[UEFI Practice] UART Initialization

原创 jiangwei0512

Category Column: UEFI Development Basics Article Tags: uefi uart

摘要 This article deeply introduces the working principle, register configuration and programming method of UART, and analyzes in detail the UART initialization process, read and write operation implementation under the x86 platform, and the UART programming model under UEFI BIOS.

The summary is generated in C Know , supported by DeepSeek-R1 full version, go to experience>

illustrate

UART stands for Universal Asynchronous Receiver/Transmitter. Here it refers to a chip that implements serial port communication. Its position in the entire serial port system is shown in the following figure:


```
RS232      +-----------+   +-----------+   +-----------+   +-----------+
Interface  | Line      |   |           |   | Interface |   |           |
-----------+ Driver /  +---+   UART    +---+ Logic     +---+    CPU    |
           | Receiver  |   |           |   |           |   |           |
           +-----------+   +-----+-----+   +-----+-----+   +-----------+
                                 |               |
                                 |               |
                           +-----+-----+         |
                           | Baud Rate |         |
                           | Generator +---------+
                           |           |
                           +-----------+
```


The commonly used UART chip at present is the 8250 series chip.

UART Registers

The UART chip has the following registers:

Offset	DLAB	Access	Abbr	Name
0	0	Write	THR	Transmit Holding Register
0	0	Read	RBR	Receive Buffer Register
0	1	Read/Write	DLL	Divisor Latch LSB
1	1	Read/Write	DLM	Divisor Latch MSB
1	0	Read/Write	IER	Interrupt Enable Register
2	x	Read	IIR	Interrupt Identification Register
2	x	Write	FCR	FIFO Control Register
3	x	Read/Write	LCR	Line Control Register
4	x	Read/Write	MCR	Modem Control Register
5	x	Read	LSR	Line Status Register
6	x	Read	MSR	Modem Status Register
7	x	Read/Write	SCR	Scratch Pad Register

The first column is the offset of the register, and they can be accessed by IO or MMIO (the earliest ones were IO, but now there is also MMIO).

From the table above, we can see that the UART chip has a total of 12 registers, but only 8 ports can be used, so there is multiplexing in the middle. There are different ways of multiplexing, some use the same register for reading and writing, and others have different effects based on the value of the "Divisor Latch Access Bit" (this BIT will be introduced later) (0, 1 means valid, x means unaffected).

The functions of the second and third columns have been explained before.

The fourth line is the abbreviation, which will be used in the macro name in the code.

The fifth column is the name of the serial port register.

THR/RBR : Registers used to store or receive data. For early UART chips, they were sent and received one by one. Modern chips (such as 16550) have an internal buffer space that can store (even simultaneously) 16 bytes (or more bytes) of sent or received data (FIFO). These two registers are the most important parts of UART.

DLL/DLM : These two registers are used to set the baud rate. The data put into them is 115200/BaudRate. The following are all the possible values:

Divisor Latch Byte Values (common baud rates)

Baud Rate	Divisor (in decimal)	Divisor Latch High Byte	Divisor Latch Low Byte
50	2304	\$09	\$00
110	1047	\$04	\$17
220	524	\$02	\$0C
300	384	\$01	\$80
600	192	\$00	\$C0
1200	96	\$00	\$60
2400	48	\$00	\$30
4800	24	\$00	\$18
9600	12	\$00	\$0C
19200	6	\$00	\$06
38400	3	\$00	\$03
57600	2	\$00	\$02
115200	1	\$00	\$01

IER : This is an interrupt related register. The meaning of each BIT is as follows:

Interrupt Enable Register (IER)	
Bit	**Notes**
7	Reserved
6	Reserved
5	Enables Low Power Mode (16750)
4	Enables Sleep Mode (16750)
3	Enable Modem Status Interrupt
2	Enable Receiver Line Status Interrupt
1	Enable Transmitter Holding Register Empty Interrupt
0	Enable Received Data Available Interrupt

IIR : This is a read-valid register. The read content describes the characteristics of this UART (related to interrupts and others). The following is a description of each bit of this register:

Interrupt Identification Register (IIR)

Bit	Notes				
	Bit 7	**Bit 6**			
	0	0	No FIFO on chip		
7 and 6	0	1	Reserved condition		
	1	0	FIFO enabled, but not functioning		
	1	1	FIFO enabled		
5	64 Byte FIFO Enabled (16750 only)				
4	Reserved				
	Bit 3	**Bit 2**	**Bit 1**		**Reset Method**
	0	0	0	Modem Status Interrupt	Reading Modem Status Register(MSR)
3, 2 and 1	0	0	1	Transmitter Holding Register Empty Interrupt	Reading Interrupt Identification Register(IIR) or Writing to Transmit Holding Buffer(THR)
	0	1	0	Received Data Available Interrupt	Reading Receive Buffer Register(RBR)
	0	1	1	Receiver Line Status Interrupt	Reading Line Status Register(LSR)
	1	0	0	Reserved	N/A
	1	0	1	Reserved	N/A
	1	1	0	Time-out Interrupt Pending (16550 & later)	Reading Receive Buffer Register(RBR)
	1	1	1	Reserved	N/A
0	Interrupt Pending Flag				

FCR : FIFO was introduced in the subsequent 8250 chip. It is a write-valid register used to control FIFO characteristics. The various bits are as follows:

FIFO Control Register (FCR)

Bit	Notes			
	Bit 7	**Bit 6**	**Interrupt Trigger Level (16 byte)**	**Trigger Level (64 byte)**
7 & 6	0	0	1 Byte	1 Byte
	0	1	4 Bytes	16 Bytes
	1	0	8 Bytes	32 Bytes
	1	1	14 Bytes	56 Bytes
5	Enable 64 Byte FIFO (16750)			
4	Reserved			
3	DMA Mode Select			
2	Clear Transmit FIFO			
1	Clear Receive FIFO			
0	Enable FIFOs			

LCR : This register has two functions, one is to set DLAB (this bit has been mentioned before), and the other is to set the mode (such as 8-1-None, 5-2-Even, etc.). The following is the meaning of each bit:

Line Control Register (LCR)

Bit	Notes			
7	Divisor Latch Access Bit			
6	Set Break Enable			
3, 4 & 5	**Bit 5**	**Bit 4**	**Bit 3**	**Parity Select**

Line Control Register (LCR)

Bit	Notes		
0	0	0	No Parity
0	0	1	Odd Parity
0	1	1	Even Parity
1	0	1	Mark
1	1	1	Space
2	0	One Stop Bit	
1	1.5 Stop Bits or 2 Stop Bits		
0 & 1	**Bit 1**	**Bit 0**	**Word Length**
0	0	5 Bits	
0	1	6 Bits	
1	0	7 Bits	
1	1	8 Bits	

MCR : This register is used to set the hardware control. The meaning of each bit is as follows:

Modem Control Register (MCR)

Bit	Notes
7	Reserved
6	Reserved
5	Autoflow Control Enabled (16750)
4	Loopback Mode
3	Auxiliary Output 2
2	Auxiliary Output 1
1	Request To Send
0	Data Terminal Ready

LSR : This register is used to describe the errors that occurred in the UART chip. The meaning of each bit is as follows:

Line Status Register (LSR)

Bit	Notes
7	Error in Received FIFO
6	Empty Data Holding Registers
5	Empty Transmitter Holding Register
4	Break Interrupt
3	Framing Error
2	Parity Error
1	Overrun Error
0	Data Ready

MSR : This register is used to describe the status of the UART chip:

Modem Status Register (MSR)

Bit	Notes
7	Carrier Detect
6	Ring Indicator
5	Data Set Ready
4	Clear To Send
3	Delta Data Carrier Detect
2	Trailing Edge Ring Indicator
1	Delta Data Set Ready
0	Delta Clear To Send

SCR : This is hard to say, the function is unknown. But it can be used to test whether the chip exists.

UART Programming

To program the UART (here only for the x86 platform), the first thing you need to know is the base address. Only with the base address can you access all the registers introduced above.

As for how to set the base address, the x86 platform has its own set of rules, which is not covered in this topic. Here is a brief explanation:

1. For early UART chips, the addresses are basically fixed through IO access, including the following:

Common UART IRQ and I/O Port Addresses

COM Port	IRQ	Base Port I/O address
COM1	IRQ4	$3F8
COM2	IRQ3	$2F8
COM3	IRQ4	$3E8
COM4	IRQ3	$2E8

2. Some UARTs are packaged as PCI devices on the x86 PCH and accessed through MMIO. Before PCI scanning, the address is fixed to MMIO, and then the MMIO address scanned by PCI is used.

The difference between the above two is that, in addition to the different addresses, the access size is also different, one is 1 byte and the other is basically 4 bytes.

initialization

There are quite a few libraries or codes involved in the initialization of UART under BIOS, but they are all similar.

Here we take MdeModulePkg\Library\BaseSerialPortLib16550\BaseSerialPortLib16550.inf as an example.

The following is the code description of the SerialPortInitialize() function:

cpp

```cpp
//
// Perform platform specific initialization required to enable use of the 16550 device
// at the location specified by PcdSerialUseMmio and PcdSerialRegisterBase.
//
Status = PlatformHookSerialPortInitialize ();
if (RETURN_ERROR (Status)) {
  return Status;
}
```

The first is some OEM operations, which can usually return success directly.

cpp

```cpp
//
// Calculate divisor for baud generator
//    Ref_Clk_Rate / Baud_Rate / 16
//
Divisor = PcdGet32 (PcdSerialClockRate) / (PcdGet32 (PcdSerialBaudRate) * 16);
if ((PcdGet32 (PcdSerialClockRate) % (PcdGet32 (PcdSerialBaudRate) * 16)) >= PcdGet32 (PcdSerialBaudRate) * 8) {
  Divisor++;
}
```

Configure the value used for baud rate setting. As mentioned before, its value is usually (115200/baud rate).

cpp

```cpp
//
// Get the base address of the serial port in either I/O or MMIO space
//
SerialRegisterBase = GetSerialRegisterBase ();
if (SerialRegisterBase ==0) {
  return RETURN_DEVICE_ERROR;
}
```

Get the UART base address.

```cpp
//
// See if the serial port is already initialized
//
Initialized = TRUE;
if ((SerialPortReadRegister (SerialRegisterBase, R_UART_LCR) & 0x3F) != (PcdGet8 (PcdSerialLineControl) & 0x3F)) {
  Initialized = FALSE;
}
SerialPortWriteRegister (SerialRegisterBase, R_UART_LCR, (UINT8)(SerialPortReadRegister (SerialRegisterBase, R_UART_LCR) | B_UART_LCR_DLAB));
CurrentDivisor =  SerialPortReadRegister (SerialRegisterBase, R_UART_BAUD_HIGH) << 8;
CurrentDivisor |= (UINT32) SerialPortReadRegister (SerialRegisterBase, R_UART_BAUD_LOW);
SerialPortWriteRegister (SerialRegisterBase, R_UART_LCR, (UINT8)(SerialPortReadRegister (SerialRegisterBase, R_UART_LCR) & ~B_UART_LCR_DLAB));
if (CurrentDivisor != Divisor) {
  Initialized = FALSE;
}
if (Initialized) {
  return RETURN_SUCCESS;
}
```



```cpp
//
// Wait for the serial port to be ready.
// Verify that both the transmit FIFO and the shift register are empty.
//
while ((SerialPortReadRegister (SerialRegisterBase, R_UART_LSR) & (B_UART_LSR_TEMT | B_UART_LSR_TXRDY)) != (B_UART_LSR_TEMT | B_UART_LSR_TXRDY));
```

Wait for UART to be available normally.


```cpp
//
// Configure baud rate
//
SerialPortWriteRegister (SerialRegisterBase, R_UART_LCR, B_UART_LCR_DLAB);
SerialPortWriteRegister (SerialRegisterBase, R_UART_BAUD_HIGH, (UINT8) (Divisor >> 8));
SerialPortWriteRegister (SerialRegisterBase, R_UART_BAUD_LOW, (UINT8) (Divisor & 0xff));
```

Set baud rate related registers.


```cpp
//
// Clear DLAB and configure Data Bits, Parity, and Stop Bits.
// Strip reserved bits from PcdSerialLineControl
//
SerialPortWriteRegister (SerialRegisterBase, R_UART_LCR, (UINT8)(PcdGet8 (PcdSerialLineControl) & 0x3F));
```

Set the serial port mode.


```cpp
//
// Enable and reset FIFOs
// Strip reserved bits from PcdSerialFifoControl
//
SerialPortWriteRegister (SerialRegisterBase, R_UART_FCR, 0x00);
SerialPortWriteRegister (SerialRegisterBase, R_UART_FCR, (UINT8)(PcdGet8 (PcdSerialFifoControl) & (B_UART_FCR_FIFOE | B_UART_FCR_FIFO64)));
```

Initialize FIFO.


```cpp
//
// Put Modem Control Register(MCR) into its reset state of 0x00.
//
SerialPortWriteRegister (SerialRegisterBase, R_UART_MCR, 0x00);
```

Set the MCR to reset state.

After that the UART is ready for use.

UART Read Operation

The read operation function in the BaseSerialPortLib16550.inf module is as follows:


```cpp
/**
  Reads data from a serial device into a buffer.

  @param  Buffer           Pointer to the data buffer to store the data read from the serial device.
  @param  NumberOfBytes    Number of bytes to read from the serial device.

  @retval 0                NumberOfBytes is 0.
  @retval >0               The number of bytes read from the serial device.
                           If this value is less than NumberOfBytes, then the read operation failed.

**/
UINTN
EFIAPI
SerialPortRead (
  OUT UINT8     *Buffer,
  IN  UINTN     NumberOfBytes
  )
```


The parameters are the bytes to be read and the number of bytes.

The number of bytes corresponds to a for loop:


```cpp
for (Result = 0; NumberOfBytes-- != 0; Result++, Buffer++) {
```

The loop contains the write operation of each byte:

```cpp
//
// Wait for the serial port to have some data.
//
while ((SerialPortReadRegister (SerialRegisterBase, R_UART_LSR) & B_UART_LSR_RXRDY) == 0) {
  if (PcdGetBool (PcdSerialUseHardwareFlowControl)) {
    //
    // Set RTS to let the peer send some data
    //
    SerialPortWriteRegister (SerialRegisterBase, R_UART_MCR, (UINT8)(Mcr | B_UART_MCR_RTS));
  }
}
```

Here is to read BIT0 in the LSR register:

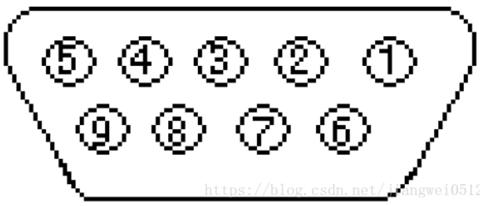
Line Status Register (LSR)

Bit	Notes
0	Data Ready

There is a PCD here, PcdSerialUseHardwareFlowControl, which indicates whether Hardware Flow Control is set. As for what Hardware Flow Control is, you first need to understand Flow Control.

Since UART is a low-speed device, there may be a situation where the data cannot be processed. At this time, it is necessary to tell the other party to delay sending data. This operation is called **Flow Control** . How to implement Flow Control? There are two ways, one is hardware and the other is software.

Hardware Flow Control requires additional hardware to implement, using two PINs: RTS/CTS:



9-pin	25-pin	pin definition	Direction (PC view)
1	8	DCD (Data Carrier Detect)	input
2	3	RX (Receive Data)	input
3	2	TX (Transmit Data)	output
4	20	DTR (Data Terminal Ready)	output
5	7	GND (Signal Ground)	-
6	6	DSR (Data Set Ready)	input
7	4	RTS (Request To Send)	output
8	5	CTS (Clear To Send))	input
9	22	RI (Ring Indicator)	input

The corresponding registers are on MCR/MSR:

Modem Control Register (MCR)

Bit	Notes
7	Reserved
6	Reserved
5	Autoflow Control Enabled (16750)
4	Loopback Mode
3	Auxiliary Output 2
2	Auxiliary Output 1
1	Request To Send
0	Data Terminal Ready

Modem Status Register (MSR)

Bit	Notes
7	Carrier Detect
6	Ring Indicator
5	Data Set Ready
4	Clear To Send
3	Delta Data Carrier Detect
2	Trailing Edge Ring Indicator
1	Delta Data Set Ready
0	Delta Clear To Send

Going back to the above code, in the Hardware Control Flow judgment, we need to write RTS to let the other party send data.

When there is data, RTS will be cleared first:

```cpp
if (PcdGetBool (PcdSerialUseHardwareFlowControl)) {
  //
  // Clear RTS to prevent peer from sending data
  //
  SerialPortWriteRegister (SerialRegisterBase, R_UART_MCR, Mcr);
}
```

Then read the data:

```cpp
//
// Read byte from the receive buffer.
//
*Buffer = SerialPortReadRegister (SerialRegisterBase, R_UART_RXBUF);
```

The above is the reading process.

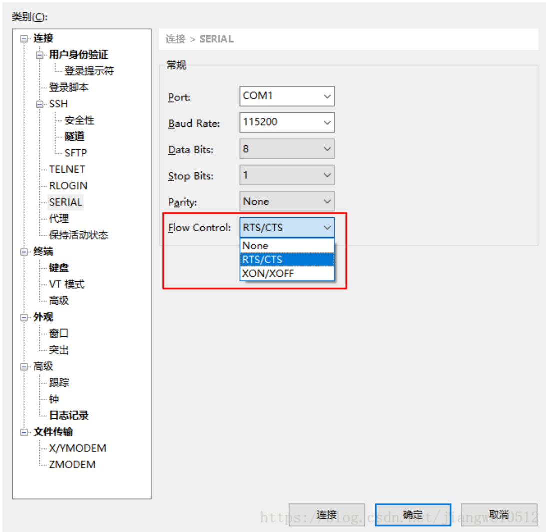
However, there is no special code support for the Software Control Flow code. As the name suggests, it does not require hardware support. In fact, it puts the content that needs to be told to the other party into the transmitted data. This data (actually two data) is XOFF and XON.

In order to stop the other party from sending data, the receiver sends an XOFF, otherwise it sends an XON, **which are both ASCII codes** :

ASCII控制字符

二进制	十进制	十六进制	缩写	可以显示的表示法	名称/意义
0000 0000	0	00	NUL	NUL	空字符（Null）
0000 0001	1	01	SOH	SOH	标题开始
0000 0010	2	02	STX	STX	本文开始
0000 0011	3	03	ETX	ETX	本文结束
0000 0100	4	04	EOT	EOT	传输结束
0000 0101	5	05	ENQ	ENQ	请求
0000 0110	6	06	ACK	ACK	确认回应
0000 0111	7	07	BEL	BEL	响铃
0000 1000	8	08	BS	BS	退格
0000 1001	9	09	HT	HT	水平定位符号
0000 1010	10	0A	LF	LF	换行键
0000 1011	11	0B	VT	VT	垂直定位符号
0000 1100	12	0C	FF	FF	换页键
0000 1101	13	0D	CR	CR	归位键
0000 1110	14	0E	SO	SO	取消变换（Shift out）
0000 1111	15	0F	SI	SI	启用变换（Shift in）
0001 0000	16	10	DLE	DLE	跳出数据通讯
0001 0001	17	11	DC1	DC1	设备控制一（XON 启用软件速度控制）
0001 0010	18	12	DC2	DC2	设备控制二
0001 0011	19	13	DC3	DC3	设备控制三（XOFF 停用软件速度控制）

Regarding Control Flow, there is usually a configuration in the serial port tool:



UART Write Operation

The write operation function in the BaseSerialPortLib16550.inf module is as follows:

cpp　　AI generated projects　　登录复制　　run

```cpp
/**
  Write data from buffer to serial device.

  Writes NumberOfBytes data bytes from Buffer to the serial device.
  The number of bytes actually written to the serial device is returned.
  If the return value is less than NumberOfBytes, then the write operation failed.

  If Buffer is NULL, then ASSERT().

  If NumberOfBytes is zero, then return 0.

  @param  Buffer           Pointer to the data buffer to be written.
  @param  NumberOfBytes    Number of bytes to written to the serial device.

  @retval 0                NumberOfBytes is 0.
  @retval >0               The number of bytes written to the serial device.
                           If this value is less than NumberOfBytes, then the write operation failed.

**/
UINTN
EFIAPI
SerialPortWrite (
  IN UINT8     *Buffer,
  IN UINTN     NumberOfBytes
  )
```

收起 ^

The parameters are the bytes to be written and the number of bytes.

Before writing, you first need to get the value of FIFO:

cpp　　AI generated projects　　登录复制　　run

```cpp
  //
  // Compute the maximum size of the Tx FIFO
  //
  FifoSize = 1;
  if ((PcdGet8 (PcdSerialFifoControl) & B_UART_FCR_FIFOE) != 0) {
    if ((PcdGet8 (PcdSerialFifoControl) & B_UART_FCR_FIFO64) == 0) {
      FifoSize = 16;
    } else {
      FifoSize = PcdGet32 (PcdSerialExtendedTxFifoSize);
    }
  }
```

收起 ^

The reason why we need to get the value of FIFO is because there is a cache in the UART chip, which can write multiple values at a time.

Then there is a loop to write all the bytes:


```cpp
while (NumberOfBytes != 0) {
```

In the loop all characters are processed.


```cpp
    //
    // Wait for the serial port to be ready, to make sure both the transmit FIFO
    // and shift register empty.
    //
    while ((SerialPortReadRegister (SerialRegisterBase, R_UART_LSR) & B_UART_LSR_TEMT) == 0);
```

The first thing is to wait for the FIFO to be emptied, and then write FifoSize characters:


```cpp
    //
    // Fill then entire Tx FIFO
    //
    for (Index = 0; Index < FifoSize && NumberOfBytes != 0; Index++, NumberOfBytes--, Buffer++) {
      //
      // Wait for the hardware flow control signal
      //
      while (!SerialPortWritable (SerialRegisterBase));

      //
      // Write byte to the transmit buffer.
      //
      SerialPortWriteRegister (SerialRegisterBase, R_UART_TXBUF, *Buffer);
    }
```


What needs to be explained here is the SerialPortWritable() function:


```cpp
BOOLEAN
SerialPortWritable (
  UINTN  SerialRegisterBase
  )
{
  if (PcdGetBool (PcdSerialUseHardwareFlowControl)) {
    if (PcdGetBool (PcdSerialDetectCable)) {
      //
      // Wait for both DSR and CTS to be set
      //   DSR is set if a cable is connected.
      //   CTS is set if it is ok to transmit data
      //
      //   DSR  CTS  Description                               Action
      //   ===  ===  ========================================  ========
      //    0    0   No cable connected.                       Wait
      //    0    1   No cable connected.                       Wait
      //    1    0   Cable connected, but not clear to send.   Wait
      //    1    1   Cable connected, and clear to send.       Transmit
      //
      return (BOOLEAN) ((SerialPortReadRegister (SerialRegisterBase, R_UART_MSR) & (B_UART_MSR_DSR | B_UART_MSR_CTS)) == (B_UART_MSR_DSR | B_UART_MSR_CTS));
    } else {
      //
      // Wait for both DSR and CTS to be set OR for DSR to be clear.
      //   DSR is set if a cable is connected.
      //   CTS is set if it is ok to transmit data
      //
      //   DSR  CTS  Description                               Action
      //   ===  ===  ========================================  ========
      //    0    0   No cable connected.                       Transmit
      //    0    1   No cable connected.                       Transmit
      //    1    0   Cable connected, but not clear to send.   Wait
      //    1    1   Cable connected, and clar to send.        Transmit
      //
      return (BOOLEAN) ((SerialPortReadRegister (SerialRegisterBase, R_UART_MSR) & (B_UART_MSR_DSR | B_UART_MSR_CTS)) != (B_UART_MSR_DSR));
    }
  }

  return TRUE;
}
```


Basically you can understand it by reading the comments, so I won't explain it here.

The above is the writing process.

PciSioSerialDxe

The content introduced above is the most common UART initialization, reading and writing.

They are usually used in the final implementation of the DEBUG macro.

What we want to introduce here is the programming model of UART (or serial port) under UEFI BIOS, which is part of the entire UEFI BIOS input and output protocol stack. There are some basic introductions about the input and output of this protocol stack in [UEFI Practice] EFI System Table , mainly the installation of gEfiSerialIoProtocolGuid.

The programming model of UART is as follows:

```cpp
SERIAL_DEV  gSerialDevTemplate = {
  SERIAL_DEV_SIGNATURE,
  NULL,
  {
    SERIAL_IO_INTERFACE_REVISION,
    SerialReset,
    SerialSetAttributes,
    SerialSetControl,
    SerialGetControl,
    SerialWrite,
    SerialRead,
    NULL
  },                                       // SerialIo
  {
```

```
    SERIAL_PORT_SUPPORT_CONTROL_MASK,
    SERIAL_PORT_DEFAULT_TIMEOUT,
    0,
    16,
    0,
    0,
    0
  },                                    // SerialMode
  NULL,                                 // DevicePath
  NULL,                                 // ParentDevicePath
  {
    {
      MESSAGING_DEVICE_PATH,
      MSG_UART_DP,
      {
        (UINT8) (sizeof (UART_DEVICE_PATH)),
        (UINT8) ((sizeof (UART_DEVICE_PATH)) >> 8)
      }
    },
    0, 0, 0, 0, 0
  },                                    // UartDevicePath
  0,                                    // BaseAddress
  FALSE,                                // MmioAccess
  1,                                    // RegisterStride
  0,                                    // ClockRate
  16,                                   // ReceiveFifoDepth
  { 0, 0 },                             // Receive;
  16,                                   // TransmitFifoDepth
  { 0, 0 },                             // Transmit;
  FALSE,                                // SoftwareLoopbackEnable;
  FALSE,                                // HardwareFlowControl;
  NULL,                                 // *ControllerNameTable;
  FALSE,                                // ContainsControllerNode;
  0,                                    // Instance;
  NULL                                  // *PciDeviceInfo;
};
```

收起 ︿

The implementation of UART initialization (SerialReset), reading (SerialRead) and writing (SerialWrite) are already available in the file SerialIo.c. The corresponding structure is as follows:

cpp AI generated projects 登录复制 run

```cpp
typedef struct {
  UINT32                   Signature;
  EFI_HANDLE               Handle;
  EFI_SERIAL_IO_PROTOCOL   SerialIo;
  EFI_SERIAL_IO_MODE       SerialMode;
  EFI_DEVICE_PATH_PROTOCOL *DevicePath;

  EFI_DEVICE_PATH_PROTOCOL *ParentDevicePath;
  UART_DEVICE_PATH         UartDevicePath;

  EFI_PHYSICAL_ADDRESS     BaseAddress;       ///< UART base address
  BOOLEAN                  MmioAccess;        ///< TRUE for MMIO, FALSE for IO
  UINT8                    RegisterStride;    ///< UART Register Stride
  UINT32                   ClockRate;         ///< UART clock rate

  UINT16                   ReceiveFifoDepth;  ///< UART receive FIFO depth in bytes.
  SERIAL_DEV_FIFO          Receive;           ///< The FIFO used to store received data

  UINT16                   TransmitFifoDepth; ///< UART transmit FIFO depth in bytes.
  SERIAL_DEV_FIFO          Transmit;          ///< The FIFO used to store to-transmit data

  BOOLEAN                  SoftwareLoopbackEnable;
  BOOLEAN                  HardwareFlowControl;
  EFI_UNICODE_STRING_TABLE *ControllerNameTable;
  BOOLEAN                  ContainsControllerNode; ///< TRUE if the device produced contains Controller node
  UINT32                   Instance;
  PCI_DEVICE_INFO          *PciDeviceInfo;
} SERIAL_DEV;
```

收起 ︿

The more important Protocol is EFI_SERIAL_IO_PROTOCOL, in which SerialReset is initialized and there is nothing much to say. The remaining SerialRead and SerialWrite will be introduced later.

The most important one is SERIAL_DEV_FIFO:

cpp AI generated projects 登录复制 run

```cpp
typedef struct {
  UINT16  Head;                      ///< Head pointer of the FIFO. Empty when (Head == Tail).
  UINT16  Tail;                      ///< Tail pointer of the FIFO. Full when ((Tail + 1) % SERIAL_MAX_FIFO_SIZE == Head).
  UINT8   Data[SERIAL_MAX_FIFO_SIZE]; ///< Store the FIFO data.
} SERIAL_DEV_FIFO;
```

There is a FIFO buffer corresponding to the UART chip, and SerialReceiveTransmit() will operate on this buffer.

SerialWrite

The prototype is as follows:

cpp AI generated projects 登录复制 run

```cpp
/**
  Write the specified number of bytes to serial device.

  @param This               Pointer to EFI_SERIAL_IO_PROTOCOL
  @param  BufferSize         On input the size of Buffer, on output the amount of
                       data actually written
  @param  Buffer             The buffer of data to write

  @retval EFI_SUCCESS        The data were written successfully
  @retval EFI_DEVICE_ERROR   The device reported an error
  @retval EFI_TIMEOUT        The write operation was stopped due to timeout

**/
EFI_STATUS
EFIAPI
SerialWrite (
  IN EFI_SERIAL_IO_PROTOCOL  *This,
  IN OUT UINTN               *BufferSize,
  IN VOID                    *Buffer
  )
```

The parameters are basically the same, and there is nothing particularly good to explain.

The core part of this function is as follows:


```cpp
  for (Index = 0; Index < *BufferSize; Index++) {
    SerialFifoAdd (&SerialDevice->Transmit, CharBuffer[Index]);

    while (SerialReceiveTransmit (SerialDevice) != EFI_SUCCESS || !SerialFifoEmpty (&SerialDevice->Transmit)) {
      //
      //  Unsuccessful write so check if timeout has expired, if not,
      //  stall for a bit, increment time elapsed, and try again
      //
      if (Elapsed >= Timeout) {
        *BufferSize = ActualWrite;
        gBS->RestoreTPL (Tpl);
        return EFI_TIMEOUT;
      }

      gBS->Stall (TIMEOUT_STALL_INTERVAL);

      Elapsed += TIMEOUT_STALL_INTERVAL;
    }

    ActualWrite++;
    //
    //  Successful write so reset timeout
    //
    Elapsed = 0;
  }
```


It is divided into several steps:

1. Put the string into FIFO, i.e. SerialFifoAdd(), which is implemented by putting data into a 16-byte array in SerialDevice->Transmit, simulating the buffer in the UART chip;

2. SerialReceiveTransmit() is the function that transmits data;

3. After the transmission is completed, it is necessary to determine whether the data in SerialDevice->Transmit has been cleared. It is judged by SerialFifoEmpty(). It must be empty to be normal. As for why it is empty, the key point is that SerialReceiveTransmit() will clear the data in SerialDevice->Transmit using the SerialFifoRemove() function.

SerialRead

The prototype is as follows:


```cpp
/**
  Read the specified number of bytes from serial device.

  @param This               Pointer to EFI_SERIAL_IO_PROTOCOL
  @param BufferSize         On input the size of Buffer, on output the amount of
                            data returned in buffer
  @param Buffer             The buffer to return the data into

  @retval EFI_SUCCESS        The data were read successfully
  @retval EFI_DEVICE_ERROR   The device reported an error
  @retval EFI_TIMEOUT        The read operation was stopped due to timeout

**/
EFI_STATUS
EFIAPI
SerialRead (
  IN EFI_SERIAL_IO_PROTOCOL  *This,
  IN OUT UINTN               *BufferSize,
  OUT VOID                   *Buffer
  )
```


First do a send and receive:


```cpp
  Status  = SerialReceiveTransmit (SerialDevice);

  if (EFI_ERROR (Status)) {
    *BufferSize = 0;

    REPORT_STATUS_CODE_WITH_DEVICE_PATH (
      EFI_ERROR_CODE,
      EFI_P_EC_INPUT_ERROR | EFI_PERIPHERAL_SERIAL_PORT,
      SerialDevice->DevicePath
      );

    gBS->RestoreTPL (Tpl);

    return EFI_DEVICE_ERROR;
  }
```


If there is data, the data will be written to the array in SerialDevice->Receive.

Then read each data:

```cpp
  for (Index = 0; Index < *BufferSize; Index++) {
    while (SerialFifoRemove (&SerialDevice->Receive, &(CharBuffer[Index])) != EFI_SUCCESS) {
      //
      //  Unsuccessful read so check if timeout has expired, if not,
      //  stall for a bit, increment time elapsed, and try again
      //  Need this time out to get conspliter to work.
      //
      if (Elapsed >= This->Mode->Timeout) {
        *BufferSize = Index;
        gBS->RestoreTPL (Tpl);
        return EFI_TIMEOUT;
```

```
      }
      gBS->Stall (TIMEOUT_STALL_INTERVAL);
      Elapsed += TIMEOUT_STALL_INTERVAL;

      Status = SerialReceiveTransmit (SerialDevice);
      if (Status == EFI_DEVICE_ERROR) {
        *BufferSize = Index;
        gBS->RestoreTPL (Tpl);
        return EFI_DEVICE_ERROR;
      }
    }
    //
    //  Successful read so reset timeout
    //
    Elapsed = 0;
  }
```


The reading action is actually completed in SerialFifoRemove(), because the data has been placed in the array in SerialDevice->Receive when SerialReceiveTransmit() is called.

Finally, I sent and received again:


```cpp
SerialReceiveTransmit (SerialDevice);
```

This function calls SerialReceiveTransmit() twice to receive data.

SerialReceiveTransmit

Whether reading or writing, SerialReceiveTransmit() is used. Here the focus is on the SerialReceiveTransmit() function. This function is divided into two parts. The first part is as follows:


```cpp
if (SerialDevice->SoftwareLoopbackEnable) {
  do {
    ReceiveFifoFull = SerialFifoFull (&SerialDevice->Receive);
    if (!SerialFifoEmpty (&SerialDevice->Transmit)) {
      SerialFifoRemove (&SerialDevice->Transmit, &Data);
      if (ReceiveFifoFull) {
        return EFI_OUT_OF_RESOURCES;
      }
```


Normally this is not used under normal circumstances because it is only used for testing on a software.

The second part is actually used, the code is described below:


```cpp
  ReceiveFifoFull = SerialFifoFull (&SerialDevice->Receive);
  //
  // For full handshake flow control, tell the peer to send data
  // if receive buffer is available.
  //
  if (SerialDevice->HardwareFlowControl &&
      !FeaturePcdGet(PcdSerialUseHalfHandshake)&&
      !ReceiveFifoFull
      ) {
    Mcr.Data     = READ_MCR (SerialDevice);
    Mcr.Bits.Rts = 1;
    WRITE_MCR (SerialDevice, Mcr.Data);
  }
```


1. First, determine whether the buffer in SerialDevice->Receive is full. If it is not full, notify the other party to send data (assuming that Hardware Control Flow is enabled).


```cpp
    Lsr.Data = READ_LSR (SerialDevice);
```

2. Read the value of LSR, which indicates the status of the UART chip (mostly an error status). LSR has been introduced before, and many of its bits will be used, so I will list them here again:

Line Status Register (LSR)

Bit	Notes
7	Error in Received FIFO
6	Empty Data Holding Registers
5	Empty Transmitter Holding Register
4	Break Interrupt
3	Framing Error
2	Parity Error
1	Overrun Error

Line Status Register (LSR)

Bit	Notes
0	Data Ready


```cpp
    //
    // Flush incomming data to prevent a an overrun during a long write
    //
    if ((Lsr.Bits.Dr == 1) && !ReceiveFifoFull) {
```

3. Check if there is readable data and whether the buffer of SerialDevice->Receive is full. If it is full, an error is reported:

```cpp
      ReceiveFifoFull = SerialFifoFull (&SerialDevice->Receive);
      if (!ReceiveFifoFull) {
        // Other operations.
      } else {
        REPORT_STATUS_CODE_WITH_DEVICE_PATH (
          EFI_PROGRESS_CODE,
          EFI_P_SERIAL_PORT_PC_CLEAR_BUFFER | EFI_PERIPHERAL_SERIAL_PORT,
          SerialDevice->DevicePath
          );
```

```
        }
```

收起 ^

4. If it is not full, execute the above (Other operations) and check the other bits of LSR:

cpp AI generated projects 登录复制 run

```cpp
        if (Lsr.Bits.FIFOe == 1 || Lsr.Bits.Oe == 1 || Lsr.Bits.Pe == 1 || Lsr.Bits.Fe == 1 || Lsr.Bits.Bi == 1) {
          REPORT_STATUS_CODE_WITH_DEVICE_PATH (
            EFI_ERROR_CODE,
            EFI_P_EC_INPUT_ERROR | EFI_PERIPHERAL_SERIAL_PORT,
            SerialDevice->DevicePath
            );
          if (Lsr.Bits.FIFOe == 1 || Lsr.Bits.Pe == 1|| Lsr.Bits.Fe == 1 || Lsr.Bits.Bi == 1) {
            Data = READ_RBR (SerialDevice);
            continue;
          }
        }
```

收起 ^

Then the data is still read, but the data is problematic, so it is not put into the buffer of SerialDevice->Receive, and then it returns to step 2.

5. If it goes down, it means there is no error, so read the data and put it into the buffer of SerialDevice->Receive:

cpp AI generated projects 登录复制 run

```cpp
        Data = READ_RBR (SerialDevice);

        SerialFifoAdd (&SerialDevice->Receive, Data);
```

6. Then notify the other party not to send data:

cpp AI generated projects 登录复制 run

```cpp
        //
        // For full handshake flow control, if receive buffer full
        // tell the peer to stop sending data.
        //
        if (SerialDevice->HardwareFlowControl &&
            !FeaturePcdGet(PcdSerialUseHalfHandshake)   &&
            SerialFifoFull (&SerialDevice->Receive)
            ) {
          Mcr.Data     = READ_MCR (SerialDevice);
          Mcr.Bits.Rts = 0;
          WRITE_MCR (SerialDevice, Mcr.Data);
        }


        continue;
```

收起 ^

Then go back to step 2.

7. The above contents are all related to reading. The following code will not be executed until the buffer of SerialDevice->Receive is full.

AI generated projects 登录复制

```
      //
      // Do the write
      //
      if (Lsr.Bits.Thre == 1 && !SerialFifoEmpty (&SerialDevice->Transmit)) {
        // Write data to UART.
      }
```

8. In the above code, Thre==1 means that the UART chip can receive data. At the same time, if the SerialDevice->Transmit buffer is not empty, then write data to the UART. However, there are two cases here. For non-Hardware Control Flow, write the register directly:

cpp AI generated projects 登录复制 run

```cpp
        SerialFifoRemove (&SerialDevice->Transmit, &Data);
        WRITE_THR (SerialDevice, Data);
```

Otherwise there are more operations:

```
      if (SerialDevice->HardwareFlowControl) {
        //
        // For half handshake flow control assert RTS before sending.
        //
        if (FeaturePcdGet(PcdSerialUseHalfHandshake)) {
          Mcr.Data     = READ_MCR (SerialDevice);
          Mcr.Bits.Rts= 0;
          WRITE_MCR (SerialDevice, Mcr.Data);
        }
        //
        // Wait for CTS
        //
        TimeOut   = 0;
        Msr.Data  = READ_MSR (SerialDevice);
        while ((Msr.Bits.Dcd == 1) && ((Msr.Bits.Cts == 0) ^ FeaturePcdGet(PcdSerialUseHalfHandshake))) {
          gBS->Stall (TIMEOUT_STALL_INTERVAL);
          TimeOut++;
          if (TimeOut > 5) {
            break;
          }

          Msr.Data = READ_MSR (SerialDevice);
        }

        if ((Msr.Bits.Dcd == 0) || ((Msr.Bits.Cts == 1) ^ FeaturePcdGet(PcdSerialUseHalfHandshake))) {
          SerialFifoRemove (&SerialDevice->Transmit, &Data);
          WRITE_THR (SerialDevice, Data);
        }

        //
        // For half handshake flow control, tell DCE we are done.
        //
        if (FeaturePcdGet(PcdSerialUseHalfHandshake)) {
```

```
          Mcr.Data = READ_MCR (SerialDevice);
          Mcr.Bits.Rts = 1;
          WRITE_MCR (SerialDevice, Mcr.Data);
        }
      }
```

收起

9. Finally, it is a matter of judging whether to send data:

cpp AI generated projects 登录复制 run

```cpp
while (Lsr.Bits.Thre == 1 && !SerialFifoEmpty (&SerialDevice->Transmit));
```

If you want to send data, go back to step 2.

That's all for SerialReceiveTransmit().

In the simplest case, if we do not use Hardware Control Flow, the operation is simplified to the following form:

cpp AI generated projects 登录复制 run

```cpp
ReceiveFifoFull = SerialFifoFull (&SerialDevice->Receive);
do {
  Lsr.Data = READ_LSR (SerialDevice);
  if ((Lsr.Bits.Dr == 1) && !ReceiveFifoFull) {
    ReceiveFifoFull = SerialFifoFull (&SerialDevice->Receive);
    if (!ReceiveFifoFull) {
      Data = READ_RBR (SerialDevice);
      SerialFifoAdd (&SerialDevice->Receive, Data);
      continue;
    } else {
      REPORT_STATUS_CODE_WITH_DEVICE_PATH (
        EFI_PROGRESS_CODE,
        EFI_P_SERIAL_PORT_PC_CLEAR_BUFFER | EFI_PERIPHERAL_SERIAL_PORT,
        SerialDevice->DevicePath
        );
    }
  }
  if (Lsr.Bits.Thre == 1 && !SerialFifoEmpty (&SerialDevice->Transmit)) {
    SerialFifoRemove (&SerialDevice->Transmit, &Data);
    WRITE_THR (SerialDevice, Data);
  }
} while (Lsr.Bits.Thre == 1 && !SerialFifoEmpty (&SerialDevice->Transmit));
```

收起

The above is an introduction to UART.