Application Note

Document No.: AN1135

G32R501 IPC Application Note

Version: V1.0

1 Introduction

This document mainly introduces relevant content of G32R5xx IPC module. The specific content includes basic introduction to IPC module, register access methods, shared RAM storage area, and dual-core boot process. The document also provides application examples of the IPC module, including the implementation methods of dual-core communication and resource sharing using message mailbox, and the event control use process of the IPC module.

For detailed introduction to the IPC module, please refer to the *User Manual of G32R5xx*.

Contents

2 G32R5xx IPC Module Introduction

2.1 Overview

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

In dual-core operating mode, these two cores need to communicate with each other. G32R501 provides an IPC (Inter-Processor Communication) module to enable communication between dual cores.

The G32R501 IPC module includes three functions:

1. Interrupt control and data sharing (mailbox):
   - Data is shared through 4 registers, and 12 interrupts are generated for each CPU.
   - Exchange data through the shared memory.
2. Event control
   - Communicate between two CPU using polling method and RXEV.
3. WRPRT
   - Implement the WRPRT function. The write register protection function can prevent CPU from performing incorrect write operations on the registers of some peripheral modules.

2.2 Register Access

The IPC module of G32R501 does not exist as an on-chip peripheral module, but is connected to the coprocessor interface of the Arm core, and the coprocessor number is 1. Therefore, access to IPC module registers can only be read and written through the coprocessor instructions provided by the Arm core.

The Arm core provides MCR/MRC or MCRR/MRRC coprocessor instructions for accessing the IPC module located on the coprocessor interface. The MCR/MRC instruction can only transmit 32-bit data at a time, while the MCRR/MRRC instruction can transmit 64-bit data at a time.

For detailed information on these coprocessor instructions, please refer to the *Cortex-M Core Technical Reference Manual* officially provided by Arm.

The formats of MCR and MRC instructions are briefly introduced below. The MCR instruction is used to write data from the Arm register to the coprocessor register, while the MRC instruction is used to read data from the coprocessor register to the Arm register.

Formats of MCR and MRC instructions:

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

MRR{cond} coproc, <opc1>, <Rt>, <CRn>, <CRm>, <opc2>

The meanings of each parameter are as follows:

Table 1 Interpretation to MCR/MRC Coprocessor Instruction Parameters

Command parameter	**Meaning**
cond	The condition code for instruction execution, which can be ignored
coproc	Coprocessor number. The coprocessor number of the IPC module is 1
opc1	The opcode to be executed by the coprocessor
Rt	ARM core register
CRn	The target register of the coprocessor
CRm	The target register of the coprocessor. If only one is used, set it to c0
opc2	Optional coprocessor specific opcode, set to 0 if not needed

The IPC module driver code in G32R5xx SDK has packaged the relevant coprocessor instruction operations and provided corresponding interface functions for users to call. Users do not need to directly use these complex instructions to operate the registers of the IPC module.

2.3 RAM Storage Area Shared by Dual Cores

G32R501 has three types of RAM storage areas, namely, ITCM, DTCM, and SRAM, with a total of 7 isolated RAM storage areas, as shown in the table below:

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 the ITCM and DTCM storage areas of CPU0 and CPU1 have the same address, they have isolated physical storage space. In addition, CPU0 can only access its own ITCM and DTCM, and cannot access the ITCM and DTCM of CPU1; similarly, CPU1 can only access its own ITCM and DTCM, and cannot access the TCM and DTCM of CPU0.

SRAM1/2/3 is a shared RAM storage area that both CPU0 and CPU1 can access. If a data block needs to be transferred from one core to the other, using the IPC module combined with the shared RAM storage area is a good choice.

The typical workflow is as follows:

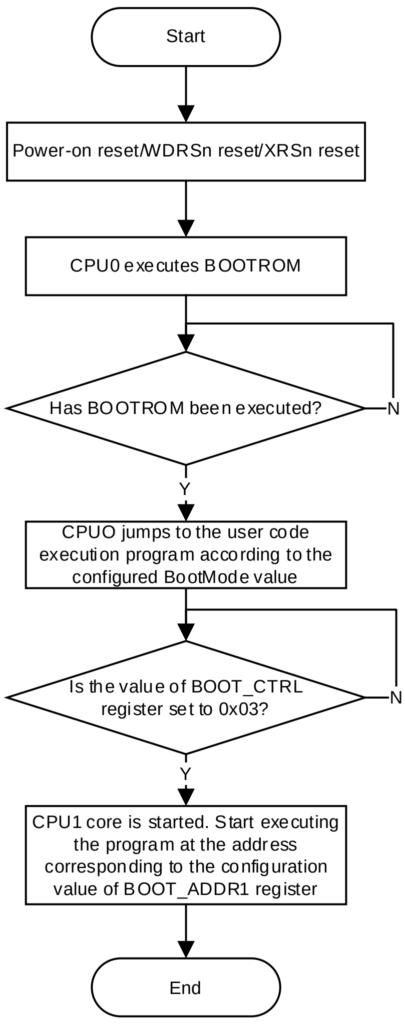
1. CPU0 writes the data block to the shared storage area.
2. After that, CPU0 triggers the GP interrupt of CPU1 IPC module, and CPU1 will know that the data block from CPU0 has been ready.
3. CPU1 reads the data block and processes it.
4. After Step 3 is completed, CPU1 triggers the GP interrupt of CPU0 IPC module, and CPU0 will know that CPU1 has processed the data block, and then CPU0 can continue to transfer new data blocks to CPU0.

2.4 Dual-core Boot Process

G32R501 first starts from the master core CPU0 and then executes BOOTROM. After CPU0 is started, the boot address of CPU1 is configured by writing to the BOOT_ADDR1 register, and the clock of CPU1 is enabled by writing to the BOOTCTRL register. In this way, the slave core CPU1 can start executing programs from the address configured in the BOOT_ADDR1 register.

The flowchart of the dual-core boot process is as follows:

Figure 1 Dual-core Boot Flowchart


Start
Power-on reset/WDRSn reset/XRSn reset
CPU0 executes BOOTROM
Has BOOTROM been executed?
N
Y
CPUO jumps to the user code execution program according to the configured BootMode value
Is the value of BOOT_CTRL register set to 0x03?
N
Y
CPU1 core is started. Start executing the program at the address corresponding to the configuration value of BOOT_ADDR1 register
End

The basic steps for booting the slave core CPU1 are:

1. Compile the image file of the slave core into the image file of the master core, and then download the obtained image file of the master core to G32R501.

2. The master core executes the program, configures the boot address of the slave core, and enables the clock of the slave core to boot the slave core.

3. The slave core CPU1 starts executing the program.

In G32R501, the master core and slave core share the same Flash storage space, so when compiling the image file from the slave core to the master core, it is necessary to consider the allocation of Flash space occupied by these two cores. The dual-core routine provided in the G32R5xx SDK evenly divides the size of Flash to the two CPU cores for use. If users want to modify the size of Flash space occupied by the master core and slave core, they can do so by modifying the .sct link file of the dual-core startup routine.

3 IPC Module Application Examples

The IPC module can implement such functions as message passing, resource sharing, and event control between dual cores. In the G32R5xx SDK, the application example code of these features has been provided, and users can refer to the example code in the SDK. The example code of these functions is introduced in detail as follows.

For the dual-core routines, the master core (CPU0) and slave core (CPU1) have their own engineering project. When running these examples, it is necessary to first compile the engineering of CPU1 to obtain the image file of CPU1, and then compile the image file into the engineering of CPU0.

3.1 Message Mailboxes

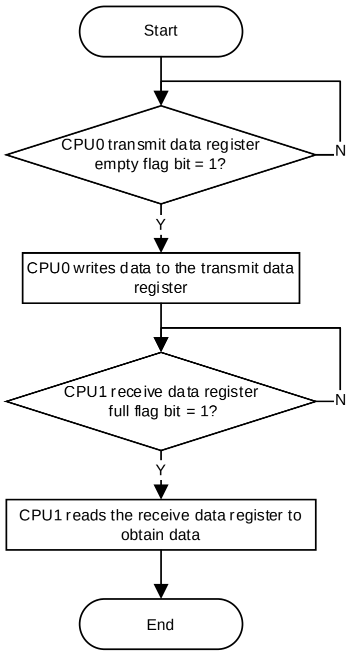
The G32R501 IPC module has four 32-bit word transmit and receive data registers, and users can use these registers to transmit 32-bit word data or write message frame information (e.g. initial address, data length, and message type code) in the shared memory.

Transmitting and receiving data can be done by polling the flag bit or by interrupts. The G32R5xx SDK provides examples programs of these two methods for users' reference to transmit and receive data.

3.1.1 Transmitting and receiving data by polling

The driverlib\g32r501\examples\eval\ipc\ipc_ex2_mailbox_polling routine provided in the SDK demonstrates how to implement communication between dual cores by polling. The process of transmitting and receiving data by polling is as follows:

Figure 2 Flowchart of Transmitting and Receiving Data in Dual-core Communication by Polling


Start
CPU0 transmit data register empty flag bit = 1?
N
Y
CPU0 writes data to the transmit data register
CPU1 receive data register full flag bit = 1?
N
Y
CPU1 reads the receive data register to obtain data
End

In this example, CPU0 first transmits data to CPU1, and then CPU1 waits to receive the data transmitted by CPU0 through polling.

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]);
```

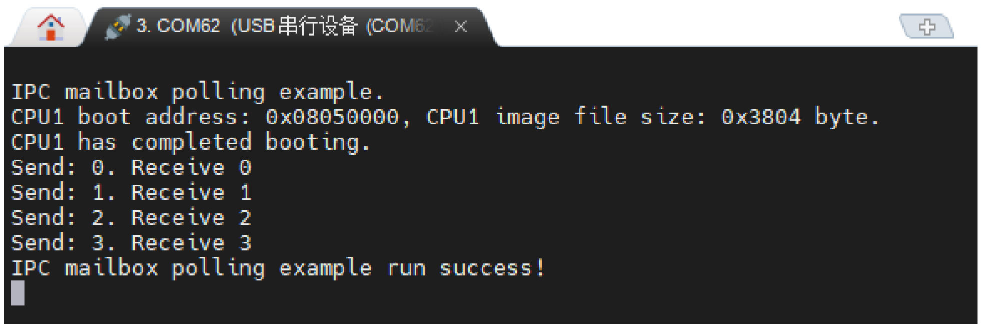
Code snippet of data received by cpu1 by polling:

```
//
// 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 routines, this example code can run according to the following steps:

1. Compile the engineering of the slave core (CPU1) to generate the binary image file.
2. Compile the engineering of the master core (CPU0) using the binary image file of the CPU1.
3. Download the obtained master core image file to the G32R501 development board.
4. Open the serial port terminal tool according to the configuration of 115200 baud rate + 8 data bits + no parity check + one stop bit + no flow control.
5. Press the reset button on the development board and observe the print information on the serial port terminal.

Figure 3 Output Results of An Example of Transmitting and Receiving Data by Polling on Serial Port Terminal

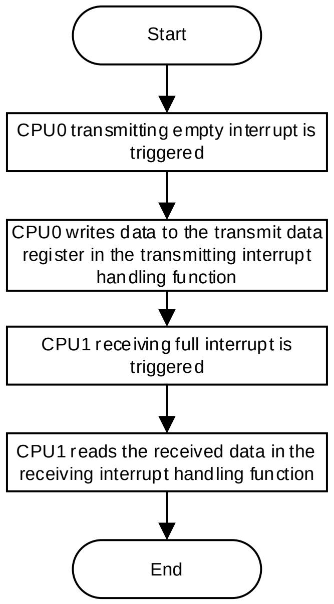


```
IPC mailbox polling example.
CPU1 boot address: 0x08050000, CPU1 image file size: 0x3804 byte.
CPU1 has completed booting.
Send: 0. Receive 0
Send: 1. Receive 1
Send: 2. Receive 2
Send: 3. Receive 3
IPC mailbox polling example run success!
```

3.1.2 Transmitting and receiving data by interrupt

In dual-core communication, transmitting and receiving data can also be implemented by interrupt. The process of transmitting and receiving data by interrupt is as follows:

Figure 4 Flowchart of Transmitting and Receiving Data in Dual-core Communication by Interrupt


Start
CPU0 transmitting empty interrupt is triggered
CPU0 writes data to the transmit data register in the transmitting interrupt handling function
CPU1 receiving full interrupt is triggered
CPU1 reads the received data in the receiving interrupt handling function
End

The driverlib\g32r501\examples\eval\ipc\ipc_ex1_mailbox_interrupt routine provided in the SDK demonstrates how to implement communication between dual cores by interrupt.

When the transmission of CPU0 is triggered as an empty interrupt, CPU0 will transmit data to CPU1 in transmitting interrupt handler, and the code is as follows:

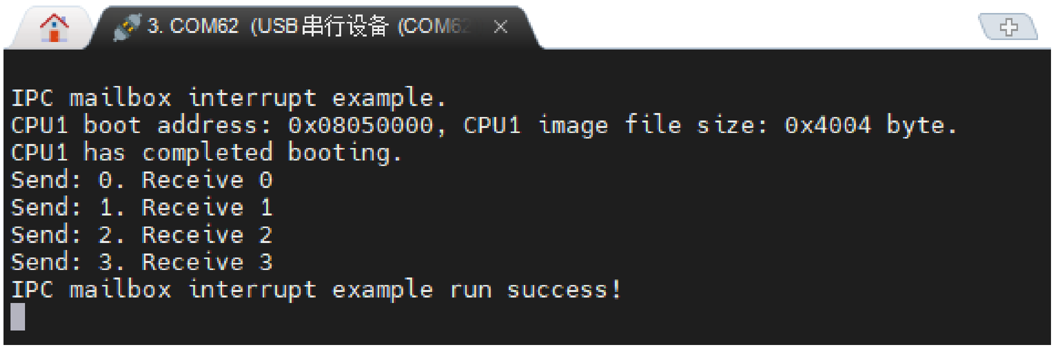
```
//
// IPC TE0 Interrupt ISR
//
void INT_IPC_TE0_IRQHandler(void)
{
    if (IPC_isTxBufferEmpty(IPC_TX_0) == true)
    {
        IPC_transmit32Bits(IPC_TX_0, g_msgSend[g_curSend++]);
        IPC_disableInterrupt(IPC_INT_TE0);
    }
}
```

When the receiving of CPU1 is triggered as a full interrupt, CPU1 will receive data from CPU0 in receiving interrupt handler, and the code is as follows:

```
//
// 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);
    }
}
```

The running result of this example is as follows:

Figure 5 Output Results of An Example of Transmitting and Receiving Data by Interrupt on Serial Port Terminal



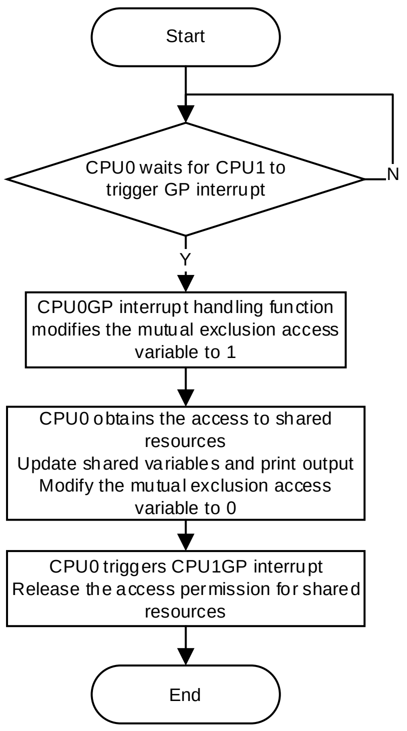
```
IPC mailbox interrupt example.
CPU1 boot address: 0x08050000, CPU1 image file size: 0x4004 byte.
CPU1 has completed booting.
Send: 0. Receive 0
Send: 1. Receive 1
Send: 2. Receive 2
Send: 3. Receive 3
IPC mailbox interrupt example run success!
```

3.2 Resource Sharing

On G32R501, both the peripheral modules and shared SRAM1/2/3 storage areas can be accessed by both the master and slave cores. Therefore, when designing the dual-core routines, special attention shall be paid to it to avoid resource competition. If it is necessary for dual cores to share the same peripheral or access the same RAM storage area, the GP (general-purpose) interrupt in the IPC module can be used to achieve mutual exclusion access.

When GP interrupt is used to achieve mutual exclusion access to resources, the flowchart on the CPU0 side of the master core is shown below:

Figure 6 Flowchart of CPU0 Using GP Interrupt to Achieve Mutual Exclusion Access


Start
CPU0 waits for CPU1 to trigger GP interrupt
N
Y
CPU0GP interrupt handling function modifies the mutual exclusion access variable to 1
CPU0 obtains the access to shared resources
Update shared variables and print output
Modify the mutual exclusion access variable to 0
CPU0 triggers CPU1GP interrupt
Release the access permission for shared resources
End

The mutual exclusion access process of resources on the CPU1 side is the same as that on the CPU0 side.

The driverlib\g32r501\examples\eval\ipc\ipc_ex4_resource_share routine provided in the G32R5xx SDK demonstrates how to use the GP interrupt of the IPC module to achieve mutual exclusion access to resources when dual cores use UART serial peripherals and update shared variables simultaneously.

In the CPU0 GP interrupt handler, assign the variable g_cpu0Mutex used for mutual exclusion access 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, the two cores will continuously monitor and control the mutual exclusion access variable in their respective main loop code. Only when the variable is modified to 1, can they use UART peripherals and update the shared variables. The main loop code of the two cores is as follows.

Main loop process of master core (CPU0):

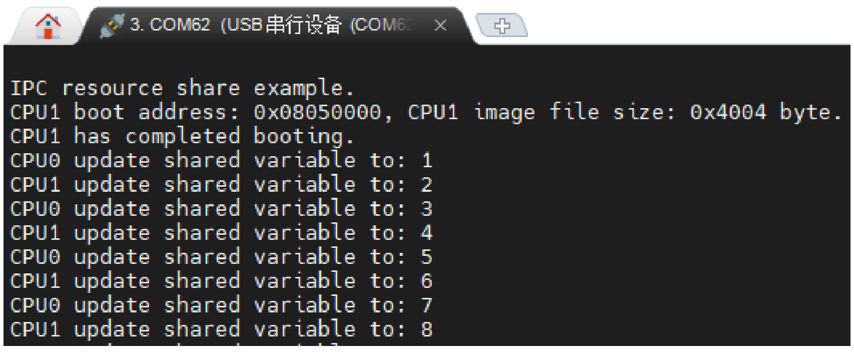
```
//
// Loop.
//
for(;;)
{
    //
    // Wait until CPU0 can access the shared resource
    //
    while (g_cpu0Mutex != 1);

    //
    // 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);
}
```

Main loop process of slave core (CPU1):

```
//
// 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);
}
```

The running result of this example is as follows:

Figure 7 Output Results of An Example of Mutual Exclusion Access of Resources on Serial Port Terminal



```
IPC resource share example.
CPU1 boot address: 0x08050000, CPU1 image file size: 0x4004 byte.
CPU1 has completed booting.
CPU0 update shared variable to: 1
CPU1 update shared variable to: 2
CPU0 update shared variable to: 3
CPU1 update shared variable to: 4
CPU0 update shared variable to: 5
CPU1 update shared variable to: 6
CPU0 update shared variable to: 7
CPU1 update shared variable to: 8
```

3.3 Event Control

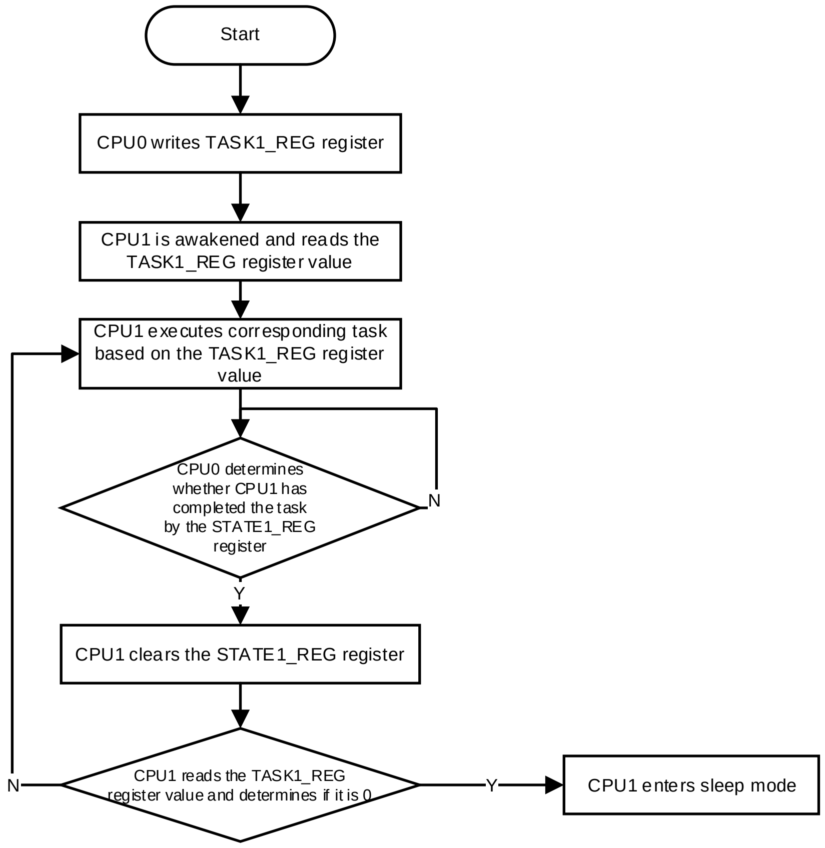
The IPC module can implement the event control mechanism through TASK and STATE registers. The working principle of this mechanism is:

- CPU0 writes data to the TASK register, and this operation will generate a wake-up signal to notify CPU1;
- CPU1 can read data from the TASK register and execute corresponding task codes according to the value of the register.
- CPU0 can learn the execution status of CPU1 by reading the value of the STATE register.

The event control process is as follows:

1. CPU0 writes data to the TASK1_REG register by calling the IPC_issueTask function, so that CPU1 can execute corresponding tasks based on different data (task ID). Besides, this operation can wake up CPU1 in sleep mode.
2. After CPU1 is awakened, it obtains the value of the TASK1_REG register through the IPC_fetchTas function, and this operation will cause the hardware to clear the value of the TASK1_REG register.
3. CPU0 can obtain the value of the STATE1_REG register through the IPC_getTaskState function to learn the execution status of CPU1.
4. After CPU1 completes the corresponding task, it writes the value of the STATE1_REG register to 0 through the IPC_updateTaskState function to clear its content of this register.
5. CPU1 obtains the value of the TASK1_REG register again through the IPC_fetchTask function. If the value of the register is not zero, CPU1 will repeat the above steps; otherwise, CPU1 will enter sleep mode.

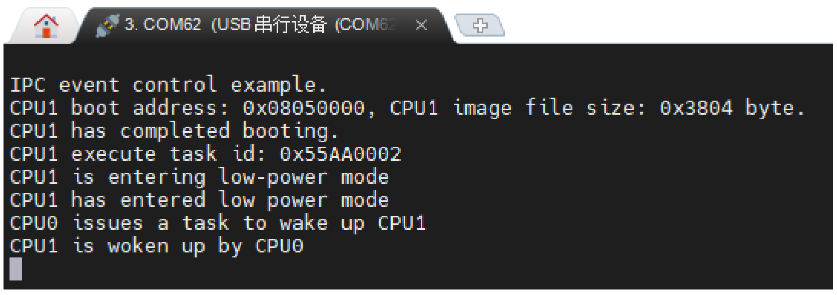
Figure 8 Event Control Flowchart of IPC Module


Start
CPU0 writes TASK1_REG register
CPU1 is awakened and reads the TASK1_REG register value
CPU1 executes corresponding task based on the TASK1_REG register value
CPU0 determines whether CPU1 has completed the task by the STATE1_REG register
N
Y
CPU1 clears the STATE1_REG register
CPU1 reads the TASK1_REG register value and determines if it is 0
N
Y
CPU1 enters sleep mode

The driverlib\g32r501\examples\eval\ipc\ipc_ex5_event_control routine provided in G32R5xx SDK demonstrates how to use the event control mechanism of the IPC module for CPU1 to execute different tasks.

The running result of this example is as follows:

Figure 9 Output Results of IPC Event Control Routine on Serial Port Terminal



```
IPC event control example.
CPU1 boot address: 0x08050000, CPU1 image file size: 0x3804 byte.
CPU1 has completed booting.
CPU1 execute task id: 0x55AA0002
CPU1 is entering low-power mode
CPU1 has entered low power mode
CPU0 issues a task to wake up CPU1
CPU1 is woken up by CPU0
```

4 Revision

Table 3 Document Revision History

Date	Version	Change History
January, 2025	1.0	New

Statement

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.