Getting Started with Multicore Programming for i.MX 6SoloX

1 Overview

i.MX 6SoloX is a multicore processor with asymmetric architecture. ARM® Cortex®-A9 and Cortex-M4 cores share a lot of resources, including power, clock, memory and peripherals.

To help programming the AMP (asymmetric multiprocessing) application on i.MX 6SoloX, this guide intends to introduce the problems and default solution in Linux and FreeRTOS BSP release.

Contents

1 Overview 1
2 Shared Resources 2
2.1 Power 2
2.2 Clock 2
2.3 Memory 2
2.4 Peripherals 2
3 Hardware mechanism for shared resource access 4
3.1 RDC 4
3.2 RDC SEMA42 4
3.3 SEMA4 4
4 Multicore default setting in Linux and FreeRTOS BSP 5
4.1 Power 5
4.2 Clock 7
4.3 Memory 10
4.4 Peripherals 12
5 Customize Linux and FreeRTOS BSP 16
5.1 Memory Customization 16
5.2 Peripheral Customization 16
5.3 RDC Setting Customization 17
6 Revision History 19

2 Shared Resources

2.1 Power

i.MX 6SoloX has centralized power generation and management system, which means if Linux on Cortex-A9 want to gate off some power domain to save power, it has to make sure the memory/peripheral used by Cortex-M4 core and Cortex-M4 itself is not in that domain. Otherwise the application on Cortex-M4 core will fail.

2.2 Clock

i.MX 6SoloX has centralized clock generation and management system as well. There's a huge clock tree that provides clock sources to CPUs, buses and peripherals with different frequency.

Once a clock node is updated with different parameter, the children nodes are impacted. Depending on the node position in the clock tree and the memory/peripheral allocation between Cortex-A9 and Cortex-M4 cores, the node change might influence only one core's application, or both.

The other problem is clock gating. In low power application, it's required that once some clock is no longer required, it should be gated off, and when all children clock nodes are gated off, the parent node should be gated off as well. In AMP system, the clock node might be used by both cores. The software clock manager has to synchronize the use count from both cores before take actions on the node.

So in multicore application, clock setting and modification must be considered very carefully.

2.3 Memory

There are quite some memory types in i.MX 6SoloX: DDR and QSPI flash are external memories and OCRAM, TCM (Tightly Coupled Memory), caches (L1 and L2) are internal memories. TCM and caches are seldom shared between cores while the others are different.

For example, DDR shared memory is used by default in RPMsg to exchange messages between cores; Cortex-M4 application can run in QSPI flash to achieve XIP (execution in place) to save boot time. In any case, we need to make sure there's no memory accessing conflicts.

Always keeping memory allocation between cores in mind can help avoiding a lot of crash issues.

2.4 Peripherals

In i.MX 6SoloX, all peripherals on the bus can be accessed by both Cortex-A9 and Cortex-M4 cores. It's obvious the collision will happen if both core want to access same peripheral.

It's recommended to allocate peripherals between cores before writing the multicore application, so that each core only accesses its own peripheral without conflict.

However, sometimes it’s hard to avoid shared access to same peripheral. For example, one GPIO register can control at most 32 pins, even Cortex-A9 and Cortex-M4 cores use different bits to control different pins, some mutual exclusive access is mandatory when updating the bit in GPIO register (write after read).

3 Hardware mechanism for shared resource access

On i.MX 6SoloX, some hardware mechanism is provided to help managing the share resources between Cortex-A9 and Cortex-M4 cores.

3.1 RDC

The Resource Domain Controller (RDC) provides robust support for the isolation of destination memory mapped locations such as peripherals and memory to a single core, a bus master, or set of cores and bus masters.

The key concept is RDC domain. RDC has 4 domains, memory or peripheral access permission is assigned to the domains. Each bus master (including CPU) can only be bound to one domain. The default setting after reset is that all the bus masters are bound to domain 0 and all domains have read/write permission to all memory regions and peripherals.

3.2 RDC SEMA42

Mutual exclusive control is required when some peripheral need to be accessed by more than 1 domain. RDC allocates one hardware semaphore (SEMA42) for each peripheral. The core can acquire the semaphore when it’s free; otherwise it has to wait the occupier to release it.

When setting peripheral access permissions for RDC domains, it’s possible to force acquiring SEMA42 before peripheral access. This prevents unexpected violation of shared peripheral access.

3.3 SEMA4

Hardware Semaphore (SEMA4) is a common semaphore that can be used anywhere as required. For example, it can be used to access shared memory, or guarantee some sequence between multicore.

In addition to the fundamental mutual exclusive control, SEMA4 provides interrupt which indicates the semaphore is released by the occupier. By this way, if some core wants to acquire the semaphore, it doesn’t have to wait on it by busy polling.

4 Multicore default setting in Linux and FreeRTOS BSP

When Linux is running on Cortex-A9 core and FreeRTOS application is running on Cortex-M4 at same time, the shared resources problems must be well handled.

In Linux release package, there are several device tree files named "*-m4.dtb". Only with such kind of device tree configuration, Linux can run along with FreeRTOS application.

The resources obtained by Cortex-M4 application differ among the examples and demos. To make Linux image work with all FreeRTOS applications, the "*-m4" dtb reserves the union set of resources used by all FreeRTOS applications. The purpose is just to demonstrate how to allocate resources between cores to make 2 cores run simultaneously without error and the user of Linux and FreeRTOS BSP need to customize the resource allocation per requirement.

4.1 Power

i.MX 6SoloX has 3 power modes: RUN, WAIT and STOP.

System power is managed in Linux, the Cortex-M4 core is considered as a peripheral of Cortex-A9 core. Only when the application on Cortex-M4 tells Linux that it runs into low power mode, Linux can switch the system to WAIT or STOP mode.

There are 2 FreeRTOS applications at "*examples/<board>/demo_apps/low_power_imx6sx*" in FreeRTOS BSP release package that demonstrate the low power switch mechanism. The periodic WFI demo presents how to run Cortex-M4 in low power mode and be woken up by a timer event; And the CAN wakeup demo shows how to wake up the system in WAIT/STOP mode by a CAN message.

The other FreeRTOS BSP examples or demos don't leverage low power feature so that Linux will never make system WAIT/STOP when running with them.

4.1.1 Understanding FreeRTOS low power scenario

4.1.1.1 Low power management module for multicore

The module is defined in *examples/<board>/demo_apps/low_power_imx6sx/common/lpm_mcore.h*.

It mainly provides 3 features:

1) ***LPM_MCORE_[GS]etPowerStatus()*** - Set/Get power status of Cortex-M4 core

   STATUS_NORMAL_RUN indicates Cortex-M4 core is in running mode, Linux can't switch system into WAIT/STOP mode.

   STATUS_LOWPOWER_RUN indicates Cortex-M4 core enters low power mode, and Linux can switch the system into WAIT/STOP mode.

   NOTE

   1. The API here just set/get the status variable and won't really handshake with Linux.

2. Cortex-M4 status can be set to low power mode only when all peripherals used by Cortex-M4 core are inactive or can work in STOP mode;

2) ***LPM_MCORE_RegisterPeerWakeup()*** – Enable/Disable system wakeup source

   When i.MX 6SoloX enters STOP mode, only registered interrupt source can wake up the system. This API notifies Linux that some wakeup source from FreeRTOS application needs to be registered. When the interrupt is triggered, system in STOP mode will first wake up Cortex-A9 core, and then Linux will restore the power/clock/memory etc and wake up FreeRTOS application.

3) ***LPM_MCORE_WaitForInt()*** - Wait for interrupt

   This API transacts with Linux according to the status of Cortex-M4 core. If current mode is *STATUS_LOWPOWER_RUN*, the application will run into TCM, and process status synchronization with Linux. Otherwise it only runs WFI instruction.

   FreeRTOS idle task context is recommended for this function call because it ensures no other task is active. However if there's only one task or the application is running in bare metal environment, it can also be called in main context.

4.1.1.2 Typical sequence of low power application

1) System initialization – including low power management initialization
2) Run application logic
3) If the application is idle (no working peripheral and CPU logic), then
   a. *LPM_MCORE_RegisterPeerWakeup()* – Register the wakeup source
   b. *LPM_MCORE_SetPowerStatus()* to low power mode
   c. Wait on some event or call *LPM_MCORE_WaitForInt()* directly
4) After the application is woken up
   a. *LPM_MCORE_SetPowerStatus()* to normal run mode
   b. *LPM_MCORE_RegisterPeerWakeup()* – Unregister the wakeup source
5) Run application logic

Although system won't always go into STOP mode (depending on Cortex-A9 status) when Cortex-M4 core goes into low power mode, it's recommended to register wakeup source to ensure Cortex-M4 core can process the event in time.

4.1.2 Understanding Linux low power scenario

Linux has many perspective of low power scenario, but on i.MX6SX, we only have bus-freq and suspend/resume need to consider M4's behavior.

1) Bus-freq: scale bus frequency according to system bus loading requirement in order to save as much power as possible as well as make system performance high enough for user experience. M4 is considered as a peripheral device of A9, it needs to request/release bus-freq when it is running at

>24MHz and 24MHz using OSC, below message are used for busfreq communication between M4 and A9:

#define MU_LPM_BUS_HIGH_READY_FOR_M4 0xFFFF6666
--When M4 receive this command from MU, means linux high bus setpoint is ready for M4, M4 can now do high speed operations.
#define MU_LPM_M4_REQUEST_HIGH_BUS 0x2222CCCC
--When A9 receive this command from MU, means M4 request a high bus setpoint, linux need to make sure busfreq is staying at high bus.
#define MU_LPM_M4_RELEASE_HIGH_BUS 0x2222BBBB
--When A9 receive this command from MU, means M4 does NOT need high bus, Linux can enter low bus setpoint if it can.

2) Suspend/resume: when Linux kernel suspend, system will enter STOP/WAIT mode according to M4's status:
    a. M4 is sleeping and its frequency is low(24MHz): A9 will put CCM into STOP mode and A9 core will be powered off as well as all power domains that can be off, it is also called DSM mode;
    b. M4 is NOT sleeping or its frequency is high(> 24Mhz): A9 will put CCM into WAIT mode and M4 can still proceed its tasks.
    c. When DSM is entered, all necessary context will be saved/restored by A9 when resume, such as QSPI, OCRAM etc..
    d. M4 can register wakeup source via MU, then when system is in WAIT/STOP mode, the wakeup source registered by M4 can wake up system from WAIT/STOP mode, below message is for registering wakeup source of M4:

    #define MU_LPM_M4_WAKEUP_SRC_VAL 0x55555xxx
    --When A9 receive this command from MU, means M4 want to add wakeup source, bit [4~11] is the interrupt number and bit [0~3] means enable or disable, a value NOT 0 means enable this wakeup source.

4.2 Clock

There are 2 main problems in clock management of multicore: clock configuration update (for example routing or divider change) and clock gate control.

- Clock configuration update

In Linux/FreeRTOS BSP, the easiest solution is used: Statically align the clock setting between Linux and FreeRTOS application.

FreeRTOS application only uses slow peripherals (UART/I2C/eCSPI/CAN etc) which can be assigned OSC 24M clock source, and divider is always 1. Linux won't touch those configurations so that running together has no problem.

- Clock gate control

Due to clock sharing is unavoidable on i.MX 6SoloX, a shared region in OCRAM is allocated to synchronize use count of certain clock node. When FreeRTOS application needs some clock to be always open, it needs to mark use count area in OCRAM to be “1”, then Linux won’t disable the gate.

The layout of OCRAM is as following (base address is 0x0091F000):

Offset	Content
0x0	A9 Ready Magic
0x4	M4 Ready Magic
0x8	A9 node0 context
0x10	M4 node0 context
0x18	A9 node0 use count
0x19	M4 node0 use count
0x1C	A9 node1 context
0x24	M4 node1 context
0x2C	A9 node1 use count
0x2D	M4 node1 use count
…	…

The first 2 “ready magic” are important to know the validity of the node values. If the magic number is wrong, the remaining node values should just be ignored. The magic number should be cleared by bootloader, otherwise it might indicate clock nodes ready by mistake after reset because the OCRAM data is not cleared automatically by reset.

Node context is used by Cortex-A9 and Cortex-M4 to store its own information related to shared clock nodes. The context data format is private so one core doesn’t care about the context used by the other.

The node use count indicates whether the clock node is in use by some core. Each core needs to check the other one’s clock use count before enabling or disabling the clock gate.

Before updating the content in the share memory, SEMA4 is used to make sure there's no access collision. The SEMA4 gate number is 6.

4.2.1 Understanding FreeRTOS clock gating scenario

The FreeRTOS demo of clock gate control is same as the low power demo at "*examples/<board>/demo_apps/low_power_imx6sx*". To make it simple and minimize the code size, FreeRTOS BSP doesn't provide a clock manager that controls whole clock tree. Instead, there's only a utility at *examples/<board>/demo_apps/low_power_imx6sx/common/shared_clock_manage.h* which provides the capability how to operate shared clock node. The demo just needs to make sure Linux won't disable the clock by mistake, and it initializes all required clock nodes in shared memory. In real application, user might need to operate the node info in shared memory as well as control the clock gate, in critical section protected by SEMA4.

Here's the features provided by shared_clock_manager.h:

1) ***SharedClk_[Acquire/Release]Lock()*** – Acquire/Release SEMA4
2) ***SharedClk_[Enabe/Disable]Node ()*** – Set clock node use count to 1 or 0
3) ***SharedClk_GetPeerNodeStatus ()*** – Check whether the clock node is used by Linux
4) ***SharedClk_GetPeerReady()*** – Check whether Linux is ready for shared clock operation

4.2.2 Understanding Linux clock gating scenario

Linux has its clock management framework, the only difference for the scenario of both A9 and M4 are active is as below:

1) Before A9/M4 clock/power management handshake is done, for those shared clock nodes, all clock disable operations will be skipped and clock enable operations are still proceed;
2) When clock/power management handshake is done, for those shared clock nodes:
    a. when Linux is about to disable a clock, it will check FreeRTOS's clock usage, if the clock is enabled in FreeRTOS, Linux will only set the disable flag in shared memory for this clock index, and skip the hardware operation, if the clock is disabled in FreeRTOS, Linux will proceed the hardware operation to disable the clock;
    b. when Linux is about to enable a clock, it will check FreeRTOS's clock usage, if the clock is enabled in FreeRTOS, Linux will only set the enable flag in shared memory for this clock index, and skip the hardware operation, if the clock is disabled in FreeRTS, Linux will proceed the hardware operation to enable the clock.

The handshake magic number and the shared clock nodes location are as below:

#define SHARED_MEM_MAGIC_NUMBER 0x12345678
arch/arm/boot/dts/imx6sx-sdb-m4.dts
arch/arm/boot/dts/imx6sx-sabreauto-m4.dts

4.3 Memory

FreeRTOS application can run on different memory area, and “hello world” demo with different linker file are provided to demonstrate the capability. Linux reserved all the areas that FreeRTOS demo requires by “*-m4.dtb”. In real application, user needs to customize the Linux device tree on demand. For example, if the Cortex-M4 application runs in OCRAM, the DDR and QSPI areas don’t need to be reserved in Linux.

4.3.1 Memory allocation

The memory allocation between Cortex-A9 and Cortex-M4 is as following:

- OCRAM

Start Address	End Address	Size	Content
0x00900000	0x0090FFFF	64KB	Linux OCRAM area
0x00910000	0x00917FFF	32KB	FreeRTOS code region
0x00918000	0x0091EFFF	28KB	Reserved for FreeRTOS code region extension
0x0091F000	0x0091FFFF	4KB	Shared clock nodes

- QSPI flash

Start Address	End Address	Size	Content
0x68000000	0x68007FFF	32KB	FreeRTOS code region on Sabre-AI board
0x68008000	-	-	FreeRTOS code region extension on Sabre-AI board or used for other purpose
0x78000000	0x78007FFF	32KB	FreeRTOS code region on Sabre-SD board
0x78008000	-	-	FreeRTOS code region extension on Sabre-SD board or used for other purpose

- DDR

Start Address	End Address	Size	Content

0x80000000	0x9FEFFFFF	511MB	Linux kernel
0x9FF00000	0x9FF07FFF	32KB	FreeRTOS code region
0x9FF08000	0x9FFFFFFF	1MB - 32KB	Reserved for FreeRTOS code region extension
0xA0000000	0xBFEFFFFF	511MB	Linux kernel
0xBFF00000	0xBFFEFFFF	1MB - 64KB	Reserved for RPMsg vring extension
0xBFFF0000	0xBFFFFFFF	64KB	RPMsg vring share memory
0xC0000000	0xFFFFFFFF	1GB	Linux kernel if board has 2GB DDR

4.3.2 Memory isolation

Since both cores can access all memory area, some mechanism need to be provided to avoid the area dedicated for one core being polluted by the other core. RDC is used to achieve this mechanism, Cortex-M4 core is assigned to RDC domain 1 and Cortex-A9 core keeps unchanged after reset in RDC domain 0.

4.3.2.1 Understanding FreeRTOS memory isolation scenario

There are several hello world demos in FreeRTOS BSP release, each one represents the case that FreeRTOS application runs on specific space.

Memory	Demo Path
TCM	*examples/<board>/demo_apps/hello_world*
OCRAM	*examples/<board>/demo_apps/hello_world_ocram*
QSPI flash	*examples/<board>/demo_apps/hello_world_qspi*
DDR	*examples/<board>/demo_apps/hello_world_ddr*

Open the hardware_init.c file in each folder to see the difference of memory isolation. *RDC_SetMrAccess()* is used to protect the running area of Cortex-M4 core.

- TCM memory is Cortex-M4 internal memory and memory isolation is not required.

- DDR and QSPI memory for Cortex-M4 application is reserved for access by domain 1 only.
- OCRAM is a special case that Linux needs access permission in low power mode to save/restore OCRAM content. Therefore although RDC is set, it allows all domains access permission.

4.3.2.2 Understanding Linux memory isolation scenario

When working with M4, Linux only define the memory ranges used for kernel only. The memory ranges for M4 are all reserved and hided for kernel. The DTB source file is the only place for this memory range definition. A typical memory definition according to the memory allocation table described above could be found in *imx6sx-sdb-m4.dts*

- **OCRAM**

  &ocram {

  reg = <0x00901000 0xf000>;

  };

  Linux kernel only uses OCRAM from 0x901000 to 0x910000. Other spaces are reserved for M4.

- **QSPI**

  &qspi2 {

  status = "disabled";

  };

  The QSPI2 is disabled in Linux kernel, so memory mapped space of QSPI2 from 0x70000000 to 0x80000000 are reserved for M4.

- **DDR**

  memory {

  linux,usable-memory = <0x80000000 0x1ff00000>,

  <0xa0000000 0x1ff00000>;

  };

  Linux kernel only uses DDR memory from 0x80000000 to 0x9FF00000 and 0xA0000000 to 0xBFF00000. The memory holes 0x9FF00000 – 0x9FFFFFFF and 0xBFF00000 – 0xBFFFFFFF are reserved for M4.

4.4 Peripherals

Peripheral allocation between cores is mainly dependent on the application running on Cortex-M4 core. If some peripherals are obtained by Cortex-M4 exclusively, then it must be removed from Linux dtb. In

addition to exclusive peripherals, there are also shared peripherals which cannot be easily divided between cores.

4.4.1 Exclusive Peripherals

There are quite some examples/demos in FreeRTOS BSP release, each needs specific peripherals. To make all the applications in FreeRTOS BSP work with Linux, Linux reserves the union set of all the peripherals required by all FreeRTOS applications.

Here's the list of peripherals reserved for Cortex-M4 core:

- ADC1
- ADC2
- ECSPI4
- ECSPI5
- EPIT1
- EPIT2
- FLEXCAN1
- FLEXCAN2
- I2C3
- UART2
- WDOG3

Some peripherals are designed for multicore and it could be regarded as exclusive peripheral as well, including MU, SEMA4 and RDC SEMA42. Following peripherals are used in Linux/FreeRTOS BSP by default:

Peripheral	Use
MU TX/RX 0	Low power protocol between cores
MU TX/RX 1	RPMsg message exchange and notification
SEMA4 gate 6	Shared clock nodes protection

4.4.1.1 Understanding FreeRTOS peripheral isolation scenario

In each example or demo's hardware_init.c, *RDC_SetPdapAccess()* is used to preserve the peripherals for Cortex-M4 application.
For example, *examples/imx6sx_sdb_m4/driver_examples/i2c_imx/i2c_polling_sensor/hardware_init.c* contains RDC setting that allows I2C3 only accessible by domain 1.
In addition to the specific peripheral assignment in FreeRTOS application, UART2 is needed by all applications to support the debug console. UART2 RDC setting is called in *dbg_uart_init()* from *examples/<board>/board.c*.

4.4.1.2 Understanding Linux peripheral isolation scenario

Linux does not implement the RDC protection for peripherals. It uses a specific DTB to disable devices used by M4 to avoid being accessed by Linux kernel. The DTB source file is *arch/arm/boot/dts/imx6sx-xxxx-m4.dts*. Customers who have implemented customized M4 peripherals need to modify the Linux DTB source file to disable them explicitly.

Peripherals below are disabled in *imx6sx-sdb-m4.dts* as they will be used for M4.

```
&flexcan1 {
        status = "disabled";
};

&flexcan2 {
        status = "disabled";
};
&i2c3 {
      status = "disabled";
};
&uart2 {
        status = "disabled";
};

&adc1 {
      status = "disabled";
};

&adc2 {
      status = "disabled";
};
```

Other peripherals like WDOG3, EPIT1, EPIT2, ECSPI4 and ECSPI5 are also disabled in *imx6sx.dtsi*, since they are not used by Linux kernel at default.

U-boot implements a RDC driver can support to set peripheral isolation and sharing. Since M4 sets its dedicated peripherals by itself, the peripherals RDC setting in u-boot are unnecessary. Currently, only shared peripheral GPIO1 use the u-boot to configure its RDC setting.

4.4.2 Shared Peripherals

Although it's recommended to separate peripherals on different cores exclusively, some peripherals are shared. The typical peripherals are IOMUXC, CCM and GPIO.

IOMUXC has separate registers for each pin, so in most cases, it's safe to access from both cores without collision (each core only configures its own peripheral pins).

Sharing CCM is problematic as each register controls several clock nodes. By default FreeRTOS application always runs before Linux and the clock configuration happens only at initialization. So there's no conflict in accessing CCM. User need to think more about it in real application. If it's possible to access CCM from both sides at same time, SEMA4 must be used from both sides, just like we do for shared clock use count update.

Like CCM, each GPIO register controls many pins, if some pin needed by Cortex-M4 and some pin needed by Cortex-A9 reside on same register, access protection is necessary. Linux doesn't provide GPIO access protection so that user needs to implement such protection by customization if GPIO shared access is required.

4.4.2.1 Understanding FreeRTOS shared peripheral scenario

In FreeRTOS BSP, an example introduces how to share peripheral by different cores by RDC SEMA42.

In *examples/<board>/demo_apps/blinking_imx_demo/gpio_ctrl.c*, *RDC_SEMAPHORE_Lock()* and *RDC_SEMAPHORE_Unlock()* create a critical section which could make GPIO access safe, in case Linux is customized to access GPIO in RDC SEMA42 critical section as well.

To make the GPIO peripheral accessible by both sides, blinking_imx_demo assigns GPIO to all domains in *examples/<board>/demo_apps/blinking_imx_demo/hardware_init.c*. It's also possible to force SEMA42 being acquired before peripheral access, if "sreq" parameter is set to "true" in *RDC_SetPdapAccess()*.

5 Customize Linux and FreeRTOS BSP

User needs to know how to tailor Linux and FreeRTOS BSP for specific application, especially in memory and peripherals customization. All modifications for one core must be aligned with the other one.

5.1 Memory Customization

5.1.1 FreeRTOS BSP memory customization

Here's the location for memory region definition:

Region	Location
code region	Link file at platform/devices/MCIMX6X/linker/<toolchain>
RDC Access Control	examples/<board>/demo_apps/hello_world_xxx/hardware_init.c

5.1.2 Linux BSP memory customization

Modify dts file for specific board.

5.2 Peripheral Customization

5.2.1 FreeRTOS BSP peripheral customization

Here's the location for peripheral allocation:

Peripheral	Location
RDC Access Control	examples/<board>/ driver_examples/…/hardware_init.c examples/<board>/demo_apps/…/hardware_init.c

5.2.2 Linux BSP peripheral customization

Modify dts file for specific board.

5.3 RDC Setting Customization

Once memory and peripheral allocation is decided, user needs to know where to set RDC for resource isolation and resource sharing. Such setting should be as early as possible so that both Cortex-A9 and Cortex-M4 cores can run without resource access violation.

Default RDC setting is done in FreeRTOS application to demonstrate how to allocate resource on demand. In real application, user can put RDC settings into bootloader (e.g. U-Boot) to make sure resources are well accommodated at very beginning.

U-boot has implemented a RDC driver and shared GPIO driver. The RDC driver supports to configure masters and peripherals' RDC attributes.

5.3.1.1 Peripherals RDC setting

Peripherals could be set to isolated or shared. Interface *imx_rdc_setup_peripherals* is used for peripheral RDC configuration. Customers need to define the peripherals RDC domain in a array, then pass the array as a parameter to this function.

Two examples for this array definition

1. GPIO1 shared by domain 0 and domain 1. The SEMA42 is enabled implicitly once the peripheral is set to multiple domains.

   ```
   static rdc_peri_cfg_t const shared_resources[] = {
           (RDC_PER_GPIO1 | RDC_DOMAIN(0) | RDC_DOMAIN(1)),
   };
   ```

2. GPIO1 isolated by domain 0. Other domain can't access it.

   ```
   static rdc_peri_cfg_t const isolated_resources[] = {
           (RDC_PER_GPIO1 | RDC_DOMAIN(0) ),
   };
   ```

5.3.1.2 Masters RDC setting

RDC only supports to arrange masters to one domain. Masters can't share multiple domains like peripherals. The RDC driver provides an interface *imx_rdc_setup_masters* for masters RDC configuration. An array of *rdc_ma_cfg_t* must be passed to this function as a parameter.

Example for array definition:

Here we set the M4 core to domain 1. *RDC_MASTER_CFG* macro is used for generating the item.

```
static rdc_ma_cfg_t const master_resources[] = {
```

```
        RDC_MASTER_CFG (RDC_MA_M4 , 1),
};
```

6 Revision History

This table summarizes the revisions made to this document.

Revision number	Date	Substantive changes
0	11/2015	Initial version.