12. 如何基于Cortex-A9的UART从头实现printf函数

原创 土豆居士 一口Linux 2020-12-30 13:32

收录于合集
#从0学arm 27 #所有原创 206

ARM系列文章合集如下：

《从0学arm合集》

0. 前言

Uart在一个嵌入式系统中是一个非常重要的模块，他承担了CPU与用户交互的桥梁。用户输入信息给程序、CPU要打印一些信息给终端都要依赖UART。

本文将以Exynos4412的UART控制器为基础，讲解UART的原理以及驱动程序如何编写。

1. UART是什么

UART是通用异步收发传输器（Universal Asynchronous Receiver/Transmitter），通常称作UART，是一种异步收发传输器,是设备间进行异步通信的关键模块。UART负责处理数据总线和串行口之间的串/并、并/串转换，并规定了帧格式；通信双方只要采用相同的帧格式和波特率，就能在未共享时钟信号的情况下，仅用两根信号线（Rx 和Tx）就可以完成通信过程，因此也称为异步串行通信。UART总线双向通信，可以实现全双工传输和接收。在嵌入式设计中，UART用于主机与辅助设备通信，如汽车音响与外接AP之间的通信，与PC机通信包括与监控调试器和其它器件，如EEPROM通信。

通常需要加入一个合适的电平转换器，如SP3232E、SP3485，UART还能用于RS-232、RS-485 通信，或与计算机的端口连接。UART 应用非常广泛，手机、工业控制、PC 等应用中都要用到UART。

在这里插入图片描述

2. UART通信方式

UART使用的是 异步，串行通信方式。

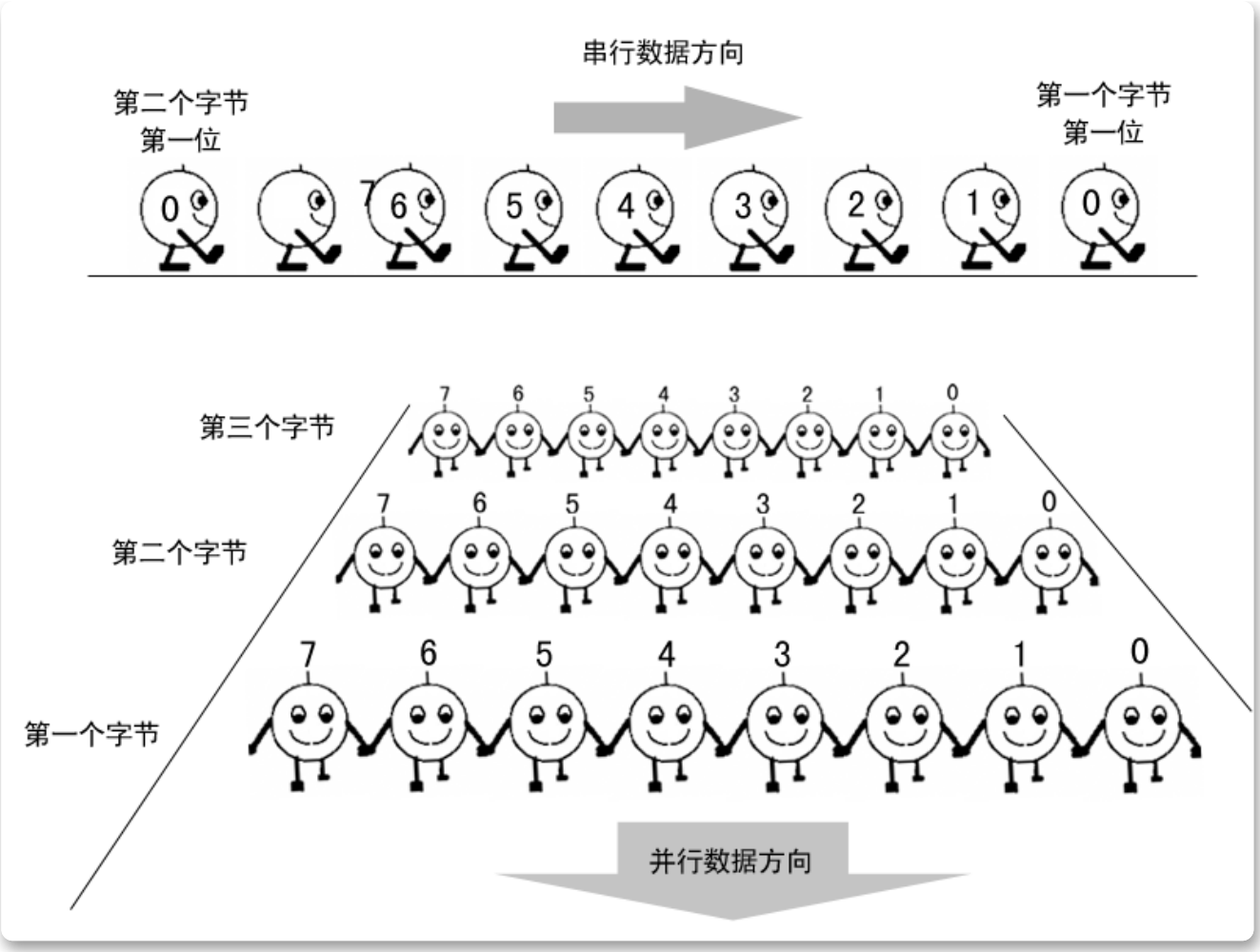
串行通信

串行通信是指利用一条传输线将资料一位位地顺序传送。好比是一列纵队，每个数据元素依次纵向排列。如下图所示，传输时一个比特一个比特的串行传输，每个时钟周期传输一个比特，这种传输方式相对比较简单，速度较慢，但是使用总线数较少，通常一根接收线，一根发送线即可实现串行通信。

它的缺点是要增加额外的数据来控制一个数据帧的开始和结束。特点是通信线路简单，利用简单的线缆就可实现通信，降低成本，适用于远距离通信，但传输速度慢的应用场合。

并行通信

并行通信好比一排横队，齐头并进同时传输。这种通信方式每个时钟周期传输的数据量和其总线宽度成正比，但是实现较为复杂。



在这里插入图片描述

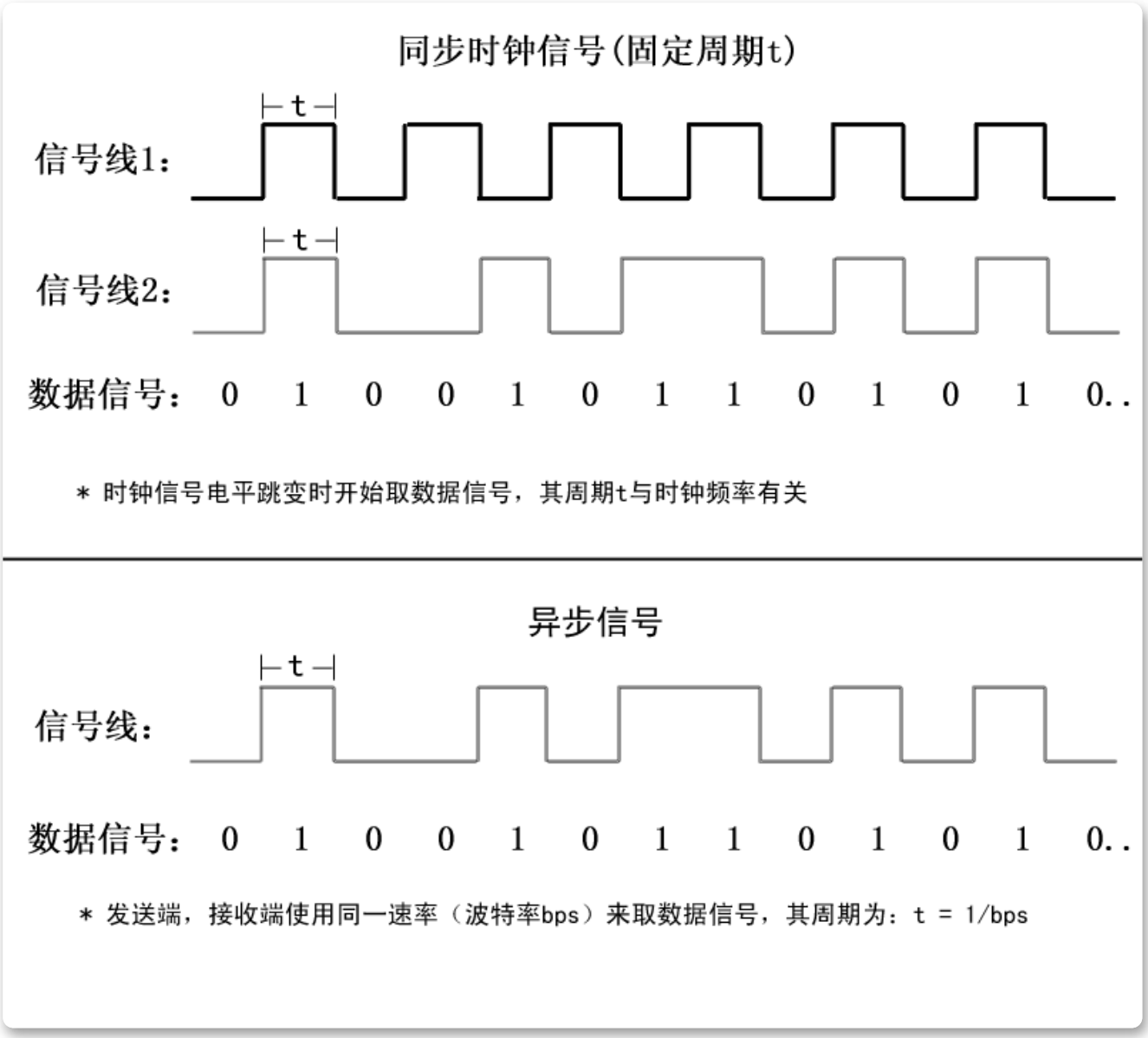
异步通信

异步通信以一个字符为传输单位，通信中两个字符间的时间间隔多少是不固定的，然而在同一个字符中的两个相邻位间的时间间隔是固定的。

在异步通信技术中，数据发送方和数据接收方没有同步时钟，只有数据信号线，只不过发送端和接收端会按照协商好的协议（固定频率）来进行数据采样。数据发送方以每秒钟57600bits的速度发送数据，接收方也以57600bits的速度去接收数据，这样就可以保证数据的有效和正确。通常异步通信中使用波特率（Baud-Rate）来规定双方传输速度，其单位为bps（bits per second每秒传输位数）。

同步通信

在发送数据信号的时候，会同时送出一根同步时钟信号， 用来同步发送方和接收方的数据采样频率。如下图所示，同步通信时，信号线1是一根同步时钟信号线，以固定的频率进行电平的切换，其频率周期为t，在每个电平的上升沿之后进行对同步送出的数据信号线2进行采样（高电平代表1，低电平代表0），根据采样数据电平高低取得输出数据信息。如果双方没有同步时钟的话，那么接收方就不知道采样周期，也就不能正常的取得数据信息。



在这里插入图片描述

3. 帧格式

数据传送速率用波特率来表示，即每秒钟传送的二进制位数。例如数据传送速率为120字符/秒，而每一个字符为10位（1个起始位，7个数据位，1个校验位，1个结束位），则其传送的波特率为10×120＝1200字符/秒＝1200波特。数据通信格式如下图：

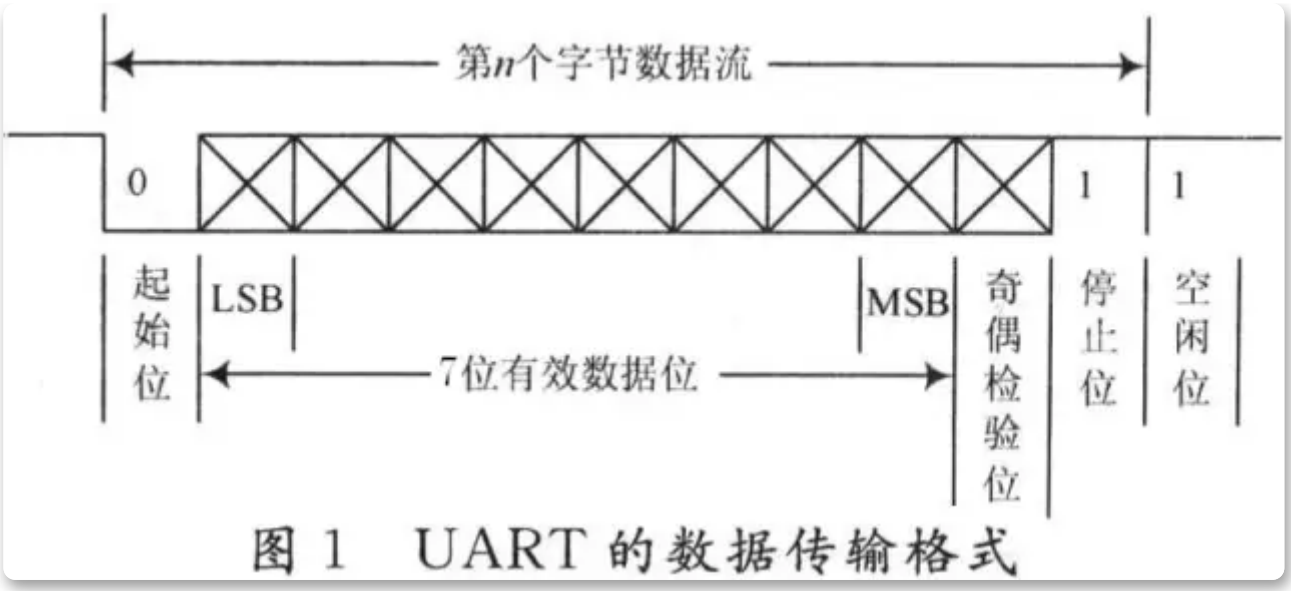


图 1　UART 的数据传输格式

在这里插入图片描述

其中各位的意义如下：

- 起始位：先发出一个逻辑”0”信号，表示传输字符的开始。
- 数据位：可以是5~8位逻辑”0”或”1”。如ASCII码（7位），扩展BCD码（8位）。小端传输
- 校验位：数据位加上这一位后，使得“1”的位数应为偶数(偶校验)或奇数(奇校验)
- 停止位：它是一个字符数据的结束标志。可以是1位、1.5位、2位的高电平。
- 空闲位：处于逻辑“1”状态，表示当前线路上没有资料传送。

注：异步通信是按字符传输的，接收设备在收到起始信号之后只要在一个字符的传输时间内能和发送设备保持同步就能正确接收。

下一个字符起始位的到来又使同步重新校准（依靠检测起始位来实现发送与接收方的时钟自同步的）

关于RS-232、RS-422、RS-485等标准，大家可以参考文章《一篇文章了解什么是串口，UART、RS-232、RS-422、RS-485 》

4. Exynos4412　Uart

本文讨论UART 是基于Cortex-A9架构的Exynos4412 为例。

1）特性

- Exynos4412 中UART，有4 个独立的通道，每个通道都可以工作于中断模式或DMA 模式，即UART 可以发出中断或 DMA 请求以便在UART 、CPU 间传输数据。使用系统时

钟时，Exynos4412 的 UART 波特率可以达到 4Mbps 。每个UART通道包含两个FIFO用来接收和发送：

- 通道 0有 256 字节的发送 FIFO 和 256 字节的接收FIFO
- 通道 1、4有 64 字节的发送 FIFO 和 64 字节的接收FIFO
- 通道 2、3有 16 字节的发送FIFO 和 16 字节 的接收 FIFO 。

UART include:

- 波特率可以通过编程进行 。
- 红外接收/发送
- 每个通道支持停止位有 1位、 2位
- 数据位有 5、6、7或 8位

每个UART还包括

- 波特率发生器、发送器、接收器、控制逻辑组成。

2）Uart控制器

功能模块

Figure 28-1 illustrates the block diagram of UART.

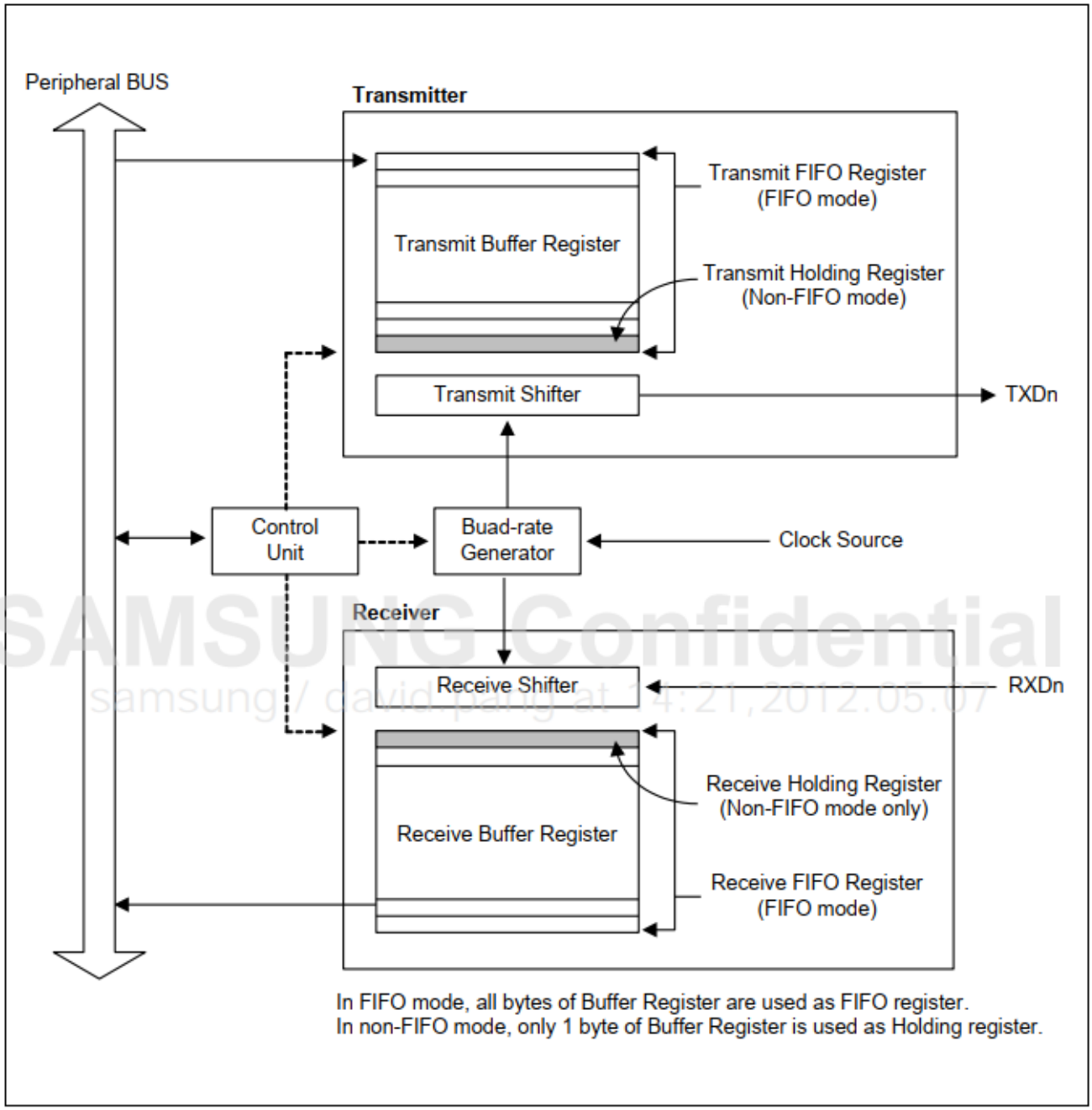


Figure 28-1 Block Diagram of UART

在这里插入图片描述

每个UART包含一个波特率产生器，发送器，接收器和一个控制单元，如上图所示：

- 发送数据 CPU 先将数据写入发送FIFO 中，然后 UART 会自动将FIFO 中的数据复制到“发送移位器”(Transmit Shifter)中，发送移位器将数据一位一位地发送到 TxDn 数据线上（根据设定的格式，插入开始位 、校验和停止)。
- 接收数据“移位器”(Receive Shifter)将 RxDn 数据线上的数据一位一位的接收进来，然后复制到FIFO 中， CPU即可从中读取数据。

UART是以异步方式实现通信的，其采样速度由波特率决定，波特率产生器的工作频率可以由PCLK（外围设备频率），FCLK/n（CPU工作频率的分频），UEXTCLK（外部输入时钟）

三个时钟作为输入频率，波特率设置寄存器是可编程的，用户可以设置其波特率决定发送和接收的频率。

发送器和接收器包含了64Byte的FIFO和数据移位器。UART通信是面向字节流的，待发送数据写到FIFO之后，被拷贝到数据移位器（1字节大小）里，数据通过发送数据管脚TXDn发出。

同样道理，接收数据通过RXDn管脚来接收数据（1字节大小）到接收移位器，然后将其拷贝到FIFO接收缓冲区里。

（1）数据发送 发送的数据帧可编程的，它的一个帧长度是用户指定的，它包括一个开始位，5~8个数据位，一个可选的奇偶校验位和1~2个停止位，数据帧格式可以通过设置ULCONn寄存器来设置。发送器也可以产生一个终止信号，它是由一个全部为0的数据帧组成。在当前发送数据被完全传输完以后，该模块发送一个终止信号。在终止信号发送后，它可以继续通过FIFO（FIFO）或发送保持寄存器（NON-FIFO）发送数据。

（2）数据接收 同样接收端的数据也是可编程的，接收器可以侦测到溢出错误奇偶校验错误，帧错误和终止条件，每个错误都可以设置一个错误标志。• 溢出错误 ：在旧数据被读取到之前，新数据覆盖了旧数据 • 奇偶校验错误：接收器侦测到了接收数据校验结果失败，接收数据无效 • 帧错误 ：接收到的数据没有一个有效的停止位，无法判定数据帧结束 • 终止条件 ：RxDn接收到保持逻辑0状态持续长于一个数据帧的传输时间

（3）自动流控AFC（Auto Float Control） UART0和UART1支持有nRTS和nCTS的自动流控。在AFC情况下，通信双方nRTS和nCTS管脚分别连接对方的nCTS和nRTS管脚。通过软件控制数据帧的发送和接收。在开启AFC时，发送端接收发送前要判断nCTS信号状态，当接收到nCTS激活信号时，发送数据帧。该nCTS管脚连接对方nRTS管脚。接收端在准备接收数据帧前，其接收器FIFO有大于32个字节的空闲空间，nRTS管脚会发送激活信号，当其接收FIFO小于32个字节的空闲空间，nRTS必须置非激活状态。

Figure 28-2 illustrates the UART AFC interface.

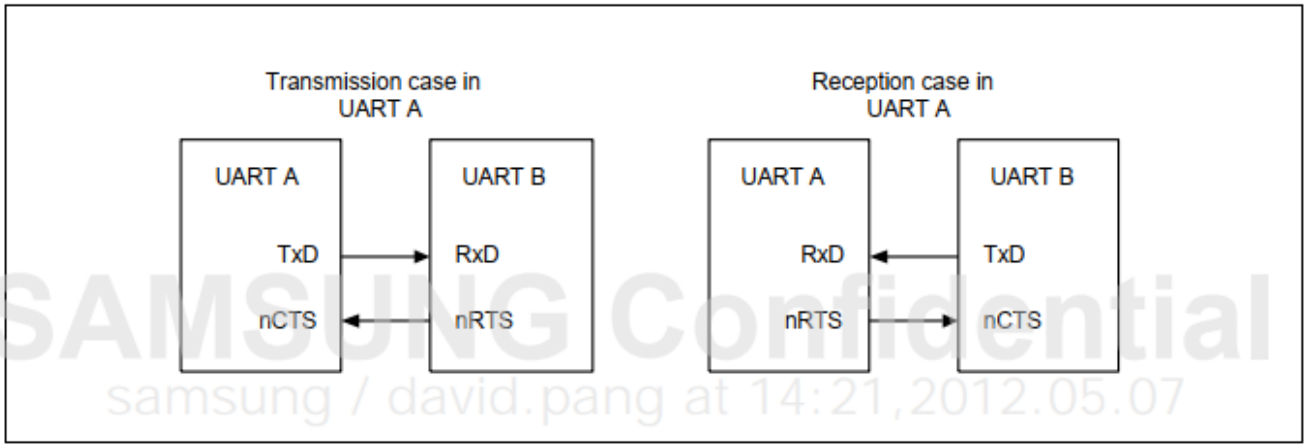


Figure 28-2 UART AFC Interface

3）选择时钟源

Figure 28-8 illustrates the input clock diagram for UART.

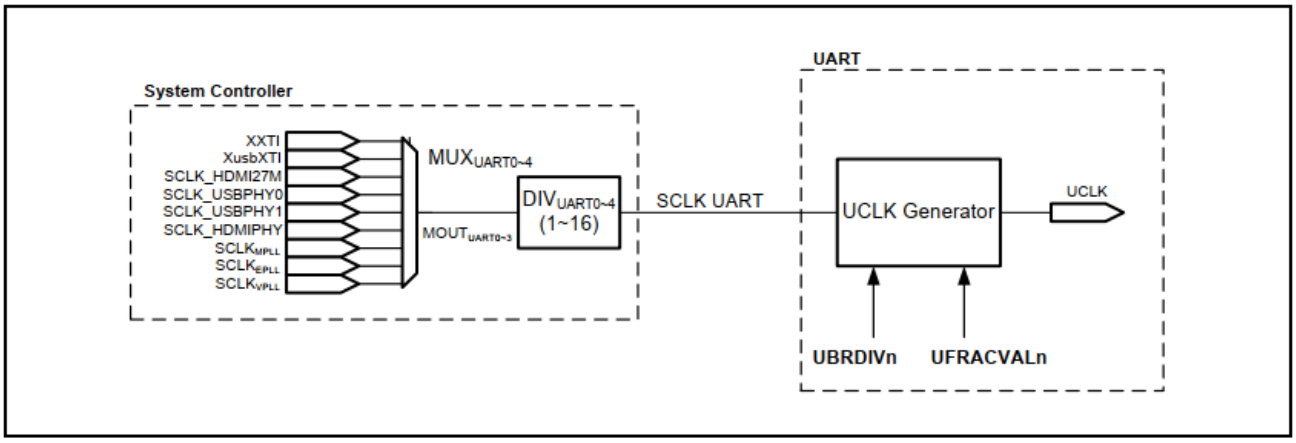


Figure 28-8 Input Clock Diagram for UART

Exynos 4412 SCP provides UART with a variety of clocks. Figure 28-8 illustrates that UART uses SCLK_UART clock, which is from clock controller. You can also select SCLK_UART from various clock sources. Refer to Chapter 7, Clock Controller, for more information.

Exynos4412 UART的时钟源有八种选择：XXTI 、XusbXTI 、SCLK_HDMI24M 、SCLK_USBPHY0 、 SCLK_HDMIPHY 、SCLKMPLL_USER_T 、SCLKEPLL 、SCLKVPLL ，由 CLK_SRC_PERIL0 寄存器控制。

选择好时钟源后，还可以通过 DIVUART0 ～4设置分频系数，由 CLK_DIV_PERIL0 寄存器控制。从分频器得到的时钟被称为SCLK UART 。

SCLK UART 经过上图中的“ UCLK Generator”后，得到UCLK ，它的频率就是UART的波特率。“ Generator UCLK Generator ”通过这 2个寄存器来设置：UBRDIVn(UART BAUD RATE DIVISOR) 、UFRACVALn 。

4）UART配置寄存器

28.6.1 Register Map Summary

- Base Address: 0x1380_0000, 0x1381_0000, 0x1382_0000, 0x1383_0000, 0x1384_0000

Register	Offset	Description	Reset Value
ULCONn	0x0000	Specifies line control	0x0000_0000
UCONn	0x0004	Specifies control	0x0000_3000
UFCONn	0x0008	Specifies FIFO control	0x0000_0000
UMCONn	0x000C	Specifies modem control	0x0000_0000
UTRSTATn	0x0010	Specifies Tx/Rx status	0x0000_0006
UERSTATn	0x0014	Specifies Rx error status	0x0000_0000
UFSTATn	0x0018	Specifies FIFO status	0x0000_0000
UMSTATn	0x001C	Specifies modem status	0x0000_0000
UTXHn	0x0020	Specifies transmit buffer	Undefined
URXHn	0x0024	Specifies receive buffer	0x0000_0000
UBRDIVn	0x0028	Specifies baud rate divisor	0x0000_0000
UFRACVALn	0x002C	Specifies divisor fractional value	0x0000_0000
UINTPn	0x0030	Specifies interrupt pending	0x0000_0000
UINTSPn	0x0034	Specifies interrupt source pending	0x0000_0000
UINTMn	0x0038	Specifies interrupt mask	0x0000_0000

在这里插入图片描述

ULCONn

28.6.1.1 ULCONn (n = 0 to 4)

- Base Address: 0x1380_0000, 0x1381_0000, 0x1382_0000, 0x1383_0000, 0x1384_0000
- Address = Base Address + 0x0000, Reset Value = 0x0000_0000

Name	Bit	Type	Description	Reset Value
RSVD	[31:7]	–	Reserved	0
Infrared Mode	[6]	RW	Determines whether to use the infra-red mode. 0 = Normal mode operation 1 = Infra-red Tx/Rx mode	0
Parity Mode	[5:3]	RW	Specifies the type of parity that UART generates and checks during UART transmit and receive operation. 0xx = No parity 100 = Odd parity 101 = Even parity 110 = Parity forced/ checked as 1 111 = Parity forced/ checked as 0	
Number of Stop Bit	[2]	RW	Specifies how many stop bits UART uses to signal end-of-frame signal. 0 = One stop bit per frame 1 = Two stop bit per frame	0
Word Length	[1:0]	RW	Indicates the number of data bits UART transmits or receives per frame. 00 = 5 bits 01 = 6 bits 10 = 7 bits 11 = 8 bits	0

在这里插入图片描述

- bite［6］ 红外模式 选择串口0是否使用红外模式：0 = 正常通信模式 1 = 红外通信模式

- bite［5:3］ 校验模式 设置串口0在数据接收和发送时采用的校验方式：0xx = 无校验 100 = 奇校验 101 = 偶校验 110 = 强制校验/检测是否为1 111 = 强制校验/检测是否为0

- ［2］停止位 设置串口0停止位数：0 = 每个数据帧一个停止位 1 = 每个数据帧二个停止位

- ［1:0］数据位 设置串口0数据位数：00 = 5个数据位 01 = 6个数据位 10 = 7个数据位 11 = 8个数据位

该寄存器我们通用的配置是：

```
ULCON2 = 0x3; //Normal mode, No parity,One stop bit,8 data bits
```

UCONn

28.6.1.2 UCONn (n = 0 to 4)

- Base Address: 0x1380_0000, 0x1381_0000, 0x1382_0000, 0x1383_0000, 0x1384_0000
- Address = Base Address + 0x0004, Reset Value = 0x0000_0000

Name	Bit	Type	Description	Reset Value
RSVD	[31:24]	–	Reserved	0
RSVD	[23]	WO	Reserved	0
Tx DMA Burst Size	[22:20]	RW	Tx DMA Burst Size It is the data transfer size of one DMA transaction. Tx DMA request triggers the DMA transaction. You must program the DMA program to transfer the same data size as this is the value for a single Tx DMA request. 000 = 1 byte (Single) 001 = 4 bytes 010 = 8 bytes 011 = 16 bytes 100 = Reserved 101 = Reserved 110 = Reserved 111 = Reserved	0
RSVD	[19]	WO	Reserved	0
Rx DMA Burst Size	[18:16]	RW	Rx DMA Burst Size It is the data transfer size of one DMA transaction. Rx DMA request triggers the DMA transaction. You must program the DMA program to transfer the same data size as this is the value for a single Rx DMA request. 000 = 1 byte (Single) 001 = 4 bytes 010 = 8 bytes 011 = 16 bytes 100 = Reserved 101 = Reserved 110 = Reserved 111 = Reserved	0
Rx Timeout Interrupt Interval	[15:12]	RW	Rx Timeout Interrupt Interval Rx interrupt occurs if UART receives no data during 8 × (N + 1) frame time. The default value of this field is 3. It means that the timeout interval is 32 frame time.	0x3
Rx Time-out with empty Rx FIFO (4)	[11]	R/W	Enables Rx time-out feature when Rx FIFO counter is 0. This bit is valid only when UCONn[7] is 1. 0 = Disables Rx time-out feature when Rx FIFO is empty. 1 = Enables Rx time-out feature when Rx FIFO is empty.	0
Rx Time-out DMA suspend enable	[10]	R/W	Enables the suspension of Rx DMA FSM when Rx Time-out occurs. 0 = Disables suspension of Rx DMA FSM 1 = Enables suspension of Rx DMA FSM	0

Tx Interrupt Type	[9]	RW	Interrupt request type. (2) 0 = Pulse (UART requests interrupt when the Tx buffer is empty in the non-FIFO mode or when it reaches the trigger level of Tx FIFO in the FIFO mode.) 1 = Level (Interrupt is requested when Tx buffer is empty in the non-FIFO mode or when it reaches the trigger level of Tx FIFO in the FIFO mode.)	0
Rx Interrupt Type	[8]	RW	Interrupt request type. (2) 0 = Pulse (UART requests interrupt when instant Rx buffer receives data in the non-FIFO mode or when it reaches the trigger level of Rx FIFO in the FIFO mode.) 1 = Level (UART requests interrupt when Rx buffer receives data in the non-FIFO mode or when it reaches the trigger level of Rx FIFO in the FIFO mode.)	0
Rx Time Out Enable	[7]	RW	Enables/disables Rx time-out interrupts when you enable UART FIFO. The interrupt is a receive interrupt. 0 = Disables 1 = Enables	0
Rx Error Status Interrupt Enable	[6]	RW	Enables the UART to generate an interrupt upon an exception, such as a break, frame error, parity error, or overrun error during a receive operation. 0 = Does not generate receive error status interrupt. 1 = Generates receive error status interrupt.	0
Loop-back Mode	[5]	RW	To set this bit to 1 triggers the UART to enter the loop-back mode. This mode is for test purposes only. 0 = Normal operation 1 = Loop-back mode	0

Send Break Signal	[4]	RWX	To set this bit to 1 triggers UART to send a break during 1 frame time. This bit is automatically cleared after sending the break signal. 0 = Normal transmit 1 = Sends the break signal	0
Transmit Mode	[3:2]	RW	Determines which function is able to Write Tx data to the UART transmit buffer. 00 = Disables 01 = Interrupt request or polling mode 10 = DMA mode 11 = Reserved	00
Receive Mode	[1:0]	RW	Determines which function is able to Read data from UART receive buffer. 00 = Disables 01 = Interrupt request or polling mode 10 = DMA mode 11 = Reserved	00

- [15:12] FCLK分频因子 当UART0选择FCLK作为时钟源时，设置其FCLK的分频因子 UART0 工作时钟频率 = FCLK/ FCLK分频因子 + 6

- [11:10] UART时钟源选择 选择UART0的工作时钟PCLK，UEXTCLK，FCLK/n：00,10 = PCLK 01 = UEXTCLK 11 = FCLK/n 当选择FCLK/n作为UART0工作时钟时还要做其它设置，具体请读者自行查看硬件手册

- [9] 发送数据中断产生类型 设置UART0中断请求类型，在非FIFO传输模式下，一旦发送数据缓冲区为空，立即产生中断信号，在FIFO传输模式下达到发送数据触发条件时立即产生中断信号：0 = 脉冲触发 1 = 电平触发

- [8] 接收数据中断产生类型 设置UART0中断请求类型，在非FIFO传输模式下，一旦接收到数据，立即产生中断信号，在FIFO传输模式下达到接收数据触发条件时立即产生中断信号：0 = 脉冲触发 1 = 电平触发

- [7] 接收数据超时 设置当接收数据时，如果数据超时，是否产生接收中断：0 = 不开启超时中断 1 = 开启超时中断 10 = 7个数据位 11 = 8个数据位

- [6] 接收数据错误中断 设置当接收数据时，如果产生异常，如传输中止，帧错误，校验错误时，是否产生接收状态中断信号：0 = 不产生错误状态中断 1 = 产生错误状态中断

- [5] 回送模式 设置该位时UART会进入回送模式，该模式仅用于测试 0 = 正常模式 1 = 回送模式

- [4] 发送终止信号 设置该位时，UART会发送一个帧长度的终止信号，发送完毕后，该位自动恢复为0 0 = 正常传输 1 = 发送终止信号

- [3:2] 发送模式 设置采用哪个方式执行数据写入发送缓冲区 00 = 无效 01 = 中断请求或查询模式 10 = DMA0请求

- [1:0] 接收模式 设置采用哪个方式执行数据写入接收缓冲区 00 = 无效 01 = 中断请求或查询模式 10 = DMA0请求

该寄存器通用配置为：

```
UCON2 = 0x5;  //Interrupt request or polling mode
```

一般裸机情况下，采用轮询模式。

UTRSTATn

UTRSTAT n寄存器用来表明数据是否已经发送完毕、是否已经接收到数据，格式如下图所示，上面说的“缓冲区”，其实就是下图中的 FIFO ，不使用 FIFO 功能时可以认为其深度为 1。

当我们读取数据时，就轮询检查bit[0]置1之后，然后再从URXHn寄存器读取数据；当我们读取数据时，就轮询检查bit[1]置1之后，然后再向UTXHn寄存器写入数据来发送数据；

Transmitter empty	[2]	R	This bit is automatically set to 1 when the transmit buffer has no valid data to transmit, and the transmit shift is empty. 0 = Not empty 1 = Transmitter (includes transmit buffer and shifter) empty	1

在这里插入图片描述

Name	Bit	Type	Description	Reset Value
Transmit buffer empty	[1]	R	This bit is automatically set to 1 when transmit buffer is empty. 0 = Buffer is not empty 1 = Buffer is empty (in non-FIFO mode, it requests interrupt or DMA). In FIFO mode, it requests interrupt or DMA, when the trigger level of Tx FIFO is set to 00 (Empty). When UART uses FIFO, check for Tx FIFO Count bits and Tx FIFO Full bit in UFSTAT instead of this bit.	1
Receive buffer data ready	[0]	R	It automatically sets this bit to 1 when receive buffer contains valid data, which is received over the RXDn port. 0 = Buffer is empty 1 = Buffer has a received data (In Non-FIFO mode, it requests interrupt or DMA) When UART uses the FIFO, check for Rx FIFO Count bits and Rx FIFO Full bit in UFSTAT instead of this bit.	0

在这里插入图片描述

UTXHn寄存器(UART TRANSMIT BUFFER REGISTER)

CPU 将数据写入这个寄存器， UART即会将它保存到缓冲区中，并自动发送出去。

URXHn寄存器(UART RECEIVE BUFFER REGISTER)

当 UART 接收到数据时，读取这个寄存器，即可获得数据。

UFRACVALn 计算波特率

28.6.1.12 UFRACVALn (n = 0 to 4)

- Base Address: 0x1380_0000, 0x1381_0000, 0x1382_0000, 0x1383_0000, 0x1384_0000
- Address = Base Address + 0x002C, Reset Value = 0x0000_0000

Name	Bit	Type	Description	Reset Value
RSVD	[31:4]	–	Reserved	0
UFRACVALn	[3:0]	RW	Determines the fractional part of Baud-rate divisor.	0x0

在这里插入图片描述

根据给定的波特率、所选择时钟源频率，可以通过以下公式计算 UBRDIVn 寄存器（n 为 0～4，对应 5个 UART 通道)的值。

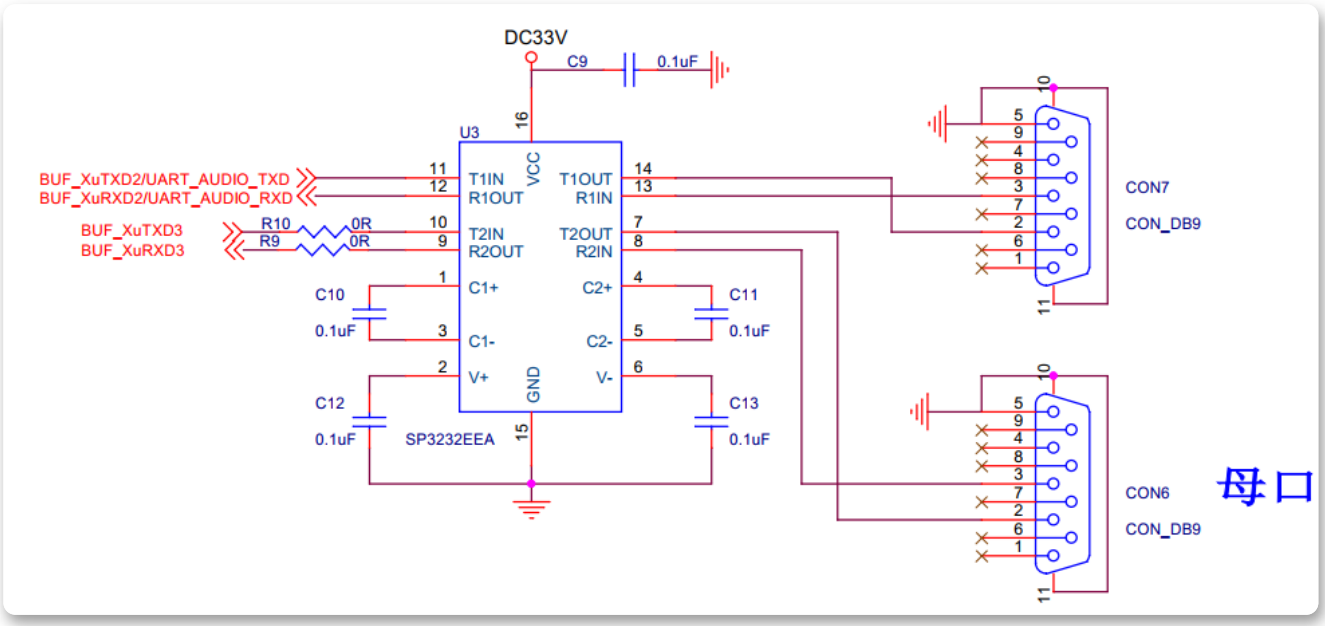
```
UBRDIVn = (int)( UART clock / ( buad rate x 16) ) - 1
```

上式计算出来的 UBRDIVn 寄存器值不一定是整数， UBRDIVn 寄存器取其整数部分，小部分由 UFRACVALn 寄存器设置， UFRACVALn 寄存器的引入，使产生波特率更加精确。「【举例】」当UART clock为100MHz时，要求波特率为115200 bps，则：

```
100000000/(115200 x 16) - 1 = 54.25 - 1 = 53.25
    UBRDIVn = 整数部分 = 53
    UFRACVALn/16 = 小数部分 = 0.25
    UFRACVALn = 4
```

5）电路图

外设电路图：



在这里插入图片描述

SP3232EEA 用来将TTL电平转换成RS232电平。我们使用的是COM2。

外设与核心板连接电路图

在这里插入图片描述

可见UART的收发引脚连接到了GPA上，打开exynos4412芯片手册：

GPA1CON[3]	[15:12]	RW	0x0 = Input 0x1 = Output 0x2 = UART_2_RTSn 0x3 = I2C_3_SCL 0x4 to 0xE = Reserved 0xF = EXT_INT2[3]	0x00
GPA1CON[2]	[11:8]	RW	0x0 = Input 0x1 = Output 0x2 = UART_2_CTSn 0x3 = I2C_3_SDA 0x4 to 0xE = Reserved 0xF = EXT_INT2[2]	0x00
GPA1CON[1]	[7:4]	RW	0x0 = Input 0x1 = Output 0x2 = UART_2_TXD 0x3 = Reserved 0x4 = UART_AUDIO_TXD 0x5 to 0xE = Reserved 0xF = EXT_INT2[1]	0x00
GPA1CON[0]	[3:0]	RW	0x0 = Input 0x1 = Output 0x2 = UART_2_RXD 0x3 = Reserved 0x4 = UART_AUDIO_RXD 0x5 to 0xE = Reserved 0xF = EXT_INT2[0]	0x00

在这里插入图片描述

我们只需要将GPA1 的低8位设置为0x22。

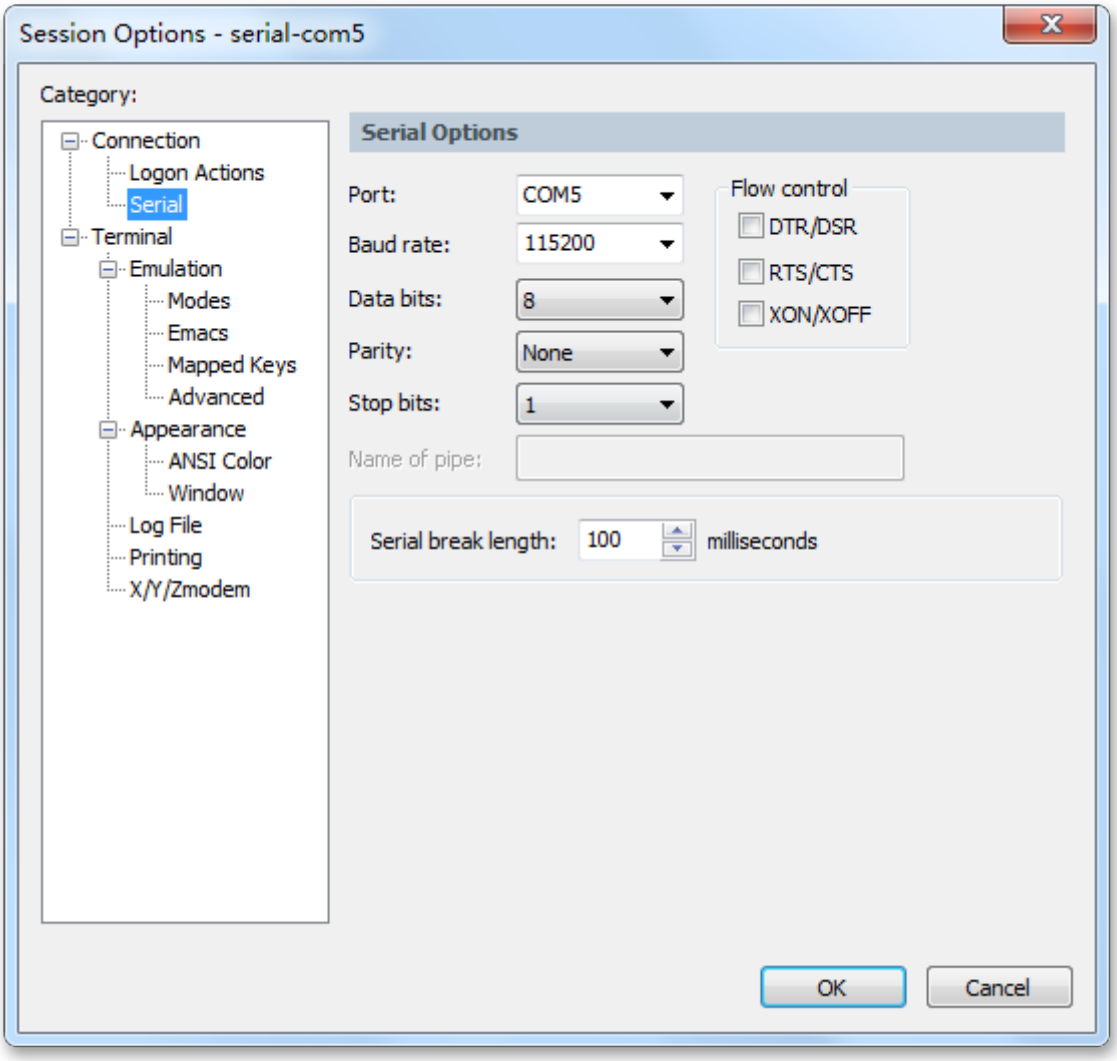
6.实例代码

裸机代码，主要实现uart_init（）、putc（）、getc（）这三个函数。

uart_init（）

该函数主要配置UART的，波特率115200，数据位：8，奇偶校验位：0，终止位：1，不设置流控。

如下图：是运行在windows下常用的串口工具配置信息，配置信息必须完全一致。



在这里插入图片描述

putc ()

该函数是向串口发送一个数据data，他的实现逻辑就是轮询检查寄存器UART2.UTRSTAT2，判断其bite【1】是否置1，如果置1，则向UART2.UTXH2存入要发送的数据即可。

getc ()

该函数是从串口接收一个数据data，他的实现逻辑就是轮询检查寄存器UART2.UTRSTAT2，判断其bite【0】是否置1，如果置1，说明数据准备好，则可以从寄存器UART2.URXH2取出数据。

代码

```
/*
 * UART2
 */
```

```
typedef struct {
    unsigned int ULCON2;
    unsigned int UCON2;
    unsigned int UFCON2;
    unsigned int UMCON2;
    unsigned int UTRSTAT2;
    unsigned int UERSTAT2;
    unsigned int UFSTAT2;
    unsigned int UMSTAT2;
    unsigned int UTXH2;
    unsigned int URXH2;
    unsigned int UBRDIV2;
    unsigned int UFRACVAL2;
    unsigned int UINTP2;
    unsigned int UINTSP2;
    unsigned int UINTM2;
}uart2;
#define UART2 ( * (volatile uart2 *)0x13820000 )
/* GPA1 */
typedef struct {
    unsigned int CON;
    unsigned int DAT;
    unsigned int PUD;
    unsigned int DRV;
    unsigned int CONPDN;
    unsigned int PUDPDN;
}gpa1;
#define GPA1 (* (volatile gpa1 *)0x11400020)
void uart_init()
{ /*UART2 initialize*/
 GPA1.CON = (GPA1.CON & ~0xFF ) | (0x22); //GPA1_0:RX;GPA1_1:TX
 UART2.ULCON2 = 0x3; //Normal mode, No parity,One stop bit,8 data bits
 UART2.UCON2 = 0x5;  //Interrupt request or polling mode
 //Baud-rate : src_clock:100Mhz
 UART2.UBRDIV2 = 0x35;
 UART2.UFRACVAL2 = 0x4;
}
void putc(const char data)
{ while(!(UART2.UTRSTAT2 & 0X2));
 UART2.UTXH2 = data;
 if (data == '\n')
```

```
    putc('\r');
}
char getc(void)
{ char data;
 while(!(UART2.UTRSTAT2 & 0x1));
 data = UART2.URXH2;
 if ((data == '\n')||(data == '\r'))
 {
   putc('\n');
   putc('\r');
 }else
   putc(data);
 return data;
}
```

puts/gets

```
void puts(const  char  *pstr)
{ while(*pstr != '\0')
  putc(*pstr++);
}
void gets(char *p)
{ char data;
 while((data = getc())!= '\r')
 {  if(data == '\b')
  {p--;
  }
  *p++ = data;
 }
 if(data == '\r')
 *p++ = '\n';
 *p = '\0';
}
```

7.如何裸机程序可以支持printf函数

首先看下文件的目录结构：

代码架构

老规矩，关注，后台回复【armprintf】，就可以得到代码。

这里我们只贴出部分文件的代码。

「cpu/start.s」改文件主要是实现异常向量表，实现各个模式的栈初始化

```
.text
.global _start
_start:
  b  reset
  ldr  pc,_undefined_instruction
  ldr  pc,_software_interrupt
  ldr  pc,_prefetch_abort
  ldr  pc,_data_abort
  ldr  pc,_not_used
  ldr  pc,=irq_handler
  ldr  pc,_fiq

_undefined_instruction: .word  _undefined_instruction
_software_interrupt: .word  _software_interrupt
_prefetch_abort:  .word  _prefetch_abort
_data_abort:   .word  _data_abort
_not_used:    .word  _not_used
_irq:     .word  irq_handler
_fiq:     .word  _fiq
```

```
reset:

 ldr r0,=0x40008000
 mcr p15,0,r0,c12,c0,0  @ 协处理器指令设置异常向量表地址


init_stack:
  ldr  r0,stacktop          /*get stack top pointer*/

 /********svc mode stack********/
  mov  sp,r0
  sub  r0,#128*4           /*512 byte  for irq mode of stack*/
 /****irq mode stack**/
  msr  cpsr,#0xd2
  mov  sp,r0
  sub  r0,#128*4           /*512 byte  for irq mode of stack*/
 /***fiq mode stack***/
  msr  cpsr,#0xd1
  mov  sp,r0
  sub  r0,#0
 /***abort mode stack***/
  msr  cpsr,#0xd7
  mov  sp,r0
  sub  r0,#0
 /***undefine mode stack***/
  msr  cpsr,#0xdb
  mov  sp,r0
  sub  r0,#0
   /*** sys mode and usr mode stack ***/
  msr  cpsr,#0x10
  mov  sp,r0              /*1024 byte  for user mode of stack*/

  b  main @跳转到c语言的main函数

 .align 4

 /****  swi_interrupt handler  ****/


 /****  irq_handler  ****/
irq_handler:

 sub  lr,lr,#4
 stmfd sp!,{r0-r12,lr}
 .weak do_irq   @该函数可以没有定义
 bl do_irq  @跳转到中断入口
 ldmfd sp!,{r0-r12,pc}^

stacktop:    .word   stack+4*512 @定义栈顶
.data
```

```
stack:  .space  4*512  @分配一块栈空间
```

「lib/printf.c」

该文件主要实现打印函数printf一些格式控制，一些字符串转换算数运算需要借助头文件ctype.h、stdarg.h中一些宏。其中vsprintf 具体的实现我们就不再详解，有兴趣读者自行研究。

```
……
void printf (const char *fmt, ...)
{
 va_list args;
 unsigned int i;
 char printbuffer[100];
 va_start (args, fmt);

 /* For this to work, printbuffer must be larger than
  * anything we ever want to print.
  */
 i = vsprintf (printbuffer, fmt, args);//对输入的参数进行格式整理
 va_end (args);
 puts (printbuffer); //调用上一章我们封装的puts函数实现向串口打印书字符串
}
```

「main.c」该文件可以直接调用printf（）函数来打印信息了。

```
void  delay_ms(unsigned int num)
{
    int i,j;
    for(i=num; i>0;i--)
 for(j=1000;j>0;j--)
  ;
}
/*
 *  裸机代码，不同于LINUX 应用层，  一定加循环控制
 */

int main (void)
{
 int i = 0;
```

```
 while (1) {
  printf("aaaaaaaaaaaaa\n");
  delay_ms(500);
 }
   return 0;
}
```

「Makefile」

```
CROSS_COMPILE = arm-none-eabi-
NAME =gcd
CFLAGS=-mfloat-abi=softfp -mfpu=vfpv3 -mabi=apcs-gnu -fno-builtin  -fno-bui
LD = $(CROSS_COMPILE)ld
CC = $(CROSS_COMPILE)gcc
OBJCOPY = $(CROSS_COMPILE)objcopy
OBJDUMP = $(CROSS_COMPILE)objdump
OBJS=./cpu/start.o ./driver/uart.o  \
        ./driver/_udivsi3.o ./driver/_divsi3.o ./driver/_umodsi3.o main.o .
#=============================================================================
all:  $(OBJS)
 $(LD)  $(OBJS) -T map.lds -o $(NAME).elf
 $(OBJCOPY)  -O binary  $(NAME).elf $(NAME).bin
 $(OBJDUMP) -D $(NAME).elf > $(NAME).dis
%.o: %.S
 $(CC) $(CFLAGS) -c -o  $@ $<
%.o: %.s
 $(CC) $(CFLAGS) -c -o  $@ $<
%.o: %.c
 $(CC) $(CFLAGS) -c -o  $@ $<
clean:
 rm -rf $(OBJS) *.elf *.bin *.dis *.o
```

Makefile、map.lds 参考《7. 从0开始学ARM-GNU伪指令、代码编译，lds使用》

后续我们都会在这个模板上来编写其他硬件的驱动代码。

推荐阅读

【1】【从0学ARM】你不了解的ARM处理异常之道

【2】为什么使用结构体效率比较高？ **必读**

【3】9. 基于Cortex-A9 LED汇编、C语言驱动编写**必读**

【4】一文包你学会网络数据抓包**必读**

【5】10. 基于Cortex-A9的pwm详解 **必读**

【6】11. 基于ARM Cortex-A9中断详解 **必读**

进群，请加一口君个人微信，带你嵌入式入门进阶。

收录于合集 #从0学arm 27

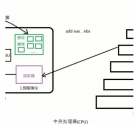
上一篇
11. 基于ARM Cortex-A9中断详解

下一篇
13.一文搞懂Cortex-A9 RTC

阅读原文

喜欢此内容的人还喜欢

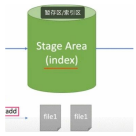
详细讲解MMU——为什么嵌入式linux没他不行？
一口Linux



我的第一本书终于要印刷出版了！

一口Linux

Git 基本原理介绍



一口Linux