

# 应用笔记

## Application Note

文档编号: **AN1135**

**G32R501 IPC 应用笔记**

版本: **V1.0**

# 1 引言

本文档主要介绍 **G32R5xx IPC** 模块的相关内容。具体内容包括 **IPC** 模块基本介绍、寄存器访问方式、共享的 **RAM** 存储区以及双核的启动过程。文中还提供了 **IPC** 模块的应用示例，包括使用消息邮箱实现双核通信、资源共享的实现方法，以及 **IPC** 模块的事件控制使用流程。

关于 **IPC** 模块的详细介绍，可参考 **G32R5xx** 的用户手册。

## 目录

|          |                              |           |
|----------|------------------------------|-----------|
| <b>1</b> | <b>引言 .....</b>              | <b>1</b>  |
| <b>2</b> | <b>G32R5xx IPC 模块介绍.....</b> | <b>3</b>  |
| 2.1      | 概述.....                      | 3         |
| 2.2      | 寄存器访问.....                   | 3         |
| 2.3      | 双核共享的 RAM 存储区 .....          | 4         |
| 2.4      | 双核启动过程.....                  | 5         |
| <b>3</b> | <b>IPC 模块应用示例.....</b>       | <b>7</b>  |
| 3.1      | 消息邮箱.....                    | 7         |
| 3.2      | 资源共享.....                    | 11        |
| 3.3      | 事件控制.....                    | 15        |
| <b>4</b> | <b>版本历史.....</b>             | <b>17</b> |

## 2 G32R5xx IPC 模块介绍

### 2.1 概述

G32R501 是一款同构多核非对称架构的双核微控制器芯片，内部集成了两颗 Arm® Cortex®-M52 内核，其中 CPU0 为主核，CPU1 为从核。

在双核运行模式下，两个内核需要相互进行通信。G32R501 提供了 IPC（处理器间通信）模块，以实现双核之间的通信。

其中，G32R501 IPC 模块包含三个部分功能：

1. 中断控制和数据共享（邮箱）：

- 通过 4 个寄存器共享数据，并可以为每个 CPU 生成 12 个中断。
- 通过共享内存交换数据。

2. 事件控制

- 使用轮询方法和 RXEV 在两个 CPU 之间进行通信。

3. WRPRT

- 实现了 WRPRT 功能。写寄存器保护功能可以防止 CPU 对一些外设模块的寄存器进行错误的写操作。

### 2.2 寄存器访问

G32R501 的 IPC 模块并非作为一个片上外设模块存在，而是与 Arm 内核的协处理器接口相连，协处理器编号为 1。因此，对 IPC 模块寄存器的访问，只能通过 Arm 内核提供的协处理指令进行读写操作。

Arm 内核提供了 MCR/MRC 或 MCRR/MRRC 协处理指令，用于访问位于协处理接口上的 IPC 模块。其中，MCR/MRC 指令一次仅能传输 32 位数据，而 MCRR/MRRC 指令则可以一次传输 64 位数据。

有关这些协处理指令的详细信息，请参考 Arm 官方提供的 Cortex-M 内核技术参考手册。

以下简要说明 MCR 和 MRC 指令的格式。MCR 指令用于将 Arm 寄存器中的数据写入协处理器寄存器，而 MRC 指令则用于将协处理器寄存器中的数据读取到 Arm 寄存器中。

MCR 与 MRC 指令格式：

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

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

其中各个参数含义如下表所示：

表格 1 MCR/MRC 协处理指令参数解释

| 指令参数   | 含义                        |
|--------|---------------------------|
| cond   | 指令执行的条件码，可忽略              |
| coproc | 协处理器编号。IPC 模块是协处理器编号是 1   |
| opc1   | 协处理器要执行的操作码               |
| Rt     | ARM 内核寄存器                 |
| CRn    | 协处理器的目标寄存器                |
| CRm    | 协处理器的目标寄存器。如果只用到一个则设置为 c0 |
| opc2   | 可选的协处理器特定操作码，当不需要时设置为 0   |

G32R5xx SDK 中的 IPC 模块驱动代码已经对相关的协处理指令操作进行了封装，并提供了相应的接口函数供用户调用，用户无需直接使用这些复杂的指令来操作 IPC 模块的寄存器。

## 2.3 双核共享的 RAM 存储区

G32R501 有 ITCM、DTCM、SRAM 三种 RAM 存储区类型，一共有 7 块独立的 RAM 存储区，如下表所示：

表格 2 G32R501 RAM 存储区类型

| RAM 存储区类型 | 起始地址        |
|-----------|-------------|
| 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 |

尽管 CPU0 和 CPU1 的 ITCM 和 DTCM 存储区具有相同的地址，但它们各自拥有独立的物理存储空间。此外，CPU0 只能访问其自身的 ITCM 和 DTCM，无法访问 CPU1 的 ITCM 和 DTCM；同样，CPU1 也只能访问其自身的 ITCM 和 DTCM，而无法访问 CPU0 的 ITCM 和 DTCM。

SRAM1/2/3 则是 CPU0 和 CPU1 都可以访问的共享 RAM 存储区，如果有数据块需要从一个核传递到另一个核，那么通过 IPC 模块再结合共享的 RAM 存储区是一个很好的选择。

典型的工作流程如下:

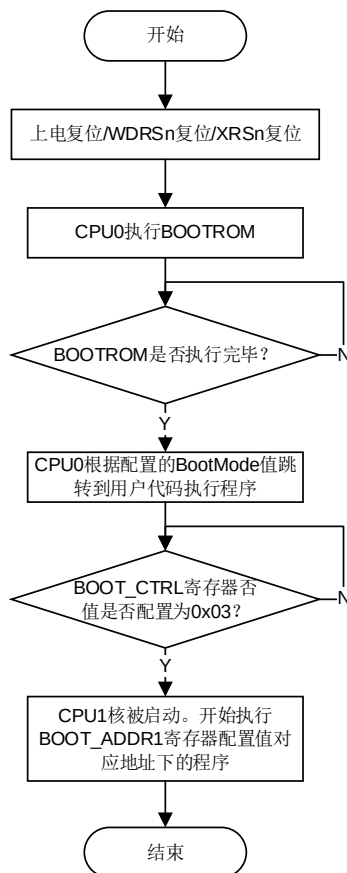
1. CPU0 将数据块写入共享存储区。
2. 完成之后, CPU0 触发 CPU1 IPC 模块的 GP 中断, 那么 CPU1 就会知道来自 CPU0 的数据块已经准备就绪。
3. CPU1 读取数据块, 并进行处理。
4. 步骤 3 完成之后, CPU1 触发 CPU0 IPC 模块的 GP 中断, 那么 CPU0 就会知道 CPU1 已经处理完该数据块, 然后 CPU0 可以接着继续传递新的数据块给 CPU0。

## 2.4 双核启动过程

G32R501 首先从主核 CPU0 启动, 然后执行 BOOTROM。在 CPU0 启动后, 通过写入 BOOT\_ADDR1 寄存器配置 CPU1 的启动地址, 并通过写入 BOOTCTRL 寄存器以开启 CPU1 的时钟。这样, 从核 CPU1 就可以从 BOOT\_ADDR1 寄存器配置的地址处开始执行程序。

双核启动过程的流程图如下:

图 1 双核启动流程图



启动 CPU1 从核的基本步骤是:

1. 将从核的映像文件编译到主核的映像文件中, 然后把得到的主核映像文件下载到 G32R501。

2. 主核执行程序，配置从核的启动地址并使能从核的时钟，从而启动从核。
3. CPU1 从核开始执行程序。

在 G32R501 中，主核和从核共用同一片 Flash 存储空间，因此将从核的映像文件编译到主核的映像文件中时，需要考虑两个核所占用的 Flash 空间分配问题。在 G32R5xx SDK 中所提供的双核例程，将 Flash 的大小均等划分给两个 CPU 核使用。如果用户希望修改主核和从核所占用的 Flash 空间大小，可以通过修改双核启动例程的.sct 链接文件来实现。

### 3 IPC 模块应用示例

IPC 模块能够实现双核之间的消息传递、资源共享以及事件控制等功能。在 G32R5xx SDK 中，已经提供了这些功能的应用示例代码，用户可以参考 SDK 中的示例代码。以下将对这些功能的示例代码进行详细介绍。

对于双核例程，主核（CPU0）和从核（CPU1）分别具有各自的工程项目。在运行这些示例时，必须首先编译 CPU1 的工程，以获取 CPU1 的映像文件，然后再将该映像文件编译进 CPU0 的工程中。

#### 3.1 消息邮箱

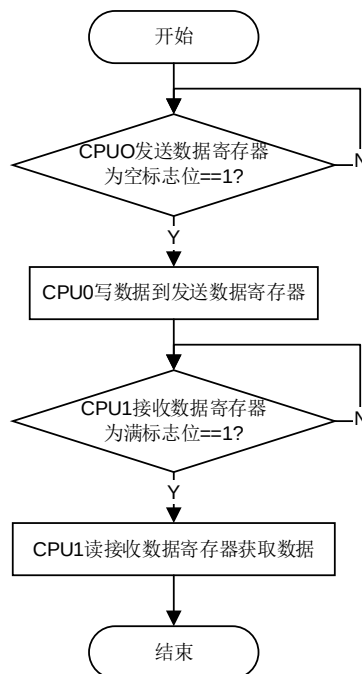
G32R501 IPC 模块分别有 4 个 32 位字的发送和接收数据寄存器，用户可以使用这些寄存器来传输 32 位字数据或写入共享内存的消息帧信息（比如初始地址、数据长度和消息类型码等）。

发送和接收数据，可以采用轮询标志位的方式或中断方式。G32R5xx SDK 提供了这两种方式的示例程序供用户参考，以实现数据的发送和接收。

##### 3.1.1 使用轮训方式发送接收数据

在 SDK 中提供的 `driverlib\g32r501\examples\eval\ipc\ipc_ex2_mailbox_polling` 例程中，展示了如何使用轮训方式在双核之间进行通信。轮训方式发送和接收数据流程如下：

图 2 双核通信轮训方式发送接收数据流程图



在该示例中，CPU0 首先将数据发送到 CPU1，CPU1 则通过轮询的方式等待接收来自 CPU0 发送的数据。



cpu0 轮训发送数据的代码片段:

```
//  
// CPU0 send message to CPU1  
//  
while (IPC_isTxBufEmpty(IPC_TX_0) != true);  
IPC_transmit32Bits(IPC_TX_0, g_msgSend[0]);  
  
while (IPC_isTxBufEmpty(IPC_TX_1) != true);  
IPC_transmit32Bits(IPC_TX_1, g_msgSend[1]);  
  
while (IPC_isTxBufEmpty(IPC_TX_2) != true);  
IPC_transmit32Bits(IPC_TX_2, g_msgSend[2]);  
  
while (IPC_isTxBufEmpty(IPC_TX_3) != true);  
IPC_transmit32Bits(IPC_TX_3, g_msgSend[3]);
```

cpu1 轮训接收数据的代码片段:

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

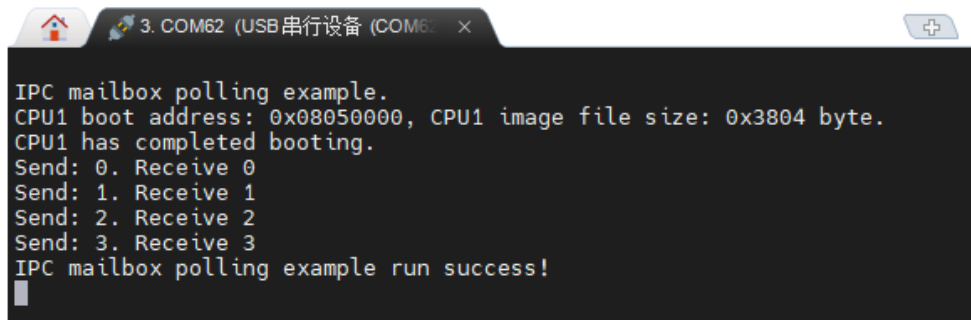
对于双核例程, 都可以按照以下步骤运行该示例代码:

1. 编译从核 (CPU1) 的工程以生成二进制映像文件。

[www.geehy.com](http://www.geehy.com)

2. 使用 CPU1 的二进制映像文件，编译主核（CPU0）的工程。
3. 将得到的主核映像文件下载到 G32R501 开发板上。
4. 按照 115200 波特率+ 8 个数据位+无奇偶校验+一个停止位+无流控制的配置，打开串口终端工具。
5. 按下开发板上的复位按键，观察串口终端的打印信息。

图 3 轮训方式发送接收数据示例在串口终端上的输出结果



```

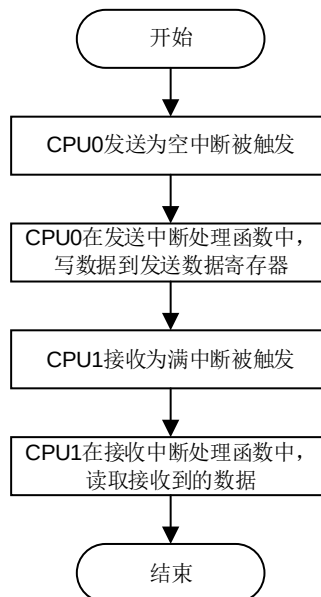
IPC mailbox polling example.
CPU1 boot address: 0x08050000, CPU1 image file size: 0x3804 byte.
CPU1 has completed booting.
Send: 0. Receive 0
Send: 1. Receive 1
Send: 2. Receive 2
Send: 3. Receive 3
IPC mailbox polling example run success!

```

### 3.1.2 使用中断方式发送接收数据

双核通信中，发送接收数据也可以通过中断方式实现。中断方式发送和接收数据流程如下：

图 4 双核通信中断方式发送接收数据流程图



在 SDK 中提供的 `driverlib\g32r501\examples\eval\ipc\ipc_ex1_mailbox_interrupt` 例程中，展示了如何使用中断方式在双核之间进行通信。

当 CPU0 的发送为空中断被触发时，CPU0 将在发送中断处理函数中向 CPU1 发送数据，代码如下：

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

当 CPU1 的接收为满中断被触发时，CPU1 将在接收中断处理函数中接收来自 CPU0 发送的数据，代码如下：

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

该示例运行结果如下:

图 5 中断方式发送接收数据示例在串口终端上的输出结果

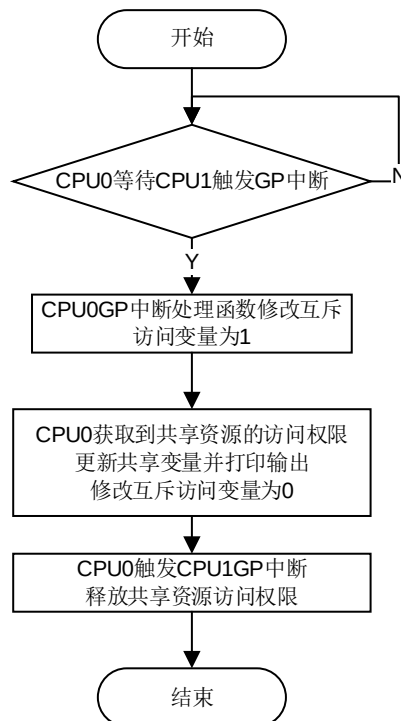
```
IPC mailbox interrupt example.
CPU1 boot address: 0x08050000, CPU1 image file size: 0x4004 byte.
CPU1 has completed booting.
Send: 0. Receive 0
Send: 1. Receive 1
Send: 2. Receive 2
Send: 3. Receive 3
IPC mailbox interrupt example run success!
```

## 3.2 资源共享

在 G32R501 上, 外设模块和共享的 SRAM1/2/3 存储区均可被主核和从核访问。因此, 在设计双核例程时, 需要特别注意避免出现资源竞争的情况。如果确实需要双核共同使用同一个外设或访问同一块 RAM 存储区, 可以利用 IPC 模块中的 GP (通用) 中断来实现互斥访问。

使用 GP 中断实现资源的互斥访问, 主核 CPU0 侧的流程图:

图 6 CPU0 使用 GP 中断实现资源互斥访问流程图



对于 CPU1 侧的资源互斥访问流程, 与 CPU0 侧是一样的。

在 G32R5xx SDK 中提供的 driverlib\g32r501\examples\eval\ipc\ipc\_ex4\_resource\_share 例程

中，展示了在双核同时使用 **UART** 串口外设并更新共享变量的情况下，如何使用 **IPC** 模块的 **GP** 中断实现对资源的互斥访问。

**CPU0 GP** 中断处理函数中，把用于互斥访问的变量 **g\_cpu0Mutex** 赋值为 1。

**CPU0 GP** 中断处理代码：

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

然后，两个内核在各自的主循环代码中会持续监测控制互斥访问的变量，只有当该变量被修改为 1 之后，它们才能使用 **UART** 外设并更新共享变量。两个内核的主循环代码如下。

主核（CPU0）主循环过程:

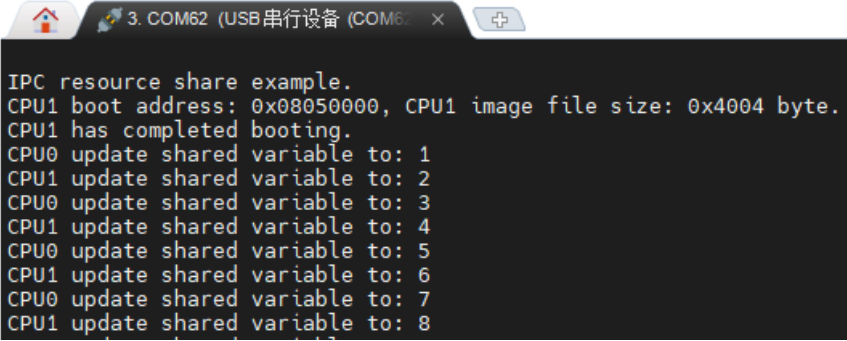
```
//  
// Loop.  
//  
for(;;)  
{  
    //  
    // Wait until CPU0 can access the shared resource  
    //  
    while (g_cpu0Mutex != 1);  
  
    //  
    // CPU0 can change shared variable g_shared  
    //  
    g_shared++;  
  
    printf("CPU0 update shared variable to: %d\r\n", g_shared);  
  
    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);  
}
```

从核（CPU1）主循环过程:

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

该示例运行结果如下:

图 7 资源互斥访问示例在串口终端上的输出结果



```
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 事件控制

IPC 模块可通过 TASK 和 STATE 寄存器实现事件控制机制。该机制的工作原理是:

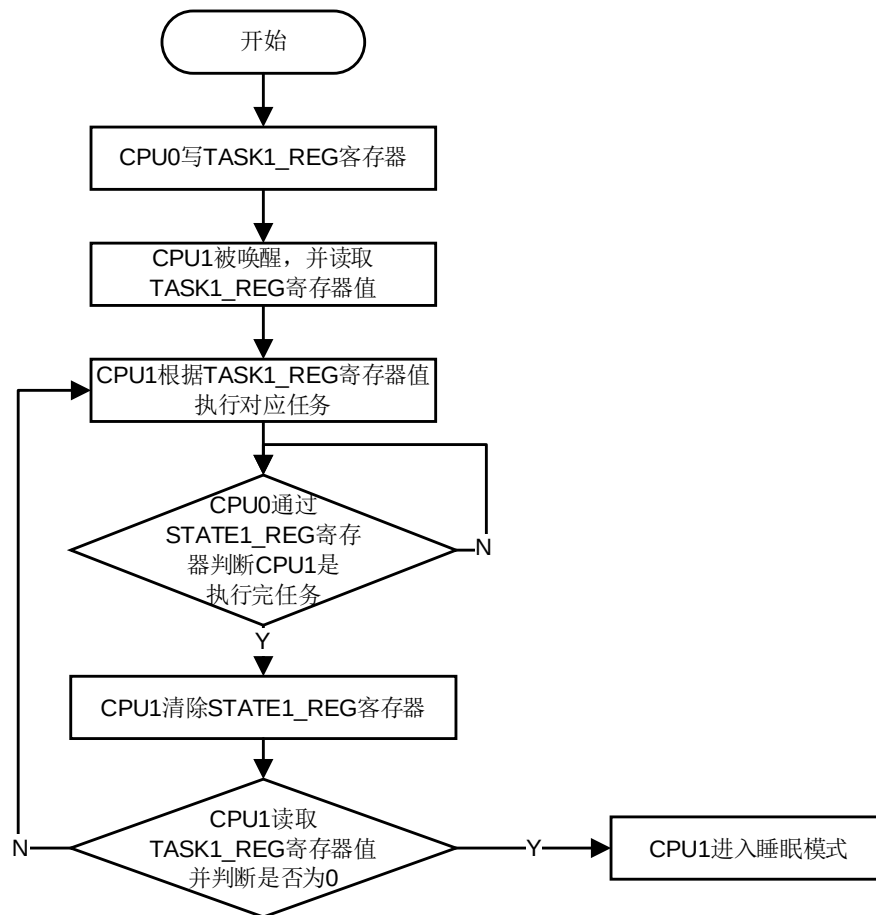
- CPU0 向 TASK 寄存器写入数据, 该操作会生成唤醒信号通知 CPU1;
- CPU1 可以读取 TASK 寄存器的数据, 并根据该寄存器的值执行相应的任务代码。
- CPU0 可以通过读取 STATE 寄存器的值来了解 CPU1 的执行状态。

事件控制的流程如下:

1. CPU0 通过调用 IPC\_issueTask 函数, 将数据写入 TASK1\_REG 寄存器, 从而使 CPU1 能够根据不同的数据 (任务 ID) 执行相应的任务。此外, 该操作可以唤醒处于睡眠模式中的 CPU1。
2. CPU1 在被唤醒后, 通过 IPC\_fetchTask 函数获取 TASK1\_REG 寄存器的值, 该操作会导致硬件清除 TASK1\_REG 寄存器的值。
3. CPU0 可以通过 IPC\_getTaskState 函数获取 STATE1\_REG 寄存器的值, 从而了解 CPU1 的执行状态。
4. 当 CPU1 执行完对应任务后, 通过 IPC\_updateTaskState 函数将 STATE1\_REG 寄存器的值写为 0, 以清除该寄存器的内容。
5. CPU1 再次通过 IPC\_fetchTask 函数获取 TASK1\_REG 寄存器的值。如果该寄存器的值不为零, CPU1 将重复上述步骤; 否则, CPU1 将进入睡眠模式。



图 8 IPC 模块事件控制流程图



在 G32R5xx SDK 中提供的 driverlib\g32r501\examples\eval\ipc\ipc\_ex5\_event\_control 例程中，展示了如何使用 IPC 模块的事件控制机制，CPU1 执行不同的任务。

该示例运行结果如下：

图 9 IPC 事件控制例程在串口终端上的输出结果

```

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 版本历史

表格 3 文件版本历史

| 日期      | 版本  | 变更历史 |
|---------|-----|------|
| 2025.01 | 1.0 | 新建   |

# 声明

本手册由珠海极海半导体有限公司（以下简称“极海”）制订并发布，所列内容均受商标、著作权、软件著作权相关法律法规保护，极海保留随时更正、修改本手册的权利。使用极海产品前请仔细阅读本手册，一旦使用产品则表明您（以下称“用户”）已知悉并接受本手册的所有内容。用户必须按照相关法律法规和本手册的要求使用极海产品。

## 1、权利所有

本手册仅应当被用于与极海所提供的对应型号的芯片产品、软件产品搭配使用，未经极海许可，任何单位或个人均不得以任何理由或方式对本手册的全部或部分内容进行复制、抄录、修改、编辑或传播。

本手册中所列带有“®”或“™”的“极海”或“Geehy”字样或图形均为极海的商标，其他在极海产品上显示的产品或服务名称均为其各自所有者的财产。

## 2、无知识产权许可

极海拥有本手册所涉及的全部权利、所有权及知识产权。

极海不应因销售、分发极海产品及本手册而被视为将任何知识产权的许可或权利明示或默示地授予用户。

如果本手册中涉及任何第三方的产品、服务或知识产权，不应被视为极海授权用户使用前述第三方产品、服务或知识产权，也不应被视为极海对第三方产品、服务或知识产权提供任何形式的保证，包括但不限于任何第三方知识产权的非侵权保证，除非极海在销售订单或销售合同中另有约定。

## 3、版本更新

用户在下单购买极海产品时可获取相应产品的最新版的手册。

如果本手册中所述的内容与极海产品不一致的，应以极海销售订单或销售合同中的约定为准。

#### 4、信息可靠性

本手册相关数据经极海实验室或合作的第三方测试机构批量测试获得，但本手册相关数据难免会出现校正笔误或因测试环境差异所导致的误差，因此用户应当理解，极海对本手册中可能出现的该等错误无需承担任何责任。本手册相关数据仅用于指导用户作为性能参数参照，不构成极海对任何产品性能方面的保证。

用户应根据自身需求选择合适的极海产品，并对极海产品的应用适用性进行有效验证和测试，以确认极海产品满足用户自身的需求、相应标准、安全或其它可靠性要求；若因用户未充分对极海产品进行有效验证和测试而致使用户损失的，极海不承担任何责任。

#### 5、合规要求

用户在使用本手册及所搭配的极海产品时，应遵守当地所适用的所有法律法规。用户应了解产品可能受到产品供应商、极海、极海经销商及用户所在地等各国有关出口、再出口或其它法律的限制，用户（代表其本身、子公司及关联企业）应同意并保证遵守所有关于取得极海产品及/或技术与直接产品的出口和再出口适用法律与法规。

#### 6、免责声明

本手册由极海“按原样”（as is）提供，在适用法律所允许的范围内，极海不提供任何形式的明示或暗示担保，包括但不限于对产品适销性和特定用途适用性的担保。

极海产品并非设计、授权或担保适合用于军事、生命保障系统、污染控制或有害物质管理系统中的关键部件，亦非设计、授权或担保适合用于在产品失效或故障时可导致人员受伤、死亡、财产或环境损害的应用。

如果产品未标明“汽车级”，则表示不适用于汽车应用。如果用户对产品的应用超出极海提供的规格、应用领域、规范，极海不承担任何责任。

用户应该确保对产品的应用符合相应标准以及功能安全、信息安全、环境标准等要求。用户对极海产品的选择和使用负全部的责任。对于用户后续在针对极海产品进行设计、使用的过程中所引起的任何纠纷，极海概不承担责任。

#### 7、责任限制

在任何情况下，除非适用法律要求或书面同意，否则极海和/或以“按原样”形式提供本手册及产品的任何第三方均不承担损害赔偿 responsibility，包括任何一般、特殊因使用或无法使用本手册及产品而产生的直接、间接或附带损害（包括但不限于数据丢失或数据不准确，或用户或第三方遭受的损失），这涵盖了可能导致的人身安全、财产或环境损害等情况，对于这些损害极海概不承担责任。

## 8、适用范围

本手册的信息用以取代本手册所有早期版本所提供的信息。

©2025 珠海极海半导体有限公司 – 保留所有权利