Playing with OurBMC Episode 15: IPMI Knowledge Series

原创 OurBMC Community Posted on 2025-01-22 14:24:54 Read 1.1k Collection 32 Likes 8

Article Tags: linux github Operations OurBMC

> Column introduction: "Play with OurBMC" is a knowledge sharing column created by the OurBMC community, focusing on sharing basic knowledge related to the community and BMC full-stack technology, covering all aspects of knowledge transfer from theoretical principles to practical operations. Through the "Play with OurBMC" column, the OurBMC community will help developers gain a deeper understanding of the community culture, concepts and characteristics, and enhance developers' understanding of BMC full-stack technology.
> Welcome to follow the "Play with OurBMC" column and explore the wonderful world of the OurBMC community. At the same time, we sincerely invite all developers to contribute to the "Play with OurBMC" column, learn and progress together, and build the column into a knowledge garden that gathers everyone wisdom and inspires creativity.

In this information age, the management of servers and data centers is becoming increasingly important. IPMI (Intelligent Platform Management Interface) is an open standard whose core purpose is to achieve comprehensive monitoring and efficient management of hardware status through a mechanism that is not restricted by the operating system and CPU. This article aims to popularize the basic knowledge of IPMI for readers, helping them to deeply understand and effectively use this technology.

IPMI Overview

IPMI was jointly proposed by Intel, DELL, HP and NEC in 1998 and has been supported by more than 170 vendors. It provides a standardized method to monitor and manage the physical health status of servers, including temperature, voltage, fan speed and power status. Its history and development are briefly described as follows:

- 1998: IPMI 1.0 version was released, which initially defined the basic functions of hardware management.
- 2004: IPMI 2.0 version was released, introducing many new features such as remote management, security enhancements, and VLAN support.
- 2010 to present: IPMI continues to evolve, adding more features and optimizations to meet increasingly complex management needs.

IPMI is a standardized management interface that is primarily used to monitor and manage computer hardware. It runs independently of the host operating system and is usually implemented through a baseboard management controller (BMC: a dedicated microcontroller that processes IPMI instructions and monitors hardware status) . IPMI provides a set of standardized commands and protocols for remotely monitoring system status (such as temperature, voltage, fan speed, etc.), performing remote power on/off, reset operations, and obtaining hardware fault information.

IPMI has the following main functions:

(1) Hardware monitoring: monitors the server's temperature, voltage, fan speed, and power status, etc.

(2) Remote management: Remotely control the startup, shutdown and restart of the server through the network.

(3) Event Log: records system events and alarms to help with diagnosis and troubleshooting.

(4) Security management: Provides security authentication and encryption functions to ensure the security of the management interface.

IPMI has a wide range of application scenarios, including but not limited to:

(1) Data center management: monitoring and managing large-scale server clusters.

(2) Remote IT Support: Remotely diagnose and solve problems without on-site operation.

(3) Internet of Things (IoT): used to manage embedded devices and edge computing devices.

(4) Enterprise-level servers: ensuring the stable operation of key business systems.

IPMI has shown significant advantages in practical applications, but it also has some limitations:

- **Advantages:**

(1) Standardization: open and free standards with wide support.

(2) Independence: Independent of the operating system and CPU, suitable for a variety of hardware platforms.

(3) Remote management: Powerful remote management function improves operation and maintenance efficiency.

- **limitation:**

(1) Complexity: Configuring and using IPMI requires certain technical knowledge.

(2) Performance impact: The operation of BMC may have a slight impact on server performance.

IPMI has significant advantages in hardware management interface, such as standardization, independence and remote management, but it also has limitations in terms of complexity, performance impact, security and scalability. In practical applications, it is necessary to comprehensively consider these factors and select the appropriate management interface solution according to specific needs.

The core of IPMI is to interact with system hardware through BMC, but it does not define a specific physical transmission interface. Instead, it relies on a variety of transport layer protocols (such as KCS, BT, SSIF, etc.) to achieve communication with BMC. Next, we will mainly explain the KCS interface.

KCS Interface

KCS (Keyboard Controller Style) is an interface specification for communicating with BMC. It is a transport layer interface of IPMI and is used to transmit IPMI messages between the host and BMC. It defines how the host communicates with BMC through the system bus (such as LPC bus). Its features are as follows:

(1) Based on LPC bus: KCS is usually implemented on the low-speed LPC (Low Pin Count) bus and is suitable for older system architectures.

(2) Simple and efficient: KCS is a relatively simple interface, suitable for scenarios that require low latency and real-time response.

(3) Mainly used for communication between the host and BMC: The host can send commands to the BMC through KCS to obtain system status or perform management operations.

The following introduces the KCS interface principle and protocol format.

1. KCS Interface-BMC Request Message Format

The request message is sent from the system software to the BMC via the KCS interface in the form of a write transfer.

The bytes of the message are organized in a specific format to ensure that the message can be correctly parsed and processed by the BMC.

Table 1 KCS Interface/BMC Request Message Format

Byte 1	Byte 2	Byte 3
NetFn/LUN	Cmd	Data

- **Byte 1: NetFn/LUN**

NetFn (Network Function): NetFn is a network function code used to perform functional routing on messages received through the KCS interface. The NetFn field occupies the most significant six bits of the first byte and ranges from 6-bit binary numbers, ie, 0x00 to 0x3F. The NetFn field provides the first level of message routing for the BMC. Different NetFn values correspond to different functional groups. Even NetFn values: used to send request messages to the BMC. Odd NetFn values: used for response messages returned by the BMC.

LUN (Logical Unit Number): LUN is the logical unit number, which is used to route messages to different logical units behind the same physical interface. The LUN field occupies the least significant two bits of the first byte, ranging from 2-bit binary numbers, that is, 0b00 to 0b11.

Note: Functional group refers to the collection of different functional modules inside BMC (Baseboard Management Controller) (such as network management functional group, system control functional group). Logical unit refers to multiple logical entities (such as temperature sensor, voltage sensor) that can independently process messages under the same interface (such as KCS interface).

- **Byte 2: Cmd (Command)**

Cmd is a command code, which is used to indicate the specific operation to be performed under the specified NetFn function, and occupies the second byte. The Cmd field is used in conjunction with the NetFn field. The BMC finds the corresponding function group according to NetFn, and then performs specific operations according to Cmd.

- **Byte 3:N: Data**

Data is the data part required by the command, which contains zero or more bytes of data. The length of the data depends on the requirements of the specific command. The data transmission order is usually transmitted in the low byte first (LS-byte first) manner.

2. BMC-KCS Interface Response Message Format

The request message is sent from the system software to the BMC via the KCS interface in a write transfer. The bytes of the message are organized in a specific format to ensure that the message can be correctly parsed and processed by the BMC. The response message format is as follows:

Byte1	Byte2	Byte3	Byte4
NetFn/LUN	Cmd	Completion	Code Data

- Byte 1: NetFn/LUN

LUN (Logical Unit Number): Logical Unit Number, this is the return value of LUN passed in the request message. NetFn (Network Function): Network Function Code, this is the return value of NetFn code passed in the request message. However, the NetFn value is an odd number when returned.

- **Byte 2: Cmd**

Cmd: Command code, which is the return value of the Cmd code passed in the request message.

- **Byte 3: Completion Code**

Completion Code: A completion code that indicates whether the request was completed successfully. A successful completion code usually indicates a successful operation, while an e
error information.

- **Byte 4:N: Data**

Data: Zero or more data bytes, depending on the requirements of the specific command. Even if there is no data to be returned, the BMC will return a response to confirm that the request has been received.

3. KCS Interface Registers

The registers of the KCS interface are mapped to the system I/O space. The default system base address is 0xCA2h, and the register size is four bytes wide, as follows:

- Status Register: Provides flags and status bits for various defined operations.
- Command Register: Provides a port for writing "write control code".
- Data Input Register (Data_In): Provides a port for writing data bytes and "control commands to request the next data".
- Data Output Register (Data_Out): Provides a port for reading data bytes.

(1) Status register

The status register information is shown in Table 2 below:

Table 2 Status Register

Bit	Name	Description	R/W[1]
7	S1	State bit 1. Bits 7 & 6 are used to indicate the current state of the KCS Interface. Host Software should examine these bits to verify that it's in sync with the BMC. See below for more detail.	R/O
6	S0	State bit 0. See bit 7.	R/O
5	OEM2	OEM - reserved for BMC implementer / system integrator definition.	R/O
4	OEM1	OEM - reserved for BMC implementer / system integrator definition.	R/O
3	C/D#	Specifies whether the last write was to the Command register or the Data_In register (1=command, 0=data). Set by hardware to indicate whether last write from the system software side was to Command or Data_In register.	R/O
2	SMS_ATN	Set to 1 when the BMC has one or more messages in the Receive Message Queue, or when a watchdog timer pre-timeout, or event message buffer full condition exists[2]. OEMs may also elect to set this flag is one of the OEM 1, 2, or 3 flags from the *Get Message Flags* command becomes set. This bit is related to indicating when the BMC is the source of a system interrupt. Refer to sections *9.12, KCS Communication and Non-communication Interrupts, 9.13, Physical Interrupt Line Sharing*, and *9.14, Additional Specifications for the KCS interface* for additional information on the use and requirements for the SMS_ATN bit.	R/O
1	IBF	Automatically set to 1 when either the associated Command or Data_In register has been written by system-side software.	R/O
0	OBF	Set to 1 when the associated Data_Out register has been written by the BMC.	R/O

Note: The above figure refers to <Intelligent Platform Management Interface Specification Second Generation v2.0 >

Status Register: Located at I/O address base+1, it is a read-only register. It contains the following bits:

- **S1 (bit 7) and S0 (bit 6):**

Together, these two bits represent the current state of the KCS interface. Host software needs to monitor these two bits to ensure synchronization with the BMC. These status bits may ind
data, or in some other mode. By checking these bits, the software can determine when it is safe to read or write, and how to respond to requests from the BMC.

- **OEM2 (bit 5) and OEM1 (bit 4):**

These bits are reserved for custom use by BMC implementers or system integrators. The specific usage depends on the specific implementation and may be used for specific system man

- **C/D# (bit 3):**

This bit indicates whether the most recent write operation was to the command register or the data input register. A value of 1 indicates that the most recent write was to the command register, and a value of 0 indicates that the write was to the data input register. This bit helps the software track its interaction history with the BMC and ensure the correct command and data sequence.

- **SMS_ATN (bit 2):**

This bit is an important interrupt flag bit. This bit is set to 1 when the BMC has a message to send, the watchdog timer pre-times out, or the event message buffer is full. In addition, the OEM can configure this bit to be set when a specific OEM flag is set. The SMS_ATN bit is mainly used to indicate that the BMC is the source of the system interrupt, so that the system software can respond to the BMC's request or event in a timely manner.

- **IBF (bit 1):**

Input buffer full flag. When the system software writes to the command register or data input register, this bit is automatically set to 1, indicating that the input buffer is full and the BMC needs to process the data.

- **OBF (bit 0):**

Output buffer full flag. When the BMC writes to the data output register, this bit is set to 1, indicating that there is data available for the system software to read.

Note: It is clearly pointed out here that the read/write direction is relative to the "system side" (i.e. system software) of the interface. In other words: a read operation is data flowing from the BMC to the system software, and a write operation is data flowing from the system software to the BMC.

Bits 7:6 are the status bits (S1 and S0) of the KCS interface, which are used to indicate the current status of the interface and help the BMC and system software stay synchronized. Different status bit combinations represent different working modes or error states. The status bits are shown in Table 3 below:

Table 3 Status bits

S1 (bit 7)	S0 (bit 6)	Definition
0	0	IDLE_STATE. Interface is idle. System software should not be expecting nor sending any data.
0	1	READ_STATE. BMC is transferring a packet to system software. System software should be in the “Read Message” state.
1	0	WRITE_STATE. BMC is receiving a packet from system software. System software should be writing a command to the BMC.
1	1	ERROR_STATE. BMC has detected a protocol violation at the interface level, or the transfer has been aborted. System software can either use the “Get_Status’ control code to request the nature of the error, or it can just retry the command.

Note: The above figure refers to <Intelligent Platform Management Interface Specification Second Generation v2.0 >

- **IDLE_STATE (00):**

The interface is in idle state, indicating that no data transmission or operation is currently in progress.

System software should not wait for or send any data. This is the default state of the interface and indicates that no activity is occurring.

- **READ_STATE (01):**

The BMC is transmitting data packets to the system software.

The system software should be in the "read message" state, waiting to receive data from the BMC.

- **WRITE_STATE (10):**

The BMC is receiving packets from the system software.

The system software should be in the "write command" state to send commands or data to the BMC.

- **ERROR_STATE (11):**

The BMC detected a protocol violation at the interface level, or the transfer has been aborted.

System software can query the nature of the error via the "Get_Status" control code, or simply retry the previous command.

Note: Whenever the BMC is reset (either power-on reset or hard reset), the status bits will be initialized to "11 - Error Status".

(2) Command register

The command register can be written from the system side only when the IBF flag is cleared, because the status register and the command register are the same address. Only WRITE_START, WRITE_END or GET_STATUS/ABORT control codes can be written to the command register.

(3) Data register

Data packets to and from the BMC are passed through the data registers. These bytes contain all the fields of the message, such as the Network Function code, Command Byte, and any additional data required for the request or response message. The Data_In register can only be written from the system side when the IBF flag is cleared. The OBF flag must be set to 1 to read the Data_Out register (see Status Register).

(4) KCS control code and status code

The Kcs control code is shown in Table 4 below:

Table 4 Kcs control code

Code	Name	Description	Target register	Output Data Register
60h	GET_STATUS / ABORT	Request Interface Status / Abort Current operation	Command	Status Code
61h	WRITE_START	Write the First byte of an Write Transfer	Command	N/A.
62h	WRITE_END	Write the Last byte of an Write Transfer	Command	N/A
63h-67h	reserved	reserved		
68h	READ	Request the next data byte	Data_In	Next byte
69h-6Fh	reserved	reserved		

Note: The above figure refers to <Intelligent Platform Management Interface Specification Second Generation v2.0 >

KCS interface status codes are used during the communication process of the KCS interface, mainly to indicate the current status of the interface or errors that have occurred. Specifically, when the host communicates with the BMC (Baseboard Management Controller) through the KCS interface, these status codes are used to report the results of the operation. The KCS interface status codes are as follows:

Table 5 Kcs interface status code

Code	Description
00h	No Error
01h	Aborted By Command (Transfer in progress was aborted by SMS issuing the Abort/Status control code)
02h	Illegal Control Code
06h	Length Error (e.g.overrun)
C0h-FEh	OEM Error (Error must not fit into one of above categories.)
FFH	Unspecified Error
all other	Reserved

Note: The above figure refers to <Intelligent Platform Management Interface Specification Second Generation v2.0 >

4. Basic communication process of KCS interface

Next, the communication process between the system management software (SMS) and the baseboard management controller (BMC) of the KCS (Keyboard Controller Style) interface is introduced, focusing on the write transfer (Write Transfer) and read transfer (Read Transfer) processes as well as the related states and control mechanisms.

- **Write Transfer:**

The system management software (SMS) first sends a request message to the BMC through a write transfer. Each write operation (control code or data byte) to the Data_In register will set the IBF (Input Buffer Full) flag, triggering the BMC to read the corresponding control code or data byte. After the BMC completes the processing, if the interface uses an interrupt, the BMC will write a virtual value 00h to the Data_Out register and update the status register, triggering the OBF (Output Buffer Full) interrupt to notify the system management software.

- **Read Transfer:**

The system management software writes a read control code (READ Control Code) to the Data_In register to set IBF. The BMC responds to the read control code and writes the data byte to the Data_Out register. If the interface uses interrupts, the OBF interrupt will also be triggered when the data byte is written.

- **Pairing request and response:**

The request message is sent to the BMC through a write transfer, and the response message is read from the BMC through a read transfer. The write transfer and the read transfer together constitute a complete command/response transaction.

- **IDLE_STATE (idle state):**

The interface automatically returns to the idle state after successfully completing a complete command/response transaction.

This means that system management software does not need to explicitly use the GET_STATUS/ABORT command to return to the idle state.

- **ERROR_STATE (error state):**

If an error occurs, the system management software can choose to simply retry the command, or send a "known good" command to clear the error state. This means that when an error occurs, a new command can be sent to clear the previous erroneous register state.

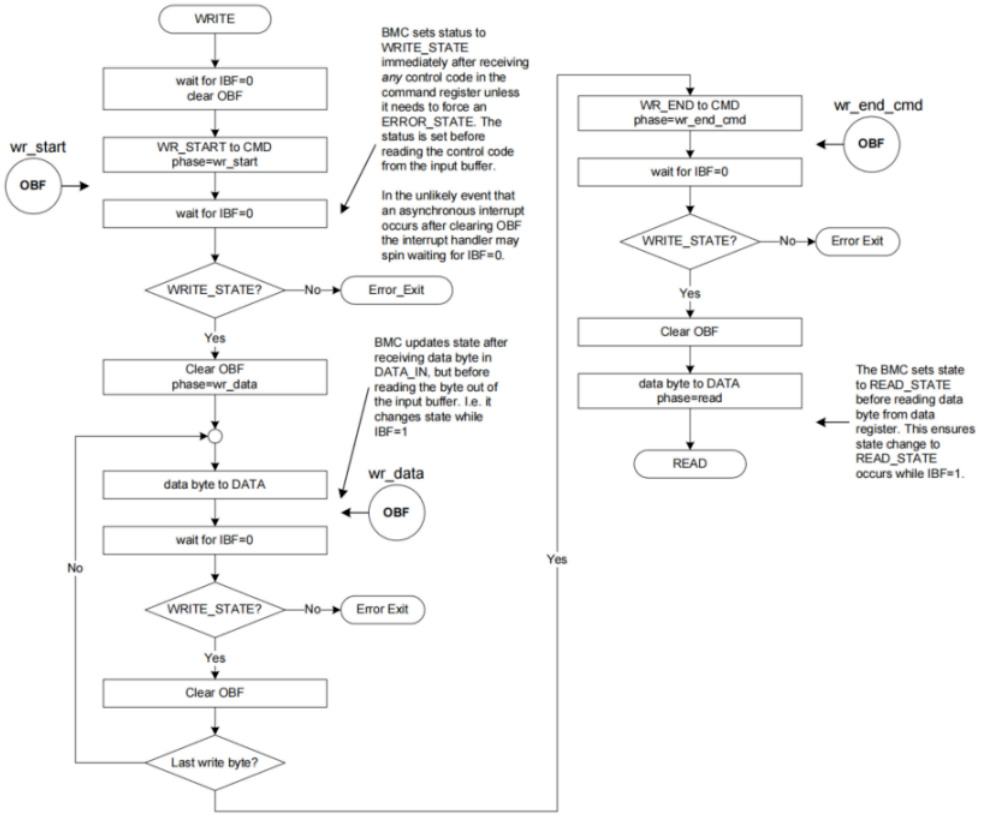
The interface design allows command transmission to be started or restarted at any time when the input buffer is empty.

Note: The purpose of the known good command is to confirm whether the communication between the system management software (SMS) and the baseboard management controller (BMC) is normal, or when an error state (such as ERROR_STATE) is detected, it is cleared by sending a simple command that does not cause an error (such as Get BMC Status, a simple query command).

- **OBF interrupt:**

During a write transfer, the BMC notifies the system management software by writing a dummy value of 00h to the Data_Out register and updating the status register, triggering the OBF interrupt. During a read transfer, the BMC also triggers the OBF interrupt when it writes a data byte to the Data_Out register.

The KCS HOST TO BMC flow chart is as follows:



Note: The above figure refers to <Intelligent Platform Management Interface Specification Second Generation v2.0 >

1) Initial waiting stage: wait for IBF=0 to clear OBF. SMS first waits for the input buffer (IBF) to be empty (IBF=0), indicating that the BMC is ready to receive new commands or data, and clears the OBF initialization environment. Ensure that the previous operation has been completed to avoid data conflicts.

2) Send the WR_START command and wait for IBF to be equal to 0. SMS sends a control code (WR_START) to the command register, indicating the start of the write operation. Notify the BMC that the write operation is about to begin. Wait for IBF to be equal to 0, indicating that the BMC has received the WR_START command.

3) Query WRITE_STATE. SMS checks whether the BMC is in WRITE_STATE state and determines whether the BMC is ready to receive data.

4) Clear OBF. Before sending data, SMS clears the output buffer (OBF) to ensure that the buffer is clean and avoid interference from old data.

5) Send data bytes. SMS writes the data bytes into the data register, that is, transmits the actual data to the BMC.

6) Wait for IBF=0, indicating that the BMC has received the data byte, ensuring that the BMC has processed the data.

7) Continue to query WRITE_STATE and determine whether it is the last byte. If the current byte is not the last byte, loop through steps (5), (6), and (7). If it is the last byte, SMS sends a V
indicating the end of the write operation.

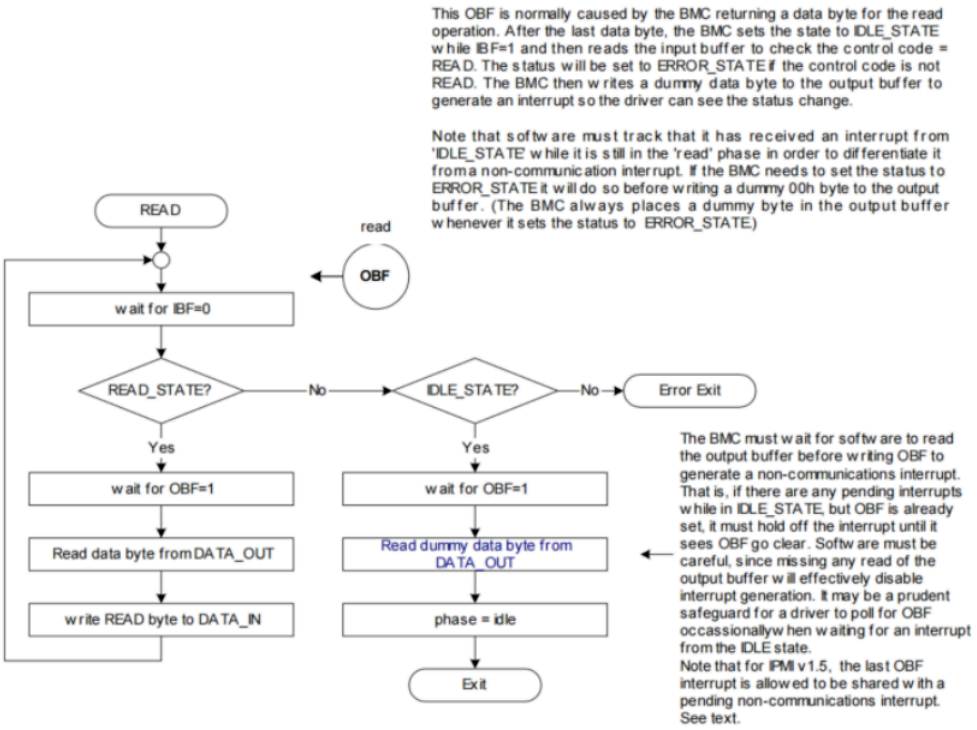
8) SMS will finally send a byte of data to DataIn, and before BMC reads the data byte, it will first set its own state to READ_STATE and then start the reading process.

Points to think about

1. Why does HOST need to send data byte to DATA after writing the last byte?

A: After the HOST finishes writing, it will enter the read phase, but the BMC does not know when to enter the read phase, so it is necessary to write another byte to the BMC to notify the BMC to switch from the write phase to the read phase. (Previously, the HOST wrote WR_END only to indicate the end of writing, and did not mean to tell the BMC to prepare for the phase switch.)

The BMC TO HOST flow chart of KCS is as follows:



Note: The above figure refers to <Intelligent Platform Management Interface Specification Second Generation v2.0 >

1) In the SMS read process, we must first wait until IBF=0, that is, there is no data in the input buffer.

2) If it is a read process, you need to wait for OBF=1, which means that the BMC has put the data into the output buffer and waits for the HOST to read the data.

3) Host reads data byte by byte.

4) After each reading is completed, the Host sends a READ request to read the next data.

5) After BMC sends the last data byte to DataOut, BMC will set the state to IDLE_STATE, which can be understood as false IDLE.

6) After reading the last byte, HOST will determine the current state. Under normal circumstances, BMC is in IDLE state. HOST needs to read the last byte (wait for OBF to be equal to 1, stage will become IDLE.

2. Why does the BMC enter the IDLE state after sending the last byte, and the HOST end needs to wait for OBF to equal 1 and read again after reading the last byte?

A: Before entering the IDLE state, the BMC sent data to Data_OUT, but it was not clear whether the host successfully received and processed the last byte of data. Therefore, it is necessary to confirm that the data transmission is complete.

Note: Whether it is Host To BMC or BMC To Host, as long as you encounter the Error Exit situation, that is, the current communication is abnormal, you can choose to resend this packet, but you must ensure that the status register is reset so that the next transmission can proceed normally. Or you can also refer to the part about exception handling in the IPMI specification. For details, please see the IPMI specification.

This issue mainly explains the basic knowledge of IPMI and the knowledge of Kcs interface. With the development of cloud computing and Internet of Things, IPMI will continue to evolve and add more management and security functions. The future IPMI standard will be more intelligent and automated, and better adapt to diverse application scenarios.

FAQ

Q1: How to determine whether the server supports IPMI?

Usually, the server's BMC module provides IPMI support, which can be confirmed through the server's user manual or BMC management interface.

Q2: Does IPMI require special software?

IPMI can be managed through a variety of tools (such as ipmitool, Web interface, etc.) without installing special software.

LAN interface: Remote management via the network.