NXP Semiconductors
User's Guide

Document identifier: IIoTBaremetalUG
Rev. 1.10, 12/2020

Industrial IoT Baremetal Framework Developer Guide

Contents

Chapter 1
Introduction

This document provides an overview of the OpenIL Baremetal framework, the features it supports, and getting started with baremetal framework using the supported NXP Layerscape platforms. It also describes how to run a sample baremetal framework on the host environment and develop customer specific applications based on QorIQ Layerscape baremetal framework.

1.1 QorIQ layerscape processor

QorIQ layerscape processors are based on Arm® technology.

With the addition of NXP's next-generation QorIQ Layerscape series processors built on Arm core technology, the processor portfolio extends performance to the smallest form factor—from power-constrained networking and industrial applications to new virtualized networks and embedded systems requiring an advanced datapath and network peripheral interfaces.

1.2 Baremetal framework for QorIQ layerscape

The baremetal framework targets to support the scenarios that need low latency, real-time response, and high-performance. There is no OS running on the cores and customer specific application runs on that directly. The figure below depicts the baremetal framework architecture.

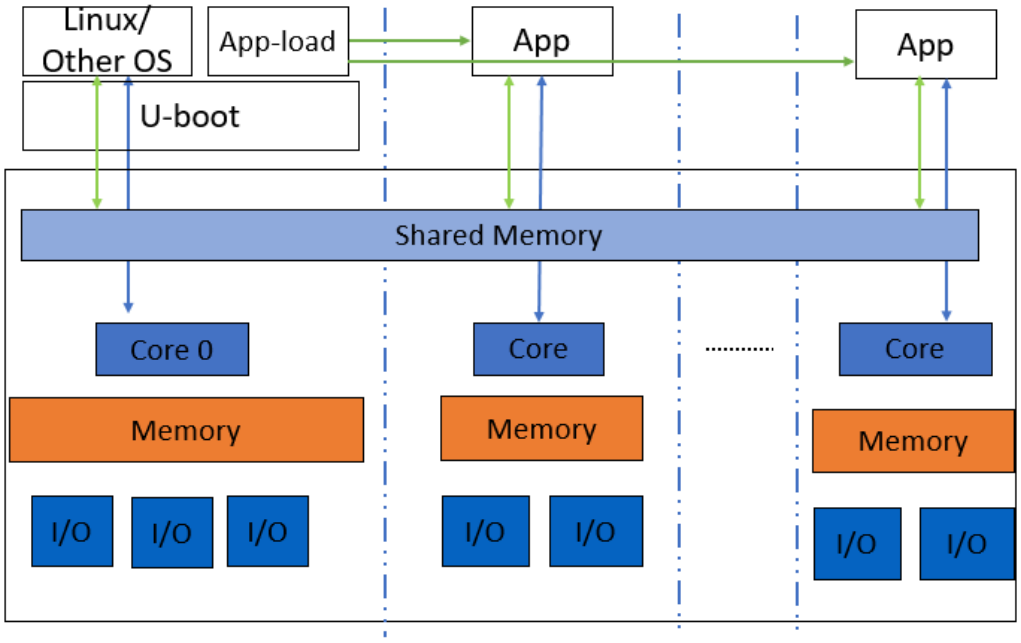


Figure 1. Baremetal framework architecture

The main features of the baremetal framework are as follows:

- Core0 runs as master which runs the operating system such as Linux, Vxworks.
- Slave cores run on the baremetal application.
- Easy assignment of different IP blocks to different cores.
- Interrupts between different cores and high-performance mechanism for data transfer.
- Different UART for core0 and slave cores for easy debug.

- Communication via shared memory.

The master core0 runs the U-Boot, it then loads the baremetal application to the slave cores and starts the baremetal application. The following figure depicts the boot flow diagram:

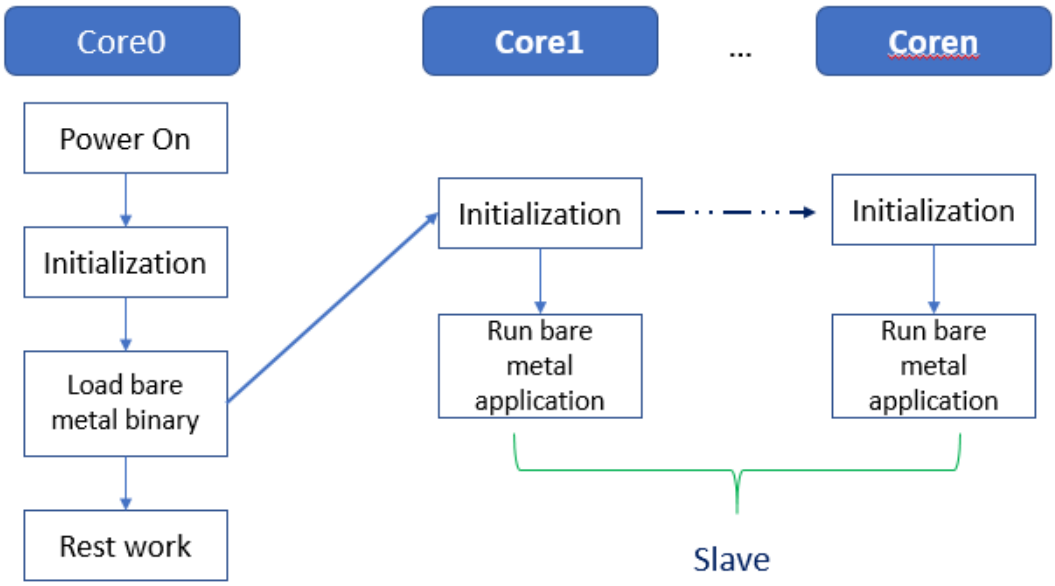


Figure 2. Baremetal framework boot flow diagram

1.3 Supported processors and boards

The table below lists the industrial IoT features supported by various NXP processors and boards.

Table 1. Industrial IoT features supported by NXP processors

Processor	Board	Main features supported
LS1021A	LS1021A-IoT	GPIO, IRQ, IPI, data transfer, IFC, I2C, UART, QSPI, USB, PCIE, Flexcan
LS1043A	LS1043ARDB	IRQ, IPI, data transfer, Ethernet, IFC, I2C, UART, FMan, USB, PCIe
LS1046A	LS1046ARDB	IRQ, IPI, data transfer, Ethernet, IFC, I2C, UART, FMan, QSPI, USB, PCIe
LS1028A	LS1028ARDB	I2C, UART, ENETC, IPI, data transfer, SAI
LX2160A/Rev2	LX2160ARDB	UART, IPI, data transfer
I.MX6Q	IMX6Q-SABRESD	UART, IPI, data transfer
I.MX8M Plus	I.MX8MP-EVK	UART, IPI, data transfer, Ethernet, GPIO
I.MX8M Mini	IMX8MM-EVK	UART, IPI, data transfer, Ethernet, GPIO

Chapter 2
Getting started

This section describes how to set up the environment and run the baremetal examples on slave cores (assuming that the core0 is the master core and the other cores are the slave cores).

2.1 Hardware and software requirements

The following are required for running baremetal framework scenarios:

- Hardware: LS1021A-IoT, LS1043ARDB, LS1046ARDB, LS1028ARDB, i.mx6q-sabresd, I.MX8MM-EVK, I.MX8MP-EVK, or other QorIQ Layerscape boards and serial cables.
- Software: OpenIL v1.4 release or later.

2.2 Hardware setup

This section describes the hardware setup required for the NXP boards for running the baremetal framework examples.

2.2.1 LS1021A-IoT board

Two serial cables are needed. One is used for core0, which connects to USB0/K22 port for UART0, and another one is used for slave cores, which connect J8 and J17 together for UART1. The table below describes the pins for UART1.

Table 2. UART pins

Pin name	Function
J8 pin7	Ground
J17 pin1	Uart1_SIN
J17 pin2	Uart1_SOUT

The figure below depicts the UART1 hardware connections.

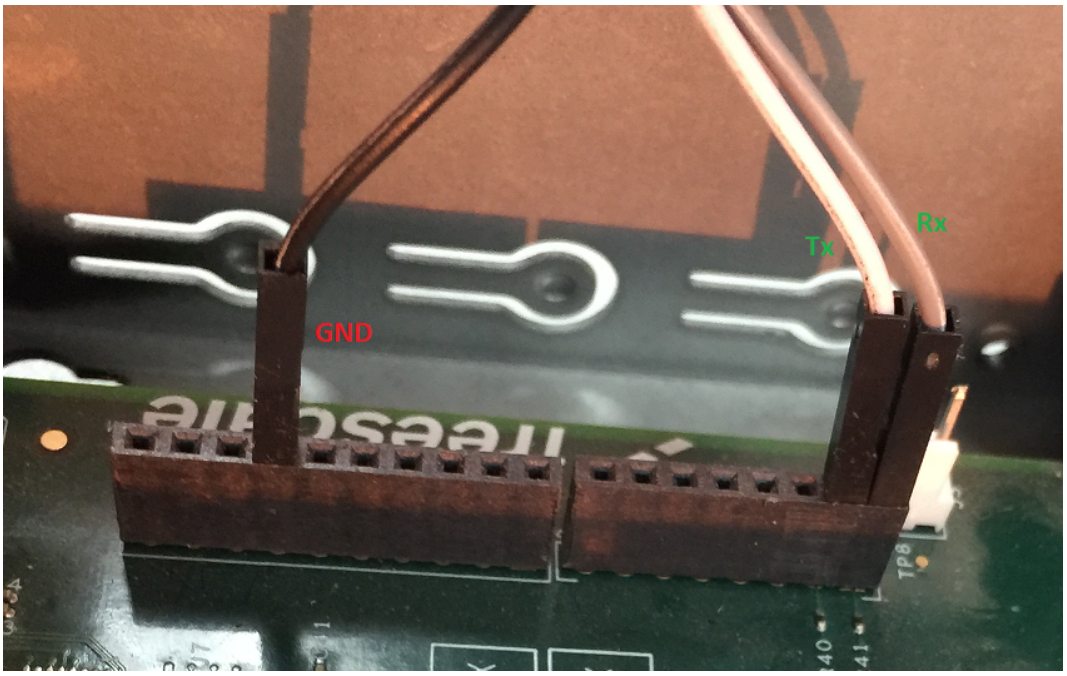


Figure 3. UART1 hardware connection

In order to test GPIO, connect the two pins, GPIO24 and GPIO25 together, which are through J502.3 and J502.5.

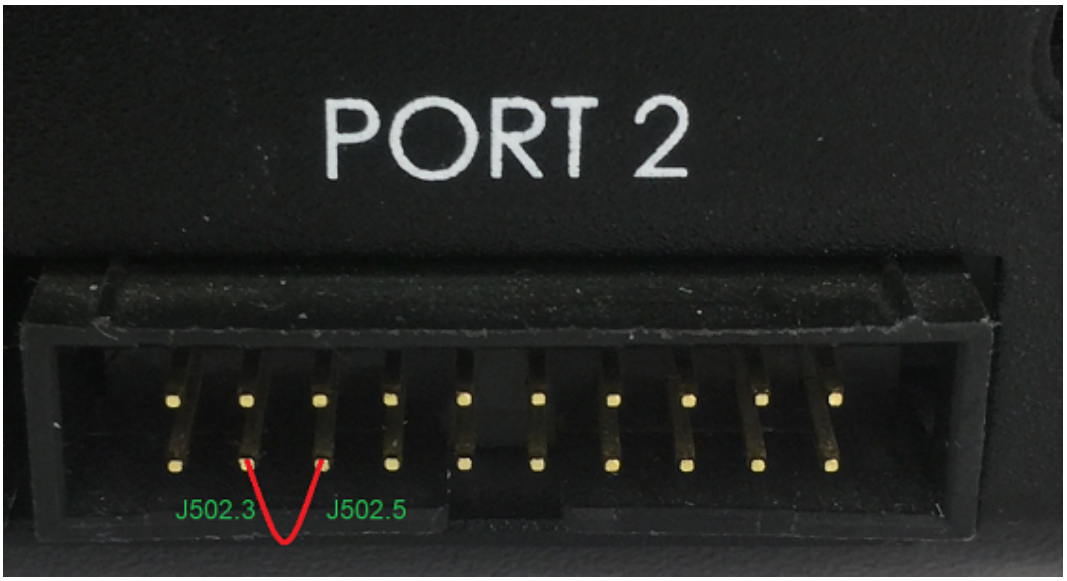
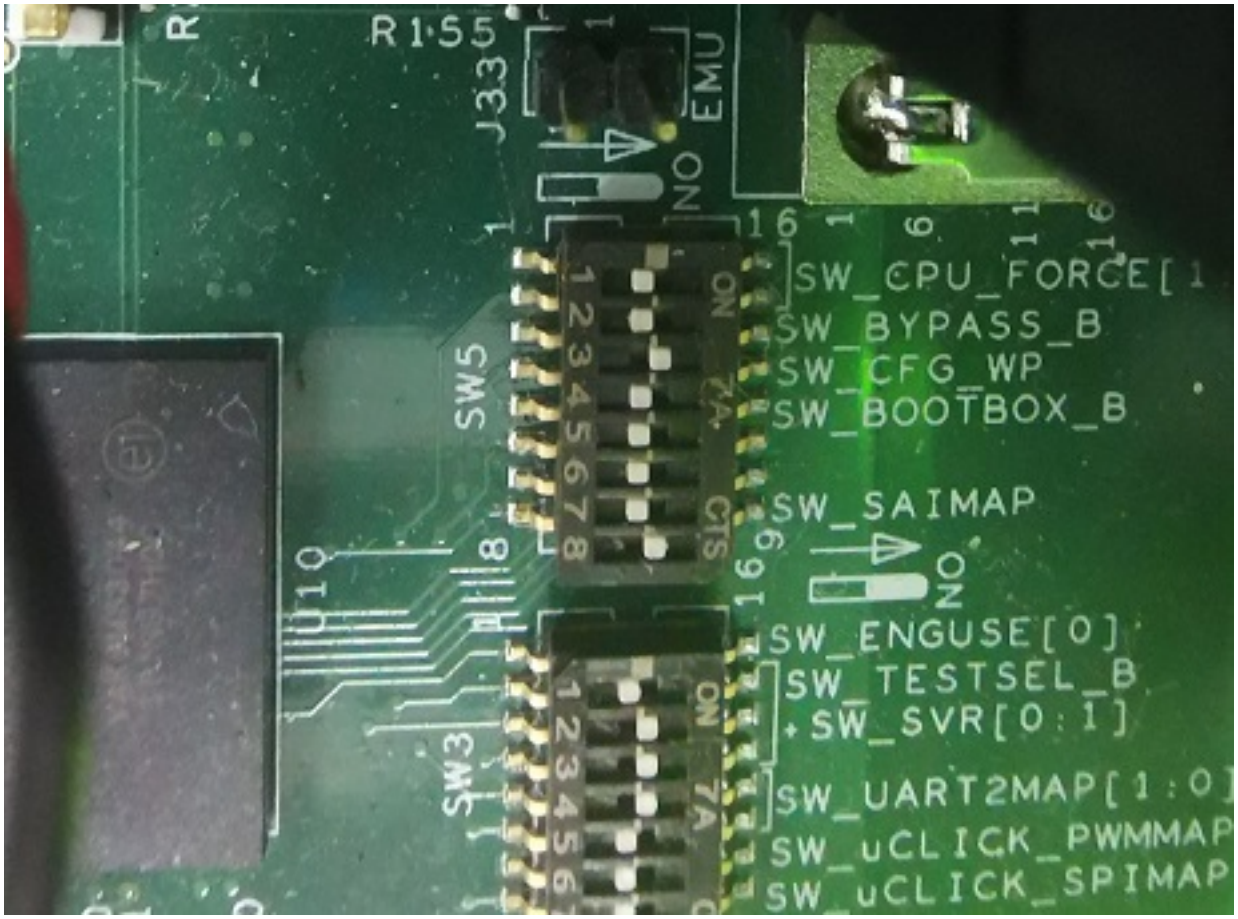


Figure 4. GPIO24 and GPIO25 connection on LS1021A-IoT

2.2.2 LS1028ARDB, LX2160ARDB,LS1043ARDB, or LS1046ARDB

In case, either the LS1028ARDB, LX2160ARDB,LS1043ARDB, or LS1046ARDB is used for developing the OpenIL Baremetal framework, two serial cables are needed. One is used for core0, which connects to UART1 port, and the other one is used for slave cores, which connects to the UART2 port.

To support SAI feature on LS1028ARDB, the SW5_8 switch needs to set to ON status.



2.2.3 i.mx6q-sabresd board

Two serial cables are needed. One is used for core0, which connects to USB port for UART1, and another one is used for slave cores, which connect J8 and U19 together for UART3. The table below describes the pins for UART3.

Table 3. Pins for UART3

Pin	Description
U19 pin36	Ground
U19 pin30	UART3_RXD
U19 pin31	UART3_TXD

The figure below depicts the UART3 hardware connections.

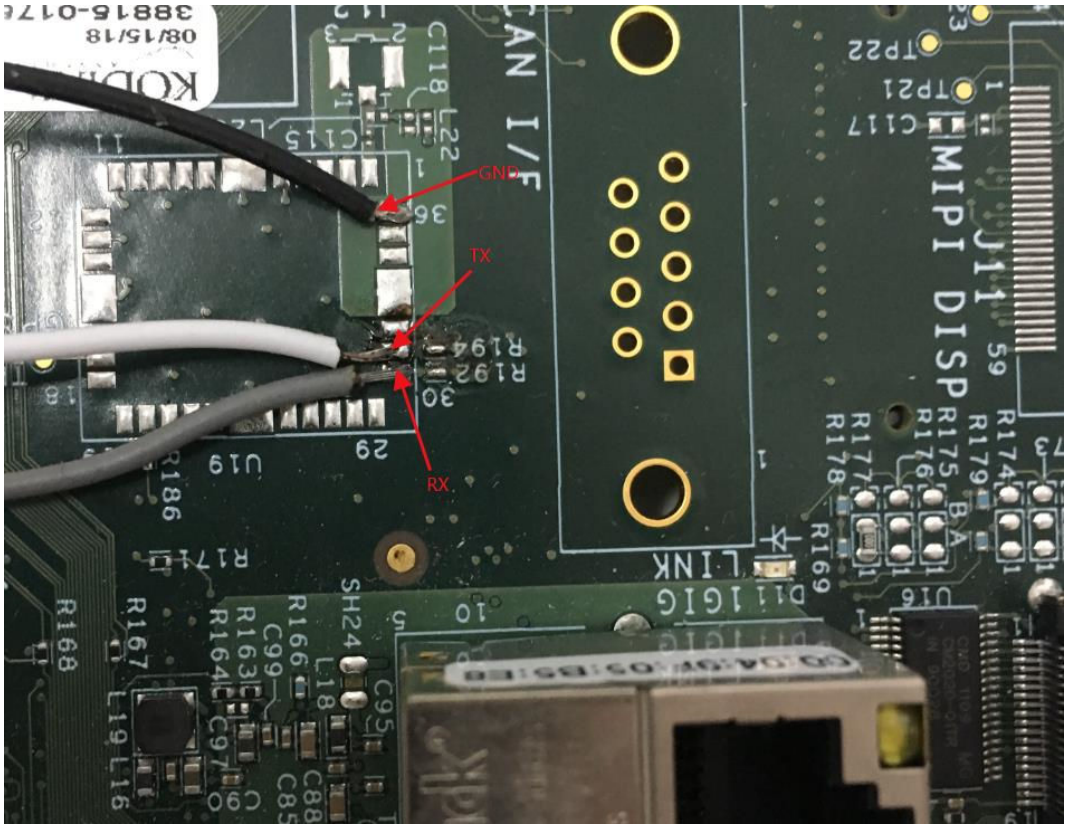


Figure 5. UART3 hardware connections

2.2.4 i.MX8MM-EVK and i.MX8MP-EVK board

1. i.MX8MPEVK

There is one USB MicroB Debug port on board, four Uart ports can be found when the MicroB cable connects to PC,

```
/dev/ttyUSB0
/dev/ttyUSB1
/dev/ttyUSB2
/dev/ttyUSB3
```

/dev/ttyUSB2 is used for core0 (master core), /dev/ttyUSB3 is used for core1, core2 and core3 (slave core)

2. i.MX8MMEVK

There is one USB MicroB Debug port on board, four Uart ports can be found when the MicroB cable connects to PC,

```
/dev/ttyUSB0
/dev/ttyUSB1
```

/dev/ttyUSB1 is used for core0 (master core), /dev/ttyUSB0 is used for core1, core2 and core3 (slave core)

3. GPIO setup

For i.MX8MM-EVK or i.MX8MP-EVK gpio test, pin7 and pin8 of J1003 should be connected directly.

2.3 Building the baremetal images from U-Boot source code

There are two methods to build the baremetal images. One method is to compile the images in a standalone way, which is described in this section. Another method is to build the bare metal images using OpenIL framework, which is described in Building the image through OpenIL.

2.3.1 Building U-Boot binary for the master core

Perform the steps mentioned below:

1. Download U-Boot source from the path below:

   https://github.com/openil/u-boot.git.

2. Check it out to the tag `OpenIL-V1.10-u-boot-202012` for Layscape and I.mx6q or `OpenIL-V1.10-u-boot-imx-202012` for imx8mmevk and imx8mpevk.

3. Configure cross-toolchain on your host environment.

4. Then, build the U-Boot and flash it into the SD card as SD boot for core0 using the commands below:

```
/* building u-boot image for ls1021aiot board */
    $make ls1021aiot_sdcard_baremetal_defconfig
    $make
```

NOTE

The default setting enables CAN3/4 feature for baremetal. If you want to enable the GPIO baremetal feature, you need to run the following commands:

```
$make ls1021aiot_sdcard_baremetal_defconfig
$make menuconfig
ARM architecture --->
    Baremetal feature select (GPIO3_15/27 supported) --->
$make
```

5. For the LS1028ARDB, LS1043ARDB, LS1046ARDB, or LX2160ARDB use the following commands:

```
/* building u-boot image for ls1028ardb board */
    $make ls1028ardb_tfa_baremetal_defconfig
    $make

/* building u-boot image for ls1043ardb board */
    $make ls1043ardb_tfa_baremetal_defconfig
    $make

/* building u-boot image for ls1046ardb board */
    $make ls1046ardb_tfa_baremetal_defconfig
    $make


/* building u-boot image for lx2160ardb board */
    $make lx2160ardb_tfa_baremetal_defconfig
    $make
```

6. For the i.mx6q-sabresd, use the following commands:

```
/* building u-boot image for i.mx6q-sabresd board */
   $make imx6q-sabresd_baremetal_defconfig
```

```
    $make
```

7. For the i.MX8MM-EVK board, use the following commands:

```
/* building u-boot image for i.MX8MM-EVK RevC board */
   $make imx8mm_evk_baremetal_defconfig
   $make

/* building u-boot image for i.MX8MM-EVK RevB board */
   $make imx8mm_evk_revb_baremetal_defconfig
   $make
```

8. For the i.MX8MP-EVK board, use the following commands:

```
/* building u-boot image for i.MX8MP-EVK board */
   $make imx8mp_evk_baremetal_defconfig
   $make
```

2.3.2 Building baremetal binary for slave cores

Perform the steps mentioned below:

1. Download the project source from the following path:

   https://github.com/openil/u-boot.git.

2. Check it out to the tag:
   - For Layerscape SoCs and imx6q: `OpenIL-v1.10-Baremetal-202012`
   - For imx8mmevk and imx8mpevk: `OpenIL-v1.10-Baremetal-imx-202012`
3. Configure cross-toolchain on your host environment.
4. Then, run the following commands:

```
$git checkout baremetal

/* build baremetal image for ls1021-iot board */
   $make ls1021aiot_baremetal_defconfig
   $make

/* build baremetal image for ls1028ardb board */
   $make ls1028ardb_sdcard_baremetal_defconfig
   $make

/* build baremetal image for ls1043ardb board */
   $make ls1043ardb_defconfig
   $make

/* build baremetal image for ls1046ardb board * /
   $make ls1046ardb_baremetal_defconfig
   $make

/* build baremetal image for i.mx6q-sabresd board * /
   $make mx6sabresd_baremetal_defconfig
   $make

/* build baremetal image for lx2160ardb board * /
```

```
   $make lx2160ardb_tfa_baremetal_defconfig
   $make

/* build baremetal image for I.MX8MM-EVK board * /
   $make imx8mm_evk_baremetal_defconfig  /* RevC */
or $make imx8mm_evk_revb_baremetal_defconfig  /* RevB */
   $make

/* build baremetal image for I.MX8MP-EVK board * /
   $make imx8mp_evk_baremetal_defconfig
   $make
```

5. Finally, the file, `U-Boot.bin` used for bare metal is generated. Copy it to the *tftp* server directory.

2.4 Building the image through OpenIL

There are two methods to build the baremetal images. One method is to compile the images in a standalone way which is described in Building the baremetal images from U-Boot source code. The second method is to build the bare metal images using OpenIL framework, which is described in this section.

The OpenIL project (Open Industry Linux) is designed for embedded industrial usage. It is an integrated Linux distribution for industry. With the OpenIL v1.1 or later releases, the Baremetal can be built and implemented conveniently.

Currently, there are two LS1021A-IoT baremetal defconfig files, a LS1028ARDB baremetal defconfig file, a LS1043ardb baremetal defconfig file, a LS1046ardb baremetal defconfig file, and an i.mx6q-sabresd baremetal defconfig file, as listed below:

- `configs/nxp_ls1021aiot_baremetal_defconfig`
- `configs/nxp_ls1028ardb_baremetal-64b_defconfig`
- `configs/nxp_ls1028ardb_baremetal_sai-64b_defconfig`
- `configs/nxp_ls1043ardb_baremetal-64b_defconfig`
- `configs/nxp_ls1046ardb_baremetal-64b_defconfig`
- `configs/nxp_lx2160ardb_baremetal-64b_defconfig`
- `configs/nxp_lx2160ardb_rev2_baremetal-64b_defconfig`
- `configs/imx6q-sabresd_baremetal_defconfig`
- `configs/imx8mmevk_baremetal_defconfig`
- `configs/imx8mmevk_revb_baremetal_defconfig`
- `configs/imx8mpevk_baremetal_defconfig`

2.4.1 Getting OpenIL

OpenIL latest release is available at the following URL:

https://github.com/openil/openil.git.

To follow development, make a clone of the *Git* repository using the command below:

```
$ git clone https://github.com/openil/openil.git
$ cd openil
# checkout to the tag: OpenIL-v1.10-202012
$ git checkout -b OpenIL-v1.10-202012 OpenIL-v1.10-202012
```

NOTE

- Build everything as a normal user. Root user permissions are not required for configuring and using OpenIL. By running all commands as a regular user, you protect your system against packages behaving badly during compile and installation.
- Do not use `make -jN` to build OpenIL as the top-level parallel `make` is currently not supported.

CAUTION

The parameter `PERL_MM_OPT` would be defined in case Perl `local::lib` is installed on your system. You should unset this option before starting Buildroot, otherwise the compilation of Perl related packages will fail. For example, you might encounter the error information of the PERL_MM_OPT parameter while running the `make` command in a host Linux environment such as this:

```
make[1]: *** [core-dependencies] Error 1
make: *** [_all] Error 2
```

To resolve it, just unset the PERL_MM_OPT option.

```
$ unset PERL_MM_OPT
```

2.4.2 Building the baremetal images

This section describes the steps for building the baremetal images for various boards such as LS1021A-IoT, LS1043ARDB, LS1046ARDB, LX2160ARDB, i.mx6q-sabresd I.mx8mpevk and I.mx8mmevk.

2.4.2.1 Building the baremetal images for LS1021-IoT board

The two LS1021A-IoT Baremetal default configuration files can be found in the *configs* directory.

• `configs/nxp_ls1021aiot_baremetal_defconfig`

The files include all the necessary U-Boot, Baremetal, kernel configurations, and application packages for the filesystem. Based on these files, you can build a complete Linux + Baremetal environment for the LS1021A-IoT platform without any major changes.

To build the final LS1021A-IoT Baremetal images, run the following commands:

```
$ cd openil
$ make nxp_ls1021aiot_baremetal_defconfig
$ make
# or make with a log
$ make 2>&1 | tee build.log
```

NOTE

The `make clean` command should be implemented before any other new compilation.

After the correct compilation, you can find all the images for the platform in the `output/images` directory.

Here is a view of the directory, *output/images/*:

```
├── bm-u-boot.bin --- baremetal image run on slave core
├── boot.vfat
├── ls1021a-iot.dtb --- dtb file for ls1021a-iot
├── rootfs.ext2
├── rootfs.ext2.gz
├── rootfs.ext2.gz.uboot --- ramdisk can be used for debug on master core
├── rootfs.ext4.gz -> rootfs.ext2.gz
├── rootfs.tar
├── sdcard.img --- entire image can be programmed into the SD
```

```
├── uboot-env.bin
├── u-boot-with-spl-pbl.bin --- uboot image for ls1021a-iot master core
├── uImage --- kernel image for ls1021a-iot master core
└── version.json
```

For more details about Open Industry Linux, refer to the document Open_Industrial_Linux_User_Guide.

2.4.2.2 Building the baremetal images for LS1028ARDB, LS1043ARDB, or LS1046ARDB

The baremetal default configuration files for LS1028ARDB, LS1043ARDB, or LS1046ARDB are located in the *configs* directory. These are:

- `configs/nxp_ls1028ardb_baremetal-64b_defconfig`
- `configs/nxp_ls1028ardb_baremetal_sai-64b_defconfig`
- `configs/nxp_ls1043ardb_baremetal-64b_defconfig`
- `configs/nxp_ls1046ardb_baremetal-64b_defconfig`

These files include all the necessary U-Boot, Baremetal, kernel configurations, and application packages for the filesystem. Based on these files, you can build a complete Linux + Baremetal environment for the LS1043A-RDB or LS1046A-RDB platform without any major changes.

Run the following commands to build the final LS1043ARDB or LS1043ARDB baremetal images:

```
$ cd openil
```

Then:

```
$ make nxp_ls1028ardb_baremetal-64b_defconfig
```

or

```
$ make nxp_ls1028ardb_baremetal_sai-64b_defconfig
```

or

```
$ make nxp_ls1043ardb_baremetal-64b_defconfig
```

or

```
$ make nxp_ls1046ardb_baremetal-64b_defconfig
```

Then, use:

```
$ make
# or make with a log
$ make 2>&1 | tee build.log
```

NOTE

The `make clean` command should be implemented before any other new compilation.

After the correct compilation, you can find all the images for the platform in the `output/images` directory.

Here is a view of the directory, *output/images/* for ls1043a-rdb:

```
├── bm-u-boot.bin          --- baremetal image run on slave core
├── bl2_sd.pbl             --- BL2 and RCW binary
├── fip.bin                --- BL31 and uboot image for ls1043ardb master core
├── Image                  --- Linux kernel Image
```

```
├── rcw_1000_default.bin
├── boot.vfat
├── fsl-ls1043a-rdb-sdk.dtb --- dtb file for ls1043a-rdb
├── rootfs.ext2
├── rootfs.ext2.gz
├── rootfs.ext2.gz.uboot    --- ramdisk can be used for debug on master core
├── rootfs.ext4.gz
├── rootfs.tar
├── sdcard.img              --- entire image can be programmed into the SD
├── uboot-env.bin
└── version.json
```

For more details about Open Industry Linux, refer to the document Open_Industrial_Linux_User_Guide.

2.4.2.3 Building the baremetal images for LX2160A-RDB board

The two LX2160A-RDB Baremetal default configuration files can be found in the *configs* directory.

- `configs/nxp_lx2160ardb_baremetal-64b_defconfig`
- `configs/nxp_lx2160ardb_rev2_baremetal-64b_defconfig`

The files include all the necessary U-Boot, Baremetal, kernel configurations, and application packages for the filesystem. Based on these files, you can build a complete Linux + Baremetal environment for the LX2160A-RDB platform without any major changes.

To build the final LX2160-RDB Baremetal images, run the following commands:

```
$ cd openil
$ make nxp_lx2160ardb_baremetal-64b_defconfig
# or
$ make nxp_lx2160ardb_rev2_baremetal-64b_defconfig
# build the images
$ make
# or make with a log
$ make 2>&1 | tee build.log
```

NOTE

The `make clean` command should be implemented before any other new compilation.

After the correct compilation, you can find all the images for the platform in the `output/images` directory.

Here is a view of the directory, *output/images/*:

```
├── bm-u-boot-dtb.bin       --- baremetal image run on slave core
├── boot.vfat
├── bl2_sd.pbl              --- BL2 and RCW binary
├── fsl-lx2160a-rdb.dtb     --- dtb file for lx2160a-rdb
├── rootfs.ext2
├── rootfs.ext2.gz
├── rootfs.ext2.gz.uboot    --- ramdisk can be used for debug on master core
├── rootfs.ext4.gz -> rootfs.ext2.gz
├── rootfs.tar
├── sdcard.img              --- entire image can be programmed into the SD
├── uboot-env.bin
├── fip.bin                 --- uboot image for lx2160a-rdb master core
└── Image                   --- kernel image for lx2160a-rdb master core
```

For more details about Open Industry Linux, refer to the document Open_Industrial_Linux_User_Guide.

2.4.2.4 Building the baremetal images for i.mx6q-sabresd board

The two i.mx6q-sabresd Baremetal default configuration files can be found in the `configs` directory.

`configs/imx6q-sabresd_baremetal_defconfig`

The files include all the necessary U-Boot, Baremetal, kernel configurations, and application packages for the filesystem. Based on these files, you can build a complete Linux + Baremetal environment for the i.mx6q-sabresd platform without any major changes. To build the final i.mx6q-sabresd Baremetal images, run the following commands:

```
$ cd openil $ make imx6q-sabresd_baremetal_defconfig
$ make # or make with a log $ make 2>&1 | tee build.log
```

NOTE

The `make clean` command should be implemented before any other new compilation.

After the correct compilation, you can find all the images for the platform in the output/images directory. Here is a view of the directory, `output/images/`:

```
├── bm-u-boot.bin --- baremetal image run on slave core
├── boot.vfat
├── imx6q-sabresd.dtb --- dtb file for imx6q-sabresd
├── rootfs.ext2
├── rootfs.ext2.gz
├── rootfs.ext4.gz -> rootfs.ext2.gz
├── rootfs.tar
├── sdcard.img --- entire image can be programmed into the SD
├── SPL
├── u-boot.bin --- uboot image for imx6q-sabresd master core
├── u-boot.img
├── zImage     --- kernel image for imx6q-sabresd master core
```

For more details about Open Industry Linux, refer to the document, Open_Industrial_Linux_User_Guide.

2.4.2.5 Building the baremetal images for I.MX8MM-EVK board

This section describes the steps to be followed for building the baremetal images for I.MX8MM-EVK board.

1. The baremetal default configuration files for I.MX8MM-EVK are located in the `configs` directory. These are:

   ```
   configs/imx8mmevk_baremetal_defconfig
   ```

   ```
   configs/imx8mmevk_revb_baremetal_defconfig
   ```

2. Run the following commands:

   - For imx8mm RevC:

     ```
     $make imx8mmevk_baremetal_defconfig
     ```

   - For imx8mm RevB:

     ```
     $make imx8mmevk_revb_baremetal_defconfig
     $make
     ```

NOTE

The `make clean` command should be implemented before any other new compilation.

After the correct compilation, you can find all the images for the platform in the output/images directory. Here is a view of the directory, `output/images/`:

```
├── bm-u-boot.bin               --- baremetal image run on slave core
├── bl31.bin                    --- ATF secure
├── boot.vfat
├── imx8mm-evk-baremetal.dtb    --- dtb file for imx6q-sabresd
├── lpddr4_pmu_train_fw.bin     --- DDR firmware
├── imx8-boot-sd.bin            --- boot image
├── rootfs.ext2
├── rootfs.ext4.gz -> rootfs.ext2
├── rootfs.tar
├── sdcard.img                  --- entire image can be programmed into the SD
├── u-boot.bin
├── u-boot.itb                  --- BL31 + uboot image
├── u-boot-nodtb.bin            --- dtb file in uboot
├── u-boot-spl.bin              --- uboot image for imx8mmevk master core
├── u-boot-spl-ddr.bin          --- uboot spl + ddr firmware
├── signed_hdmi_imx8m.bin       --- HDMI firmware
├── Image                       --- kernel image for imx8mmevk master core
```

For more details about Open Industry Linux, refer to the document, Open_Industrial_Linux_User_Guide.

2.4.2.6 Building the baremetal images for I.MX8MP-EVK board

This section describes the steps to be followed for building the baremetal images for I.MX8MP-EVK board. The baremetal default configuration files are located in the `configs` directory.

1. The configuration file is:

   ```
   configs/imx8mpevk_baremetal_defconfig
   ```

2. Run the following commands: for I.MX8MP-EVK:

   ```
   $make imx8mpevk_baremetal_defconfig
   $make
   ```

NOTE

The `make clean` command should be implemented before any other new compilation.

After the correct compilation, you can find all the images for the platform in the output/images directory. Here is a view of the directory, `output/images/`:

```
├── bm-u-boot.bin               --- baremetal image run on slave core
├── bl31.bin                    --- ATF secure
├── boot.vfat
├── imx8mp-evk-baremetal.dtb    --- dtb file for imx6q-sabresd
├── lpddr4_pmu_train_fw.bin     --- DDR firmware
├── imx8-boot-sd.bin            --- boot image
├── rootfs.ext2
├── rootfs.ext4.gz -> rootfs.ext2
├── rootfs.tar
├── sdcard.img                  --- entire image can be programmed into the SD
├── u-boot.bin
├── u-boot.itb                  --- BL31 + uboot image
├── u-boot-nodtb.bin            --- dtb file in uboot
├── u-boot-spl.bin              --- uboot image for imx8mmevk master core
├── u-boot-spl-ddr.bin          --- uboot spl + ddr firmware
├── signed_hdmi_imx8m.bin       --- HDMI firmware
```

```
                ├── Image                       --- kernel image for imx8mmevk
master core
```

For more details about Open Industry Linux, refer to the document, Open_Industrial_Linux_User_Guide.

2.4.3 Booting up the Linux with Baremetal

Use the following steps to bootup the Linux + Baremetal system with the images built from OpenIL.

For platforms that can be booted up from the SD card, there is just one step required to program the `sdcard.img` into SD card.

1. Insert an SD card (of at least 4 GB size) into any Linux host machine.
2. Then, run the following commands:

```
$ sudo dd if=./output/images/sdcard.img of=/dev/sdx
# or in some other host machine:
$ sudo dd if=./output/images/sdcard.img of=/dev/mmcblkx
# find the right SD Card device name in your host machine and replace the “sdx” or “mmcblkx”.
```

3. Then, insert the SD card into the target board (LS1021A-IoT) and power on.

After the above mentioned steps are complete, the Linux system is booted up on the master core (core 0), and the Baremetal system is booted up on slave core (core 1) automatically.

Chapter 3
Running examples

The following sections describe how to run the baremetal examples on the host environment for LS1021A-IoT board. Similar steps can be followed for LS1028A, LS1043A, and LS1046A reference design boards (RDB), imx6q-sabresd, imx8mmevk and imx8mpevk board.

3.1 Preparing the console

Prepare a USB to TTL serial line, connect the pins to LPUART of Arduino pins to use it as UART1, please refer to Hardware setup.

In order to change LPUART to UART pins, modify 44th byte of RCW to 0x00.

In current baremetal framework design, two UART ports are used as console. UART1 is used for master core and UART2 is used for slave cores.

- For LS1021A, UART2 is pin-muxed with LPUART, so need to change the RCW to enable UART2 on master core. This modification has been implemented in the code for LS1021A-IoT board, so no need to do any code modification for this board.
- For customer specific boards, you need to apply the change to the RCW file:

```
bit[354~356] = 000
bit[366~368] = 111
```

3.2 Running the bare metal binary

As described earlier, there are two methods to compile the baremetal framework, a standalone method and using OpenIL. These are described in Building the baremetal images from U-Boot source code and Building the image through OpenIL respectively. If you are using OpenIL to compile the baremetal image, the baremetal image is included in the *sdcard.img* and the master core runs the baremetal image on slave cores automatically. If using the standalone compilation method, you need to perform the steps below to run the baremetal binary from U-Boot prompt of master core:

1. After starting U-Boot on the master, download the bare metal image: `u-boot.bin` on 0x84000000 using the command below:

   ```
   => tftp 0x84000000 xxxx/u-boot.bin
   ```

   Where

   - `xxxx` is your tftp server directory.
   - 0x84000000 is the address of `CONFIG_SYS_TEXT_BASE` on bare metal for LayerScape platforms.

   Note: The address is 0x50200000 for i.MX8MPEVK and i.MX8MMEVK boards.

2. Then, start the baremetal cores using the command below:

   ```
   => cpu start 0x84000000
   ```

3. Last, the UART1 port displays the logs, and the bare metal application runs on slave cores successfully.

The figure below displays a sample output log.

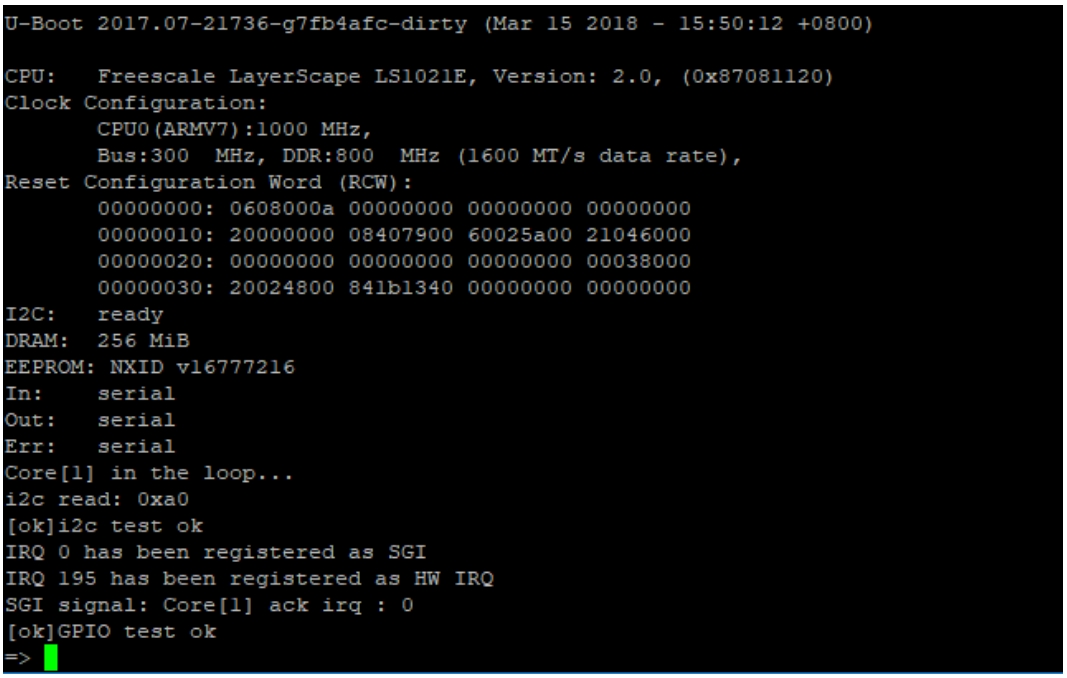


Figure 6. Baremetal output logs

Chapter 4
Development based on baremetal framework

This chapter describes how to develop customer specific application based on QorIQ layerscape baremetal framework.

4.1 Developing the baremetal application

The directory “app” in the U-boot repository includes the test cases for testing the I2C, GPIO and IRQ init features. You can write actual applications and store them in this directory.

4.2 Example software

This section describes how to analyze a GPIO sample code and use it to write the actual application.

4.2.1 Main file app.c

The file `<industry-Uboot path>/app/app.c`, is the main entrance for all applications. Users can modify the app.c file to add their applications.

- If using standalone method to build the baremetal image as described in Building the baremetal images from U-Boot source code, just change the directory to industry-Uboot path to check the app.c file.
- If using OpenIL to compile the baremetal binary, you need to change the directory to `output/build/bm-uboot-Baremetal-Framework/` to check the app.c file.

The following is a sample code of the file app.c that shows how to add the example test cases of I2C, IRQ, and GPIO.

```
void core1_main(void)
{
    test_i2c();
    test_irq_init();
    test_gpio();
    return;
}
```

4.2.2 Common header files

There are some common APIs provided by baremetal. The table below describes the header files that include the APIs.

Table 4. Common header file description

Header file	Description
`asm/io.h`	Read/Write IO APIs. For example, __raw_readb, __raw_writeb, out_be32, and in_be32.
`linux/string.h`	APIs for manipulating strings. For example, strlen, strcpy, and strcmp.
`linux/delay.h`	APIs used for small pauses. For example, udelay and mdelay.
`linux/types.h`	APIs specifying common types.

Table continues on the next page...

Table 4. Common header file description (continued)

Header file	Description
	For example, __u32 and __u64.
common.h	Common APIs. For example, printf and puts.

4.2.3 GPIO file

The file uboot/app/test_gpio.c is one example to test the GPIO feature, and shows how to write a GPIO application.

Here is an example for the ls1021aiot board:

On ls1021aiot board, first you need the GPIO header file, asm-generic/gpio.h, which includes all interfaces for the GPIO. Then, configure GPIO25 to OUT direction, and configure GPIO24 to IN direction. Last, by writing the value 1 or 0 to GPIO25, you can receive the same value from GPIO24.

The table below shows the APIs used in the file test_gpio.c example.

Table 5. GPIO APIs and their description

Function declaration	Description
gpio_request (ngpio, label)	Requests GPIO. • ngpio - The GPIO number, such as 25, that is for GPIO25. • label – the name of GPIOI request. Returns 0 if OK, -1 on error.
gpio_direction_output (ngpio, value)	Configures the direction of GPIO to OUT and writes the value to it. • ngpio - The GPIO number, such as 25,that is, for GPIO25. • value – the value written to this GPIO. Returns 0 if low, 1 if high, -1 on error.
gpio_direction_input (ngpio);	Configures the direction of GPIO to IN. ngpio - The GPIO number, such as 24, that is for GPIO24; Returns 0 if ok, -1 on error.
gpio_get_value (ngpio)	Reads the value. • ngpio - The GPIO number, such as 24, that is for GPIO24; • value – the value written to this GPIO. Returns 0 if low, 1 if high, -1 on error.
gpio_free (ngpio)	Frees the GPIO just requested. • ngpio - The GPIO number, such as 24, that is for GPIO24; Returns 0 if ok, -1 on error.

4.2.4 I2C file

The file uboot/app/test_i2c.c can be used as an example to test the I2C feature and shows how to write an I2C application.

On ls1021aiot board, first, you need to first include the I2C header file, `i2c.h`, which includes all interfaces for I2C. Then, you need to read a fixed data from offset 0 of Audio codec device (0x2A), if the data is 0xa0, the message `[ok]I2C test ok` is displayed on the console.

On ls1043ardb board, read a fixed data from offset 0 of INA220 device(0x40). If the data is 0x39, a message, `[ok]I2C test ok` is displayed on the console.

The table below shows the APIs used in the sample file, *test_i2c.c*.

Table 6. I2C APIs and their description

Function declaration	Description
int i2c_set_bus_num (unsigned int bus)	Sets the I2C bus. `bus`- bus index, zero based Returns 0 if OK, -1 on error.
int i2c_read (uint8_t chip, unsigned int addr, int alen, uint8_t *buffer, int len)	Read data from I2C device chip. • `chip` - I2C chip address, range 0..127 • `addr` - Memory (register) address within the chip • `alen` - Number of bytes to use for addr (typically 1, 2 for larger memories, 0 for register type devices with only one register) • `buffer` - Where to read/write the data • `len` - How many bytes to read/write Returns 0 if ok, not 0 on error
int i2c_write (uint8_t chip, unsigned int addr, int alen, uint8_t *buffer, int len)	Writes data to i2c device chip. • `chip` - I2C chip address, range 0..127 • `addr` - Memory (register) address within the chip • `alen` - Number of bytes to use for addr (typically 1, 2 for larger memories, 0 for register type devices with only one register) • `buffer` - Where to read/write the data • `len` - How many bytes to read/write Returns 0 if ok, not 0 on error

4.2.5 IRQ file

The file, *uboot/app/test_irq_init.c* is an example to test the IRQ and IPI (Inter-Processor Interrupts) feature, and shows how to write an IRQ application.

First, you need the IRQ's header `asm/interrupt-gic.h`, which includes all interfaces for IRQ. Then, register an IRQ function for SGI 0. After setting a SGI signal, the CPU gets this IRQ and runs the IRQ function. You also need to register a hardware interrupt function to show how to use the external hardware interrupt.

SGI IRQ is used for inter-processor interrupts, and it can only be used between bare metal cores. In case you want to communicate between baremetal core and Linux core, refer to ICC module . SGI IRQ id is 0-15, but 8 is reserved to be used for ICC.

NOTE

For i.MX8MM board, SGI IRQ id is 9.

The table below shows the APIs used in the sample file, *test_irq_init.c*.

Table 7. IRQ APIs and their description

Return type API name (parameter list)	Description
void gic_irq_register (int irq_num, void (*irq_handle)(int))	Registers an IRQ function. • `irq_num`- IRQ id, 0-15 for SGI, 16-31 for PPI, 32-1019 for SPI • `irq_handle` – IRQ function
void gic_set_sgi (int core_mask, u32 hw_irq)	Sets a SGI IRQ signal. • `core_mask` – target core mas • `hw_irq` – IRQ id
void gic_set_target (u32 core_mask, unsigned long hw_irq)	Sets the target core for hw IRQ. • `core_mask` – target core mas • `hw_irq` – IRQ id
void gic_set_type (unsigned long hw_irq)	Sets the type for hardware IRQ to identify whether the corresponding interrupt is edge-triggered or level-sensitive. • `hw_irq` – IRQ id

4.2.6 QSPI file

The file `uboot/app/test_qspi.c` provides an example that can be used to test the QSPI feature. The below steps show how to write a QSPI application:

1. First, locate the QSPI header files `spi_flash.h` and `spi.h`, which include all interfaces for QSPI.
2. Then, initialize the QSPI flash. You need to erase the corresponding flash area and confirm erase operation is successful.
3. Now, read or write to the flash with an offset of 0x3f00000 and size of 0x40000.

The table below shows the APIs used in the file `test_qsip.c` example.

Table 8. QSPI APIs

API name (type)	Description
spi_find_bus_and_cs(bus,cs, &bus_dev, &new)	Finds if the SPI device already exists. • “bus” - bus index, zero based. • “cs” – the value to chip select mode. • “bus_dev” - If the bus is found. • “new” – If the device is found. Returns `0` if ok, `-ENODEV` on error.
spi_flash_probe_bus_cs(bus, cs, speed, mode, &new)	Initializes the SPI flash device. • “bus” - bus index, zero based. • “cs” – the value to Chip Select mode. • “speed” – SPI flash speed, can use 0 or CONFIG_SF_DEFAULT_SPEED.

Table continues on the next page...

Table 8. QSPI APIs (continued)

API name (type)	Description
	• “mode” –SPI flash mode, can use 0 or CONFIG_SF_DEFAULT_MODE. • “new” – If the device is initialized. Returns `0` if ok, `-ENODEV` on error.
dev_get_uclass_priv(new)	Gets the SPI flash. • “new” - The device being initialized. Returns `flash` if ok , `NULL` on error.
spi_flash_erase(flash, offset, size)	Erases the specified location and length of the flash content, erases the content of all. • “flash” - Flash is being initialized. • “offset” – Flash offset address. • “size” - Erase the length of the data. Returns `0` if ok, `!0` on error.
spi_flash_read(flash, offset, len, vbuf)	Reads flash data to memory. • “flash” - The flash being initialized. • “offset” – Flash offset address. • “size” - Read the length of the data. Returns 0 if ok, !0 on error.
spi_flash_write(flash, offset, len, buf)	Writes memory data to flash. • “flash” - The flash being initialized. • “offset” – Flash offset address. • “size” - Write the length of the data. Returns `0` if ok, `!0` on error.

4.2.7 IFC

LS1043A and LS1046A has IFC controller. Both Nor flash and NAND flash are supported in LS1043ARDB, only NAND flash is supported in LS1046ARDB.

Nor and NAND flash message will be displayed during booting baremetal cores, like below:

```
1:NAND:  512 MiB
1:Flash: 128 MiB
```

or （LS1046ARDB）

```
1:NAND:  512 MiB
```

There is no example codes to test it, but we can use some commands to verify this features.

For LS1043ARDB Nor Flash (the map memory address is 0x60000000), below command can be used to verify it:

```
=> md 0x60000000
1:60000000: 55aa55aa 0001ee01 10001008 0000000a    .U.U............
1:60000010: 00000000 00000000 02005514 12400080    .........U....@.
1:60000020: 005002e0 002000c1 00000000 00000000    ..P... .........
1:60000030: 00000000 00880300 00000000 01110000    ................
1:60000040: 96000000 01000000 78015709 10e00000    .........W.x....
1:60000050: 00001809 08000000 18045709 9e000000    .........W......
1:60000060: 1c045709 9e000000 20045709 9e000000    .W.......W. ....
1:60000070: 00065709 00000000 04065709 00001060    .W.......W..`...
1:60000080: c000ee09 00440000 58015709 00220000    ......D..W.X..".
1:60000090: 40800089 01000000 40006108 f56b710a    ...@.....a.@.qk.
1:600000a0: ffffffff ffffffff ffffffff ffffffff    ................
```

For NAND flash on LS1043ARDB and LS1046ARDB, "nand" command can be used to verify it (nand erase, nand read, nand write, etc.):

```
=> nand info

1:Device 0: nand0, sector size 128 KiB
1:  Page size       2048 b
1:  OOB size          64 b
1:  Erase size    131072 b
1:  subpagesize     2048 b
1:  options     0x00004200
1:  bbt options 0x00028000
```

```
=> nand
1:nand - NAND sub-system


1:Usage:
nand info - show available NAND devices
nand device [dev] - show or set current device
nand read - addr off|partition size
nand write - addr off|partition size
    read/write 'size' bytes starting at offset 'off'
    to/from memory address 'addr', skipping bad blocks.
nand read.raw - addr off|partition [count]
nand write.raw[.noverify] - addr off|partition [count]
    Use read.raw/write.raw to avoid ECC and access the flash as-is.
nand erase[.spread] [clean] off size - erase 'size' bytes from offset 'off'
    With '.spread', erase enough for given file size, otherwise,
    'size' includes skipped bad blocks.
nand erase.part [clean] partition - erase entire mtd partition'
nand erase.chip [clean] - erase entire chip'
nand bad - show bad blocks
nand dump[.oob] off - dump page
nand scrub [-y] off size | scrub.part partition | scrub.chip
    really clean NAND erasing bad blocks (UNSAFE)
nand markbad off [...] - mark bad block(s) at offset (UNSAFE)
nand biterr off - make a bit error at offset
```

4.2.8 Ethernet

The file `uboot/app/test_net.c` provides an example to test the Ethernet feature and shows how to write a net application for using this feature.

Here is an example for the LS1043ARDB (or LS1046ARDB) board.

1. Connect one of Ethernet port of LS1043ARDB board to one host machine using Ethernet cable (For LS1046ARDB, the default ethact is FM1@DTSEC5, network cable should be connected to SGMII1 port; For LS1043ARDB, the default ethact is FM1@DTSEC3, network cable should be connected to RGMII1 port).
2. Configure the IP address of the host machine as 192.168.1.2.
3. Power up the LS1043ARDB board. If the network is connected, the mesage `host 192.168.1.2 is alive` is displayed on the console.
4. The IP address of the board and host machine are defined in the file `test_net.c`. In this file, modify the IP address of LS1043ARDB board using variable `ipaddr` and change the IP address of host machine using variable `ping_ip`.

The table below lists the Net APIs and their description.

Table 9. Net APIs and their description

API name (type)	Description
void net_init (void)	Initializes the network
int net_loop (enum proto_t protocol)	Main network processing loop. • enum proto_t protocol - protocol type
int eth_receive (void *packet, int length)	Read data from NIC device chip. • void *packet • length - Network packet length Returns length
int eth_send (void *packet, int length)	Writes data to NIC device chip. • packet - Network packet length • length - Network packet length Returns length.

4.2.9 USB file

The file `uboot/app/test_usb.c` provides an example that can be used to test the USB features. The steps below show how to write a USB application:

1. Connect a USB disk to the USB port.
2. Include the header file, `usb.h`, which includes all APIs for USB.
3. Initialize the USB device using the `usb_init` API.
4. Scan the USB storage device on the USB bus using the `usb_stor_scan`API.
5. Get the device number using the `blk_get_devnum_by_type` API.
6. Read data from the USB disk using the `blk_dread` API.
7. Write data to the USB disk using the `blk_dwrite` API.

The table below shows the APIs used in the file `test_usb.c` example:

Table 10. USB APIs and their description

API name (type)	Description

Table continues on the next page...

Table 10. USB APIs and their description (continued)

int usb_init(void)	Initializes the USB controller.
int usb_stop(void)	Stops the USB controller.
int usb_stor_scan(int mode)	Scans the USB and reports device information to the user if mode = 1 • Mode – if mode = 1, the information is returned to user. Returns the current device or -1.
struct blk_desc *blk_get_devnum_by_type(enum if_type if_type, int devnum)	Get a block device by type and number. • If_type – Block device type • devnum - device number Returns point to block device descriptor, or NULL if not found.
unsigned long blk_dread(struct blk_desc *block_dev, lbaint_t start, lbaint_t blkcnt, void *buffer);	Reads data from USB device • block_dev – block device descripter • start – start block • blkcnt – block number • buffer – buffer to store the data Returns the block number from which, data is read.
unsigned long blk_dwrite(struct blk_desc *block_dev, lbaint_t start, lbaint_t blkcnt, const void *buffer);	Writes data to USB device • block_dev – block device descripter • start – start block • blkcnt – block number • buffer – buffer to store the data Returns the block number to which data is written.

4.2.10 PCIe file

The file `app/test_pcie.c` provides a sample code to test PCIe and network card (such as e1000) features. The steps below show how to write a PCIe and net application:

1. Insert a PCIe network card (such as e1000) into PCIe2, or PCIe3 slot (if it exists).
2. Configure the IP address of the host machine to 192.168.1.2.
3. Include the files `include/pci.h` and `include/ netdev.h`.
4. Initialize the PCIe controller using the `pci_init` API.
5. Get uclass device by its name using the `uclass_get_device_by_seq` API.
6. Initialize the PCIe network device using the `pci_eth_init` API.
7. Begin pinging the host machine using the `net_loop` API.

The table below shows the APIs used in the file `test_pcie.c` example.

Table 11. PCIe APIs and their description

API name (type)	Description

Table continues on the next page...

Table 11. PCIe APIs and their description (continued)

void pci_init(void)	Initializes the PCIe controller. Does not return a value.
int uclass_get_device_by_seq(enum uclass_id id, int seq, struct udevice **devp)	Gets the uclass device based on an ID and sequence: • id – uclass ID • seq – sequence • devp – Pointer to device Returns 0 if Ok, negative on error.
static inline int pci_eth_init(bd_t *bis)	Initializes network card on the PCIe bus • Bis – struct containing variables accessed by shared code Returns the number of network cards.
int net_loop (enum proto_t protocol)	Main network processing loop. • enum proto_t protocol - protocol type Returns 0 if Ok, negative value on error.

4.2.11 CAN file

The file `app/test_flexcan.c` provides a sample test case to test flexcan and CANopen features. The following steps show the design process:

1. Register the receive interruption function for flexcan.
2. Register an overflow interruption function for flextimer.
3. Initialize a list of callback functions.
4. Set the node ID of this node.

The table below shows the APIs used in the file `test_flexcan.c` example.

Table 12. CAN APIs and their description

API name (type)	Description
void test_flexcan(void)	It is the test code entry for flexcan.
void flexcan_rx_irq(struct can_module *canx)	Flexcan receive interruption function. • canx – flexcan interface
void flexcan_receive(struct can_module *canx, struct can_frame *cf)	Flexcan receives CAN data frame. • canx – flexcan interface • cf – CAN message
UNS8 setState(CO_Data* d, e_nodeState newState)	Sets node state • d – object dictionary • newState – The state that needs to be set Returns 0 if Ok, > 0 on error
void canDispatch(CO_Data* d, Message *m)	CANopen handles data frames that CAN receive:

Table continues on the next page...

Table 12. CAN APIs and their description (continued)

	• d – object dictionary • m – Received CAN message
int flexcan_send(struct can_module *canx, struct can_frame *cf)	flexcan interface sends CAN message • canx – flexcan interface • cf – CAN message
void flextimer_overflow_irq(void)	Flextimer overflow interruption handler function
void timerForCan(void)	CANopen virtual clock. Call this function per 100us.

- The following log shows the CANopen slave node state:

```
=> flexcan error: 0x42242!
Note: slave node entry into the stop mode!
Note: slave node initialization is complete!
Note: slave node entry into the preOperation mode!
Note: slave node entry into the operation mode!
Note: slave node initialization is complete!
Note: slave node entry into the preOperation mode!
Note: slave node entry into the operation mode!
```

4.2.12 ENETC file

The file `app/test_net.c` provides an example to test ENETC Ethernet feature and shows how to write a net application for using this feature. This is a special case of using Net APIs.

test_net for ENETC is only an example for the LS1028ARDB board (CONFIG_ENETC_COREID_SET enabled).

1. Connect ENETC port of LS1028ARDB board to one host machine using Ethernet cable.
2. Configure the IP address of the host machine as `192.168.1.2`.
3. Power up the LS1028ARDB board. If the network is connected, the mesage `host 192.168.1.2 is alive` is displayed on the console.
4. The IP address of the board and host machine are defined in the file `test_net.c`. In this file, modify the IP address of LS1028ARDB board using variable `ipaddr` and change the IP address of host machine using variable `ping_ip`.

The table below lists the Net APIs for ENETC and their description, refer to section 4.2.7 for other Net APIs.

Table 13. ENETC APIs and their description

API name (type)	Description
void pci_init(void)	Initializes the PCIe controller. Does not return a value.
void eth_initialize(void)	Initializes the ethernet.

4.2.13 SAI file

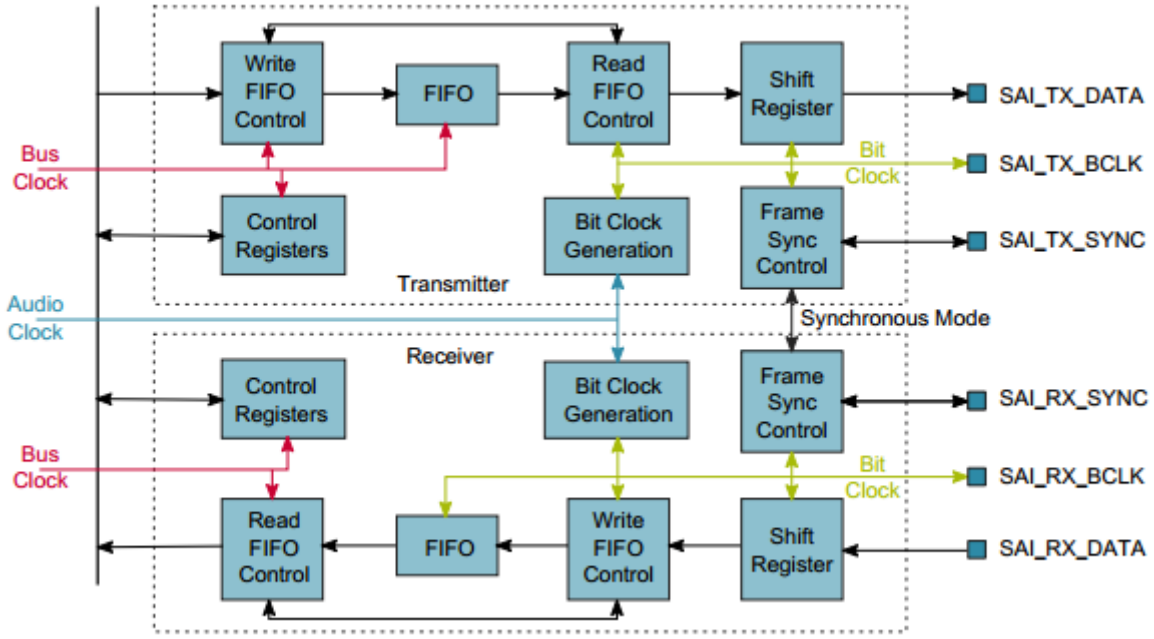
The audio feature needs SAI module and codec drivers. The following sections provide an introduction to SAI module, audio codec (SGTL5000), details of how to integrate audio with Baremetal and running an audio application on baremetal.

4.2.13.1 Synchronous Audio Interface (SAI)

The LS1028A integrates six SAI modules, but only SAI4 is used by LS1028ARDB board. The synchronous audio interface (SAI) supports fullduplex serial interfaces with frame synchronization. The bit clock and frame sync of SAI are both generated externally (SGTL5000).

- Transmitter with independent bit clock and frame sync supporting 1 data line
- Receiver with independent bit clock and frame sync supporting 1 data line
- Maximum Frame Size of 32 words
- Word size of between 8-bits and 32-bits
- Word size configured separately for first word and remaining words in frame
- Asynchronous 32 × 32-bit FIFO for each transmit and receive channel
- Supports graceful restart after FIFO error

Figure SAI block diagram

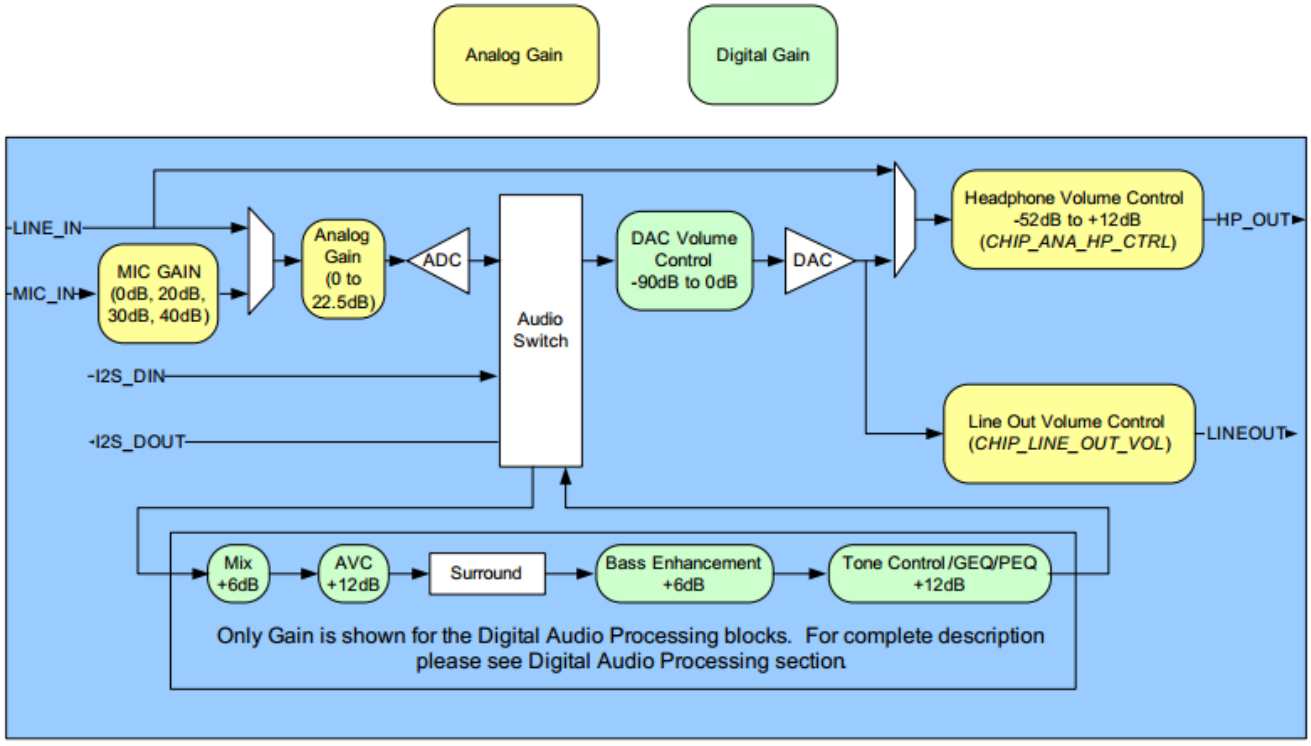


4.2.13.2 Audio codec (SGTL5000)

The SGTL5000 is a Low Power Stereo Codec with Headphone Amp from Freescale, and is designed to provide a complete audio solution for products needing LINEIN, MIC_IN, LINEOUT, headphone-out, and digital I/O. It allows input of an 8.0 MHz to 27 MHz system clock and supports 8.0 kHz, 11.025 kHz, 12 kHz, 16 kHz, 22.05 kHz, 24 kHz, 32 kHz, 44.1kHz, 48 kHz, 96 kHz sampling frequencies.The LS1028ARDB board provides a 25MHz crystal oscillator to the SGTL5000.

The SGTL5000 provides two interfaces (I2C and SPI) to setup registers. The LS1028ARDB board uses I2C interface.

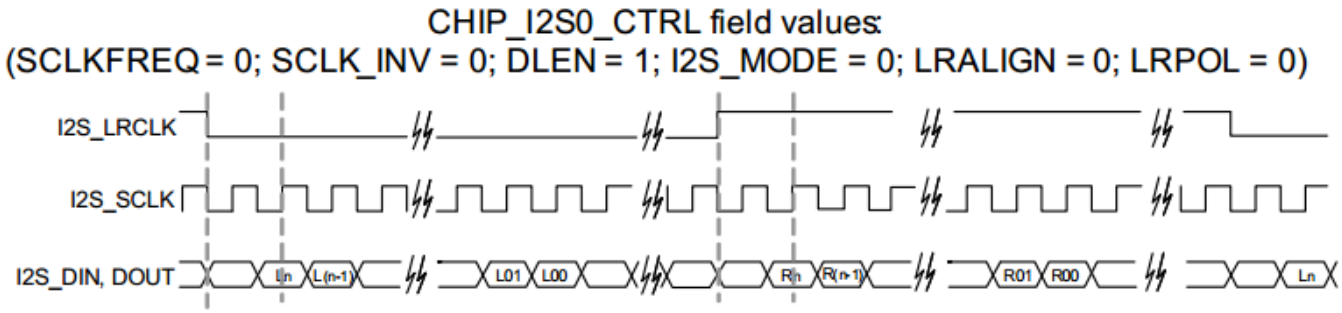
System Block Diagram, Signal Flow and Gain



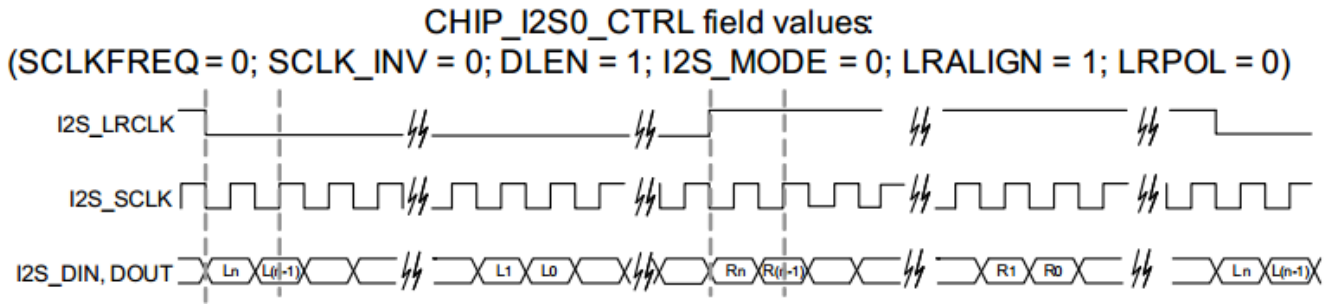
4.2.13.3 Digital interface formats

The SGTL5000 provides five common digital interface formats. The SAI and SGTL5000 digital interface formats must be the same.

- I2S Format (n = bit length)

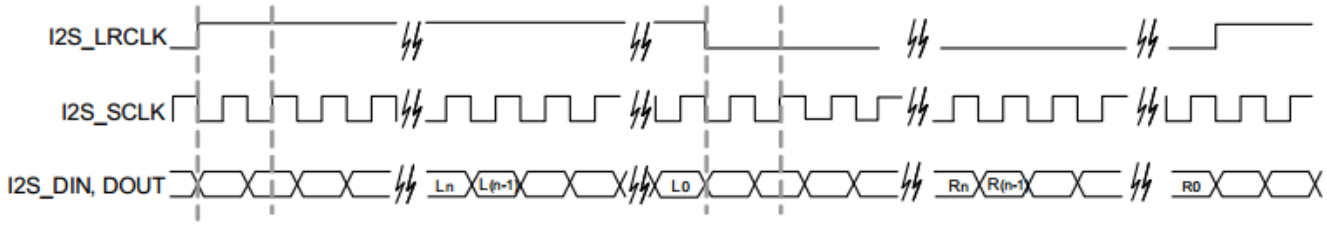


- Left Justified Format (n = bit length)



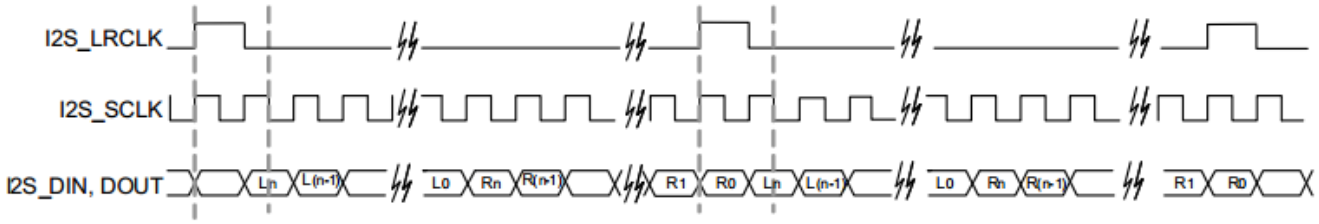
- Right Justified Format (n = bit length)

CHIP_I2S0_CTRL field values:
SCLKFREQ = 0; SCLK_INV = 0; DLEN = 1; I2S_MODE = 1; LRALIGN = 1; LRPOL = 0)



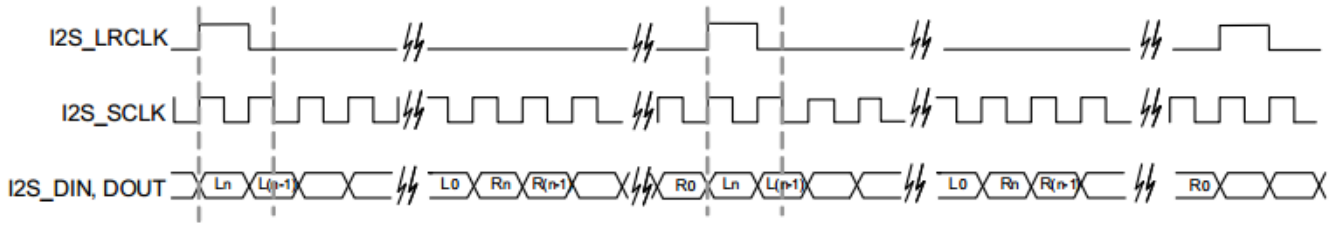
- PCM Format A

CHIP_I2S0_CTRL = 0x01F4
(SCLKFREQ = 1; MS = 1; SCLK_INV = 1; DLEN = 3; I2S_MODE = 2; LRALIGN = 0)



- PCM Format B

CHIP_I2S0_CTRL = 0x01F6
(SCLKFREQ = 1; MS = 1; SCLK_INV = 1; DLEN = 3; I2S_MODE = 2; LRALIGN = 1)



4.2.13.4 running application

Play a demo audio file:

```
=> wavplayer
************************************************************
audioformat: PCM
nchannels: 1
samplerate: 16000
bitrate: 256000
blockalign: 2
bps: 16
datasize: 67968
datastart: 44
************************************************************
sgtl5000 revision 0x11
fsl_sai_ofdata_to_platdata
Probed sound 'sound' with codec 'codec@a' and i2s 'sai@f130000'
```

```
i2s_transfer_tx_data
The music waits for the end!
The music is finished!
*******************************************************
```

4.3 ICC module

Inter-core communication (ICC) module works on Linux core (master) and Baremetal core (slave), provides the data transfer between cores via SGI inter-core interrupt and share memory blocks. It can support multi-core silicon platform and transfer the data concurrently and efficiently.

ICC module is structured based on two basics:

- SGI: Software-generated Interrupts in ARM GIC, used to generate inter-core interrupts. The ICC module uses the number 8 SGI interrupt for all Linux and Baremetal cores.
- Shared memory: A memory space shared by all platform cores. The base address and size of the share memory should be defined in header files before compilation.

ICC modules can work concurrently, lock-free among multi-core platform, and support broadcast case with Buffer Descriptor Ring mechanism.

The figure below shows the basic operating principle for data transfer from Core 0 to Core 1. After the data writing and head point moving to next, Core 0 triggers a SGI (8) to Core 1, then Core 1 gets the BD ring updated status and reads the new data, then moves the tail point to next.

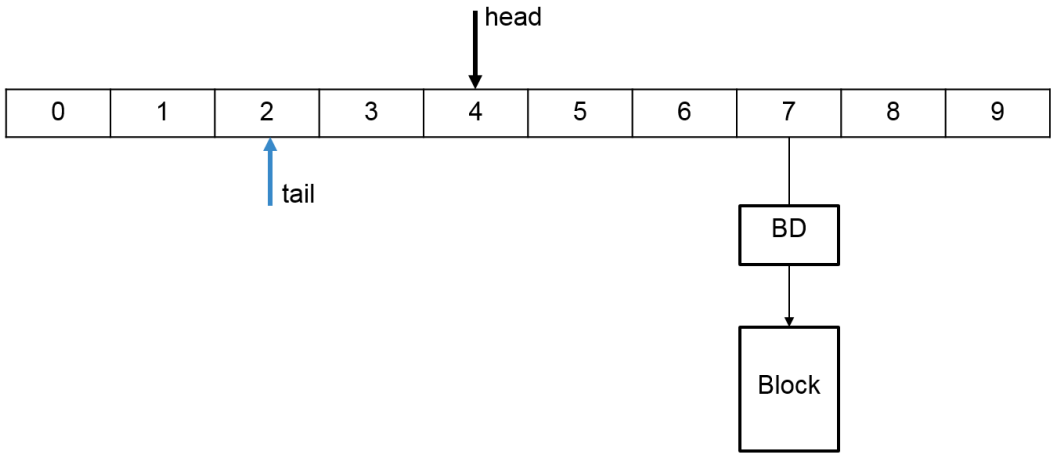


Figure 7. BD rings for inter-core communication

For a multi-core platform (that is, four cores), the total BD rings are arranged as shown in the following figure. (See the BD rings on Core 0 and Core 1.)

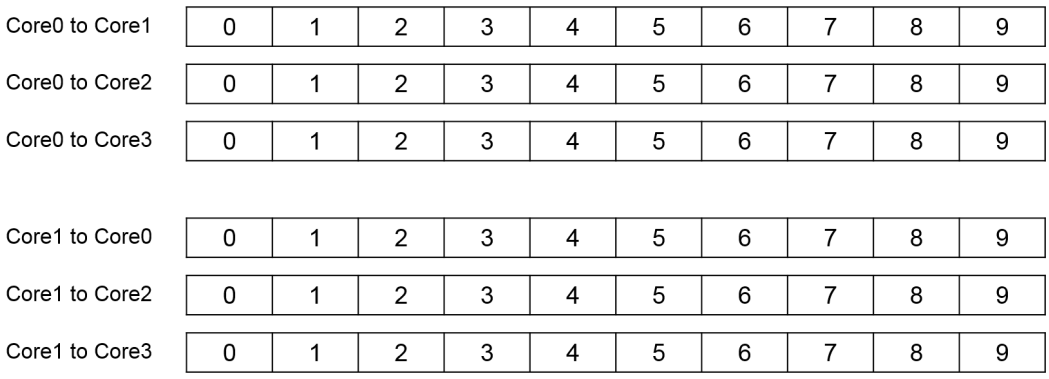


Figure 8. BD rings for multi-core platform

All the ICC ring structures, BD structures and blocks for data are in the shared memory. A four-core platform ICC module would map the shared memory as shown in the figure below.

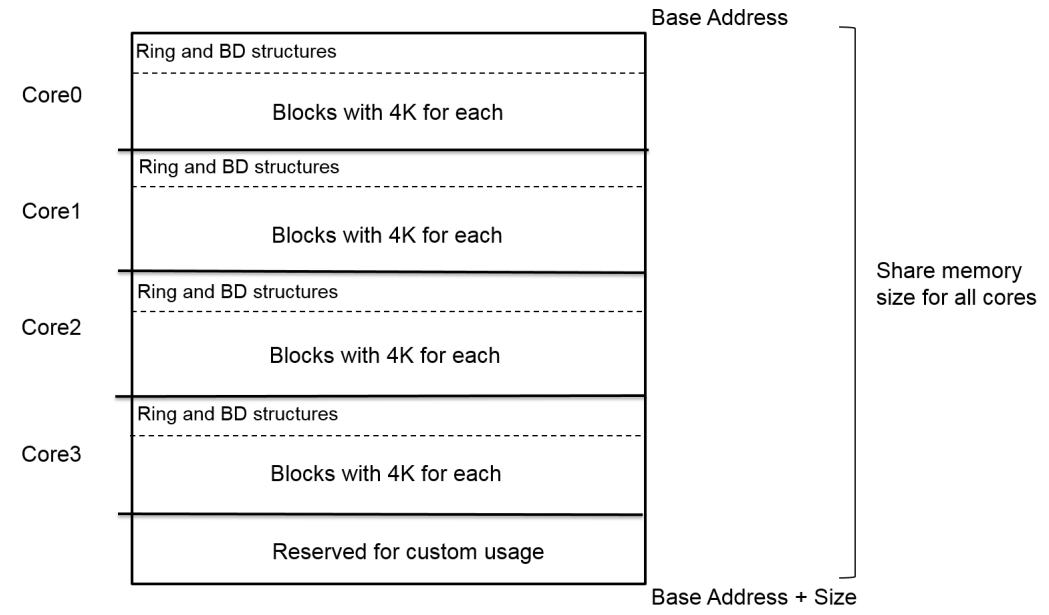


Figure 9. ICC shared memory map for the four-core platform

Generally, Core 0 runs Linux as master core, other cores run Baremetal as slaves. They obtain the same size of share memory to structure the rings and BDs, and split the blocks space with 4k unit for each block. The reserved space at the top of the share memory is out of the ICC module and for the custom usage.

For LS1021A platform with two cores, the share memory map is defined as:

- The total share memory size is 256 MB.
- The reserved space for custom usage is 16 MB at the top of the share memory space.
- Core 0 runs Linux as master core, the share memory size for ICC is 120 MB, in which the ring and BD structure space is 2 M, and the block space for data is 118 MB with 4K for each block.
- Core 1 runs Baremetal as slave core, the share memory size for ICC is 120 MB, in which the ring and BD structure space is 2M, and the block space for data is 118 MB with 4K for each block.

The ICC module includes two parts of the code:

- ICC code for Linux user space, works for data transfer between master core and slave cores. The code is integrated into the OpenIL and named `icc package`. After the compilation, the icc binary is put into the Linux filesystem.
- ICC code for Baremetal, runs on every slave core, works for data transfer between baremetal cores and master core.

The ICC code for Linux user space in OpenIL directory:

```
package/icc/src/
├── icc-main.c --- the example case commands
├── inter-core-comm.c
├── inter-core-comm.h --- include the header file to use ICC module
└── Makefile
```

The ICC code for Baremetal in Baremetal directory:

```
baremetal/
├── arch/arm/lib/inter-core-comm.c
├── arch/arm/include/asm/inter-core-comm.h --- include the header file to use ICC module
└── cmd/icc.c --- the example case commands
```

The APIs ICC exported out for usage in both Linux user space and Baremetal code.

Table 14. ICC APIs

APIs	Description
unsigned long icc_ring_state(int coreid)	Checks the ring and block state. Returns: • 0 - if empty • !0 - the working block address currently.
Unsigned long icc_block_request(void)	Requests a block, which is ICC_BLOCK_UNIT_SIZE size. Returns: • 0 - failed • !0 - block address can be used.
void icc_block_free(unsigned long block)	Frees a block requested. Be careful if the destination cores are working on this block.
int icc_irq_register(int src_coreid, void (*irq_handle)(int, unsigned long, unsigned int))	Registers ICC callback handler for received data. Returns: • 0 - on success • -1 - if failed.
int icc_set_block(int core_mask, unsigned int byte_count, unsigned long block)	Sends the data in the block to a core or multi-core. This triggers the SGI interrupt. Returns:

Table continues on the next page...

Table 14. ICC APIs (continued)

APIs	Description
	• 0 - on success • -1 - if failed.
void icc_show(void)	Shows the ICC basic information.
int icc_init(void)	Initializes the ICC module.

4.3.1 ICC examples

This section provides example commands for use cases in both Linux user space and Baremetal code. They can be used to check and verify the ICC module conveniently.

1. In Linux user space, use the command `icc` to display the supported cases.

```
[root@LS1046ARDB ~] # icc
icc show - Shows all icc rings status at this core
icc perf <core_mask> <counts> - ICC performance to cores <core_mask> with <counts> bytes
icc send <core_mask> <data> <counts> - Sends <counts> <data> to cores <core_mask>
icc irq <core_mask> <irq> - Sends SGI <irq> ID[0 - 15] to <core_mask>
icc read <addr> <counts> - Reads <counts> 32bit register from <addr>
icc write <addr> <data> - Writes <data> to a register <addr>
```

   Likewise, in Baremetal system, use the command `icc` to view the supported cases.

```
=> icc
1:icc - Inter-core communication via SGI interrupt

1:Usage:
icc show                                - Show all icc rings status at this core
icc perf <core_mask> <counts>           - ICC performance to cores <core_mask> with
<counts> bytes
icc send <core_mask> <data> <counts>    - Send <counts> <data> to cores <core_mask>
icc irq <core_mask> <irq>               - Send SGI <irq> ID[0 - 15] to <core_mask>
```

2. The ICC module command examples on LS1046ARDB with Linux (Core 0) + Baremetal (Core 1, 2, 3) system:

   Run `icc send 0x2 0x55 128` to send 128 bytes data 0x55 to core 1.

```
[root@LS1046ARDB ~] # icc send 0x2 0x55 128
gic_base: 0xffffa033f000, share_base: 0xffff9133f000, share_phy: 0xd0000000,
block_phy: 0xd0200000

ICC send testing ...
Target cores: 0x2, bytes: 128
ICC send: 128 bytes to 0x2 cores success

all cores: reserved_share_memory_base: 0xdf000000; size: 16777216

mycoreid: 0; ICC_SGI: 8; share_memory_size: 62914560
block_unit_size: 4096; block number: 14848; block_idx: 0

#ring 0 base: 0xffff9133f000; dest_core: 0; SGI: 8
desc_num: 128; desc_base: 0xd00000c0; head: 0; tail: 0
busy_counts: 0; interrupt_counts: 0
```

```
#ring 1 base: 0xffff9133f030; dest_core: 1; SGI: 8
desc_num: 128; desc_base: 0xd00008c0; head: 1; tail: 1
busy_counts: 0; interrupt_counts: 1

#ring 2 base: 0xffff9133f060; dest_core: 2; SGI: 8
desc_num: 128; desc_base: 0xd00010c0; head: 0; tail: 0
busy_counts: 0; interrupt_counts: 0

#ring 3 base: 0xffff9133f090; dest_core: 3; SGI: 8
desc_num: 128; desc_base: 0xd00018c0; head: 0; tail: 0
busy_counts: 0; interrupt_counts: 0
```

At the same time, Core 1 prints the receival information.

```
=> 1:Get the ICC from core 0; block: 0xd0200000, bytes: 128, value: 0x55
```

3. ICC command run on baremetal side

```
=> icc send 0x1 0xaa 128
1:ICC send testing ...
1:Target cores: 0x1, bytes: 128
1:ICC send: 128 bytes to 0x1 cores success

1:all cores: reserved_share_memory_base: 0xdf000000; size: 16777216

1:mycoreid: 1; ICC_SGI: 8; share_memory_size: 62914560
1:block_unit_size: 4096; block number: 14848; block_idx: 0

1:#ring 0 base: 00000000d3c00000; dest_core: 0; SGI: 8
1:desc_num: 128; desc_base: 00000000d3c000c0; head: 1; tail: 1
1:busy_counts: 0; interrupt_counts: 1

1:#ring 1 base: 00000000d3c00030; dest_core: 1; SGI: 8
1:desc_num: 128; desc_base: 00000000d3c008c0; head: 0; tail: 0
1:busy_counts: 0; interrupt_counts: 0

1:#ring 2 base: 00000000d3c00060; dest_core: 2; SGI: 8
1:desc_num: 128; desc_base: 00000000d3c010c0; head: 0; tail: 0
1:busy_counts: 0; interrupt_counts: 0

1:#ring 3 base: 00000000d3c00090; dest_core: 3; SGI: 8
1:desc_num: 128; desc_base: 00000000d3c018c0; head: 0; tail: 0
1:busy_counts: 0; interrupt_counts: 0
```

Then, Core 0 side (Linux) will receive these data:

```
[root@LS1046ARDB ~] # [ 4247.733753] 000: Get the ICC from core 1; block: 0xd3e00000, bytes:
128, value: 0xaa
```

4.4 Hardware resource allocation

This section describes how to modify the hardware resource allocation depending on the application and used reference design board.

4.4.1 LS1021A-IoT board

4.4.1.1 Linux DTS

Remove cpu1 node on DTS, and remove all the devices that bare metal has used.

4.4.1.2 Memory configuration

LS1021A-IoT board has a 1 GB size DDR. The DDR memory can be configured into three partitions: 512M for core0 (Linux), 256M for core1 (bare metal), and 256M for shared memory.

The configuration is in the path: `include/configs/ls1021aiot_config.h`.

```
#define CONFIG_SYS_DDR_SDRAM_SLAVE_SIZE (256 * 1024 * 1024)
#define CONFIG_SYS_DDR_SDRAM_MASTER_SIZE (512 * 1024 * 1024)
```

NOTE

Memory configuration must be consistent with the U-Boot configuration of core0.

Modify "`CONFIG_SYS_MALLOC_LEN`" in *include/configs/ls1021aiot_config.h* to change the maximum size of malloc.

You can use functions included in `malloc.h` to allocate or free memory in your program. These functions are listed in the table below.

Table 15. Description of memory APIs

API name (type)	Description
void_t* malloc (size_t n)	Allocates memory • `n` – length of allocated chunk • Returns a pointer to the newly allocated chunk
void free (void *ptr)	Releases the chunk of memory pointed to by ptr (where `ptr` is a pointer to the chunk of memory)

The memory configuration for bare metal is shown in the figure below.

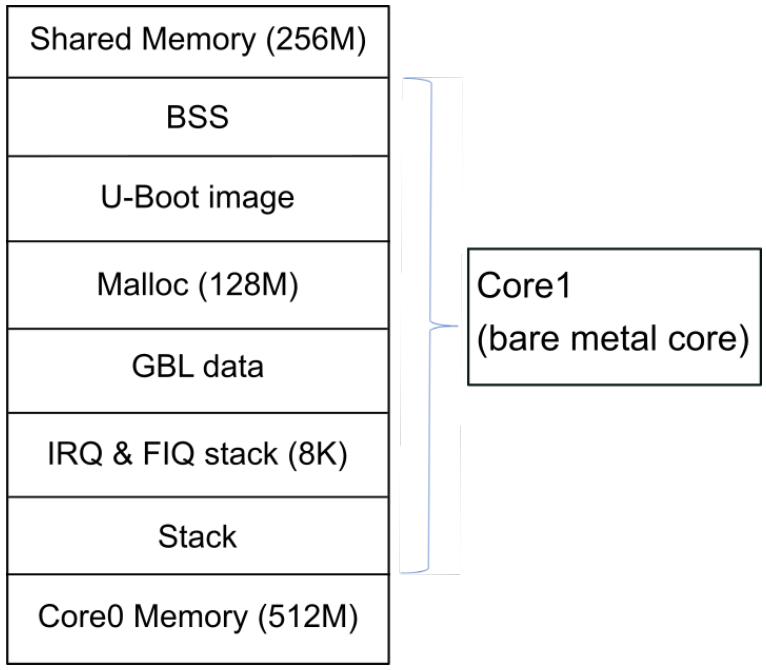


Figure 10. Memory configuration

4.4.1.3 GPIO

LS1021A has four GPIO controllers. The configuration is defined in the file:*arch/arm/dts/ls1021a-iot.dtsi*. You can add a GPIO node in the file ls1021a-iot.dtsi to assign a GPIO resource to different cores. Following is a sample code for adding a GPIO node.

```
&gpio2
{
 status = "okay";
};
```

4.4.1.4 I2C

LS1021A has three I2C controllers. Configure the I2C bus on `ls1021aiot_config.h` using the commands below:

```
// include/configs/ls1021aiot_config.h:
#define CONFIG_SYS_I2C_MXC_I2C1 /* enable I2C bus 1 */
#define CONFIG_SYS_I2C_MXC_I2C2 /* enable I2C bus 2 */
#define CONFIG_SYS_I2C_MXC_I2C3 /* enable I2C bus 3 */
```

4.4.1.5 Hardware interrupts

LS1021A has six IRQs as external IO signals connected to interrupt the controller. We can use these six IRQs on bare metal cores. The ids for these signals, IRQ0-IRQ5 are: 195, 196, 197, 199, 200, and 201.

GIC interrupt APIs are defined in the file, `asm/interrupt-gic.h`. The following example shows how to register a hardware interrupt:

```
//register HW interrupt
void gic_irq_register(int irq_num, void (*irq_handle)(int));
void gic_set_target(u32 core_mask, unsigned long hw_irq);
void gic_set_type(unsigned long hw_irq);
```

4.4.1.6 IFC

LS1021A-IoT board has no IFC device, but the LS1021A SoC has an IFC interface. Since IFC is multiplexed with QSPI, you need to modify the RCW to use the IFC interface, and add a configuration such as shown in the sample code provided below. Users can modify the code to support IFC as per the actual scenario.

```
#define CONFIG_FSL_IFC
#define CONFIG_SYS_CPLD_BASE 0x7fb00000
#define CPLD_BASE_PHYS CONFIG_SYS_CPLD_BASE
#define CONFIG_SYS_FPGA_CSPR_EXT (0x0)

#define CONFIG_SYS_FPGA_CSPR (CSPR_PHYS_ADDR(CPLD_BASE_PHYS) | \
CSPR_PORT_SIZE_8 | \
CSPR_MSEL_GPCM | \
CSPR_V)

#define CONFIG_SYS_FPGA_AMASK IFC_AMASK(64 * 1024)
#define CONFIG_SYS_FPGA_CSOR (CSOR_NOR_ADM_SHIFT(4) | \
CSOR_NOR_NOR_MODE_AVD_NOR | \
CSOR_NOR_TRHZ_80)

/* CPLD Timing parameters for IFC GPCM */
#define CONFIG_SYS_FPGA_FTIM0 (FTIM0_GPCM_TACSE(0xf) | \
FTIM0_GPCM_TEADC(0xf) | \
FTIM0_GPCM_TEAHC(0xf))

#define CONFIG_SYS_FPGA_FTIM1 (FTIM1_GPCM_TACO(0xff) | \
FTIM1_GPCM_TRAD(0x3f))

#define CONFIG_SYS_FPGA_FTIM2 (FTIM2_GPCM_TCS(0xf) | \
FTIM2_GPCM_TCH(0xf) | \
FTIM2_GPCM_TWP(0xff))

#define CONFIG_SYS_FPGA_FTIM3 0x0
#define CONFIG_SYS_CSPR1_EXT CONFIG_SYS_FPGA_CSPR_EXT
#define CONFIG_SYS_CSPR1 CONFIG_SYS_FPGA_CSPR
#define CONFIG_SYS_AMASK1 CONFIG_SYS_FPGA_AMASK
#define CONFIG_SYS_CSOR1 CONFIG_SYS_FPGA_CSOR
#define CONFIG_SYS_CS1_FTIM0 CONFIG_SYS_FPGA_FTIM0
#define CONFIG_SYS_CS1_FTIM1 CONFIG_SYS_FPGA_FTIM1
#define CONFIG_SYS_CS1_FTIM2 CONFIG_SYS_FPGA_FTIM2
#define CONFIG_SYS_CS1_FTIM3 CONFIG_SYS_FPGA_FTIM3
```

4.4.1.7 USB

LS1021AIOT has a single DW3 USB controller, which is assigned to the second core, by default. Use the command `make menuconfig` to re-configure the U-Boot to assign it to other cores.

```
ARM architecture  --->
[*] Enable baremetal
[*] Enable USB for baremetal
(1) USB0 is assigned to core1
(1) USB Controller numbers
```

4.4.1.8 PCIe

LS1021AIOT has two PCIe controllers. By deafult, one is assigned to core0 and the other is assigned to core1. Use the `make menuconfig` command to re-configure the U-Boot, in order to re-assign the cores.

```
ARM architecture  --->
[*] Enable baremetal
[*] Enable PCIE for baremetal
(0)   PCIe1 is assigned to core0
(1)   PCIe2 is assigned to core1
(2)   PCIe Controller numbers
```

4.4.1.9 FlexCAN

Assigning CAN3 to Baremetal

In baremetal, the port is allocated through the `flexcan.c` file. The `flexcan.c` path is `industry-uboot/drivers/flexcan/flexcan.c`. In this file, you need to define the following variables:

```
struct can_bittiming_t flexcan3_bittiming = CAN_BITTIM_INIT(CAN_500K);
```

Note: You also need to set the bit timing and baud rate (500K) of the CAN port.

```
struct can_ctrlmode_t flexcan3_ctrlmode = {
.loopmode = 0, /* Indicates whether the loop mode is enabled */
.listenonly = 0, /* Indicates whether the only-listen mode is enabled */
.samples = 0,
.err_report = 1,
};
struct can_init_t flexcan3 = {
.canx = CAN3,   /* Specify CAN port  */
.bt = &flexcan3_bittiming,
.ctrlmode = &flexcan3_ctrlmode,
.reg_ctrl_default = 0,
.reg_esr = 0
};
```

Optional Parameters

- **CAN port**

```
#define CAN3      (struct can_module *)CAN3_BASE)
#define CAN4      (struct can_module *)CAN4_BASE)
```

- **Baud rate**

```
#define CAN_1000K 10
#define CAN_500K 20
#define CAN_250K 40
#define CAN_200K 50
#define CAN_125K 80
#define CAN_100K 100
#define CAN_50K 200
#define CAN_20K 500
#define CAN_10K 1000
#define CAN_5K 2000
```

Use the command `make menuconfig` for configuring the FlexCAN setting for LS1021A reference design boards, as shown below: .

```
Device Drivers   --->
CAN support   --->
[*] Support for Freescale FLEXCAN based chips
[*]   Support for canfestival
```

4.4.2 LS1028ARDB board

This section describes the ENETC configuration setting for LS1028A reference design boards.

4.4.2.1 ENETC

LS1028ARDB has only one ENETC controller in use, which is assigned to core1 as the default setting. The controller can be reconfigured by using the command, `make menuconfig`.

See the following:

```
ARM architecture --->
[*] Enable baremetal
[*] Enable ENETC for baremetal
   (1)  Enetc1 is assigned to core1
   (1)  ENETC Controller numbers
```

4.4.2.2 I2C

This section describes how to configure the I2C bus on LS1028A reference design boards.

LS1028ARDB has eight I2C controllers, but only controller 0 is used for I2C devices (for example RTC, Thermal Monitor), and Linux (core 0) will use this controller for some features (for example RTC). Therefore, below codes is just used to demo how to enable I2C in baremetal side.

Note: Operating the I2C devices in baremetal side CAREFULLY.

```
#define CONFIG_SYS_I2C_MXC_I2C1 /* enable I2C bus 0 */
#define CONFIG_SYS_I2C_MXC_I2C2 /* enable I2C bus 1 */
#define CONFIG_SYS_I2C_MXC_I2C3 /* enable I2C bus 2 */
#define CONFIG_SYS_I2C_MXC_I2C4 /* enable I2C bus 3 */

#define CONFIG_I2C_BUS_CORE_ID_SET
#define CONFIG_SYS_I2C_MXC_I2C0_COREID  1
```

The `CONFIG_SYS_I2C_MXC_I2C0_COREID` defines the slave core that runs the I2C bus.

Because I2C is enabled with DM mode in baremetal side, there is not automatic codes to test it. Follow below steps to read RTC (0x51 address, is on bus 2) in baremetal side :

```
=> i2c bus
Bus 0:  i2c@2000000  (active 0)
   77: i2c-mux@77, offset len 1, flags 0
   57: generic_57, offset len 1, flags 0
Bus 1:  i2c@2000000->i2c-mux@77->i2c@1
Bus 2:  i2c@2000000->i2c-mux@77->i2c@3
   51: rtc@51, offset len 1, flags 0
Bus 3:  i2c@2010000
Bus 4:  i2c@2020000
Bus 5:  i2c@2030000
Bus 6:  i2c@2040000
Bus 7:  i2c@2050000
Bus 8:  i2c@2060000
```

```
Bus 9:  i2c@2070000
=> i2c md 0x51 0
Error reading the chip: -121
=> i2c dev 2
Setting bus to 2
=> i2c md 0x51 0
0000: 04 00 36 03 12 15 02 12 20 80 80 80 80 80 00 c2    ..6..... .......
```

4.4.2.3 SAI

LS1028ARDB has only one SAI module in use, which is assigned to core1 as the default setting. The SAI module can be reconfigured by using the command, `make menuconfig`.

See the following:

```
Command line interface --->
   Misc commands --->
      [*] wavplayer

Device Drivers --->
   Sound support --->
      [*] Enable sound support
      [*]   Enable I2S support
      [*] Freescale sound
      [*] Freescale sgtl5000 audio condec
      [*] Freescale SAI module
```

4.4.2.3.1 Audio integratation in Baremetal

For audio feature, we need to add SAI and SGTL5000 drivers to Baremetal.

- Add SAI driver source code to the *drivers/sound* directory
- Add SGTL5000 driver source code to the *drivers/sound* directory
- Add sound device source code to the *drivers/sound* directory
- Add a command that can play wav files to the *cmd* directory
- Add support for SAI and sgtl5000 in LS1028ARDB dts file

  In fsl-ls1028a.dtsi file:

  ```
  sai4: sai@f130000 {
      #sound-dai-cells = <0>;
      compatible = "fsl,ls1028a-sai";
      reg = <0x0 0xf130000 0x0 0x10000>;
      status = "disabled";
  };
  ```

  In fsl-ls1028a-rdb.dts file:

  ```
  sound {
      compatible = "fsl,audio-sgtl5000";
      model = "ls1028a-sgtl5000";
      audio-cpu = <&sai4>;
      audio-codec = <&sgtl5000>;
      audio-routing =
        "LINE_IN", "Line In Jack",
        "MIC_IN", "Mic Jack",
        "Mic Jack", "Mic Bias",
        "Headphone Jack", "HP_OUT";
  };
  ```

```
i2c@1 {
    #address-cells = <1>;
    #size-cells = <0>;
    reg = <0x1>;
    sgtl5000: codec@a {
        #sound-dai-cells = <0>;
        compatible = "fsl,sgtl5000";
        reg = <0xa>;
        VDDA-supply = <1800>;
        VDDIO-supply = <1800>;
        sys_mclk = <25000000>;
        sclk-strength = <3>;
    };
};
&sai4 {
    status = "okay";
};
```

- Add all source code to the corresponding makefile file
- Add new default configurations to ls1028ardb_sdcard_baremetal_defconfig file

4.4.3 LS1043ARDB or LS1046ARDB board

The following sections describe the hardware resource allocation for the LS1043ARDB or LS1046ARDB boards for implementing the supported features.

4.4.3.1 Linux DTS

Remove cpu1, cpu2, cpu3 nodes on DTS, and remove all the devices that bare metal has used.

4.4.3.2 Memory configuration

This section describes the memory configuration for LS1043ARDB or LS1046ARDB boards.

The LS1043ARDB or LS1046ARDB boards have a 2GB size DDR. To use the baremetal framework, configure DDR into three partitions:

- 512M for core0 (Linux)
- 256M for core1(bare metal)
- 256M for core2(bare metal)
- 256M for core3(bare metal), and 256M for shared memory.

The configuration can be defined in the file `include/configs/ls1043ardb.h`.

```
 #define CONFIG_SYS_DDR_SDRAM_SLAVE_SIZE (256 * 1024 * 1024)
 #define CONFIG_SYS_DDR_SDRAM_MASTER_SIZE (512 * 1024 * 1024)
 #define CONFIG_SYS_DDR_SDRAM_SHARE_RESERVE_SIZE (16 * 1024 * 1024)
 #define CONFIG_SYS_DDR_SDRAM_SHARE_SIZE \ ((256 * 1024 * 1024)
- CONFIG_SYS_DDR_SDRAM_SHARE_RESERVE_SIZE)
```

NOTE

The memory configuration must be consistent with the U-Boot configuration of core0.

The memory configuration for bare metal is shown in the figure below.

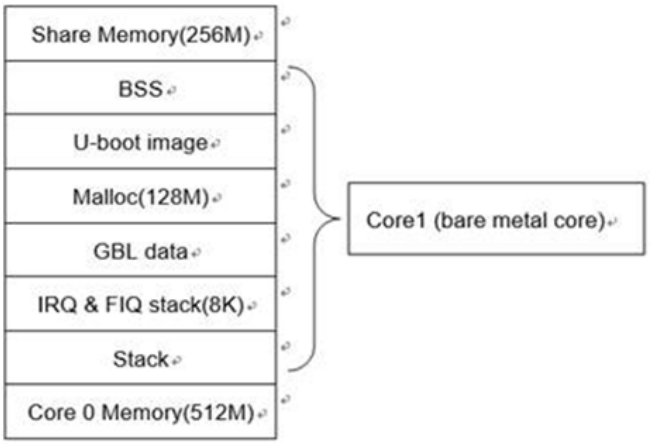


Figure 11. Memory configuration for LS1043ARDB or LS1046ARDB

The functions included in `malloc.h` in the table below can be used to allocate or free memory in program. Modify `CONFIG_SYS_MALLOC_LEN` in `include/configs/ls1043a_common.h` to change the maximum size of malloc.

Table 16. Memory APIs description

API name (type)	Description
void_t* malloc (size_t n)	Allocates memory • “n” – length of allocated chunk • Returns a pointer to the newly allocated chunk
void free (void *ptr)	Releases the chunk of memory pointed to by ptr (where “ptr” is a pointer to the chunk of memory)

The GPIO for LS1043ARDB (or LS1046ARDB) has four GPIO controllers. You need to add a GPIO node in the file `ls1043/6a-rdb.dts` to assign a GPIO resource to different cores. The configuration can be done in the file `arch/arm/dts/fsl-ls1043/6a-rdb.dts`.

4.4.3.3 GPIO

LS1043/6A has four GPIO controllers. You can add a GPIO node in the file `ls1043/6a-rdb.dts` to assign a GPIO resource to different cores. The configuration is in `arch/arm/dts/fsl-ls1043/6a-rdb.dts`. Use the command below to add a GPIO node:

```
&gpio2 {
     status = "okay";
 };
```

4.4.3.4 I2C

This section describes how to configure the I2C bus on LS1028A, LS1043A, or LS1046A reference design boards.

The LS1043ARDB (or LS1028ARDB / LS1046ARDB) has four I2C controllers. You can configure the I2C bus using the `ls1043ardb_config.h` (or `ls1043ardb_config.h`) file using the commands below:

```
// include/configs/ls1043ardb_config.h:
#define CONFIG_SYS_I2C_MXC_I2C1     /* enable I2C bus 0 */
#define CONFIG_SYS_I2C_MXC_I2C2     /* enable I2C bus 1 */
```

```
#define CONFIG_SYS_I2C_MXC_I2C3     /* enable I2C bus 2 */
#define CONFIG_SYS_I2C_MXC_I2C4     /* enable I2C bus 3 */
#define CONFIG_SYS_I2C_MXC_I2C0_COREID  1
#define CONFIG_SYS_I2C_MXC_I2C1_COREID  2
#define CONFIG_SYS_I2C_MXC_I2C2_COREID  3
#define CONFIG_SYS_I2C_MXC_I2C3_COREID  1
```

The `CONFIG_SYS_I2C_MXC_I2C0_COREID` defines the slave core that runs the I2C bus.

4.4.3.5 Hardware interrupts

LS1043A has twelve IRQs as external IO signals connected to interrupt the controller. These twelve IRQs can be used on bare metal cores. The ids for these signals, IRQ0-IRQ11 are: 163, 164, 165, 167, 168, 169, 177, 178, 179, 181, 182, and 183. GIC interrupt APIs are defined in `asm/interrupt-gic.h`. The following example shows how to register a hardware interrupt:

```
//register HW interrupt
void gic_irq_register(int irq_num, void (*irq_handle)(int));
void gic_set_target(u32 core_mask, unsigned long hw_irq);
void gic_set_type(unsigned long hw_irq);
```

4.4.3.6 QSPI

LS1046ARDB has a QSPI flash device. To configure the QSPI on ls1046ardb_config.h, use the command below:

```
#define CONFIG_FSL_QSPI_COREID 1
```

Here, the `CONFIG_FSL_QSPI_COREID` defines the slave core that runs this QSPI.

4.4.3.7 IFC

LS1043A and LS1046A has IFC controller. Both Nor flash and NAND flash are supported in ls1043ardb, only NAND flash is supported in ls1046ardb.

1. IFC is disabled in Linux kernel via disabling "ifc" node

   ```
   &ifc {
          status = "disabled";
   };
   ```

2. Enter the `Baremetal-Framework` directory and then execute the commands below: (IFC is enabled by default)

   ```
   make menuconfig
   ARM architecture  --->
      [*] Enable baremetal
      [*]   Enable IFC for baremetal
      (1)     IFC is assigned to that core
   ```

4.4.3.8 Ethernet

This section describes the Ethernet configuration settings for LS1043A or LS1046A reference design boards.

LS1043A or LS1046A has only one FMAN, so you need to remove the DPAA driver in Linux.

1. Disable the Linux DPAA driver using the settings below:

   ```
     $make linux-menuconfig
    Device Drivers --->
   ```

```
       Staging drivers--->
          < > Freescale Datapath Queue and Buffer management
```

2. Enter the `Baremetal-Framework` directory and then execute the commands below:

```
make menuconfig
ARM architecture  --->
   [*] Enable baremetal
   [*] Enable fman for baremetal
   (1) FMAN1 is assigned to that core
```

Configure FMAN to the specified core by modifying the `FMAN1 is assigned to that core` value, which is the default configuration, to `core1`.

4.4.3.9 USB

This section describes the USB configuration setting for LS1043A and LS1046A reference design boards.

Both LS1043A and LS1046A have three DW3 USB controllers, we assign them to core1, core2 and core3 as the default setting. Re-configure them with command ‘make menuconfig’.

```
ARM architecture  --->
[*] Enable baremetal
[*]   Enable USB for baremetal
(1)     USB0 is assigned to core1
(2)     USB1 is assigned to core2
(3)     USB2 is assigned to core3
(3)     USB Controller numbers
```

4.4.3.10 PCIE

This section describes the PCIe configuration setting for LS1043A and LS1046A reference design boards.

Both LS1043A and LS1046A have three PCIe controllers, we assign them to core0, core1 and core2 as the default setting. Re-configure them with command ‘make menuconfig’.

```
ARM architecture  --->
[*] Enable baremetal
(0)     PCIe1 is assigned to core0
(1)     PCIe2 is assigned to core1
(2)     PCIe3 is assigned to core2
(3)     PCIe Controller numbers
```

4.4.4 LX2160ARDB board

The following sections describe the hardware resource allocation for the LX2160ARDB boards for implementing the supported features.

4.4.4.1 Memory configuration

This section describes the memory configuration for LX2160ARDB boards.

The LX2160ARDB boards have a 16GB size DDR. To use the baremetal framework, configure DDR into three partitions:

- 15G for core0 (Linux)
- 64M per core from core1 to core15(bare metal), and 64M for shared memory.

The configuration can be defined in the file `include/configs/lx2160ardb_config.h`.

```
 #define CONFIG_SYS_DDR_SDRAM_SLAVE_SIZE (64 * 1024 * 1024)
 #define CONFIG_SYS_DDR_SDRAM_MASTER_SIZE (512 * 1024 * 1024)
 #define CONFIG_SYS_DDR_SDRAM_SHARE_RESERVE_SIZE (16 * 1024 * 1024)
 #define CONFIG_SYS_DDR_SDRAM_SHARE_SIZE \ ((64 * 1024 * 1024)
- CONFIG_SYS_DDR_SDRAM_SHARE_RESERVE_SIZE)
```

Figure 12. Memory configuration for LX2160ARDB

The functions included in `malloc.h` in the table below can be used to allocate or free memory in program. Modify `CONFIG_SYS_MALLOC_LEN` in `include/configs/lx2160ardb_config.h` to change the maximum size of malloc.

Table 17. Memory APIs description

API name (type)	Description
void_t* malloc (size_t n)	Allocates memory • “n” – length of allocated chunk • Returns a pointer to the newly allocated chunk
void free (void *ptr)	Releases the chunk of memory pointed to by ptr (where “ptr” is a pointer to the chunk of memory)

4.4.5 I.MX6Q-SABRESD board

The following sections describe the hardware resource allocation for the I.MX6Q-SABRESD boards for implementing the supported features.

4.4.5.1 Memory configuration

This section describes the memory configuration for I.MX6Q-SABRESD boards.

The I.MX6Q-SABRESD boards have a 1GB size DDR. To use the baremetal framework, configure DDR into three partitions:

- 512M for core0 (Linux)
- 128M for core1(bare metal)
- 128M for core2(bare metal)
- 128M for core3(bare metal), and 128M for shared memory.

The configuration can be defined in the file `include/configs/mx6sabresd_config.h`.

```
 #define CONFIG_SYS_DDR_SDRAM_SLAVE_SIZE (128 * 1024 * 1024)
 #define CONFIG_SYS_DDR_SDRAM_MASTER_SIZE (512 * 1024 * 1024)
 #define CONFIG_SYS_DDR_SDRAM_SHARE_RESERVE_SIZE (16 * 1024 * 1024)
 #define CONFIG_SYS_DDR_SDRAM_SHARE_SIZE \ ((128 * 1024 * 1024)
- CONFIG_SYS_DDR_SDRAM_SHARE_RESERVE_SIZE)
```

NOTE

The memory configuration must be consistent with the U-Boot configuration of core0.

The memory configuration for bare metal is shown in the figure below.

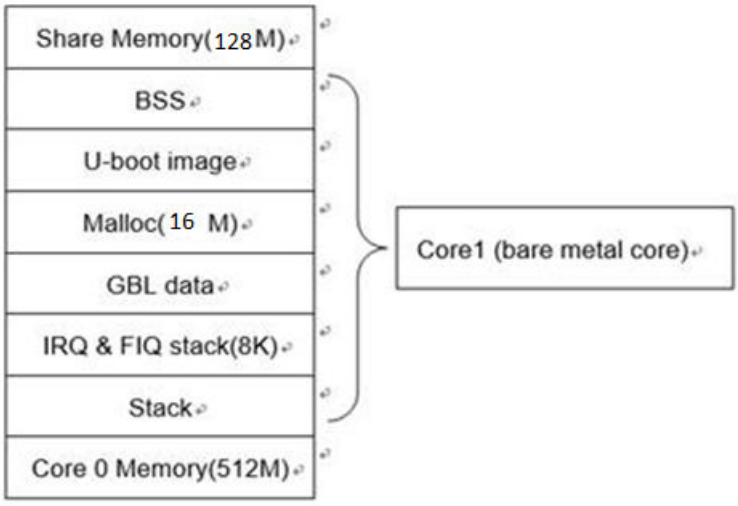


Figure 13. Memory configuration for I.MX6Q-SABRESD

The functions included in `malloc.h` in the table below can be used to allocate or free memory in program. Modify `CONFIG_SYS_MALLOC_LEN` in `include/configs/mx6sabresd_config.h` to change the maximum size of malloc.

Table 18. Memory APIs description

API name (type)	Description
void_t* malloc (size_t n)	Allocates memory • “n” – length of allocated chunk • Returns a pointer to the newly allocated chunk
void free (void *ptr)	Releases the chunk of memory pointed to by ptr (where “ptr” is a pointer to the chunk of memory)

4.4.6 i.MX8MM-EVK or i.MX8MP-EVK board

4.4.6.1 Linux DTS

When using baremetal, users should remove all the devices from kernel that baremetal has used, for example:

```
&fec1 {
status = "disabled";
      };
&gpio5
        {
status = "disabled";
        };
&uart3  {
status = "disabled";
        };
```

4.4.6.2 Memory configuration

This section describes the memory configuration for I.MX8MM-EVK or I.MX8MP-EVK boards.

1. The boards have a 6 GB DDR memory. To use the baremetal framework, configure DDR into

five partitions:

• 6016M for core0 (Linux)

• 32M for core1(bare metal)

• 32M for core2(bare metal)

• 32M for core3(bare metal)

• 32M for shared memory.

The configuration can be defined in the file `include/configs/imx8mm_evk.h.` or `include/configs/imx8mp_evk.h.`

```
#define CONFIG_SYS_DDR_SDRAM_SLAVE_RESERVE_SIZE (SZ_32M)
#define CONFIG_SYS_DDR_SDRAM_SHARE_RESERVE_SIZE (SZ_4M)
#define CONFIG_SYS_DDR_SDRAM_SLAVE_SIZE (SZ_32M)
```

2. Memory Reserve

For IPI data transfer, baremetal needs to share memory between master core and slave core, so users should reserve some memory from linux kernel, as shown in the following dts file:

```
        reserved-memory {
                #address-cells = <2>;
                #size-cells = <2>;
                ranges;
                bm_reserved: baremetal@0x60000000 {
                        no-map;
                        reg = <0 0x60000000 0 0x10000000>;
                };
        };
```

4.4.6.3 GPIO

1. Connect pin7 and pin8 of J1003

Because test_gpio case in baremetal use pin7 and pin8 of J1003, so should connect these two pins.

2. Boot the baremetal on slave core

if gpio is working well, will see the message:

[ok]GPIO test ok

3. Disable the devices from kernel

test_gpio case use GPIO5_7(pin8 of J1003) and GPIO5_8(pin7 of J1003), and these two pins are muxed as UART3_TXD and UART3_CTS, so should disable GPIO5 and UART3 from kernel.

```
&gpio5 {
status = "disabled";
};
&uart3 {
status = "disabled";
};
```

4.4.6.4 Ethernet

This section describes the Ethernet configuration settings for .MX8MM-EVK or I.MX8MP-EVK boards.

1. Disable the ethernet card from dts files:

```
&fec1 {
status = "disabled";
};
```

NOTE

1. I.MX8MM-EVK has only one NIC, default status of eth0(fec1) is disabled. if user don't use eth0 in baremetal, can enable fec1 in kernel dts file.
2. I.MX8MP-EVK has two NICs, default setting is eth0 for baremetal, eth1 for Linux.

2. Confirm baremetal configuration

```
make menuconfig
ARM architecture --->
[*] Enable baremetal
[*] Enable NIC for baremetal
(1)  which core that NIC is assigned to
```

Configure NIC to the specified core by modifying the NIC to assign that core value, which is the default configuration, to core1.

Chapter 5
Revision history

The table below summarizes the revisions to this document.

Table 19. Document revision history

Date	Version	Topic cross-reference	Change description
12/2020	1.10	Getting OpenIL	Updated the Open IL Git tag to `OpenIL-v1.10-202012`.
		i.MX8MM-EVK or i.MX8MP-EVK board	Added the section and sub-sections under it.
09/2020	1.9	Getting OpenIL	Updated the Open IL Git tag to `OpenIL-v1.9-202009`.
05/2020	1.8	Getting OpenIL	Updated the Open IL Git tag to `OpenIL-v1.8-202005`.
		Building the baremetal images for LX2160A-RDB board	Added the section.
		Building the baremetal images	Updated the section.
17/01/2020	1.7	Getting OpenIL	Updated the Open IL Git tag to `OpenIL-v1.7-202001`.
		SAI file and SAI	Added the sections.
31/08/2019	1.6	Getting OpenIL	Updated the Open IL Git tag to `OpenIL-201908`.
		Getting started	Added support for i.mx6q-sabresd board. • Added the section describing hardware setup for i.mx6q-sabresd board. • Added the section Building the baremetal images for i.mx6q-sabresd board.
30/04/2019	1.5	ENETC file	Added the section (a sample file for testing the ENETC feature).
		LS1028ARDB board	Added the section describing the ENETC configuration setting for LS1028ARDB
		Getting OpenIL	Updated the Open IL Git tag.
25/01/2019	1.4	Supported processors and boards	Added the support for LS1028ARDB. Modified and updated relevant sections throughout the document, where applicable.
		Getting OpenIL	Updated the Open IL Git tag to OpenIL-u-boot-201901.
15/10/2018	1.3.1	Ethernet	Updated the section, which describes Ethernet configuration settings for LS1043A or LS1046A reference design boards.
		Getting OpenIL	Updated the Open IL Git tag.
		Ethernet	Updated the section.

Table continues on the next page...

Table 19. Document revision history (continued)

Date	Version	Topic cross-reference	Change description
31/08/2018	1.3	Building the baremetal images from U-Boot source code	OpenIL Git tag updated to OpenIL-u-boot-201808.
		Getting OpenIL	Updated the section.
		USB file, PCIe file, and CAN file.	Support for the USB, PCIE, and FlexCAN features enabled. Added the sections in Section 4.2, Example software.
		USB, PCIe, and FlexCAN	Added these sections to describe the hardware resource allocation for LS1021A-IoT board.
31/05/2018	1.2	LS1043ARDB or LS1046ARDB board	Added the section for hardware resource allocation for these boards.
		Getting started	Added information relevant for LS1043ARDB and LS1046ARDB boards and for Open IL v1.2 support.
		Ethernet	Updated the section.
02/04/2018	1.1	-	Initial release aligned to Open IL v1.1.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 12/2020

Document identifier: IIoTBaremetalUG

arm