 1G on lane
	SerDes1 (Mode 3): lane0 N/A,
	PFE_EMAC_1 1G on lane1
	Switches: SW17 all OFF, SW8 ON
	Note: SJA1110 with 1G FW
		setenv hwconfig "serdes0:mode=xpcs0&xpcs1,clock=ext,fmhz=100;
		xpcs0_0:speed=1G;xpcs0_1:speed=1G;
		serdes1:mode=xpcs0&xpcs1,clock=ext,fmhz=125;
		xpcs1_0:speed=1G;xpcs1_1:speed=1G"
		setenv pfeng_mode enable,none,sgmii,sgmii

Table 10: SerDes configuration examples

Note 1: For specific configuration you can check arch/arm/mach-s32/include/s32-cc/serdes_hwconfig.h in U-Boot.

Note 2: For changes required in order to have the modes in the above table working in Linux, please see section Linux Ethernet Support, NOTE-9.

Please refer to the "S32G3Reference Manual" for more detailed information on SerDes configuration.

7.2.3 SerDes configuration for SGMII in Linux

Linux SerDes driver reconfigures SerDes based on device tree. The device tree is automatically updated from U-Boot to correspond to the `hwconfig` variable. The SerDes driver is necessary for allowing suspend/resume of the SerDes.

Example of default node:

```
        serdes0: serdes@40480000 {
                #phy-cells = <3>;
                compatible = "nxp,s32cc-serdes";
                clocks = <&clks S32GEN1_SCMI_CLK_SERDES_AXI>,
                         <&clks S32GEN1_SCMI_CLK_SERDES_AUX>,
                         <&clks S32GEN1_SCMI_CLK_SERDES_APB>,
                         <&clks S32GEN1_SCMI_CLK_SERDES_REF>;
                clock-names = "axi", "aux", "apb", "ref";
                resets = <&reset S32GEN1_SCMI_RST_SERDES0>,
                         <&reset S32GEN1_SCMI_RST_PCIE0>;
                reset-names = "serdes", "pcie";
                fsl,sys-mode = <XPCSX2_MODE>;
                reg = <0x0 0x40480000 0x0 0x108>,
                      <0x0 0x40483008 0x0 0x10>,
                      <0x0 0x40482000 0x0 0x800>,
                      <0x0 0x40482800 0x0 0x800>;
                reg-names = "ss_pcie", "pcie_phy", "xpcs0", "xpcs1";
        };
```

Following values are updated based on `hwconfig` variable in U-Boot:

- SerDes mode (fsl,sys-mode allowed options are PCIE_GEN3X2_MODE, PCIE_XPCS0_MODE, PCIE_XPCS1_MODE, XPCSX2_MODE)
- Clock external/internal frequency (clocks, clock-names)

By default internal reference clock is used. To use external clock, add its phandle to 'clocks' and add "ext" to corresponding index in 'clock-names'.

The SerDes driver allows multiple configuration with SGMII and supports SGMII auto-negotiation. When SGMII AN is enabled, PHY provides speed/link status over SGMII link to MAC. The driver supports only the MAC role in SGMII AN process.

Configure the link management properties for the 'ethernet' device tree node, depending on the PHY's SGMII AN support and MDIO connection, according to the following table:

PHY SGMII AN	MDIO to PHY	device tree properties	Information
disabled	connected	phy-handle = <...>;	SGMII link is statically configured based on PHY auto-negotiation result(clause 28).
enabled	connected	phy-handle = <...>; managed = "in-band-status";	SGMII link is configured based on SGMII AN result. Some statuses from the PHY are also used.
enabled	not connected	managed = "in-band-status";	SGMII link is configured based on SGMII AN result.

disabled	not connected	fixed-link {
		speed = ...;
		}
			SGMII link is statically configured based on fixed-link properties.

NOTE: With incorrect configuration the SGMII link between PHY and SoC SerDes may not work.

7.2.4 Ethernet support on the S32G3 EVB/S32G3 EVB3 board(s)

As a prerequisite for the EVB board revision 'B': erratum E-S32G_EVB-0001 must be applied.
The evaluation board contains two parts:

1. `S32G-PROCEVB-S or S32G-PROCEVB3-S` with SoC and two Ethernet sockets:
   - 2500Base-T with AQR107 (or AQR113 on EVB3) PHY connected on SGMII to the PFE_EMAC_0;
   - 1000Base-T with KSZ9031 PHY connected on RGMII to the GMAC.
2. `S32GRV-VNP-PLATEEVB` with Ethernet switch SJA1105Q and the following sockets:
   - 1000Base-T with KSZ9031 PHY connected to the RGMII PFE_EMAC_1/GMAC, but disabled as the S32G-VNP-PROC-S RGMII is used by default;
   - 2x1000Base-T with KSZ9031 PHYs connected to the SJA1105Q switch;
   - 2x100BASE-T1 with one TJA1102 automotive PHY connected to the SJA1105Q switch;
   - 2500Base-T with AQR107/113 PHY connected on SGMII to the PFE_EMAC_1.

For more detailed information, please refer to the "S32G-VNP-EVB – User Guide" or "S32G-PROCEVB3 – User Guide" .

7.2.5 Ethernet support on the S32G399A RDB3 board

The evaluation board integrates all necessary components in one chassis, including Automotive TSN/AVB SJA1110A SGMII switch. The sockets are:

- 1x2500Base-T with AQR113 PHY connected to PFE_EMAC_1
- 2x1000Base-T with AR8035 PHYs connected to the SJA1110A switch;
- 1000Base-T with AR8035 PHY connected to the RGMII PFE_EMAC_2;
- 1000Base-T with KSZ9031 PHY connected to GMAC;
- 100Base-TX directly connected to the SJA1110 switch;
- 6x100Base-T1 directly connected to the SJA1110 switch;

For more detailed information, please refer to the "S32G-VNP-RDB2/3 Reference Manual".

7.2.6 RMII PHYs support in S32G3 Linux BSP with PFE

RMII on the S32G3 platform is supported, but requires custom HW setup and custom configurations that are described in separate documents.

The following sections briefly describe how RMII PHYs are supported, based on the following configuration:

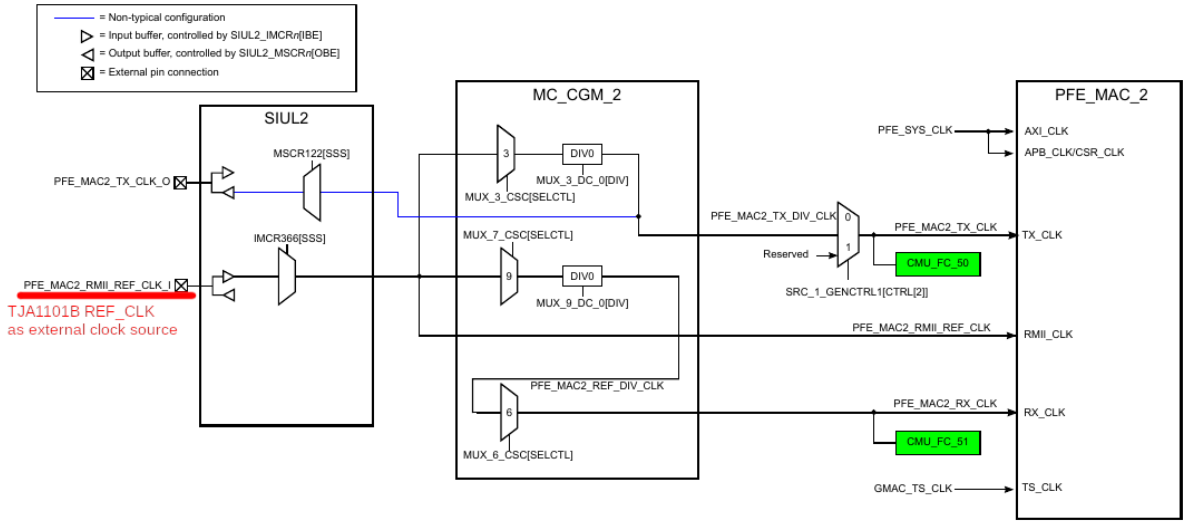
- `S32G-PROCEVB3-S` (SCH-50784)
- `S32GRV-PLATEVB` (SCH-30081)
- `ADTJA1101-RMII` (SCH-30211) - attached in `ETHERNET CARD (B PORT)` of `S32GRV-PLATEVB`

7.2.6.1 Linux device tree

Linux PFE driver device tree configuration: arch/arm64/boot/dts/freescale/s32g3xxa-evb3-rmii.dts
Port mapping is the same as described in Linux Ethernet support, see the *EVB3 default port mapping in Linux* table, with the exception that the PFE_EMAC_2 - `pfe2` is mapped to the RMII interface on `ADTJA1101-RMII` board.

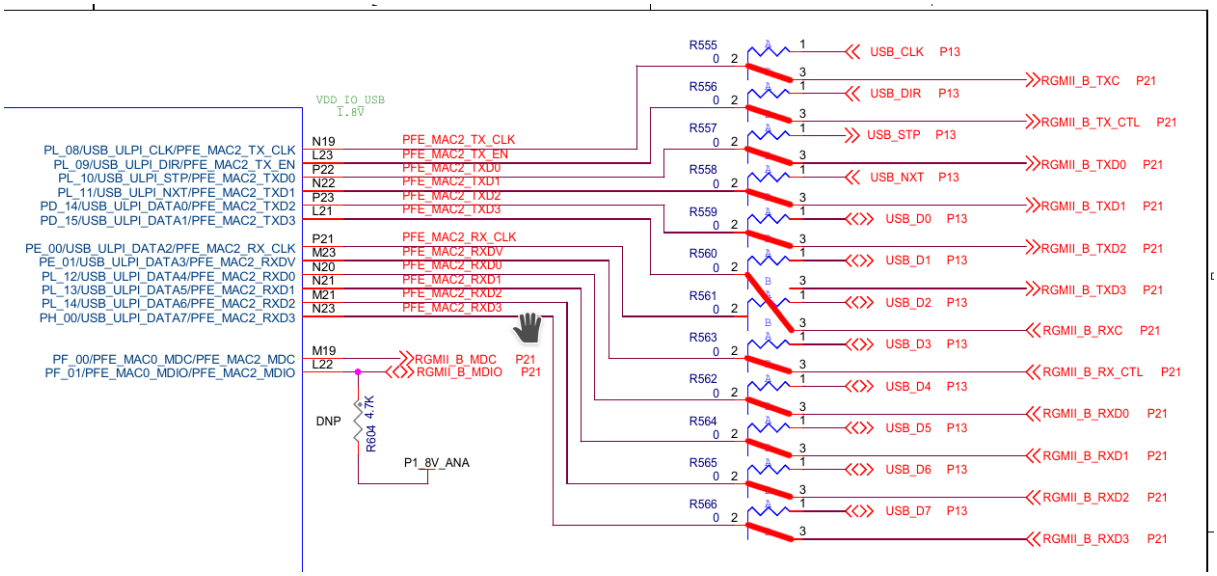
7.2.6.2 Clocking scheme

Typical RMII S32G3 clocking scheme is used, TJA1101B PHY works as external 50MHz clock source for PFE_EMAC_2.



7.2.6.3 S32G-PROCEVB3-S HW modification

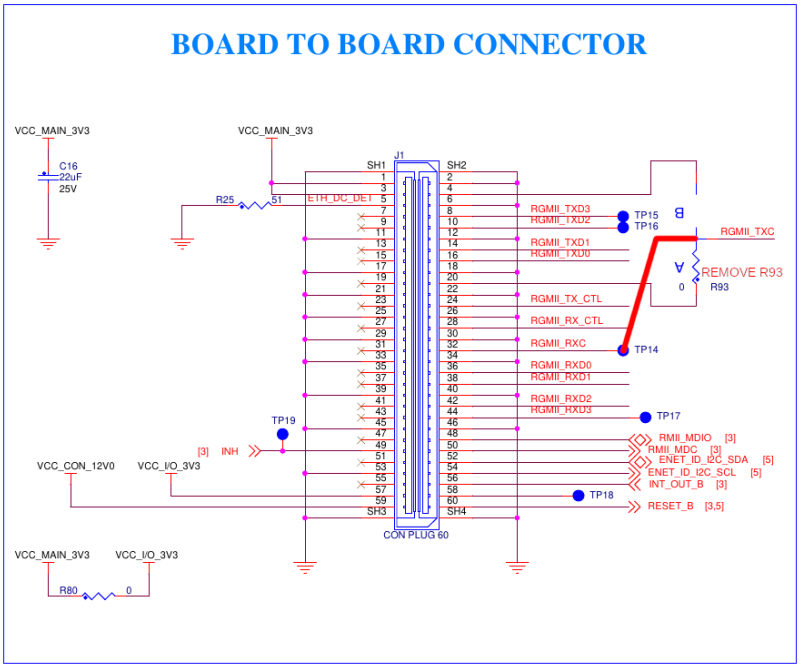
In order to enable `ETHERNET CARD (B PORT)` on `S32GRV-PLATEVB`, a HW modification of `S32G-PROCEVB3-S` must be done.



- Disable USB transceiver `USB83340`, enable `RGMII_B`: Move R555 .. R559 & R562 .. R566 from position `A` to `B`.
- Make S32G3 pin `PD_15` as `PFE_MAC2_RMII_REF_CLK_I` - RMII Reference Clock input: Remove R560 & R561, connect R560 middle pad to R561 `B` pad.

7.2.6.4 ADTJA1101-RMII HW modification

By default `ADTJA1101-RMII REF_CLK` is configured as 50MHz OUTPUT, wired to OUTPUT pin `CARD_RGMII_RXCLK` of U44 (74AVC8T245PW) on `S32GRV-PLATEVB`. An additional HW modification is required, to be able to act `ADTJA1101-RMII` as external 50MHz `REF_CLK` source for S32G3.



Remove R93 on `ADTJA1101-RMII` board, wire `RGMII_TXC` i.e. R93 `B` pad to TP14.

7.3 Ethernet Software support on S32G3

7.3.1 U-Boot Ethernet support

On the S32G3 EVB/S32G3 EVB3 and S32G399A RDB3 targets we have both controllers enabled by default. It can be visible on U-Boot start-up or later by the `dm` command:

```
=> dm uclass
uclass 0: root
- * root_driver @ 9fb2b050, seq 0, (req -1)

uclass 11: simple_bus
uclass 21: eth
- * eth_eqos @ 9fb2b640, seq 0, (req -1)
- * eth_pfeng @ 9fb2b7e0, seq 1, (req -1)
```

In U-Boot, in multi Ethernet configuration, the system provides the environment variable `ethact` which contains the active network controller. It can be checked by `printenv ethact` and changed by `setenv ethact <eth-driver-name>`. For S32G it is `eth_pfeng` or `eth_eqos`.

GMAC in U-Boot: the GMAC driver is using embedded device tree for configuration:
In ATF device tree file fdts/s32cc.dtsi:

```
gmac0: ethernet@4033c000 {
        compatible = "nxp,s32cc-dwmac";
        reg = <0x0 0x4033c000 0x0 0x2000>, /* gmac IP */
              <0x0 0x4007c004 0x0 0x4>;    /* S32 CTRL_STS reg */
        interrupt-parent = <&gic>;
        interrupts = <GIC_SPI 57 IRQ_TYPE_LEVEL_HIGH>;
        interrupt-names = "macirq";
        tx-fifo-depth = <20480>;
        rx-fifo-depth = <20480>;
        status = "disabled";
        phy-names = "gmac_xpcs";
        phys = <&serdes0 PHY_TYPE_XPCS 0 0>;
        dma-coherent;
        clocks = <&clks S32GEN1_SCMI_CLK_GMAC0_AXI>,
                 <&clks S32GEN1_SCMI_CLK_GMAC0_AXI>,
                 <&clks S32GEN1_SCMI_CLK_GMAC0_TX_SGMII>,
                 <&clks S32GEN1_SCMI_CLK_GMAC0_TX_RGMII>,
                 <&clks S32GEN1_SCMI_CLK_GMAC0_TX_RMII>,
                 <&clks S32GEN1_SCMI_CLK_GMAC0_TX_MII>,
                 <&clks S32GEN1_SCMI_CLK_GMAC0_RX_SGMII>,
                 <&clks S32GEN1_SCMI_CLK_GMAC0_RX_RGMII>,
                 <&clks S32GEN1_SCMI_CLK_GMAC0_RX_RMII>,
                 <&clks S32GEN1_SCMI_CLK_GMAC0_RX_MII>,
                 <&clks S32GEN1_SCMI_CLK_GMAC0_TS>;
        clock-names = "stmmaceth", "pclk",
                      "tx_sgmii", "tx_rgmii",
                      "tx_rmii", "tx_mii",
                      "rx_sgmii", "rx_rgmii",
                      "rx_rmii", "rx_mii",
                      "ptp_ref";
        gmac0_mdio: mdio0 {
                compatible = "snps,dwmac-mdio";
                #address-cells = <1>;
                #size-cells = <0>;
        };
};
```

In ATF device tree file fdts/s32gxxxa-evb.dtsi:

```
&gmac0 {
        status = "okay";
        phy-mode = "rgmii";
        phy-handle = <&mdio_c_phy4>;
};

&gmac0_mdio {
        #address-cells = <1>;
        #size-cells = <0>;
        /* ARQ107 on S32RGV-VNP-PLAT */
        mdio_c_phy1: ethernet-phy@1 {
                compatible = "ethernet-phy-ieee802.3-c45";
                reg = <1>;
```

```
        };
        /* KSZ9031RNX on S32G-VNP-PROC */
        mdio_c_phy4: ethernet-phy@4 {
                reg = <4>;
                max-speed = <1000>;
        };
        /* KSZ9031RNX on S32RGV-VNP-PLAT */
        mdio_c_phy5: ethernet-phy@5 {
                status = "disabled"; /* blocked by USB by default */
                reg = <5>;
                max-speed = <1000>;
        };
};
```

The driver is using MDIO connection to the PHY to manage speed autoconfiguration.

PFE in U-Boot:
The PFE ethernet driver is controlled by several U-Boot variables and commands:

- The environment variable `pfeng_mode`: The syntax is the following: `<state>,<port0>,<port1>,<port2>`
  - `<state>` - Enable or disable the pfeng driver.
  - `<portN>` - The PFE_EMAC interface type `rgmii`, `sgmii` or `none`.

  U-Boot doesn't check the configuration validity as it is mostly board specific. The `sgmii` port type requires also corresponding `hwconfig` variable setup. The variable is read on U-Boot startup, in driver binding phase only. Any changes require U-Boot restart.
- The environment variable `pfengemac` is used to signal the active PFE port number. It can be checked by `printenv pfengemac`. It points to the PFE_EMAC interface, required for next network operation (ping, tftp-boot). The variable is checked by pfeng driver on every network operation so it is perfectly allowed to switch between PFE_EMAC ports when it is necessary. It can be changed by `setenv pfengemac <emac-number>`.
- The environment variables `pfeaddr`, `pfe1addr` and `pfe2addr`. These are the Ethernet MAC address variables for particular PFE_EMAC port.
- The command `xpcs` is used to configure SerDes XPCS module directly. The usage is the following:

```
xpcs list                                    - List all registered XPCS modules
xpcs <instance_id> transit <1000M|2500M>     - Change serdes mode
xpcs <instance_id> ss <10M|100M|1000M|2500M> - Change speed and serdes mode when required
xpcs <instance_id> an <enable|disable>       - Auto-negotiation control
xpcs <instance_id> an_auto <enable|disable>  - Auto-negotiation control with automatic speed change
xpcs <instance_id> lo <enable|disable>       - PMA loopback enable/disable
xpcs <instance_id> dump                      - Dump XPCS indirect registers
```

  `instance_id` is obtained using command `xpcs list`. Only XPCS modules which are probed without issues (e.g. clocking mismatch or corresponding lane occupied by PCIe) will be displayed.
- The command `pfeng` is used to display PFE controller info. The usage is the following:

```
pfeng info                              - important hw info
pfeng [disable|enable]                  - disable/enable full PFE/EMACs subsystem
pfeng stop                              - stop the driver but do not disable PFE/EMACs
pfeng emacs [<intf0>,<intf1>,<intf2>]   - read or set EMAC0-2 interface mode
pfeng emacs reapply-clocks              - reapply clock setting for all EMACs
pfeng reg <offset>                      - read register
```

The PFE U-Boot driver requires PFE firmware version 1.5.0 or newer located on SD card boot partition under the name *s32g_pfe_class.fw*.

The entire PFE subsystem can be disabled by the `pfeng disable` command.

Limitation: The U-Boot PFE driver does not support speed autoconfiguration. The link speed expected by the driver is 1000 Mbit/s or optionally 2500 Mbit/s for PFE_EMAC_0 and PFE_EMAC_1.

S32G3 EVB/S32G3 EVB3 default port mapping in U-Boot:

SoC port	interface	board port	note
PFE_EMAC_0	SGMII(2.5G)	J102	NOTE-1 for SGMII config
PFE_EMAC_1	SGMII(2.5G) on PLAT board	J102 on PLAT board	–
PFE_EMAC_2	RGMII on the SJA1105Q switch on PLAT board	J108, J109, J111, J112	NOTE-2 for SJA1105 firmware load
GMAC	RGMII	J108	NOTE-3 for RGMII
hwconfig	'serdes0:...;serdes1:mode=xpcs0,clock=ext,fmhz=125;xpcs1_0:speed=2G5'		

- NOTE-1: The PFE_EMAC_0 can use the SGMII interface and connect to the AQR107 PHY on the PROCEVB board. Aquantia PHY firmware is required on external flash connected to the PHY for this setup to work. If S32G EVB board is populated with sticker SCH-32170 REV B2 the Aquantia PHY firmware is populated.
- NOTE-2: The connection over the SJA1105 automotive switch requires firmware to be preloaded to the switch on every reboot:

```
=> sja probe 5:0
=> sja speed -,-,-,1G,1G 5:0
```

- NOTE-3: GMAC can share RGMII port with PFE_EMAC_1. It is possible to instantly switch RGMII to the GMAC by explicitly disabling and enabling the GMAC driver, which also reapplies pin/clocks settings:

```
=> s32ccgmac disable
=> s32ccgmac enable
=> setenv ethact eth_eqos
=> ping
```

To assign RGMII back to PFE, the U-Boot helper command for reapplying pins/clocks can be used:

```
=> pfeng emacs reapply-clocks
```

S32G399A RDB3 default port mapping in U-Boot:

SoC port	interface	board port	note
PFE_EMAC_0	SGMII(2.5G) to switch SJA1110A	P2, P4	–
PFE_EMAC_1	SGMII(2.5G)	P5	–
PFE_EMAC_2	RGMII	P3 top	–
GMAC	RGMII	P3 bottom	–
hwconfig	'serdes0:...;serdes1:mode=xpcs0,clock=ext,fmhz=125;xpcs1_0:speed=2G5'		

U-Boot MDIO support:

Ethernet controller	MDIO bus controller	MDIO PHY support
GMAC	clause 22 / clause 45	yes
PFE	clause 22 / clause 45	no

7.3.2 Ethernet fixup procedure

When U-Boot is about to start the OS (e.g. Linux, QNX, etc.), it allows to update some properties in the provided device tree file. It is used to synchronize MAC addresses on all ports, to be in sync between U-Boot and Linux OS.

The second usage is for fixing PHY addresses, if particular hardware is able to provide revision information. This is valid for RDB2 and RDB3 boards, but not for EVB/PLAT boards.

The properties **for PFE and GMAC ethernet nodes** which are managed by fixup:

1. local-mac-address (both PFE and GMAC)

   The MAC address is set to the value defined in the following variables:

U-Boot environment variable	Ethernet controller	Corresponding ethernet node
pfeaddr	PFE	pfe0
pfe1addr	PFE	pfe1
pfe2addr	PFE	pfe2
ethaddr	GMAC	eth0

2. Aquantia PHY address on S32G399A RDB3
   The Aquantia PHY may have different address on different S32G399A RDB3 revisions. This is handled automatically by the fixup via ADC measurement of resistor coding. Correct PHY address is stored also in U-Boot variable `pfe1_phy_addr`.

NOTE: NXP U-Boot has incorporated option for modification of loaded DTB before passing processing to the underlying Linux OS. The U-Boot variable `fdt_override`, if not empty, is invoked during final stage of loading Linux OS.

7.3.3 Linux Ethernet support

By default, both drivers, GMAC and PFE, are enabled for all targets. The GMAC driver is called `dwmac-s32cc` and is kernel built-in. The PFE driver is called `pfeng` and is loaded on system boot.

The PFE Linux driver and manual (PFE_LNX_DRV_S32G_UserManual.pdf) is an external module and is located in the git repository: https://github.com/nxp-auto-linux/pfeng.

The drivers are using device tree configurations, which are described in the following sources:

- GMAC: arch/arm64/boot/dts/freescale/s32cc.dtsi, arch/arm64/boot/dts/freescale/s32gxxxa-evb.dtsi;
- PFE: arch/arm64/boot/dts/freescale/s32g-pfe.dtsi, arch/arm64/boot/dts/freescale/s32gxxxa-evb.dtsi, arch/arm64/boot/dts/freescale/s32gxxxa-rdb.dtsi;

The general note: the default port mapping is the state which is valid for first start, when nothing is changed inside U-Boot environment. You can see the mapping in next two figures:

S32G3 EVB/S32G3 EVB3 default port mapping in Linux:

SoC port	interface	board port	Linux name	note
PFE_EMAC_0	SGMII(2.5G)	J102	`pfe0`	NOTE-6 for SGMII
PFE_EMAC_1	SGMII(2.5G) on PLAT board	J102 on PLAT board	`pfe1`	NOTE-7
PFE_EMAC_2	RGMII to the switch SJA1105Q on PLAT board	J108, J109, J111, J112	`pfe2`	NOTE-8
GMAC	RGMII	J108	`eth0`	–

- NOTE-6: PFE_EMAC_0 can use the SGMII interface and connect to the AQR107 PHY on the PROCEVB board. Aquantia PHY firmware is required on external flash connected to the PHY for this setup to work. If S32G EVB board is populated with sticker SCH-32170 REV B2 the Aquantia PHY firmware is populated.
- NOTE-7: The PFE_EMAC_1 is using SGMII interface connected to the AQR107/113 on PLAT board. Unfortunately the PLAT board hw rev. CX1 and newer has changed the AQR PHY address from 0x1 to 0x3. This change cannot be covered by U-Boot ethernet fixup and requires adding the following script to the U-Boot variable `fdt_override`:

```
=> setenv fdt_override 'fdt addr ${fdt_addr}; fdt resize;
                        fdt set "/soc/ethernet/mdio0/ethernet-phy@1/" reg <0x3>;
                        ${fdt_override}'
=> saveenv; reset
```

- NOTE-8: The connection over the SJA1105 automotive switch requires firmware to be preloaded to the switch on every reboot in U-Boot:

```
=> sja probe 5:0
=> sja speed -,-,-,1G,1G 5:0
```

S32G399A RDB3 default port mapping in Linux:

SoC port	interface	board port	Linux name	note
PFE_EMAC_0	SGMII(2.5G) to switch SJA1110A	P2, P4	`pfe0`	–
PFE_EMAC_1	SGMII(2.5G)	P5	`pfe1`	–
PFE_EMAC_2	RGMII	P3 top	`pfe2`	–
GMAC	RGMII	P3 bottom	`eth0`	–

- NOTE-9: The device tree nodes for PFE_EMACs (0,1,2) hold a configuration that matches:
  - the default configuration set on the board by dip-switches
  - the default `hwconfig` configuration set in U-Boot: Serdes0: Mode 0 (PCIe X2 Gen3); SerDes1: Mode 4 (PFE_EMAC_0 SGMII (2.5G) connected to SJA1110 port 4; PFE_EMAC_1 SGMII (2.5G) connected to AQR113); PFE_EMAC_2: RGMII.
  - SJA1110 firmware image for 2.5Gbps connection speed

  To change this default SerDes configuration for Linux and match e.g. one described in Table SerDes configuration examples, the device tree properties for the corresponding PFE_EMAC interface may require as well to be fixed-up, the easiest way being the U-Boot variable `fdt_override`:

  - `PFE_EMAC_0` connected to SJA1110 port 4, SGMII (unchanged); SerDes1 modes 1 (1G) and 3 (1G)

```
=> setenv fdt_override 'fdt addr ${fdt_addr}; fdt resize;
                        fdt set pfe0/fixed-link "speed" "<1000>";
                        ${fdt_override}'
```

  - `PFE_EMAC_1` connected to SJA1110 port 4 instead of AQR113, SGMII; SerDes1 mode 2 (1G)

```
=> setenv fdt_override 'fdt addr ${fdt_addr}; fdt resize;
                        fdt rm pfe1 "phy-handle"; fdt mknode pfe1 "fixed-link";
                        fdt set pfe1/fixed-link "speed" "<1000>";
                        fdt set pfe1/fixed-link "full-duplex";
                        ${fdt_override}'
```

  - `PFE_EMAC_1` connected to SJA1110 port 4 instead of AQR113, SGMII; SerDes1 mode 2 (2.5G)

```
=> setenv fdt_override 'fdt addr ${fdt_addr}; fdt resize;
                        fdt rm pfe1 "phy-handle"; fdt mknode pfe1 "fixed-link";
                        fdt set pfe1/fixed-link "speed" "<2500>";
                        fdt set pfe1/fixed-link "full-duplex";
                        ${fdt_override}'
```

  - `PFE_EMAC_2` connected to SJA1110 port 4, SGMII instead of RGMII; SerDes0 modes 2 (1G) and 3 (1G)

```
=> setenv fdt_override 'fdt addr ${fdt_addr}; fdt resize;
                        fdt rm pfe2 "phy-handle"; fdt set pfe2 "phy-mode" "sgmii";
                        fdt set pfe2/fixed-link "speed" "<1000>";
                        fdt set pfe2/fixed-link "full-duplex"
                        ${fdt_override}'
```

  - `PFE_EMAC_2` connected to SJA1110 port 4, SGMII instead of RGMII; SerDes0 mode 2 (2.5G)

```
=> setenv fdt_override 'fdt addr ${fdt_addr}; fdt resize;
                        fdt rm pfe2 "phy-handle"; fdt set pfe2 "phy-mode" "sgmii";
                        fdt set pfe2/fixed-link "speed" "<2500>";
                        fdt set pfe2/fixed-link "full-duplex"
                        ${fdt_override}'
```

Linux MDIO support:

Ethernet controller	MDIO bus controller	MDIO PHY support
GMAC	clause 22 / clause 45	yes
PFE	clause 22 / clause 45	yes

7.3.4 Build by Auto Linux BSP

To get the PFE standalone driver integrated into the NXP Auto Linux BSP, the following two lines must be added to the conf/local.conf file:

```
DISTRO_FEATURES:append = " pfe"
NXP_FIRMWARE_LOCAL_DIR = "/path/to/firmware/binaries/folder"
```

where `/path/to/firmware/binaries/folder` is the local path to the local s32g_pfe_class.fw fw file and optionally s32g_pfe_util.fw.

To build PFE multi instance driver and populate it, the following should be appended to DISTRO_FEATURES:

```
DISTRO_FEATURES:append = " pfe pfe-slave"
```

When `pfe` feature is added without `pfe-slave` feature, the populated driver will be standalone. If both features `pfe` and `pfe-slave` are present, PFE drivers are built as multi instance. Feature `pfe-slave` can be used without `pfe` feature, when the PFE master driver runs in different OS.

7.4 Setting up DHCP client

Prerequisites for using DHCP client:

- A DHCP server has to be available and correctly configured in the same network as the board.
  - Manual execution of DHCP client
    Just run the below command to obtain dynamically the IP on eth0:

```
udhcpc -i eth0
```

7.5 SSH information

The SSH server is started by default at system startup.

7.6 Setting up SJA1105 switch

There are two drivers for the SJA1105 switch. One driver is out-of-tree and is just a simple configuration loader. The second driver uses the DSA framework and is part of the Linux kernel.

7.6.1 SJA1105 Out-of-tree driver

The out-of-tree Linux kernel module is automatically inserted during boot time. Below is a kernel boot log example:

```
[    3.939877] sja1105pqrs spi1.0: Loading SJA1105P SPI driver
[    3.945225] sja1105pqrs spi1.0: Detected Device ID ae00030e ()
[    3.951026] sja1105pqrs spi1.0: Detected device id is invalid: 00000000
[    3.969679] sja1105pqrs spi1.0: 1 switch initialized successfully!
```

7.6.2 SJA1105 DSA driver

The SJA1105 DSA driver is built into the Linux Kernel image. It presents useful switch ports as network interfaces and provides support for other features of the switch like VLAN and PTP.

Example of network interfaces when the SJA1105 DSA driver is loaded:

```
root@s32g399aevb3:~# ip addr show
...
7: pfe2: <BROADCAST,MULTICAST> mtu 1504 qdisc noop state DOWN group default qlen 1000
    link/ether 00:01:be:be:ef:33 brd ff:ff:ff:ff:ff:ff
8: enet_p1@pfe2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether 00:01:be:be:ef:33 brd ff:ff:ff:ff:ff:ff
9: enet_p3@pfe2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether 00:01:be:be:ef:33 brd ff:ff:ff:ff:ff:ff
10: enet_p4@pfe2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether 00:01:be:be:ef:33 brd ff:ff:ff:ff:ff:ff
```

In order to use the switch interfaces, the DSA master port (pfe2 interface in case of the S32G274A EVB board) must be up, otherwise the link of these interfaces cannot be set up.
Example output with the DSA master down:

```
root@s32g399aevb3:~# ip link set dev enet_p4 up
RTNETLINK answers: Network is down
```

Example output with the DSA master up:

```
root@s32g399aevb3:~# ip link set dev pfe2 up
root@s32g399aevb3:~# [  828.805756] 002: pfeng 46000000.pfe: HIF2 started
[  828.806018] 002: pfeng 46000000.pfe pfe2: configuring for fixed/rgmii link mode
[  828.806037] 002: pfeng 46000000.pfe pfe2: Set TX clock to 125000000Hz
[  828.806262] 002: pfeng 46000000.pfe pfe2: Set TX clock to 125000000Hz
[  828.806279] 002: pfeng 46000000.pfe pfe2: Link is Up - 1Gbps/Full - flow control off
[  828.806494] 002: IPv6: ADDRCONF(NETDEV_CHANGE): pfe2: link becomes ready
root@s32g399aevb3:~# ip link set dev enet_p4 up
[  845.913867] 002: sja1105 spi5.0 enet_p4: configuring for phy/rgmii-id link mode
```

In the default use case, when there is no VLAN Filtering enabled, no bridge is set up and so on, the IP addresses should be assigned to the switch interface.

```
root@s32g399aevb3:~# ip link set dev enet_p4 up
[   75.109811] 002: sja1105 spi5.0 enet_p4: configuring for phy/rgmii-id link mode
[   80.218319] 002: sja1105 spi5.0 enet_p4: Link is Up - 1Gbps/Full - flow control off
[   80.218358] 002: IPv6: ADDRCONF(NETDEV_CHANGE): enet_p4: link becomes ready
root@s32g399aevb3:~# ip addr add 10.42.0.101/24 dev enet_p4
root@s32g399aevb3:~# ping 10.42.0.1
PING 10.42.0.1 (10.42.0.1) 56(84) bytes of data.
64 bytes from 10.42.0.1: icmp_seq=1 ttl=64 time=0.643 ms
64 bytes from 10.42.0.1: icmp_seq=2 ttl=64 time=0.315 ms
^C
--- 10.42.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1027ms
rtt min/avg/max/mdev = 0.315/0.479/0.643/0.164 ms
```

If the driver of the DSA master is compiled as a module and that module is removed at runtime and inserted again, the DSA slave needs to be bound manually. Example of binding sja1105 on s32g399aevb3:

```
[root@s32g399aevb3 ~]# echo spi5.0 > /sys/bus/spi/drivers/sja1105/bind
[ 1342.167204] sja1105 spi5.0: Probed switch chip: SJA1105Q
[ 1342.185861] sja1105 spi5.0: configuring for fixed/rgmii-id link mode
[ 1342.187121] sja1105 spi5.0: Link is Up - 1Gbps/Full - flow control off
[ 1342.247496] sja1105 spi5.0 enet_p1 (uninitialized): PHY [PFEng Ethernet MDIO.2:06]
        driver [NXP TJA1102 Port 0] (irq=POLL)
[ 1342.288843] sja1105 spi5.0 enet_p1: configuring for phy/mii link mode
[ 1342.311675] sja1105 spi5.0 enet_p2 (uninitialized): PHY [PFEng Ethernet MDIO.2:07]
        driver [NXP TJA1102 Port 1] (irq=POLL)
[ 1342.353331] sja1105 spi5.0 enet_p2: configuring for phy/mii link mode
[ 1342.426419] sja1105 spi5.0 enet_p3 (uninitialized): PHY [PFEng Ethernet MDIO.2:02]
        driver [Micrel KSZ9031 Gigabit PHY] (irq=POLL)
[ 1342.471707] sja1105 spi5.0 enet_p3: configuring for phy/rgmii-id link mode
[ 1342.538420] sja1105 spi5.0 enet_p4 (uninitialized): PHY [PFEng Ethernet MDIO.2:03]
        driver [Micrel KSZ9031 Gigabit PHY] (irq=POLL)
[ 1342.539755] DSA: tree 0 setup
[ 1342.583596] sja1105 spi5.0 enet_p4: configuring for phy/rgmii-id link mode
```

For more information see:

- https://www.kernel.org/doc/html/latest/networking/dsa/dsa.html
- https://www.kernel.org/doc/html/latest/networking/dsa/sja1105.html

7.7 Setting up SJA1110 switch

7.7.1 Building firmware image for SJA1110 switch

Instructions for building the firmware image for SJA1110 can be found in: `RDB3 Ethernet Enablement Guide`, chapter 3. SJA1110 Port Configuration.

See the `References` section of this manual.

7.7.2 Setting up SJA1110 switch over SPI

The Linux kernel module is automatically inserted at boot time on S32G399A RDB3. In order for the driver to update the firmware of the switch and/or the uC, please provide the path to the binaries on the host machine in `conf/local.conf` before the yocto build.

```
SJA1110_SWITCH_FW = <path_to_sja_switch_binaries>
SJA1110_UC_FW = <path_to_sja_uc_binaries>
```

Where path_to_sja_switch_binaries is `.../sja1110_switch.bin` and path_to_sja_uc_binaries is `.../sja1110_uc.bin`.

NOTE: The user can also simply copy the binaries in rootfs at the following path: `/lib/firmware`.

In order for the firmware to be updated at boot time, J189 must be configured as Pin1-2: Short, Pin3-4: Short.

8 PTP Information

8.1 Precision Time Protocol - IEEE 1158

The Precision Time Protocol (PTP) is a high-precision time synchronization protocol used to synchronize clocks throughout a computer network. PFE and GMAC interfaces have support for hardware timestamping. To check the support for hardware timestamping, *ethtool* can be used. The *linuxptp* tools are installed by default in *fsl-image-auto*. Depending of the *ptp4l* configuration parameters, the interfaces can act as a master clock or as a slave clock.

Example for checking the timestamping support:

```
root@s32g399aevb3:~# ethtool -T eth0
Time stamping parameters for eth0:
Capabilities:
        hardware-transmit
        software-transmit
        hardware-receive
        software-receive
        software-system-clock
        hardware-raw-clock
PTP Hardware Clock: 0
Hardware Transmit Timestamp Modes:
        off
        on
Hardware Receive Filter Modes:
        none
        all
        ptpv1-l4-event
        ptpv1-l4-sync
        ptpv1-l4-delay-req
        ptpv2-l4-event
        ptpv2-l4-sync
        ptpv2-l4-delay-req
        ptpv2-event
        ptpv2-sync
        ptpv2-delay-req
```

Example of synchronization between a foreign master and GMAC using hardware timestamping(`-H`) at layer 2(`-2`):

```
root@s32g399aevb3:~# ptp4l -2 -H -i eth0 -m -s
ptp4l[136.327]: selected /dev/ptp0 as PTP clock
ptp4l[136.380]: port 1: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[136.380]: port 0: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[143.364]: selected local clock 7684ef.fffe.c74e40 as best master
ptp4l[170.418]: port 1: new foreign master 7824af.fffe.d9dce0-1
ptp4l[174.418]: selected best master clock 7824af.fffe.d9dce0
ptp4l[174.419]: port 1: LISTENING to UNCALIBRATED on RS_SLAVE
ptp4l[176.418]: master offset -1627543141783358 s0 freq      +0 path delay     39752
ptp4l[177.418]: master offset -1627543141839918 s1 freq  -56556 path delay     39752
ptp4l[178.418]: master offset         70 s2 freq  -56486 path delay     39752
```

```
ptp4l[178.418]: port 1: UNCALIBRATED to SLAVE on MASTER_CLOCK_SELECTED
ptp4l[179.418]: master offset        150 s2 freq  -56385 path delay     39752
ptp4l[180.418]: master offset      11740 s2 freq  -44750 path delay     28112
ptp4l[181.418]: master offset      11557 s2 freq  -41411 path delay     16650
ptp4l[182.418]: master offset      -3775 s2 freq  -53276 path delay     16862
...
ptp4l[204.420]: master offset       -239 s2 freq  -56406 path delay     16501
ptp4l[205.420]: master offset       -209 s2 freq  -56448 path delay     16501
ptp4l[206.420]: master offset        -63 s2 freq  -56364 path delay     16465
ptp4l[207.420]: master offset        -73 s2 freq  -56393 path delay     16435
ptp4l[208.420]: master offset         -8 s2 freq  -56350 path delay     16435
...
ptp4l[232.423]: master offset         79 s2 freq  -56273 path delay     16413
ptp4l[233.423]: master offset        -40 s2 freq  -56368 path delay     1645
```

NOTE: On EVB boards, where PFE2 or PFE0 (depending by the pinmuxing) is connected to SJA1105, the switch network interfaces should be used instead of the PFE interface connected to the switch.

9 PCIe Support

9.1 PCIe in U-Boot

S32G3 SoC has two SerDes modules (0 and 1), each of them with one PCIe controller.
A SerDes can be shared between PCIe and PFE or GMAC.
U-Boot enables PCIe by default on SerDes0 and PFE0 at 2.5Gbps on SerDes1; the default configuration on PCIe0 is Root Complex (RC) mode, using the external clock.
Boot configuration for both SerDes modules is set using the environment variable `hwconfig`.
The format for this variable is (all on one row):

```
=> setenv hwconfig "serdes0:<ss_settings_0>[;pcie0:<pci_settings_0>][;xpcs0_j:<xpcs_settings_0_j>];
                    serdes1:<ss_settings_1>[;pcie1:<pci_settings_1>][;xpcs1_j:<xpcs_settings_1_j>]"
```

Where:

- `<ss_settings_i>` is :

  `mode=<ss_mode_i>,clock=<clock_i>[,skip=<value_i>][,fmhz=<fmhz_i>]`

  `i` is the index of the SerDes module (0 or 1), and the valid values for SerDes options `mode`, `clock`, `fmhz` are defined below:

mode	clock	fmhz	description
'pcie'	'ext' (Gen3) or 'int' (Gen2)	(unset)	Mode 0, Serdes 0 or 1
'pcie&xpcs0' (1G)	'ext' (Gen3) or 'int' (Gen2)	'100'	Mode 1, Serdes 1 only (PFE0) Note: GMAC0 as SGMII is not supported
'pcie&xpcs1' (1G)	'ext' (Gen3) or 'int' (Gen2)	'100'	Mode 2, Serdes 0 or 1
'pcie&xpcs1' (Gen2/2.5G)	'ext' or 'int'	'100'	Mode 2, Serdes 0 or 1
'xpcs0&xpcs1' (1G)	'ext' or 'int'	'100' or '125'	Mode 3, SerDes 0 or 1
'xpcs0&xpcs1' (2.5G)	'ext' or 'int'	'125'	Mode 4, SerDes 1 only

Table 11: `serdesi` options values

SerDes option	values	description
clock	'ext' 'int'	External clock (PCIE*i*_CLK_N/P is the reference clock for the PCIe PHY PLL). Can be used for all PCIe modes (Gen1 to Gen3). Internal clock (SERDES_REF_CLK is the reference clock for the PCIe PHY PLL). Can be used only for Gen1 or Gen2 PCIe modes.
fmhz	'100' or '125'	Clock frequency used for SGMII. Not required for PCIe only modes. For 'clock=ext' it should match the rate set for the board by jumper or switch.
skip	'1'	If set, skips configuration of the corresponding SerDes and PCIe controller in U-Boot.

Table 12: `serdesi` options descriptions

The skip is optional, it allows skipping SerDes and PCIe controller configuration in U-Boot, thus allowing full configuration to be performed from Linux directly. Some PCIe devices (e.g. PCIe switches) cannot be configured twice, as the PHY does not link and remains in POLL_ACTIVE state. skip is enabled if <value_*i*> is "1".

Note: SerDes modes with one lane PCIe, one lane SGMII, i.e. pcie&xpcs0 or pcie&xpcs1 are also called combo modes.
When using combo modes, the only frequency supported (fmhz) is 100 (MHz).

- <pci_settings_*i*> is :

  mode=<pci_mode_*i*>[,phy_mode=<phy_*i*>]

  i is the index of the SerDes module (0 or 1), the valid values for mode are or, and phy_mode is optional and is described in "PCIE PHY CRSS and SRIS support"

  Note: pcie*i* is required only if the mode for the corresponding SerDes*i* is one of pcie, pcie&xpcs0 or pcie&xpcs1.

- <xpcs_settings_*i*_*j*> is :

  speed=<10M|100M|1G|2G5>

  i is the index of the SerDes module (0 or 1), j is the index of the XPCS interface (0 or 1).

  Note: XPCS0 is GMAC0 on SerDes0 and PFE0 on SerDes1.
  XPCS1 is PFE2 on SerDes0 and PFE1 on SerDes1.
  GMAC0 can't be used as SGMII for now.

  Note: Options xpcs*i*_*j* are required only if the string xpcs*j* can be found in the value of the option mode for the corresponding SerDes*i* module.

In case of SGMII only modes (Mode 3 and Mode 4), serdes*i* option mode must be set to xpcs0&xpcs1 and both xpcs*i*_0 and xpcs*i*_1 should be configured for SerDes*i*.
Note: hwconfig has to be in sync with pfeng_mode.
The syntax for pfeng_mode is: <state>,<pfe0>,<pfe1>,<pfe2>, where state should be enabled and pfe*n* should be:

- sgmii if the corresponding xpcs*i*_*j* is set by hwconfig
- rgmii or none otherwise.

It is possible to enable only one XPCS interface for SGMII only modes (Mode 3 or 4); in this case, the corresponding pfe*n* in pfeng_mode should be set to none.

The default value of hwconfig for S32G3 EVB and S32G399A RDB3 platforms is (all on one row):

```
=> setenv hwconfig "serdes0:mode=pcie,clock=ext;pcie0:mode=rc;
                    serdes1:mode=xpcs0&xpcs1,clock=ext,fmhz=125;xpcs1_0:speed=2G5"
```

To override the default settings, the U-Boot environment variable `hwconfig` must be changed at runtime. If the variable is not set, neither PCIe nor GMAC nor PFE is enabled on any of the two SerDes modules.

Note: If an SD-card is reused on different SoC versions, the user must make sure that both the U-Boot binary and the `hwconfig` variable are properly set/reset, since writing a new U-Boot will not change a pre-existing `hwconfig` value. For more details on the steps to set/reset the U-Boot environment, please refer to the section "How to reset U-Boot environment" in this manual.

9.1.1 PCIE PHY CRSS and SRIS support

CRSS (Common Reference Clock, Spread Spectrum Clock) and SRIS (Split Reference, Independent Spread Spectrum Clock) can be enabled using the environment variable 'hwconfig'. The token which needs to be used is 'phy_mode', and can only be used in PCIe-only mode with external clock.

With this, the format of 'hwconfig' is (all on one row):

```
setenv hwconfig "serdes0:mode=pcie,clock=ext;pcie0:mode=<rc|ep>,phy_mode=<mode>;
                 serdes1:mode=pcie,clock=ext;pcie1:mode=<rc|ep>,phy_mode=<mode>"
```

where `<mode>` has one of the following valid values:

crns	crss	sris

Note: CRSS or SRIS PCIe PHY mode cannot be used with internal clock.

Spread Spectrum clocking can be enabled on S32G3 EVB platform, on PCIe0, by using the SW12 switch. Please note that Spread Spectrum clock should be provided externally, otherwise the non-Spread Spectrum clock will be used.

Note: By default, if <phy_mode> is not specified in the 'hwconfig' environment variable, PCIe PHY mode is set to CRNS (Common Reference Clock, No Spread Spectrum Clock).

Note: When using CRSS or SRIS PCIe PHY mode, no SGMII mode can be used on the same SerDes subsystem. Therefore, an example of 'hwconfig' environment variable modified in order to enable CRSS on PCIe0 and SRIS on PCIe1, would be:

```
setenv hwconfig "serdes0:mode=pcie,clock=ext;pcie0:mode=rc,phy_mode=crss;
                 serdes1:mode=pcie,clock=ext;pcie1:mode=rc,phy_mode=sris"
```

Note: If CRSS or SRIS PCIe PHY mode is enabled in 'hwconfig' environment variable, there are no other changes required for Linux Kernel, in order for CRSS or SRIS PCIe PHY mode to be enabled.

9.2 PCIe in Linux

With respect to Linux kernel setup, PCIe support is enabled by default.

PCIe enablement and selection EP versus RC is done exclusively from U-Boot, following the steps in section "PCIe in U-Boot" to select the mode from U-Boot's `hwconfig` variable. In particular, that means that U-Boot must have PCIe support in order to enable it in the kernel.

9.2.1 Enable PCIe manually

In order to have PCIe enabled, the kernel *.config* file must contain the following CONFIGs:

```
CONFIG_PCI=y
CONFIG_PCI_MSI=y
CONFIG_PCI_MSI_IRQ_DOMAIN=y
CONFIG_PCI_S32GEN1=y
CONFIG_PCIE_DW=y
CONFIG_PCIE_DW_PLAT_HOST=y
CONFIG_PCI_ENDPOINT=y
CONFIG_PCIE_DW_PLAT_EP=y
CONFIG_PCI_S32GEN1_INIT_EP_BARS=y
```

This can be achieved by adding them manually to *.config* then running `make olddefconfig`

Note: Both RC and EP PCIe support must be enabled at build time in the kernel to allow any of the two modes to be selected at runtime from U-Boot.

9.2.2 Rescan PCIe Bus

In the case when the S32 board is configured as RootComplex and for some reason an EndPoint is not enumerated or is not enumerated correctly (e.g. slower initialization or another reason), we may need to rescan the corresponding PCIe bus.

For example, for PCIe0:

remove the bus pcie0:

```
echo 1 > /sys/bus/pci/devices/0000\:00\:00.0/remove
```

rescan the bus pcie0:

```
echo 1 > /sys/bus/pci/rescan
```

This will trigger a re-enumeration of the PCIe bus and re-initialization of its EndPoints.

9.2.3 PCIe MSIs

There are two PCIe MSI delivery mechanisms supported by S32G3 and configurable in the Linux BSP:

1. Writes to the GIC-500 configuration space, triggering a message-based GIC SPI. Linux BSP currently reserves a range of 16 physical interrupt numbers, starting with GIC interrupt 167. This is configurable through the Linux device-tree, should this range need to be modified. If you decide to use a different interrupt range, make sure to choose one that is not in use by active SoC peripherals.

   Refer to the S32G3 Reference Manual (the Interrupt Routing annex) for the list of SoC interrupts.

Refer to Documentation/devicetree/bindings/pci/nxp,s32-pcie.yaml for the relevant device-tree bindings in Linux BSP.

This is the default configuration. It is most useful for multi-queue PCIe Endpoints such as NVMe-s, advanced Ethernet cards, or for multiple Endpoints connected to the PCIe RootComplex.

2. Incoming MSIs can also be handled by iMSI-RX (Integrated MSI Receiver [AXI Bridge]), an integrated MSI reception module in the PCIe controller's AXI Bridge which detects and terminates inbound MSI requests received on the RX wire. Instead, a hard-wired interrupt documented as "DSP AXI MSI Interrupt" in the SoC RM (Interrupt Routing annex) is asserted.

   The hardware defaults enable this option at reset, but the PCIe Host device-tree node overrides it with the other option, presented above.

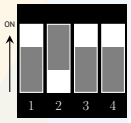
   The disadvantage of this approach is that only one hardware interrupt can be used, thus multiplexing multi-queue Endpoints onto a single local SPI and potentially producing a single-core performance bottleneck.

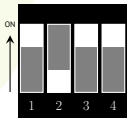
Changing the default MSI handling from GIC SPIs to hard-wired interrupts can be done fairly easily by looking up and commenting the *"msi-parent"* node property in arch/arm64/boot/dts/freescale/s32cc.dtsi for the PCIe node of interest and recompiling the device-tree:

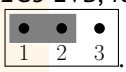
```
/* msi-parent = <&gic>; */
```

9.3 PCIe Clock Configuration

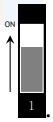
In order to use SerDes lanes as PCIe, the clock should be set to 100 MHz.

In case of S32G3 EVB, for PCIe0, set **SW12** 

in order to select 100 MHz. For PCIe1, set **SW13** 

along with **J165** 

.

In case of S32G399A RDB3, for PCIe1, set **SW8** 

.

10 HSE Security Support

The NXP Hardware Security Engine is a security subsystem aimed at running relevant security functions for applications with stringent confidentiality and authenticity requirements. This chapter describes the supported HSE features.

10.1 Prerequisites

The HSE Firmware is required to leverage the features described in the next sections. The latest supported firmware versions are the following:

- HSE_FW_S32G2_0_1_0_5
- HSE_FW_S32G2_1_1_0_1

- HSE_FW_S32G3_0_2_16_1
- HSE_FW_S32G3_1_2_16_1

- HSE_FW_S32R45_0_1_0_4

The following older firmware versions are also compatible with the crypto driver:

- HSE_FW_S32G2_0_1_0_0
- HSE_FW_S32G2_0_1_0_1
- HSE_FW_S32G2_1_1_0_0

HSE is a separate NXP product, available through Flexera. For more information, and to find out how to obtain HSE, please contact our marketing department.

Information on how to configure the HSE Firmware is available in the HSE Reference Manual, available on the NXP DocStore. More information regarding the HSE API is available in the HSE Service API Reference Manual, available on Flexera.

10.2 PKCS11 Support

The HSE support for PKCS11 provides a userspace module (`libpkcs-hse.so`) that enables communication with HSE from userspace applications. Moreover, the userspace HSE messaging module (`libhse.so.1.0`) provides direct low-level access to HSE. The PKCS11 support is based on the PKCS11 v3.0 spec [3].

[3] https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.html

10.2.1 Prerequisites

The Linux Kernel must be compiled with the CONFIG_UIO_NXP_HSE config enabled. The Messaging Unit (MU) used by the Userspace I/O (UIO) driver can be selected using the CONFIG_UIO_NXP_HSE_MU. See section 3.2.3 for more information on building the Linux kernel.

The current implementation has been tested with the following:

- libp11 0.4.11 (2020-Oct-11)
- OpenSSL 1.1.1 (2018-Sep-11)
- OpenSC 0.21.0 (2020-Nov-24)

Testing can be done with either libp11 or OpenSC's *pkcs11-tool*, both of which depend on OpenSSL. Since the HSE PKCS11 module is compiled for aarch64, all of them need to be cross-compiled for aarch64, as well.

Take note that the paths provided in the following sections are simply examples, and can be changed to fit your build environment.

Before beginning cross-compilation, make sure a cross-compiler is installed, which fits the desired target system's architecture. The CROSS_COMPILE environment variable should already be set, if you have followed 3.2.2
Going forth, it is assumed CROSS_COMPILE is set.

10.2.2 Building libp11 0.4.11 for aarch64

The source code for libp11 can be download from the project's GitHub page:

https://github.com/OpenSC/libp11/releases

Configure the build system for aarch64 cross-compilation:

```
./configure --host=aarch64-linux-gnu --prefix=/<path>/<to>/libp11-aarch64
```

-host specifies the target architecture, while -prefix indicates a directory in which to place the cross-compiled files. It is recommended to use the option, otherwise the files will be placed in the host's *$PATH*. The path provided for -prefix is an example.

libp11 can now be built:

```
make
sudo make install
```

The compiled files are under *<path>/<to>/libp11-aarch64*. *lib/libp11.so.3.5.0* must be copied to the target's */usr/lib* directory. Run ldconfig afterwards, so that the library can be dynamically linked to.

10.2.3 Building OpenSSL 1.1.1 for aarch64

The OpenSSL source code can be download from the project's website:

https://www.openssl.org/source/old/1.1.1/

Configure the build system for aarch64 cross-compilation:

```
./Configure linux-aarch64 --prefix=/<path>/<to>/openssl-aarch64
```

As before, `-prefix` indicates a directory in which to place the cross-compiled files. It is recommended to use the option, otherwise the files will be placed in the host's *$PATH*. The path provided is an example.

OpenSSL can now be built:

```
make
sudo make install
```

The compiled files can be found under */<path>/<to>/openssl-aarch64*. *lib/libcrypto.so.1.1* must be copied to the target's */usr/lib* directory. Run `ldconfig` afterwards, so that the library can be dynamically linked to.

10.2.4 Building OpenSC's pkcs11-tool

The OpenSC source code can be downloaded from the project's GitHub page:

https://github.com/OpenSC/OpenSC/releases

Install the prerequisites beforehand:

```
sudo apt-get install pcscd libccid libpcsclite-dev libssl-dev libreadline-dev \
        autoconf automake build-essential docbook-xsl xsltproc libtool pkg-config
```

Configure the build system for aarch64 cross-compilation. OpenSSL must have been cross-compiled beforehand, as OpenSC needs to link against it:

```
./bootstrap
./configure --host=aarch64-linux --prefix=/<path>/<to>/opensc-aarch64-test --enable-openssl \
        CC=<path>/<to>/<cross>/<compiler>/aarch64-linux-gnu-gcc \
        LDFLAGS=-g -Wl,-rpath,/<path>/<to>/openssl-aarch64/lib \
        OPENSSL_LIBS=-lcrypto -L/<path>/<to>/openssl-aarch64/lib \
        OPENSSL_CFLAGS=-I/<path>/<to>/openssl-aarch64/include
```

As before, `-prefix` indicates a directory in which to place the cross-compiled files. It is recommended to use the option, otherwise the files will be placed in the host's *$PATH*. The path provided is an example.

OpenSC can now be built:

```
make
sudo make install
```

The compiled files are under */<path>/<to>/opensc-aarch64*. *lib/libopensc.so.7.0.0* must be copied to the target's */usr/lib* directory. Run `ldconfig` afterwards, so that the library can be dynamically linked to.

bin/pkcs11-tool can be placed anywhere on the target (e.g. */home/<user>*).

10.2.5 Building the HSE PKCS11 module

Clone the module's source code:

```
git clone https://github.com/nxp-auto-linux/pkcs11-hse
cd pkcs11-hse/
git checkout release/bsp36.0_cd
```

Since the module communicates with HSE, the HSE FW interface sources are needed and must be present in the host file system. As such, `HSE_FWDIR` should be set before running make (or provided when running make). Depending on firmware, the interface files have to correspond to premium or standard HSE FW.

The value used for `HSE_FWDIR` should point to the folder that contains the HSE FW *interface* folder, */<path>/<to>/HSE_FW_<platform>_<premium>_<vmajor>_<vminor>_<vpatch>*. The options represent:

1. <platform> - S32G2, S32G3, or S32R45
2. <premium> - 1 for premium FW, 0 for standard
3. <vmajor> - major version number
4. <vminor> - minor version number
5. <vpatch> - patch version number

In the case of standard S32G3 FW, `HSE_FWDIR` would be:

```
/<path>/<to>/HSE_FW_S32G3_0_2_16_1
```

To compile, run:

```
make HSE_FWDIR=/<path>/<to>/HSE_FW_<platform>_<premium>_<vmajor>_<vminor>_<vpatch>
```

This generates the `libpkcs-hse.so` and `libhse.so.1.0` modules. The `libhse.so.1.0` library should be copied to the target's rootfs in */usr/lib*. Afterwards, run `ldconfig` so that it can be dynamically linked to.

10.2.6 Building and running the libp11 PKCS11 HSE example

A usage example is provided in the repo under examples/. The sample application links against OpenSSL and libp11 to use functions provided by both. The demo application will:

- initialize libp11 with the `libpkcs-hse.so` module
- find the slot and token corresponding to HSE
- use openssl to generate two RSA key pairs
- store the key pairs in HSE
- enumerate the stored keys
- remove one of the key pairs from HSE

Since the application must be linked against both *libcrypto* and *libp11*, the directories in which the cross-compiled OpenSSL and libp11 are stored must be specified:

```
make OPENSSL_DIR=/<path>/<to>/openssl-aarch64 LIBP11_DIR=/<path>/<to>/libp11-aarch64
```

Copy the demo application, then run from the target's rootfs:

```
./pkcs-keyop /<path>/libpkcs-hse.so
```

The application outputs a message for each step described previously.

10.2.7 Running the pkcs11-tool PKCS11 HSE examples

Alternatively, instead of `pkcs-keyop`, `pkcs11-tool` can be used to load RSA keys (public/pair), EC keys (public) and AES keys. The `-id` switch corresponds to the key's slot (`00`), group (`06`) and catalog (`01`), in hexadecimal, from the HSE Key Catalog. Some examples:

```
./pkcs11-tool --module /<path>/libpkcs-hse.so --write-object /<path>/rsa_keypair.der \
        --type privkey --id 000601 --label "HSE-RSAPRIV-KEY"
./pkcs11-tool --module /<path>/libpkcs-hse.so --write-object /<path>/rsa_keypub.der \
        --type pubkey --id 000701 --label "HSE-RSAPUB-KEY"
./pkcs11-tool --module /<path>/libpkcs-hse.so --write-object /<path>/ec_keypub.der \
        --type pubkey --id 000401 --label "HSE-ECPUB-prime256v1-KEY"
./pkcs11-tool --module /<path>/libpkcs-hse.so --write-object /<path>/aes.key \
        --type secrkey --key-type AES:256 --id 000101 --label "HSE-AES-256-KEY"
```

The tool displays a message if the key import operation is successful.

10.2.8 Building the libhse examples

Two examples demonstrating the usage of the HSE Userspace Library (`libhse`) are provided: `hse-sysimg` and `hse-encrypt`. As they both depend on the HSE FW interface, *make* must be invoked in the *examples* folder while indicating the path to the HSE FW *interface* directory:

```
make HSE_FWDIR=<path>/<to>/HSE_FW_<platform>_<premium>_<vmajor>_<vminor>_<vpatch>
```

10.2.9 Building the HSE Secure Boot demo application

For the HSE Secure Boot support, a userspace demo application is provided as part of HSE PKCS#11 repo. The demo application requires the headers from a cross-compiled OpenSSL to read and interpret the key used for the authentication of the TF-A FIP, which can be built as described in section 10.2.3. The HSE FW interface is also required.

From the examples/ folder:

```
make OPENSSL_DIR=/<path>/<to>/openssl-aarch64 \
        HSE_FWDIR=/<path>/<to>/HSE_FW_<platform>_<premium>_<vmajor>_<vminor>_<vpatch>
```

The *hse-secboot* demo application must be copied to the target board's rootfs, as well as the OpenSSL cross-compiled library, as described in section 10.2.3.

The help for the demo can be brought up with:

```
./hse-secboot -h
```

10.3 HSE Secure Boot

Secure boot refers to a two-stage process of successive authentication of the TF-A FIP and Linux kernel images. This process requires a "root of trust", which is known to be secure. In this case, the root of trust is HSE itself.

Each image is authenticated by the preceding step. Thus, the secure boot flow is the following:

1. BootROM passes control over to HSE FW;
2. HSE FW authenticates the TF-A FIP, including U-Boot;
3. U-Boot authenticates the Kernel image.

NOTE
HSE Secure Boot is currently only supported when booting from an SD card!

NOTE
Currently, enabling secure boot cannot be done via Yocto. As such, the TF-A FIP must be built manually with HSE-provided secure boot options enabled!

10.4 Manually Building TF-A FIP and Kernel for Authentication

Before beginning, make sure you have followed sections 3.2.1, 3.2.2 and 3.2.4, and obtained the U-Boot and TF-A source code repositories.

10.4.1 Configure TF-A FIP Authentication

10.4.1.1 Building U-Boot with support for HSE features

Begin by running make with the desired defconfig:

```
make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- <board>_defconfig
```

You can also export the CROSS_COMPILE environment variable, so as not to repeat it for every build:

```
export CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu-
```

To be able to use HSE features, the following configuration options must be enabled or changed in menuconfig (`make menuconfig`):

```
CONFIG_NXP_HSE_SUPPORT=y
CONFIG_NXP_HSE_FW_FILE=<path>/<to>/<HSE>/<directory>/<hse_fw_file>.bin.pink
```

Afterwards, U-Boot can be compiled:

```
make
```

This generates the *u-boot-nodtb.bin* file, which is needed for the next step, compiling ARM Trusted Firmware.

10.4.1.2 Building TF-A FIP with support for HSE features

To build TF-A with support for HSE Secure Boot, the following make command needs to be run from the TF-A repo:

```
make ARCH=aarch64 PLAT=<board> BL33=/<path>/<to>/<u-boot-dir>/u-boot-nodtb.bin HSE_SECBOOT=1
```

The `HSE_SECBOOT=1` option reserves space in the FIP for the signature.

```
IVT:                  Offset: 0x1000     Size: 0x100
DCD:                  Offset: 0x1200     Size: 0x2000    (optional)
HSE Reserved:         Offset: 0x1400
HSE SYS Image:        Offset: 0x52600    Size: 0xc000
AppBootCode Header:   Offset: 0x5e600    Size: 0x40
Application:          Offset: 0x5e640    Size: 0x150400

IVT Location:        SD/eMMC
Load address:        0x3407a2e0
Entry point:         0x34080000
```

NOTE
Depending on the targeted board and U-Boot configuration, the output for component offsets, component sizes, IVT location, load address or entry point will differ.

This generates the *build/<board>/release/fip.bin* and *build/<board>/release/fip.s32* files, which will be needed for the next steps.

A pair of RSA keys to sign and verify the FIP files generated by TF-A must be created. In this case, a pair of 2048-bit RSA keys will be generated.

```
openssl genrsa -out rsa2048_private.pem 2048
openssl rsa -in rsa2048_private.pem -pubout -out rsa2048_public.pem
```

NOTE
Standard HSE FW supports key sizes up to 2048-bits, while Premium HSE FW supports up to 4096-bits, though it requires a different Key Catalog configuration.

Since the FIP signature is stored in the FIP, the *fip.bin* must be signed without the section reserved for it. First, determine how much of the FIP must be signed:

```
tools/fiptool/fiptool info fip.bin

Trusted Boot Firmware BL2: offset=0x100, size=0x7D6AC, cmdline="--tb-fw"
EL3 Runtime Firmware BL31: offset=0x7D7B0, size=0x2FF5D, cmdline="--soc-fw"
Non-Trusted Firmware BL33: offset=0xAD710, size=0xAADD0, cmdline="--nt-fw"
SOC_FW_CONFIG: offset=0x1584E0, size=0x5C1F, cmdline="--soc-fw-config"
Trusted Boot Firmware BL2 certificate: offset=0x15E100, size=0x100, cmdline="--tb-fw-cert"
```

The relevant part of the FIP can be extracted and signed with the private key. In this example, the first `0x15E100` bytes of the FIP must be signed:

```
dd if=fip.bin of=tosign-fip.bin bs=1 count=`printf "%d" 0x15e100` conv=notrunc
openssl dgst -sha1 -sign rsa2048_private.pem -out fip-signature.bin tosign-fip.bin
```

The resulting *fip-signature.bin* must be written back into *fip.bin*:

```
tools/fiptool/fiptool update --align 16 --tb-fw-cert fip-signature.bin fip.bin
```

The public RSA key must be copied to the boot partition of the SD Card, next to the Linux Kernel Image and DTB file. *fip.s32* must be written to the SD Card. *fip.bin* must be written in *fip.s32* at the address pointed to by the `Application` label in the compile log; in this example, the address is `0x5e640`.

```
dd if=build/<board>/release/fip.bin of=build/<board>/release/fip.s32 seek=`printf "%d" 0x5e640` \
        oflag=seek_bytes conv=notrunc,fsync
dd if=build/<board>/release/fip.s32 of=/dev/<sdcard_dev> seek=512 skip=512 \
        iflag=skip_bytes oflag=seek_bytes conv=notrunc,fsync
```

10.4.1.3 Enable TF-A FIP authentication

To authenticate the TF-A FIP image, HSE uses Advanced Secure Boot (ASB). The system configuration is saved as a **System Image** (SYS_IMG), which contains data related to imported keys, Secure Memory Region (SMR) configuration

and Core Reset (CR) entry configuration. The SYS_IMG is signed with a device-specific key.

The keys used for ASB are stored in a **NVM key catalog**, which contains multiple slots for multiple key types and authentication schemes.

The **Secure Memory Region entry** is used to define which data needs to be authenticated from the boot medium, which key and authentication scheme is required, and where to place the data in memory after authentication.

The **Core Reset entry** is used to define which core should be enabled and what address to jump to after authentication, as well as what action to take in case authentication fails.

To enable the authentication of the TF-A FIP by HSE, the `hse-secboot` userspace binary is provided as part of the *pkcs11-hse* repository examples, as described in section 10.2.9. After building and copying the *hse-secboot* demo application to the target board's rootfs, you can use it to enable HSE Secure Boot:

```
./hse-secboot -s -d <device> -k rsa2048_public.pem
```

The options represent:

1. -s - enable secboot
2. -d - specifies the <device> used as boot media (e.g. /dev/mmcblk0)
3. -k - specifies the public key used for signature verification

After the command is finished, simply reset the board to boot in secure mode. To verify if the board has booted in secure mode, check the BOOT_SEQ bit in the IVT, at offset 0x48. In the U-Boot command line, run:

```
=> mmc read ${loadaddr} 8 1
=> md.b ${loadaddr} 30

             0    2    4    6    8    A    C    E
800000020    d101 0060 0000 0000 0000 0000 0000 0000
800000030    0000 0000 0000 0000 0014 0000 0000 0000
800000040    00e6 0500 0000 0000 0900 0000 0000 0000
```

10.4.2 Configure Kernel Authentication

Kernel image authentication is provided by U-Boot, using the upstream verified boot method. This method uses an .its file, which defines a dtb/kernel configuration and a signing scheme. The following commands are given as if from a working directory which contains the Linux Kernel and U-Boot repository directories.

1. Create a directory in which to store the generated private key and certificate

   ```
   mkdir kernel_keys
   ```

2. Generate the RSA2048 key pair

```
openssl genrsa -out kernel_keys/boot_key.key 2048
openssl req -batch -new -x509 -key kernel_keys/boot_key.key -out kernel_keys/boot_key.crt
```

Note that the names of the keys and the folder holding the keys are given as an example. If a different name is used for the keys or the folder, the private key and public key names must match. Moreover, the name of the private key must be used for the `key-name-hint` field in the .its file.

3. Create an .its file for Linux

The .its file will be used to create a FIT image, which will contain the kernel and dtb (optionally, the rootfs). An example .its file, for signing kernel and dtb configurations:

```
        /dts-v1/;
        / {
                description = "kernel+dtb/fdt fit image";
                #address-cells = <1>;
                images {
                        kernel@1 {
                                description = "kernel image";
                                data = /incbin/("<path>/<to>/Image");
                                type = "kernel";
                                arch = "arm64";
                                os = "linux";
                                compression = "none";
                                load = <load address>;
                                entry = <entry address>;
                                kernel-version = <1>;
                                hash@1 {
                                        algo = "sha1";
                                };
                        };
                        fdt@1 {
                                description = "dtb blob";
                                data = /incbin/("<path>/<to>/<board>.dtb");
                                type = "flat_dt";
                                arch = "arm64";
                                compression = "none";
                                fdt-version = <1>;
                                hash@1 {
                                        algo = "sha1";
                                };
                        };
                };
                configurations {
                        default = "conf@1";
                        conf@1 {
                                kernel = "kernel@1";
                                fdt = "fdt@1";
                                signature@1 {
                                        algo = "sha1,rsa2048";
                                        key-name-hint = "<key_name>";
```

```
                                        sign-images = "kernel", "fdt";
                                };
                        };
                };
        };
```

NOTE

Make sure the `data` field for both `fdt@1` and `kernel@1` are updated when using a different configuration.

This .its file is provided as an example for defining a configuration (kernel and dtb combination), which is hashed and signed using the private key. Alternatively, the kernel image and dtb can be signed, by moving the signature block from the configuration block to both the kernel image and fdt blocks, replacing the hash blocks. For an example of this, see doc/uImage.FIT/sign-images.its.

The .its file should be placed in the Linux Kernel directory.

4. Build the mkimage tool

   From the U-Boot directory, run:

```
make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- tools
```

5. Build Verified Boot components

   Before beginning, make sure you have followed sections 3.2.1 and 3.2.2.2. See section 10.4 for more details.

   If you haven't built U-Boot before (i.e. if you haven't completed section 10.4.1.1), start by running:

```
make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- <board>_defconfig
```

   If you've already run the above command previously, you can skip it and continue by configuring the required options for Kernel authentication from `menuconfig`. If you want to authenticate both TF-A FIP and Linux, make sure you still have the `CONFIG_NXP_HSE_SUPPORT` and `CONFIG_NXP_HSE_FW_FILE` config options for FIP authentication enabled, from section 10.4.1.1.

   From the U-Boot directory, run:

```
make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- menuconfig
```

   In menuconfig, make sure the following options are enabled:

```
CONFIG_OF_CONTROL=y
CONFIG_FIT_SIGNATURE=y
CONFIG_NXP_HSE_SUPPORT=y
CONFIG_NXP_HSE_FW_FILE=<path>/<to>/<HSE>/<directory>/<hse_fw_file>.bin.pink
```

After saving the configuration, from the root of the U-Boot directory, run:

```
make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu-
```

6. Build the ARM Trusted Firmware

   TF-A can be built normally, by following chapter 3.2.4, or with support for HSE Secure Boot, by following chapter 10.4.1.

7. Build the Linux Kernel

   This step is optional, as you can use the Kernel Image built in section 3.2.3.

   Otherwise, from the linux directory, run:

```
make ARCH=arm64 CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- \
        <soc_name>_defconfig
make ARCH=arm64 CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu-
```

   See section 3.2.3 for more information on building the Linux Kernel.

8. Sign the kernel image and pack it into a FIT image

   After the kernel has been compiled, from the U-Boot directory, run:

```
tools/mkimage -f ../linux/<its-filename>.its
              -K ../arm-trusted-firmware/build/<board>/release/fdts/<dtb-filename>.dtb
              -k ../kernel_keys
              -r <itb-filename>.itb
```

   The output of the mkimage command should display the parts of the FIT image: the kernel, dtb and configuration. The kernel and dtb have a hash value, while the configuration has a signed hash, if using the configuration signing method. Otherwise, the kernel and fdt have a signed hash.

   The options for mkimage specify:

```
        -f    image tree source file describing the contents and structure
        -K    public key destination; specify the dtb in which to write the
              public key/certificate, used for authentication
        -k    private key source; specify directory where the keys are stored;
              private key is used for signing
        -r    required; specify that keys are required, cannot boot if
              signature cannot be verified
```

   The resulting FIT image must be copied to the boot partition of the SD Card.

9. Rebuild the ARM Trusted Firmware

   The ARM Trusted Firmware must be rebuilt to include the DTB in which the key was placed in the previous step.

10. Boot the board and stop at U-Boot command line

    Copy the FIT image from the SD Card to memory:

```
=> fatload mmc 0:1 <load-address> <fit-image-name>
```

    U-Boot must use the bootm command to boot, not booti. Trying to boot using booti displays the `Bad Linux ARM64 Image magic!` error, since the header for the FIT image is not recognized by booti.

```
=> run mmcargs
=> bootm <load-address>
```

    To check if the FIT image loaded on the board is correct, run:

```
iminfo
```

    The command checks the hashes of the image, dtb, and configuration, but does not check if the FIT image is properly signed.

10.5 HSE Linux Driver

This chapter contains general information about the HSE crypto driver, which provides support for offloading cryptographic operations to HSE's dedicated coprocessors through the kernel crypto API.

10.5.1 Prerequisites

In order to use the HSE cryptographic offloading capabilities, HSE firmware must be installed and the key catalogs must be formatted before the kernel driver is probed. The AES-256 and HMAC groups from the RAM key store are used.

Following the steps described in section 10.2.9, *hse-secboot* must be used to format the HSE key catalogs:

```
./hse-secboot -f -d <device>
```

The options represent:

1. -f - format HSE key catalogs
2. -d - specifies boot media, e.g. /dev/mmcblk0

This can be used to format the key catalog without needing to boot in secure mode on the target device. Please verify that the RAM key group IDs and sizes are defined appropriately to your key catalog configuration:

- `CONFIG_CRYPTO_DEV_NXP_HSE_AES_KEY_GROUP_ID`
- `CONFIG_CRYPTO_DEV_NXP_HSE_AES_KEY_GROUP_SIZE`
- `CONFIG_CRYPTO_DEV_NXP_HSE_HMAC_KEY_GROUP_ID`

- CONFIG_CRYPTO_DEV_NXP_HSE_HMAC_KEY_GROUP_SIZE

A single Messaging Unit instance is used by the driver to communicate with HSE. The user can select a MU instance (out of a total of 4) in the configuration options (all MU instances have to be enabled beforehand in HSE firmware, except MU0) and must ensure exclusive access to it. In the crypto driver the MU is used in interrupt mode, so sharing channels of the same instance across multiple users is not possible.

10.5.2 Supported Algorithms

This driver currently supports the following crypto operations:

- Hashing: SHA1, SHA2
- Symmetric Key Ciphering: AES-CTR, AES-CBC, AES-ECB, AES-CFB, AES-OFB
- Message Authentication Codes: HMAC-SHA1, HMAC-SHA2
- Authenticated Encryption with Associated Data: AES-GCM
- True Random Number Generator: PTG.3 class

10.5.3 Configuration

The following Kconfig options are available:

- Messaging Unit (MU) Instance (CONFIG_CRYPTO_DEV_NXP_HSE_MU):

  There are 4 Messaging Unit instances available for interfacing application processor subsystems with HSE and the user can configure which one is used by the Linux driver for sending service requests and receiving replies. The MU instance indicated here shall be used in interrupt mode and therefore should be entirely reserved for the Linux crypto driver. Sharing an instance with another driver or application shall result in requests being dropped.

- Message Digest Support (CONFIG_CRYPTO_DEV_NXP_HSE_AHASH):

  Enables hash and hash-based MAC offloading to HSE.

- Symmetric Key Cipher Support (CONFIG_CRYPTO_DEV_NXP_HSE_SKCIPHER):

  Enables symmetric key cipher offloading to HSE.

- AuthEnc and AEAD Support (CONFIG_CRYPTO_DEV_NXP_HSE_AEAD):

  Enables authenticated encryption and AEAD offloading to HSE.

- Hardware RNG support (CONFIG_CRYPTO_DEV_NXP_HSE_RNG):

  Enables hardware true random number generation via HSE.

- Entropy cache maximum size (CONFIG_CRYPTO_DEV_NXP_HSE_RNG_CACHE):

  Total size of driver entropy cache. Used to improve request latency.

- RAM Key Catalog AES Group Configuration:

  AES 256-bit Key Group ID within RAM Key Catalog (CRYPTO_DEV_NXP_HSE_AES_KEY_GROUP_ID):

  This option specifies which key group is used by the driver for programming the AES 256-bit keys into HSE, depending on how the RAM catalog was initialized by the firmware.

  Number of key slots in the AES 256-bit Key Group (CRYPTO_DEV_NXP_HSE_AES_KEY_GROUP_SIZE):

  This option specifies the maximum number of keys that can be stored in the AES 256-bit key group.

- RAM Key Catalog HMAC Group Configuration:

  HMAC 1024-bit Key Group ID within RAM Key Catalog (CRYPTO_DEV_NXP_HSE_HMAC_KEY_GROUP_ID):

  This option specifies which key group is used by the driver for programming HMAC 1024-bit keys into HSE, depending on how the RAM catalog was initialized by firmware.

  Number of key slots in the HMAC 1024-bit Key Group (CRYPTO_DEV_NXP_HSE_HMAC_KEY_GROUP_SIZE):

  This option specifies the maximum number of keys that can be stored in the HMAC 1024-bit key group.

- Debug Information for HSE Crypto Driver (CONFIG_CRYPTO_DEV_NXP_HSE_DEBUG):

  Enables printing driver debug messages to the kernel log.

10.6 HSE Crypto Driver in OP-TEE

The HSE Crypto Driver in OP-TEE provides cryptographic offloading capabilities for the OP-TEE Core and Trusted Applications (TAs). The design and configuration of the HSE OP-TEE Driver resembles the HSE Linux Driver. However, the HSE OP-TEE Driver is still in an early phase and does not provide all the cryptographic capabilities of the Linux Driver. Its usage is oriented towards demonstrating how HSE's cryptographic and key management services could be useful for Trusted Applications employing the Secure Storage feature of OP-TEE.

10.6.1 Build Instructions

1. Build optee_os with HSE Crypto Driver Support by adding the **CFG_CRYPTO_DRIVER=y** make parameter:

   ```
   make CROSS_COMPILE64=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- ARCH=arm \
   PLATFORM=s32 PLATFORM_FLAVOR=s32g3 CFG_CRYPTO_DRIVER=y
   ```

   Please refer to Building the optee_os component of OP-TEE section for more information on manually building the optee_os component.

2. Follow the instructions in the Configure TF-A FIP Authentication section to build a FIP image containing the HSE Firmware binary. It is not mandatory to boot in Secure Boot mode in order to use the HSE Crypto Driver in OP-TEE, so the FIP signing and authentication operations can be skipped. Make sure U-Boot and ATF are built

with the specific HSE parameters as provided in the referenced section at the beginning of the paragraph. The optee_os component must be included in the FIP image by using the **BL32** and **BL32_EXTRA1** make parameters in the ATF build command:

```
make ARCH=aarch64 PLAT=<board> BL33=<path>/<to>/<u-boot-dir>/u-boot-nodtb.bin
BL32=<path>/<to>/tee-header_v2.bin \
BL32_EXTRA1=<path>/<to>/tee-pager_v2.bin SPD=opteed \
HSE_SECBOOT=1
```

NOTE: It is not currently possible to build OP-TEE with HSE Crypto Driver support in Yocto. For a complete OP-TEE image with HSE Crypto Driver support, one would have to:

1. Generate a fip.s32 binary following the above steps.
2. Build a Yocto image with OP-TEE support, as instructed in the Building OP-TEE-Enabled Image and BSP Yocto Build sections.
3. Burn these two images onto an SD Card:

```
sudo dd if=./fsl-image-base-s32g399aevb3.sdcard of=${DEVSD} bs=1M && sync
sudo dd if=fip.s32 of=${DEVSD} conv=notrunc,fsync seek=512 skip=512 \
oflag=seek_bytes iflag=skip_bytes
```

Please respect the order of the `dd` commands, as the generated fip.s32 must overwrite the `fsl-image-base's` included fip.

10.6.2 Configure HSE OP-TEE Crypto Driver

Being yet another HSE Driver, the HSE OP-TEE Crypto Driver uses the same types of resources as the Linux Driver, i.e. a Messaging Unit and the RAM Key Catalog. The driver implements the symmetric cipher crypto API and OP-TEE can be instructed to use this implementation in its cipher operations. The configuration is done at build time by adding the following parameters in the build command:

- **CFG_HSE_AES_KEY_GROUP_ID** & **CFG_HSE_AES_KEY_GROUP_SIZE**
  CFG_HSE_HMAC_KEY_GROUP_ID & **CFG_HSE_HMAC_KEY_GROUP_SIZE**
  CFG_HSE_SHARED_SECRET_KEY_ID & **CFG_HSE_SHARED_SECRET_KEY_GROUP_SIZE**

  Each pair selects the Group ID and, respectively, the Group Size for a specific type of key. Their default values are synced with the current format of the Key Catalog. If one changes the catalog, these values must be updated. Example of setting the AES pair to different values:

```
make ... CFG_CRYPTO_DRIVER=y CFG_HSE_AES_KEY_GROUP_ID=1 \
CFG_HSE_AES_KEY_GROUP_SIZE=11
```

- **CFG_CRYPTO_DRV_CIPHER**

  Enables/Disables the HSE Crypto Cipher Driver. It's enabled by default, but can be disabled using:

```
make ... CFG_CRYPTO_DRV_CIPHER=n
```

By default, the HSE OP-TEE Crypto Driver uses the third MU. The MU selection is done in the TF-A software component using the dedicated Device Tree fdts/s32cc-crypto.dtsi. The `status` property in the MU nodes determines which MU would be used by the HSE OP-TEE Crypto Driver, with `status = "okay"` value making the selection. Please note that only one MU is used by the driver. Hence, if multiple MU nodes have the property value set to `status = "okay"`, only the first node in the device tree containing this attribute value is registered by the driver.

Example of MU Device Tree Node:

```
mu2b@40212000 {
        compatible = "nxp,s32cc-hse";
        reg = <0x0 0x40212000 0x0 0x1000>,
              <0x0 0x22c02000 0x0 0x1000>;
        reg-names = "hse-regs",
                    "hse-desc";
        interrupts = <GIC_SPI 109 IRQ_TYPE_EDGE_RISING>, /* GIC 141 */
                     <GIC_SPI 110 IRQ_TYPE_EDGE_RISING>, /* GIC 142 */
                     <GIC_SPI 111 IRQ_TYPE_EDGE_RISING>; /* GIC 143 */
        interrupt-names = "hse-ack",
                          "hse-rx",
                          "hse-err";
        status = "okay";
};
```

10.6.3 Enable the OP-TEE Crypto Driver

Upon booting a freshly built FIP Image, the HSE OP-TEE Driver fails during its first initialization.

This happens because the Key Catalog is not yet formatted at that point. The only way to format the catalog is to build the `hse-secboot` example, as instructed in the paragraph 10.2.9, and run it in the Linux Userspace. Reset the board once the operation is complete:

```
hse-secboot -f -d <device>
reboot -n
```

Once the Key Catalog is formatted and the board resets, there should be no errors displayed by the HSE OP-TEE Driver during any subsequent initializations and the driver should function properly.

10.6.4 Features

This driver currently provides the following features:

- Symmetric Key Ciphering: AES-CBC, AES-ECB
- True Random Number Generator: PTG.3 class
- Hardware Unique Key (HUK) derived from a HSE Shared Secret Key

11 DDR

11.1 Integration of DDR Tool generated code into ATF

Below are the steps for integrating the code generated by the NXP DDR Tool into the BSP.
ATF is compatible with the code generated using **S32 Design Studio 3.5** with the **Real-Time Drivers 4.4 Version 4.0.0** extension package.

The default frequency used by S32G3 is 800 MHz. The clock frequency can be adjusted using the following defines within include/dt-bindings/clock/s32gen1-clock-freq.h

```
#define S32GEN1_DDR_PLL_VCO_FREQ (1600 * MHZ)
#define S32GEN1_DDR_FREQ (800 * MHZ)
```

Notice that the above values from ATF are not the default configuration values in the DDR Tool. Please customize the `Clock Cycle Freq` and `DDR_CLK frequency` fields in the DDR Tool to match the above defines from ATF.

After generating the DDR Code with the DDR Tool, it can be externally linked in ATF by using the `CUSTOM_DDR_DRV` parameter for `make`. It must be set to a path (relative to the ATF root folder) where all the DDR source and header files are, in the state they were generated by DDR Tool.

The first step is to prepare the code for the ATF, which is done by calling the `import_ddr_drv` target with `make`. This step is not required for code generated with a DDR Tool version starting with the one included in the **Real-Time Drivers 4.4 Version 4.0.0** extension package. Then, the compilation is done as described in section Building the ARM Trusted Firmware, but also with the `CUSTOM_DDR_DRV` parameter set.

```
$ CUSTOM_DDR_FOLDER="$(realpath --relative-to=/path/to/arm-trusted-firmware /path/to/ddr/generated/folder)"
$ make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- \
       ARCH=aarch64 PLAT=<plat> BL33=<path-to-u-boot-nodtb.bin> \
       CUSTOM_DDR_DRV="$CUSTOM_DDR_FOLDER" import_ddr_drv
$ make CROSS_COMPILE=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- \
       ARCH=aarch64 PLAT=<plat> BL33=<path-to-u-boot-nodtb.bin> \
       CUSTOM_DDR_DRV="$CUSTOM_DDR_FOLDER"
```

11.2 Checking if Inline ECC Feature is Enabled

Checking if the Inline ECC feature is enabled can be done at runtime, by inspecting the *ECC_MODE* bits of the *ECCCFG0* DDR Controller register.

This can be done in U-Boot by running the following command:

```
=> md 0x403C0070 0x1
403c0070: 033f7f54                              T.?.
```

To confirm that ECC is enabled, the last hex digit of the read value should be 0x4 or 0x5. For other values, ECC is not enabled.

If the check needs to be done in Linux, it can be done using the *devmem2* tool:

```
root@s32g399aevb3:~# devmem2 0x403c0070
/dev/mem opened.
Memory mapped at address 0x7f877a3000.
Read at address  0x403C0070 (0x7f877a3070): 0x033F7F54
```

The same value checking logic as for U-Boot can be applied here to confirm if ECC is enabled or not.

12 LLCE Support

The NXP Low Latency Communication Engine (LLCE) is an autonomous accelerator designed to offload communication tasks on the CAN, LIN and Flexray Protocols. It is a self-contained module, separate from the application cores that would run applications (host cores).

12.1 Prerequisites

LLCE drivers rely on LLCE firmware, which is a separate NXP product available through https://www.nxp.com/. For more information, and to find out how to obtain LLCE Firmware, please contact NXP marketing department.

NOTE

LLCE Logging capability is only available using the advanced version of the LLCE firmware.

All Linux LLCE drivers are compatible with LLCE firmware:

- Product ID: S32G_LLCE_1.0.4
- Firmware archive: S32G_LLCE_GATEWAY_1.0.4_D2204.exe

12.2 LLCE CAN

Linux BSP includes a set of drivers that help to expose all LLCE interfaces as SocketCAN interfaces in Linux. This set includes drivers responsible for:

1. LLCE firmware loading
2. LLCE mailbox management
3. LLCE CAN interfaces management

12.2.1 Enabling LLCE CAN from Yocto

LLCE CAN support can be enabled directly from Yocto using the following steps:

1. Download LLCE firmware (see 12.1)
2. Unpack the package to a folder of your choice and locate the following files in subdirectory `firmware/llce_bin/s32g3/bin/ghs/enablement`, which are required by Yocto:
    - dte.bin
    - frpe.bin
    - ppe_tx.bin
    - ppe_rx.bin
3. Enable LLCE CAN feature and specify the location where firmware binaries can be found. You can copy the .bin files to a common firmware location (e.g. `/path/to/firmware/binaries/folder`) or use unpack subdirectory above directly. This step requires to add the following lines in `<builddirectory>/conf/local.conf` file.

```
DISTRO_FEATURES:append = " llce-can"
NXP_FIRMWARE_LOCAL_DIR = "/path/to/firmware/binaries/folder"
```

12.2.2 Usage example

1. Configure `llcecan0` and `llcecan14` interfaces

```
root@s32g399aevb3:~# ip link set up llcecan0 type can bitrate 500000 dbitrate 5000000 fd on
root@s32g399aevb3:~# ip link set up llcecan14 type can bitrate 500000 dbitrate 5000000 fd on
```

2. Capture everything

```
root@s32g399aevb3:~# candump any,0:0,#FFFFFFFF >log &
```

3. Send a packet using `llcecan14`

```
root@s32g399aevb3:~# cansend llcecan14 15575555##3001122334455667788 99aabbcceedd112233
```

More information on how to send / receive packets through a SocketCAN interface in Linux can be found in section 13.0.1.

12.2.3 Statistics

LLCE CAN driver offers a set of statistics through `ip` and `ethtool` tools. Using `ip`, one can check the RX/TX counters and the number of CAN protocol errors.

```
root@s32g399aevb3:~# ip -details -statistics link show llcecan1
8: llcecan1: <NOARP,UP,LOWER_UP,ECHO> mtu 72 qdisc pfifo_fast state UP mode DEFAULT group
    default qlen 1000
    link/can  promiscuity 0 minmtu 0 maxmtu 0
    can <FD> state STOPPED restart-ms 1
          bitrate 500000 sample-point 0.875
          tq 12 prop-seg 69 phase-seg1 70 phase-seg2 20 sjw 1
          llce_can: tseg1 4..256 tseg2 4..128 sjw 1..128 brp 1..512 brp-inc 1
          dbitrate 5000000 dsample-point 0.750
          dtq 12 dprop-seg 5 dphase-seg1 6 dphase-seg2 4 dsjw 1
          llce_can: dtseg1 2..32 dtseg2 2..16 dsjw 1..16 dbrp 1..32 dbrp-inc 1
          clock 80000000
          re-started bus-errors arbit-lost error-warn error-pass bus-off
          0          2116       0          0          0          0
          numtxqueues 1 numrxqueues 1 gso_max_size 65536 gso_max_segs 65535
    RX: bytes  packets  errors  dropped overrun mcast
```

```
    0          0        0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    7936       125      2300    0       0       0
```

A wider set of errors is reported using `ethtool`:

```
root@s32g399aevb3:~# ethtool -S llcecan1
NIC statistics:
     txack_fifo_full: 0
     rxout_fifo_full: 0
     hw_fifo_empty: 0
     hw_fifo_full: 0
     sw_fifo_empty: 0
     sw_fifo_full: 0
     mb_notavailable: 0
     short_mb_notavailable: 0
     bcan_frz_exit: 0
     bcan_sync: 0
     bcan_frz_enter: 0
     bcan_lpm_exit: 0
     bcan_srt_enter: 0
     bcan_unknown_error: 0
     bcan_ackerr: 47013
     bcan_crcerr: 0
     bcan_bit0err: 2013
     bcan_bit1err: 5
     ...
```

12.3 CAN Logger

The LLCE CAN Logger driver receives CAN frames and timestamps from LLCE hardware and logs them.

The driver is able to operate in two modes: along with Linux LLCE CAN driver or in scenarios where the LLCE CAN host management is part of an another software stack.

For the cases when the LLCE subsystem is managed by Linux, the following lines should be added in `<builddirectory>/conf/local.conf` file.

```
DISTRO_FEATURES:append = " llce-can llce-logger"
NXP_FIRMWARE_LOCAL_DIR = "/path/to/firmware/binaries/folder"
```

For the cases when the LLCE subsystem is managed by other software stacks, the added line should be:

```
DISTRO_FEATURES:append = " llce-logger"
```

12.3.1 Using the Driver

Upon inserting the module, a set of virtual CAN (vcan) interfaces will be created. Each `llcelogger` interface will capture the traffic received on its associated hardware interface. For example, on `llcelogger12` will be logged the traffic received on `llcecan12`.

NOTE: If the `llcelogger12` interface is down, the CAN frames are not read from the LLCE CAN Logger FIFO and the receipt of CAN frames on `llcecan12` will stall. To avoid the stalling of a LLCE CAN interface it is recommended to set up the corresponding LLCE CAN Logger interface.

```
root@s32g399aevb3:~# ip address show type can
2: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN group default qlen 10
    link/can
3: can1: <NOARP,ECHO> mtu 16 qdisc noop state DOWN group default qlen 10
    link/can
6: llcelogger0: <NOARP> mtu 16 qdisc noop state DOWN group default qlen 10
    link/can
7: llcelogger1: <NOARP> mtu 16 qdisc noop state DOWN group default qlen 10
    link/can
8: llcelogger2: <NOARP> mtu 16 qdisc noop state DOWN group default qlen 10
    link/can
...
```

By default, all LLCE Logger interfaces are in down state and not marked as virtual CAN interfaces. This can be changed using:

```
root@s32g399aevb3:~# ip link set up llcelogger0 type vcan
```

The logs along with the hardware timestamp can be inspected using:

```
root@s32g399aevb3:~#  candump -t a -H llcelogger0,0:0,#FFFFFFFF
  (2535568793.000000)  llcelogger0  029  [8]  01 02 03 04 05 06 07 08)
  (2535605966.000000)  llcelogger0  029  [8]  02 03 04 05 06 07 08 09
  (2535643006.000000)  llcelogger0  029  [8]  03 04 05 06 07 08 09 0A
  (2535680046.000000)  llcelogger0  029  [8]  04 05 06 07 08 09 0A 0B
  (2535717086.000000)  llcelogger0  029  [8]  05 06 07 08 09 0A 0B 0C
  (2535754260.000000)  llcelogger0  029  [8]  06 07 08 09 0A 0B 0C 0D
```

The first column from above output contains timestamps in '(seconds.nanoseconds)' format. In case of the LLCE Logger, the `seconds` field represents the number of cycles reported by LLCE. In above example '(2535568793.000000)' corresponds to 0x9721B599 cycles.

13 CAN Information

This chapter describes the steps to configure CAN and lists basic examples for CAN message transmission.

With respect to using FlexCAN on S32G3 EVB together with S32GRV-PLATEVB, first make sure that you have the following jumpers on the required position: J166 on 1-3 and 2-4, J155 on 3-5 and 4-6, J169 on 1-3 and 2-4.
The mapping of the hardware pins of the CAN interface to the userspace interface is as follows:

- FlexCAN_0 has the hardware interface J79 on PROCEVB associated to `can0` in userspace.
- FlexCAN_2 has the hardware interface J19 on PLATEVB associated to `can1` in userspace.
- FlexCAN_3 has the hardware interface J20 on PLATEVB associated to `can2` in userspace.

NOTE: For S32G399A RDB3 only `can0` and `can1` interfaces are available.

13.0.1 Example of sending a CAN message

- Set the CAN interfaces:
  - Set the bitrate of can0 and can1 interfaces to 500kbps:

```
ip link set can0 type can bitrate 500000
ip link set can1 type can bitrate 500000
```

  - Start the can interfaces like a standard net interface:

```
ip link set can0 up
ip link set can1 up
```

- Example of testing can1 transmission:

```
cansend can0 011#22335566778899
```

- Example of testing can0 reception:

```
candump can1
  can1  123   [8]  0A AA AA AA BB BB BB BB
  can1  123   [8]  0A AA AA AA BB BB BB BB
  can1  123   [8]  0A AA AA AA BB BB BB BB
  can1  123   [8]  0A AA AA AA BB BB BB BB
  can1  123   [8]  0A AA AA AA BB BB BB BB
  can1  123   [8]  0A AA AA AA BB BB BB BB
  can1  123   [8]  0A AA AA AA BB BB BB BB
```

14 HS400 Support

14.1 MMC HS400 support

HS400 is only supported if the MMC memory runs at 1.8V. Before attempting to enable this option, please ensure the board is configured accordingly. For example, on S32G-PROCEVB-S, setting `J110` and `J23` on 1-2, switches the eMMC memory to 1.8V. Afterwards, use the steps below to enable HS400.

NOTE: Do not use the steps below if the memory that the controller talks to (i.e. as defined by the position of the J50 jumper on S32G-PROCEVB-S) is not actually supplied at 1.8V, as it may lead to undefined behavior.

NOTE: The HS400 mode has only been validated on the S32G-PROCEVB-S board.

14.1.1 Enabling HS400 support in U-Boot

Go to the board specific device-tree in TF-A (e.g. fdts/s32g399a-evb3.dts) and amend the usdhc node as follows:

```
&usdhc0 {
    /delete-property/ no-1-8-v;
};
```

If building with yocto, see chapter Building the Linux BSP using YOCTO for more information about patching various components.

14.1.2 Enabling HS400 in linux

Go to the board specific device-tree and amend the usdhc node as follows:

```
&usdhc0 {
    /delete-property/ no-1-8-v;
};
```

If building with yocto, see chapter Building the Linux BSP using YOCTO for more information about patching various components.

Alternatively, you can enable it at runtime, from the U-Boot environment. Executing the following command at the U-Boot prompt will enable HS400/ES in Linux. Executing the 'saveenv' command makes the changes persistent across multiple boots.

```
setenv fdt_override "run fdt_enable_hs400es; ${fdt_override}"
saveenv
```

14.2 MMC HS400 Enhanced Strobe support

The HS400ES mode is selected automatically if HS400 is enabled and the connected memory supports HS400ES.

NOTE: None of the supported platforms has an eMMC which supports HS400ES. This feature has been validated on a modified S32G-PROCEVB-S board with a different memory.

15 Temperature Monitoring Unit

15.1 Reading Temperatures from the Thermal Monitoring Unit

The Thermal Monitoring Unit (TMU) allows to read the SoC temperature from 3 different sites.

The simplest way to access the temperatures is to use the lm-sensors package (available for the "fsl-image-auto Yocto recipe").

For example:

```
root@s32g399aevb3:~# sensors
s32tmu-isa-0000 Adapter: ISA adapter
Immediate temperature for site 0:  +45.0 C
Average temperature for site 0:    +45.0 C
Immediate temperature for site 1:  +44.0 C
Average temperature for site 1:    +44.0 C
Immediate temperature for site 2:  +46.0 C
Average temperature for site 2:    +46.0 C
```

The sensors command is actually using the sysfs entries available at path: */sys/devices/platform/soc/400a8000.tmu/hwmon/hwmon0/tempX_input*, where X takes values from 1 to 6.

The sysfs entries at this path contain the temperature value multiplied by 1000, as expected by the standard Linux API.

The associated label found at */sys/devices/platform/soc/400a8000.tmu/hwmon/hwmon0/tempX_label* explains if tempX_input is an immediate temperature or an average one.

16 SoC Level Time Source

16.1 General Information

This module provides a global timestamp which can be accessed from the other cores of the SoC, so that events on different CPUs of the SoC can be globally ordered.

For this purpose, we use a System Timer Module (STM), 32-bits long. This type of timer has the advantage that its registers can be accessed at SoC level, hence providing global access. The input clock signal for the timer has the frequency of ~133.33 MHz. It can also be obtained from */sys/kernel/debug/clk/stm_module/clk_rate*. In order to obtain rarer overflows, a clock divider of 256 was used. This results in a resolution of ~1.9 μs and an overflow period of ~137 min.

16.2 Kernel Configuration

In order to obtain the module for this driver you need to set CONFIG_FSL_GLOBAL_TIME_STM=m in `s32cc_defconfig`
You need to activate the STM2 in device tree. This can be done by simply adding the following lines to arch/arm64/boot/dts/freescale/s32cc.dtsi before compiling.

```
&stm2 {
        status = "okay";
};
```

Refer to subsection Patch the kernel to apply the above changes.
Generate the new Yocto image by rerunning `bitbake <imagename>` and deploy it.

16.3 Userspace Interface

The STM can be used after inserting */lib/modules/5.15.85-rt/kernel/drivers/clocksource/fsl_global_time.ko*.
This driver exposes two sysfs entries as userspace interface:

- */sys/devices/platform/40124000.stm/value* - the current timestamp
- */sys/devices/platform/40124000.stm/init* - start/stop timer command/status

16.4 Supported commands

```
Starting timer:         echo 1 > /sys/devices/platform/40124000.stm/init
Reading timer value:    cat /sys/devices/platform/40124000.stm/value
Stopping timer:         echo 0 > /sys/devices/platform/40124000.stm/init
```

16.5 Accessing Time Source and Control from Outside Linux

In order to access the driver from outside Linux (say from real-time cores or the hardware accelerators) you need to read the corresponding counting register (CNT) and read/write the corresponding enable bit (TEN bit in CR register) of STM2 (System Timer Module 2), as described in the Reference Manual for S32G.

17 Switch the Linux System Timer from Generic Timer to PIT timer

The Linux kernel can switch the timers used for clocksource and clockevents as base. For a proper events timming/system timming, a timer should be assigned per core. S32G3 has only two PIT timers and two STM timers.

To enable the PIT/STM, you will have to follow the procedure:

- Select which core will use the PIT/STM timer.
  In arch/arm64/boot/dts/freescale/s32cc.dtsi can be found one entry for each PIT/STM timer:

```
cpus {
        #address-cells = <2>;
        #size-cells = <0>;

        pit0: pit@40188000{
                compatible = "fsl,s32gen1-pit";
                reg = <0x0 0x40188000 0x0 0x3000>;
                interrupts= <0 53 4>;
                clocks = <&clks S32GEN1_SCMI_CLK_PIT_MODULE>;
                clock-names = "pit";
                cpu = <0>;
                status = "disabled";
        };
```

  “cpu” field indicates which virtual cpu will use the timer module. Any value between 0 and NR_CPUS can be used. NOTE: The design of the STM module makes it unsuitable as clocksource so it is recommended not to be used for the master core.
- Enable PIT from menuconfig:
  Location:
  -> Platform selection
  -> Freescale S32 family
  -> PIT timer used as system timer
  -> PIT used as system timer
- Enable STM from menuconfig:
  Location:
  -> Platform selection
  -> Freescale S32 family
  -> STM timer used as system timer
  -> STM used as system timer

18 SPI Slave Support

18.1 SPI Slave Support in Linux

The DSPI controller can be configured to run in slave mode if `spi-slave` property is added in SPI node from dts.

The SPI controller node in dts can define both master and slave pin control configuration. The selection between them will be done based on pin control configuration names. The `default` pin control configuration will be loaded first and will be used by the master. If the controller is configured as slave, the driver will try to find `slave` pin control configuration and apply it if found. If not found, default will be used.

Example:

```
&spi1 {
        pinctrl-0 = <&pinctrl0_dspi1>, <&pinctrl1_dspi1>;
        pinctrl-1 = <&pinctrl0_dspi1_slave>, <&pinctrl1_dspi1_slave>;
        pinctrl-names = "default", "slave";
        #address-cells = <0>;
        #size-cells = <0>;
        spi-slave;
        status = "okay";

        slave {
                        compatible = "rohm,dh2228fv";
        };
};
```

Use `rohm,dh2228fv` compatible for binding the spidev driver to the device.
Information on spidev can be found on:

https://www.kernel.org/doc/Documentation/spi/spidev

The slave will run in DMA mode otherwise the driver is too slow to feed TX FIFO and underflow occurs frequently. DMA can work only with 8 and 16 bits per word. 32bits per word is not supported.

The slave supports a maximum of 1024 words per transfer (this is the maximum of the internal buffer used by the dspi driver). For more than 1024 words per transfer, the driver splits the transfer in several DMA transfers. Between DMA transfers, underflow occurs in the slave and the last transferred word is duplicated.

19 SAR-ADC Driver

19.1 Using the SAR-ADC in U-Boot

The Linux BSP U-Boot driver allows the user to operate the on-chip 12-bit analog-to-digital converter (SAR-ADC) in single and multi-channel conversion mode.

The SAR-ADC modules can be probed in U-Boot by running:

```
=> adc list
- adc@401f8000
- adc@402e8000
```

NOTE: The above listed module names have the following correspondence:

adc@401f8000	saradc0
adc@402e8000	saradc1

1. One-shot single channel conversion mode:

```
=> adc single adc@401f8000 0
0
=> adc single adc@401f8000 1
2518
=> adc single adc@401f8000 3
4095
```

2. One-shot multi channel conversion mode:

```
=> adc scan adc@401f8000
[00]: 1
[01]: 2523
[02]: 1985
[03]: 4095
[04]: 2180
[05]: 2135
[06]: 2038
[07]: 2130
```

In order to obtain the voltage in millivolts, this value must be multiplied with the scale of value: 0.439453125. For example, for channel 1, the result is 2518 x 0.439453125 ~ 1106.54 mV.

19.2 Using the SAR-ADC in Linux

The Linux BSP Linux driver allows the user to operate the on-chip 12-bit analog-to-digital converter (SAR-ADC) through an IIO (Industrial Input/Output) driver.

Linux BSP supports two modes of operation for the SAR-ADC:

1. One-shot conversions in normal mode

   The files needed for reading converted values from the eight external ADC channels are located in the sysfs attributes directory of the IIO device. The directory path is `/sys/bus/iio/devices/iio:deviceX` where `X` is the IIO index and has a value between 0 and 255. In case of S32-GEN1 platforms, most probably, there will be two IIO devices with index 0 and 1. If you see more directories, find the one corresponding to the ADC by reading the name file located inside:

   ```
   $ cat /sys/bus/iio/devices/iio:device0/name
   401f8000.adc
   $ cat /sys/bus/iio/devices/iio:device1/name
   402e8000.adc
   ```

   Connect a DC power supply between the ADC ground and one of the first 8 pins. The voltage applied must have a value between 0 and 1.8V. Read the `in_voltageX_raw file`, where `X` is the channel number. For example:

   ```
   $ cat /sys/bus/iio/devices/iio:device0/in_voltage0_raw
   2738
   ```

   In order to obtain the voltage in millivolts, this value must be multiplied with the scale in `in_voltage_scale`

   ```
   cat /sys/bus/iio/devices/iio:device0/in_voltage_scale
   0.439453125
   ```

   In this case, the result is 2738 x 0.439453125 ~ 1203.22 mV.

2. Scan (continuous) conversions in normal mode

   To read SAR_ADC data continuously you need to create and bind a sysfs trigger and then enable the channels (which are used) and the buffer.

   The following steps are a setup example for reading data from channels 0 and 3 in scan (continuous) mode, from SAR_ADC0 device:

   (a) Create a new sysfs trigger:

   ```
   $ echo 0 > /sys/bus/iio/devices/iio_sysfs_trigger/add_trigger
   ```

   (b) Bind device with the newly created trigger:

   ```
   $ cat /sys/bus/iio/devices/trigger0/name > /sys/bus/iio/devices/iio:device0/trigger/current_trigger
   ```

   (c) Enable the channels in use (you can enable any combination of the channels)

   ```
   $ echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage0_en
   $ echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage3_en
   ```

(d) The next step is to enable the buffer and optionally set up a buffer length. You can do this manually, or you can use the `iio_generic_buffer` userspace application, located under kernel sources at tools/iio/iio_generic_buffer.c

i. Enable buffer and read data manually:

```
$ echo 100 > /sys/bus/iio/devices/iio:device0/buffer/length (optional)
$ echo 1 > /sys/bus/iio/devices/iio:device0/buffer/enable
$ hexdump -e '"iio0 :" 8/2 "%04x " "\n"' /dev/iio:device0 | head -20
```

NOTE: ADC data can be read from `/dev/iio:device0`. In the above example we dump only the first 20 lines to the console. (20 read operations from channels 0 and 3). Data read from `/dev/iio:device0` must be multiplied with the scale in `in_voltage_scale`, in order to obtain the voltage in millivolts.
If you enable and read the adc data manually, you should disable buffer mode when you finish using the adc device, as follows:

```
$ echo 0 > /sys/bus/iio/devices/iio:device0/buffer/enable
```

NOTE: You can re-enable the data capture by writing value `1` to `/sys/bus/iio/devices/iio:device0/buffer/enable` at any later time.

ii. Using the iio_generic_buffer application: The `iio_generic_buffer` application can be compiled as follows:

```
$ make ARCH=arm64 -C <kernel-src-dir>/tools/iio
```

The `iio_generic_buffer` application performs all the ADC channel `enable` and `disable` actions for you.

```
root@s32g399aevb3:~# ./iio_generic_buffer -N 0 -T 0
iio device number being used is 0
iio trigger number being used is 0
/sys/bus/iio/devices/iio:device0 sysfstrig0
0.439453 1799.560547
0.439453 1799.560547
0.439453 1799.560547
0.439453 1799.560547
0.439453 1799.560547
0.439453 1799.560547
0.878906 1799.560547
0.439453 1799.560547
0.439453 1799.560547
0.439453 1799.560547
0.439453 1799.560547
0.439453 1799.560547
0.000000 1799.560547
0.439453 1799.560547
```

In the above example `-N` is the device number (0 -> SAR_ADC0 or 1 -> SAR_ADC1) and `-T` is the trigger number (0 in this case, as created above in step `a`).

NOTE: If you want to use all SAR_ADC channels, please do not use '-a' parameter of the `iio_generic_buffer` application because it will also try to enable the timestamp soft channel, which is not the desired behaviour. Instead, you should enable all of SAR_ADC channels manually from `scan_elements/in_voltageX_en`.

NOTE: By using `iio_generic_buffer` there is no need to multiply the output values with the scale in `in_voltage_scale`.

Check the Documentation/ABI/testing/sysfs-bus-iio file inside the Linux source code for a full reference of the standard IIO device attributes. Currently, the driver only supports software triggers, by performing one-shot or continuous conversions in normal mode for one or multiple channels at a time.

NOTE: On S32G399A RDB3 on each SAR_ADC channel there is a resistor divider circuit, which increases the S32G399A RDB3 voltage detection range (from 0-1.8V to 0-3.6V). Therefore, on S32G399A RDB3 platform, the SAR_ADC Linux driver outputs the voltage read after the resistor divider circuit, which is half of the voltage applied.

NOTE: Channel 5 of ADC0 is not available for use on S32G399A RDB3 platform. This is due to channel 5 of ADC0 being used to determine which RDB board revision is in use.

20 Controlling GPIOs in Linux

20.1 About libgpiod

The libgpiod library allows users to write userspace applications that interact with GPIOs. The sysfs interface is now deprecated starting with kernel version 4.8. Therefore, new designs should use the libgpiod library. The library works by using the GPIO character device and it converts the API calls into ioctl calls [4] to the GPIO character device. The term GPIO chip has the same meaning as the GPIO character device. The library can be used to do the following:

- request input/output GPIOs
- set output/read input GPIO value
- enable interrupts for a GPIO
- configure GPIO bias i.e. pull-up, pull-down

Moreover, the Yocto libgpiod-tools package provides the following tools [5]:

gpiodetect	list available GPIOs
gpioinfo	list details of GPIO lines
gpioget	read the value for the requested GPIO lines
gpioset	set the value for the requested GPIO lines
gpiofind	find the name of the GPIO chip and the offset of GPIO line given the line name
gpiomon	wait for interrupts on the specified GPIO lines

Note: this section refers to requesting GPIO pins in userspace. In order to request GPIO pins for use inside the kernel see: https://www.kernel.org/doc/html/latest/driver-api/gpio/board.html.

20.2 Using libgpiod-tools

20.2.1 gpiodetect

This tool is used to identify the GPIO chips available and the number of lines (i.e. GPIOs) they expose.

```
root@s32g399aevb3:~# gpiodetect
gpiochip0 [...siul2-gpio0] (... lines)
```

20.2.2 gpioinfo

This tool is used to print out information about a GPIO chip's GPIO lines i.e. whether they are used or not, input or output, active-high or active low.

```
root@s32g399aevb3:~# gpioinfo gpiochip0
gpiochip0 - ... lines:
line   0:      "PA_00"       unused   input  active-high
line   1:      "PA_01"       unused   input  active-high
line   2:      "PA_02"       unused   input  active-high
        [...]
```

[4] Check out `man ioctl` for more details.
[5] More details can be found at https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git.

20.2.3 gpioget

This tool is used to sample the values of the given GPIOs for a GPIO chip. It also allows the configuration of a pin bias and active-low operation. The following are some examples of its usage (Note that the GPIO pin is floating in the examples):

```
root@s32g399aevb3:~# gpioget --bias=pull-up gpiochip0 27
1
root@s32g399aevb3:~# gpioget --bias=pull-down gpiochip0 27
0
root@s32g399aevb3:~# gpioget --active-low --bias=pull-down gpiochip0 27
1
```

20.2.4 gpioset

This tool is used to set the value of the given GPIOs for a GPIO chip. This command allows the user to configure pin properties such as: `--active-low` operation, `--bias` and `--drive`.

The user also has the option to specify via the `--mode` parameter what happens after setting the GPIOs' values. Note that after the program quits, the corresponding pad will revert to the previous settings.

A few examples:

```
# set it to 0V and wait for an Enter
root@s32g399aevb3:~# gpioset --mode=wait --active-low 0 27=1
# set it to 0V and wait 3s
root@s32g399aevb3:~# gpioset --mode=time  --sec=3 0 27=0
# set it to 0V and wait for SIGINT
root@s32g399aevb3:~# gpioset --mode=signal --background 0 27=0
root@s32g399aevb3:~# kill -INT $(pgrep gpioset)
```

20.2.5 gpiofind

This tool is used to return the GPIO chip's name and line offset given a GPIO name.

```
root@s32g399aevb3:~# gpiofind PA_00
gpiochip0 0
```

20.2.6 gpiomon

This tool is used to wait for the specified number of interrupts (events) on the given GPIOs of a GPIO chip. It allows the user to wait for both fronts or only `--rising-edge`/`--falling-edge`. It can also configure the `--bias`.

A few examples:

```
# wait for 3 rising-edge interrupts
root@s32g399aevb3:~# gpiomon --num-events=3 --rising-edge 0 27
# record 100 interrupts for GPIO27 and set its bias to pull-up
# print the seconds from timestamp and interrupt edge
root@s32g399aevb3:~# gpiomon --bias=pull-up --num-events=100 --format="%s %e" 0 27
```

20.3 Devicetree Example of using a GPIO external interrupt (EIRQ)

On S32G3 SoC, an EIRQ can be mapped in some cases to more GPIOs. Therefore, the GPIO number is being used in order to know the exact EIRQ, since there isn't a case where a GPIO can have more EIRQs attached to it.

The following example illustrates the way the EIRQ associated to GPIO 15 is attached to a driver which is bounded to a device tree node having 'nxp,eirq' compatible string.

```
        nxpeirqs: nxpeirqs-node {
                interrupt-parent = <&gpio>;
                interrupts = <15 IRQ_TYPE_EDGE_FALLING>;
                compatible = "nxp,eirq";
                status = "okay";
        };
```

21 Clocking

21.1 Read clock frequency at runtime

To be able to read the clock frequency `CONFIG_DEBUG_FS` must be enabled in the kernel configuration. By default, it is enabled on the supported platforms.

The clock frequency can be obtained from */sys/kernel/debug/clk/a53/clk_rate* path. To read it, the following Linux command can be run:

```
cat /sys/kernel/debug/clk/a53/clk_rate
```

22 Setting up SIUL2 pads

The SIUL2 driver contains definitions for a restricted number of pads. If your driver needs additional PADS to be defined the following steps must be followed:

- in file drivers/pinctrl/freescale/pinctrl-s32g.c update:
  - `enum s32_pins` with entries that have the format `<PAD_NAME>= <MSCR_ID>`
  - the `s32_pinctrl_pads_siul2` array of `struct pinctrl_pin_desc`, for SIUL2_0 and SIUL2_1, with entries having the format:
    `S32_PINCTRL_PIN(<PAD_NAME_DEFINED_IN_s32_pins>)`
    Refer to the section "Register configuration" in the SoC Reference Manual for the mapping of MSCR/IMCR registers to SIUL2_0 and SIUL2_1.
- update the dts by adding a group for your driver: [6]

```
&pinctrl {
        status = "okay";

        <new_driver_pins> {
                <new_driver_grp0>: <new_driver_pins0> {
                        pinmux = <S32CC_PINMUX(<pin0>, <FUNC1>)>,
                                 <S32CC_PINMUX(<pin1>, <FUNC2>);
                        output-enable;
                        slew-rate = <S32CC_SLEW_150MHZ>;
                };

                <new_driver_grp1>: <new_driver_pins1> {
                        pinmux = <S32CC_PINMUX(<pin0>, <FUNC1>)>;
                        input-enable;
                        bias-pull-up;
                };
        };
}
```

- refer the group of pins in the dts entry for your driver

```
<new_driver> {
        pinctrl-names = "default";
        pinctrl-0 = <&<new_driver_pins>> [, ...];
        status = "okay";
};
```

[6] More details about the generic pinmux/pinconf syntax can be found at: Documentation/devicetree/bindings/pinctrl/pincfg-node.yaml and Documentation/devicetree/bindings/pinctrl/pinmux-node.yaml

23 Power Management

This chapter covers the power management features supported by BSP 36.0_cd. Power state changes are typically requested by Linux (kernel or user-space) and are executed by the Secure Monitor component of the ARM Trusted Firmware on behalf of the non-secure world.

23.1 Platform Reset

The SoC can be software reset by typing

```
reboot
```

at the Linux command prompt.

23.2 CPU Hotplug

BSP 36.0_cd supports manually removing or adding individual CPUs via the command-line. The following command-line snippet unplugs, then plugs CPU1. The *nproc* command gives the number of online CPUs at a given moment:

```
root@s32g399ardb3:~# nproc
8
root@s32g399ardb3:~# echo 0 > /sys/devices/system/cpu/cpu1/online
[   50.081441] CPU1: shutdown
[   50.082472] psci: CPU1 killed (polled 0 ms)
root@s32g399ardb3:~# nproc
7
root@s32g399ardb3:~# echo 1 > /sys/devices/system/cpu/cpu1/online
[   73.585384] Detected VIPT I-cache on CPU1
[   73.585425] GICv3: CPU1: found redistributor 1 region 0:0x0000000050920000
[   73.585479] CPU1: Booted secondary processor 0x0000000001 [0x410fd034]
root@s32g399ardb3:~# nproc
8
```

23.3 Suspend to RAM

Suspend to RAM, or STR, is a complex power management feature supported by BSP 36.0_cd. STR is a high-level concept, implemented by the Linux kernel and the TF-A, and underlaid by the S32G SoC's Standby state.

STR saves the software context and transitions the SoC to Standby state, which is an SoC low power state (not to be confounded with the VR5510 PMIC's Standby and Deep Sleep modes!). For details on the S32G hardware support for this feature, please refer to the Power Management chapters in the S32G SoC Reference Manual.

STR software puts LPDDR in self-refresh mode and all but the essential hardware blocks are powered off. A wakeup source must be configured prior to suspend. Suspend to RAM can be triggered from the Linux command-line as described below.

On executing the system suspend command on behalf of the Linux kernel, the TF-A Secure Monitor (see ARM Trusted Firmware) performs the necessary DDR and SoC configurations, then programs the chip for the Run→Standby transition. The SoC will remain in Standby until a wakeup source is triggered.

After a wakeup source is triggered, the SoC executes the Standby→Run transition and TF-A is loaded back into memory(as it does during a cold boot). During the BL2 setup stage, a check is made to find out the reset cause. In case it was a resume from Standby, the DDR and A53 clocks are enabled, the DDR is taken out of the self-refresh mode and execution continues into BL31, skipping a part of the initialization that is made during a cold boot. After that, execution will eventually resume into Linux.

While the TF-A saves and restores the core and DDR context, peripherals are generally expected to be dealt with in the Linux suspend/resume code. The kernel has a core framework for STR, and peripheral specifics are the responsibility of each Linux driver.

Because of the characteristics of the Standby→Run transition, the TF-A resume code is to a large extent common with the cold boot code. Note, however, that U-Boot is *not* involved in the Suspend/Resume sequence; only the TF-A is.

23.3.1 Wakeup Sources

The S32G3 SoC supports multiple wakeup sources. External sources can be GPIO pins, while the Real-Time Clock (RTC) can be configured as an internal source. The TF-A by default configures RTC and one external wakeup source. Actually configuring these sources to trigger a wakeup event is the responsibility of the caller of STR functionality (e.g. Linux).

23.3.2 Suspend to RAM with RTC as Wakeup Source

In order to use the RTC as a wakeup source, one needs to:

1. Preprogram the RTC module to trigger an interrupt at a later point in time
2. Put the platform into Standby

The simplest way of performing these steps is via the standard Linux *rtcwake* command, which configures one of the available RTC drivers (default */dev/rtc0*), then requests the kernel to execute a platform suspend. The Linux RTC driver will appropriately configure the RTC device, then will ask the TF-A to perform the transition to Standby.

The following code snippet showcases the use of *rtcwake* for STR:

```
s32g399ardb3 login: root
root@s32g399ardb3:~# echo;date;echo; rtcwake -d rtc0 -m mem -s 180; echo;date

Fri Nov 26 01:43:51 UTC 2021

rtcwake: assuming RTC uses UTC ...
rtcwake: wakeup from "mem" using rtc0 at Thu Jan  1 00:13:09 1970
[  460.260205] PM: suspend entry (deep)
[  460.260286] Filesystems sync: 0.000 seconds
[  460.260761] Freezing user space processes ... (elapsed 0.007 seconds) done.
[  460.268581] OOM killer disabled.
[  460.268585] Freezing remaining freezable tasks ... (elapsed 0.006 seconds) done.
[  460.275512] printk: Suspending console(s) (use no_console_suspend to debug)
[  460.294060] pci_bus 000b:01: busn_res: [bus 01-ff] is released
[  460.294268] pci_bus 000b:00: busn_res: [bus 00-ff] is released
[  460.310919] fsl_fccu 4030c000.fccu: FCCU status is 0 (normal)
[  460.312278] s32cc-dwmac 4033c000.ethernet: phy mode set to RGMII
[  460.378881] phy-s32gen1-serdes 40480000.serdes: Using mode 0 for SerDes subsystem
[  460.380232] phy-s32gen1-serdes 44180000.serdes: Using mode 0 for SerDes subsystem
[  461.382301] s32gen1-pcie 40400000.pcie: Failed to stabilize PHY link
```

```
[  461.382310] s32gen1-pcie 40400000.pcie: Configuring as RootComplex
[  462.381151] s32gen1-pcie 40400000.pcie: Phy link never came up
[  462.381286] pci_bus 000c:00: parent pci000c:00 should not be sleeping
[  462.381361] s32gen1-pcie 40400000.pcie: PCI host bridge to bus 000c:00
[  462.381373] pci_bus 000c:00: root bus resource [bus 00-ff]
[  462.381383] pci_bus 000c:00: root bus resource [io  0x0000-0xffff]
[  462.381394] pci_bus 000c:00: root bus resource [mem 0x5800000000-0x5ffffdffff]
(bus address [0x00000000-0x7fffdfffff])
[  462.381444] pci 000c:00:00.0: [1957:4002] type 01 class 0x060400
[  462.381470] pci 000c:00:00.0: reg 0x10: [mem 0x5800000000-0x58000fffff]
[  462.381492] pci 000c:00:00.0: reg 0x38: [mem 0x5800000000-0x580000ffff pref]
[  462.381594] pci 000c:00:00.0: supports D1
[  462.381600] pci 000c:00:00.0: PME# supported from D0 D1 D3hot D3cold
[  462.381753] pci 000c:00:00.0: parent pci000c:00 should not be sleeping
[  462.391404] pci 000c:00:00.0: BAR 0: assigned [mem 0x5800000000-0x58000fffff]
[  462.391418] pci 000c:00:00.0: BAR 6: assigned [mem 0x5800100000-0x580010ffff pref]
[  462.391431] pci 000c:00:00.0: PCI bridge to [bus 01-ff]
[  463.392374] s32gen1-pcie 44100000.pcie: Failed to stabilize PHY link
[  463.392383] s32gen1-pcie 44100000.pcie: Configuring as RootComplex
[  464.389151] s32gen1-pcie 44100000.pcie: Phy link never came up
[  464.389238] pci_bus 000d:00: parent pci000d:00 should not be sleeping

[  464.389277] s32gen1-pcie 44100000.pcie: PCI host bridge to bus 000d:00


Fri Nov 26 01:46:55 UTC 2021

root@s32g399ardb3:~#[  464.389288] pci_bus 000d:00: root bus resource [bus 00-ff]
[  464.389298] pci_bus 000d:00: root bus resource [io  0x10000-0x1ffff] (bus address [0x0000-0xffff])
[  464.389308] pci_bus 000d:00: root bus resource [mem 0x4800000000-0x4ffffdffff]
(bus address [0x00000000-0x7fffdfffff])
[  464.389343] pci 000d:00:00.0: [1957:4002] type 01 class 0x060400
[  464.389363] pci 000d:00:00.0: reg 0x10: [mem 0x4800000000-0x48000fffff]
[  464.389380] pci 000d:00:00.0: reg 0x38: [mem 0x4800000000-0x480000ffff pref]
[  464.389446] pci 000d:00:00.0: supports D1
[  464.389452] pci 000d:00:00.0: PME# supported from D0 D1 D3hot D3cold
[  464.389582] pci 000d:00:00.0: parent pci000d:00 should not be sleeping
[  464.399001] pci 000d:00:00.0: BAR 0: assigned [mem 0x4800000000-0x48000fffff]
[  464.399015] pci 000d:00:00.0: BAR 6: assigned [mem 0x4800100000-0x480010ffff pref]
[  464.399025] pci 000d:00:00.0: PCI bridge to [bus 01-ff]
[  464.406456] pcieport 000c:00:00.0: PME: Signaling with IRQ 86
[  464.406765] pcieport 000d:00:00.0: PME: Signaling with IRQ 87
[  464.406940] OOM killer enabled.
[  464.406944] Restarting tasks ... done.
[  464.411736] PM: suspend exit

root@s32g399ardb3:~#
```

The above command uses */dev/rtc0* , which is the S32G RTC device. That being the default, the *-d rtc0* argument might be omitted. It specifies the type of low-power transition (*mem* means "suspend to RAM") and the number of seconds before the wakeup interrupt (here, *180* seconds). The platform goes to sleep, then after the specified time it wakes on RTC input and resumes execution.

Note that the kernel timestamps printed during suspend and resume do not reflect the amount of time the platform has actually been sleeping. This is because the ARM Generic Timer, which is used as System Timer in Linux, is not running while the SoC is in Standby, and so it cannot count the time elapsed while suspended. The system date, however,

correctly reflects the time slept.

23.3.3 Suspend to RAM with GPIO as Wakeup Source

In order to use an external GPIO pin as a wakeup source, one needs to make the following steps:

1. Enable a GPIO pin as an external wakeup source
   The TF-A must be configured via the device-tree to use a GPIO as an external wakeup source. Before building and deploying the TF-A binaries, the `nxp,irqs` property in the `wkpu` node of the fdts/s32cc.dtsi device tree should be configured as in the following example:

```
wkpu: wkpu@40090000 {
        compatible = "nxp,s32gen1-wkpu";
        reg = <0x0 0x40090000 0x0 0x10000>, /* WKPU */
                        <0x0 0x4007cb04 0x0 0x4>; /* S32G_STDBY_GPR */
        /*
        * Enable GPIO as wakeup source
        */
        nxp,irqs = <S32GEN1_WKPU_EXT_IRQ(16)
                        S32GEN1_WKPU_IRQ_RISING
                        S32GEN1_WKPU_PULL_DIS>;

        nxp,warm-boot = <S32GEN1_WKPU_LONG_BOOT>;

        status = "disabled";
};
```

   In the above example the external wakeup interrupt 16 is set using the `S32GEN1_WKPU_EXT_IRQ()` macro. The WKPU unit supports up to 23 external wakeup interrupts numbered from 0 to 22. By default there is no external wakeup interrupt set.

2. Disable the RTC module

```
echo disabled > /sys/devices/platform/40060000.rtc/power/wakeup
```

   This is important: if the RTC is set as a wakeup source (default behavior) and it is not configured, the Linux Kernel power management core will error out.

3. Perform the suspend to RAM transition

```
echo mem > /sys/power/state
```

   This will leave the platform in Standby state indefinitely. In order to manually wake the platform up, one needs to apply 3V3 voltage to the configured GPIO pin. This specific process is outside the scope of this document.

23.4 Dynamic Frequency Scaling

BSP 36.0_cd supports Dynamic Frequency Scaling (DFS) for the A53 cores, meaning that the clock frequency of the CPUs can be adjusted on the fly in order to save energy. Note that this is an experimental feature for the time being.

CPU frequency scaling is achieved using the `cpufreq` subsystem [7] offered by the Linux Kernel. Requests to change the clock rate are forwarded through SCMI calls to the Secure Monitor component of ATF, which handles the actual frequency change operation.

To enable DFS, invoke `menuconfig` before building the kernel and use the following configuration options:

```
CONFIG_CPU_FREQ=y
CONFIG_ARM_SCMI_CPUFREQ=y
```

The `cpufreq` subsystem provides a list of CPU governors, each governor implementing a specific frequency scaling algorithm. The default scaling governor is called `schedutil` and it adjusts the clock rate according to the utilization metrics from the scheduler [8]. From `menuconfig`, one can change the default governor and select the scaling governors available on the system, under:

-> CPU Power Management

-> CPU Frequency scaling

Make sure that *SCMI based CPUfreq driver* option (corresponding to *CONFIG_ARM_SCMI_CPUFREQ*) remains selected after doing this step.

In order to configure DFS, go to the sysfs directory associated with this feature:

```
root@s32g399aevb3:~# cd /sys/devices/system/cpu/cpufreq/policy0
```

Note: All the CPUs are grouped under the same cpufreq policy object, which means that they share the same configuration interface, hence any change on one CPU also affects the other ones.

To see the current clock frequency of the A53 cores run:

```
root@s32g399aevb3:/sys/devices/system/cpu/cpufreq/policy0# cat cpuinfo_cur_freq
1300000
```

Note: Frequency values are expressed in kHz.

To list the available clock frequencies use:

```
root@s32g399aevb3:/sys/devices/system/cpu/cpufreq/policy0/# cat scaling_available_frequencies
48148 54166 61904 72222 86666 104000 136842 200000 371428 1300000
```

To see the current governor use:

```
root@s32g399aevb3:/sys/devices/system/cpu/cpufreq/policy0# cat scaling_governor
schedutil
```

To get the list of the available governors in the system use:

[7] https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html

[8] Refer to the official Linux Kernel documentation for a full description of the scaling governors

```
root@s32g399aevb3:/sys/devices/system/cpu/cpufreq/policy0# cat scaling_available_governors
schedutil performance userspace
```

To set a new governor run the following command, where `<governor>` is one of the available governors from `scaling_available_governors` attribute:

```
root@s32g399aevb3:/sys/devices/system/cpu/cpufreq/policy0# echo <governor> > scaling_governor
```

To manually set a frequency (only for the `userspace` governor) use the following command, where `<frequency>` is one of the available values from `scaling_available_frequencies` attribute:

```
root@s32g399aevb3:/sys/devices/system/cpu/cpufreq/policy0# echo <frequency> > scaling_setspeed
```

24 Virtualization

24.1 Linux Containers User Guide

Linux Containers is a lightweight virtualization technology that allows the creation of environments in Linux called "containers" in which Linux applications can be run in isolation from the rest of the system and with fine grained control over resources allocated to the container (e.g. CPU, memory, network).

LXC is a user space package that provides a set of commands to create and manage containers and uses existing Linux kernel features to accomplish the desired isolation and control.

Why are containers useful?

Below are a few examples of container use cases:

- Application partitioning – control CPU utilization between high priority and low priority applications, control what resources applications can access.
- Virtual private server – boot multiple instances of user space, each which effectively looks like a private instance of a server. This approach is commonly used in website infrastructure.
- Software upgrade – run Linux user space in a container, when it becomes necessary to upgrade applications in the system, create and test upgraded software in a new container. The old container can be stopped and the new container can be started as desired.
- Terminal servers – user accesses the system with a thin client, with containers on the server providing applications. Each user gets a private, sandboxed workspace.

There are two general usage models for containers:

- Application containers – Running a single application in a container. In this scenario, a single executable program is started in the container.
- System containers – Booting an instance of user space in a container. Booting multiple system containers allows multiple isolated instances of user space to run at the same time.

Containers are conceptually different than virtual machine technologies such as QEMU/KVM. Virtual machines emulate a hardware platform and are capable of booting an operating system kernel. A container is a mechanism to isolate Linux applications. In a system using containers there is only one Linux kernel running – the host Linux kernel.

24.1.1 Build, Installation and Configuration

24.1.1.1 Building with Yocto

LXC is a Linux user space package that can easily be added to a root filesystem using the Yocto build system. Update the DISTRO_FEATURES:append variable in the *conf/local.conf* file in the Yocto build environment as follows:

```
DISTRO_FEATURES:append = " lxc"
```

Then bitbake the image:

```
bitbake fsl-image-auto
```

24.1.1.2 Update the root filesystem so the system is ready to support containers

In order to use containers, mount the 'cgroup' pseudo-filesystem:

```
root@s32g399aevb3:~# mount -t cgroup cgroups /sys/fs/cgroup
```

24.1.2 LXC How To's (Getting Started with a BusyBox System Container)

The following article describes steps to run a simple container example. All the commands below are issued on a target running Linux.

1. Confirm that your kernel environment is configured correctly using lxc-checkconfig

```
root@s32g399aevb3:~# lxc-checkconfig
---Namespaces---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: enabled
Network namespace: enabled
Multiple /dev/pts instances: enabled
---Control groups---
Cgroup: enabled
Cgroup namespace: required
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: missing
Cgroup cpuset: enabled
---Misc---
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled
```

Note : Before booting a new kernel, you can check its configuration
Usage :

```
CONFIG=/path/to/config/usr/bin/lxc-checkconfig
```

If the cgroup namespace option shows as required:

```
 Cgroup namespace: required
```

The */cgroup* directory needs to be created and/or mounted.
See 24.1.1.2

2. Create a container

   Create a system container using lxc-create and specify the BusyBox template and *lxc-empty-netns.conf* config file. *lxc-empty-netns.conf* is a simple config file with no networking:

```
root@s32g399aevb3:~# lxc-create -n foo -t busybox -f /usr/share/doc/lxc/examples/lxc-empty-netns.conf
/home/root
```

3. List containers that exist

```
root@s32g399aevb3:~# lxc-ls -f
NAME STATE   AUTOSTART GROUPS IPV4 IPV6 UNPRIVILEGED
foo  STOPPED 0         -      -    -    false
```

4. Modify lxc.mount.auto parameter

   Specify which standard kernel file system should be automatically mounted. The file systems are:

proc:mixed (or proc)	mount */proc* as read-write, but remount */proc/sys* and */proc/sysrq-trigger* read-only for security / container isolation purposes
proc:rw	mount */proc* as read-write
sys:mixed	mount */sys* as read-only but with */sys/devices/virtual/net* writable
sys:ro	mount */sys* as read-only for security / container isolation purposes
sys:rw	mount */sys* as read-write

Note : In order to have access rights to the file system, the configuration file of the container */var/lib/lxc/container_name/config* should have the lxc.mount.auto parameter **proc:rw**

```
root@s32g399aevb3:~# cat /var/lib/lxc/foo/config
# Template used to create this container: /usr/share/lxc/templates/lxc-busybox
# Parameters passed to the template: examples/lxc-empty-netns.conf
# For additional config options, please look at lxc.container.conf(5)
lxc.rootfs = /var/lib/lxc/foo/rootfs
lxc.haltsignal = SIGUSR1
lxc.rebootsignal = SIGTERM
lxc.utsname = foo
lxc.tty = 1
lxc.pts = 1
lxc.cap.drop = sys_module mac_admin mac_override sys_time

# When using LXC with apparmor, uncomment the next line to run unconfined:
#lxc.aa_profile = unconfined
lxc.mount.entry = /lib lib none ro,bind 0 0
lxc.mount.entry = /usr/lib usr/lib none ro,bind 0 0
lxc.mount.entry = /sys/kernel/security sys/kernel/security none ro,bind,optional
lxc.mount.auto = proc:rw
```

5. How to start the container
   From a shell on the target, start the container.

```
root@s32g399aevb3:~# lxc-start -n foo
```

If no error message is displayed, then the container was started successfully (as a daemon).

In order to actually use the container, connect to its console then login using user 'root' and an empty password:

```
root@s32g399aevb3:~# lxc-console -n foo

Connected to tty 1
Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter Ctrl+a itself

foo login: root
~ #
```

Note that the shell is now running within the container. Normal Linux commands can be executed. To exit the container console, press <Ctrl+a q> as instructed.

Important notice: the container can be started in foreground, with argument '-F', in which case it will directly connect to one of its terminals:

```
root@s32g399aevb3:~# lxc-start -n foo -F

init started: BusyBox v1.32.0 ()

Please press Enter to activate this console.
/ #
```

When prompted, press 'Enter'. However, this mode has a minor caveat: the terminal will be stuck in this container console until the container is halted (either from here, by running 'halt', or from another terminal by running lxc-stop).

6. List processes in the container
   From the container shell use the ps command to list processes:

```
~ # ps
 PID USER TIME COMMAND
 1   root 0:00 init
 6   root 0:00 /bin/syslogd
 8   root 0:00 /bin/getty -L tty1 115200 vt100
 9   root 0:00 /bin/sh
10   root 0:00 ps
```

7. Stop the container (from a host shell)

```
root@s32g399aevb3:~# lxc-stop -n foo
root@s32g399aevb3:~# lxc-info -n foo
Name:      foo
State:     STOPPED
```

8. Destroy the container

```
root@s32g399aevb3:~# lxc-destroy -n foo
```

24.2 Docker Containers

24.2.1 What is Docker?

Docker is a lightweight virtualization solution, which extends the capabilities of LXC. It is an Open-Source containerization technology which focuses on running applications in isolated environments.

Docker containers rely on Docker Engine, which is composed of a daemon process (dockerd), an API that applications can use to interact with the Docker daemon, and a CLI interface. The daemon process manages the underlying OS infrastructure to run the containers, which are based on templates named Docker Images.

24.2.2 Enabling Docker in Auto Linux BSP

To enable Docker in a Linux BSP build, the Kernel Image needs to be correctly configured to include all the required support, and the Docker Engine package needs to be installed into the root file system. This ensures basic Docker

support for running containers.

For the Ubuntu target image, Docker support is included by default when building `fsl-image-ubuntu`. The next steps should only be done when enabling Docker in images based on Yocto root file system (e.g. `fsl-image-auto`).

The required Kernel support can be enabled in a Yocto build by adding the following line to the *conf/local.conf* in the build directory:

```
DISTRO_FEATURES:append = " docker"
```

Also, the Docker Container Engine should be installed into the built image, by adding the following line to the image recipe (e.g. recipes-fsl/images/fsl-image-auto.bb):

```
IMAGE_INSTALL:append = " docker-ce"
```

Note: Depending on the size of the Docker Images that are fetched from Docker Repository, the available free space in root file system may not be enough and should be extended. For example, increasing the extra space to 2GB can be done by adding the following lines to the *conf/local.conf* in the build directory:

```
# Leave 2GB extra space
IMAGE_ROOTFS_EXTRA_SPACE = "2097152"
```

Build the *.sdcard* image, boot to Linux prompt, ensure that the board has Internet connectivity and the date is correctly set. Then, run the following command to test that Docker is working:

```
root@s32g399aevb3:~# docker run hello-world
```

24.2.3 Docker Compose

Docker Compose is a Docker tool used to describe and run multi-container Docker applications. The configuration of the entire application is written in a YAML file, which should contain details about all the containers and what services they need from the Docker Engine. All the containers can be launched using the "`docker-compose up`" command.

To enable Docker Compose for images based on Yocto root file system, in addition to Docker Engine, also add the following line to the image recipe (e.g. recipes-fsl/images/fsl-image-auto.bb):

```
IMAGE_INSTALL:append = " python3-docker-compose"
```

For the Ubuntu target image (`fsl-image-ubuntu`), Docker Compose can be installed:

- before building the Ubuntu image, by adding the following line to the image recipe (e.g. recipes-fsl/images/fsl-image-ubuntu.bb):

  ```
  APTGET_EXTRA_PACKAGES += "docker-compose"
  ```

- after booting, by using the Ubuntu Package Manager:

```
root@ubuntu-s32g399aevb3:~# apt-get install docker-compose
```

In order to use Docker Compose, a multi-container application is needed. A useful resource is the official Docker Compose Getting Started Guide.

After getting all the necessary code (Python App, Dockerfile, requirements.txt and Docker Compose YAML) for the example application, the application can be built and run inside Docker Containers as following:

```
root@s32g399aevb3:~# docker-compose up -d
```

24.3 Xen Hypervisor

24.3.1 What is Xen?

Xen is an Open-Source Type-1 (Bare-metal) hypervisor. Type-1 hypervisors run directly on the hardware, having one or more Operating Systems running on top of it. It is hosted by The Linux Foundation.

Xen on ARM supports ARM devices using the virtualization extensions of the ARM CPUs.

There are 2 types of Virtual Machines (VMs) that run on top of Xen:

- Privileged:
  - *Dom0* (the first virtual machine)
- Unprivileged:
  - *DomUs* (booted from Dom0 via the xl toolstack)

Xen virtualizes CPU, memory, interrupts and timers, providing virtual machines with virtualized resources. All hardware devices are assigned to Dom0 by default, that runs the native device drivers for them.

Disk, network, console and other I/O devices are shared between VMs using a paravirtualized backend/frontend approach. Dom0 runs all the backend drivers and services one or more frontends.

24.3.2 Dom0less Virtual Machines

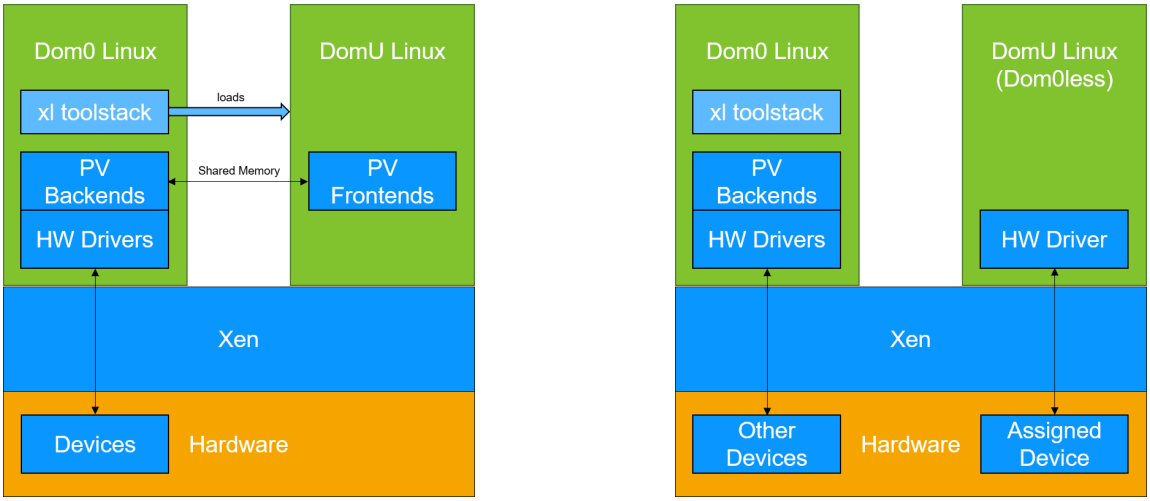
Dom0less VMs are a special kind of DomUs, since they cannot be managed via *xl toolstack*, and therefore cannot benefit from any paravirtualized drivers. They have access to the devices that Xen virtualizes for them (CPU, memory, interrupts and timers), and also to an emulated vPL011 console.

Access to other peripherals can be granted using Device Passthrough, a virtualization technique for assigning various system peripherals to an unprivileged VM. S32 Platforms do not have an IOMMU and this adds some limitations regarding virtualization. For S32 Platforms, Dom0less VMs need to have 1:1 mapping from intermediate physical address (IPA) to physical address (PA) in order to support Device Passthrough.

The configuration for Dom0less VMs is done through Device Tree, making it easy to deploy a statically-configured system.

The difference between normal DomUs and Dom0less DomUs is illustrated in Figure 1. Subfigure 1a shows the interaction between Dom0 and a DomU VM. The DomU is loaded from Dom0 using *xl toolstack*, and DomU's access to hardware is granted by the paravirtualized backend-frontend drivers approach. [9]

[9] For platforms with an IOMMU, direct assignment of devices to normal DomUs is also available.



(a) Dom0 and DomU Configuration

(b) Dom0 and Dom0less DomU Configuration

Figure 1: Normal and Dom0less Use Cases

In contrast, Subfigure 1b shows that a Dom0less DomU has no interaction with Dom0. Xen boots the Dom0less DomUs and directly assigns devices to them. Advantages of Dom0less DomUs are:

- Faster boot time (no need to wait for Dom0 to boot first)
- Static configuration (no need to dynamically configure anything after booting)

24.3.3 How to enable Xen in S32 Linux BSP build

Xen support can be enabled by adding the following line in the *conf/local.conf* file of your Yocto build directory:

```
DISTRO_FEATURES:append = " xen"
```

This will build Xen and all its dependencies, along with Xen tools (xl toolstack) that are needed to manage the DomUs from Dom0.

After building a *fsl-image-auto* or *fsl-image-base* image, the following files are deployed in the */boot* folder of the rootfs partitions:

- Xen Image (*xen*)
- Linux Kernel Image (*Image*)
- Linux Device Tree binary (*fsl-<board_name>.dtb*)
- Boot script and its text source (*boot.scr*, *boot.source*)
- *optional:* Ramdisk image for Dom0less-DomUs (*fsl-image-dom0less-<board_name>.cpio.gz*)
- *optional:* Passthrough Device Tree binary (*passthrough.dtb*)
- xl toolstack (installed in the rootfs partitions, in */usr/sbin*)

The deployed boot script (*boot.scr*) is executed by U-Boot. It contains all the necessary commands for setting up the environment to boot Xen.

The optional files are deployed depending on the custom Xen setup chosen through the *XEN_EXAMPLES* variable. The setups are described in the section - Building Custom Xen Setups.

Note 1: The generated SDCard image has an additional partition that contains a root filesystem that can be used by a DomU.

Note 2: Another extra rootfs partition (i.e. for another DomU) can be enabled by adding the following lines to *conf/local.conf* file, and then *domU2_mmc.cfg* configuration file can be used to boot DomU2, as described in subchapter "Booting Xen, Dom0 and DomU":

```
SDCARD_ROOTFS_EXTRA2_FILE = "${SDCARD_ROOTFS_EXTRA1_FILE}"
SDCARD_ROOTFS_EXTRA2_SIZE = "${SDCARD_ROOTFS_EXTRA1_SIZE}"
```

After the build is complete, write the generated SDCard image as in an usual S32 BSP build.

24.3.4 Building Custom Xen Setups

With the increased complexity of setting up a statically-configured system, the user can also select a pre-configured Xen configuration, defined by the *xen-examples-*.bb* recipes within Yocto BSP layer. This can be done by setting the *XEN_EXAMPLE* variable in the *conf/local.conf* to one of the values mentioned in Table 13.

XEN_EXAMPLE value	Dom0	Dom0less-DomU1	Device Passthrough
"xen-examples-default"	512MB of RAM 1 vCPU RootFS on SD part.	-	-
"xen-examples-dom0less"		1024MB of RAM 1 vCPU RootFS on initrd	-
"xen-examples-dom0less-passthrough"			Manually configurable
"xen-examples-dom0less-passthrough-gmac"			GMAC controller assigned to DomU1

Table 13: Xen Custom Setups

Note 1: The default value for the *XEN_EXAMPLE* variable is *"xen-examples-default"*. This means that the default Xen setup is built in case the variable is not explicitly set.

For example, to build a basic setup with only Dom0 booting and no Dom0less VMs, the user can add the following to the *conf/local.conf* file:

```
XEN_EXAMPLE = "xen-examples-default"
```

For each value assigned to *XEN_EXAMPLE*, there is a corresponding recipe that is selected, built and deployed along with its runtime dependencies. These recipes can be found in recipes-extended/xen-examples, and each has an

associated config file, which contains the parameters of the system that can be changed.

All the recipes automatically build a complete Xen setup as per their description in Table 13. Automatically configured device passthrough for the GMAC controller is supported, through the "*xen-examples-dom0less-passthrough-gmac*" recipe.

24.3.5 Booting Xen, Dom0 and DomU

After having a complete image on the SDCard, the */boot* folder in each rootfs partition contains the items mentioned in section How to enable Xen in S32 Linux BSP build.

When the *boot* command is run in U-Boot, the boot script (*boot.scr*) is executed. This loads all the required files into memory and sets up the */chosen* node for providing the system settings to Xen. Then, U-Boot boots Xen (instead of the Linux Image), and Xen loads the Kernel Image. This is the image for Dom0. The same image is also used for Dom0less-DomUs, that boot in parallel with Dom0.

After Dom0 is up and running, the user can manage the DomUs from Dom0 userspace, using the *xl toolstack* installed in the rootfs. DomUs are configured via config files, written using the xl configuration syntax. Example configuration files can be found in Dom0's rootfs in */etc/xen/* folder.

Example for booting a DomU as described by tools/examples/domU1_mmc.cfg (384MB of RAM, 1 vCPU pinned to a pCPU, rootfs mounted directly on sdcard):

```
# Starting the DomU
root@s32g399aevb3:~# xl create /etc/xen/domU1_mmc.cfg
# Switching to DomU's console
root@s32g399aevb3:~# xl console domu1
...
# Switching back to Dom0's console
root@s32g399aevb3:~# CTRL-]
```

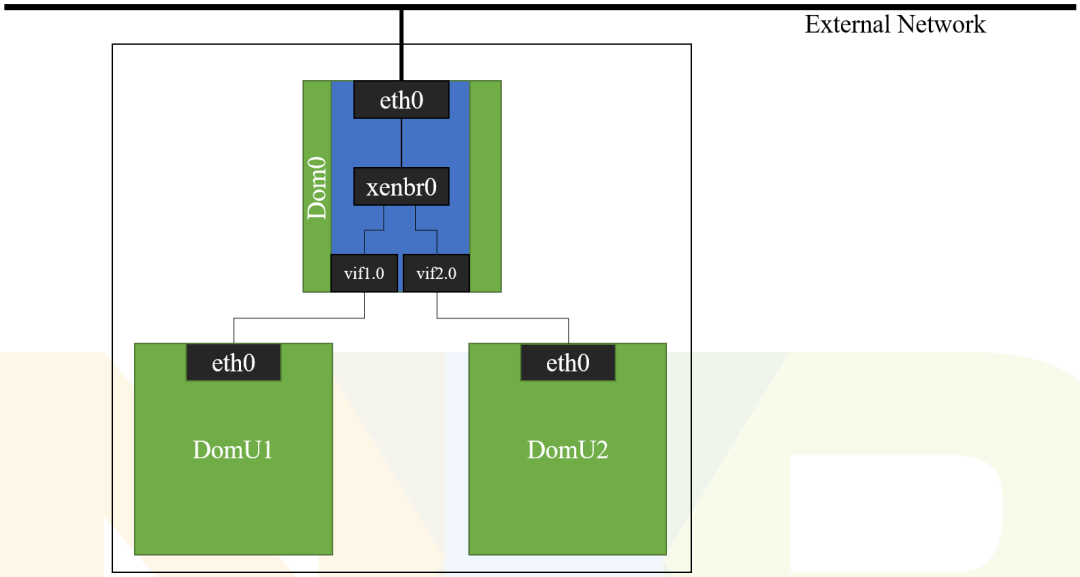
Note 1: When booting a Dom0less VM, its console cannot be reached using the *xl console* command, but by pressing "*CTRL-AAA*", which cycles between Dom0's console, the Dom0less VMs' consoles and the Xen debug console.

Information about the system and the running VMs can be accessed using other xl commands:

```
# Info about system (RAM, CPUs etc.)
root@s32g399aevb3:~# xl info

# Info about running VMs (ID, state, resources)
root@s32g399aevb3:~# xl list

# Info about resource usage
root@s32g399aevb3:~# xl top

# Shutdown a DomU
root@s32g399aevb3:~# xl destroy <domU_ID>
```

24.3.6 Networking in DomUs

In order to have networking access in DomUs, Xen enables paravirtualized network interfaces (ethY) in the guests which are then linked to virtual interfaces created in Dom0. The virtual interfaces in Dom0 (vifX.Y) are connected to the physical host interface (eth0) via a bridge (xenbrX), which must be explicitly created and will act as a virtual switch. The virtual interfaces are created automatically by Xen at guest creation time. The bridge will take packets from the virtual machines and forward them on to the physical network.



First, we need to make sure the Linux Kernel is compiled with the bridging modules enabled:

```
CONFIG_BRIDGE_NETFILTER=m
CONFIG_BRIDGE=y
CONFIG_BRIDGE_IGMP_SNOOPING=y
```

To create the bridge and attach the physical host interface to it we must also include "bridge-utils" package in our BSP Image via *conf/local.conf*, besides "xen":

```
DISTRO_FEATURES:append = " xen bridge-utils"
```

Therefore, for a conventional networking setup, in which the DomU is connected to the external network, the user needs to setup the bridge by running the following commands in Dom0:

```
# Remove all IPs from physical interface
ip addr flush dev eth0

# Create bridge
brctl addbr xenbr0

# Disable default port VLAN
```

```
echo 0 > /sys/class/net/xenbr0/bridge/default_pvid

# Add physical interface to bridge
brctl addif xenbr0 eth0

# Set bridge and physical interface up
ip link set dev xenbr0 up
ip link set dev eth0 up

# Request IP from DHCP Server on bridge
udhcpc -i xenbr0

# Or manually assign an IP Address
ip address add 192.168.0.2/24 dev xenbr0
```

Now, for Xen to attach the PV interface in DomU (eth0) to a virtual interface in Dom0, the following line must be added to the DomU config file used:

```
vif = [ 'bridge=xenbr0' ]
```

Additionally, the MAC and IP addresses of the DomU network interface can be specified in the same above-mentioned line, by specifying configuration keys 'mac=...,ip=...'.

Note 1: The physical host interface in Dom0 (e.g. "*eth0*", "*pfe0*") must not have any assigned IP address. Only "*xenbr0*" should have an assigned IP. Otherwise, the DomU will have no external network access.

Note 2: When the GMAC controller is assigned to a Dom0less-DomU, it cannot be used by Dom0. Thus, Dom0 cannot share the access to it through PV drivers.

25 ARM Trusted Firmware

The ARM Trusted Firmware (ATF, or TF-A) is a software component of the Linux BSP with a double role as a bootloader and Secure Monitor. As a bootloader, it runs before U-Boot on the A53 boot core and takes over some of the core and SoC initializations such as clocks or MC_ME partitions (mostly for those peripherals and subsystems that the ATF needs). As a Secure Monitor, the TF-A provides power management services to other software components, via PSCI and SCMI.

The TF-A is divided into several stages, called Boot Levels (BLs). ARM TF-A specification includes BL1 (Trusted ROM), BL2 (Trusted Boot Firmware), BL31 (Secure Monitor), BL32 (Trusted OS) and BL33 (non-trusted world, e.g. U-Boot or UEFI). The BSP 36.0_cd TF-A implements the BL2 and BL31 stages:

- BL2 is the component pointed to by the boot IVT, see chapter TF-A SMP Boot. It is deployed in SRAM at a statically-defined address. Its main role is to initialize the platform and in particular the DDR, and load BL31. BL2 is a transient component. After the the next stage (BL33/U-Boot) has booted, BL2 is no longer used and its SRAM space can be reclaimed.
- BL31 is loaded by BL2 in DDR, at a statically-defined address. Currently, BL31 is deployed at the top of the first 2GB of DDR, minus 2MB. BL31 is a resident component. It occupies the DDR region for the entire uptime of the platform, being involved in runtime power management, platform reset, suspend to RAM, etc. See Power Management chapter for details. It is to be noted that U-Boot cooperates with the TF-A in actually reserving the 2MB off the top of DDR's first 2GB, and hiding that region from the Linux kernel. To do so, U-Boot alters the Linux device tree before loading the kernel. See the SMP Information chapter SMP Boot for more details.

26 OP-TEE

The OP-TEE software component will provide Linux BSP users with a Trusted Execution Environment (TEE) so that critical applications working with sensitive data can leverage the hardware isolation of Arm TrustZone. Such applications are called Trusted Applications (TAs), and they run at Secure EL0.

The OP-TEE project is mainly split in two modules:

- **The optee_client module**, residing entirely in the user space of Linux, facilitates the communication between normal world applications (running at Non-Secure EL0 level) and secure world through the TEE Client API and the tee-supplicant daemon. When a normal world application invokes a TA via an API call, behind the scenes the normal world - the OP-TEE driver in the Linux Kernel - executes a Secure Monitor Call (SMC) instruction. The Secure Monitor (SM) traps the exception and will switch execution to optee_os in the secure world. The process of switching to the normal world from the secure world uses the same mechanisms.
- **The optee_os module**, which runs at Secure EL1, ensures that TAs have all the necessary resources to be executed safely and securely. It has its minimal kernel, drivers and libraries, and implements the TEE Internal Core API that can leverage the TrustZone (TZ) hardware isolation. While optee_client implementation is platform-independent, the optee_os module is configured according to the platform specifications. Trusted memory zones, UART communication and GIC initialization are configured to ensure proper usage. ATF is responsible for loading and executing the optee_os binaries. Moreover, as mentioned above, OP-TEE also relies on the Secure Monitor implemented in the ATF.

Besides the optee_os and optee_client components, which are mandatory for creating a functional Trusted Execution Environment, the Linux BSP comes with another two modules offered by the OP-TEE project: optee_examples and optee_test. The contents of both these modules reside in the user space of the Linux BSP. **Before using these modules, please note:** Once the Linux has booted and you have logged in, start the tee-supplicant daemon in the background to ensure communication between Linux and optee_os with this command:

```
tee-supplicant &
```

As the name implies, **the optee_examples module's** purpose is to give a glimpse into the basic building blocks and behavior of some TAs by exposing typical use-cases. There are a number of normal applications in the rootfs, all of them prefixed with `optee_example_` under the path */usr/bin/*, each corresponding to and calling a different TA. As an example, for running the `optee_example_hello_world` application, one must enter the following command:

```
optee_example_hello_world
```

The source code for these applications can be found on the official GitHub page: OP-TEE Sample Applications.

The optee_test module contains several thousands of tests checking that the OP-TEE integration has been done with no regression. The easiest way of running all the default tests is by entering the following command:

```
xtest
```

More information about running `xtest` can be found in the official OP-TEE documentation: optee_test.

26.1 RPMB Secure Storage

RPMB stands for Replay-Protected Memory Block. It is a partition on the eMMC where the host can read/write blocks of data, but cannot delete them. Every block must be authenticated and signed with a key. This specific key is One-Time Programmable (OTP): it can be programmed only once into eMMC's RPMB partition, in plain text. Thus, this action must take place in a secure environment. The protection against replay attacks is granted through a read-only counter that is incremented after every write operation and cannot be decremented. This way, an attacker cannot tamper with previously written blocks and only new blocks can be written up to the limit of the counter.

OP-TEE does not have an eMMC driver and instead relies on the Linux Kernel driver to read/write blocks from/to eMMC. These requests are managed by the `tee-supplicant` which further uses the `ioctl` interface to communicate with the device. Each block of data is encrypted using the AES-CBC algorithm to prevent exposure of sensitive data to the normal world. Requesting to program the RPMB Key into the eMMC is possible, though the key would show up in plain text in the normal world - take caution in performing this operation, otherwise the key might leak.

OP-TEE derives the RPMB key from the Hardware Unique Key (HUK). The Hardware Security Engine (HSE) generates the HUK through a key derivation service. Starting from a secret key embedded within HSE and using a Key Derivation Function (KDF), HSE exports a new key that is used as the HUK in OP-TEE. This secret key is unique for each SoC, meaning that HUKs and RPMB Keys are also unique. As such, once a RPMB Key has been generated and programmed into the RPMB partition, the eMMC is tied to that specific SoC. If the SoC is replaced on the board containing the eMMC chip, then the existing RPMB partition will no longer be accessible.

To demonstrate RPMB Secure Storage capabilities, the Linux filesystem hosts a Trusted Application called `optee_example_secure_storage`. The application performs basic I/O operations with objects stored in the RPMB Secure Storage.

Note: By default, `tee-supplicant` emulates a RPMB partition and, for safety considerations, the Linux BSP uses this emulation. Disabling the emulation translates to a commitment of permanently programming the RPMB Key into the eMMC partition, assuming the corresponding consequences that come with it.

Since the RPMB partition is emulated by default, it is not mandatory to boot from an eMMC to use this feature. Any boot medium supported in the Linux BSP does the job. To use the actual RPMB partition on the eMMC, please refer to the Setup eMMC section for instructions on using the eMMC.

Enable the RPMB Secure Storage by compiling optee_os with the following build parameters:

- `CFG_RPMB_FS=y`
  Enables the RPMB File System in optee_os

- `CFG_RPMB_WRITE_KEY=y`
  Write the generated RPMB Key into the RPMB partition

- `CFG_REE_FS=n`
  By default, Secure Storage is enabled using the normal world file system. Disable it since we're using RPMB

- `CFG_CRYPTO_DRIVER=y`
  Enable the HSE Crypto Driver to get the HUK and offload AES-CBC operations to the crypto engine

Command example:

```
make CROSS_COMPILE64=/path/to/your/toolchain/dir/bin/aarch64-none-linux-gnu- ARCH=arm \
PLATFORM=s32 PLATFORM_FLAVOR=s32g3 CFG_CRYPTO_DRIVER=y CFG_RPMB_FS=y CFG_RPMB_WRITE_KEY=y \
CFG_REE_FS=n
```

Including the HSE Firmware in the fip image is a prerequisite to using the RPMB Secure Storage feature. Please refer to the HSE Crypto Driver in OP-TEE section for instructions on how to build a fip image with HSE Firmware & optee_os included.

27 Adaptive Autosar Yocto Layer

The AUTOSAR Adaptive Platform (AP) is the standardized platform for microprocessor-based ECUs supporting use cases like highly automated and autonomous driving as well as high speed on-board and off-board communication.

27.1 Building the Linux image containing the Adaptive Autosar R20-11 support by Yocto

To build the Adaptive Autosar R20-11 image, please follow the steps below:

1. In the Yocto directory, get the Auto Yocto Linux BSP with meta-aa-integration repository:

```
repo init -u https://github.com/nxp-auto-linux/auto_yocto_bsp -b release/bsp36.0_cd
-m aa-integration.xml
repo sync
```

2. Download the Adaptive Autosar R20-11 official source code and yocto layers:

ara-api	Source code for Adaptive Autosar Platform Demonstrator
sample-applications	Source code for Adaptive Autosar sample applications
yocto-layer	yocto-layers/meta-ara

External location:
https://www.autosar.org/standards/adaptive-platform/

and place them under “sources” directory of Auto Yocto BSP distribution.
Note: An Adaptive Autosar license must be obtained in order to download the official Adaptive Autosar R20-11 source code.

3. Create build folder:

```
source nxp-setup-alb.sh -m <machine> -e "yocto-layers/meta-ara"
```

4. Build the image:

```
bitbake <image-name>
```

where <image_name> is one of the following:

core-image-apd-minimal	core-image-apd-minimal-radar
core-image-apd-minimal-fusion	core-image-apd-minimal-radar-fusion
core-image-apd-ecu1-scenario2	core-image-apd-ecu2-scenario2

27.2 Adding Adaptive Autosar R20-11 over an existing Yocto distribution

In the case an existing Yocto distribution has already built without the Adaptive Autosar support, the Adaptive Autosar can be added to the distribution by following the steps below:

1. Get the meta-aa-integration repository: clone the repository with the following commands:

```
git clone https://github.com/nxp-auto-linux/meta-aa-integration -b release/bsp36.0_cd
cd meta-aa-integration
git checkout <bsp_version>
```

   with <bsp_version> is the version of the Linux BSP. The Linux BSP currently is 'bsp36.0_cd'.

2. Add the new meta-aa-integration layer in the existing Yocto distribution by editing build folder's *conf/bblayers.conf* directly:

```
BBLAYERS ?= " \
  ...
  <path_to_repository>/meta-aa-integration \
  "
```

3. Follow step 2 from section "Building the Linux image containing the Adaptive Autosar R20-11 support by Yocto" in order to download and specify the path to the Adaptive Autosar source code and yocto layers

4. Make the following additions to the Adaptive Autosar Yocto layers in order to add Yocto Gatesgarth compatibility:

   (a) in *yocto-layers/meta-ara/conf/layer.conf* :

```
-  LAYERSERIES_COMPAT_apd = "zeus"
+ LAYERSERIES_COMPAT_apd = "zeus gatesgarth"
```

5. Continue to step 4 of the previous section "Building Adaptive Autosar R20-11" to build the image

28 Optional Features

28.1 Unrestricted userspace access to hardware registers

For debugging purposes, unrestricted access to hardware registers can be enabled in Linux. Invoke menuconfig before building the kernel (see "Invoke menuconfig to setup specific configuration for Linux".), and enable 'S32_DEBUG_REGACCESS' from 'Device Drivers: SOC (System On Chip) specific Drivers: NXP SoC drivers'.

After booting, the files in */sys/kernel/s32_regaccess* can be used to access hardware registers. Usage:

```
echo '0x<address> <access_width>' > /sys/kernel/s32_regaccess/read
echo '0x<address> <access_width> 0x<value>' > /sys/kernel/s32_regaccess/write
```

Use with caution. An incorrect access width, an incorrect address or an access to a device without an enabled clock may lead to a kernel crash.

28.2 SysVInit setup

Currently we are using the SystemD manager, by default, as inherited from Yocto, but Yocto Community also supports SysVInit manager. It is a matter of configuration on the Linux BSP to select Systemd or SysVInit manager. This configuration is based on Yocto upstream community support.

NOTE: SysVInit is not part of the testing and validation of Linux BSP releases.

SysVInit manager can be enabled by adding to conf/local.conf the following settings:

```
DISTRO_FEATURES:remove = " systemd"
DISTRO_FEATURES_BACKFILL_CONSIDERED:remove = " sysvinit"
VIRTUAL-RUNTIME_init_manager = "sysvinit"
VIRTUAL-RUNTIME_initscripts = "initscripts"
```

28.3 DM-Verity

DM-Verity is a security feature meant to provide block integrity checking for Linux's rootfs. Each block of data is hashed and subsequently added in a hash-tree. The root node (or the root hash) is calculated based on every block hash. One change in a single device block leads to a different root hash overall. This way, only the root hash needs to be verified to guarantee the integrity of the whole hash tree.

The verification data (the hashes) are appended at the end of the rootfs. At boot time, the Linux kernel will verify the integrity of the rootfs by checking the appended hash values against values computed at boot time. The hash function used in this process is SHA-256 and the Linux kernel will use the sha256 implementation with the highest

crypto algorithm priority. If hardware acceleration is not available, 'sha256-generic' will be used.

DM-Verity can be enabled in Yocto by adding the following distro feature in *conf/local.conf*:

```
DISTRO_FEATURES:append = " dm-verity"
```

Please note that the newly generated image will have a .wic extension instead of the usual .sdcard. Still, the image will boot from an SD Card. Currently, only fsl-image-base image type supports dm-verity.

For more information on dm-verity support in Linux, please consult https://www.kernel.org/doc/html/v5.10/admin-guide/device-mapper/verity.html

29 Demo applications

The demo applications illustrate some basic capabilities over the Linux BSP.
These samples had been structured in a modular way in order to be not just easy to read but even easy to understand by the users of our product.

Because the processor of the S32G3 platform is an ARM Cortex-A53, we use the ARM compiler to generate the binary files of our code examples.

The demos can be automatically built from Yocto or manually using a cross-compiler (see section 3.2.1).

Building the Demo Applications from Yocto:

In order to build the demos and add them to the *.sdcard* image, simply add the following lines to *conf/local.conf*:

```
COMPATIBLE_MACHINE_pn-demo-samples = "${MACHINE}"
CORE_IMAGE_EXTRA_INSTALL += "demo-samples"
```

Then build any image, e.g. *fsl-image-auto*. The demo applications will be deployed on target in directory */opt/samples*.

Manually Building the Demo Applications:

In the root directory will be a global *Makefile* which will recursively call each sample's *Makefile*. The common variables will be defined in *common.mk* file.
In order to build all samples you can simply call "make" and "make clean" to delete all the already compiled object files. To personalize the build operation you can call "make sample_name" and will be executed just the sample's specified Makefile.
Some examples of building the samples:

- `make CFLAGS="-I" CROSS_COMPILE="aarch64-none-linux-gnu-"`

- `make SYSROOT="/opt/rootfs"`

29.1 Multicore sample application

This C sample illustrates the benefits of multicore processing using kernel level threads. It is known that the naive multiplication algorithm of two bidimensional arrays has a $\Theta(n^3)$ time complexity because of its three 'for' structures. To push this case to the limits, we simultaneously take five different bidimensional arrays from five different files in

order to multiply them several times.

The loop number is given as an argument for this sample. For example you can run ./multicore 1000000 and the array will be multiplied by 1000000 times.

We consider that our machine has a processor with 4 cores. Cores 0 to 3 have assigned the first four threads while the fifth one will go on the first core that will become free (almost a random choice).

In *functions.c* are the functions that will be used.

- void make_matrix(char path[], int matrix[max_size][max_size]) is a function that reads from the file a bidimensional array. This function receives as parameters the path of the file and the resulting matrix.
- void display_matrix(int matrix[max_size][max_size], int size) writes to stdout the matrix whose address will be the parameters of this function
- void multi_matrix(int matrix[max_size][max_size], int size) deals with multiplying a matrix with a given size
- void* thread_multi_matrix(void *params) is the multi_matrix() alternative for threads implementation; parameters are given through a structure that contains the size of the matrix, the matrix itself and the id of the thread associate.

Output of this sample:

```
root@s32g399aevb3:/opt/samples# ./multicore 1000000

loop_number = 1000000

The number of cores: 4
The online cores :
0-3

The offline cores :



Without threads- - - > 12933ms

With threads- - - > 4297ms

root@s32g399aevb3:/opt/samples#
```

For details on this demo, please refer to the Multicore sample application README file.

29.2 Networking sample applications

29.2.1 One to one chat application

The one to one chat application will simulate a TCP server-client chat. The connection will be established through sockets and the communication is basically a stop-and-wait sequence: both server and client wait for a response after sending a message.

The messages for both the client and the server are received as input from the console.

In order to run this demo on target, it is recommended to have networking enabled and use two different consoles, e.g. the serial console for the server and a ssh session for the client. Otherwise there will be issues when writing the input messages for the two processes.

Execution flow:

- From the server compilation will result an executable file that receives as parameter the port.
  Practically, after the "make"/"make server"/"make client" command, the server must be turned on and receive number for the port. For example, in order to use the port with the number 1923, the right command will be "./server 1923"
- In order to find the server(if it is available), the client should know the name or its IP address. There are two arguments that should be specified at the execution: the IP address and the and the port number used for communication.
- Once the server is turned on, the client can connect to it. Assuming that the IP address of the server is 192.168.0.1, the client will connect with the command "./client 192.168.0.1 1923". After that, the client can start the conversation.

Example:

- Start server using port 1923:

```
./server 1923
```

- Start client on the same host:

```
./client 127.0.0.1 1923
```

- Write messages for the client then the server, alternately.

For details on this demo, please refer to the Networking sample application README file.

Output of this demo:

- Server side console:

```
root@s32g399aevb3:/opt/samples# ./server 1923
Client: test1
Server: reply1
Client: test2
Server: reply2
Client closed the connection
root@s32g399aevb3:/opt/samples#
```

- Client side console:

```
root@s32g399aevb3:/opt/samples# ./client 127.0.0.1 1923
Client: test1
Server: reply1
Client: test2
Server: reply2
Client: quit
root@s32g399aevb3:/opt/samples#
```

29.2.2 One to many echo application

The one to many echo application will simulate a TCP server-multiclient chat. The connection will be established through sockets and the communication is basically a single sequence: The client sends some data to the server and the server ping it back with a standard package of data.

Execution flow:

- From the server compilation will result an executable file that receives as parameter the port for communication.
- In order to find the server (if it is available), the client should know the name or its IP address. There are three arguments that should be specified at the execution, the IP address, the number of the port that the socket will use and an identification name.
- Once the server is turned on, the clients can connect to it. Assuming that the IP address of the server is 192.168.0.1, the clients will connect with the command "./client 192.168.0.1 1923".

Example:

- Start server using port 1923:

```
./server 1923
```

- Start clients on the same host, each on a different console/ssh session:

```
./client 127.0.0.1 1923
```

- Write messages for the clients, alternately. The server will echo them.

For details on this demo, please refer to the Networking-extended sample application README file.

Output of this demo:

- Server and 1st client console:

```
root@s32g399aevb3:/opt/samples# ./server_multiclient 1923 &
[1] 477
root@s32g399aevb3:/opt/samples#
root@s32g399aevb3:/opt/samples# ./client 127.0.0.1 1923
Client: Client4 connected
Client5 connected
```

```
Client: test1
Client4: test1
Server: test1
Client: Client5: test2

Client: test3
Client4: test3
Server: test3
Client: Client5: test4

Client: quit
Client4 disconnected!
root@s32g399aevb3:/opt/samples#
root@s32g399aevb3:/opt/samples# Client5 disconnected!

root@s32g399aevb3:/opt/samples# jobs
[1]+  Running                 ./server_multiclient 1923 &
root@s32g399aevb3:/opt/samples# fg
./server_multiclient 1923

^C
root@s32g399aevb3:/opt/samples#
```

- 2nd client console:

```
root@s32g399aevb3:/opt/samples# ./client 127.0.0.1 1923
Client: test2
Server: test2
Client: test4
Server: test4
Client:
Client: quit
root@s32g399aevb3:/opt/samples#
```

29.3 GPIO libgpiod application

This demo uses the "Libgpiod Library" C API for interacting with GPIOs. This application can be used to manipulate GPIO pins, that is to set outputs and read inputs.

Apart from identifying 2 GPIO pins to be configured as input and output, the GPIO chip which they are part of should be also identified and passed as the first argument to the application. To list the available GPIO chips use the 'gpiodetect' command.

Let's consider that IPIN is the input pin and OPIN is the output pin, both being part of 'gpiochip0'. Launch the demo instances:

```
root@s32g399aevb3:/opt/samples# ./gpio_libgpiod gpiochip0 OPIN 1 &
root@s32g399aevb3:/opt/samples# ./gpio_libgpiod gpiochip0 IPIN
```

The former command exports OPIN, sets its direction to "out", writes 1 as the initial pin value, then toggles it for 50 times. The latter command reads and prints each second the value of IPIN. The values of IPIN should alternate between 0 and 1 since the pins are connected.

For more details on this demo, please refer to the gpio_libgpiod/README file.