NXP Semiconductors

Document Number: IIoT_Baremetal_DG

Rev. 1.2, 05/2018

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.

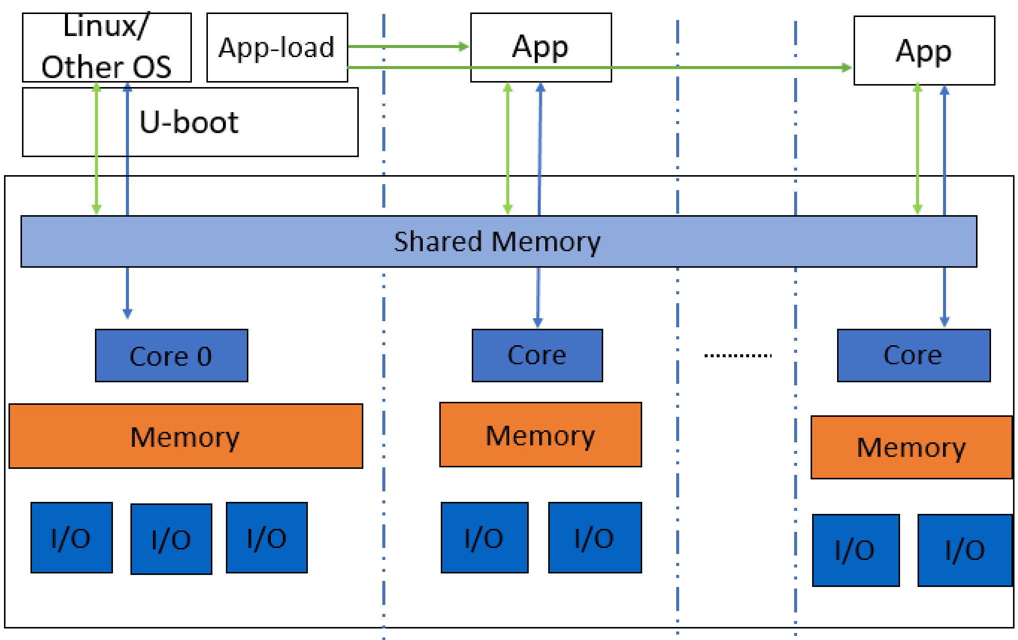

Figure 1. Baremetal framework architecture

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:

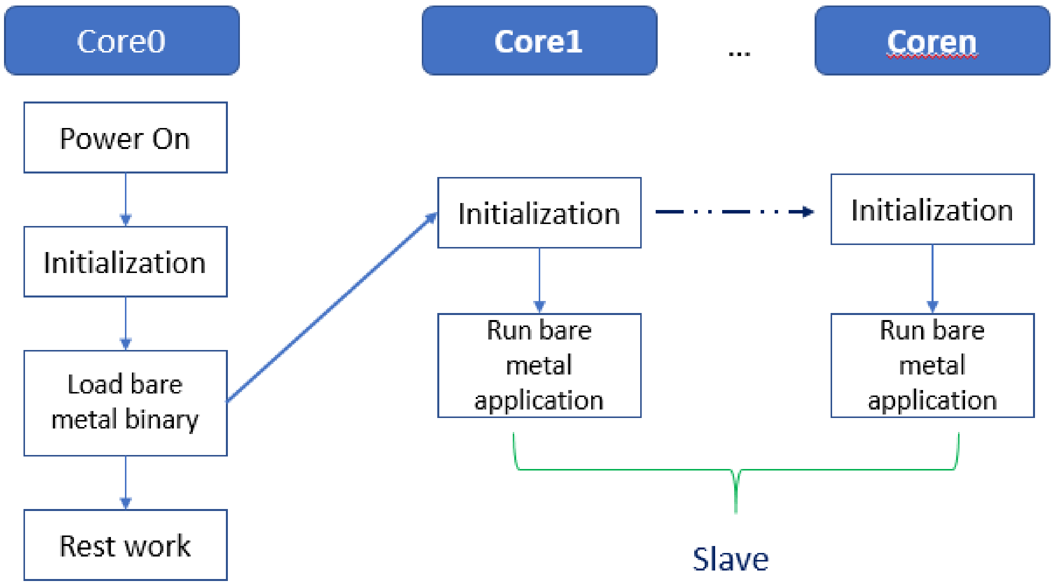

Figure 2. Baremetal framework boot flow diagram

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	IRQ, GPIO, IPI, data transfer, IFC, I2C, UART, QSPI
LS1043A	LS1043ARDB	IRQ, GPIO, IPI, data transfer, IFC, I2C, UART, FMan
LS1046A	LS1046ARDB	IRQ, GPIO, IPI, data transfer, IFC, I2C, UART, FMan, QSPI

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, or other QorIQ Layerscape board, serial cables
- Software: OpenIL v1.2 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.

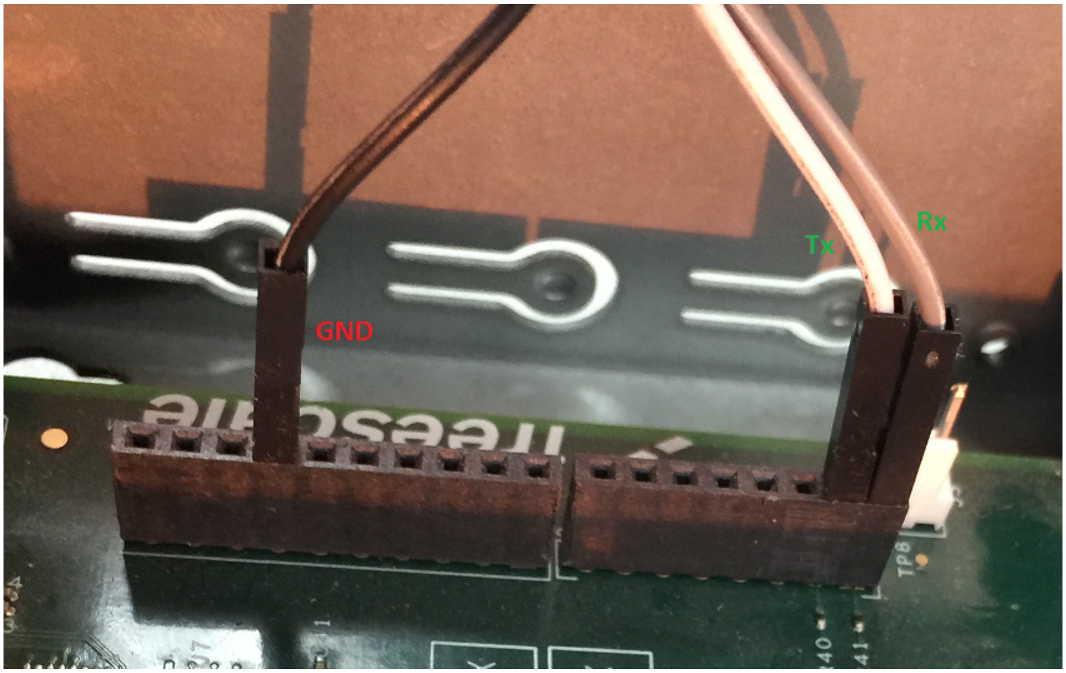

Figure 3. UART1 hardware connection

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

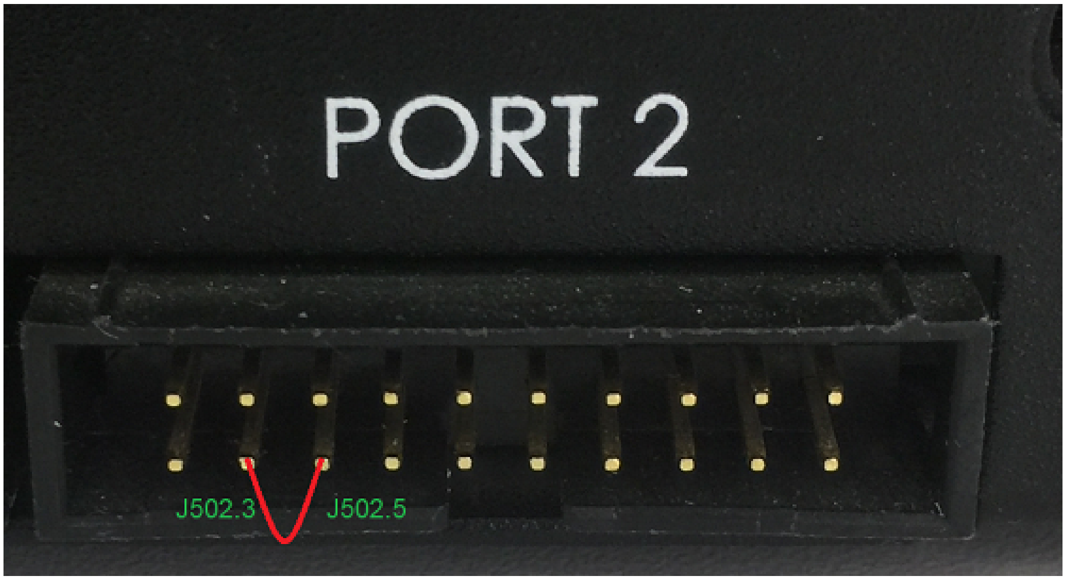

Figure 4. GPIO24 and GPIO25 connection on LS1021A-IoT

2.2.2 LS1043ARDB or LS1046ARDB

In case, either LS1043ARDB or LS1046ARDB board is used, 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 connect to the UART2 port.

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 on page 8.

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-u-boot-201803`.
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:

```
     $git clone https://bitbucket.sw.nxp.com/scm/dnind/industry-uboot.git
     $git checkout LSDK-17.09

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

/* building u-boot image for ls1043ardb board */
    $make ls1043ardb_sdcard_baremetal_defconfig
    $make

/* building u-boot image for ls1046ardb board */
    $make ls1046ardb_sdcard_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 `OpenIL-Baremetal-201803`.
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 ls1043ardb board */
   $make ls1043ardb_defconfig
   $make
/* build baremetal image for ls1046ardb board * /
   $make ls1046ardb_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 on page 7. 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 release, the Baremetal can be built and implemented conveniently.

Currently, there are two LS1021A-IoT baremetal defconfig files, one LS1043ardb baremetal defconfig file, and one LS1046ardb baremetal defconfig file:

- `configs/nxp_ls1021aiot_baremetal_defconfig`
- `configs/nxp_ls1021aiot_baremetal_ubuntu_defconfig`
- `configs/nxp_ls1043ardb_baremetal-64b_defconfig`
- `configs/nxp_ls1046ardb_baremetal-64b_defconfig`

2.4.1 Getting OpenIL

OpenIL v1.2 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-201803
$ git checkout -b OpenIL-201803 OpenIL-201803
```

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 LS1021A-IoT, LS1043ARDB, and LS1046ARDB boards.

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`
- `configs/nxp_ls1021aiot_baremetal_ubuntu_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 LS1043ARDB or LS1046ARDB

The two LS1043ardb or LS1046ardb baremetal default configuration files can be found in the *configs* directory. These are:

- `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_ls1043ardb_baremetal-64b_defconfig
```

or

```
$ make nxp_ls1043ardb_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

├── 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.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 ls1043ardb master core

├── kernel-ls1043a-rdb.itb --- kernel, dtb, and rootfs image for ls1043ardb master core

└── version.json

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 quickly

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 2GB 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. The similar steps can be followed for LS1043ARDB and LS1046ARDB.

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 on page 5.

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 on page 7 and Building the image through OpenIL on page 8 respectively. If 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.
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.

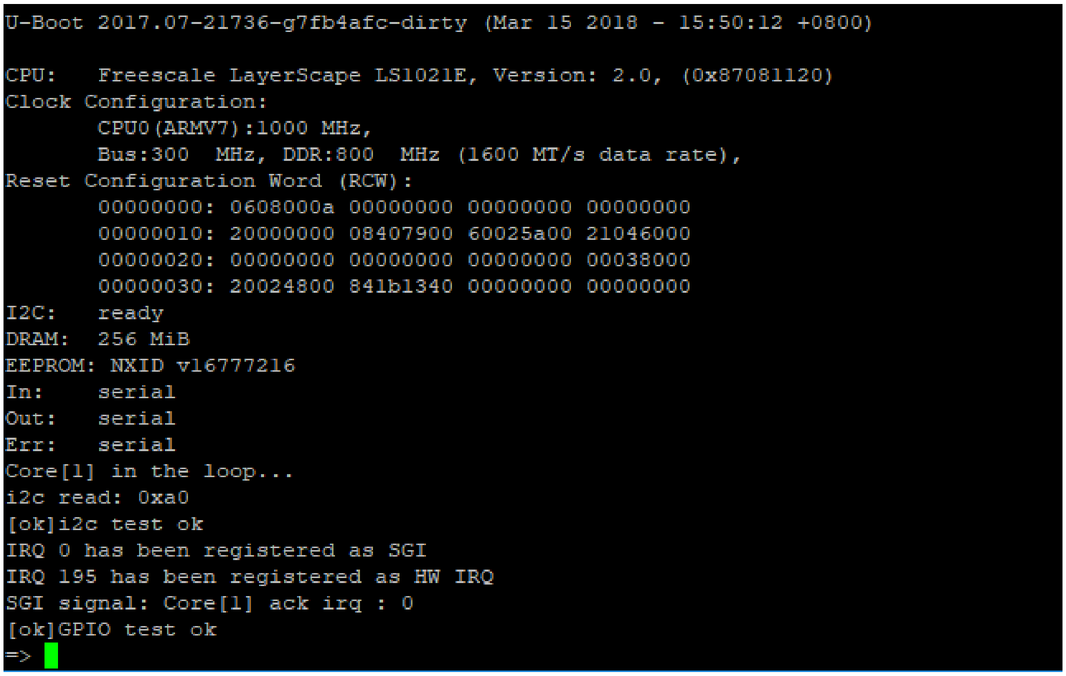

Figure 5. Baremetal output logs

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 on page 7, 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 3. 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.

Table continues on the next page...

Table 3. Common header file description (continued)

Header file	Description
`linux/types.h`	APIs specifying common types. 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 4. 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 5. 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 the ICC chapter. SGI IRQ id is 0-15, but 8 is reserved to be used for ICC.

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

Table 6. 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 7. 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.

Table continues on the next page...

Table 7. QSPI APIs (continued)

API name (type)	Description
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. • “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 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.
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 8. 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 I2C device chip. • void *packet • length - Network packet length Returns length
int eth_send (void *packet, int length)	Writes data to i2c device chip. • packet - Network packet length • length - Network packet length Returns length.

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.

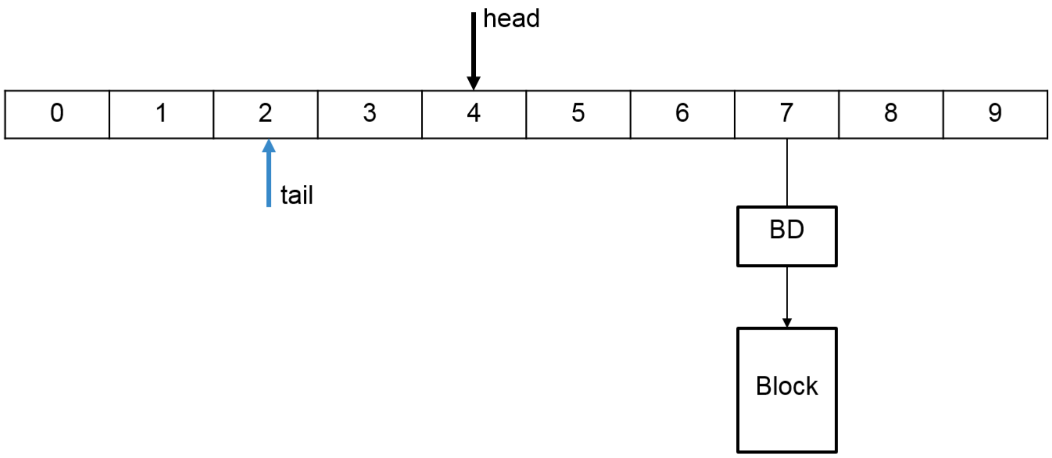

Figure 6. BD rings for inter-core communication

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.)

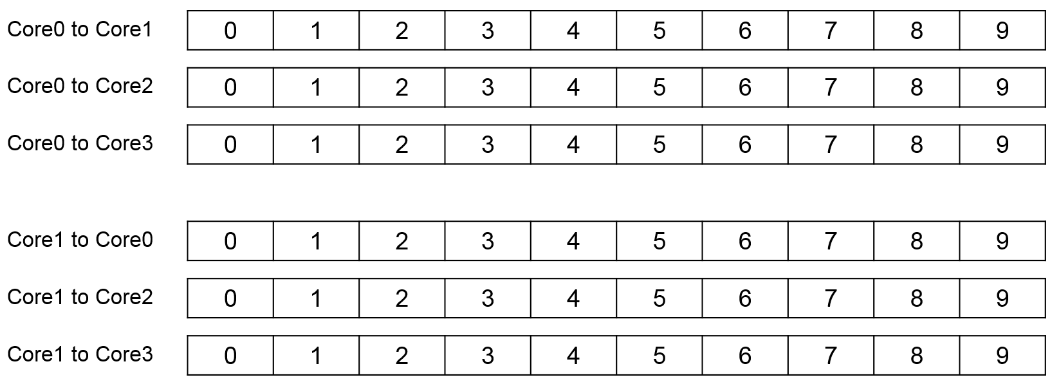

Figure 7. BD rings for multi-core platform

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.

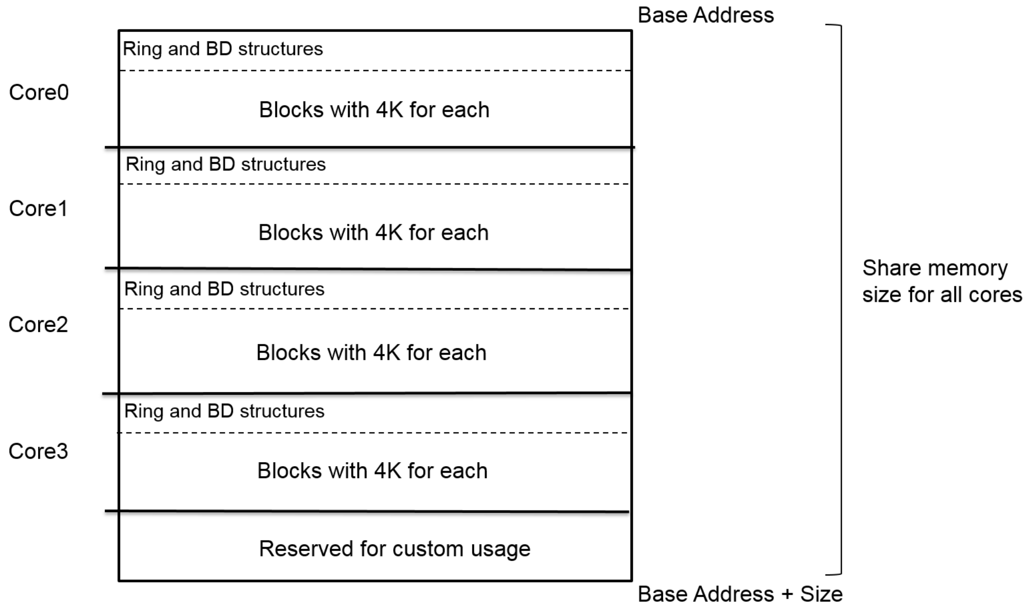

Figure 8. ICC shared memory map for the four-core platform

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 9. 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: • 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@OpenIL-Ubuntu:~# 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>
   ```

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

```
root@OpenIL-Ubuntu:~# icc send 0x2 0x1f 128
gic_base: 0xb6fa0000, share_base: 0xa7e87000, share_phy: 0xb0000000, block_phy:
0xb0200000
ICC send testing ...
Target cores: 0x2, bytes: 128
ICC send: 128 bytes to 0x2 cores success
all cores: reserved_share_memory_base: 0xbf000000; size: 16777216
mycoreid: 0; ICC_SGI: 8; share_memory_size: 125829120
block_unit_size: 4096; block number: 30208; block_idx: 0
#ring 0 base: 0xa7e87000; dest_core: 0; SGI: 8
desc_num: 128; desc_base: 0xb0000048; head: 0; tail: 0
busy_counts: 0; interrupt_counts: 0
#ring 1 base: 0xa7e87024; dest_core: 1; SGI: 8
desc_num: 128; desc_base: 0xb0000448; head: 1; tail: 1
busy_counts: 0; interrupt_counts: 1
```

3. The ICC module command examples on LS1021A-IoT Linux (Core 0) + Baremetal (Core 1) system:

   Run `icc send 0x2 0x1f 128` to send 128 bytes data 0x1f to core 1.

```
root@OpenIL-Ubuntu:~# icc send 0x2 0x1f 128
gic_base: 0xb6fa0000, share_base: 0xa7e87000, share_phy: 0xb0000000, block_phy:
0xb0200000
ICC send testing ...
Target cores: 0x2, bytes: 128
ICC send: 128 bytes to 0x2 cores success
all cores: reserved_share_memory_base: 0xbf000000; size: 16777216
mycoreid: 0; ICC_SGI: 8; share_memory_size: 125829120
block_unit_size: 4096; block number: 30208; block_idx: 0
#ring 0 base: 0xa7e87000; dest_core: 0; SGI: 8
desc_num: 128; desc_base: 0xb0000048; head: 0; tail: 0
busy_counts: 0; interrupt_counts: 0
#ring 1 base: 0xa7e87024; dest_core: 1; SGI: 8
desc_num: 128; desc_base: 0xb0000448; head: 1; tail: 1
busy_counts: 0; interrupt_counts: 1
```

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

```
=> Get the ICC from core 0; block: 0xb0200000, bytes: 128, value: 0x1f
```

4.4 Hardware resource allocation

This section describes how to modify the codes for our actual application.

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 10. 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.

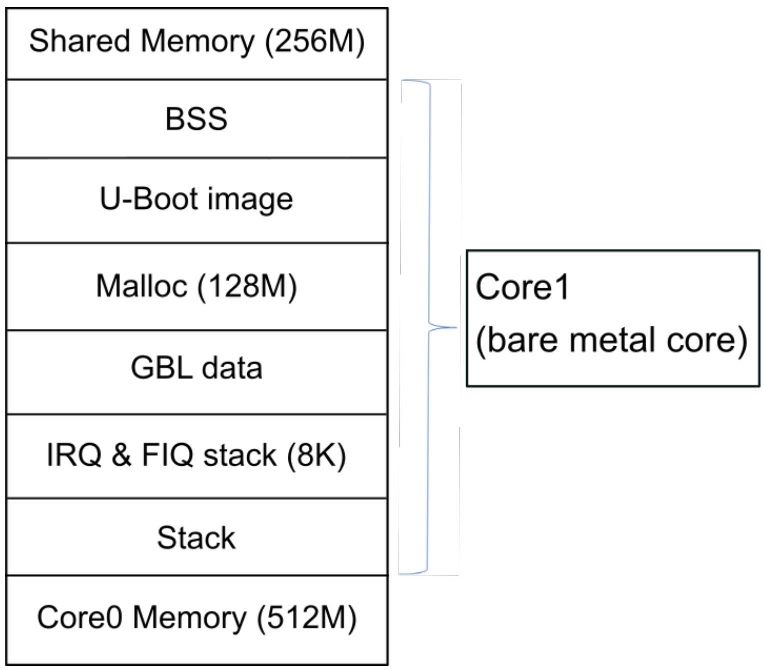

Figure 9. Memory configuration

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
```

4.4.2 LS1043ARDB or LS1046ARDB board

The following sections describe the hardware resource allocation for the LS1043ARDB or LS1046ARDB board.

4.4.2.1 Linux DTS

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

4.4.2.2 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.

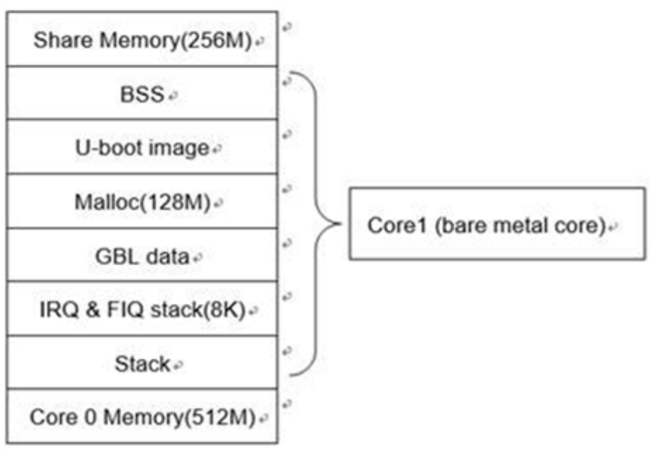

Figure 10. Memory configuration for LS1043ARDB or LS1046ARDB

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 11. 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

Table continues on the next page...

Table 11. Memory APIs description (continued)

API name (type)	Description
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`.

4.4.2.3 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";
 };
```

4.4.2.4 I2C

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

The LS1043ARDB (or 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.

4.4.2.5 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);
```

4.4.2.6 IFC

The LS1043ARDB has an IFC flash device. Configure the IFC bus in `ls1043ardb_config.h` using the command below:

```
#define CONFIG_FSL_IFC_COREID 1
```

Here, `CONFIG_FSL_IFC_COREID` defines the slave core which runs this IFC bus.

4.4.2.7 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.

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

```
  $make linux-menuconfig
 Drivers --->
      Staging drivers--->
         < > Freescale Datapath Queue and Buffer management
```

Now, edit the configuration file to add the FMan using the commands below:

```
 // include/configs/ls1043ardb.h:
 #define CONFIG_SLAVE_FMAN_CORE    1

 // include/configs/ls1046ardb.h:
 #define CONFIG_SLAVE_FMAN_CORE    1
```

Chapter 5
Revision history

The table below summarizes the revisions to this document.

Table 12. Revision history

Date	Version	Topic cross-reference	Change Description
31/05/2018	1.2	LS1043ARDB or LS1046ARDB board on page 24	Added the section for hardware resource allocation for these boards.
		Getting started on page 5	Added information relevant for LS1043ARDB and LS1046ARDB boards and for Open IL v1.2 support.
		Ethernet on page 17	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

Document Number: IIoT_Baremetal_DG
Rev. 1.2, 05/2018