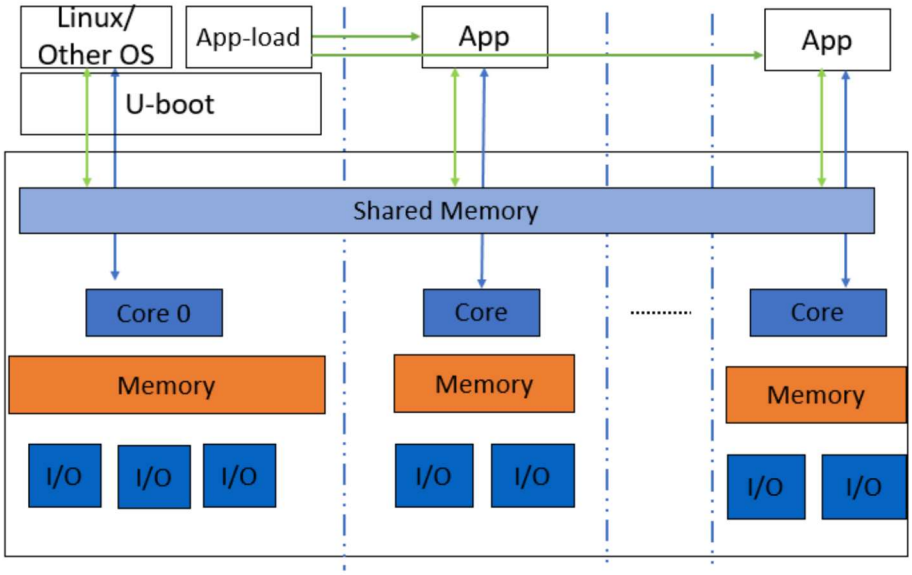
Application Development based on BareMetal framework in Real-time Edge Software

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



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 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:

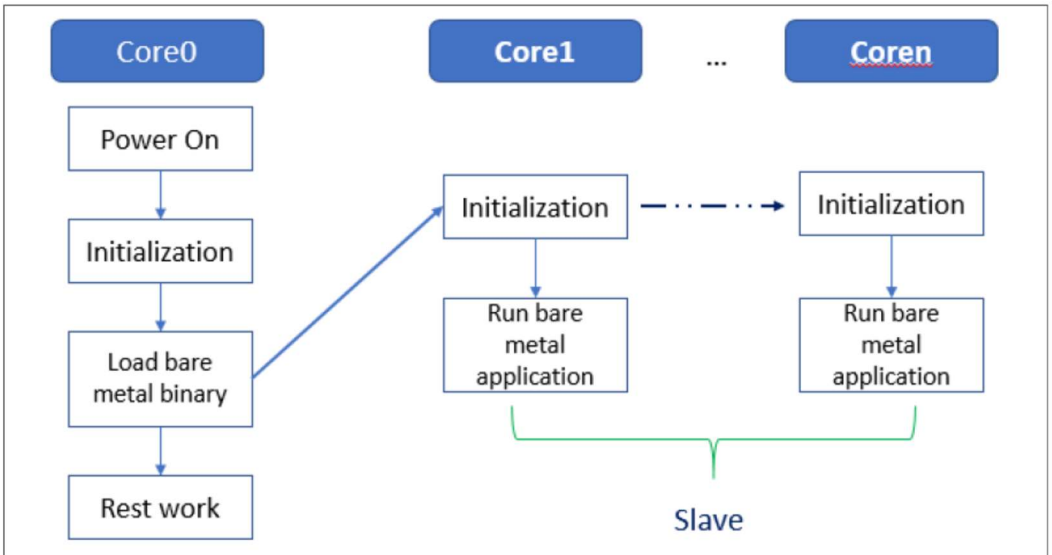


Figure 2. BareMetal framework boot flow diagram

This document describes how to develop customer-specific application based on BareMetal framework. The directory “app” stored in u-boot repository includes the use cases for testing GPIO, I2C, IRQ init, QSPI, Ethernet, USB, PCIe, CAN, ENETC and SAI features.

1. GPIO use case

The file app/test_gpio.c is one example to test the GPIO feature.

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

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; 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**.

2. I2C use case

app/test_i2c.c can be used as a sample to test the I2C feature.
On ls1021aiot board, include the I2C header file, i2c.h, which contains all interfaces for I2C. Then, 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*.

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 address (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 address (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.

3. IRQ use case

The file *app/test_irq_init.c* is a sample to test the IRQ and IPI (Inter-Processor Interrupts) feature. The process is described in brief below.
The file asm/interrupt-gic.h, is the header file of IRQ, and includes all its interfaces. Then, register an IRQ function for SGI 0. After setting an SGI signal, the CPU gets this IRQ and runs the IRQ function. Then, 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. SGI IRQ id is 0-15. The SGI IRQ id '8' is reserved for ICC.

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 mask • `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 mask • `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. QSPI Use case

The file app/test_qspi.c provides a sample 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. Subsequently, erase the corresponding flash area and confirm that the 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.

API name (type)	Description
`spi_find_bus_and_cs(bus,cs, &bus_dev, &new)`	The API finds if a 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. • "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. • "len" - Read the length of the data. • "vbug" - the buffer to store the data read 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. • "len" - Write the length of the data. • "buf" - the buffer to store the data write Returns 0 if OK, !0 on error.

5. Ethernet use case

The file app/test_net.c provides a sample 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 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 message host 192.168.1.2 is alive is displayed on the console.
4. The IP addresses 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.

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)`	Reads 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 - pointer to the packet is sent • length - Network packet length Returns length.

6. USB Use case

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_scanAPI.
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:

API name (type)	Description
`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 (if device not found).
`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 • Points 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.

7. PCIe use case

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.

API name (type)	Description
`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 value 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.

8. CAN Use Case

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

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 requires to be set. Returns: • 0 if Ok, • > 0 on error.
`void canDispatch(CO_Data* d, Message *m)`	CANopen handles data frames that CAN receives: • `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 100 µs.

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!

9. ENETC Use Case

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 example is a special case of using Net APIs.
The file test_net for ENETC is an example for the LS1028ARDB board with (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 message host 192.168.1.2 is alive is displayed on the console.
4. The IP addresses of the board and host machines 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.

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

10. SAI Use Case

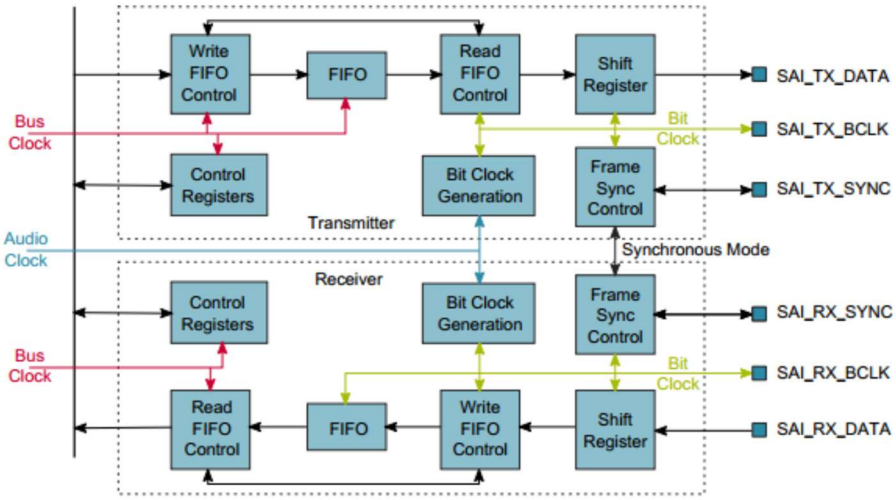
The audio feature needs SAI module and codec drivers. The following sections provide an introduction to SAI module and the audio codec (SGTL5000). These sections also describe the steps for integrating audio with BareMetal and running an audio application

on BareMetal.

Synchronous Audio Interface (SAI)

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

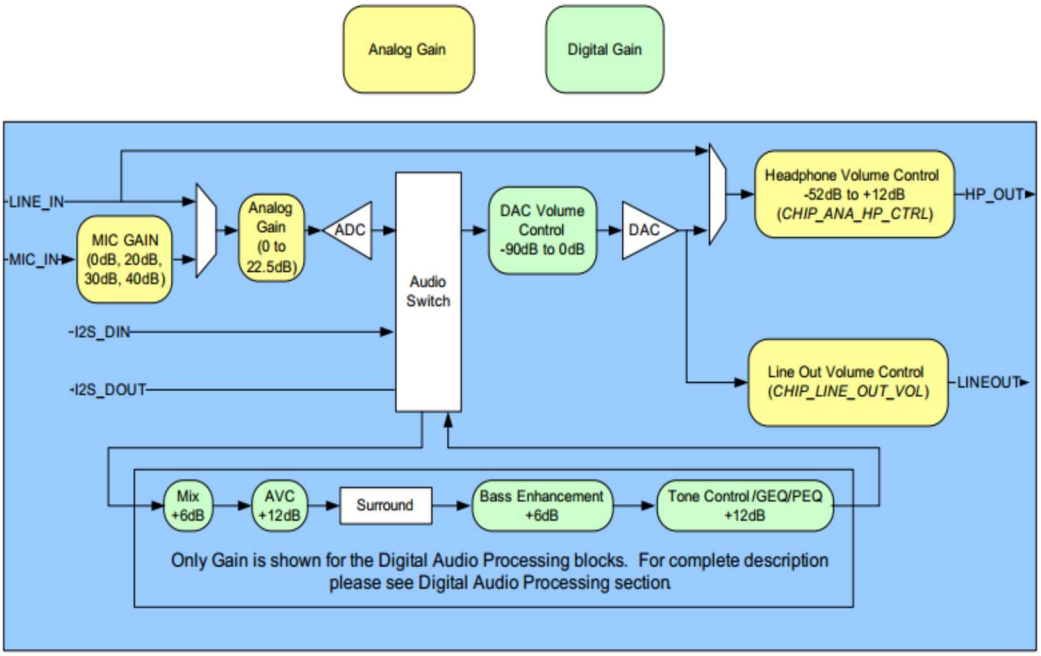
• Transmitter with independent bit clock and frame sync supporting 1 data line
• Receiver with independent bit clock and frame sync supporting 1 data line
• Maximum Frame Size of 32 words
• Word size of between 8-bits and 32-bits
• Word size configured separately for first word and remaining words in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



Audio codec (SGTL5000)

The SGTL5000 is a low-power stereo codec with headphone amplifier from NXP. It is designed to provide a complete audio solution for products requiring LINEIN, MIC_IN, LINEOUT, headphone-out, and digital I/Os. It allows an 8.0 MHz to 27 MHz system clock as input. The codec supports 8.0 kHz, 11.025 kHz, 12 kHz, 16 kHz, 22.05 kHz, 24 kHz, 32 kHz, 44.1 kHz, 48 kHz, and 96 kHz sampling frequencies. The LS1028ARDB board provides a 25 MHz crystal oscillator to the SGTL5000.

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



Play a demo audio file in slave core after booting the board:
=> 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!
****

11. Build and run the baremetal application.

1. Download the project source from the following path:
https://github.com/real-time-edge-sw/real-time-edge-uboot.git
2. Check it out to the tag:
• Real-Time-Edge-v2.3-baremetal-202207
3. Deploy baremetal application.
4. Configure cross-toolchain on your host environment.
5. Then, run the following commands:
/* build BareMetal image for LS1021A-IoT board */
$ make ls1021aiot_bm_defconfig
$ make
/* build BareMetal image for LS1028ARDB board */

$ make ls1028ardb_bm_defconfig
$ make
/* build BareMetal image with SAI for LS1028ARDB board */
$ make ls1028ardb_bm_sai_defconfig
$ make
/* build BareMetal image for LS1043ARDB board */
$ make ls1043ardb_bm_defconfig
$ make
/* build BareMetal image for LS1046ARDB board */
$ make ls1046ardb_bm_defconfig
$ make
/* build BareMetal image for LX2160ARDB board */
$ make lx2160ardb_bm_defconfig
$ make
5. Finally, the file u-boot.bin (or u-boot-dtb.bin, only for lx2160ardb) used for BareMetal is generated.

perform the steps below to run the BareMetal
binary from U-Boot prompt of master core. See the below example run on Layerscape platform:
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.
2. Then, start the BareMetal cores using the command below:
=> cpu 1 start 0x84000000
Note: *in command "cpu <num> start 0x84000000", the 'num' can be 1, 2, 3, ... to the max cpu number.*
The figure below displays a sample output log.

```
U-Boot 2017.07-21736-g7fb4afc-dirty (Mar 15 2018 - 15:50:12 +0800)

CPU:   Freescale LayerScape LS1021E, Version: 2.0, (0x87081120)
Clock Configuration:
       CPU0(ARMV7):1000 MHz,
       Bus:300  MHz, DDR:800  MHz (1600 MT/s data rate),
Reset Configuration Word (RCW):
       00000000: 0608000a 00000000 00000000 00000000
       00000010: 20000000 08407900 60025a00 21046000
       00000020: 00000000 00000000 00000000 00038000
       00000030: 20024800 841b1340 00000000 00000000
I2C:   ready
DRAM:  256 MiB
EEPROM: NXID v16777216
In:    serial
Out:   serial
Err:   serial
Core[1] in the loop...
i2c read: 0xa0
[ok]i2c test ok
IRQ 0 has been registered as SGI
IRQ 195 has been registered as HW IRQ
SGI signal: Core[1] ack irq : 0
[ok]GPIO test ok
=>
```