Application Note

Document No.: AN1135

G32R501 IPC Application Note

Version: V1.1

1 Introduction

This document mainly introduces the G32R5xx IPC module, including a basic overview, register access, shared RAM, and the dual-core boot process. It also provides IPC application examples: dual-core communication with the message mailbox, resource sharing, and the event-control flow.

For a detailed description of the IPC module, refer to the G32R5xx user manual.

Contents

2 G32R5xx IPC Module Introduction

2.1 Overview

G32R501 is a dual-core microcontroller with an isomorphic multi-core asymmetric architecture. It integrates two Arm® Cortex®-M52 cores, with CPU0 as the master core and CPU1 as the slave core.

In dual-core mode, the two cores need to communicate. G32R501 provides an IPC (Inter-Processor Communication) module for dual-core communication.

The G32R501 IPC module includes three functional parts:

- Interrupt control and data sharing (mailbox):
  - Share data through 4 registers, and generate 12 interrupts for each CPU.
  - Exchange data through shared memory.
- Event control
  - Communicate between the two CPUs using the polling method and RXEV.
- WRPRT
  - Implements the WRPRT function. Write-register protection prevents a CPU from incorrectly writing registers of some peripheral modules.

2.2 Register Access

The G32R501 IPC module is not an on-chip peripheral in the usual sense; it connects to the Arm core coprocessor interface with coprocessor number 1. Therefore, IPC registers can be accessed only through coprocessor instructions provided by the Arm core.

The Arm core provides MCR/MRC or MCRR/MRRC coprocessor instructions to access the IPC module on the coprocessor interface. MCR/MRC transfers 32-bit data at a time; MCRR/MRRC can transfer 64-bit data at a time.

For details on these instructions, refer to the Cortex-M core Technical Reference Manual from Arm.

The formats of MCR and MRC are briefly described below. MCR writes data from an Arm register to a coprocessor register; MRC reads data from a coprocessor register into an Arm register.

MCR and MRC instruction formats:

```
MCR{cond} coproc, <opc1>, <Rt>, <CRn>, <CRm>, <opc2>
MRC{cond} coproc, <opc1>, <Rt>, <CRn>, <CRm>, <opc2>
```

Parameter meanings are shown in Table 1.

Table 1 MCR/MRC coprocessor instruction parameter description

Parameter	Meaning
cond	Condition code for instruction execution; may be omitted
coproc	Coprocessor number. For the IPC module this is 1
opc1	Opcode to be executed by the coprocessor
Rt	Arm core register
CRn	Coprocessor target register
CRm	Coprocessor target register. If only one is used, set to c0
opc2	Optional coprocessor-specific opcode; set to 0 when not needed

The IPC driver in the G32R5xx SDK wraps these coprocessor operations and provides interface functions, so users do not need to operate IPC registers with these instructions directly.

2.3 RAM Storage Area Shared by Dual Cores

G32R501 has three RAM types—ITCM, DTCM, and SRAM—with 7 independent RAM regions, as shown in Table 2.

Table 2 G32R501 RAM storage area types

RAM storage area type	Start address
CPU0_ITCM	0x0000_0000
CPU1_ITCM	0x0000_0000
CPU0_DTCM	0x2000_0000
CPU1_DTCM	0x2000_0000
SRAM1	0x2010_0000
SRAM2	0x2020_0000
SRAM3	0x2030_0000

Although CPU0 and CPU1 ITCM/DTCM share the same addresses, each has an independent physical space. CPU0 can access only its own ITCM/DTCM, not CPU1's; likewise, CPU1

cannot access CPU0's ITCM/DTCM.

SRAM1/2/3 is shared RAM accessible by both CPU0 and CPU1. When a data block must move from one core to the other, combining the IPC module with shared RAM is a good approach.

Typical workflow:

1. CPU0 writes the data block to the shared storage area.
2. After that, CPU0 triggers the GP interrupt of the CPU1 IPC module so CPU1 knows the data from CPU0 is ready.
3. CPU1 reads and processes the data block.
4. After Step 3, CPU1 triggers the GP interrupt of the CPU0 IPC module so CPU0 knows CPU1 has finished; CPU0 can then transfer a new data block to CPU1.

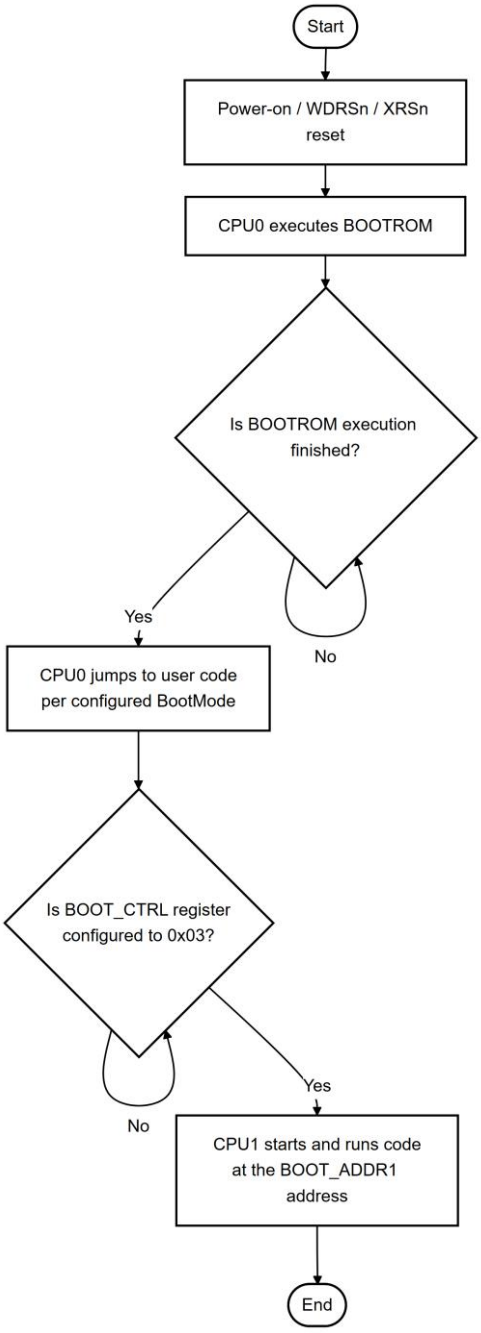
2.4 Dual-core Boot Process

G32R501 starts from master core CPU0 and then runs BOOTROM. After CPU0 starts, configure the CPU1 boot address by writing BOOT_ADDR1, and enable the CPU1 clock by writing BOOTCTRL. Slave core CPU1 then starts from the address configured in BOOT_ADDR1.

In SDK dual-core examples, "APP_bootCPU1" on CPU0 performs this sequence: it calls "SysCtl_setCPU1BootAddress" to set the CPU1 entry, then "SysCtl_enableBootCPU1" to enable CPU1 boot. Build the CPU1 image first and embed it in the CPU0 project; see IPC examples under "driverlib/g32r501/examples/eval/ipc/". For dual-core download, connection, and breakpoint debugging, see *AN1128 G32R5xx Dual-core Debug Guide*.

The dual-core boot flowchart is shown in Figure 1.

Figure 1 Dual-core boot flowchart



Basic steps to start slave core CPU1:

1. Compile the slave-core image into the master-core image, then download the resulting master-core image to G32R501.
2. The master core runs, configures the slave boot address, and enables the slave clock to start the slave core.
3. Slave core CPU1 starts executing.

On G32R501, master and slave share the same Flash. When embedding the slave image into the master image, allocate Flash space for both cores. Dual-core examples in the G32R5xx SDK split Flash evenly between the two CPUs. To change the sizes, modify the .sct linker file of the dual-core boot example.

3 IPC Module Application Examples

The IPC module supports message passing, resource sharing, and event control between cores. The G32R5xx SDK provides example code for these features; the following sections describe them.

For dual-core examples, CPU0 and CPU1 each have their own project. Always build the CPU1 project first to obtain the CPU1 image, then build that image into the CPU0 project.

3.1 Message Mailboxes

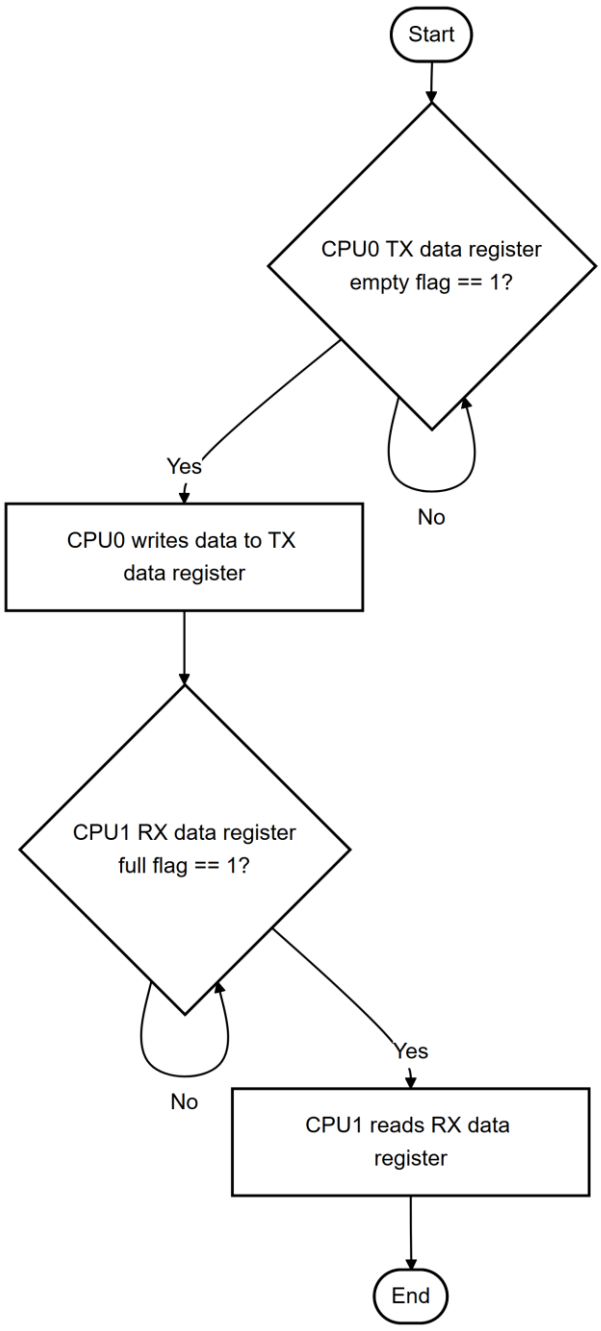
The G32R501 IPC module has four 32-bit transmit and four 32-bit receive data registers. Use them to transfer 32-bit words or to write shared-memory message-frame information (for example start address, length, and message type).

Transmit and receive can use flag polling or interrupts. The G32R5xx SDK provides examples of both.

3.1.1 Transmitting and receiving data by polling

The example driverlib\g32r501\examples\eval\ipc\ipc_ex2_mailbox_polling shows dual-core communication by polling. The flow is shown in Figure 2.

Figure 2 Dual-core communication: transmit/receive by polling



In this example, CPU0 first sends data to CPU1, and CPU1 waits by polling for data from CPU0.

Code snippet of data transmitted by CPU0 by polling:

```
//
// CPU0 send message to CPU1
//
```

```
while (IPC_isTxBufferEmpty(IPC_TX_0) != true);
IPC_transmit32Bits(IPC_TX_0, g_msgSend[0]);
while (IPC_isTxBufferEmpty(IPC_TX_1) != true);
IPC_transmit32Bits(IPC_TX_1, g_msgSend[1]);
while (IPC_isTxBufferEmpty(IPC_TX_2) != true);
IPC_transmit32Bits(IPC_TX_2, g_msgSend[2]);
while (IPC_isTxBufferEmpty(IPC_TX_3) != true);
IPC_transmit32Bits(IPC_TX_3, g_msgSend[3]);
```

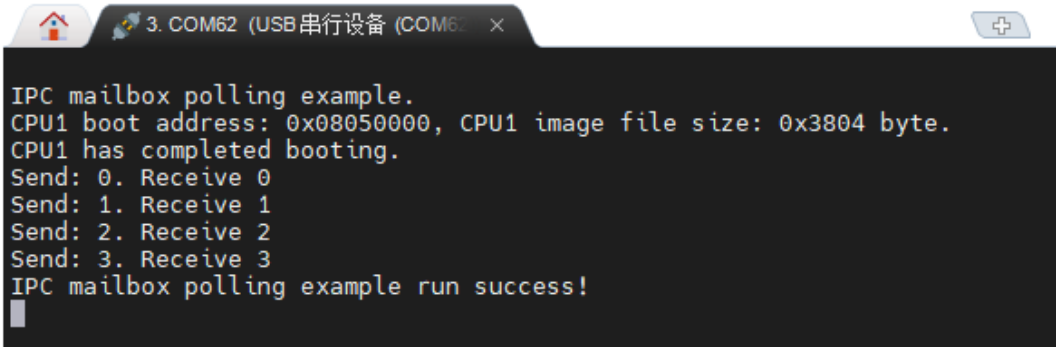
Code snippet of data received by CPU1 by polling:

```
//
// CPU1 receive message from CPU0
//
while (IPC_isRxBufferFull(IPC_RX_0) != true);
g_msgRecv[0] = IPC_receive32Bits(IPC_RX_0);
while (IPC_isRxBufferFull(IPC_RX_1) != true);
g_msgRecv[1] = IPC_receive32Bits(IPC_RX_1);
while (IPC_isRxBufferFull(IPC_RX_2) != true);
g_msgRecv[2] = IPC_receive32Bits(IPC_RX_2);
while (IPC_isRxBufferFull(IPC_RX_3) != true);
g_msgRecv[3] = IPC_receive32Bits(IPC_RX_3);
```

For dual-core examples, run as follows:

1. Build the slave-core (CPU1) project to generate the binary image.
2. Build the master-core (CPU0) project using the CPU1 binary image.
3. Download the master-core image to the G32R501 board.
4. Open a serial terminal at 115200 baud, 8 data bits, no parity, one stop bit, no flow control.
5. Press the board reset button and observe the terminal output.

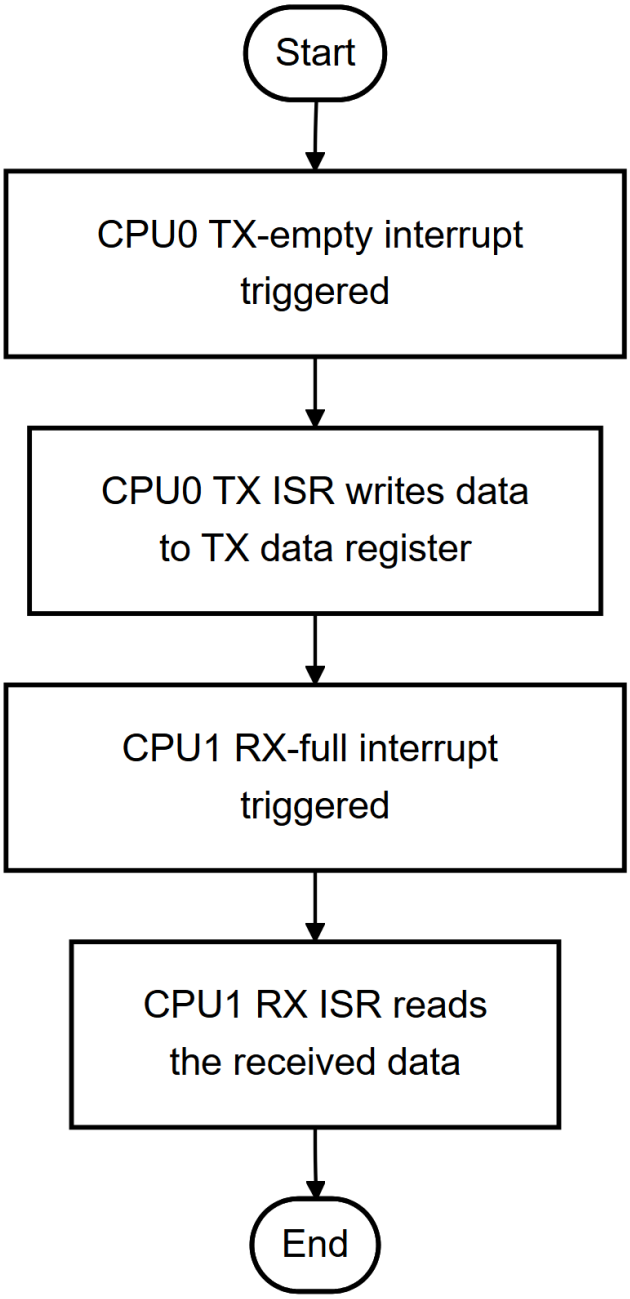
Figure 3 Polling transmit/receive example terminal output



3.1.2 Transmitting and receiving data by interrupt

Transmit and receive can also use interrupts. The interrupt flow is shown in Figure 4.

Figure 4 Dual-core communication: transmit/receive by interrupt



The example driverlib\g32r501\examples\eval\ipc\ipc_ex1_mailbox_interrupt shows interrupt-based dual-core communication.

When the CPU0 transmit-empty interrupt fires, CPU0 sends data to CPU1 in the transmit ISR:

```
//
// IPC TE0 Interrupt ISR
//
```

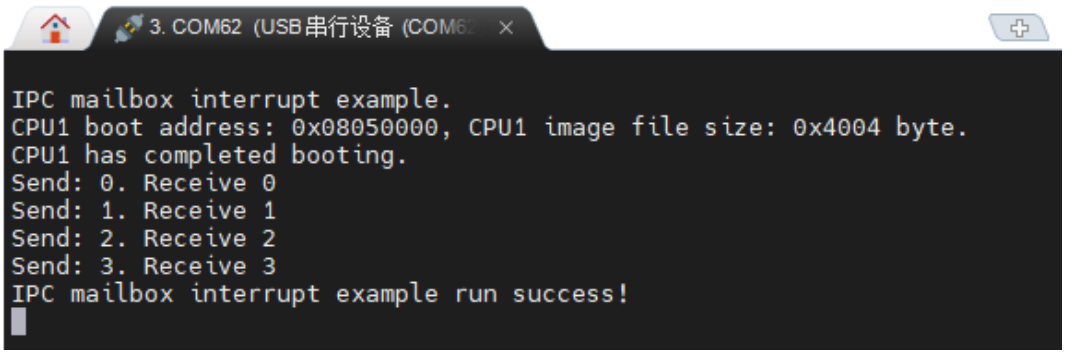
```
void INT_IPC_TE0_IRQHandler(void)
{
    if (IPC_isTxBufferEmpty(IPC_TX_0) == true)
    {
        IPC_transmit32Bits(IPC_TX_0, g_msgSend[g_curSend++]);
        IPC_disableInterrupt(IPC_INT_TE0);
    }
}
```

When the CPU1 receive-full interrupt fires, CPU1 receives data from CPU0 in the receive ISR:

```
//
// IPC RF0 Interrupt ISR
//
void INT_IPC_RF0_IRQHandler(void)
{
    if (IPC_isRxBufferFull(IPC_RX_0) == 1)
    {
        g_msgRecv[g_curRecv++] = IPC_receive32Bits(IPC_RX_0);
    }
}
```

Example result:

Figure 5 Interrupt transmit/receive example terminal output

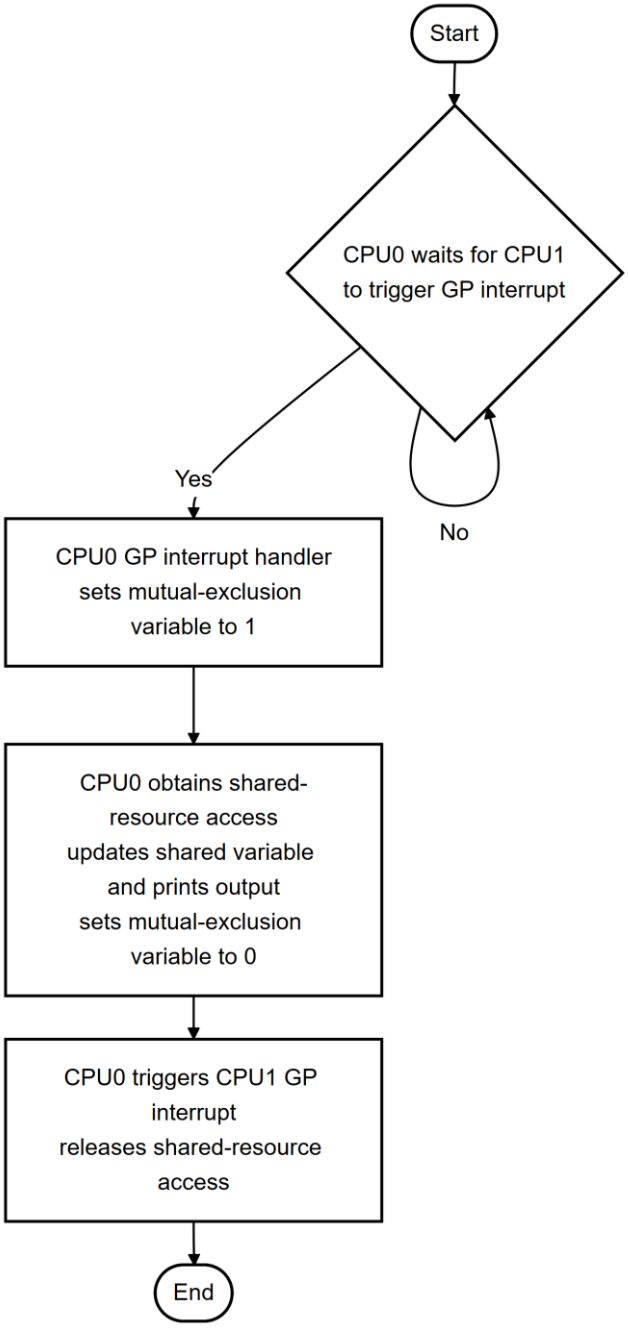


3.2 Resource Sharing

On G32R501, peripherals and shared SRAM1/2/3 can be accessed by both cores. When designing dual-core examples, avoid resource contention. If both cores must share a peripheral or the same RAM region, use the IPC GP (general-purpose) interrupt for mutual exclusion.

CPU0-side flow for GP-interrupt mutual exclusion:

Figure 6 CPU0 GP-interrupt mutual-exclusion flow



The CPU1-side flow is the same as on CPU0.

The example driverlib\g32r501\examples\eval\ipc\ipc_ex4_resource_share shows how dual cores that share a UART and update a shared variable use the IPC GP interrupt for mutual exclusion.

In the CPU0 GP ISR, set the mutual-exclusion variable g_cpu0Mutex to 1.

CPU0 GP interrupt handling code:

```
//
// IPC GP0 Interrupt ISR
//
void INT_IPC_GP0_IRQHandler(void)
{
    if (IPC_getPendingStatus() == IPC_GPFLAG_0)
    {
        IPC_clearPendingStatus(IPC_GPFLAG_0);
        g_cpu0Mutex = 1;
    }
}
```

Then both cores monitor the mutual-exclusion variable in their main loops. Only after it becomes 1 may they use the UART and update the shared variable. Main-loop outlines:

Master core (CPU0) main loop:

```
//
// Loop.
//
for(;;)
{
    //
    // Wait until CPU0 can access the shared resource
    //
    while (g_cpu0Mutex != 1);
    //
    // CPU0 can change shared variable g_shared
    //
    g_shared++;
    printf("CPU0 update shared variable to: %d\r\n", g_shared);
    g_cpu0Mutex = 0;
    //
    // CPU0 requests to trigger CPU1's GP interrupt, to notify CPU1
    // that it can access shared resources.
    //
    IPC_request(IPC_GPFLAG_0);
}
```

Slave core (CPU1) main loop:

```
//
```

```
// Loop.
//
for(;;)
{
    //
    // Wait until CPU1 can access the shared resource
    //
    while (g_cpu1Mutex != 1);
    //
    // CPU1 can change shared variable g_shared
    //
    if (g_shared != NULL)
    {
        (*g_shared)++;
        printf("CPU1 update shared variable to: %d\r\n", *g_shared);
    }
    g_cpu1Mutex = 0;
    //
    // CPU1 requests to trigger CPU0's GP interrupt, to notify CPU0
    // that it can access shared resources.
    //
    IPC_request(IPC_GPFLAG_0);
}
```

Example result:

Figure 7 Resource mutual-exclusion example terminal output

3.3 Event Control

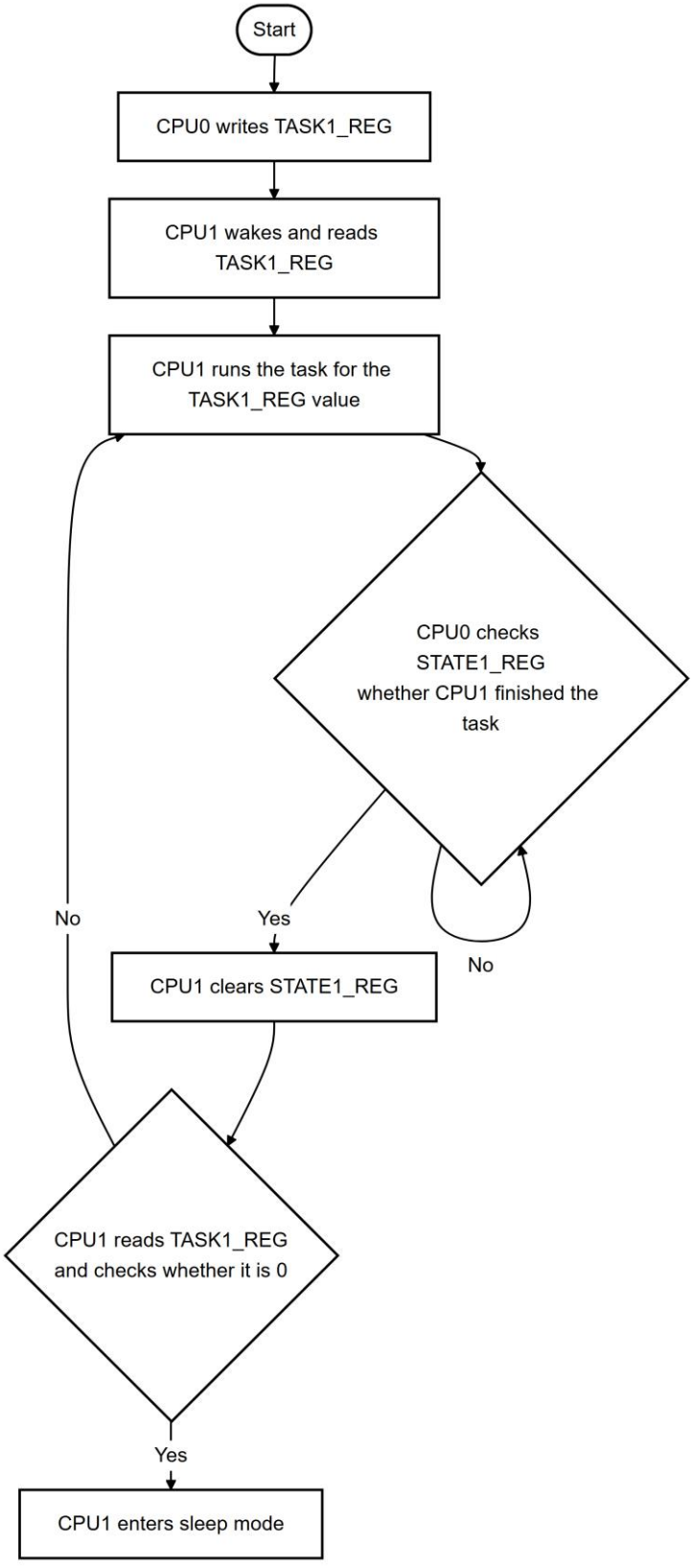
The IPC module implements event control through TASK and STATE registers. Working principle:

- CPU0 writes the TASK register; this generates a wake-up signal to notify CPU1.
- CPU1 can read the TASK register and run the corresponding task code.
- CPU0 can learn CPU1 status by reading the STATE register.

Event-control process:

1. CPU0 calls IPC_issueTask to write TASK1_REG so CPU1 can run tasks by task ID. This can also wake CPU1 from sleep.
2. After wake-up, CPU1 calls IPC_fetchTask to read TASK1_REG; hardware clears TASK1_REG.
3. CPU0 can call IPC_getTaskState to read STATE1_REG and learn CPU1 status.
4. After the task, CPU1 calls IPC_updateTaskState to write STATE1_REG to 0 and clear it.
5. CPU1 calls IPC_fetchTask again. If TASK1_REG is non-zero, repeat; otherwise enter sleep.

Figure 8 IPC event-control flowchart



The example driverlib\g32r501\examples\eval\ipc\ipc_ex5_event_control shows event control so CPU1 runs different tasks.

Example result:

Figure 9 IPC event-control example terminal output

4 Revision

Table 3 Document revision

Date	Version	Change history
2025.1	1.0	● Initial release
2026.8	1.1	● Applicable products set to G32R501Dx series dual-core products.

Statement

This document is formulated and published by Geehy Semiconductor Co., Ltd. (hereinafter referred to as “Geehy”). The contents in this document are protected by laws and regulations of trademark, copyright and software copyright. Geehy reserves the right to make corrections and modifications to this document at any time. Read this document carefully before using Geehy products. Once you use the Geehy product, it means that you (hereinafter referred to as the “users”) have known and accepted all the contents of this document. Users shall use the Geehy product in accordance with relevant laws and regulations and the requirements of this document.

1. Ownership

This document can only be used in connection with the corresponding chip products or software products provided by Geehy. Without the prior permission of Geehy, no unit or individual may copy, transcribe, modify, edit or disseminate all or part of the contents of this document for any reason or in any form.

The “极海” or “Geehy” words or graphics with “®” or “TM” in this document are trademarks of Geehy. Other product or service names displayed on Geehy products are the property of their respective owners.

2. No Intellectual Property License

Geehy owns all rights, ownership and intellectual property rights involved in this document.

Geehy shall not be deemed to grant the license or right of any intellectual property to users explicitly or implicitly due to the sale or distribution of Geehy products or this document.

If any third party’s products, services or intellectual property are involved in this document, it shall not be deemed that Geehy authorizes users to use the aforesaid third party’s products, services or intellectual property. Any information regarding the application of the product, Geehy

hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party, unless otherwise agreed in sales order or sales contract.

3. Version Update

Users can obtain the latest document of the corresponding models when ordering Geehy products.

If the contents in this document are inconsistent with Geehy products, the agreement in the sales order or the sales contract shall prevail.

4. Information Reliability

The relevant data in this document are obtained from batch test by Geehy Laboratory or cooperative third-party testing organization. However, clerical errors in correction or errors caused by differences in testing environment may occur inevitably. Therefore, users should understand that Geehy does not bear any responsibility for such errors that may occur in this document. The relevant data in this document are only used to guide users as performance parameter reference and do not constitute Geehy's guarantee for any product performance.

Users shall select appropriate Geehy products according to their own needs, and effectively verify and test the applicability of Geehy products to confirm that Geehy products meet their own needs, corresponding standards, safety or other reliability requirements. If losses are caused to users due to user's failure to fully verify and test Geehy products, Geehy will not bear any responsibility.

5. Legality

USERS SHALL ABIDE BY ALL APPLICABLE LOCAL LAWS AND REGULATIONS WHEN USING THIS DOCUMENT AND THE MATCHING GEEHY PRODUCTS. USERS SHALL UNDERSTAND THAT THE PRODUCTS MAY BE RESTRICTED BY THE EXPORT, RE-EXPORT OR OTHER LAWS OF THE COUNTRIES OF THE PRODUCTS SUPPLIERS, GEEHY, GEEHY

DISTRIBUTORS AND USERS. USERS (ON BEHALF OR ITSELF, SUBSIDIARIES AND AFFILIATED ENTERPRISES) SHALL AGREE AND PROMISE TO ABIDE BY ALL APPLICABLE LAWS AND REGULATIONS ON THE EXPORT AND RE-EXPORT OF GEEHY PRODUCTS AND/OR TECHNOLOGIES AND DIRECT PRODUCTS.

6. Disclaimer of Warranty

THIS DOCUMENT IS PROVIDED BY GEEHY "AS IS" AND THERE IS NO WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, TO THE EXTENT PERMITTED BY APPLICABLE LAW.

GEEHY'S PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED FOR USE AS CRITICAL COMPONENTS IN MILITARY, LIFE-SUPPORT, POLLUTION CONTROL, OR HAZARDOUS SUBSTANCES MANAGEMENT SYSTEMS, NOR WHERE FAILURE COULD RESULT IN INJURY, DEATH, PROPERTY OR ENVIRONMENTAL DAMAGE.

IF THE PRODUCT IS NOT LABELED AS "AUTOMOTIVE GRADE," IT SHOULD NOT BE CONSIDERED SUITABLE FOR AUTOMOTIVE APPLICATIONS. GEEHY ASSUMES NO LIABILITY FOR THE USE BEYOND ITS SPECIFICATIONS OR GUIDELINES.

THE USER SHOULD ENSURE THAT THE APPLICATION OF THE PRODUCTS COMPLIES WITH ALL RELEVANT STANDARDS, INCLUDING BUT NOT LIMITED TO SAFETY, INFORMATION SECURITY, AND ENVIRONMENTAL REQUIREMENTS. THE USER ASSUMES FULL RESPONSIBILITY FOR THE SELECTION AND USE OF GEEHY PRODUCTS. GEEHY WILL BEAR NO RESPONSIBILITY FOR ANY DISPUTES ARISING FROM THE SUBSEQUENT DESIGN OR USE BY USERS.

7. Limitation of Liability

IN NO EVENT, UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL GEEHY OR ANY OTHER PARTY WHO PROVIDES THE DOCUMENT AND PRODUCTS

"AS IS", BE LIABLE FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, DIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE DOCUMENT AND PRODUCTS (INCLUDING BUT NOT LIMITED TO LOSSES OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY USERS OR THIRD PARTIES). THIS COVERS POTENTIAL DAMAGES TO PERSONAL SAFETY, PROPERTY, OR THE ENVIRONMENT, FOR WHICH GEEHY WILL NOT BE RESPONSIBLE.

8. Scope of Application

The information in this document replaces the information provided in all previous versions of the document.