# 第11章 异步串口原理与串口通信实现

实现异步串行通信是计算机系统最基本的外设功能之一，也是读者学习更复杂计算机外设的基础。本章将详细介绍RS-232标准以及龙芯1B处理器内建的异步串行收发器模块原理，并通过设计实例说明该模块的使用方法。

本章主要内容包括RS-232协议规范、龙芯1B处理器中UART模块原理、PuTTY工具的下载和安装，以及串口重定位的设计与实现。通过这些内容的介绍，读者能够掌握异步串行通信的原理，并能够掌握c语言中重定位输入和输出函数的方法。

## 11.1 RS-232协议规范

RS-232是美国电子工业联盟（ElectronicI Industries Association，EIA）制定的串行数据通信的接口标准，原始编号全称是EIA-RS-232（简称232，RS-232）。它被广泛用于计算机串行接口外设连接。

在RS-232C标准中，232是标识号，C代表RS-232的第三次修改（1969年），在这之前，还有RS-232B、RS-232A。

目前的最新版本是由EIA所发布的TIA-232-F，它同时也是美国国家标准 ANSI /TIA-232-F-1997 (R2002)，此标准于2002年确认。在 1997 年由 TIA/EIA 发布当时的编号则是 TIA/EIA-232-F 与 ANSI/TIA/EIA-232-F-1997。在此之前的版本是 TIA/EIA-232-E。

RS-232标准规定了传输数据所使用的连接[电缆](http://zh.wikipedia.org/wiki/%E9%9B%BB%E7%BA%9C)和[机械](http://zh.wikipedia.org/wiki/%E6%A9%9F%E6%A2%B0)、[电气](http://zh.wikipedia.org/wiki/%E9%9B%BB%E6%B0%A3)特性、信号功能及传送过程。基于这个标准基础，派生出其他电气标准，包括[EIA-RS-422-A](http://zh.wikipedia.org/wiki/EIA-422)、[EIA-RS-423A](http://zh.wikipedia.org/w/index.php?title=EIA-423&action=edit&redlink=1)、[EIA-RS-485](http://zh.wikipedia.org/wiki/RS-485)。

目前，在[PC](http://zh.wikipedia.org/wiki/PC)机上的COM1、COM2接口就是RS-232C接口。

**注：**在最新的计算机和笔记本电脑中，均不再提供这种接口，用户必须通过USB转串口电缆，在计算机和笔记本电脑上虚拟出一个RS-232串行接口。

由于 RS-232C 的重大影响，即使自 IBM [PC/AT](http://zh.wikipedia.org/wiki/PC/AT) 开始改用 9 针连接器起，目前几乎不再使用 RS-232 中规定的 25 针连接器，但大多数人仍然普遍使用 RS-232C 来代表此接口。

### 11.1.1 RS-232传输特点

在RS-232标准中，有下面显著的特点：

（1）字符是按一个比特接着另一个比特的方式，使用一根信号线进行传输。这就是通常所说的串行方式传输数据，这种传输方式的优点是传输线少，连线简单，传送距离较远。

（2）对于信源（发送方）来说，需要将原始的并行数据封装（也称为打包），然后在转换成一位一位的串行比特流数据进行发送；对于信宿（目的方）来说，当接收到串行比特流数据后，对接收到的数据进行“拆包”，即从所接收到的串行比特流数据中找出原始比特流数据的信息，将原始的比特流数据转换成并行数据，如图11.1所示。

（3）在从信源（发送方）发送数据给信宿（目的方）的时候，并不需要传输时钟信号。当信宿接收到串行数据的时候，会使用信宿本地的时钟对接收到的数据进行采样和解码，然后将数据恢复出来。

（4）此外，通过RS-232在传送数据时，并不需要额外使用一个信号来传送同步信息，只需在数据头部（header）和尾部（end）加上识别标志，就能将数据从“信源”正确地传送到“信宿”。

并行转串行

原始字节数据

打包后的数据

检

测

到

比

特

流

数

据

得到并行有效数据

本地时钟 信源

本地时钟 信宿

串行电缆

图11.1 异步串行通信原理

提

取

有

效

的

位

数

据

串

行

转

并

行

在计算机中，将实现RS-232通信功能的专用芯片，典型的8251和16550，称为通用异步接收发送器 (Universal Asynchronous Receiver Transmitter，UART)。

### 11.1.2 RS-232数据传输格式

在RS-232中，使用的编码格式是异步起停数据格式，如图11.2所示。在该数据格式中：

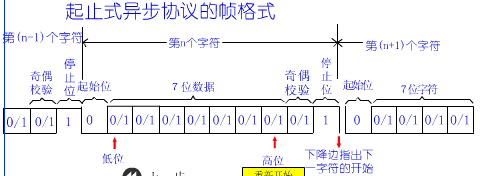


图11.2 RS-232数据格式

（1）首先有一个逻辑‘0’标识的起始位，该位标识新的一帧数据的开始。

（2）在起始位后面紧跟以串行格式存在的5~8个数据位，数据的起始位对应于原始字节数据的LSB，数据的结束位对应于原始字节数据的MSB。

（3）最后一个数据位后面跟随奇偶校验位（可选）。是否需要奇偶校验位可以在发送数据的时候设置，且发送方和接收方使用相同的奇偶校验设置。

（4）在可选的奇偶校验位后面跟着以逻辑‘1’标识的1~2个停止位。发送方在发送数据之前设置停止位的个数，且发送方和接收方使用相同的停止位设置。

从上面的描述可知，在RS-232的数据格式中，发送一个有效字符数据字节（8位表示）至少需要10个比特位（即至少需要一个数据位和一个停止位）。

在该数据格式中，每一位持续的时间与发送数据的时钟频率有关，即以发送方以多快的速度发送一个比特位。通常，将发送数据的时钟称为波特率时钟，用波特率表示，即每秒发送比特位的个数。在基于RS-232的异步串行通信中，要求发送方和接收方使用相同的波特率时钟。

### 11.1.3 RS-232电气标准

在RS-232标准中，分别定义了逻辑‘1’和逻辑‘0’的电压范围。

（1）逻辑‘1’的电压范围为－15~－3V；

（2）逻辑‘0’的电压范围为+3~+15V。

在RS-232中，接近零的电平是无效的。显然，RS-232中对逻辑‘0’和逻辑‘1’的定义与数字逻辑对逻辑‘1’和逻辑‘0’的定义不同。为了让遵守RS-232电气标准的电平信号与标准数字逻辑电平信号进行连接，就需要执行电气标准转换，即把满足数字逻辑标准的TTL/CMOS电平转换为RS-232电平，以及将RS-232电平转换为TTL/CMOS电平。典型的，美信（maxim）公司的MAX232芯片可以实现TTL/CMOS电平与RS-232电平之间的双向转换，如图11.3所示。

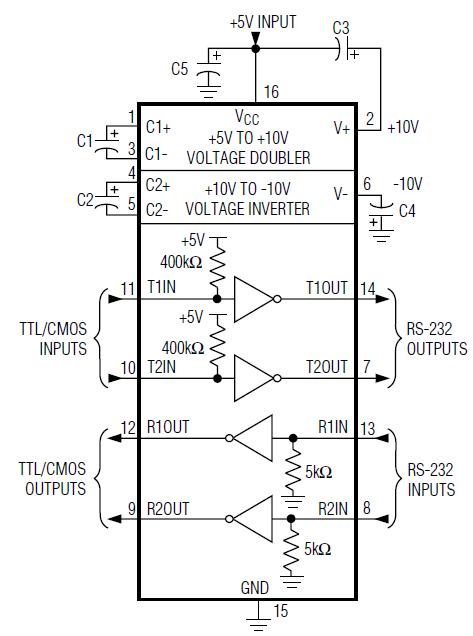
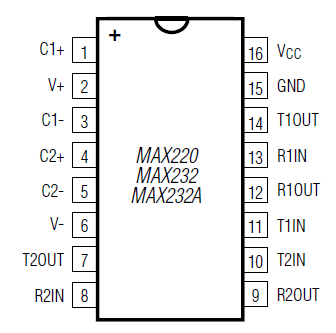


图11.3 电平转换芯片-在TTL/COMS与RS-232之间进行电平转换

TTL/CMOS

输入

TTL/CMOS

输出

RS-232

输出

RS-232

输入

+5V到+10V

电压倍增器

+10V到-10V

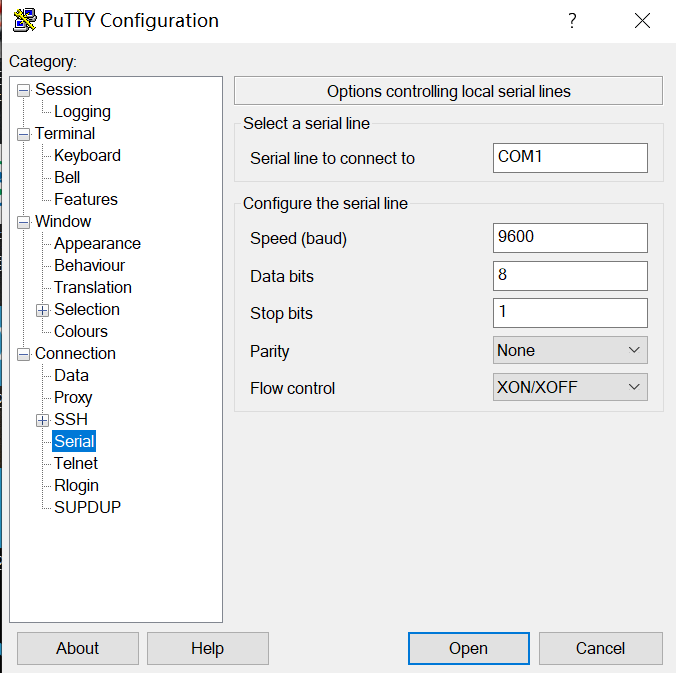
电压反相器

当把龙芯1B处理器的LVTTL/LVCMOS的引脚连接到MAX232芯片上，就可以实现通过串口电缆与其他设备进行通信。

### 11.1.4 RS-232参数设置

打开PuTTY软件，单击左侧的Serial条项，在右侧窗口中可以看到串口参数设置界面，如图11.4所示。在该界面中，可以设置[波特率](http://zh.wikipedia.org/wiki/%E6%B3%A2%E7%89%B9%E7%8E%87)（Speed，band）、奇偶校验（Parity）、停止位（Stop bits），以及流量控制（Flow control）等参数。下面对这些参数的含义进行简单介绍。

图11.4 RS-232通信参数设置



（1）波特率

它是指将数据从一个设备发送到另一个设备的速度，使用每秒钟发送位的个数来度量，单位为波特率（bits per second，bps）。典型的，可选择的波特率有300、1200、2400、9600、19200、115200 等。

（2）奇偶校验

奇偶校验是用来验证接收数据的正确性。一般不使用奇偶校验，如果使用，那么既可以选择设置为奇校验也可以选择设置为偶校验。在偶校验中，要求所有发送数据的位（包括校验位）中‘1’的个数是偶数。根据这个校验标准，将奇偶校验位置‘1’或置‘0’；在奇校验中，要求所有发送数据的位（包括校验位在内）‘1’的个数是奇数。根据这个校验标准，将校验位置‘1’或置‘0’；

（3）停止位

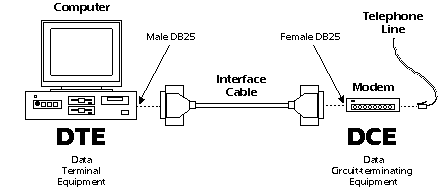
停止位是在发送完数据的最后一位或奇偶校验位之后发送的，用于帮助实现串口通信的接收方实现硬件的重新同步。例如，在传输8位原始8位数据“11001010”时，数据的前后就需加入起始位（以逻辑‘0‘表示）以及停止位（以逻辑’1‘表示）。需要注意，起始位固定为一位，而停止位可以是1位、1.5位或2位。停止位由发送方确定，并且发送方和接收方需要设置相同的停止位。

（4）流量控制

当需要发送[握手信号](http://zh.wikipedia.org/wiki/%E6%8F%A1%E6%89%8B%E4%BF%A1%E5%8F%B7)或对数据完整性进行检测时就需要额外的信号进行协助，这些额外的信号包括RTS/CTS和DTR/DSR，通常这些信号用于帮助RS-232实现硬件流量控制，即在需要时就要在使用RS-232的发送方和接收方连接这些信号线，以保证通信过程的可靠性。一般情况下为了简化硬件信号的连接和降低控制的复杂度，不使用用于硬件流量控制的信号线。

### 11.1.5 RS-232连接器

RS-232设计之初是用来连接调制解调器做传输之用，因此它的引脚定义通常也和调制解调器传输有关。RS-232的设备可以分为数据终端设备（data Terminal Equipment，DTE，如PC）和数据通信设备（data Communication Equipment, DCE）两类，这种分类定义了不同的线路用来发送和接受信号。一般来说，计算机和终端设备有DTE连接器，调制解调器和打印机有DCE连接器，如图11.5所示。



计算机

公头DB25

母头DB25

电话线

数据终端

设备

数据通信

设备

接口电缆

调制解调器

图11.5 DTE和DCE设备的连接

RS-232指定了20个不同的信号连接，由25个D-sub（微型D类）引脚构成的DB-25连接器。很多设备只使用了其中的部分引脚，出于节省资金和空间的考虑不少机器采用较小的连接器，特别是9引脚的D-sub或者是DB-9型连接器广泛使用在绝大多数自IBM的AT机之后的PC机和其他许多设备上。DB-25和DB-9型的连接器在大部分设备上是母头（即插孔），但并不一定都是这样，有些设备上就是公头（即插针）。

图11.6 RS-232串口连接器-母头(female)和公头(male)

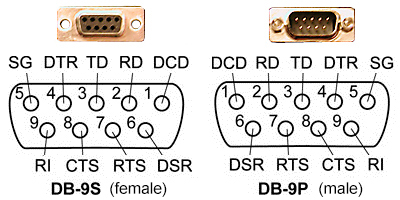


图11.6给出了DB-9连接器公头和母头连接器的信号定义顺序。每个信号的定义如表11.1所示。

表11.1 DB-9连接器信号定义

|  |  |  |
| --- | --- | --- |
| 引脚名字 | 序号 | 功能 |
| 公共接地（SG） | 5 | 地线 |
| 发送数据（TD/TXD） | 3 | 发送数据 |
| 接受数据（RD/RXD） | 2 | 接收数据 |
| 数据终端准备（Data Terminal Ready，DTR） | 4 | 终端设备通知调制解调器可以进行数据传输 |
| 数据准备好（Data Set Ready，DSR） | 6 | 调制解调器通知终端设备准备就绪 |
| 请求发送（Request To Send，RTS） | 7 | 终端设备要求调制解调器将数据提交 |
| 清除发送（Clear To Send，CTS） | 8 | 调制解调器通知终端设备可以传数据过来 |
| 数据载波检测（Carrier Detect，CD） | 1 | 调制解调器通知终端设备侦听到载波信号 |
| 振铃指示（Ring Indicator，RI） | 9 | 调制解调器通知终端设备有电话进来 |

## 11.2 龙芯1B处理器中UART模块原理

龙芯1B处理器集成的通用异步串行收发器（Universal Asynchronous Receiver/Transmitter, UART）模块兼容标准的16550A芯片。该UART模块的内部结构，如图11.7所示。

分频器

访问寄存器

模块

中断仲裁模块

Modem寄存器

接收FIFO

发送FIFO

接收模块接口

发送模块接口

总线接口

Modem接口

图11.7 龙芯1B处理器内建UART模块结构

### 11.2.1 UART寄存器组的基地址

在龙芯1B处理器中内建了12个可并行工作的UART接口，这些接口的内部寄存器完全相同，但是它们的基地址不同，如表11.2所示。

表11.2 UART接口寄存器的基地址

|  |  |
| --- | --- |
| 接口名字 | 基地址 |
| UART0 | 0xBFE4 0000 |
| UART0\_1 | 0xBFE4 1000 |
| UART0\_2 | 0xBFE4 2000 |
| UART0\_3 | 0xBFE4 3000 |
| UART1 | 0xBFE4 4000 |
| UART1\_1 | 0xBFE4 5000 |
| UART1\_2 | 0xBFE4 6000 |
| UART1\_3 | 0xBFE4 7000 |
| UART2 | 0xBFE4 8000 |
| UART3 | 0xBFE4 C000 |
| UART4 | 0xBFE6 C000 |
| UART5 | 0xBEF7 C000 |

### 11.2.2 UART寄存器组的功能

本节介绍每个UART模块中寄存器的定义及功能。

1）数据传输保持寄存器（Data Transfer Hold Register，DTHR）

该寄存器保存着要发送/接收的数据，其偏移地址为0x00，格式如表11.3所示，当复位时，该寄存器的内容为0x00。

表11.3 数据传输寄存器的格式

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位索引 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 读写属性 | W/R | | | | | | | |
| 名字 | Tx FIFO/Rx FIFO | | | | | | | |

注：当对该寄存器执行写操作时，写到该寄存器的内容为要发送的数据（TxFIFO）；当对该寄存器执行读操作时，读取该寄存器的内容为接收到的数据（Rx FIFO）。

2）中断使能寄存器（Interrupt Enable Register，IER）

该寄存器控制使能/屏蔽中断源，其偏移地址为0x01，格式如表11.4所示。当复位时，该寄存器的内容为0x00。

表11.4 中断使能寄存器的格式

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位索引 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 读写属性 | ― | ― | ― | ― | R/W | R/W | R/W | R/W |
| 名字 | ― | ― | ― | ― | IME | ILE | ITxE | IRxE |

表中：

（1）[7:4]：保留位。

（2）[3]：IME。Modem状态中断使能位。当该位为‘0’时，禁止该中断；当该位为‘1’时，使能该中断。

（3）[2]：ILE。接收器线路状态中断使能位。当该位为’0‘时，禁止该中断；当该位为’1‘时，使能该中断。

（4）[1]：ITxE。传输保存寄存器为空中断使能位。当该位为‘0‘时，禁止该中断；当该位为’1‘时，使能该中断。

（5）[0]：IRxE。接收有效数据中断使能位。当该位为‘0’时，禁止该中断；当该位为‘1‘时，使能该中断。

3）中断标识寄存器（Interrupt Indication Register，IIR）

该寄存器用于标识当前产生的中断，其偏移地址为0x02，格式如表11.5所示。当复位时，该寄存器的内容为0xc1。

表11.5 中断标识寄存器的格式

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位索引 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 读写属性 | ― | ― | ― | ― | R | | | R |
| 名字 | ― | ― | ― | ― | 中断源编码（见表10.7） | | | INTp |

表中：

（1）[7:4]：保留位。

（2）[3:1]：中断源编码字段。其含义如表11.6所示。

表11.6 中断源编码字段的含义

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 中断源编码 | | | 优先级 | 中断类型 | 中断源 | 中断复位控制 |
| 3 | 2 | 1 |
| 0 | 1 | 1 | 第一 | 接收线路状态 | 奇偶、溢出或帧错误，或打断中断 | 读LSR |
| 0 | 1 | 0 | 第二 | 接收到有效数据 | FIFO中的字符个数达到触发的值 | FIFO的字符个数低于触发的值 |
| 1 | 1 | 0 | 第二 | 接收超时 | 在FIFO中至少有一个字符，但在4个字符时间内没有任何操作，包括读和写操作 | 读接收FIFO |
| 0 | 0 | 1 | 第三 | 传输保存寄存器为空 | 传输保存寄存器为空 | 写数据到传输寄存器或者多IIR |
| 0 | 0 | 0 | 第四 | Modem状态 | CTS、DST、RI或DCD | 读MSR |

4）FIFO控制寄存器（FIFO Control Register，FCR）

该寄存器用于设置FIFO触发中断的门限，其偏移地址为0x02，格式如表11.7所示。当复位时，该寄存器的内容为0xc0。在UART模块中，FIFO的深度为16。

表11.7 FIFO控制寄存器的格式

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位索引 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 读写属性 | W | | ― | ― | ― | W | W | ― |
| 名字 | TL | | ― | ― | ― | Txset | Rxset | ― |

表中：

（1）[7:6]：TL。设置用于触发FIFO中断的门限。当该字段为“00”时，触发FIFO中断的门限为1个字节；当该字段为“01”时，触发FIFO中断的门限为4个字节；当该字段为“10”时，触发FIFO中断的门限为8个字节；当该字段为“11”时，触发FIFO中断的门限为14个字节。

（2）[5:3]：保留位。

（3）[2]：Txset。当该位置‘1’时，清除发送FIFO的内容，并复位其逻辑。

（4）[1]：Rxset。当该位置’1’时，清除接收FIFO的内容，并复位其逻辑。

（5）[0]：保留位。

注：中断标识寄存器和FIFO控制寄存器共用一个存储空间地址，实现时使用读/写操作来区分这两个寄存器。当执行写寄存器操作时，将值写入FIFO控制寄存器；当执行读寄存器操作时，将从中断标识寄存器读取中断状态标志。

5）线路控制寄存器（Line Control Register，LCR）

该寄存器用于设置线路的工作状态，其偏移地址为0x03，格式如表11.8所示。当复位时，该寄存器的内容为0x03。

表11.8 线路控制寄存器的格式

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位索引 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 读写属性 | R/W | R/W | R/W | R/W | R/W | R/W | R/W | |
| 名字 | DLAB | BCB | SPB | EPS | PE | SB | BEC | |

表中：

（1）[7]：DLAB。分频锁存器访问控制位。当该位为‘1’时，访问操作分频锁存器；当该位为‘0’时，访问操作正常寄存器。

（2）[6]：BCB。打断控制位。当该位为‘1’时，将串口的输出设置为0（打断状态）；当该位为‘0’时，正常操作。

（3）[5]：SPB。指定奇偶校验控制位。当该位为‘0’时，不用指定奇偶校验位；当该位为‘1’时，如果LCR[4]位是1则传输和检查奇偶校验位为‘0‘；如果LCR[4]位为0则传输和检查奇偶校验位为‘1‘。

（4）[4]：EPS。奇偶校验选择控制位。当该位为‘0‘时，在每个字符中有奇数个’1‘（包括数据和奇偶校验位）；当该位为’1‘时，在每个字符中有偶数个’1‘。

（5）[3]：PE。奇偶校验位使能控制位。当该位为‘0‘时，没有奇偶校验位；当该位为’1‘时，在输出时生成奇偶校验位，输入则判断奇偶校验位。

（6）[2]：SB。定义生成的停止位的个数。当该位为‘0‘时，1个停止位；当该位为’1‘时，在5位字符长度时是1.5个停止位，其他长度是2个停止位。

（7）[1:0]：BEC。设置每个字符的位数。当该字段为“00“时，数据长度为5位；当该字段为”01“时，数据长度为6位；当该字段为”10“时，数据长度为7位；当该字段为”11“时，数据长度为8位。

6）Modem控制寄存器（Modem Control Register，MCR）

该寄存器用于设置线路的工作状态，其偏移地址为0x04，格式如表11.9所示。当复位时，该寄存器的内容为0x00。

表11.9 Modem控制寄存器的格式

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位索引 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 读写属性 | ― | ― | ― | W | W | W | W | W |
| 名字 | ― | ― | ― | LOOP | OUT2 | OUT1 | RTSC | DTRC |

表中：

（1）[7:5]：保留位。

（2）[4]：LOOP。回路模式控制位。当该位为‘0‘时，正常操作；当该位为’1‘时，为环路模式。在环路模式中，TXD输出持续为’1‘，输出移位寄存器直接连接到输出移位寄存器中，其他信号连接关系为：DTR连接到DSR，RTS连接到CTS，OUT1连接到RI，OUT2连接到DCD。

（3）[3]：OUT2。在环路模式中连接到DCR的输入。

（4）[2]：OUT1。在环路模式中连接到RI的输入。

（5）[1]：RTSC。RTS信号控制位。

（6）[0]：DTRC。DTR信号控制位。

7）线路状态寄存器（Line Status Register，LSR）

该寄存器用于查看线路的工作状态，其偏移地址为0x05，格式如表11.10所示。当复位时，该寄存器的内容为0x00。

表11.10 线路状态寄存器的格式

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位索引 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 读写属性 | R | R | R | R | R | R | R | R |
| 名字 | ERROR | TE | TFE | BI | FE | PE | OE | DR |

（1）[7]：ERROR。错误标识位。当该位为‘1‘时，表示发生奇偶校验位错误，或帧错误或打断中断。当该位为’0‘时，表示没有错误。

（2）[6]：TE。传输为空标识位。当该位为‘1‘时，表示传输FIFO和传输移位寄存器都为空。给传输FIFO写数据时将清零。当该位为’0‘时，表示有数据。

（3）[5]：TFE。传输FIFO为空标识位。当该位为‘1‘时，表示当前传输FIFO为空，给传输FIFO写数据时清零；当该位为’0‘时，表示当前传输FIFO中有数据。

（4）[4]：BI。打断中断标识位。当该位为‘1‘时，表示接收到起始位+数据+奇偶位+停止位都有’0‘，即有打断中断；当该位为‘0’时，表示没有打断中断。

（5）[3]：FE。帧错误标识位。当该位为‘1’时，表示接收的数据没有停止位；当该位为‘0’时，表示没有错误。

（6）[2]：PE。奇偶校验位错误标识位。当该位为‘1’时，表示当前接收数据有奇偶错误；当该位为‘0’时，表示没有奇偶错误。

（7）[1]：OE。数据溢出标识位。当该位为‘1’时，表示有数据溢出；当该位为‘0’时，表示没有数据溢出。

（8）[0]：DR。接收数据有效标识位。当该位为‘1’时，表示在接收FIFO中有数据；当该位为‘0’时，表示在接收FIFO中无数据。

注：在对该寄存器进行读操作时，将LSR[4:1]和LSR[7]清零，LSR[6:5]在给传输FIFO写数据时清零，LSR[0]则对接收FIFO进行判断。

8）Modem状态寄存器（Modem Status Register，MSR）

该寄存器用于查看Modem的工作状态，其偏移地址为0x06，格式如表11.11所示。当复位时，该寄存器的内容为0x00。

表11.11 Modem状态寄存器的格式

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位索引 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 读写属性 | R | R | R | R | R | R | R | R |
| 名字 | CDCD | CRI | CDSR | CCTS | DDCD | TERI | DDSR | DCTS |

表中：

（1）[7]：CDCD。对DCD输入值的取反，或者在环路模式中连接到OUT2。

（2）[6]：CRI。对RI输入值取反，或者在环路模式中连接到OUT1。

（3）[5]：CDSR。对DSR输入值取反，或者在环路模式中连接到DTR。

（4）[4]：CCTS。对CTS输入值取反，或者在环路模式中连接到RTS。

（5）[3]：DDCD。DDCD指示位。

（6）[2]：TERI。RI边沿检测。RI状态从低到高变化。

（7）[1]：DDSR。DDSR指示位。

（8）[0]：DCTS。DCTS指示位。

9）分频锁存器

分频锁存器由分频锁存器1和分配锁存器2两个寄存器构成。用于生成指定的波特率时钟，其偏移地址分别为0x00和0x01，格式如表11.12和表11.13所示。当复位时，这两个寄存器的内容均为0x00。

表11.12 分频锁存器1的格式（低8位）

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位索引 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 读写属性 | R/W | | | | | | | |
| 名字 | 存放分频锁存器的低8位 | | | | | | | |

表11.13 分频锁存器2的格式（高8位）

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 位索引 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 读写属性 | R/W | | | | | | | |
| 名字 | 存放分频锁存器的高8位 | | | | | | | |

注：UART模块的时钟clk频率是DDR\_clk频率的一半。

## 11.3 PuTTY工具的下载和安装

PuTTY是一个集成虚拟终端、系统控制台和网络文件传输为一体的自由及开放源代码的程序。它支持多种网络协议，包括SCP、SSH、Telnet、rlogin和原始的套接字连接。它也可以连接到串行端口。

该工具的早期版本仅支持Windows平台，后陆续增加对各类Unix平台和Mac OS X的支持。

下载和安装PuTTY工具的主要步骤包括：

（1）在IE浏览器中，输入下面的网址：

https://www.putty.org/

（2）弹出新的页面，如图11.8所示。在该页面中，找到Download PuTTY标题。在该标题下面，找到You can download PuTTY here。鼠标单击here。

（3）自动跳转到下面新的地址

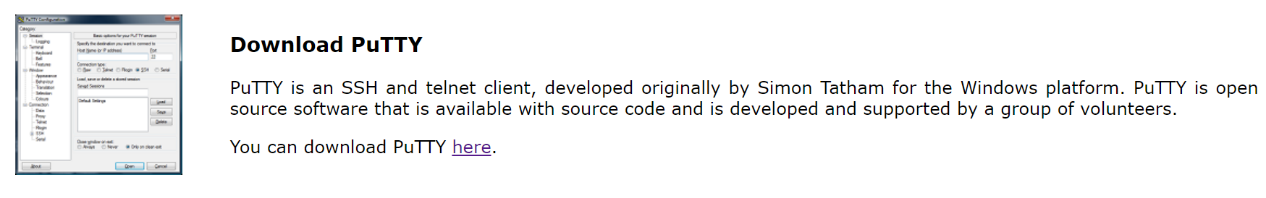


图11.8 PuTTY工具下载入口

https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

并打开新的页面，如图11.9所示。

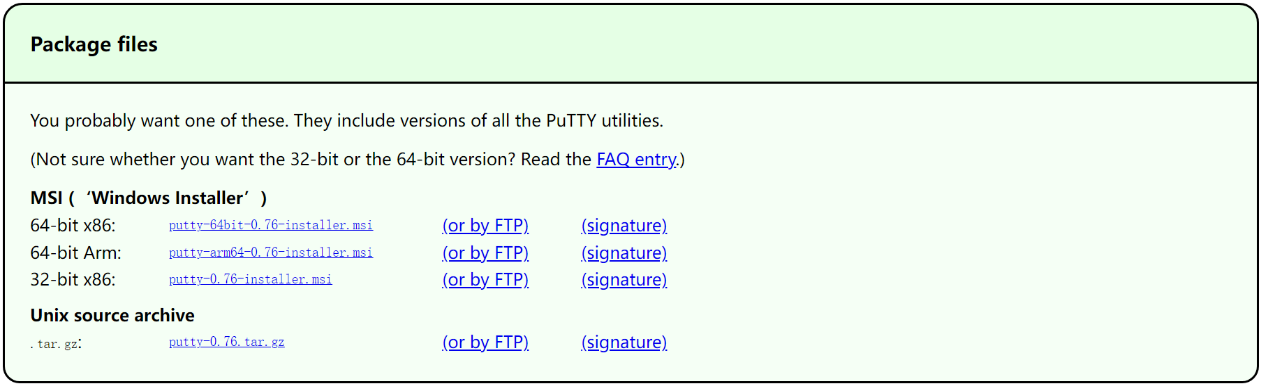


图11.9 PuTTY工具下载页面

（4）在该页面中，鼠标右键单击putty-64bit-0.76-installer.msi，弹出浮动菜单。在浮动菜单内，选择“将链接另存为...”条目。

（5）弹出另存为对话框界面。在该界面中，选择保存文件的位置，该文件的名字为putty-64bit-0.76-installer.msi。

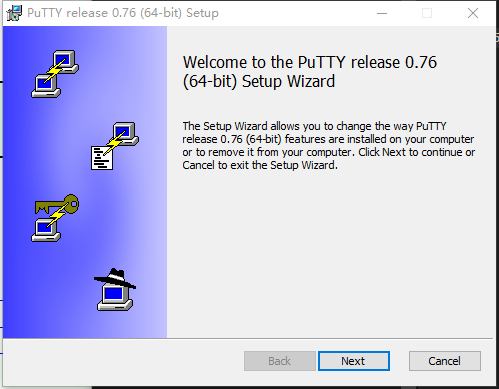


图11.10 PuTTY release 0.76(64-bit)Setup对话框界面（1）

（6）在保存该文件的文件夹中，找到并双击putty-64bit-0.76-installer.msi，启动安装过程。

（7）弹出PuTTY release 0.76 (64-bit) Setup对话框界面，如图11.10所示。在该对话框界面中，提示“Welcome to the PuTTY release 0.76（64-bit）Setup Wizard”信息。

（8）单击Next按钮。

（9）弹出PuTTY release 0.76 (64-bit) Setup-Destination Folder对话框界面，如图11.11所示。在该对话框界面中，提示“Click Next to install to the default folder or click Change to choose another”信息。选择默认的安装路径。

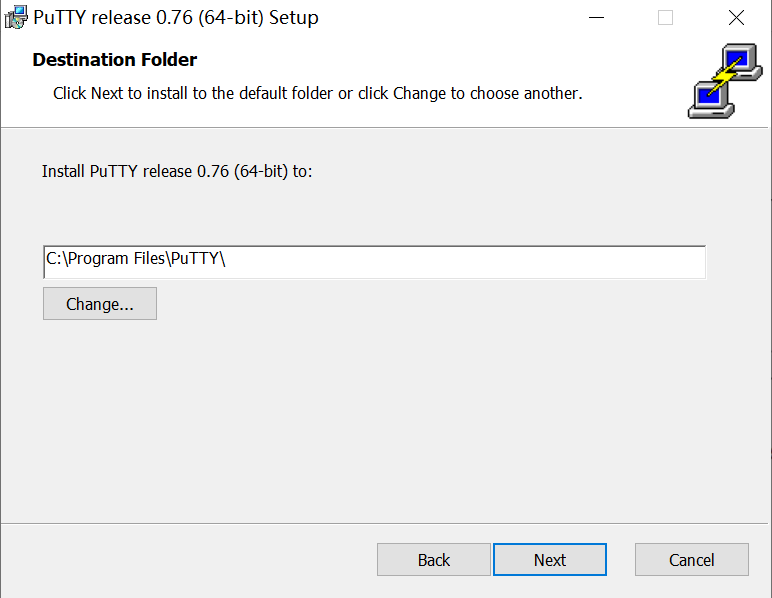


图11.11 PuTTY release 0.76(64-bit)Setup对话框界面（2）

（10）单击Next按钮。

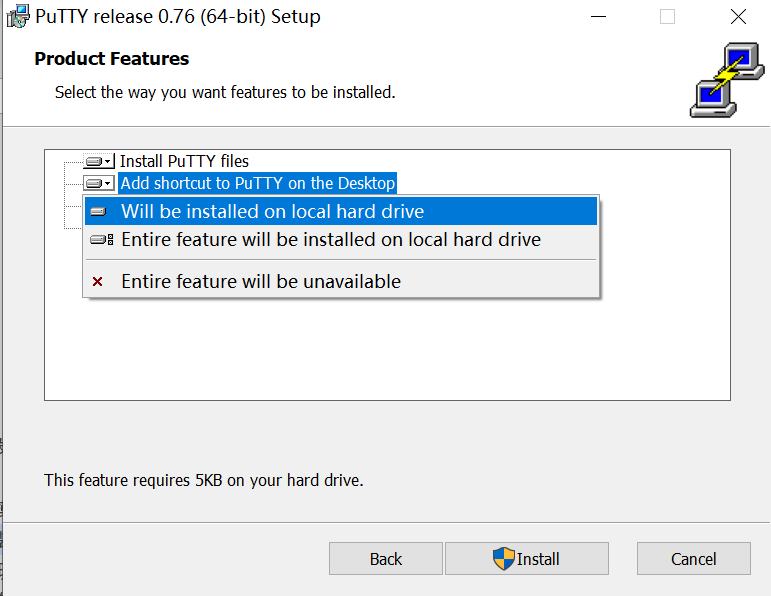


图11.12 PuTTY release 0.76(64-bit)Setup对话框界面（3）

（11）弹出PuTTY release 0.76(64-bit)Setup-对话框界面，如图11.12所示。在该对话框界面中，单击Add shortcut to PuTTY on the Desktop前面的按钮，出现浮动菜单。在浮动菜单内，选择Will be installed on local hard drive。

（12）单击Install按钮。

（13）弹出PuTTY release 0.76（64-bit）Setup-Installing PuTTY release 0.76（64-bit）对话框界面，如图11.13所示。在该界面中，将要显示安装进度。

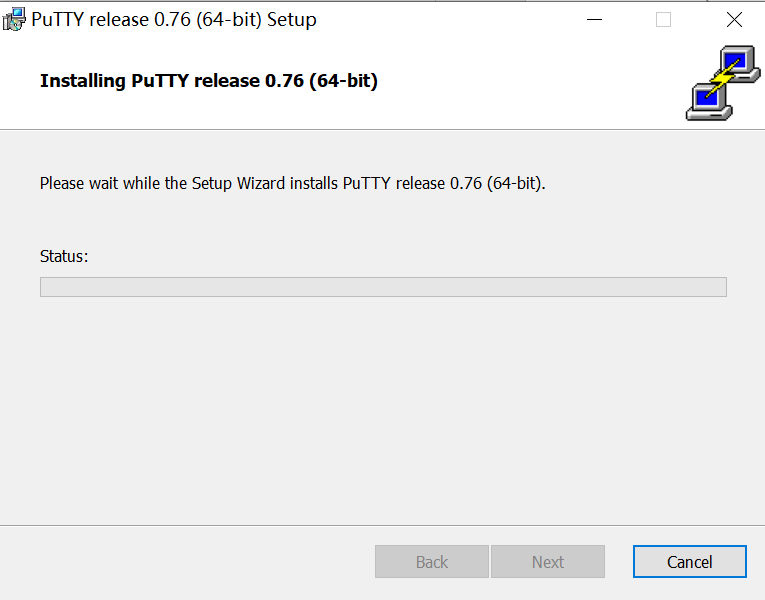


图11.13 PuTTY release 0.76(64-bit)Setup对话框界面（4）

（14）弹出用于账户控制对话框界面。在该界面中，提示“你要允许来自未知发布者的此应用对你的设备进行更改吗？”。

（15）单击按钮“是”，退出该对话框界面。

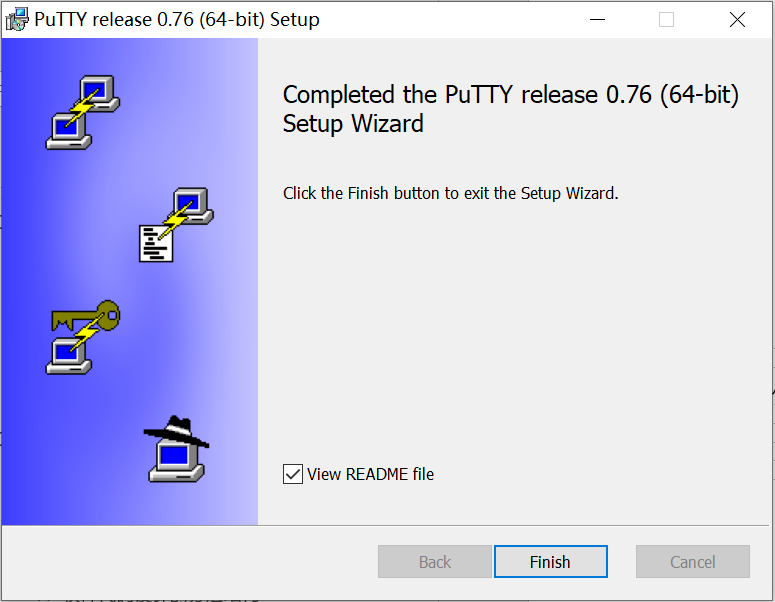


图11.14 PuTTY release 0.76(64-bit)Setup对话框界面（5）

（16）当安装完成后，自动弹出新的PuTTY release 0.76(64-bit) Setup对话框界面，如图11.14所示。在该界面中，提示“Completed the PuTTY release 0.76(64-bit)Setup Wizard”信息。

（15）单击Finish按钮，退出该对话框界面。

（16）当成功安装该软件工具后，在Windows 10操作系统桌面上出现名字为PuTTY(64-bit)图标。此外，通过Windows 10操作系统的开始->P->PuTTY(64-bit)，也可以找到该软件工具的入口。

## 11.4 串口重定位的设计与实现

本节将对C语言中的scanf语句和printf语句进行串口重定位，这样就可以通过串口调试助手软件工具为scanf语句输入变量参数，以及通过串口调试助手显示printf语句的打印信息。

### 11.4.1 软件代码设计的分层结构

在设计软件代码时，采用常用的分层和代码封装技术，如图11.15所示。采用分层结构的好处是，各层之间相对独立，易于对代码进行维护和移植。

底层寄存器的定义和声明

读/写寄存器操作的封装

应用程序接口(API)函数

应用程序代码

图11.15 程序设计的分层结构

1）当设计应用程序代码时，只需要和应用程序接口（Application Program Interface，API）函数打交道，程序开发人员只需要知道API函数的功能，以及所需要的参数即可，并不需要知道底层硬件的具体细节。比如，下面用C语言编写的串口初始化API函数。

uart3\_initialize(int baudrate, int databits, char eccmode, int stopbits)

该API函数的参数含义：

（1）bandrate：通信波特率；

（2）databits：数据位数；

（3）eccmode：校验模式，‘O’：奇校验；’E’：偶校验；‘N’：无校验；

（4）stopbits：结束位数。

2）对于API函数而言，需要调用对寄存器的读/写寄存器的操作的封装，如代码清单11-1所示。

代码清单11-1 API函数对底层的调用

int uart3\_initialize(int baudrate, int databits, char eccmode, int stopbits)

{

unsigned int divisor, bus\_freq;

unsigned char lcr;

if (baudrate < 2400)

baudrate = 115200;

/\*UART3寄存器的基地址，该基地址指向UART3的寄存器集，该寄存器集由复杂数据结构表示\*/

pUART3 = (HW\_UART\_t \*)LS1B\_UART3\_BASE;

pUART3->lcr = 0;

pUART3->R1.ier = 0;

//下面的代码通过调用LS1x\_BUS\_FREQUENCY函数，还设置总线频率

bus\_freq = LS1x\_BUS\_FREQUENCY(CPU\_XTAL\_FREQUENCY);

divisor = bus\_freq / 16 / baudrate; //计算分频系数, 总线频率为baudrate/16

pUART3->lcr = 0x80; // 设置 DLAB

pUART3->R0.dll = divisor & 0xFF; // 分频值低字节，设置寄存器

pUART3->R1.dlh = (divisor >> 8) & 0xFF; // 分频值高字节，设置寄存器

pUART3->R2.fcr = 0x07; // 复位fifo

switch (databits) //参数databits映射到寄存器lcr的字段

{

case 5: lcr = 0x00; break;

case 6: lcr = 0x01; break;

case 7: lcr = 0x02; break;

case 8:

default: lcr = 0x03; break;

}

switch (eccmode) //参数eccmode映射到寄存器lcr的字段

{

case 'O': lcr |= 0x08; break;

case 'E': lcr |= 0x18; break;

}

if (stopbits == 2) //参数stopbits映射到寄存器lcr的字段

lcr |= 0x04;

pUART3->lcr = lcr; //设置寄存器lcr

#if UART3\_USE\_INTERRUPT //如果定义了UART3\_USE\_INTERRUPT

//下面的代码用于初始化数据缓冲区

s\_RxBuf.Count = 0;

s\_RxBuf.pHead = s\_RxBuf.pTail = s\_RxBuf.Buf;

s\_TxBuf.Count = 0;

s\_TxBuf.pHead = s\_TxBuf.pTail = s\_TxBuf.Buf;

//下面的代码用于安装中断

ls1x\_install\_irq\_handler(LS1B\_UART3\_IRQ, uart3\_isr, NULL);

//下面的代码用于使能中断

LS1x\_INTC\_EDGE(LS1x\_INTC0\_BASE) &= ~INTC0\_UART3\_BIT;

LS1x\_INTC\_POL( LS1x\_INTC0\_BASE) |= INTC0\_UART3\_BIT;

LS1x\_INTC\_CLR( LS1x\_INTC0\_BASE) = INTC0\_UART3\_BIT;

LS1x\_INTC\_IEN( LS1x\_INTC0\_BASE) |= INTC0\_UART3\_BIT;

pUART3->R1.ier = 0x01; //接收到数据时，产生中断

#endif

return 0;

}

3）在底层将UART模块的寄存器集声明为一个数据结构，如代码清单11-2所示。

代码清单11-2 用于描述UART模块的寄存器集的数据结构

typedef struct

{

union //R0为联合体结构，偏移地址为0x00

{

volatile unsigned char dat; //当DLAB=’0’，作为数据传输保持寄存器

volatile unsigned char dll; //当DLAB=’1’，作为分频值低字节寄存器

} R0;

union //R1为联合体结构，偏移地址为0x01

{

volatile unsigned char ier; //当DLAB=‘0‘，作为中断使能寄存器

volatile unsigned char dlh; //当DLAB=‘1‘，作为分频值高字节寄存器

} R1;

union //R2为联合体结构，偏移地址为0x02

{

volatile unsigned char isr; //当读操作时，读取寄存器isr的标志

volatile unsigned char fcr; //当写操作时，将控制字写如寄存器fcr

} R2;

volatile unsigned char lcr; // 0x03，线路控制寄存器的偏移地址

volatile unsigned char mcr; // 0x04，MODEM控制寄存器

volatile unsigned char lsr; // 0x05，线路状态寄存器

volatile unsigned char msr; // 0x06，MODEM状态寄存器

} HW\_UART\_t;

### 11.4.2 printf函数的重定向实现

本节介绍对printf函数串口重定向的方法，通过将printf函数重定向，使得可以在串口调试工具中输出需要打印的信息。

#### 1．分析原有printf函数的执行过程

如果在main.c文件主函数main（）中调用printf（）函数，则需要在文件中使用下面的C语言语句

#include “stdio.h”

包含头文件stdio.h。当执行main.c文件调用printf函数时，就会跳转到头文件stdio.h中。

注：读者可以在C:/LoongIDE/mips-2015.05/mips-sde-elf/include路径下找到stdio.h文件。

在stdio.h文件标记为214的位置，有下面一行代码

int \_EXFUN(printf, (const char \*\_\_restrict, ...)

\_ATTRIBUTE ((\_\_format\_\_ (\_\_printf\_\_, 1, 2))));

该行代码声明了printf函数在外部定义，以及printf的参数类型以及归属。

注：不能修改此文件。

在LoongIDE主界面左侧Project Explorer窗口中，找到并展开libc文件夹。在该文件夹中，找到并展开stdio文件夹。在展开的文件夹中，找到头文件printf.h。鼠标双击printf.h，打开该文件。在该文件中，声明了printf函数：

int printf(const char\* format, ...);

因此，在调用printf语句时，程序计数器会跳转到该行代码处。

与保存文件printf.h所在相同的目录下，找到并双击打开printf.c。在该文件中，通过下面的C语言语句：

#include “printf.h”

包含了头文件printf.h。此外，也描述了函数printf的具体实现。在该函数中，主要包含如下两重要过程：

（1）跳转到当前文件中的内部函数\_vsnprintf()执行整个判断输出过程，即判断输出是否为空，是否逐字符打印，是否特殊数据类型%c%等；

（2）在输出过程中如果用\_out\_char()函数，则跳转到该文件中定义的\_out\_char()函数来执行。在该函数中，通过调用\_putchar()函数来输出字符。

在当前文件中并未定义\_putchar()函数，但是包含了头文件printf.h。在printf.h文件中，用下面的C语句：

extern void \_putchar(char character);

声明由外部定义\_putchar()函数。因此，程序会跳转到libc目录下的putchar.c文件来执行该函数。在该函数中，调用控制台输出函数console\_putch()，即

int \_putchar(int ch)

{

console\_putch(ch);

return 0;

}

需要注意，在putchar.c文件中使用下面的语句

#include”console.h”

包含了头文件console.h，因此会跳转到该文件执行与控制台相关的后续程序代码。

#### 2．创建新的头文件uart3.h

从本节介绍将printf函数重定位到串口的方法。通过创建与异步串口UART3相关的两个文件uart3.h与uart3.c，以实现对异步串口UART3的初始化、输入与输出等底层操作。创建新的头文件uart3.h的主要步骤包括：

（1）在LoongIDE中建立新的设计工程。

（2）在LoongIDE集成开发软件左侧的Project Explorer窗口中，找到并选中libc文件夹，单击鼠标右键，出现浮动菜单。在浮动菜单内，选择“New Header File”条项。

（3）弹出Enter file name对话框界面，如图11.16所示。在该对话框界面中New C header file of “libc”标题下的文本框中，输入uart3.h。

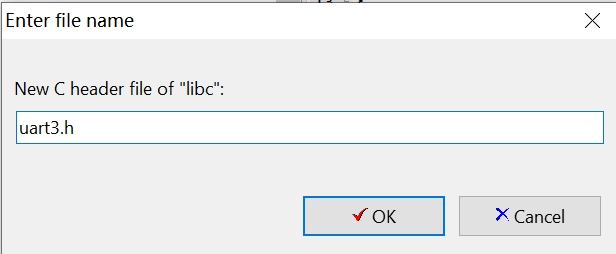


图11.16 Enter file name对话框界面（1）

（4）单击OK按钮，退出Enter file name对话框界面。

（5）自动打开文件uart3.h，在生成的模板代码后，添加设计代码，如代码清单11-3所示。

代码清单11-3 在头文件中添加的C语言代码

/\*

\* UART3 寄存器定义

\*/

#ifndef \_UART3\_H

#define \_UART3\_H

#endif // \_UART3\_H

#include "bsp.h"

#include "mips.h"

#include "cpu.h"

#include "stdio.h"

#include "ls1b.h"

#include "ls1b\_irq.h"

#define UART\_FIFO\_SIZE 16

typedef struct

{

union

{

volatile unsigned char dat; // 0x00 数据寄存器

volatile unsigned char dll; // 0x00 分频值低字节寄存器

} R0;

union

{

volatile unsigned char ier; // 0x01 中断使能寄存器

volatile unsigned char dlh; // 0x01 分频值高字节寄存器

} R1;

union

{

volatile unsigned char isr; // 0x02 中断状态寄存器

volatile unsigned char fcr; // 0x02 FIFO控制寄存器

} R2;

volatile unsigned char lcr; // 0x03 线路控制寄存器

volatile unsigned char mcr; // 0x04 MODEM控制寄存器

volatile unsigned char lsr; // 0x05 线路状态寄存器

volatile unsigned char msr; // 0x06 MODEM状态寄存器

} HW\_UART\_t;

int uart3\_initialize(int baudrate, int databits, char eccmode, int stopbits);

int uart3\_read(unsigned char \*buf, int size);

int uart3\_write(unsigned char \*buf, int size);

（6）保存该设计文件。

#### 3．创建新的源文件uart3.c

创建新的源文件uart3.c的主要步骤包括：

（1）在LoongIDE集成开发软件左侧的Project Explorer窗口中，找到并选中libc文件夹，单击鼠标右键，出现浮动菜单。在浮动菜单内，选择“New Source File”条项。

（2）弹出Enter file name对话框界面，如图11.17所示。在该对话框界面中New C source file of “libc”标题下的文本框中，输入uart3.c。

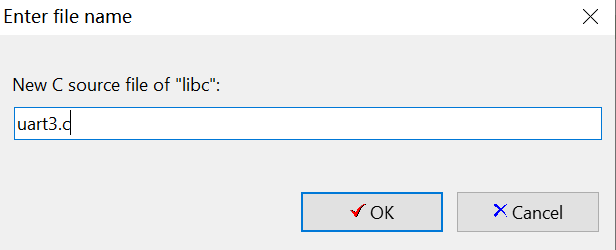


图11.17 Enter file name对话框界面（2）

（3）单击OK按钮，退出Enter file name对话框界面。

（4）自动打开文件uart3.c，在生成的模板代码后，添加设计代码，如代码清单11-4所示。

代码清单11-4 在源文件中添加的C语言代码

#include "uart3.h"

static HW\_UART\_t \*pUART3 = NULL; // UART3设备指针

#define UART3\_USE\_INTERRUPT 0 // 是否使用中断

#if UART3\_USE\_INTERRUPT

/\*

\* 数据接收缓冲区

\*/

//缓冲区: 循环模式，添加新的数据时，删除最旧的数据

/\*保存 UART\_BUF\_SIZE 个字符.

\*

\* 字节: 0 1 2 3 4 5 ...

\* \_\_ \_\_ xx xx xx xx \_\_ \_\_ ...

\* ^ ^

\* pHead |

\* pTail

\*

\*如果满或空: pHead==pTail;

\*/

#define UART\_BUF\_SIZE 256 //缓冲区大小

typedef struct

{

char Buf[UART\_BUF\_SIZE];

int Count;

char \*pHead;

char \*pTail;

} UART\_buf\_t;

static UART\_buf\_t s\_RxBuf; //接收缓冲区

static UART\_buf\_t s\_TxBuf; //发送缓冲区

//如果溢出，总是覆盖

static int enqueue\_to\_buffer(UART\_buf\_t \*data, char \*buf, int len)

{

int i;

for (i=0; i<len; i++)

{

\*data->pTail = buf[i];

data->Count++;

data->pTail++;

if (data->pTail >= data->Buf + UART\_BUF\_SIZE)

data->pTail = data->Buf;

}

/\*

\* if overflow, override the lastest data

\*/

if (data->Count > UART\_BUF\_SIZE) //溢出

{

data->Count = UART\_BUF\_SIZE;

data->pHead = data->pTail;

}

return len;

}

static int dequeue\_from\_buffer(UART\_buf\_t \*data, char \*buf, int len)

{

int i, count;

count = len < data->Count ? len : data->Count;

for (i=0; i<count; i++)

{

buf[i] = \*data->pHead;

data->Count--;

data->pHead++;

if (data->pHead >= data->Buf + UART\_BUF\_SIZE)

data->pHead = data->Buf;

}

return count;

}

/\*

\* UART3 中断句柄

\*/

static void uart3\_isr(int vector, void \*arg)

{

LS1x\_INTC\_IEN(LS1x\_INTC0\_BASE) &= ~INTC0\_UART3\_BIT;

do

{

int i, count = 0;

char buf[UART\_FIFO\_SIZE+1];

if (pUART3->R2.isr & 0x04) //收到数据，接收准备

{

for (i=0; i<UART\_FIFO\_SIZE; ++i)

{

if (pUART3->lsr & 0x01) //接收器准备

buf[i] = (char)pUART3->R0.dat;

else

break;

}

enqueue\_to\_buffer(&s\_RxBuf, buf, i);

}

//下面代码为有等待发送的数据

if ((s\_TxBuf.Count > 0) && (pUART3->lsr & 0x20)) //发送器准备

{

//从缓冲区挪出传输的字符

count = dequeue\_from\_buffer(&s\_TxBuf, buf, UART\_FIFO\_SIZE);

for (i=0; i<count; ++i)

pUART3->R0.dat = buf[i];

}

if (count > 0)

pUART3->R1.ier = 0x03; //rx &tx的中断

else

pUART3->R1.ier = 0x01; //rx上的中断

} while (!(pUART3->R2.isr & 0x01)); //无

LS1x\_INTC\_IEN(LS1x\_INTC0\_BASE) |= INTC0\_UART3\_BIT;

}

#endif

//uart3\_initialize内的代码详见代码清单11-1

int uart3\_initialize(int baudrate, int databits, char eccmode, int stopbits)

{

……

}

//读UART3数据函数，buf为数据缓冲区，size为读取读字节数，返回本次读取的字节数

int uart3\_read(unsigned char \*buf, int size)

{

int count = 0;

if ((pUART3 == NULL) || (buf == NULL))

return -1;

if (size < 0)

return 0;

#if UART3\_USE\_INTERRUPT

mips\_interrupt\_disable();

count = dequeue\_from\_buffer(&s\_RxBuf, buf, size);

mips\_interrupt\_enable();

return count;

#else

unsigned char \*p = buf;

while (count < size)

{

if (pUART3->lsr & 0x01) //接收器准备

{

\*p++ = pUART3->R0.dat; //接收字符

count++;

}

else

delay\_us(100);

}

return size;

#endif

}

// 给UART3写数据函数uart3\_write

int uart3\_write(unsigned char \*buf, int size)

{

if ((pUART3 == NULL) || (buf == NULL))

return -1;

if (size < 0)

return 0;

#if UART3\_USE\_INTERRUPT

int i, sent = 0;

/\* if idle, send immediately

\*/

if (pUART3->lsr & 0x20) //发送器准备

{

sent = size <= UART\_FIFO\_SIZE ? size : UART\_FIFO\_SIZE;

for (i=0; i<sent; ++i) //数据写到发送缓冲区

pUART3->R0.dat = buf[i];

if (sent > 0)

pUART3->R1.ier = 0x03; //rx & tx上的中断

else

pUART3->R1.ier = 0x01; //rx上的中断

}

/\* add remain data to transmit cached buffer

\*/

if (sent < size)

{

mips\_interrupt\_disable();

sent += enqueue\_to\_buffer(&s\_TxBuf, buf + sent, size - sent);

mips\_interrupt\_enable();

}

return sent;

#else

int count = 0;

unsigned char \*p = buf;

while (count < size)

{

if (pUART3->lsr & 0x20) /\* 传输准备就绪 \*/

{

pUART3->R0.dat = \*p++; /\* 传输单个字符 \*/

count++;

}

else

delay\_us(100);

}

return size;

#endif

}

（5）保存该设计文件。

#### 4．创建新的putcharr.c文件

在分析putchar.c文件实现方法的基础上，创建新的文件putcharr.c。该文件在不影响控制台输出（控制台输出已经重定向到UART 5）的情况下重定向到UART 3，以实现调用uart3.c文件中的串口底层操作代码，并通过UART 3输出单个字符，主要步骤包括：

（1）在LoongIDE集成开发软件左侧的Project Explorer窗口中，找到并选中libc文件夹，单击鼠标右键，出现浮动菜单。在浮动菜单内，选择“New Source File”条项。

（2）弹出Enter file name对话框界面。在该对话框界面中New C source file of “libc”标题下的文本框中，输入putcharr.c。

（3）单击OK按钮，退出Enter file name对话框界面。

（4）自动打开文件putcharr.c，在生成的模板代码后，添加设计代码，如代码清单11-5所示。

代码清单11-5 在源文件中添加的C语言代码

#include "uart3.h"

int \_putchar1(int ch)

{

uart3\_write((unsigned char\*)&ch,1);

return 0;

}

（5）保存该设计文件。

#### 5．修改print.h与printf.c文件

本节将修改printf.h与printf.c文件，在源文件的基础上添加一些函数，以达到在不影响原控制台功能的前提下，仍然可以使用UART 3，主要步骤包括：

（1）在printf.h文件中添加设计代码，如代码清单11-6所示。

代码清单11-6 在printf.h文件中添加设计代码

extern void \_putchar1(char character);

extern void uart3\_initialize(int baudrate, int databits, char eccmode, int stopbits);

int iprintf(const char\* format, ...);

前两行代码将函数\_putchar1()与uart3\_initialize()声明是外部的，分别在文件putcharr.c和文件uart3.c中实现，以便真正实现printf.c文件中对这两个函数的调用。第三行代码声明了要创建的UART 3输出函数iprintf（）。

注：不能随意命名该该函数的名字。

（2）在printf.c文件中添加设计代码，如代码清单11-7所示。

代码清单11-7 在printf.c文件中添加设计代码

static inline void \_out\_char1(char character, void\* buffer, size\_t idx, size\_t maxlen)

{

(void)buffer; (void)idx; (void)maxlen;

if (character)

{

\_putchar1(character);

}

}

int iprintf(const char\* format, ...)

{

uart3\_initialize(115200,8,'N',1);

va\_list va;

va\_start(va, format);

char buffer[1];

const int ret = \_vsnprintf(\_out\_char1, buffer, (size\_t)-1, format, va);

va\_end(va);

return ret;

}

（3）保存设计文件。

至此，就完成了printf（）函数的串口重定向。

### 11.4.3 测试重定位后的输出函数

在main.c文件中添加下面的C语言语句

#include “stdio.h”

即可调用printi()函数向UART 3输出要打印信息。由于上面对printf()函数的重定位没有修改控制台的显示输出（使用UART 5），所以仍然可以通过调用printf()函数向UART 5输出打印信息，进而在控制台上看到它。

此外，在main.c文件中添加下面的C语言语句

#include "libc/uart3.c"

则main主函数可直接调用用于UART 3的底层函数。

在文件main.c中添加设计代码来测试串口重定位的功能，如代码清单11-8所示。

代码清单11-8 在main.c文件中添加设计代码

#include "bsp.h"

#include "mips.h"

#include "stdio.h"

#include "libc/uart3.c"

unsigned char buf\_R[50];

int main()

{

int c=13;

uart3\_initialize(115200,8,'N',1); //初始化串口

for(int i=0;i<5;i++)

{

printf("hello UART5 %d \n",c);

iprintf("hello UART3 %d \r\n",c);

uart3\_read(buf\_R,8); //接收8个数据保存在buf\_R缓存区

}

printf("hello UART5");

}

在该测试中，打开PuTTY软件以监控控制台的UART 5的打印信息，用串口调试助手监视UART 3的打印信息。在串口调试助手中每输入一次八位数据，就在UART 3和UART 5输出不同的数据，直至for循环结束，在UART 5输出hello UART5后结束，如图11.18所示。从图中可以看出，原控制台输出的信息依旧在UART 5，没有改变。因此，上面的测试结果满足预期。

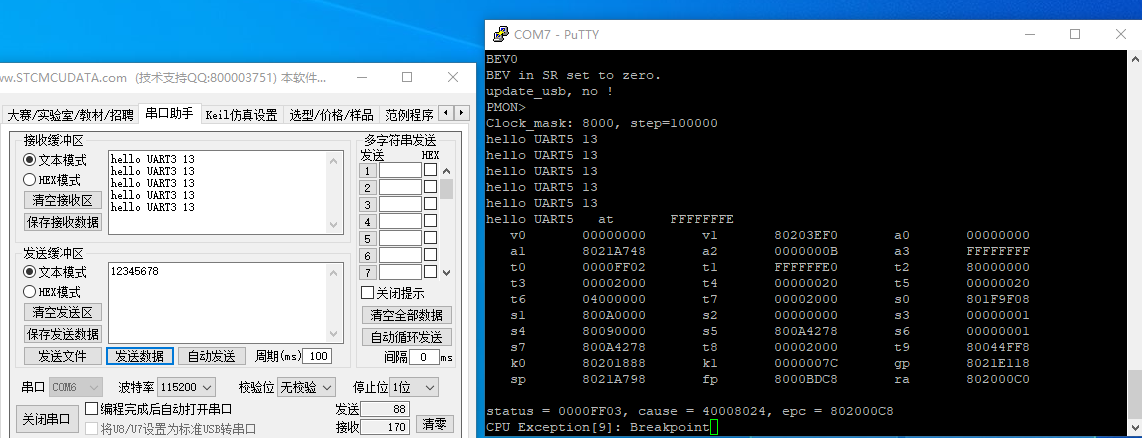


图11.18 测试窗口界面

注：UART 5换行使用“\n”，而UART 3换行使用“\r\n”。当转换到HEX模式可以更清楚的看到，UART3发送的数据“\r”为0D，“\n”为0A。

### 11.4.4 对输出函数重定位的总结

①关于串口三初始化函数的使用：

主函数main（）中不对串口三初始化则串口三不能输出任何信息，必须初始化。

重定向printi()输出函数中不对串口三初始化则会输出乱码，必须初始化。

函数\_putchar1()对串口三初始化会出现乱码，不能初始化。

②关于include路径的问题：

Include只会在同级文件下查找，否则要带文件路径，用“../”表示返回上级路径。故只要路径正确可以添加任意头文件或者源文件。

③重定向为什么不修改原有的printf（）或者printk（）函数，而是新建printi（）函数？

因为printf（）与printk（）是原有的控制台输出函数。如果修改printf（）函数则控制台提示信息“Clock\_mask”、“step”与程序运行结束输出的寄存器值、指针值等都会显示在串口三。如果修改printk（）函数，虽然控制台提示信息“Clock\_mask”、“step”会显示在串口五，但与程序运行结束输出的寄存器值和指针值等依旧会显示在串口三。这是我们不想看到的。

④为什么新建printi（）函数，而不是其他什么函数名？

因为主函数在调用输出函数的时候，是通过其包含头文件#include”stdio.h”来查找的，而此头文件是禁止我们修改的，所以只能新建此头文件下已经有声明过的函数。并且此文件中对函数参数类型等特征做了规定，并不是每个都能拿来用。我们需要找到和原始printf（）相同类型的函数，并且保证该函数没有被用过才能在外部重新声明并使用。显然printi（）是我们找到的符合要求的函数。

⑤为什么新建putcharr.c文件，而不是对原有的putchar.c文件做修改？

同第三条一样，还是为了控制台功能的正常使用。

⑥串口三与串口五在函数执行过程中，只有print.c文件中\_vsnprintf（）函数是共用的，其他都是新加的。综上所述，与其叫“串口重定位”不如叫“如何制作串口三的驱动”。因为没改变原有控制台（串口五）任何功能，还加上了串口三的功能。

### 11.4.5 浅析标准库函数scanf的执行过程

如果在主程序中调用scanf()函数时，主函数所在文件会根据其包含的头文件#include”stdio.h”，跳转到该头文件寻找scanf()的定义。

在stdio.h文件中针对scanf()函数有如下定义，int \_EXFUN(scanf, (const char \*, ...) \_ATTRIBUTE ((\_\_format\_\_ (\_\_scanf\_\_, 1, 2))));该文件声明scanf()是一个外部定义的函数，将从其他文件寻找。

在标准输入输出库中，scanf.h下有该函数的声明，scanf.c文件下有该函数的实现，所以会跳转到scanf.c执行。scanf输入主要有以下两部分：①引号里面的部分为控制输入类型的字符串，以下称之为控制流；②引号外则为个数不固定的，需要被赋值的各变量的地址（地址即存储器地址，各变量值即对应地址下存储的数据）。

因为scanf函数需要被赋值的参量个数不固定，所以要用到可变参数列表va\_list ap用于定义可变参数列表ap，va\_start用于获取可变参数列表的第一个参数的地址，va\_end用于清楚可变参数列表。可变参数列表按顺序依次存储需要被赋值的变量的地址。

在va\_start与va\_end之间调用了一个非常重要的函数\_doscan()，该函数输入一个指向控制流首地址的指针、可变参数列表以及stdin（stdin本质是一个指向文件的指针）该文件存储的是从串口输入的数据（这些数据以字符类型存储在该文件中）——标准输入数据流（以下简称为数据流）。\_doscan()则利用这些输入，将数据流的数据按照控制流的指示转化为对应数据类型后，向可变参数列表中存放的地址的对应存储空间写入数据，这样就完成了诸多变量的赋值。

\_doscan()函数执行流程如下：①跳过控制流与数据流开头没用的空格；②检测控制流，如果读取到的非空字符不是“%”则对数据流进行检测，如果数据流与控制流不匹配，则发生错退出函数执行，以此做到控制流与数据流的匹配；③读取到“%”后跳过去：④检测后边是不是“\*”符号，如果是此符号代表只读取数据，不进行存储；⑤接下来判断控制流是不是数字，如果是则代表对数据宽度（即位数）进行控制；⑥然后判断控制流是不是特殊说明符“h”代表short类型、“l”代表long类型、“L”代表double类型；⑦接下来，控制流字符肯定就是输入数据的种类（八进制整数、十进制整数、十六进制整数、指针、浮点数、字符、字符串、扫描输入模式等）；⑧根据不同的标志位以及输入数据类型，按照对应的处理逻辑，向可变参数列表中存放的地址的对应存储空间写入数据。

附：这里以\*va\_arg(ap, unsigned long \*) = (unsigned long) val;为例讲解可变参数列表的使用。va\_arg宏，获取可变参数的当前参数，返回指定类型并将指针指向下一参数。此处va\_arg宏获取可变参数列表ap（列表中存储需要被赋值变量的地址）当前参数，并规定其类型为“指向无符号长整型数据的指针”，“\*”+“该指针”就是需要被赋值的变量。该代码实现了将val进行强制数据类型转换后，赋值给对应变量的操作。而且可变参数列表指针自动指向下一个参数，下一次直接调用va\_arg即可，无需对其指针做任何修改。

### 11.4.6 scanf函数的重定向实现

#### 1．修改源文件uart3.c

首先修改uart3.c文件下的串口读取函数，以实现如下功能：

输入过程中逐字符回显，因为控制台输入与输出在同一个窗口不需要回显。串口调试助手输入与输出不在同一窗口，需要回显才能判断开发板是否受到数据以及收到的数据是否正确。

之前的uart3\_read只能实现将指定长度字符串读入到缓存区中，使用极其不灵活。经修改后，只定义输入数据的最大长度。达到最大长度或者遇到回车换行结束，灵活度大大提高。

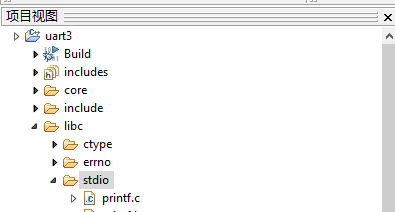
修改后的uart3\_read（）函数如下：

|  |
| --- |
| int uart3\_read(unsigned char \*buf, int size)  {  int count = 0;  if ((pUART3 == NULL) || (buf == NULL))  return -1;  if (size < 0)  return 0;  #if UART3\_USE\_INTERRUPT  mips\_interrupt\_disable();  count = dequeue\_from\_buffer(&s\_RxBuf, buf, size);  mips\_interrupt\_enable();  return count;  #else  while (count<size)  {  if (pUART3->lsr & 0x01)  {  \*buf = pUART3->R0.dat;  buf\_H[0] = \*buf;  uart3\_write(buf\_H,1); //字符逐字回显  if(\*buf==10) //如果读到了换行，代表结束读取  break;  buf++;  count++;  }  else  delay\_us(100);  }  return size;  #endif  } |

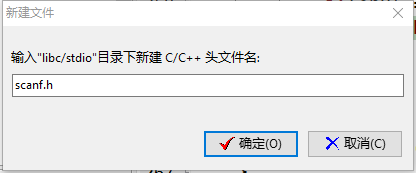
#### 2．创建新的头文件scanf.h

创建scanf.h与scanf.c两个文件，以实现串口输入重定向，具体操作如下：

如下图，在LoongIDE集成开发软件左侧的“项目视图”中双击打开“libc”文件夹，右键单击“stdio”文件夹，选择“新建头文件”选项。



如下图，在弹出的对话框中输入新建头文件的名称“scanf.h”,并点击“确定”。

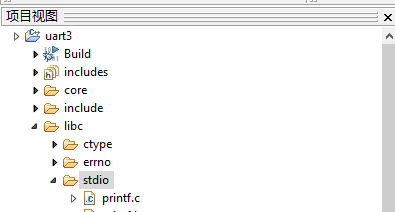


在新建的scanf.h文件中添加如下代码并保存

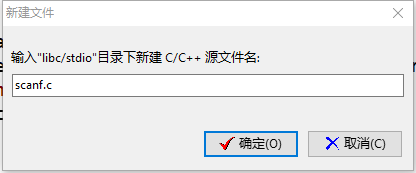
|  |
| --- |
| /\*  \* scanf.h  \*  \* created: 2021/10/26  \* author:  \*/  #ifndef \_SCANF\_H  #define \_SCANF\_H  #include "bsp.h"  #include <stdarg.h>  #include <stddef.h>  #endif // \_SCANF\_H  extern void getch(char character);  extern void uart3\_initialize(int baudrate, int databits, char eccmode, int stopbits);  extern int uart3\_read(unsigned char \*buf, int size);  int scanf(const char\* format, ...); |

#### 3．创建新的源代码文件uart3.c

如下图，在LoongIDE集成开发软件左侧的“项目视图”中，双击打开“libc”文件夹，右键单击“stdio”文件夹，选择“新建源代码文件”选项。



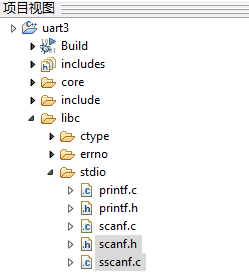
如下图，在弹出的对话框中输入新建头文件的名称“scanf.c”,并点击“确定”。



在新建的scanf.c文件中添加如下代码并保存

|  |
| --- |
| /\*  \* scanf.c  \*  \* created: 2021/10/26  \* author:  \*/  #include "bsp.h"  #include "scanf.h"  #include <stdio.h>  #include <ctype.h>  #include <stdbool.h>  #include <stdint.h>  #include <stdlib.h>  #include <stdarg.h>  #include <float.h>  #define FL\_NOASSIGN (1U << 0U) //1  #define FL\_WIDTHSPEC (1U << 1U) //2  #define FL\_SHORT (1U << 2U) //8  #define FL\_LONG (1U << 3U) //16  #define FL\_LONGDOUBLE (1U << 4U) //32  #define SUPPORT\_NOASSIGN  #define SUPPORT\_RETURN\_NUMBER  #define SUPPORT\_INTEGER  #define SUPPORT\_FLOAT  #define SUPPORT\_CHAR  #define SUPPORT\_STRING  #define SUPPORT\_SCAN  // 分配输入数据缓存区（数据流最大空间），并初始化  #define buf\_size 50  unsigned char buf\_in[buf\_size] = {0};  //清楚数据缓存区  void clean\_buf()  {  int i=0;  for(;i<buf\_size;i++)  buf\_in[i]=0;  }  int \_doscan(const char \*format, va\_list ap)  {  int flags; // 标志  int kind; // 数据类型  register int ic = 0; // 从数据缓存数组读取到的字符  int nrchars = 0; // 已读取字符数  int done = 0; // 完成项目数(已经输入几个变量)  unsigned precision = buf\_size; // 精度（实数小数点位数，整数宽度）    int base; // 整数进制  long val=0; // 一个无符号整数值  bool sign =0; // 输入的正负（0正1负）  register char \*str; // 临时指针  #ifdef SUPPORT\_FLOAT  double val\_f1=0; // 一个浮点数  double val\_f2=0; // 一个浮点数  register int ic\_buf = 0; // 从数据缓存数组读取到的字符  int nrchars\_buf = 0; // 已读取字符数  int nrchars\_precision\_buf = 0; //用来控制浮点数精度  #endif  #ifdef SUPPORT\_SCAN  char char\_buf = 0; // 扫描模式结束符  #endif  if (!\*format) return 0; // 控制流为空，立即结束    while (1)  {  if (isspace(\*format)) // 如果控制流开头就是没用的空格  {  while (isspace(\*format))  format++; // 跳过控制流那些空白符号  }  if (!\*format) break; // 控制流为空，立即结束  ic = buf\_in[nrchars]; // 从数据流读取一个字符  nrchars++;  while (isspace (ic)) // 数据流跳过空白符号  {  ic = buf\_in[nrchars];  nrchars++;  }  if (ic == 0) break; // 除了空格之外没输入，直接退出  else ; // 否则，数据流保留读取到的第一个数据  if (\*format != '%') break; // 如果控制流检测到的不是百分号，出错跳出去  else format++; // 如果控制流是%，准备读取%之后的内容      flags = 0;  if (\*format == '\*') // 如果检测到"\*"符号，只读取不存储  {  format++;  flags |= FL\_NOASSIGN;// 相应标志位变化（按位或，置一）  }  #ifdef SUPPORT\_NOASSIGN  if (isdigit (\*format)) // 如果控制流的是数字，分配精度  { //整数位数，实数为小数点后位数  flags |= FL\_WIDTHSPEC;  for (precision = 0; isdigit (\*format);)  precision = precision \* 10 + \*format++ - '0';  }  #endif  switch (\*format) // 如果在控制流检测到特殊说明符h,l,L  { // 则利用按位或运算将对应标志位置一  case 'h': flags |= FL\_SHORT; format++; break;  case 'l': flags |= FL\_LONG; format++; break;  case 'L': flags |= FL\_LONGDOUBLE; format++; break;  }        kind = \*format; // 控制流接下来肯定是数据类型  #ifdef SUPPORT\_RETURN\_NUMBER  if(kind=='n') // 数据类型是返回已经读取字符数  {  if (!(flags & FL\_NOASSIGN))  {  if (flags & FL\_SHORT)  \*va\_arg(ap, short \*) = (short) --nrchars;  //获取可变参数的当前参数，返回指定类型并将指针指向下一参数  else if (flags & FL\_LONG)  \*va\_arg(ap, long \*) = (long) --nrchars;  else  \*va\_arg(ap, int \*) = (int) --nrchars;  }  while(!isspace(ic)) // 抹除非规范数据  {  nrchars++;  ic = buf\_in[nrchars];  }  }  #endif  //八进制、十进制、十六进制整数、指针  #ifdef SUPPORT\_INTEGER  else if(kind=='o' || kind=='d' || kind=='x' || kind=='X' || kind=='p' )  {  if(ic=='-') // 先判断输入数据的正负  {  sign++; // 如果为负数则将符号标志置一  ic = buf\_in[nrchars];  nrchars++;  }  int width = 0; //用width与precision控制读入数据的位数  if (kind=='o') //八进制整数  {  base=8;  while(ic>=48 && ic<=55 && width<precision)  {  val = val\*base + (ic-48);  ic = buf\_in[nrchars];  nrchars++;  width++;  }  }  else if (kind=='d') //十进制整数  {  base=10;  while(isdigit(ic) && width<precision)  {  val = val\*10 + (ic-48);  ic = buf\_in[nrchars];  nrchars++;  width++;  }  }  else //十六进制整数  {  base=16;  while((isdigit (ic) || ( ic>64 && ic<71 ) || ( ic>96 && ic<103 ))  && width<precision)  {  if(isdigit (ic))  val = val\*16 + (ic-48);  else if( ic>64 && ic<71 )  val = val\*16 + (ic-65+10);  else  val = val\*16 + (ic-97+10);  ic = buf\_in[nrchars];  nrchars++;  width++;  }  }  width = 0; // 数据转换完毕后将width归零  if(sign==1) // 如果符号标志位为1则将正数val取反  {  val = -1\*val;  sign = 0;  }  if(!(flags & FL\_NOASSIGN)) // 不是只读模式，才会输出  { // 根据特殊说明符，附加数据类型  if (flags & FL\_LONG)  \*va\_arg(ap, unsigned long \*) = (unsigned long) val;  else if (flags & FL\_SHORT)  \*va\_arg(ap, unsigned short \*) = (unsigned short) val;  else  \*va\_arg(ap, unsigned \*) = (unsigned) val;  }  val = 0;  while(!isspace(ic)) // 抹除非规范数据  {  nrchars++;  ic = buf\_in[nrchars];  }  }  #endif  #ifdef SUPPORT\_FLOAT  else if(kind=='e'||kind=='E'||kind=='f'||kind=='F'||kind=='g'||kind=='G')  { // 如果输入的是浮点数  if(ic=='-') // 先判断正负  {  sign++;  ic = buf\_in[nrchars];  nrchars++;  }  while(isdigit(ic)) // 然后处理整数部分的数据转换  {  val\_f1 = val\_f1 \*10 + (ic-48);  ic = buf\_in[nrchars];  nrchars++;  }  if(ic=='.') // 如果识别到小数点  {  nrchars\_precision\_buf = nrchars; //记录下小数点后一位位置  ic = buf\_in[nrchars];  nrchars++;  while(isdigit(ic)) // 一直向后扫描，直至小数部分结束  {  ic = buf\_in[nrchars];  nrchars++;  }  ic\_buf=ic; // 记录当前扫描的位置  nrchars\_buf=nrchars; // 将当前位置存储在缓存区  // 根据（精度/小数长度）更改小数部分数据类型转换的起点  if(nrchars > nrchars\_precision\_buf + precision + 1)  nrchars = nrchars\_precision\_buf + precision - 1 ;  else  nrchars = nrchars - 2;  ic = buf\_in[nrchars];  nrchars--;  while(isdigit(ic)) // 小数部分数据类型转换，从后向前迭代与整数不同  {  val\_f2 = val\_f2 \*0.1 + (ic-48)\*0.1;  ic = buf\_in[nrchars];  nrchars--;  }  ic=ic\_buf; // 恢复之前记录的小数结束的位置与数据  nrchars=nrchars\_buf;  }  val\_f2 = val\_f2 + val\_f1; // 小数部分与整数部分合一  if(sign==1) // 根据符号标志位更改数据正负  {  val\_f2 = -1 \* val\_f2;  sign = 0;  }  if(!(flags & FL\_NOASSIGN)) //不是只读取不存储才输出  {  if (flags & FL\_LONGDOUBLE)  \*va\_arg(ap, long double \*) = (long double) val\_f2;  else if (flags & FL\_LONG)  \*va\_arg(ap, double \*) = (double) val\_f2;  else  \*va\_arg(ap, float \*) = (float) val\_f2;  }  while(!isspace(ic)) // 抹除非规范数据  {  nrchars++;  ic = buf\_in[nrchars];  }  val\_f1 = 0; // 整数与小数状态初始化  val\_f2 = 0; // 以便输入下一个数据  }  #endif  #ifdef SUPPORT\_CHAR  else if(kind=='c') // 字符类型（非空）  {  while(isspace(ic)) // 消除字符串的空白  {  ic = buf\_in[nrchars];  nrchars++;  }  if(!(flags & FL\_NOASSIGN)) // 不是只读取不存储才输出  \*va\_arg(ap, char \*) = (char) ic;  }  #endif  #ifdef SUPPORT\_STRING  else if(kind=='s') // 字符串类型，遇到空格结束  {  str = va\_arg(ap, char \*); //区分str指针加一与可变参数列表指针加一  while(isspace(ic)) // 消除空格  {  ic = buf\_in[nrchars];  nrchars++;  }  while(ic!=0 && ic!=32 && ic!=10 && ic!=13) // 遇到空格、回车、换行结束  {  \*str++ = (char) ic;  ic = buf\_in[nrchars];  nrchars++;  }  while(\*str!=0) //擦除原先字符串保存的多余内容，以便对字符串重复赋值  \*str++=0;  }  #endif  #ifdef SUPPORT\_SCAN  else if(kind=='[' && \*(format+1)=='^' && \*(format+3)==']') //扫描模式输入  {  if(\*(format+2)!=' ') // 不是空格则作为结束符，是空格扫描到换行才结束  char\_buf = \*(format+2);  format = format + 3;  str = va\_arg(ap, char \*); // 同上用于字符串连续输入  while(isspace(ic)) // 清除数据流的空格  {  ic = buf\_in[nrchars];  nrchars++;  }  while(ic!=0 && ic!=char\_buf && ic!=10 && ic!=13) //结束条件  {  \*str++ = (char) ic;  ic = buf\_in[nrchars];  nrchars++;  }  char\_buf = 0;  while(\*str!=0) //擦除原先字符串保存的多余的内容  \*str++=0;  }  #endif  else done--; // 数据类型不满足以上所有情况，说明没输出  precision = buf\_size ; // 每次输入数据结束都重置精度为默认最大值  if (!(flags & FL\_NOASSIGN) && kind != 'n') done++;  format++;  }  return done ;  }  int scanf(const char \*format, ...)  {  uart3\_initialize(115200,8,'N',1); // 初始化串口三  uart3\_read(buf\_in,buf\_size); // 从串口读取数据到缓存区  va\_list ap; // 定义可变参数列表  int retval;  va\_start(ap, format); // 获取可变参数列表的第一个参数的地址  retval = \_doscan(format, ap);  va\_end(ap); // 清空va\_list可变参数列表  clean\_buf(); // 清除数据缓存区  return retval;  } |
|  |

结束后将在“libc/stdio/”文件夹下看到“scanf.h”与“scanf.c”两个文件，如下图所示：



#### 4．在main.cpp下添加代码

在主程序下添加如下代码，用于测试串口重定向：

|  |
| --- |
| /\*  \* 本代码实现一个 UART3 的驱动程序  \*/  #include "bsp.h"  #include "mips.h"  #include "stdio.h"  #include "libc/uart3.c"  //#define TEST\_POINTER  //#define TEST\_NOASSIGN  //#define TEST\_MATCHING  //#define TEST\_RETURN\_NUMBER  //#define TEST\_INTEGER\_D  //#define TEST\_INTEGER\_O  //#define TEST\_INTEGER\_X  //#define TEST\_FLOAT  //#define TEST\_CHAR  //#define TEST\_STRING  //#define TEST\_SCAN  int main()  {  int a1 = 0 ,a2 = 0,a3 = 0 ;  float b1 = 0 ,b2 = 0,b3 = 0 ;  char c1 ='0',c2 ='0' ;  int \*p = NULL;  char string1[10],string2[10];  uart3\_initialize(115200,8,'N',1); //初始化串口  #ifdef TEST\_POINTER  a1 = 1;a2 = 2;a3 = 3;  iprintf(" &a1=%p\r\n &a2=%p\r\n &a3=%p\r\n",&a1,&a2,&a3);  #endif  for(int i=0;i<5;i++)  {  #ifdef TEST\_POINTER  scanf("%p",&p);  iprintf("p=%p \r\n",p);  iprintf("\*p=%d \r\n",\*p);  #endif  #ifdef TEST\_NOASSIGN  printf("hello UART5 %d %d \n",a1,a2);  iprintf("hello UART3 %d %d \r\n",a1,a2);  scanf("%\*d %d",&a1,&a2);  #endif  #ifdef TEST\_MATCHING  printf("hello UART5 %d %d \n",a1,a2);  iprintf("hello UART3 %d %d \r\n",a1,a2);  scanf("input is %d,%d",&a1,&a2);  #endif  #ifdef TEST\_RETURN\_NUMBER  printf("hello UART5 %d %d \n",a1,a2);  iprintf("hello UART3 %d %d \r\n",a1,a2);  scanf("%d %n",&a1,&a2);  #endif  #ifdef TEST\_INTEGER\_D  printf("hello UART5 %d %d %d \n",a1,a2,a3);  iprintf("hello UART3 %d %d %d \r\n",a1,a2,a3);  scanf("%2d %3d",&a1,&a2);  a3 = a1 + a2;  #endif  #ifdef TEST\_INTEGER\_O  printf("hello UART5 %d %d %d \n",a1,a2,a3);  iprintf("hello UART3 %d %d %d \r\n",a1,a2,a3);  scanf("%o %o",&a1,&a2);  a3 = a1 + a2;  #endif  #ifdef TEST\_INTEGER\_X  printf("hello UART5 %d %d %d \n",a1,a2,a3);  iprintf("hello UART3 %d %d %d \r\n",a1,a2,a3);  scanf("%x %x",&a1,&a2);  a3 = a1 + a2;  #endif  #ifdef TEST\_FLOAT  printf("hello UART5 %f %f %f \n",b1,b2,b3);  iprintf("hello UART3 %f %f %f \r\n",b1,b2,b3);  scanf("%2f %3f",&b1,&b2);  b3 = b1 + b2;  #endif  #ifdef TEST\_CHAR  printf("hello UART5 %c %c \n",c1,c2);  iprintf("hello UART3 %c %c \r\n",c1,c2);  scanf("%c %c",&c1,&c2);  #endif  #ifdef TEST\_STRING  printf("hello UART5 %s %s \n",string1,string2);  iprintf("hello UART3 %s %s \r\n",string1,string2);  scanf("%s %s",&string1,&string2);  #endif  #ifdef TEST\_SCAN  printf("hello UART5 %s %s \n",string1,string2);  iprintf("hello UART3 %s %s \r\n",string1,string2);  scanf("%[^a] %[^ ]",&string1,&string2);  #endif  }  printf("hello UART5");  } |

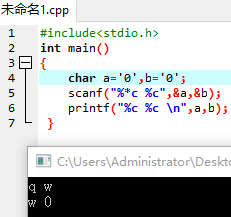
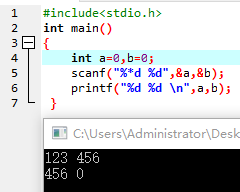
至此，串口重定向工作全部完成。

### 11.4.7 测试、对比与标准库函数的scanf的异同

在scanf.c与测试用的主程序文件main.cpp中用了大量的宏定义：scanf中的宏定义决定是否支持某一功能，注释掉对应宏定义将其禁用；main.cpp中的宏定义决定测试scanf函数的哪一功能，无需修改代码只需修改宏定义即可转变测试功能。

#### 1．测试scanf函数的只读模式（%\*）

在标准输入输出库中scanf语句控制流字符串带有符号“\*“，表示对该数据只读取但不会存储（即不会保存在变量中）。如下图所示，当输入数据为123和456时，数据123只读取不存储，所以不会存放于变量a中。注意：只读模式下没用调用va\_arg宏，所以可变参数列表的指针并没有自动加一，所以第二个输入的数据456将会保存到变量a中，而不是变量b。因为数据流读取完毕，没有值赋给变量b，所以变量b保持初始值0。在此，如果输入为123、456和789，则b的值将会是789。如下图所示，输入字符同样支持只读模式。



重定向后的只读模式测试如下：

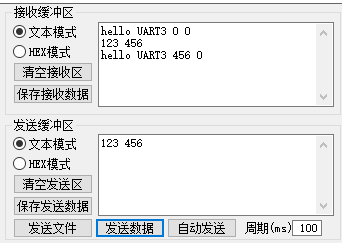
将main.cpp文件下除“#define TEST\_NOASSIGN”之外的宏定义全部注释掉，在loongIDE软件上方操作菜单中点击“项目”之后再点击“编译”（快捷键为Ctrl+F9），对源代码进行编译。

编译成功后，用EJTAG线将开发板与电脑连接，以进行裸机程序下载。用串口线将开发板串口三与电脑USB端口进行连接，以进行串口重定向的测试。最后将开发板上电。

打开设备管理器，查看串口连接的端口号，打开串口调试助手根据上述端口号选择端口，并将波特率设置为115200，校验位设置为无校验，停止位设置为1位，点击“打开串口”。并将串口调试助手的发送缓冲区与接收缓冲区都设置为文本模式。

回到loongIDE软件下，在其上方操作菜单中点击“调试”之后再点击“运行”（快捷键为F9），对源代码进行调试。

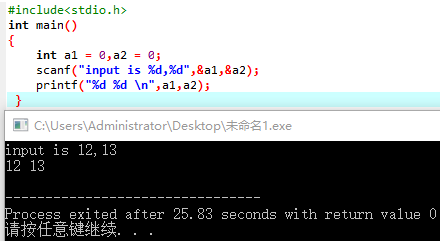
在串口调试助手的发送缓冲区下输入数据“123 456”后按下回车，再点击发送数据。



由上图可以看出，与标准库函数相同，123被识别为只读数据，456被赋值给第一个变量。同样字符类型数据也支持只读模式，在此不做测试。

#### 2．数据流与控制流匹配

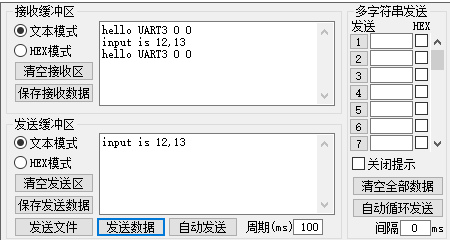
在标准输入输出库中scanf语句控制流与输入数据流的内容要匹配。例如在下图中，控制流在第一个“%”前使用了“input is”则在输入数据流的时候也必须使用“input is”，控制流在第一个“%”与第二个“%” 之间使用逗号作为分隔符，则在输入数据流的时候也必须使用逗号将两个数据隔开。



重定向后的测试如下：

将main.cpp文件下除“#define TEST\_MATCHING”之外的宏定义全部注释掉，在loongIDE软件上方操作菜单中点击“项目”之后再点击“编译”（快捷键为Ctrl+F9），对源代码进行编译，在其上方操作菜单中点击“调试”之后再点击“运行”（快捷键为F9），对源代码进行调试。

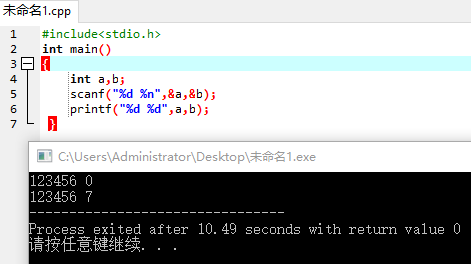
在串口调试助手的发送缓冲区下输入数据“input is 12，13”后按下回车，再点击发送数据。



如上图所示，重定向后的scanf并不支持控制流与数据流匹配。在重定向过程中对标准scanf函数的语法结构上做了以下修改：控制流“%”符号前不能有除空格之外的字符，控制流每两个“%”之间用空格隔开，不能加逗号之类的字符，否则将会被判别为非法字符，直接推出scanf函数。

#### 3．返回已读取字符数（%n）

如下图，scanf先将数据流中的“123456”进行数据类型转换后赋值给变量a。然后将已经从数据流中读取过的字符数赋值给变量b，已经读取过的字符为“1-6”六个字符外加一个空格，所以变量b被赋值为7。



重定向后的返回已读取字符数测试如下：

将main.cpp文件下除“#define TEST\_RETURN\_NUMBER”之外的宏定义全部注释掉，在loongIDE软件上方操作菜单中点击“项目”之后再点击“编译”（快捷键为Ctrl+F9），对源代码进行编译，在其上方操作菜单中点击“调试”之后再点击“运行”（快捷键为F9），对源代码进行调试。

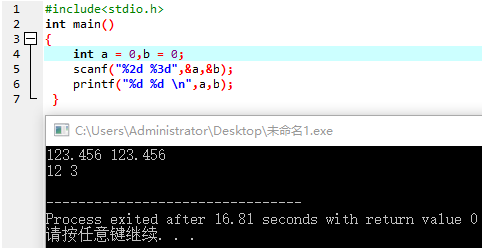
在串口调试助手的发送缓冲区下输入数据“123456 0”后按下回车，再点击发送数据。



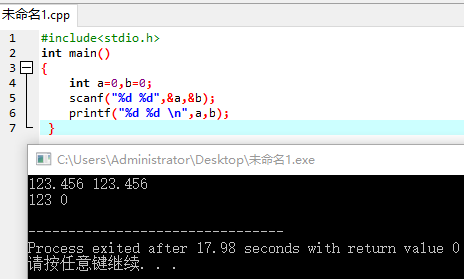
由上图可知，123456已经被赋值给了第一个变量，返回的已读取字符数7被赋值给了第二个变量。其功能和标准库下的scanf函数完全一致。

#### 4．输入十进制整数（%d）

首先是标准库函数scanf对整数位数的控制。同对浮点数的位数控制一样，控制输入整数的位数。如下图中输入第一个十进制整数型变量“%2d”读取了数据流中的“12”赋值给a，同样继续读取后续的数据进行数据类型转换时，虽然没到“%3d”位数上限3，但是读到小数点字符，程序将其识别为不符合数据类型转换的非法字符将退出数据类型转换，所以此时变量b将被赋值为3。



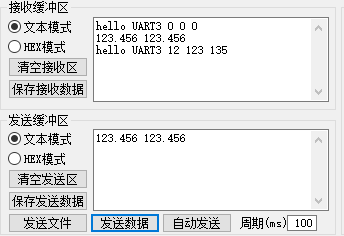
其次是数据的截断。如下图，在第一个数据读入为“123”后读取到小数点字符，中断数据类型转换，将“123”赋值给第一个变量a。接着读取后续数据流，进行第二个变量的数据类型转换，此时数据流还保存着之前的小数点字符，所以直接退出数据类型转换，没有数值赋值给变量b，所以b保持初始值0。



重定向后的十进制整数输入测试如下：

将main.cpp文件下除“#define TEST\_INTEGER\_D”之外的宏定义全部注释掉，在loongIDE软件上方操作菜单中点击“项目”之后再点击“编译”（快捷键为Ctrl+F9），对源代码进行编译，在其上方操作菜单中点击“调试”之后再点击“运行”（快捷键为F9），对源代码进行调试。

在串口调试助手的发送缓冲区下输入数据“123.456 123.456”后按下回车，再点击发送数据。

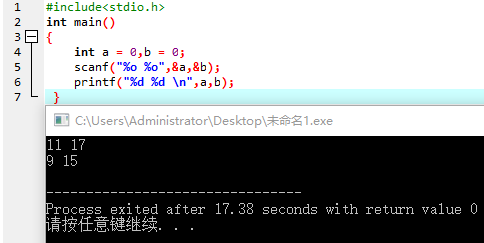


如上图所示，第一个数据接收两位即“123.456”中的12，第二个数据接收第二组“123.456”中的前三位123，计算他们求和的结果并返回12+123=135正确。可见对输入数据位数的控制与标准库函数相同，但是数据截断的位置不同。

串口重定向过程中，对标准库函数scanf做了以下优化：接收完一个整型数据后，将其后边的非法字符删除（一直到空格结束）。比如在此处，代码中定义的是整型数据，使用者误将小数输入也不会造成太大影响。

#### 5．输入八进制整数（%o）

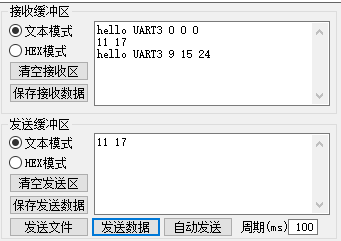
标准库函数scanf支持八进制数输入，输入八进制的“11”与“17”转换为十进制输出即9和15。



重定向后的八进制整数输入测试如下：

将main.cpp文件下除“#define TEST\_INTEGER\_O”之外的宏定义全部注释掉，在loongIDE软件上方操作菜单中点击“项目”之后再点击“编译”（快捷键为Ctrl+F9），对源代码进行编译，在其上方操作菜单中点击“调试”之后再点击“运行”（快捷键为F9），对源代码进行调试。

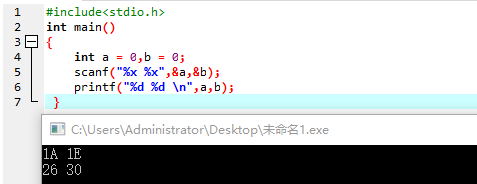
在串口调试助手的发送缓冲区下输入数据“11 17”后按下回车，再点击发送数据。



由上图可见输入八进制数“11”与“17”成功，转换为十进制数“9”和“15”输出成功，并将其求和结果成功转化为十进制“24”计算正确。故重定向后支持八进制整数输入。

#### 6．输入十六进制整数（%x或者%X）

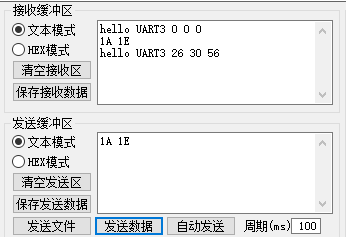
标准库函数scanf支持十六进制数输入，输入十六进制的“1A”与“1E”转换为十进制输出即26和30。



重定向后的十六进制整数输入测试如下：

将main.cpp文件下除“#define TEST\_INTEGER\_X”之外的宏定义全部注释掉，在loongIDE软件上方操作菜单中点击“项目”之后再点击“编译”（快捷键为Ctrl+F9），对源代码进行编译，在其上方操作菜单中点击“调试”之后再点击“运行”（快捷键为F9），对源代码进行调试。

在串口调试助手的发送缓冲区下输入数据“1A 1E”后按下回车，再点击发送数据。

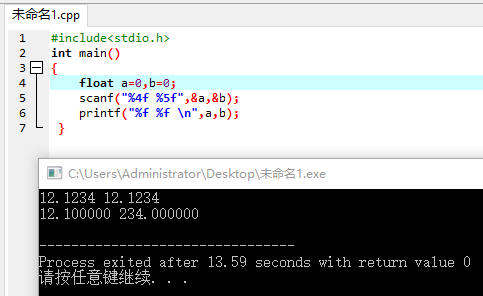


由上图可见输入十六进制数“1A”与“1E”成功，转换为十进制数“26”和“30”输出成功，并将其求和结果成功转化为十进制“56”计算正确。故重定向后支持十六进制整数输入。

#### 7．输入浮点数（%e %E %f %F %g %G）

首先是标准库函数scanf对浮点数位数的控制。标准的scanf函数不能控制浮点数的小数位数，只能控制控制浮点数宽度，如在下图中针对第一个输入的浮点数“%4f”指定其数据宽度为4，则在输入12.1234后只会读取其前四位“12.1”，将其赋值给第一个浮点型变量a。注：读取浮点型数据时小数点也算一位。

其次是数据的截断。在第一个数据读入为“12.1”后，数据流中还剩“234 12.234”，所以程序会继续依次读入之后的数据。当读完“234”后此时才读入三个字符，没到“%5f”要求的长度上限五位，但是遇到了空格字符，此时程序会终止数据类型转换，将“234”赋值给第二个浮点型变量b。

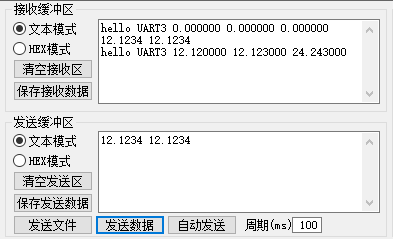


重定向后的浮点数输入测试如下：

将main.cpp文件下除“#define TEST\_FLOAT”之外的宏定义全部注释掉，在loongIDE软件上方操作菜单中点击“项目”之后再点击“编译”（快捷键为Ctrl+F9），对源代码进行编译，在其上方操作菜单中点击“调试”之后再点击“运行”（快捷键为F9），对源代码进行调试。

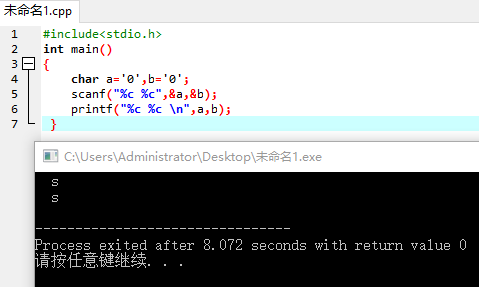
在串口调试助手的发送缓冲区下输入数据“12.1234 12.1234”后按下回车，再点击发送数据。

由图中可知，重定向后的scanf在输入浮点数时，其使用与标准库函数并不相同。主要做了如下优化：其一不再指定输入数据的位数，而是指定精度即小数点之后的位数，使用更加方便，如“%2f”为输入精度为0.01的小数。其二，对数据断点做了修改，例如输入第一个数据“12.1234”虽然只读取到“12.12”赋值给第一个变量，但是他会把后边的多余数据“32”舍弃，防止对下一数据造成影响，所以输入数据为“12.12“与”12.123“，其求和结果亦正确。



#### 8．输入字符（%c）

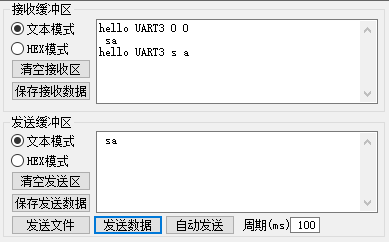
标准库函数scanf支持输入空格，如在下图中，将字符型变量a赋值为空格，将字符型变量b赋值为s。



重定向后的字符输入测试如下：

将main.cpp文件下除“#define TEST\_CHAR”之外的宏定义全部注释掉，在loongIDE软件上方操作菜单中点击“项目”之后再点击“编译”（快捷键为Ctrl+F9），对源代码进行编译，在其上方操作菜单中点击“调试”之后再点击“运行”（快捷键为F9），对源代码进行调试。

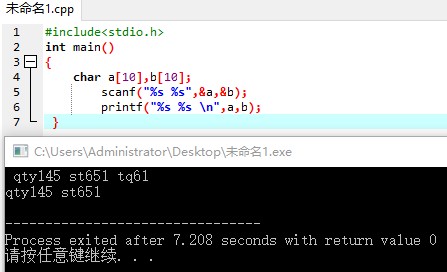
在串口调试助手的发送缓冲区下输入数据“ sa”后按下回车，再点击发送数据。



由此可见，当输入字符类型数据时，重定向后不需要用空格将两个字符隔开（这点与标准库相同），但是重定向后的scanf会自动略过空格，只接收非空数据，所以用不用空格隔开效果都一样。

#### 9．输入字符串（%s）

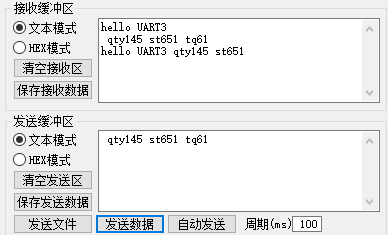
如下图所示，scanf函数读入字符串时会在检测到非空字符时开始，检测到空格后结束。



重定向后的字符串输入测试如下：

将main.cpp文件下除“#define TEST\_STRING”之外的宏定义全部注释掉，在loongIDE软件上方操作菜单中点击“项目”之后再点击“编译”（快捷键为Ctrl+F9），对源代码进行编译，在其上方操作菜单中点击“调试”之后再点击“运行”（快捷键为F9），对源代码进行调试。

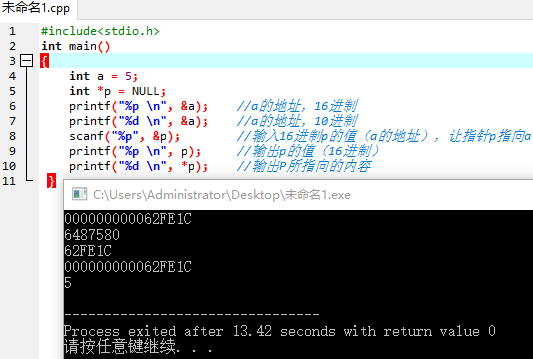
在串口调试助手的发送缓冲区下输入数据“ qty145 st651 tq61”后按下回车，再点击发送数据。



如上图所示， scanf函数读入字符串时会在检测到非空字符时开始，检测到空格后结束。与标准库函数相同。

#### 10．输入指针（%p）

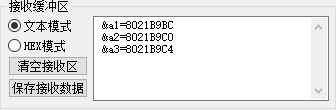
输入16进制地址，即可将指针指向该地址下存放的对应变量。本质很简单，创建一个指针\*p，利用scanf函数将a的地址赋值给p，该作用相当于p=&a，所以指针就指向了a。输出的\*p即该指针指向的a的值。



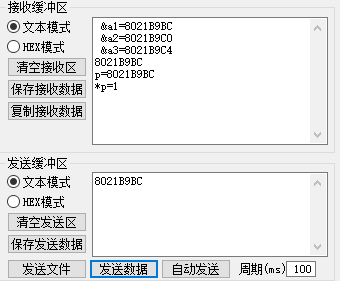
重定向后的指针输入测试如下：

将main.cpp文件下除“#define TEST\_STRING”之外的宏定义全部注释掉，在loongIDE软件上方操作菜单中点击“项目”之后再点击“编译”（快捷键为Ctrl+F9），对源代码进行编译，在其上方操作菜单中点击“调试”之后再点击“运行”（快捷键为F9），对源代码进行调试。

如下图所示，首先在串口调试助手的接收缓冲区下收到三个变量a1、a2、a3的地址。



在串口调试助手的发送缓冲区下输入数据“（上图中a1的地址）”后按下回车，再点击发送数据。



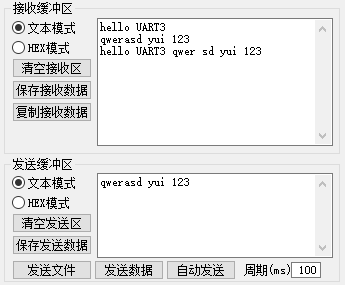
由上图中可以看到，指针p已经被赋值为变量a1的地址，指针\*p所指向的就是变量a1的值。综上所述，重定向之后支持指针输入，与标准库函数用法相同。

#### 11．扫描模式输入字符串（%[^X]）

重定向后扫描模式输入字符串测试如下：

将main.cpp文件下除“#define TEST\_STRING”之外的宏定义全部注释掉，在loongIDE软件上方操作菜单中点击“项目”之后再点击“编译”（快捷键为Ctrl+F9），对源代码进行编译，在其上方操作菜单中点击“调试”之后再点击“运行”（快捷键为F9），对源代码进行调试。

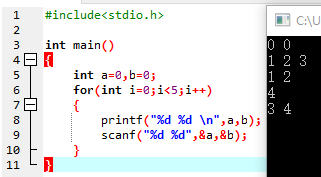
在串口调试助手的发送缓冲区下输入数据“qwerasd yui 123”后按下回车，再点击发送数据。



如上图所示，第一个字符串为“qwer“，第二个字符串为”sd yui 123“。再main.cpp文件下，调用scanf函数scanf("%[^a] %[^ ]",&string1,&string2);表示第一个字符串以字符a结束，第二个字符串以回车换行结束。在输入第一个字符串时，当扫描到字符”a“将不再运行，第一个字符串输入结束。从字符”a“之后的字符”s“一直到回车换行均填入第二个字符串中。所以，第一个字符串为“qwer“，第二个字符串为”sd yui 123“。

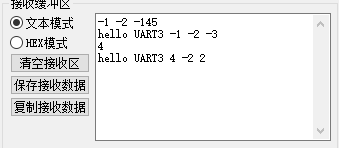
#### 12．数据流的缓存

在标准库函数中，如果scanf控制流定义接收两个数据，实际上向数据流输入了三个数据，那么第三个数据将被放在数据缓存区，当下一次输入的时候再用。



如上图所示：第一次输入了三个数据1 2 3，1被赋值给了a，2被赋值给了b，多出来的3被放在数据缓存区。当下一次输入数据4时，先读取缓存的数据3赋值给a，再将新读取出来的值4赋值给b。

但是串口重定向之后的scanf不支持缓存，会将多出来的数据舍弃掉。



如上图所示，第一次输入整数“-1 -2 -145“，只有-1被赋值给第一个变量，-2被赋值给第二个变量，计算-1+（-2）=-3并输出。由此可见重定向之后支持负数输入。然而多出来的数据-145被舍弃，当下次输入4时，4被赋值给了第一个变量，第二个变量没被赋值所以保持原状为-2，计算4+（-2）=2显示输出。

### 14.7.8 优化方向

scanf()函数的重定向的代码结构可以进一步优化，在此提出了几个可以优化的方向：

1. 将输入整数与浮点数的符号标志位并入flag中。
2. 将\_doscan中定义的变量设为全局变量，之后就可将重复率很高的功能块封装为函数，使代码变得更加简洁。
3. 在\_doscan中定义一个指向数据缓存区的指针，用该指针代替nrchars与ic。