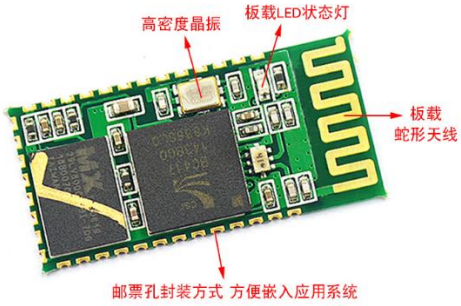
# 通用异步串行接口

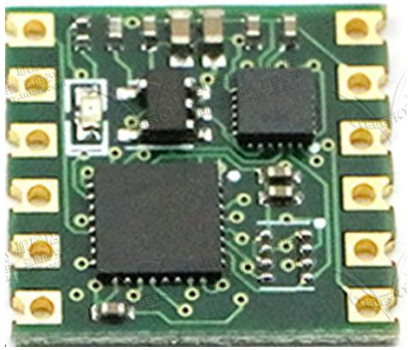
- 异步串行通信是嵌入式系统与外界联系的重要手段是嵌入式系统与外界联系的重要手段

- 按电气标准及协议来分，包括RS-232、RS-422、RS-485、USB、CAN等。


WIFI模块




蓝牙模块



加速度、角速度，地磁模块

激光测距模块


4G模块

以太网模块


数显模块

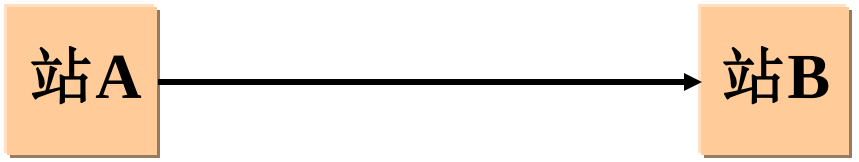

GPS模块

# 异步串行通信

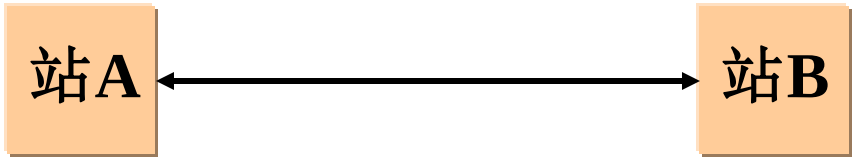
- 异步串行通信时的数据、控制和状态信息都使用同一根信号线传送
- 收发双方必须遵守共同的通信协议（通信规程），才能解决传送速率、信息格式、位同步、字符同步、数据校验等问题
- 异步串行异步通信以字节为单位进行传输，其通信协议是起止式异步通信协议

## ➢异步串行通信的传输制式

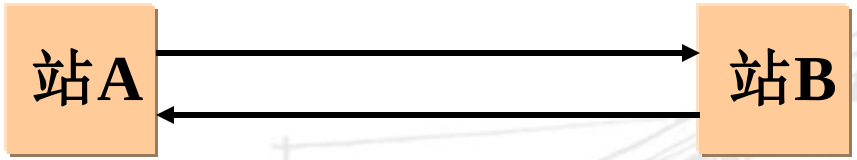
单工




半双工




**全双工**



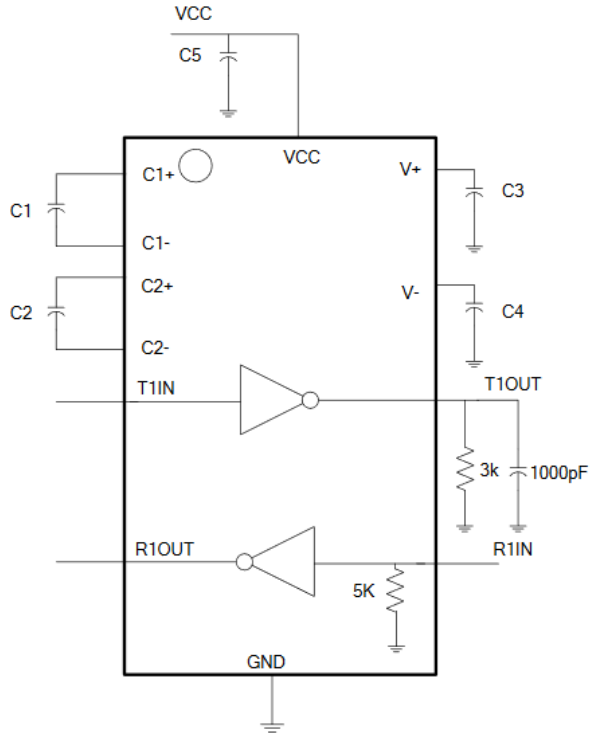
# ➢串行接口标准EIA-232D

- 美国电子工业协会EIA制定的通用标准串行接口
  - 1962年公布，1969年修订
  - 1987年1月正式改名为EIA-232D
- 现已成为数据终端设备DTE（例如计算机）与数据通信设备DCE（例如调制解调器）的标准接口
- 可实现远距离通信，也可近距离连接两台微机
- 属于网络层次结构中的最底层：物理层


主机
网络
应用层
访问网络服务的接口
例如：为操作系统或网络应用程序提供网络服务的接口
常见：Telnet、HTTP、DNS、HTP、SNMP
表示层
提供数据格式转换服务
例如：加密与解密，图片编码和解码，数据压缩和解压缩
常见：URL加密、口令加密、图片编码解码
会话层
建立端连接并提供访问验证和会话管理（session）
例如：使用校验点可使会话在通信失效时从校验点恢复通信
常见：服务器验证登录、断点续传
传输层
提供应用进程间的逻辑通信
例如：建立连接、处理数据包错误、数据包次序
常见：TCP、UDP、SPX、进程、端口（socket）
数据段
(segment)
网络层
为数据在结点之间传输创造逻辑链路、并分组转发数据
例如：对子网间的数据包进行路由选择
常见：路由器、多层交换机、防火墙、IP、IPX、RIP、OSPF
数据包
(package)
数据链路层
在通信的实体间建立数据链路连接
例如：把数据分帧、并处理流控制、物理地址寻址、重发等
常见：网卡、网桥、二层交换机
帧
(frame)
物理层
为数据端设备提供原始比特流的传输的通路
例如：网络通信的数据传输介质、由电缆与设备共同构成
常见：中继机、集线器、网线、HUB、RJ-45标准
比特
(bit)

# ➢EIA-232D的电气特性

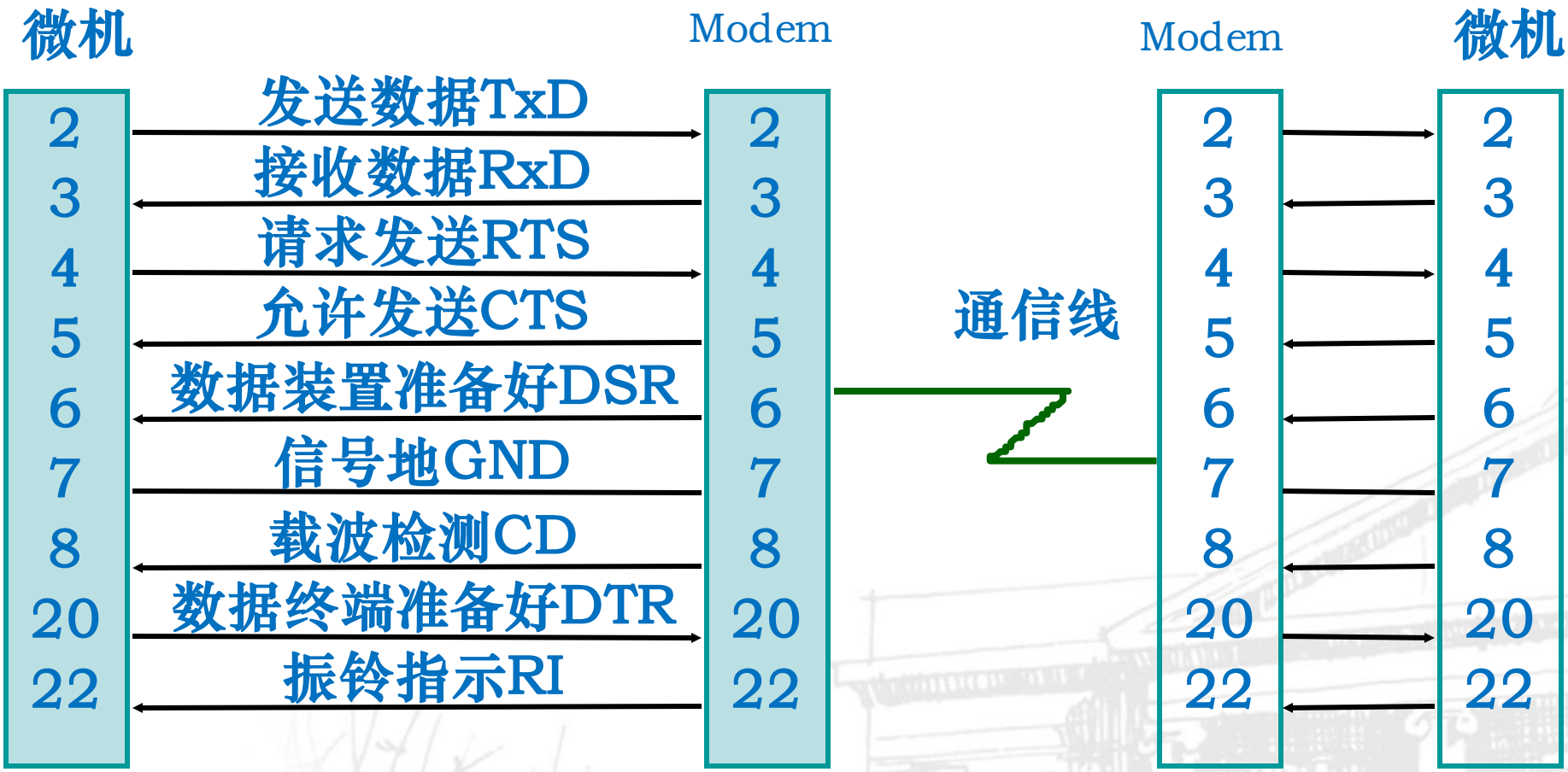
- 232D接口采用EIA电平
  - 高电平为＋3V～＋15V
  - 低电平为－3V～－15V
  - 实际常用±12V或±15V




相互转换

- 标准TTL电平
  - 高电平：＋2.4V～＋5V
  - 低电平：0V～0.4V

## ➢EIA-232D的连接

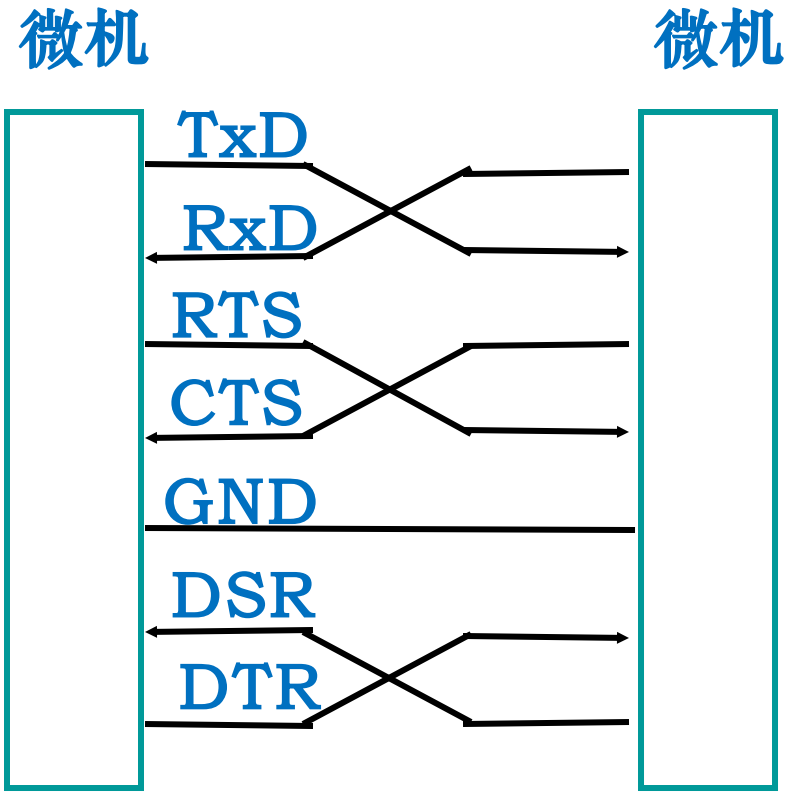

微机
Modem
Modem
微机
2
3
4
5
6
7
8
20
22
发送数据TxD
接收数据RxD
请求发送RTS
允许发送CTS
数据装置准备好DSR
信号地GND
载波检测CD
数据终端准备好DTR
振铃指示RI
通信线

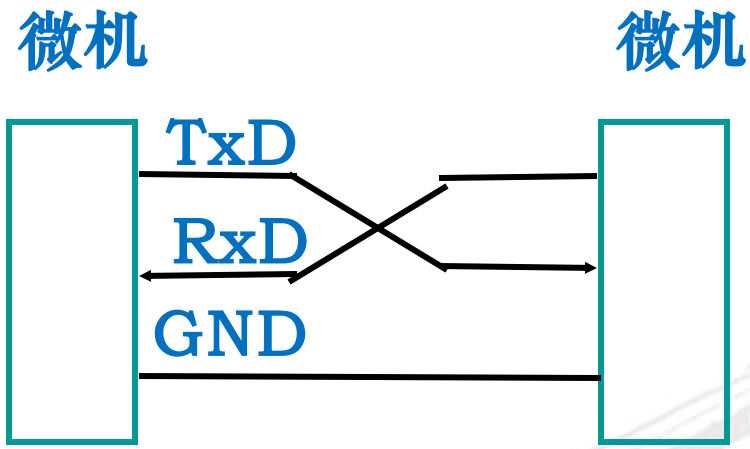
- **TxD：发送数据**
  - 串行数据的发送端
- **RxD：接收数据**
  - 串行数据的接收端
- RTS：请求发送
  - 当数据终端设备准备好送出数据时，就发出有效的RTS信号，用于通知数据通信设备准备接收数据
- CTS：清除发送（允许发送）
  - 当数据通信设备已准备好接收数据终端设备的传送数据时，发出CTS有效信号来响应RTS信号
- DTR：数据终端准备好
  - 通常当数据终端设备一加电，该信号就有效，表明数据终端设备准备就绪
- DSR：数据装置准备好
  - 通常表示数据通信设备（即数据装置）已接通电源连到通信线路上，并处在数据传输方式

- GND：信号地
  - 为所有的信号提供一个公共的参考电平
- 保护地（机壳地）
  - 起屏蔽保护作用的接地端，一般应参照设备的使用规定，连接到设备的外壳或大地
- CD：载波检测（DCD）
  - 当本地调制解调器接收到来自对方的载波信号时，该引脚向数据终端设备提供有效信号
- RI：振铃指示
  - 当调制解调器接收到对方的拨号信号期间，该引脚信号作为电话铃响的指示、保持有效

## ➢EIA-232D的连接

– 微机利用232接口直接连接进行短距离通信。这种连接不使用调制解调器，所以被称为零调制解调器（Null Modem）连接

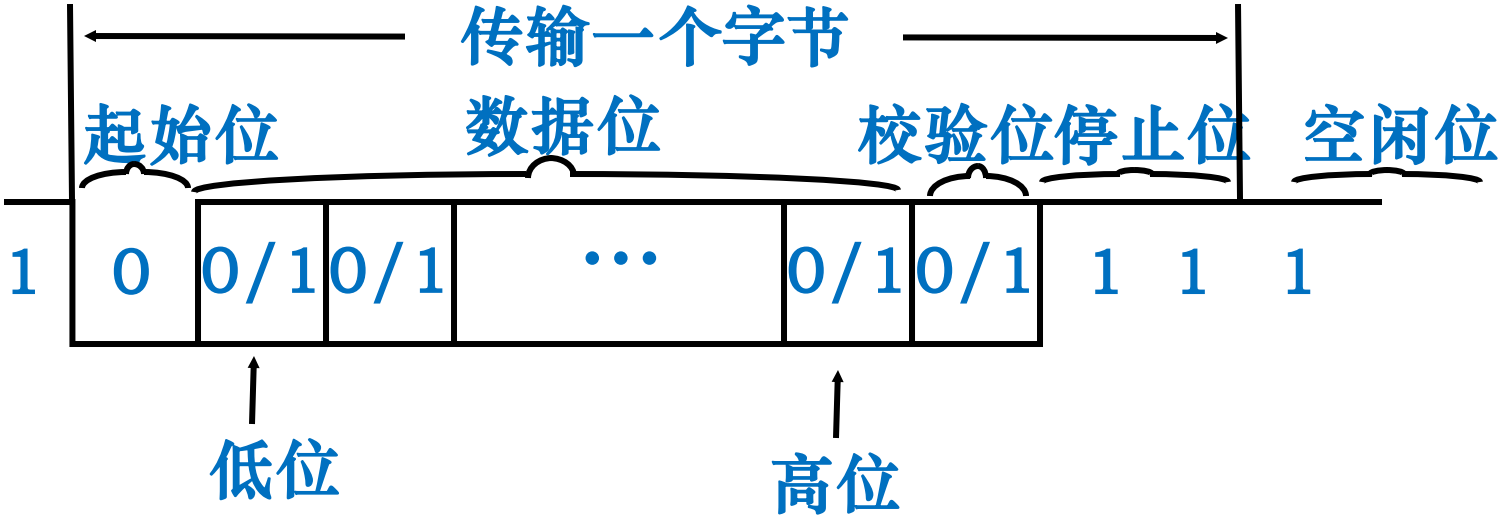



不使用联络信号的3线相连方式

通信比较可靠
所用连线较多，不如前者经济

## ➢起止式异步通信协议




**起始位**——每个字符开始传送的标志，起始位采用逻辑0电平

**数据位**——数据位紧跟着起始位传送。由5～8个二进制位组成，低位先传送

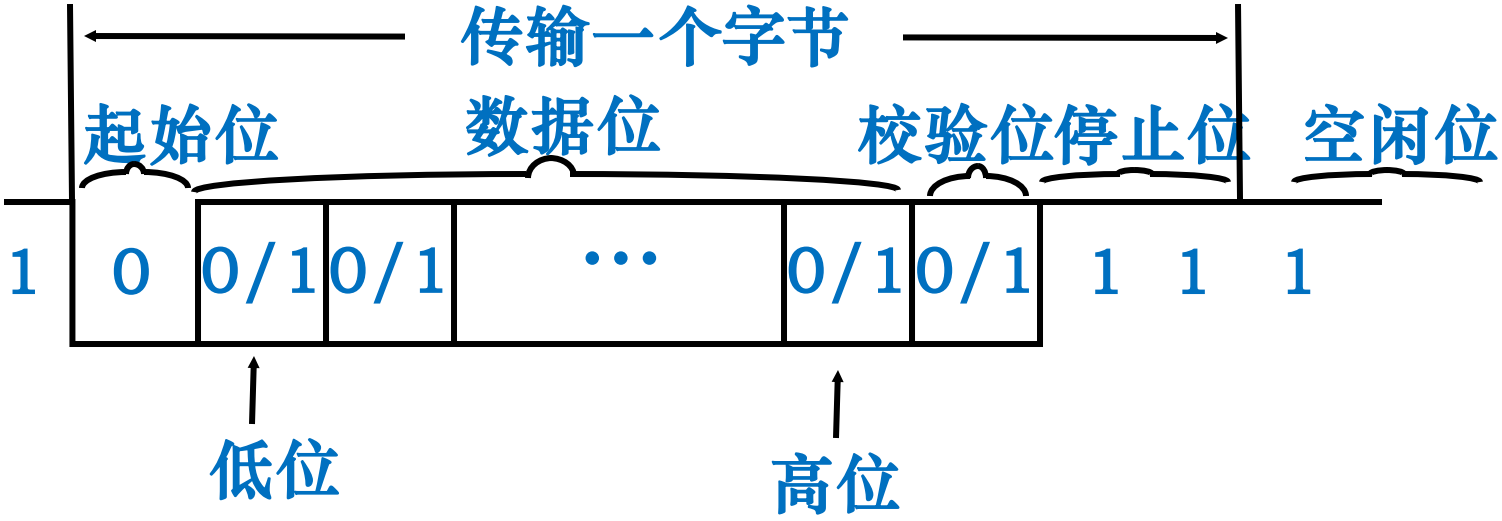
**校验位**——用于校验是否传送正确；可选择奇检验、偶校验或不传送校验位

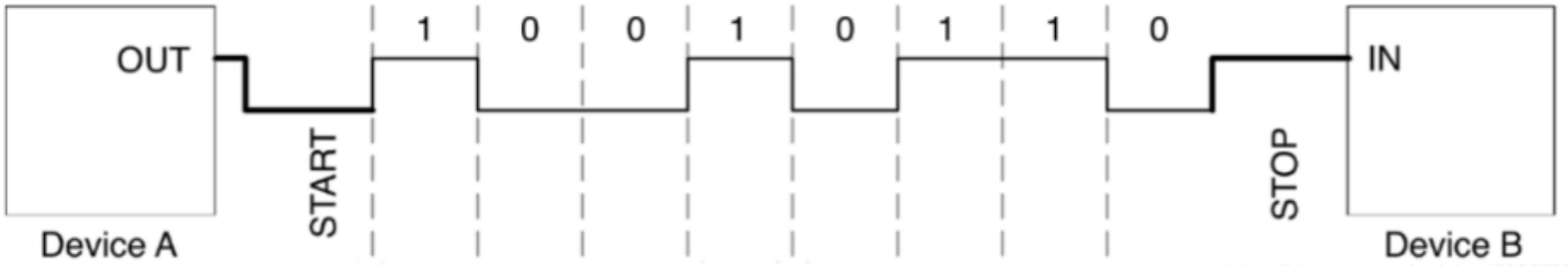
设置一个奇偶校验位，用它使这组代码中“1”的个数为奇数或偶数。若用奇校验，则当接收端收到这组代码时，校验“1”的个数是否为奇数，从而确定传输代码的正确性。

**停止位**——表示该字符传送结束；停止位采用逻辑1电平，可选择1、1.5或2位。

**空闲位**——传送字符之间的逻辑1电平，表示没有进行传送。

## ➢起止式异步通信协议



## ➢数据传输速率

- 数据传输速率也称比特率（Bit Rate）：每秒传送的比特(bit)数
- 波特率（Baud Rate）表示每秒钟传送的码元符号的个数
- 当进行二进制数码传输，且每位时间长度相等时，比特率等于波特率（Baud Rate）
- 过去，串行异步通信的数据传输速率限制在50 bps到9600 bps之间。现可以达到115200 bps或更高

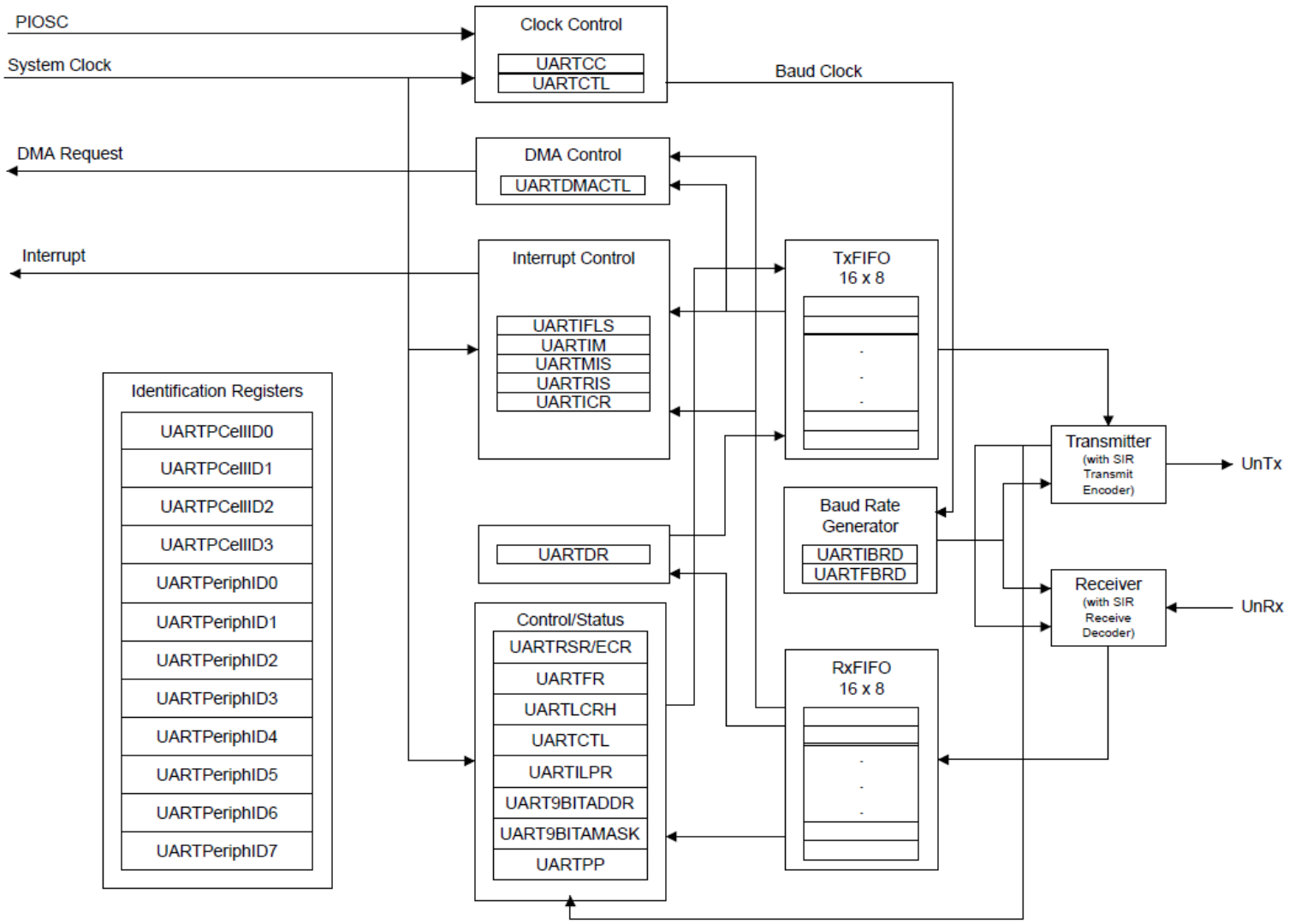
➢练习：

通用串口使用偶校验，两个停止位，波特率为115200bps，发数据 0x3C时，画出Tx上的波形。

## ➢TM4C1294中的UART模块

- 可编程波特率，最高可达7.5Mbps/15Mbps
- 发送和接受都有16*8 的先入先出寄存器（FIFO），深度可编程
- 可编程数据位、起始位、校验位
- 红外接口
- 多样化的中断系统
- 带DMA接口
- 流控制与状态信号

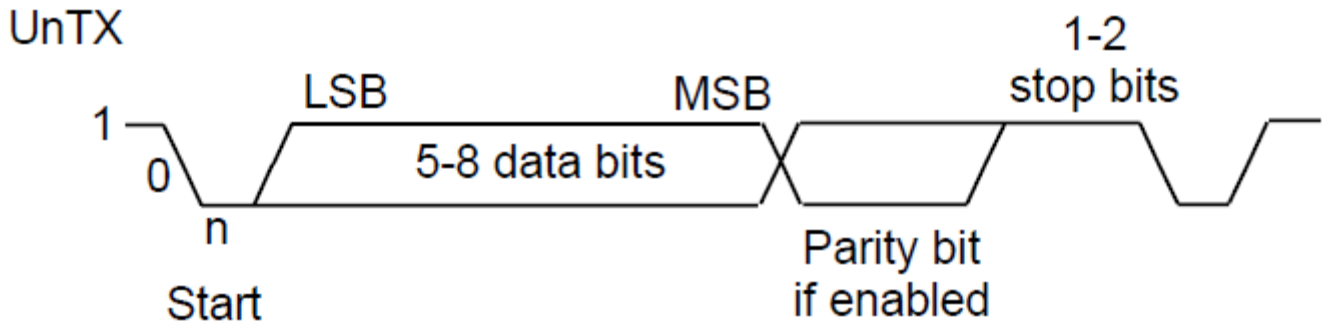
# UART模块结构


PIOSC
System Clock
DMA Request
Interrupt
Clock Control
UARTCC
UARTCTL
Baud Clock
DMA Control
UARTDMACTL
Interrupt Control
UARTIFLS
UARTIM
UARTMIS
UARTRIS
UARTICR
TxFIFO
16 x 8
Identification Registers
UARTPCellID0
UARTPCellID1
UARTPCellID2
UARTPCellID3
UARTPeriphID0
UARTPeriphID1
UARTPeriphID2
UARTPeriphID3
UARTPeriphID4
UARTPeriphID5
UARTPeriphID6
UARTPeriphID7
UARTDR
Baud Rate
Generator
UARTIBRD
UARTFBRD
Transmitter
(with SIR
Transmit
Encoder)
UnTx
Receiver
(with SIR
Receive
Decoder)
UnRx
Control/Status
UARTRSR/ECR
UARTFR
UARTLCRH
UARTCTL
UARTILPR
UART9BITADDR
UART9BITAMASK
UARTPP
RxFIFO
16 x 8

## ➢功能描述

重启后UART发送和接受功能就有效

## ➢数据收发时序



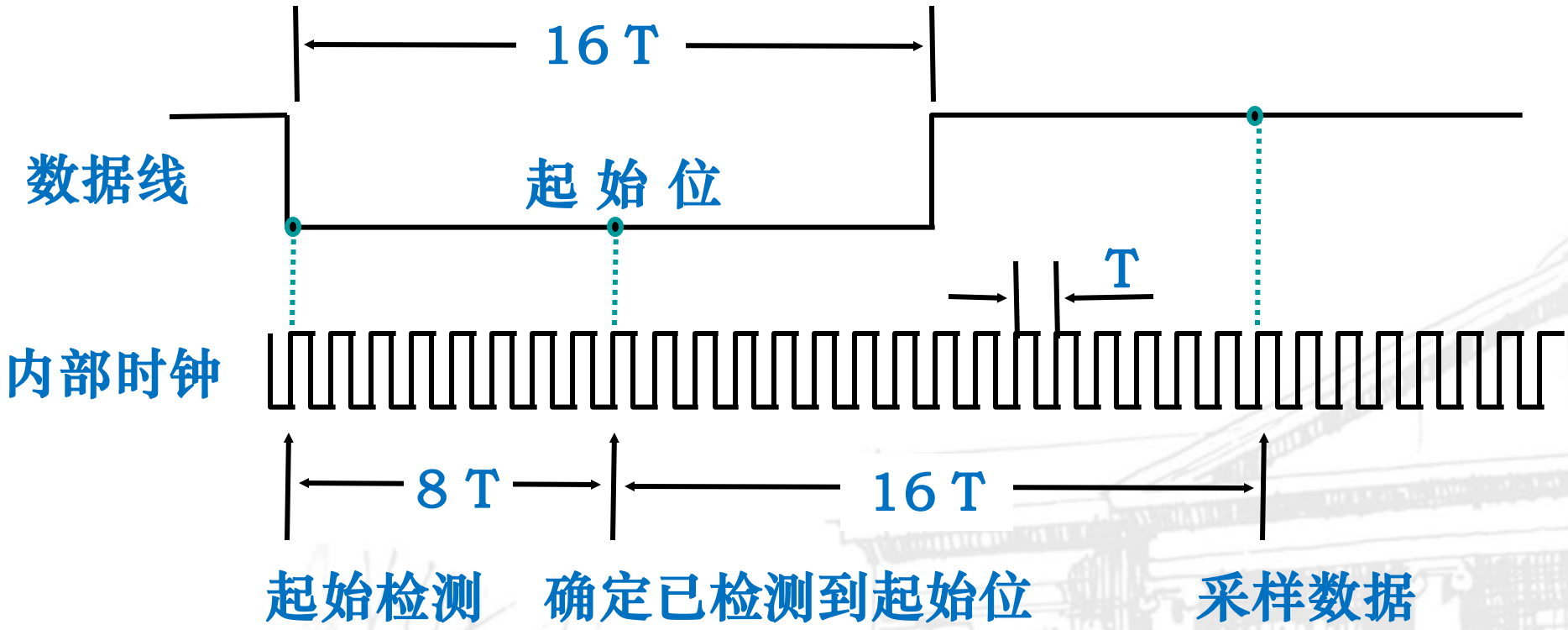
## ➢数据发送

- 先把要发送的数据写入到发送FIFO（TxFIFO）中，向UARTDR寄存器中写入数据，这个数据就会自动填入到TxFIFO中。

- 只要TxFIFO中有数据，UART模块就会不停的把TxFIFO中的数据按照异步串行数据的通信格式发送出去，直到TxFIFO中的数据发送完为止

## ➢数据接收

- 收到起始信号后，在第8个（或者第4个）内部参考时钟周期后采样引脚上的电平，如果是低电平，则确认收到起始位
- 之后每16个（或者第8个）内部参考时钟周期(T)采样一次数据，存到RxFIFO中



## ➢TM4C1294中的UART模块使用方法

- 1. 使能UART模块的时钟（操作RCGCUART 寄存器）
- 2.使能UART模块引脚用到的GPIO模块（使用RCGCGPIO 寄存器）
- 3. 配置GPIO的复用功能，使其与UART模块相连
- 4. 计算配置波特率分频器UARTIBRD和UARTFBRD
- 5. 禁用UART模块，配置数据格式等参数（UARTLCRH），配置时钟源UARTCC，使能UART模块
- 6. 使用UART模块发送和接收数据

# ➢TM4C1294中的UART模块使用方法

- 第一步：使能UART模块的时钟（操作RCGCUART 寄存器）

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
```

- 第二步：使能UART模块引脚用到的GPIO模块（使用RCGCGPIO 寄存器）

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
```

# ➢TM4C1294中的UART模块使用方法

## –第三步：配置GPIO的复用功能，使其与UART模块相连

设置GPIOAFSEL寄存器，使能引脚的复用功能。可以调用GPIOPinTypeUART函数实现

```
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
```

```
void
GPIOPinTypeUART(uint32_t ui32Port, uint8_t ui8Pins)
{
    ASSERT(_GPIOBaseValid(ui32Port));
    GPIODirModeSet(ui32Port, ui8Pins, GPIO_DIR_MODE_HW);
    // Set the pad(s) for standard push-pull operation.
    GPIOPadConfigSet(ui32Port, ui8Pins, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD);
}
```

# ➢TM4C1294中的UART模块使用方法

## –第三步：配置GPIO的复用功能，使其与UART模块相连

使用GPIOPCTL的PMCx位的配置PA0和PA1的复用功能，复用为UART0模块的Rx和Tx引脚

| IO | Pin | Analog or Special Function[a] | Digital Function (GPIOPCTL PMCx Bit Field Encoding)[b] | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 11 | 13 | 14 | 15 |
| PA0 | 33 | - | U0Rx | I2C9SCL | T0CCP0 | - | - | - | CAN0Rx | - | - | - | - | - |
| PA1 | 34 | - | U0Tx | I2C9SDA | T0CCP1 | - | - | - | CAN0Tx | - | - | - | - | - |

可以调用GPIOPinConfigure函数实现：

```
GPIOPinConfigure(GPIO_PA0_U0RX);
GPIOPinConfigure(GPIO_PA1_U0TX);
```

## ➢TM4C1294中的UART模块使用方法

### – 第三步：配置GPIO的复用功能，使其与UART模块相连

使用GPIOPCTL的PMCx位的配置PA0和PA1的复用功能，复用为UART0模块的Rx和Tx引脚，可以调用GPIOPinConfigure函数实现：

```
void
GPIOPinConfigure(uint32_t ui32PinConfig)
{
    uint32_t ui32Base, ui32Shift;
    ASSERT(((ui32PinConfig >> 16) & 0xff) < 18);
    ASSERT(((ui32PinConfig >> 8) & 0xe3) == 0);
    ui32Base = (ui32PinConfig >> 16) & 0xff;

    if(HWREG(SYSCTL_GPIOHBCTL) & (1 << ui32Base))
    {
        ui32Base = g_pui32GPIOBaseAddrs[(ui32Base << 1) + 1];
    }
    else
    {
        ui32Base = g_pui32GPIOBaseAddrs[ui32Base << 1];
    }
    ui32Shift = (ui32PinConfig >> 8) & 0xff;
    HWREG(ui32Base + GPIO_O_PCTL) = ((HWREG(ui32Base + GPIO_O_PCTL) &
                                      ~(0xf << ui32Shift)) | ((ui32PinConfig & 0xf) << ui32Shift));
```

其中**GPIO_PA0_U0RX**和GPIO_PA1_U0TX的定义如下：

```
#define GPIO_PA0_U0RX          0x00000001
#define GPIO_PA0_I2C9SCL       0x00000002
#define GPIO_PA0_T0CCP0        0x00000003
#define GPIO_PA0_CAN0RX        0x00000007

#define GPIO_PA1_U0TX          0x00000401
#define GPIO_PA1_I2C9SDA       0x00000402
#define GPIO_PA1_T0CCP1        0x00000403
#define GPIO_PA1_CAN0TX        0x00000407
```

# ➢TM4C1294中的UART模块使用方法

## – 第四步：计算配置波特率分频器UARTIBRD和UARTFBRD

UART模块的波特率发生器

由一个22位的分频器，将系统时钟分频为所需要的时钟

UARTIBRD的低16位组成分频器的整数位，UARTFBRD的低6位组成分频器的小数位

分频器与波特率的关系为

$$BRD = BRDI + BRDF = UARTSysClk / (ClkDiv * Baud\ Rate)$$

其中，

Baud Rate 为波特率

ClkDiv为发送/接收每位所用的内部时钟的个数一般为16，高速模式下，可以配置为8。

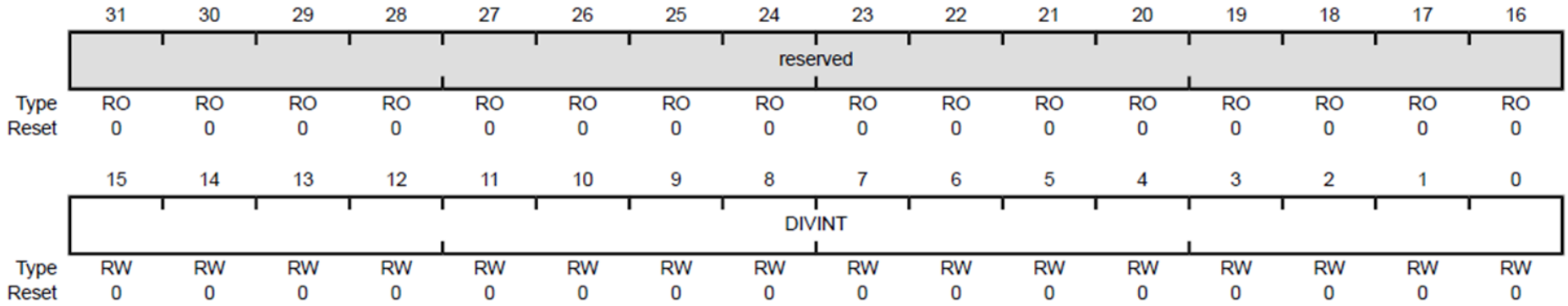
BRD为波特率分频系数，

BRDI为波特率分频系数的整数部分，直接填入到UARTIBRD寄存器中

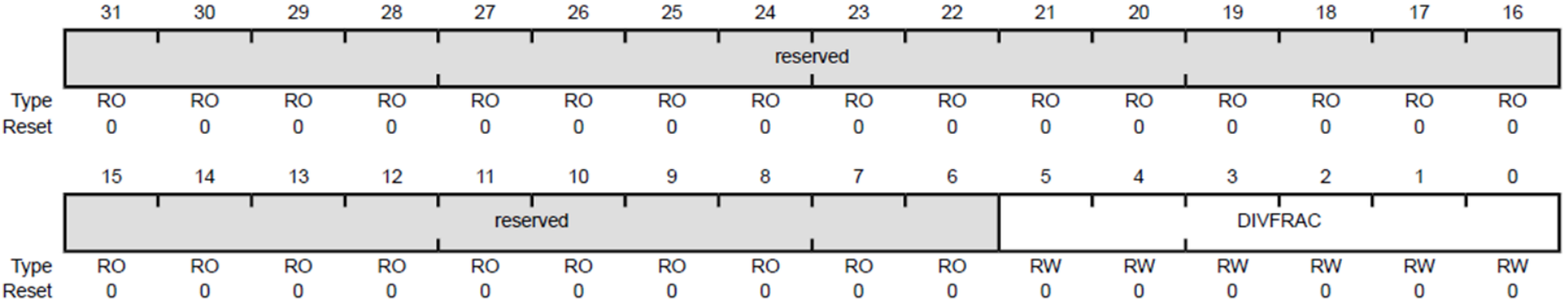
BRDF为波特率分频系数的小数部分，计算方式为：

UARTFBRD[DIVFRAC] = integer(BRDF * 64 + 0.5)

## UART Integer Baud-Rate Divisor (UARTIBRD)




## UART Fractional Baud-Rate Divisor (UARTFBRD)



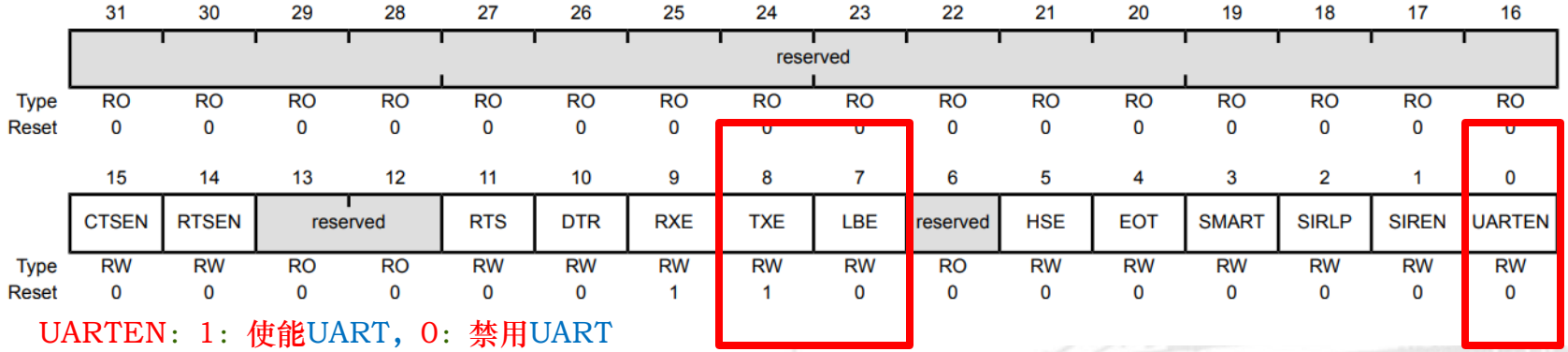
# ➢TM4C1294中的UART模块使用方法

## ➢第五步：禁用UART模块，配置数据格式等参数（UARTLCRH），配置时钟源UARTCC，使能UART模块

使能和禁止UART模块，由UART控制寄存器UARTCTL控制。

UART Control (UARTCTL)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x030
Type RW, reset 0x0000.0300




UARTEN：1：使能UART，0：禁用UART

RXE：1：接收使能，同时UARTEN 也要设置为1，0：禁用接收功能

TXE：1：发送使能，同时UARTEN 也要设置为1，0：禁用发送功能

HSE，1：高速模式，即发送或接收一个位需要8个内部时钟，最高波特率位15Mbsp。0：常规模式，即发送或接收一个位需要16个内部时钟，最高波特率位7.5Mbps。

## ➢TM4C1294中的UART模块使用方法

### ➢第五步：禁用UART模块，配置数据格式等参数（UARTLCRH），配置时钟源UARTCC，使能UART模块

使能和禁止UART模块，由UART控制寄存器UARTCTL控制。

禁用UART模块，调用函数UARTDisable，其具体实现方式为：

```
void
UARTDisable(uint32_t ui32Base)
{
    ASSERT(_UARTBaseValid(ui32Base));
    while(HWREG(ui32Base + UART_O_FR) & UART_FR_BUSY)
    {
    }

    HWREG(ui32Base + UART_O_LCRH) &= ~(UART_LCRH_FEN);
    HWREG(ui32Base + UART_O_CTL) &= ~(UART_CTL_UARTEN | UART_CTL_TXE |
                                      UART_CTL_RXE);
}
```

//UARTDisable函数将UARTEN、RXE和TXE这三位置**0**

## ➢TM4C1294中的UART模块使用方法

### ➢第五步：禁用UART模块，配置数据格式等参数（UARTLCRH），配置时钟源UARTCC，使能UART模块

使能和禁止UART模块，由UART控制寄存器UARTCTL控制。

使能UART模块，调用函数UARTEnable，其具体实现方式为：

```
void
UARTEnable(uint32_t ui32Base)
{
    HWREG(ui32Base + UART_O_LCRH) |= UART_LCRH_FEN;

    //
    // Enable RX, TX, and the UART.
    //                                                    UART_CTL_RXE);
    HWREG(ui32Base + UART_O_CTL) |= (UART_CTL_UARTEN | UART_CTL_TXE |

}
```

//UARTEnable函数将将UARTEN、RXE和TXE这三位置**1**

# ➢TM4C1294中的UART模块使用方法

## ➢第五步：禁用UART模块，配置数据格式等参数（UARTLCRH），配置时钟源UARTCC，使能UART模块

UARTDisable还需要等待数据发送完成，使用UART状态标志寄存器UARTFR查看UART的工作状态.

UART状态标志寄存器UARTFR：

UART Flag (UARTFR)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x018
Type RO, reset 0x0000.0090

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| reserved | | | | | | | | | | | | | | | |
| RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| reserved | | | | | | | RI | TXFE | RXFF | TXFF | RXFE | BUSY | DCD | DSR | CTS |
| RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

BUSY：0：UART模块空闲，1：UART模块正忙

# ➢TM4C1294中的UART模块使用方法

## ➢第五步：禁用UART模块，配置数据格式等参数（UARTLCRH），配置时钟源UARTCC，使能UART模块

### UART线路控制寄存器UARTLCRH：

UART Line Control (UARTLCRH)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x02C
Type RW, reset 0x0000.0000

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | reserved | | | | | | | | | | | | | | | |
| Type | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | reserved | | | | | | | | SPS | WLEN | | FEN | STP2 | EPS | PEN | BRK |
| Type | RO | RO | RO | RO | RO | RO | RO | RO | RW | RW | RW | RW | RW | RW | RW | RW |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

FEN：0，禁用FIFO，即FIFO的深度为1， 1：启用FIFO

UARTDisable函数和UARTEnable函数都操作了UART线路控制寄存器UARTLCRH的FEN位，UARTDisable函数禁用UART时，将FIFO关闭，而UARTEnable函数开启UART时，将FIFO打开。

## ➢TM4C1294中的UART模块使用方法

### ➢第五步：禁用UART模块，配置数据格式等参数（UARTLCRH），配置时钟源UARTCC，使能UART模块

**UART线路控制寄存器UARTLCRH：**

禁用UART后，需要设置UART的数据传输格式：设置数据格式:

WLEN：数据位数：0：5位

1：6位

2：7位

3：8位

STP2：停止位选择，0：一个停止位，1：2个停止位

EPS：奇偶校验选择：0：奇校验，1：偶校验，需要PEN位为1才有效

PEN：校验位使能：0：关闭奇偶校验功能，无校验位。1：开启奇偶校验功能，有校验位

SPS：强制校验位 1：如果SPS、EPS、PEN都为1，则校验位强制为0，如果SPS、PEN都为1，EPS为0，则校验位强制为1。

如果SPS为0，则禁用强制校验位功能，校验位由EPS、PEN和数据决定。

设置完成后，调用UARTEnable函数开启UART，就可以控制UART收发数据了。

# ➢TM4C1294中的UART模块使用方法

## ➢第五步：禁用UART模块，配置数据格式等参数（UARTLCRH），配置时钟源UARTCC，使能UART模块

TivaWare提供了UARTConfigSetExpClk函数，一次性初始化UART模块：

```
UARTConfigSetExpClk(UART0_BASE, g_ui32SysClock, 115200,
(UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));
```

表示初始化UART0模块，设置波特率位115200， 数据位8位，一个停止位，无校验位。

```
void UARTConfigSetExpClk(uint32_t ui32Base, uint32_t ui32UARTClk, uint32_t ui32Baud,
uint32_t ui32Config)
{
    uint32_t ui32Div;
   UARTDisable(ui32Base); //先关闭UART模块

    if((ui32Baud * 16) > ui32UARTClk)
    {
HWREG(ui32Base + UART_O_CTL) |= UART_CTL_HSE;

        ui32Baud /= 2;
    }
    else
    {
        HWREG(ui32Base + UART_O_CTL) &= ~(UART_CTL_HSE);
    }

   ui32Div = (((ui32UARTClk * 8) / ui32Baud) + 1) / 2;

    HWREG(ui32Base + UART_O_IBRD) = ui32Div / 64;
    HWREG(ui32Base + UART_O_FBRD) = ui32Div % 64;
    HWREG(ui32Base + UART_O_LCRH) = ui32Config;  //UART的配置参数
    HWREG(ui32Base + UART_O_FR) = 0;
    UARTEnable(ui32Base); //使能UART模块
}
```

ui32Config为UART的配置参数，写入到UART_O_LCRH寄存器中，可用的选项有：

数据长度：

```
#define UART_CONFIG_WLEN_8      0x00000060  // 8 bit data
#define UART_CONFIG_WLEN_7      0x00000040  // 7 bit data
#define UART_CONFIG_WLEN_6      0x00000020  // 6 bit data
#define UART_CONFIG_WLEN_5      0x00000000  // 5 bit data
```

停止位设置：

```
#define UART_CONFIG_STOP_ONE    0x00000000  // One stop bit
#define UART_CONFIG_STOP_TWO    0x00000008  // Two stop bits
```

校验位设置：

```
#define UART_CONFIG_PAR_NONE    0x00000000  // No parity
#define UART_CONFIG_PAR_EVEN    0x00000006  // Even parity
#define UART_CONFIG_PAR_ODD     0x00000002  // Odd parity
#define UART_CONFIG_PAR_ONE     0x00000082  // Parity bit is one
#define UART_CONFIG_PAR_ZERO    0x00000086  // Parity bit is zero
```

# ➢TM4C1294中的UART模块使用方法

## ➢第六步：使用UART模块发送和接收数据

### UART数据寄存器 UARTDR

UART Data (UARTDR)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x000
Type RW, reset 0x0000.0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| reserved | | | | | | | | | | | | | | | |
| RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| reserved | | | | OE | BE | PE | FE | DATA | | | | | | | |
| RO | RO | RO | RO | RO | RO | RO | RO | RW | RW | RW | RW | RW | RW | RW | RW |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

DATA：数据位

如果要**发送数据**，就把数据写入UARTDR 寄存器的DATA域中，写入DATA的数据会被转移到发送FIFO（TxFIFO）中，然后UART模块开始发送数据，直到TxFIFO中的数据全部发送完成为止。

如果要**读取数据**，读DATA域就会把收到的数据从接收FIFO（RxFIFO）中读出来，读数据的时候，不仅有DATA域，还有OE、BE、PE、FE四个状态位，用来指示本次接收的数据是否有错误。

OE：溢出错误。

BE: 中断错误。

PE: 奇偶校验错误。

FE：帧错误。

# ➢TM4C1294中的UART模块使用方法

## 第六步：使用UART模块发送和接收数据

在发送和接收数据之前，要先查看状态标志寄存器UARTFR，防止TxFIFO溢出、及时读取RxFIFO中的数据，防止RxFIFO溢出。

### UART状态标志寄存器UARTFR

UART Flag (UARTFR)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x018
Type RO, reset 0x0000.0090

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| reserved | | | | | | | | | | | | | | | |
| RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| reserved | | | | | | | RI | TXFE | RXFF | TXFF | RXFE | BUSY | DCD | DSR | CTS |
| RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

**BUSY**忙碌标志：0：UART模块空闲，1：UART模块正忙
**TXFF** TxFIFO满标志：0：TxFIFO未满，可以继续填入数据；1：TxFIFO满。
**RXFE** RxFIFO空标志：0：RxFIFO中有数据，需要把数据取出来；1，RxFIFO为空，还没有收到数据

# ➢TM4C1294中的UART模块使用方法

## ➢第六步：使用UART模块发送和接收数据

TivaWare提供了UARTCharPut函数和UARTCharPutNonBlocking函数发送一个字节的数据。

UARTCharPut函数的实现方式如下：**阻塞模式**

```
void
UARTCharPut(uint32_t ui32Base, unsigned char ucData)
{
    ASSERT(_UARTBaseValid(ui32Base));
    while(HWREG(ui32Base + UART_O_FR) & UART_FR_TXFF)
    {
    }
    HWREG(ui32Base + UART_O_DR) = ucData;
}
```

查看UARTFR的TXFF是否为1，如果TxFIFO已满，则等待TxFIFO有空位后，再把数据写入到UARTDR寄存器中。如果一次要发送的数据很多，很快就会把TxFIFO填满，UARTCharPut这是阻塞了CPU的运行，CPU就只能等待，降低了CPU的效率，这种方式称为阻塞模式。

## ➢TM4C1294中的UART模块使用方法

### 第六步：使用UART模块发送和接收数据

TivaWare提供了UARTCharPut函数和UARTCharPutNonBlocking函数发送一个字节的数据。

UARTCharPutNonBlocking函数的实现方式如下：

非阻塞模式

```
bool
UARTCharPutNonBlocking(uint32_t ui32Base, unsigned char ucData)
{
    ASSERT(_UARTBaseValid(ui32Base));
    if(!(HWREG(ui32Base + UART_O_FR) & UART_FR_TXFF))//查看UARTFR的TXFF是否为1，
                                                     //如果TxFIFO未满，就把数据
                                                     //填入UARTDR，并返回true
    {
        HWREG(ui32Base + UART_O_DR) = ucData;
        return(true);
    }
    else
    {
        return(false);//如果TxFIFO满，则返回false，表示数据发送失败
    }
}
```

# ➢TM4C1294中的UART模块使用方法

## –第六步：使用UART模块发送和接收数据

TivaWare提供了UARTCharGet函数和UARTCharGetNonBlocking函数接收一个字节的数据。

UARTCharGet函数的实现方式如下：阻塞模式

```
int32_t
UARTCharGet(uint32_t ui32Base)
{
    ASSERT(_UARTBaseValid(ui32Base));
    while(HWREG(ui32Base + UART_O_FR) & UART_FR_RXFE)
    {
    }
    return(HWREG(ui32Base + UART_O_DR));
}
```

# ➢TM4C1294中的UART模块使用方法

## –第六步：使用UART模块发送和接收数据

TivaWare提供了UARTCharGet函数和UARTCharGetNonBlocking函数接收一个字节的数据。

UARTCharGetNonBlocking函数的实现方式如下：非阻塞模式

```
int32_t
UARTCharGetNonBlocking(uint32_t ui32Base)
{
    ASSERT(_UARTBaseValid(ui32Base));
    if(!(HWREG(ui32Base + UART_O_FR) & UART_FR_RXFE))
    {
        return(HWREG(ui32Base + UART_O_DR));
    }
    else
    {
        return(-1);
    }
}
```

# ➢TM4C1294中的UART模块使用方法

## –UART模块的初始化及发送程序示例：

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
GPIOPinConfigure(GPIO_PA0_U0RX);
GPIOPinConfigure(GPIO_PA1_U0TX);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

UARTConfigSetExpClk(UART0_BASE, g_ui32SysClock, 115200,
                            (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                                 UART_CONFIG_PAR_NONE));
    uint32_t cThisChar;
    while(1)
    {
        cThisChar = UARTCharGet(UART0_BASE);

        UARTCharPut(UART0_BASE, cThisChar);

   }
```

这段程序的作用是什么？

# ➢TM4C1294中的UART模块使用方法

## – UART模块的中断使用方法：

UART模块支持多种中断，可以配合中断响应，提高程序的执行效率。
UART中断屏蔽寄存器UARTIM，可以设置开启或者关闭哪些中断。

UART Interrupt Mask (UARTIM)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x038
Type RW, reset 0x0000.0000

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | reserved | | | | | | | | | | | | | | DMATXIM | DMARXIM |
| Type | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RW | RW |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | reserved | | | 9BITIM | EOTIM | OEIM | BEIM | PEIM | FEIM | RTIM | TXIM | RXIM | DSRIM | DCDIM | CTSIM | RIIM |
| Type | RO | RO | RO | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW | RW |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**RXIM：接收中断**。如果RxFIFO中有数据，且数据的数量超过了UARTIFLS寄存器定义的值，就产生中断。

**RTIM：接收超时中断**。如果RxFIFO中有数据，且连续32个内部周期没有收到新数据，则产生中断。

**TXIM：发送中断**。如果UARTCTL的EOT设置为1，那么等所有的数据都发送完，就产生中断。如果UARTCTL的EOT设置为0，那么发送FIFO中的数据个数，少于UARTIFLS寄存器定义的值，就产生中断。

# ➢TM4C1294中的UART模块使用方法

## – UART模块的中断使用方法：

UART模块支持多种中断，可以配合中断响应，提高程序的执行效率。

UART**中断屏蔽寄存器**UARTIM，可以设置开启或者关闭哪些中断。

**RXIM：接收中断**。如果RxFIFO中有数据，且数据的数量超过了UARTIFLS寄存器定义的值，就产生中断。

**TXIM：发送中断**。如果UARTCTL的EOT设置为1，那么等所有的数据都发送完，就产生中断。如果UARTCTL的EOT设置为0，那么发送FIFO中的数据个数，少于UARTIFLS寄存器定义的值，就产生中断。

UARTIFLS FIFO**中断等级寄存器**，规定了触发接收和发送中断的条件：

UART Interrupt FIFO Level Select (UARTIFLS)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x034
Type RW, reset 0x0000.0012

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| reserved | | | | | | | | | | | | | | | |
| Type | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| reserved | | | | | | | | | | RXIFLSEL | | | TXIFLSEL | | |
| Type | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RW | RW | RW | RW | RW | RW |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

# ➢TM4C1294中的UART模块使用方法

## – UART模块的中断使用方法：

UART模块支持多种中断，可以配合中断响应，提高程序的执行效率。

UARTIFLS FIFO**中断等级寄存器**，规定了触发接收和发送中断的条件：

UART Interrupt FIFO Level Select (UARTIFLS)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x034
Type RW, reset 0x0000.0012

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | reserved | | | | | | | | | | | | | | | |
| Type | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | reserved | | | | | | | | | | RXIFLSEL | | | TXIFLSEL | | |
| Type | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RW | RW | RW | RW | RW | RW |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |

RXIFLSEL RW 0x2 UART Receive Interrupt FIFO Level Select

The trigger points for the receive interrupt are as follows:

| Value | Description |
|---|---|
| 0x0 | RX FIFO ≥ ⅛ full |
| 0x1 | RX FIFO ≥ ¼ full |
| 0x2 | RX FIFO ≥ ½ full (default) |
| 0x3 | RX FIFO ≥ ¾ full |
| 0x4 | RX FIFO ≥ ⅞ full |
| 0x5-0x7 | Reserved |

TXIFLSEL RW 0x2 UART Transmit Interrupt FIFO Level Select

The trigger points for the transmit interrupt are as follows:

| Value | Description |
|---|---|
| 0x0 | TX FIFO ≤ ⅞ empty |
| 0x1 | TX FIFO ≤ ¾ empty |
| 0x2 | TX FIFO ≤ ½ empty (default) |
| 0x3 | TX FIFO ≤ ¼ empty |
| 0x4 | TX FIFO ≤ ⅛ empty |
| 0x5-0x7 | Reserved |

# ➢TM4C1294中的UART模块使用方法

## –UART模块的中断使用方法：

默认情况下，UART模块累计接收到8个字节的数据才会才是接收中断，但如果一次连续接收的数据少于8个，会产生接收超时中断，在中断服务函数中及时读出RxFIFO中所有的数据。

TivaWare提供了UARTFIFOLevelSet函数，设置UARTIFLS寄存器

```
void
UARTFIFOLevelSet(uint32_t ui32Base, uint32_t ui32TxLevel,
                 uint32_t ui32RxLevel)
{
    //
    // Check the arguments.
    //
    ASSERT(_UARTBaseValid(ui32Base));
    ASSERT((ui32TxLevel == UART_FIFO_TX1_8) ||
           (ui32TxLevel == UART_FIFO_TX2_8) ||
           (ui32TxLevel == UART_FIFO_TX4_8) ||
           (ui32TxLevel == UART_FIFO_TX6_8) ||
           (ui32TxLevel == UART_FIFO_TX7_8));
    ASSERT((ui32RxLevel == UART_FIFO_RX1_8) ||
           (ui32RxLevel == UART_FIFO_RX2_8) ||
           (ui32RxLevel == UART_FIFO_RX4_8) ||
           (ui32RxLevel == UART_FIFO_RX6_8) ||
           (ui32RxLevel == UART_FIFO_RX7_8));
    HWREG(ui32Base + UART_O_IFLS) = ui32TxLevel | ui32RxLevel;//TxFIFO和RxFIFO的中断等级
}
```

ui32TxLevel和ui32RxLevel分别是TxFIFO和RxFIFO的中断等级，可以提供的设置有

```
#define UART_FIFO_TX1_8     0x00000000  // Transmit interrupt at 1/8 Full
#define UART_FIFO_TX2_8     0x00000001  // Transmit interrupt at 1/4 Full
#define UART_FIFO_TX4_8     0x00000002  // Transmit interrupt at 1/2 Full
#define UART_FIFO_TX6_8     0x00000003  // Transmit interrupt at 3/4 Full
#define UART_FIFO_TX7_8     0x00000004  // Transmit interrupt at 7/8 Full

#define UART_FIFO_RX1_8     0x00000000  // Receive interrupt at 1/8 Full
#define UART_FIFO_RX2_8     0x00000008  // Receive interrupt at 1/4 Full
#define UART_FIFO_RX4_8     0x00000010  // Receive interrupt at 1/2 Full
#define UART_FIFO_RX6_8     0x00000018  // Receive interrupt at 3/4 Full
#define UART_FIFO_RX7_8     0x00000020  // Receive interrupt at 7/8 Full
```

发生中断后，需要查看中断的状态，并且清除中断。

# ➢TM4C1294中的UART模块使用方法

## – UART模块的中断使用方法：

发生中断后，需要查看中断的状态，并且清除中断。

**在UARTMIS寄存器中查看中断的状态：**

UART Masked Interrupt Status (UARTMIS)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x040
Type RO, reset 0x0000.0000

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | reserved | | | | | | | | | | | | | | DMATXMIS | DMARXMIS |
| Type | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | reserved | | | 9BITMIS | EOTMIS | OEMIS | BEMIS | PEMIS | FEMIS | RTMIS | TXMIS | RXMIS | DSRMIS | DCDMIS | CTSMIS | RIMIS |
| Type | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

RTMIS为接收超时中断状态
TXMIS为发送中断状态
RXMIS为接收中断状态。

# ➢TM4C1294中的UART模块使用方法

## – UART模块的中断使用方法：

发生中断后，需要查看中断的状态，并且清除中断。

使用UARTICR寄存器清除中断：

UART Interrupt Clear (UARTICR)

UART0 base: 0x4000.C000
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000
Offset 0x044
Type W1C, reset 0x0000.0000

| | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | reserved | | | | | | | | | | | | | | DMATXIC | DMARXIC |
| Type | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | RO | W1C | W1C |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | reserved | | | 9BITIC | EOTIC | OEIC | BEIC | PEIC | FEIC | RTIC | TXIC | RXIC | DSRMIC | DCDMIC | CTSMIC | RIMIC |
| Type | RO | RO | RO | RW | W1C | W1C | W1C | W1C | W1C | W1C | W1C | W1C | W1C | W1C | W1C | W1C |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

把RTIX、TXIC、RXIC位写1，分别可以清除接收超时中断、发送中断和接收中断。

# ➢TM4C1294中的UART模块使用方法

## –UART模块的中断使用方法：

- UARTIntEnable来使能中断
- UARTIntStatus函数查看中断的状态
- UARTIntClear函数，清除中断。

UARTIntEnable函数的定义如下：

```
void
UARTIntEnable(uint32_t ui32Base, uint32_t ui32IntFlags)
{
    ASSERT(_UARTBaseValid(ui32Base));
    HWREG(ui32Base + UART_O_IM) |= ui32IntFlags;//将ui32IntFlags的值赋给UARTIM寄存器
}
```

各个中断标志(ui32IntFlags)的组合：

```
#define UART_INT_DMATX          0x20000     // DMA TX interrupt
#define UART_INT_DMARX          0x10000     // DMA RX interrupt
#define UART_INT_9BIT           0x1000      // 9-bit address match interrupt
#define UART_INT_OE             0x400       // Overrun Error Interrupt Mask
#define UART_INT_BE             0x200       // Break Error Interrupt Mask
#define UART_INT_PE             0x100       // Parity Error Interrupt Mask
#define UART_INT_FE             0x080       // Framing Error Interrupt Mask
#define UART_INT_RT             0x040       // Receive Timeout Interrupt Mask
#define UART_INT_TX             0x020       // Transmit Interrupt Mask
#define UART_INT_RX             0x010       // Receive Interrupt Mask
#define UART_INT_DSR            0x008       // DSR Modem Interrupt Mask
#define UART_INT_DCD            0x004       // DCD Modem Interrupt Mask
#define UART_INT_CTS            0x002       // CTS Modem Interrupt Mask
#define UART_INT_RI             0x001       // RI Modem Interrupt Mask
```

# ➢TM4C1294中的UART模块使用方法

## – UART模块的中断使用方法：

- UARTIntEnable来使能中断
- UARTIntStatus函数查看中断的状态
- UARTIntClear函数，清除中断。

**UARTIntStatus函数的实现方式如下，**

```
uint32_t
UARTIntStatus(uint32_t ui32Base, bool bMasked)
{
    ASSERT(_UARTBaseValid(ui32Base));

    if(bMasked)//如果输入参数bMasked为true
    {
        return(HWREG(ui32Base + UART_O_MIS));//把UARTMIS的值返回
    }
    else
    {
        return(HWREG(ui32Base + UART_O_RIS));
    }
}
```

# ➢TM4C1294中的UART模块使用方法

## –UART模块的中断使用方法：

- UARTIntEnable来使能中断
- UARTIntStatus函数查看中断的状态
- UARTIntClear函数，清除中断。

UARTIntClear函数用于清除中断，其实现方式如下

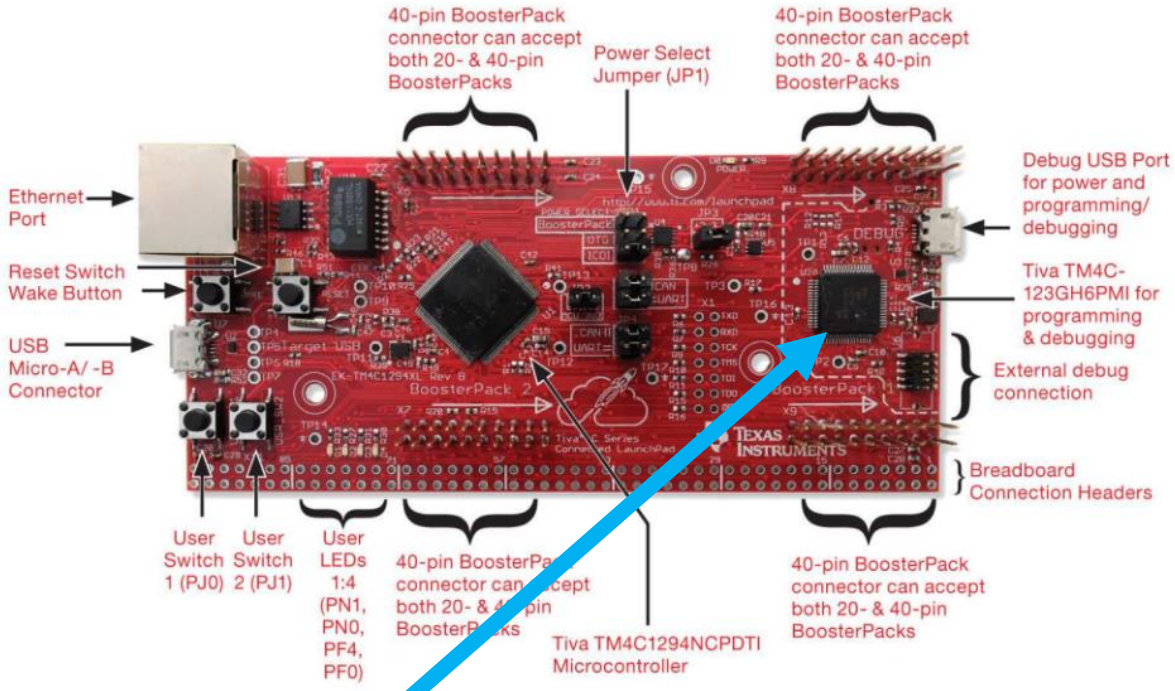
```
void
UARTIntClear(uint32_t ui32Base, uint32_t ui32IntFlags)
{
    ASSERT(_UARTBaseValid(ui32Base));
    HWREG(ui32Base + UART_O_ICR) = ui32IntFlags;//将要清除的中断标志写入到
                                                //UARTICR寄存器中
}
```

## ➢TM4C1294中的UART模块使用方法

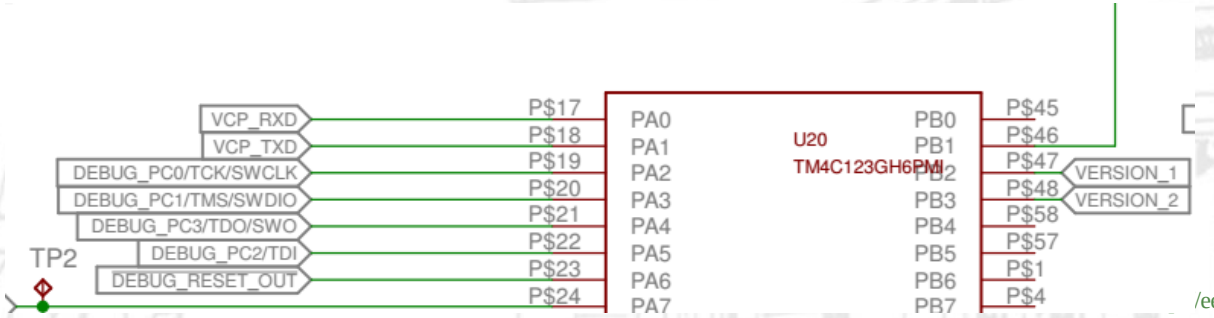
– UART模块的中断使用方法：在NVIC模块中开启UART中断，并注册中断服务函数：

```
IntMasterEnable();
IntEnable(INT_UART0);
UARTIntRegister(UART0_BASE,UARTIntHandler);
UARTIntEnable(UART0_BASE, UART_INT_RX | UART_INT_RT);//  开启接收中断和超时中断

void
UARTIntHandler(void)
{
    uint32_t ui32Status;
    ui32Status = UARTIntStatus(UART0_BASE, true);
    UARTIntClear(UART0_BASE, ui32Status);
     while(UARTCharsAvail(UART0_BASE))
    {
        UARTCharPutNonBlocking(UART0_BASE,

UARTCharGetNonBlocking(UART0_BASE));
         GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0, GPIO_PIN_0);

        SysCtlDelay(g_ui32SysClock / (1000 * 3));
        GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0, 0);
```

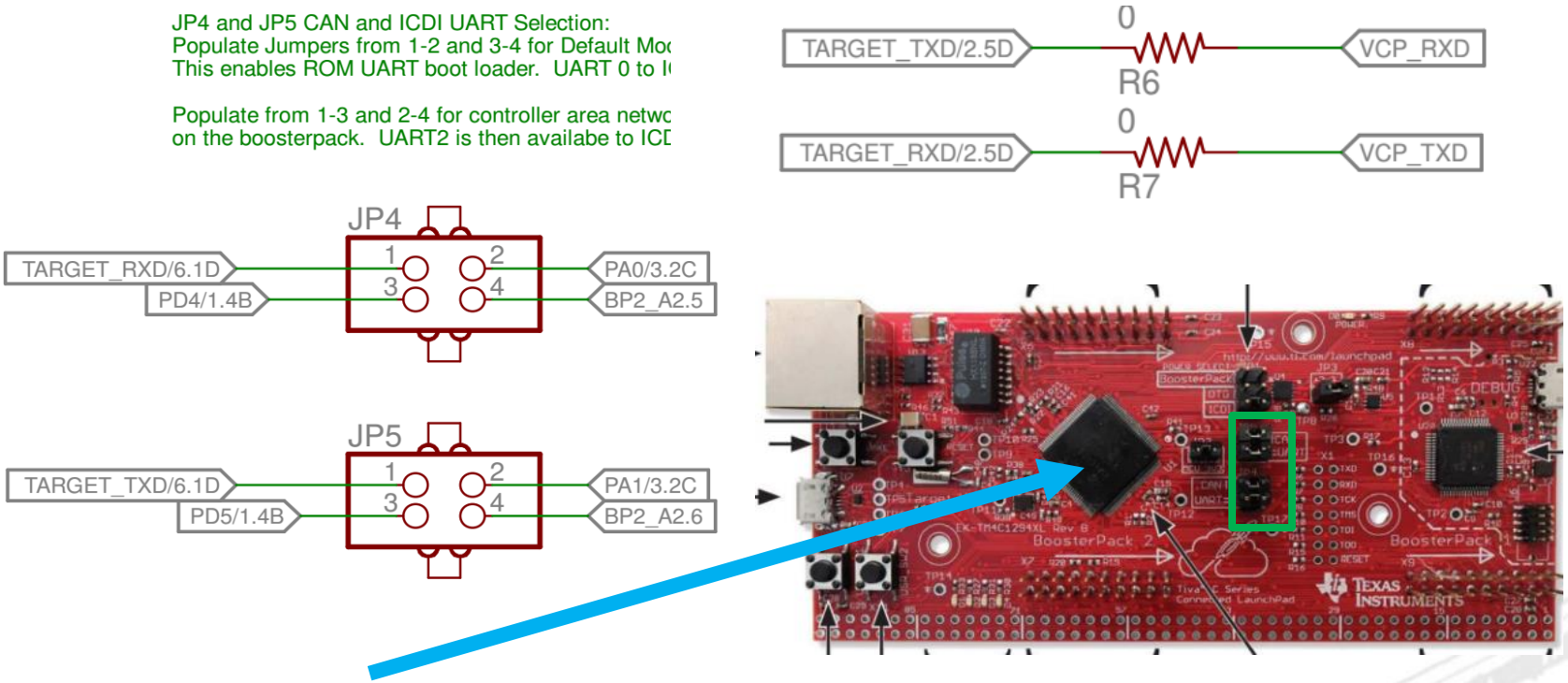
# ➢开发板的UART功能的硬件连接与调试




调试器的TM4C1294微控制器的PA0引脚和PA1引脚分别是接收线VCP_RXD和发送线VCP_TXD。



# ➢开发板的UART功能的硬件连接与调试

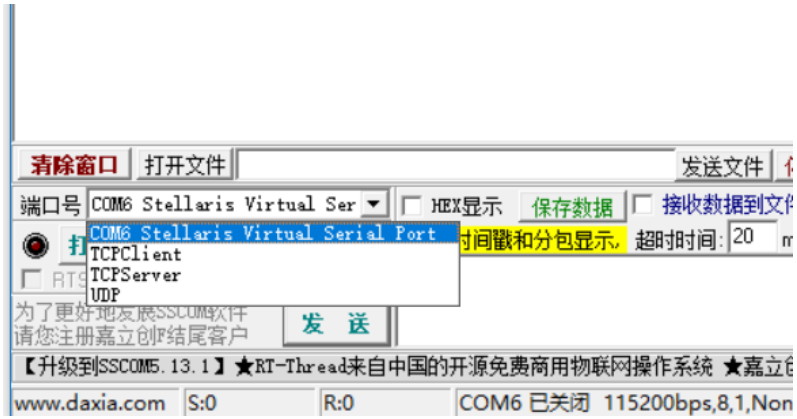



- TM4C1294微控制器的PA0引脚和PA1引脚连接至跳线插座JP4和JP5。
- 如果把跳线横向短接，即1、2短接，3、4短接
- 那么TM4C1294微控制器的PA0引脚（UART0模块的接收线）就连到了信号TARGET_RXD上，
- TM4C1294微控制器的PA1引脚（UART0模块的发送线）就连到了信号TARGET_TXD上。

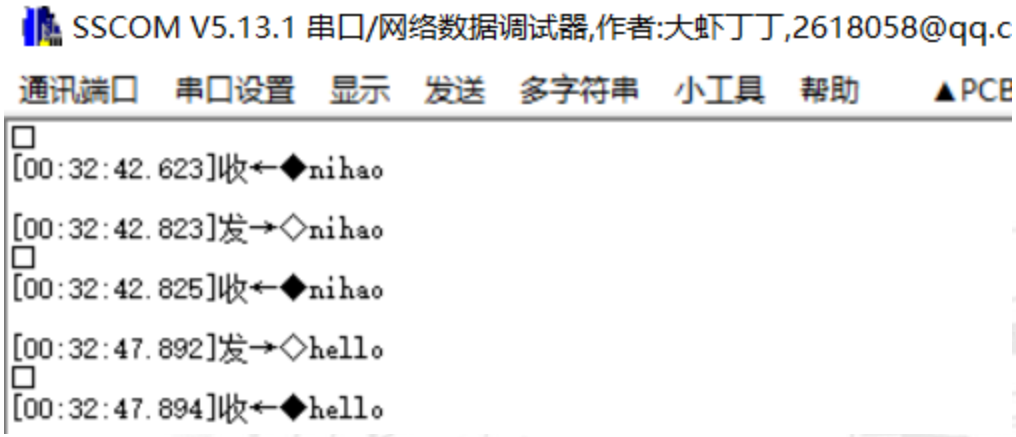
# ➢开发板的UART功能的硬件连接与调试

打开CCS软件，导入uart_echo工程，编译后运行

打开串口调试助手sscom5.1。选择端口号COM？Stellaris Virtual Serial Port，波特率选 115200，然后打开串口。







# ➢RS485总线

RS232标准的不足：

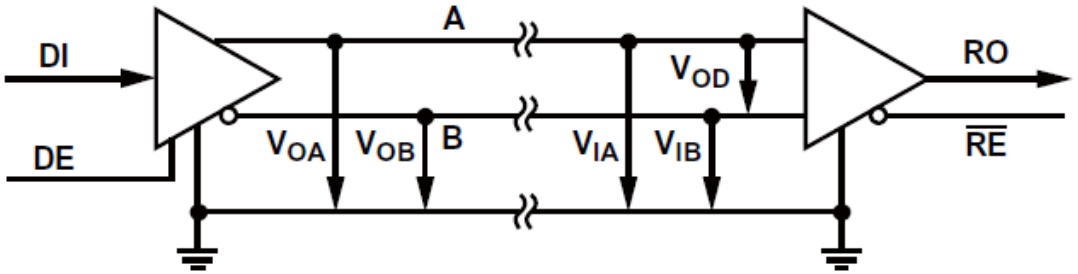
- 接口的信号电平值较高，达到十几V，容易损坏接口电路的芯片，而且和TTL电平不兼容，因此和微控制器接时，必须加转换电路。
- 接口使用的信号线与其他设备形成共地模式的通信，这种共地模式传输容易产生干扰，并且抗干扰性能也比较弱。
- 传输距离、速率都有限，最多只能通信几十米，速度也不高
- 只能两点之间进行通信，不能够实现多机联网通信。

# ➢RS485总线

- RS485标准
  - 逻辑“1”以两线间的电压差为+（2—6）V表示；逻辑“0”以两线间的电压差为-（2—6）V表示。
  - 接口信号电平比RS232降低了，不易损坏电路的芯片，且该电平与TTL电平兼容，可方便与TTL电路连接。
  - RS485通信速度快，数据最高传输速率为10Mbps以上；其内部的物理结构，采用的是平衡驱动器和差分接收器的组合，抗干扰能力大大增加。
  - 传输距离最远可达到1200米左右，但传输速率和传输距离是成反比的，只有在100KB/s以下的传输速率，才能达到最大的通信距离，如果需要传输更远距离可以使用中继。
  - 可以在总线上进行联网实现多机通信，总线上允许挂多个收发器。
  - RS485有两线制和四线制，四线制只能实现点对点的通信方式，现很少采用。两线制这种接线方式为总线式拓朴结构，在同一总线上最多可以挂接32个节点。在RS485通信网络中一般采用的是主从通信方式，即一个主机带多个从机。

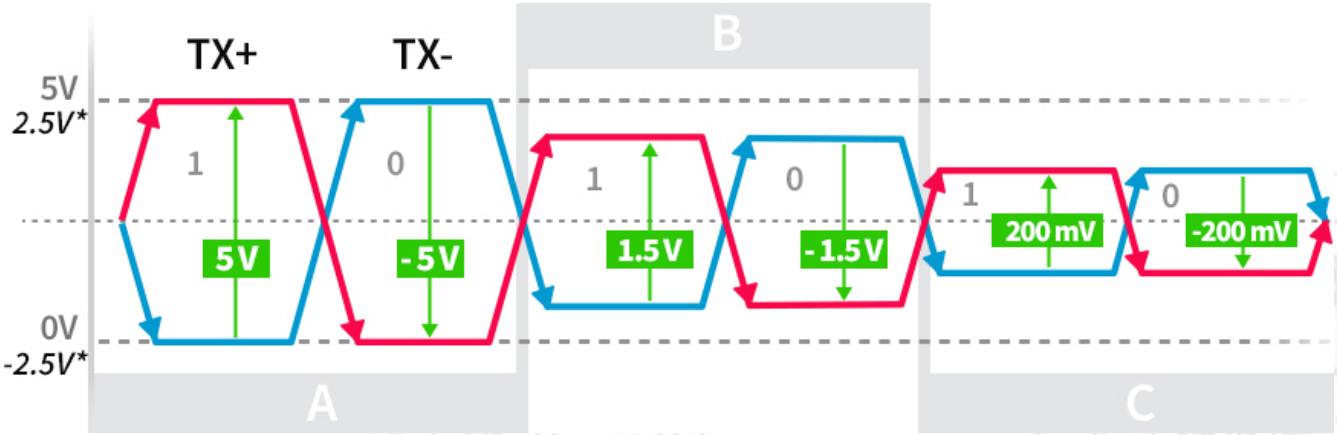
# ➢RS485总线

- 总线的连接




**DI为发送线，RO为接收线，DE为发送使能，RE为接收使能**

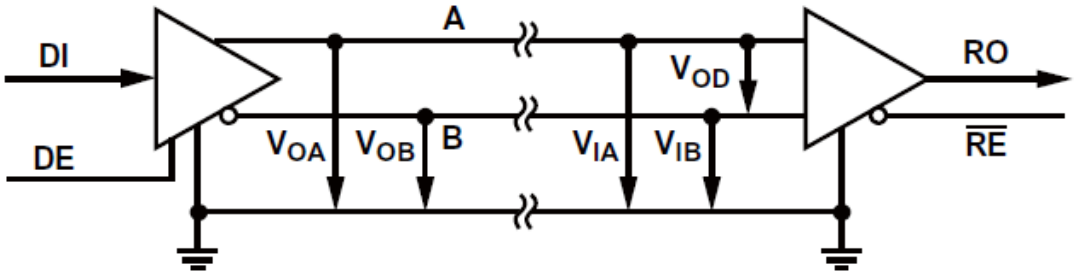
- 两条差分线组成一个差分对，表示为A和B。
- $V_{OA}>V_{OB}$为高电平，$V_{OA}<V_{OB}$为低电平
- 以上比较一般有200mV的滞环




- A = 正常电压 (e.g. 5 V)
- B = 发送的最小电压(1.5 V)
- C = 接收最小电压(200mV)

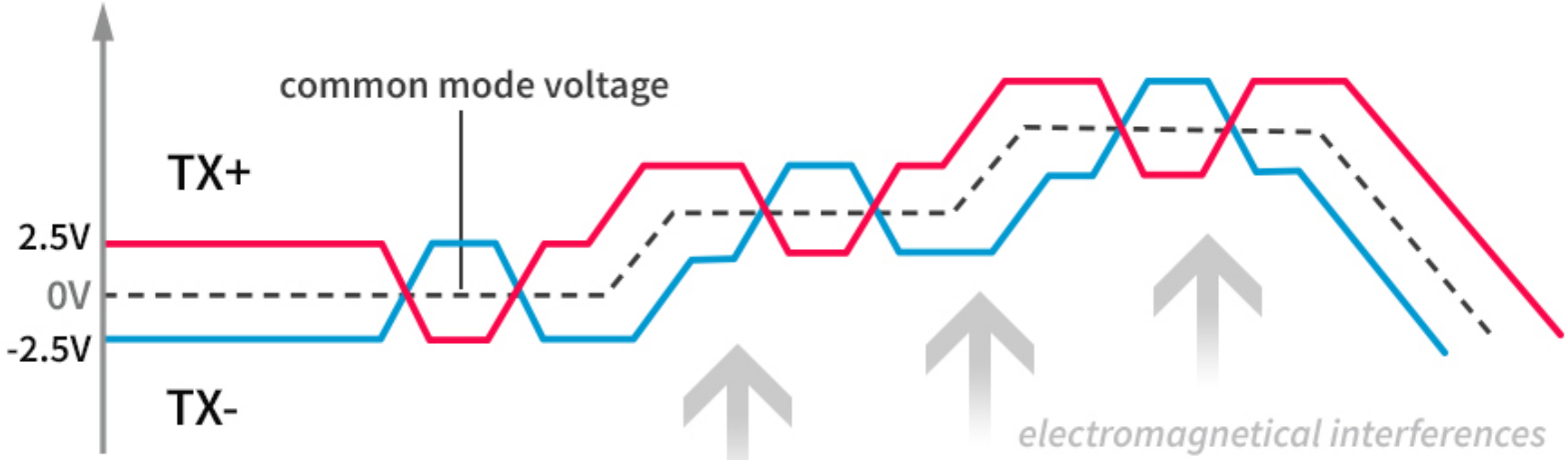
# ➢RS485总线

- 总线的连接




DI为发送线，RO为接收线，DE为发送使能，RE为接收使能

- 两条差分线组成一个差分对，表示为A和B。
- $V_{OA}>V_{OB}$为高电平，$V_{OA}<V_{OB}$为低电平
- 以上比较一般有200mV的滞环



# ➢RS485总线

- RS485总线的连接

RS-485总线为双工标准，同一总线上可以连接最多32个收发器
半双工485总线网络是最常用的485网络，通过总线组成网络后，驱动器必须能够处于高阻态，有一个使能信号DE。发送完数据后，驱动器必须处于高阻态，从而让其他设备也能在总线上发送数据。

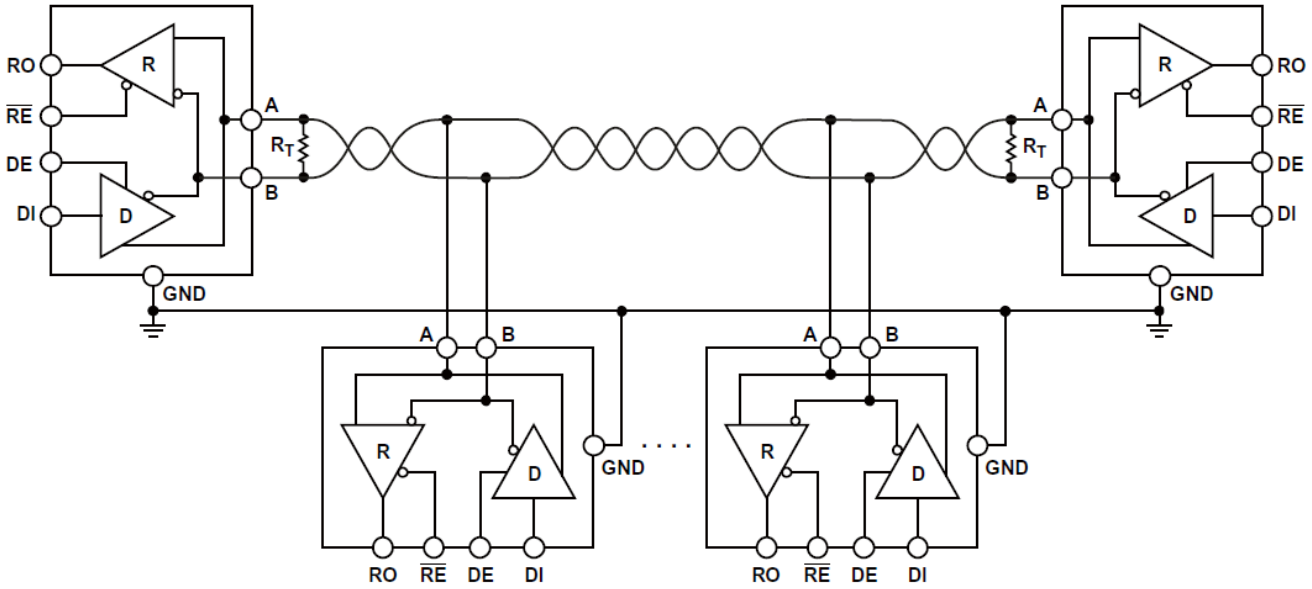



图3. 半双工RS-485总线配置

# ➢RS485总线

- 单位负载

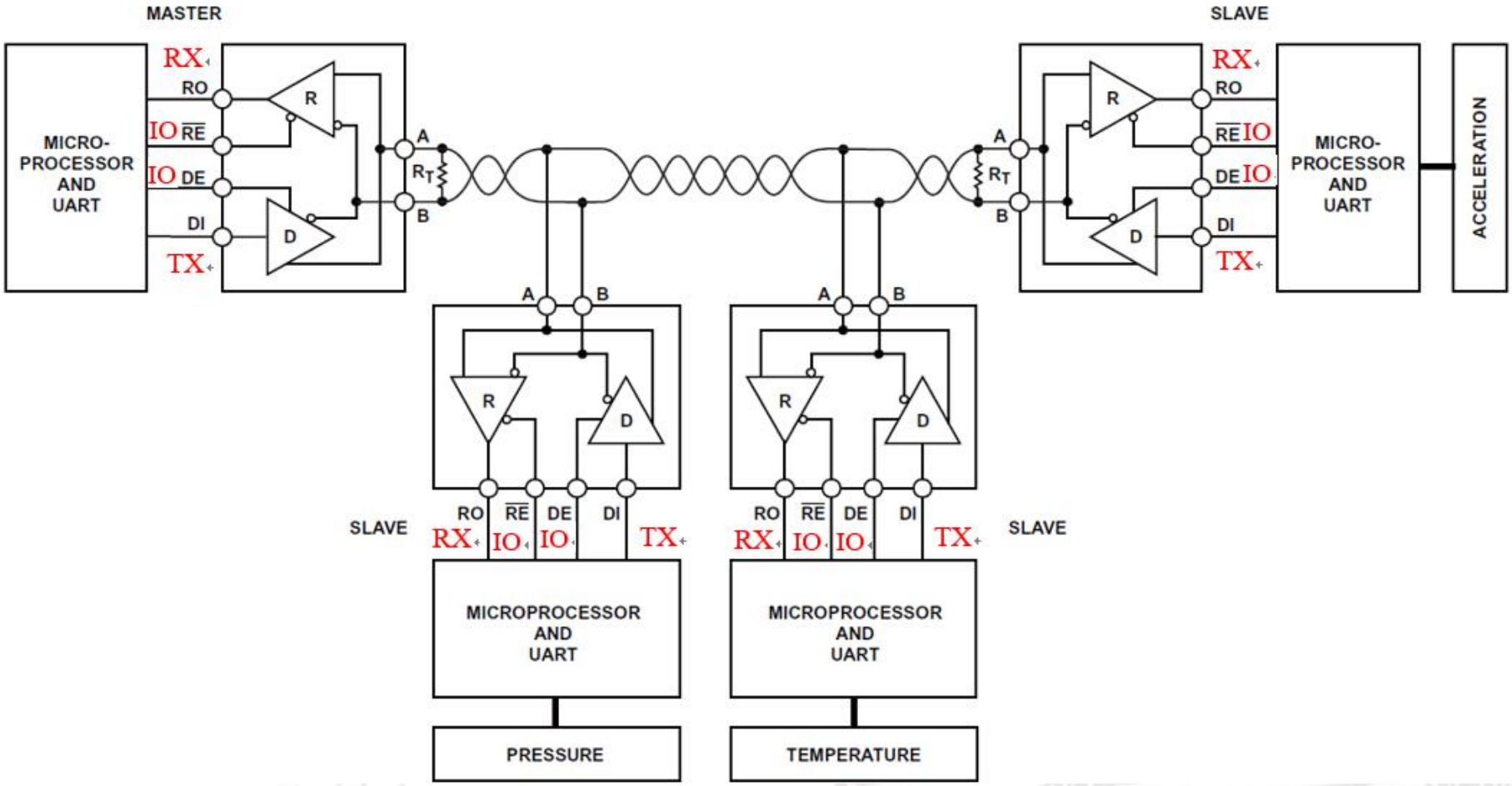
  - RS-485接收器额定的输入阻抗为大于或等于12kΩ。此阻抗被定义为一个单位负载（UL）。
  - RS-485技术规范规定驱动器的最大承受能力为32UL。
  - 随着半导体器件技术的发展，接收器的阻抗可以减小，总线上接收器节点的数量可以翻倍。

表1. UL接收器输入阻抗

| 单位负载 | 节点数 | 最小接收器输入阻抗 |
|---|---|---|
| 1 | 32 | 12 kΩ |
| ½ | 64 | 24 kΩ |
| ¼ | 128 | 48 kΩ |
| ⅛ | 256 | 96 kΩ |

# ➢RS485总线

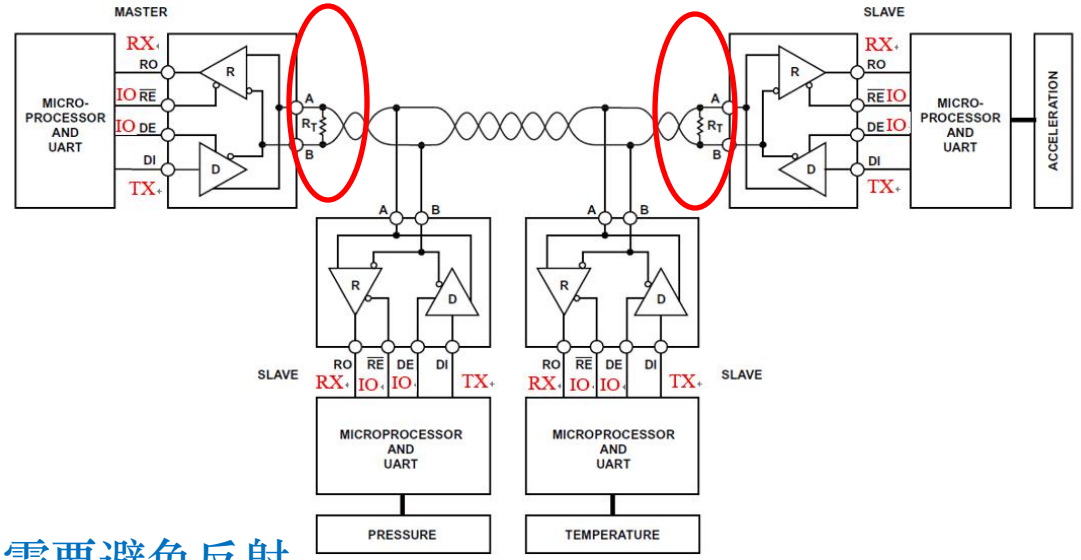
- 与微控制器UART模块的连接方式



# ➢RS485总线

- 总线的可靠传输与抗干扰

1．终端电阻RT




优点：

要实现可靠的通信，需要避免反射

选择适当的终端电阻(端接)

在半双工配置中，电缆的两端必须端接

在全双工配置中，主接收器和最远的从接收器需要端接

缺点：

降低了驱动电压

增大了通信线的压降

增大了损耗

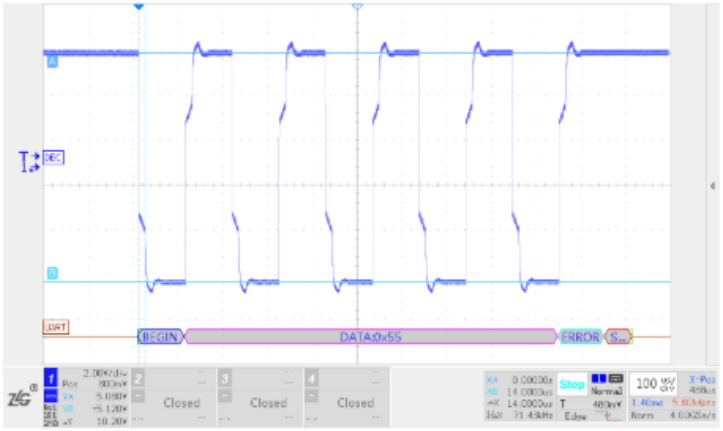



图 8 RSM485ECHT 1200m 9600bps不加终端 首端波形

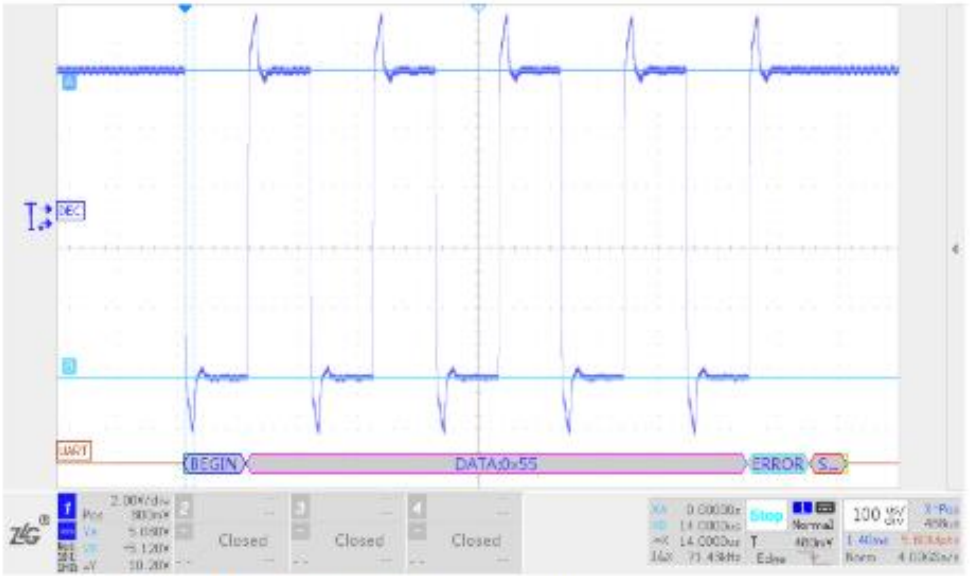



图 9 RSM485ECHT 1200m 9600bps不加终端 末端波形

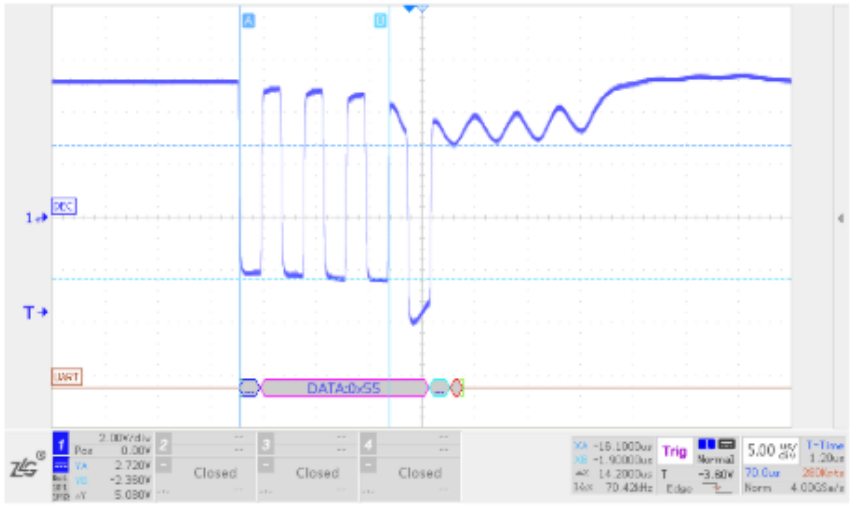



图 1 RSM485ECHT 1200m 500kbps不加终端电阻

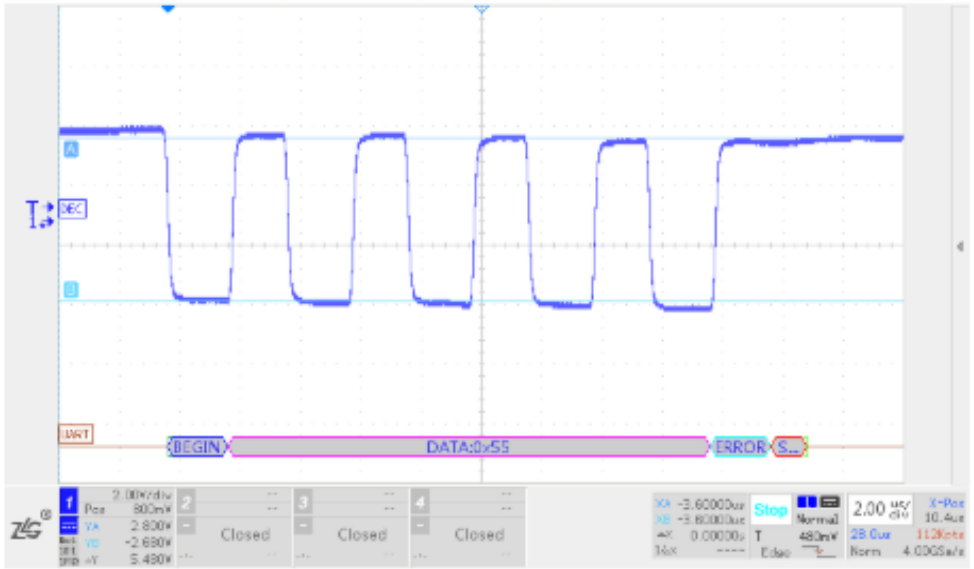



图 2 RSM485ECHT 1200m 500kbps加终端电阻

# ➢RS485总线

- 总线的可靠传输与抗干扰

## 2．故障安全偏置

- 如果总线空闲，没有节点在发送数据，则UART接收到的电平不确定，从而导致无效起始位、虚假中断，帧错误。
- 需要在总线空闲时，让驱动器输出空闲位，即高电平。
- 根据真值表，在总线上接合适的电阻，分压以产生高电平。

表3. 差分接收器真值表

| $\overline{RE}$ | A – B (输入) | RO |
|---|---|---|
| 0 | ≥+200 mV | 1 |
| 0 | ≤–200 mV | 0 |
| 0 | –200 mV ≤ (A – B) ≤ +200 mV | X |
| 1 | X | 高组态 |

$R_1 = R_2 = R$

$V_{IA} - V_{IB} \geq 200\ mV$

$$V_{IA} - V_{IB} = R_T \frac{V_{CC}}{2 + R_T} = \frac{200\ mV}{R}$$

if $V_{CC}$ = 5 V, then R = 1440 Ω

if $V_{CC}$ = 3 V, then R = 960 Ω

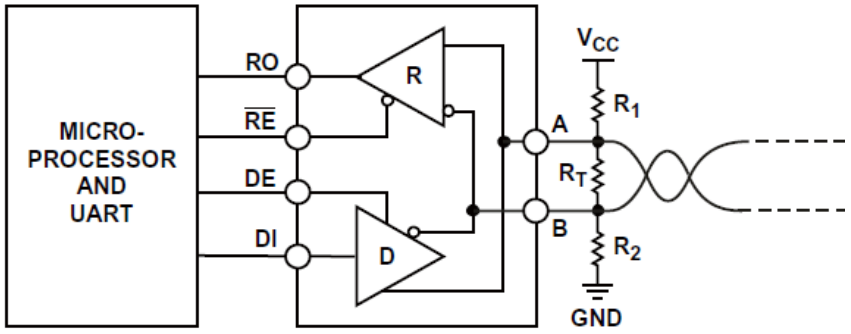



图9. 故障安全偏置电路

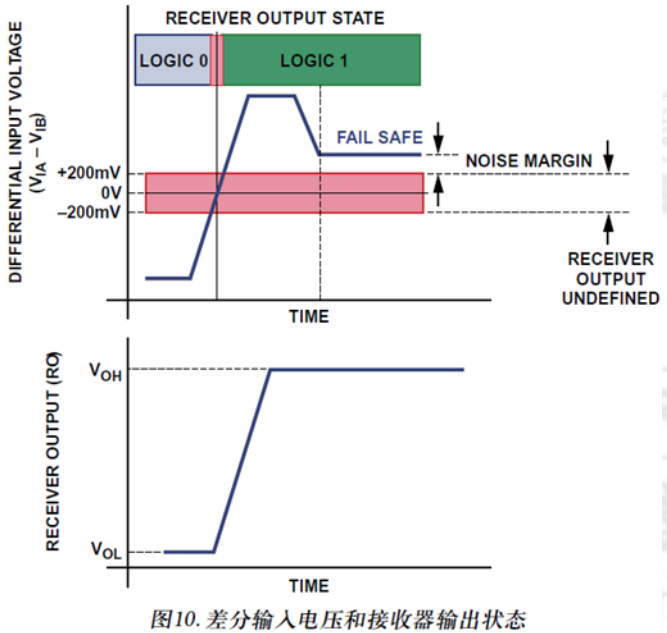



图10. 差分输入电压和接收器输出状态

# ➢RS485总线

- **总线的可靠传输与抗干扰**

## 2．故障安全偏置

- 接分压电阻，增加了元器件的数量和占用电路板的面积
- 可以选用**具有故障安全输入的接收器**，这是一种经过改进的485接收器，内置了故障安全输入，差分阈值电压从±200mV调整到了-200mV和-30mV，即使总线断开或者空闲，也不会造成错误的数据。

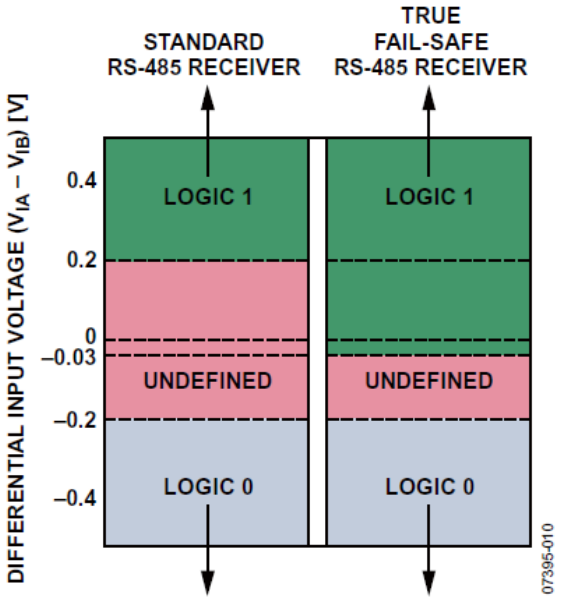



图11. 输入阈值电压

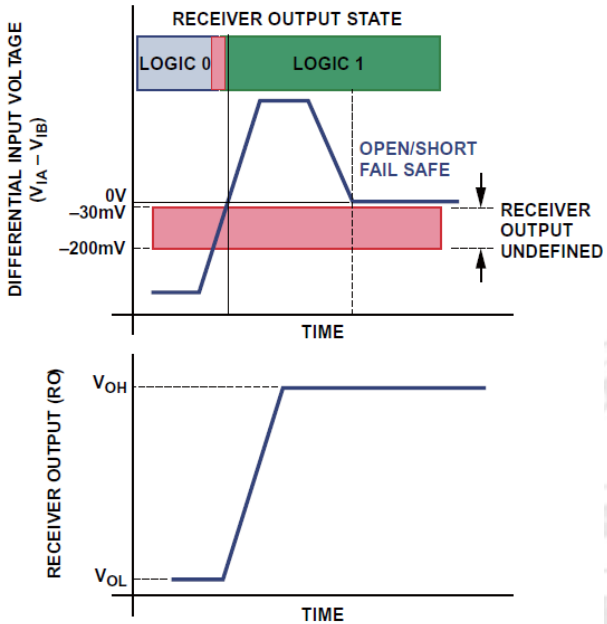



图12. 差分输入电压和接收器输出状态

# ➢RS485总线

- **总线的可靠传输与抗干扰**

### 3. 电气隔离

如果链路较长，无法保证节点之间的地是否一致，则需要电气隔离。
电气隔离阻断电流，但不阻断信息流。

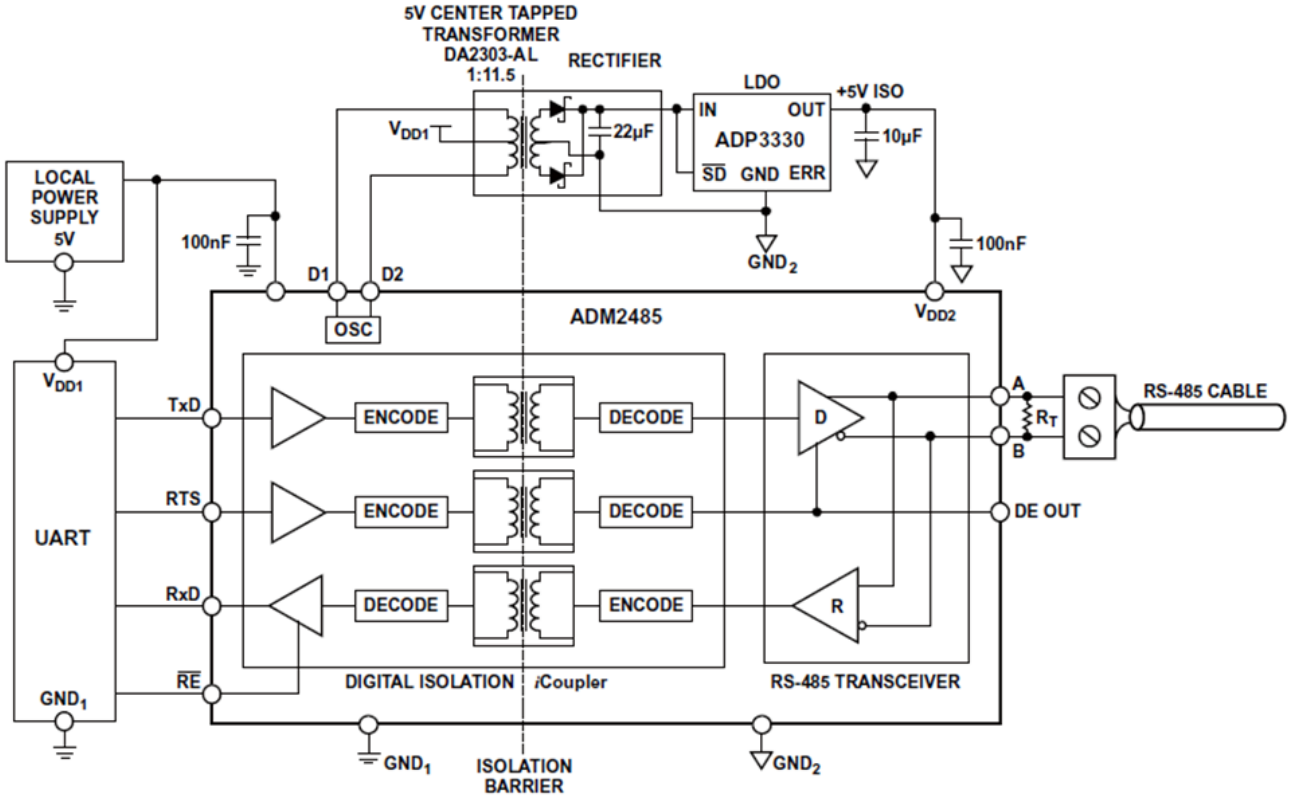



图14. 采用ADM2485的信号隔离和电源隔离

# ➢RS485总线

- 总线的可靠传输与抗干扰

## 4. 瞬变过压保护

- 工业应用中，雷击、电源波形，开关，静电放电都会产生较大的瞬变电压，对RS-485收发器造成损害
- RS-485收发器一般内置保护电路
- 如果使用外部TVS二极管，保护能力可以进一步提升

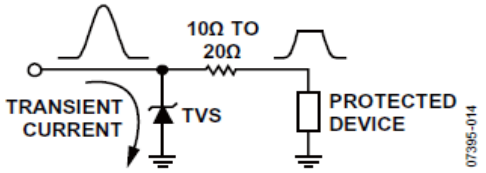



图15. 瞬变电压抑制器

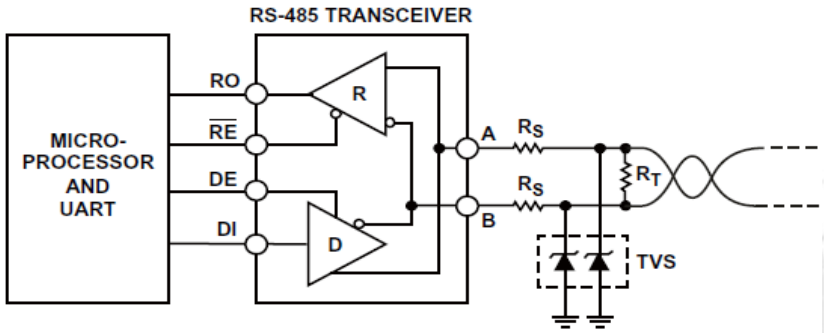


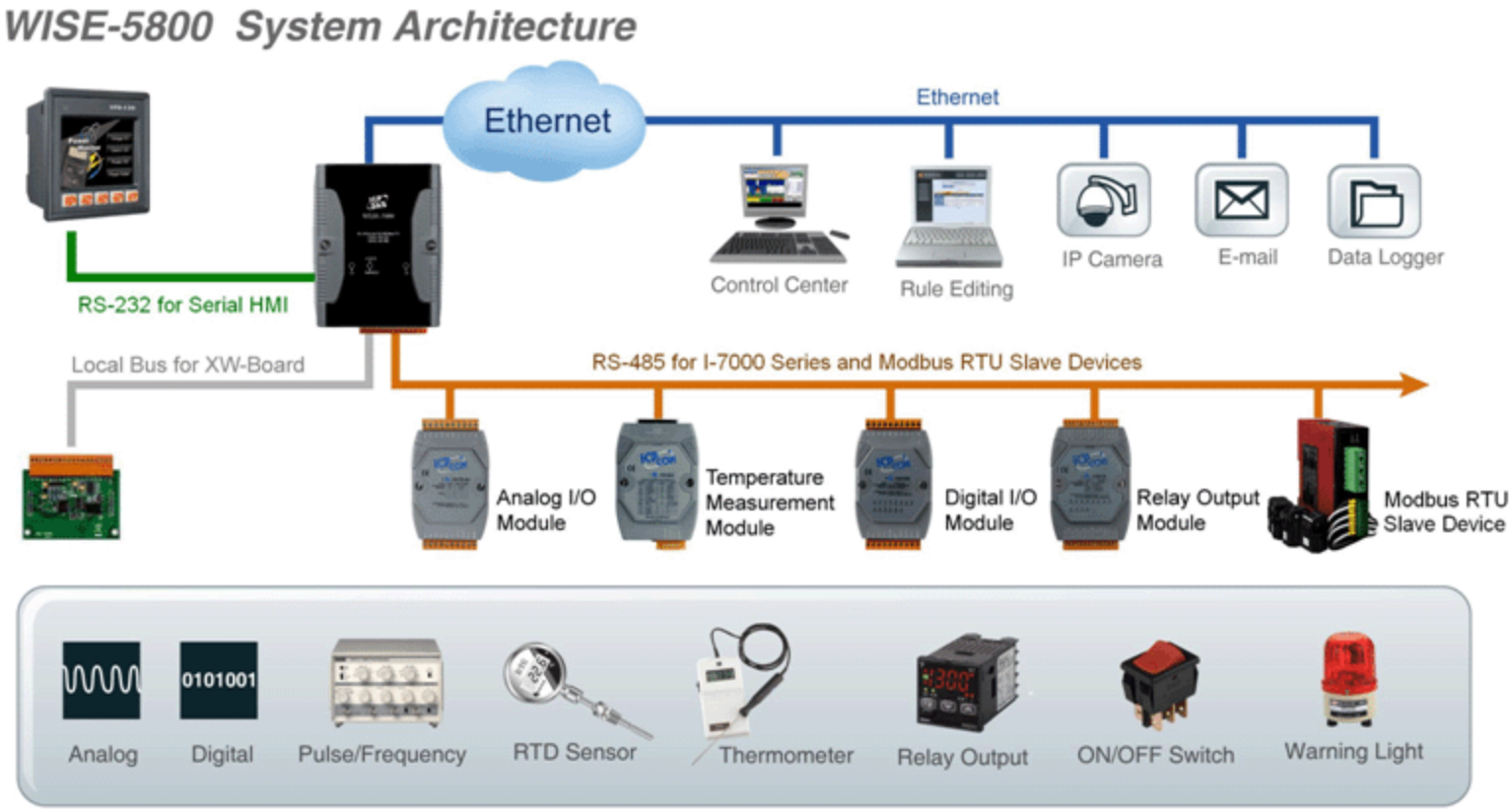

图16. TVS应用电路

# Modbus通信与CRC校验

## Modbus协议简介

- Modbus是Modicon公司为其PLC通讯而开发的一种通讯协议，从1979年问世至今，已经成为工业通讯领域的业界标准；
- 最初的Modbus通讯协议仅支持串口，分为Modbus RTU和ACSII两种信号传输模式（一般基于RS485串口通信）
- 随着时代进步，Modbus也与时俱进，新增了Modbus TCP版本，可以通过以太网进行通讯
- 和其他工业通信协议相比，Modbus主要的优点包括内容公开没有版权要求，不用支付额外费用、硬件要求简单容易部署、使用广泛便于系统集成
- 不同厂商的产品可以简单可靠的接入网络，实现系统的集中监控，分散控制功能。
- Modbus采用半双工的通讯方式，由1个主站和多个从站组成，允许多个设备连接在同一个网络上进行通讯。

# ➢Modbus协议在工业现场的应用



## ➢数据格式

- 目前Modbus规约主要使用ASCII, RTU, TCP等，并没有规定物理层，RS-232C，RS485，以太网都可以作为物理层
- Modbus数据通信采用主从方式
- Master端也可以直接发消息修改Slave端的数据，也可以接收Slave端的数据，实现双向读写。


通用 MODBUS 帧，

ADU： 应用数据单元
PDU： 协议数据单元

- **Modbus RTU (remote terminal unit )**

0-247,0为广播地址

| 起始位 | 设备地址 | 功能代码 | 数据 | CRC校验 | 结束符 |
|---|---|---|---|---|---|
| T1-T2-T3-T4 | 8Bit | 8Bit | n个8Bit | 16Bit | T1-T2-T3-T4 |

1-255

- **Modbus ASCII**

只使用字符0-9，A-F

| 起始位 | 设备地址 | 功能代码 | 数据 | LRC校验 | 结束符 |
|---|---|---|---|---|---|
| 1个字符 | 2个字符 | 2个字符 | n个字符 | 2个字符 | 2个字符 |

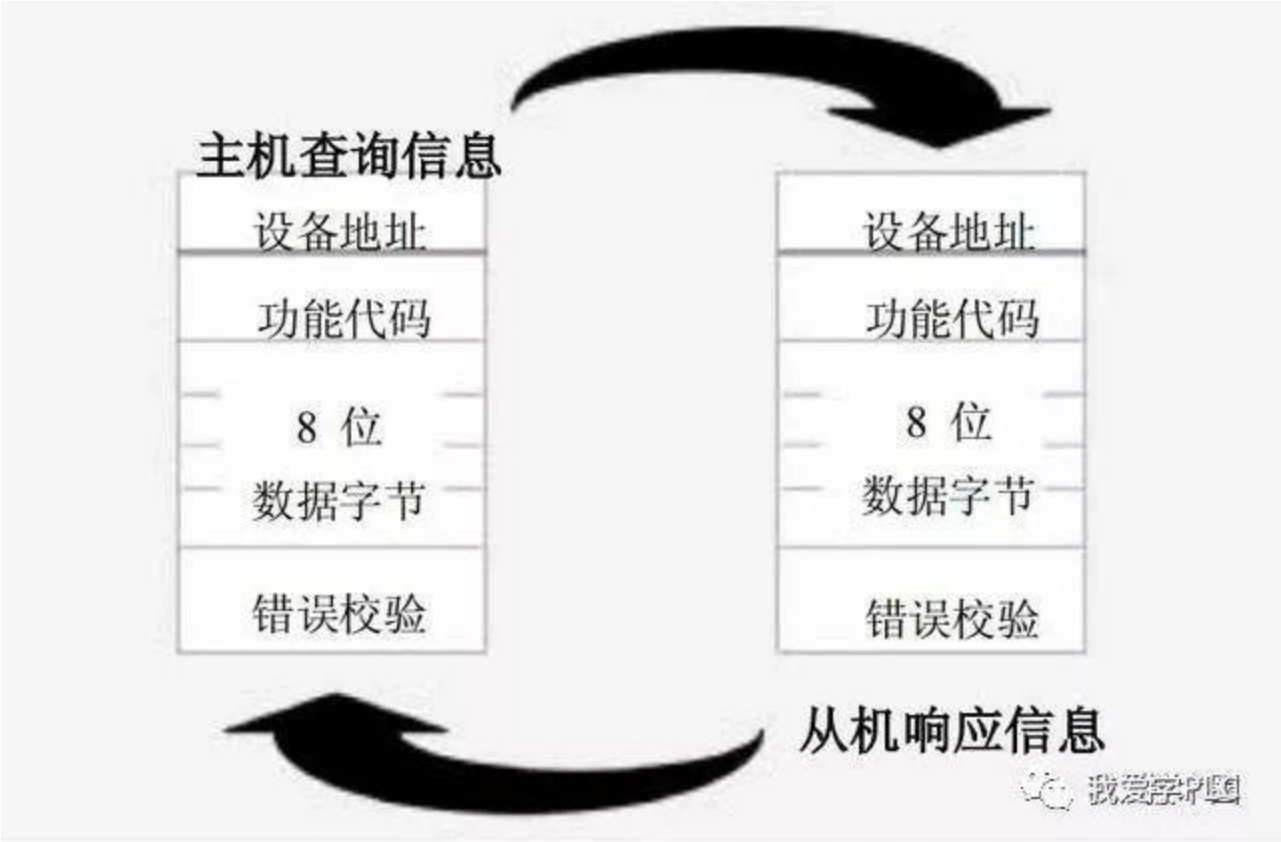
：(0x3AH)

回车0DH
换行0AH

## ➢查询—回应周期

- 查询
  - 查询消息中的功能代码告之被选中的从设备要执行何种功能。
  - **数据段**包含了从设备要执行功能的任何附加信息。
  - **错误检测域**为从设备提供了一种验证消息内容是否正确的方法。
- 回应
  - 如果从设备产生一正常的回应，在回应消息中的**功能代码**是在查询消息中的功能代码的回应。
  - **数据段**包括了从设备收集的数据
  - 如果有错误发生，功能代码将被修改以用于指出回应消息是错误的，同时数据段包含了描述此错误信息的代码。
  - **错误检测域**允许主设备确认消息内容是否可用。

# ➢查询—回应周期



# ➢VFC-CH2000变频器通讯示例

RTU mode:

Command Message:

| | |
|---|---|
| Address | 01H |
| Function | 06H |
| Data address | 01H |
| | 00H |
| Data content | 17H |
| | 70H |
| CRC CHK Low | 86H |
| CRC CHK High | 22H |

Response Message

| | |
|---|---|
| Address | 01H |
| Function | 06H |
| Data address | 01H |
| | 00H |
| Data content | 17H |
| | 70H |
| CRC CHK Low | 86H |
| CRC CHK High | 22H |

RTU mode:

Command Message:

| | |
|---|---|
| Address | 01H |
| Function | 03H |
| Starting data address | 21H |
| | 02H |
| Number of data (count by world) | 00H |
| | 02H |
| CRC CHK Low | 6FH |
| CRC CHK High | F7H |

Response Message

| | |
|---|---|
| Address | 01H |
| Function | 03H |
| Number of data (count by byte) | 04H |
| Content of data address 2102H | 17H |
| | 70H |
| Content of data address 2103H | 00H |
| | 00H |
| CRC CHK Low | FEH |
| CRC CHK High | 5CH |

# ➢应用示例


| 功能 | 指令 |
| --- | --- |
| | 06H 功能码 |
| 显示 10 进 制数 | PLC 发送：01 06 00 00 22 B8 91 18<br>● 01：数码管屏的站号（RS485 地址）<br>● 06：功能码<br>● 00 00：数码管屏的显示寄存器<br>● 22 B8：显示数据。2 字节有符号整数，高位字节在前。22 B8 表示显示 8888。 负数用补码显示，FF FF 表示 -1；FF FE 表示 -2<br>● 91 18：二个字节 CRC 码<br>数码管屏返回：01 06 00 00 22 B8 91 18<br>说明：此命令需要结合下面的设定小数点位数指令一起使用。上电缺省状态是读取保存在 flash 中的小数点位，适用于运行中小数位固定不变或者无需显示小数点情况。<br>小提示：因为 2 字节有符号数的数值范围是-32768~32767，需要显示超过此数值范围时请使用多写寄存器命令。 |


| | |
|---|---|
| 显示 10 进制数（带正负号和小数点，数值范围 -99999~999999） | PLC 发送：01 10 00 90 00 03 06 00 05 42 3F 00 0F 48 8D<br>● 01：数码管屏的站号（RS485 地址）<br>● 10：功能码，表示写多个寄存器<br>● 00 90：数码管屏的显示寄存器（带小数点和正负号的整数）<br>● 00 03：寄存器个数<br>● 06：数据个数（字节数）<br>● 00 05：05 表示小数点位数。5 表示小数点后有 5 位数字，0 表示无小数点<br>● 42 3F：长整数的低 16 位<br>● 00 0F：长整数的高 16 位。42 3F 00 0F 表示十进制 999999<br>● 6C 4E：二个字节 CRC 码<br>此命令将显示：“9.99999”<br>数码管屏返回：01 10 00 90 00 03 80 25 |

## ➢功能域

- 消息从主设备发往从设备时，功能代码域将告之从设备需要执行哪些行为。
  - 例如去读取输入的开关状态，读一组寄存器的数据内容。
- 当从设备回应时，它使用功能代码域来指示是正常回应(无误)还是有某种错误发生（称作异议回应）。
  - 对正常回应，从设备仅回应相应的功能代码。
  - 对异议回应，从设备讲功能代码的最高位置1.

| 功能码 | 名称 | 作用 |
| --- | --- | --- |
| 01 | 读取线圈状态 | 取得一组逻辑线圈的当前状态（ON/OFF) |
| 02 | 读取输入状态 | 取得一组开关输入的当前状态（ON/OFF) |
| 03 | 读取保持寄存器 | 在一个或多个保持寄存器中取得当前的二进制值 |
| 04 | 读取输入寄存器 | 在一个或多个输入寄存器中取得当前的二进制值 |
| 05 | 强置单线圈 | 强置一个逻辑线圈的通断状态 |
| 06 | 预置单寄存器 | 把具体二进值装入一个保持寄存器 |
| 07 | 读取异常状态 | 取得8个内部线圈的通断状态，这8个线圈的地址由控制器决定 |
| 08 | 回送诊断校验 | 把诊断校验报文送从机，以对通信处理进行评鉴 |
| 09 | 编程（只用于484） | 使主机模拟编程器作用，修改PC从机逻辑 |
| 10 | 控询（只用于484） | 可使主机与一台正在执行长程序任务从机通信，探询该从机是否已完成其操作任务，仅在含有功能码9的报文发送后，本功能码才发送 |
| 11 | 读取事件计数 | 可使主机发出单询问，并随即判定操作是否成功，尤其是该命令或其他应答产生通信错误时 |
| 12 | 读取通信事件记录 | 可是主机检索每台从机的ModBus事务处理通信事件记录。如果某项事务处理完成，记录会给出有关错误 |
| 13 | 编程（184/384 484 584） | 可使主机模拟编程器功能修改PC从机逻辑 |
| 14 | 探询（184/384 484 584） | 可使主机与正在执行任务的从机通信，定期控询该从机是否已完成其程序操作，仅在含有功能13的报文发送后，本功能码才得发送 |

| | | |
|---|---|---|
| 15 | 强置多线圈 | 强置一串连续逻辑线圈的通断 |
| 16 | 预置多寄存器 | 把具体的二进制值装入一串连续的保持寄存器 |
| 17 | 报告从机标识 | 可使主机判断编址从机的类型及该从机运行指示灯的状态 |
| 18 | （884和MICRO 84） | 可使主机模拟编程功能，修改PC状态逻辑 |
| 19 | 重置通信链路 | 发生非可修改错误后，是从机复位于已知状态，可重置顺序字节 |
| 20 | 读取通用参数（584L） | 显示扩展存储器文件中的数据信息 |
| 21 | 写入通用参数（584L） | 把通用参数写入扩展存储文件，或修改之 |
| 22～64 | 保留作扩展功能备用 | |
| 65～72 | 保留以备用户功能所用 | 留作用户功能的扩展编码 |
| 73～119 | 非法功能 | |
| 120～127 | 保留 | 留作内部作用 |
| 128～255 | 保留 | 用于异常应答 |

## ➢数据域

- 执行特定功能所需要的数据
- 或者终端响应查询时采集到的数据
- 数据的内容可能是数值、参考地址或者设置值
  - 例如：功能域码告诉终端读取寄存器，数据域则需要指明从哪个寄存器开始及读取多少个数据

## ➢CRC校验

- CRC即循环冗余校验码，是数据通信领域中最常用的一种查错校验码
- 其特征是信息字段和校验字段的长度可以任意选定
- 循环冗余检查（CRC）是一种数据传输检错功能，对数据进行多项式计算，并将得到的结果附在帧的后面，接收设备也执行类似的算法，以保证数据传输的正确性和完整性

RTU mode:

| Command Message: | | Response Message | |
|---|---|---|---|
| Address | 01H | Address | 01H |
| Function | 06H | Function | 06H |
| Data address | 01H | Data address | 01H |
| | 00H | | 00H |
| Data content | 17H | Data content | 17H |
| | 70H | | 70H |
| CRC CHK Low | 86H | CRC CHK Low | 86H |
| CRC CHK High | 22H | CRC CHK High | 22H |

## ➢CRC校验原理

- 其根本思想就是先在要发送的帧后面附加一个数（这个就是用来校验的校验码，但要注意，这里的数也是二进制序列的，下同），生成一个新帧发送给接收端。
- 要使所生成的新帧能与发送端和接收端共同选定的某个特定数整除
- 到达接收端后，再把接收到的新帧除以这个选定的除数。
- 结果应该是没有余数。如果有余数，则表明该帧在传输过程中出现了差错。

## ➢生成多项式：

- 发送端和接收端共同选定的某个特定数，叫做生成多项式。
- 生成多项式的选取应满足以下条件：
  - 生成多项式的最高位和最低位必须为1。
  - 当被传送信息（CRC码）任何一位发生错误时，被生成多项式做模2除后，应该使余数不为0。
  - 不同位发生错误时，应该使余数不同。
  - 对余数继续做模2除，应使余数循环。

# ➢生成多项式

| 名称 | 生成多项式 | 数值式 | 简记式 |
| --- | --- | --- | --- |
| CRC-16 | $x^{16}+x^{15}+x^{2}+1$ | 0x1'8005 | 8005 |
| CRC-CCITT | $x^{16}+x^{12}+x^{5}+1$ | 0X1'1021 | 0x1021 |
| CRC-32 | 注* | 0X1'04C11DB7 | 0x04C11DB7 |

## ➢模2除法

– 模2除法与算术除法类似，但每一位除的结果不影响其它位，即不向上一位借位，所以实际上就是异或。在循环冗余校验码（CRC）的计算中有应用到模2除法。

```
          1011
1101 ) 1111000
       1101
       001000
       001101
         01010
          1101
          0111
```

# ➢CRC校验码计算示例

– 现假设选择的CRC生成多项式为G(X) = $X^4 + X^3 + 1$，要求出二进制序列10110011的CRC校验码。

- 将多项式转化为二进制序列:11001
- 多项式的位数为5,则在数据帧的后面加上5-1个0.
- 数据帧变为101100110000，然后使用模2除法除以除数11001，得到余数
- 将计算出来的CRC校验码添加在原始帧的后面，真正的数据帧为101100110100，再把这个数据帧发送到接收端。
- 接收端收到数据帧后，用上面选定的除数，用模2除法除去，验证余数是否为0，如果为0，则说明数据帧没有出错。

```
          11010100
        ----------
11001 ) 10110011 0000   <- 在原数据帧后面添加的比特位
        11001
        -----
        11110
        11001
        -----
        11111
        11001
        -----
        11000
        11001
        -----
        0100
```

余数，因为不够4位，所以前面一位的0要加上

## ➢实际应用中的CRC

- 实际上，真正的CRC 计算通常与上面描述的还有些出入。
- 这是因为这种最基本的CRC除法有个很明显的缺陷，就是数据流的开头添加一些0并不影响最后校验字的结果。
- 因此真正应用的CRC 算法基本都在原始的CRC算法的基础上做了些小的改动。
  - 所谓的改动，也就是增加了两个概念，第一个是“余数初始值”，第二个是“结果异或值”。
  - 所谓的“余数初始值”就是在计算CRC值的开始，给CRC寄存器一个初始值。“结果异或值”是在其余计算完成后将CRC寄存器的值在与这个值进行一下异或操作作为最后的校验值。

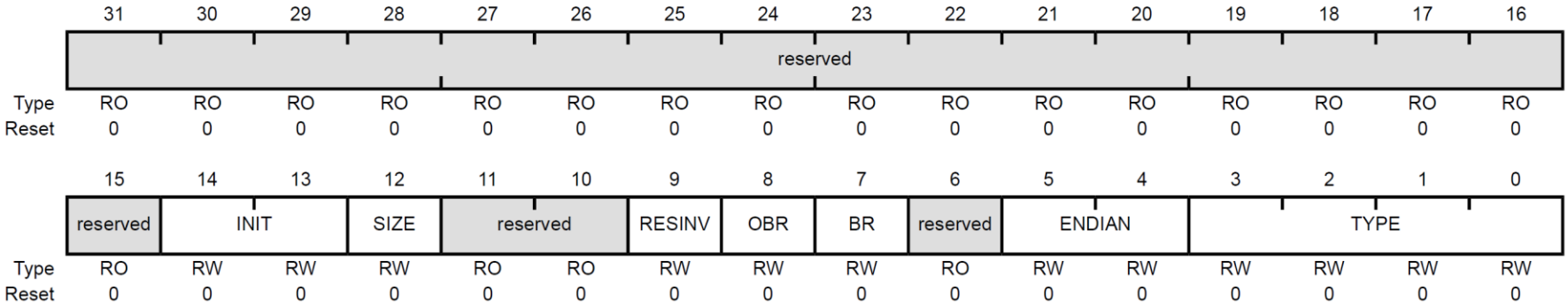
| | CRC-16/CCITT-FALSE | CRC16/MODBUS | CRC32 |
|---|---|---|---|
| 校验和位宽W | 16 | 16 | 32 |
| 生成多项式 | x16+x12+x5+1 | x16+x15+x2+1 | x32+x26+x23+x22+x16+x12+x11+x10+x8+x7+x5+x4+x2+x1+1 |
| 除数（多项式） | 0x1021 | 0x8005 | 0x04C11DB7 |
| 余数初始值 | 0xFFFF | 0xFFFF | 0xFFFFFFFF |
| 结果异或值 | 0x0000 | 0x0000 | 0xFFFFFFFF |

- 实现方式：
  - 软件计算CRC
  - 软件查表计算CRC
    - https://www.cnblogs.com/zzdbullet/p/9580502.html
  - 硬件计算CRC

## ➢TM4C的硬件CRC模块：

- 支持四种校验码
  - CRC16-CCITT as used by CCITT/ITU X.25（1021）
  - CRC16-IBM as used by USB and ANSI（8005）
  - CRC32-IEEE as used by IEEE802.3 and MPEG2
  - CRC32C as used by G.Hn
- 可以逐字输入，可一个按字节输入
- 高低有效位的顺序可以设置

# ➢CRC Control (CRCCTRL)寄存器：




余数初始值
2：全0
3：全1

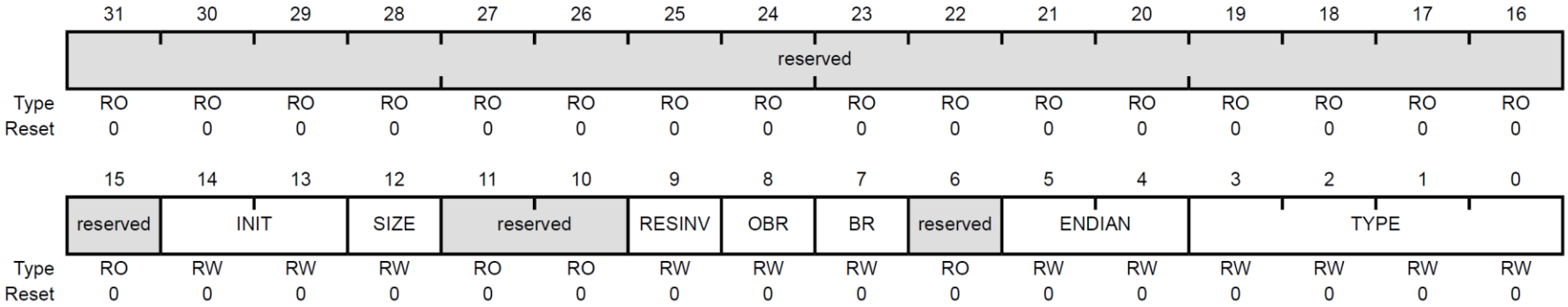
输入数据
长度
0：32 bit
1：8 bit

结果位 输出位 输入位
反转 反转 反转

| Value | Description |
|---|---|
| 0x0 | Polynomial 0x8005 |
| 0x1 | Polynomial 0x1021 |
| 0x2 | Polynomial 0x4C11DB7 |
| 0x3 | Polynomial 0x1EDC6F41 |
| 0x4-0x7 | reserved |
| 0x8 | TCP checksum |
| 0x9-0xF | reserved |

| | |
|---|---|
| 0x0 | Configuration unchanged. (B3, B2, B1, B0) |
| 0x1 | Bytes are swapped in half-words but half-words are not swapped (B2, B3, B0, B1) |
| 0x2 | Half-words are swapped but bytes are not swapped in half-word. (B1, B0, B3, B2) |
| 0x3 | Bytes are swapped in half-words and half-words are swapped. (B0, B1, B2, B3) |

# ➢CRC Control (CRCCTRL)寄存器：




余数初始值
2：全0
3：全1

输入数据
长度
0：32 bit
1：8 bit

结果位 输出位 输入位
反转 反转 反转

| Value | Description |
|---|---|
| 0x0 | Polynomial 0x8005 |
| 0x1 | Polynomial 0x1021 |
| 0x2 | Polynomial 0x4C11DB7 |
| 0x3 | Polynomial 0x1EDC6F41 |
| 0x4-0x7 | reserved |
| 0x8 | TCP checksum |
| 0x9-0xF | reserved |

CRCCTRL的BR位设置**输入位反转**
由于UART发送数据时，是先发低位，再发高位，而CRC模块处理数据是先从高位往低位计算，所以要开启输入位反转

CRCCTRL的OBR位设置**输出位反转**
于UART发送数据时，是先发低位，再发高位，而CRC模块计算得到的结果，也是高位在前，低位在后，因此输出的结果，也要位反转。

## ➢TM4C的硬件CRC模块的使用方法：

1．使能CRC模块（CCM0）的时钟：

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_CCM0);
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_CCM0))
{
}
```

# ➢TM4C的硬件CRC模块的使用方法：

## 2. 配置CRCCTRL，设置CRC模块

TivaWare提供CRCConfigSet函数设置CRC模块，其定义为

```
void
CRCConfigSet(uint32_t ui32Base, uint32_t ui32CRCConfig)
{
  //
  // Check the arguments.
  //
  ASSERT(ui32Base == CCM0_BASE);
  ASSERT((ui32CRCConfig & CRC_CFG_INIT_SEED) ||
         (ui32CRCConfig & CRC_CFG_INIT_0) ||
         (ui32CRCConfig & CRC_CFG_INIT_1) ||
         (ui32CRCConfig & CRC_CFG_SIZE_8BIT) ||
         (ui32CRCConfig & CRC_CFG_SIZE_32BIT) ||
         (ui32CRCConfig & CRC_CFG_RESINV) ||
         (ui32CRCConfig & CRC_CFG_OBR) ||
         (ui32CRCConfig & CRC_CFG_IBR) ||
         (ui32CRCConfig & CRC_CFG_ENDIAN_SBHW) ||
         (ui32CRCConfig & CRC_CFG_ENDIAN_SHW) ||
         (ui32CRCConfig & CRC_CFG_TYPE_P8005) ||
         (ui32CRCConfig & CRC_CFG_TYPE_P1021) ||
         (ui32CRCConfig & CRC_CFG_TYPE_P4C11DB7) ||
         (ui32CRCConfig & CRC_CFG_TYPE_P1EDC6F41) ||
         (ui32CRCConfig & CRC_CFG_TYPE_TCPCHKSUM));
  HWREG(ui32Base + CCM_O_CRCCTRL) = ui32CRCConfig;
}
```

# ➢TM4C的硬件CRC模块的使用方法：

2. 配置CRCCTRL，设置CRC模块

TivaWare提供CRCConfigSet函数设置CRC模块

```
CRCConfigSet(CCM0_BASE,CRC_CFG_INIT_1 | CRC_CFG_SIZE_8BIT|
CRC_CFG_TYPE_P8005|CRC_CFG_IBR|CRC_CFG_OBR);
```

```
#define CRC_CFG_INIT_SEED       0x00000000  // Initialize with seed
#define CRC_CFG_INIT_0          0x00004000  // Initialize to all '0s'
#define CRC_CFG_INIT_1          0x00006000  // Initialize to all '1s'
#define CRC_CFG_SIZE_8BIT       0x00001000  // Input Data Size
#define CRC_CFG_SIZE_32BIT      0x00000000  // Input Data Size
#define CRC_CFG_RESINV          0x00000200  // Result Inverse Enable
#define CRC_CFG_OBR             0x00000100  // Output Reverse Enable
#define CRC_CFG_IBR             0x00000080  // Bit reverse enable
#define CRC_CFG_ENDIAN_SBHW     0x00000020  // Swap byte in half-word
#define CRC_CFG_ENDIAN_SHW      0x00000010  // Swap half-word
#define CRC_CFG_TYPE_P8005      0x00000000  // Polynomial 0x8005
#define CRC_CFG_TYPE_P1021      0x00000001  // Polynomial 0x1021
#define CRC_CFG_TYPE_P4C11DB7   0x00000002  // Polynomial 0x4C11DB7
#define CRC_CFG_TYPE_P1EDC6F41  0x00000003  // Polynomial 0x1EDC6F41
#define CRC_CFG_TYPE_TCPCHKSUM  0x00000008  // TCP checksum
```

# ➢TM4C的硬件CRC模块的使用方法：

## 3. 设置余数初始值

CRC种子寄存器 (CRCSEED)保存了每次计算CRC的余数，在计算开始之前，要把余数初始值填入该寄存器。TivaWare提供了CRCSeedSet函数，设置CRCSEED寄存器：

```
CRCSeedSet(CCM0_BASE, 0xffff);
```

```
void
CRCSeedSet(uint32_t ui32Base, uint32_t ui32Seed)
{
    //
    // Check the arguments.
    //
    ASSERT(ui32Base == CCM0_BASE);

    //
    // Write the seed value to the seed register.
    //
    HWREG(ui32Base + CCM_O_CRCSEED) = ui32Seed;
}
```

## ➢TM4C的硬件CRC模块的使用方法：

### 4. 把数据按字节逐一写入到CRCDIN寄存器

TivaWare提供了CRCDataWrite函数，设置把要计算的数值填入CRCDIN寄存器：

```
CRCDataWrite(CCM0_BASE,QuaryFrame[0]);
CRCDataWrite(CCM0_BASE,QuaryFrame[1]);
CRCDataWrite(CCM0_BASE,QuaryFrame[2]);
CRCDataWrite(CCM0_BASE,QuaryFrame[3]);
CRCDataWrite(CCM0_BASE,QuaryFrame[4]);
CRCDataWrite(CCM0_BASE,QuaryFrame[5]);
```

CRCDataWrite函数的实现方式为：

```
void
CRCDataWrite(uint32_t ui32Base, uint32_t ui32Data)
{
ASSERT(ui32Base == CCM0_BASE);

HWREG(ui32Base + CCM_O_CRCDIN) = ui32Data;
}
```

# ➢TM4C的硬件CRC模块的使用方法：

## 5. 把校验结果从CRCRSLTPP寄存器中读出来

每向CRCDIN寄存器中填入一个数据，CRC校验都会自动计算，并把结果放入CRCSEED寄存器，并把结果异或值存入CRCRSLTPP寄存器
读出CRCRSLTPP寄存器，就可以得到CRC的校验值。
TivaWare提供了CRCResultRead函数读出计算结果

```
g_ui32Result=CRCResultRead(CCM0_BASE,true);
```

函数的实现方法为：

```
uint32_t
CRCResultRead(uint32_t ui32Base, bool bPPResult)
{
ASSERT(ui32Base == CCM0_BASE);

if(bPPResult)
    {
        return(HWREG(ui32Base + CCM_O_CRCRSLTPP));
    }
    else
    {
        return(HWREG(ui32Base + CCM_O_CRCSEED));
    }
}
```

## ➢TM4C的硬件CRC模块的初始化与应用

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_CCM0);
   while(!SysCtlPeripheralReady(SYSCTL_PERIPH_CCM0))
   {
   }
CRCConfigSet(CCM0_BASE,CRC_CFG_INIT_1 | CRC_CFG_SIZE_8BIT|
                       CRC_CFG_TYPE_P8005|CRC_CFG_IBR|CRC_CFG_OBR);

CRCSeedSet(CCM0_BASE, 0xffff);
CRCDataWrite(CCM0_BASE,QuaryFrame[0]);
CRCDataWrite(CCM0_BASE,QuaryFrame[1]);
CRCDataWrite(CCM0_BASE,QuaryFrame[2]);
CRCDataWrite(CCM0_BASE,QuaryFrame[3]);
CRCDataWrite(CCM0_BASE,QuaryFrame[4]);
CRCDataWrite(CCM0_BASE,QuaryFrame[5]);
g_ui32Result=CRCResultRead(CCM0_BASE,true);
```

或者:

```
CRCSeedSet(CCM0_BASE, 0xffff);
g_ui32Result=CRCDataProcess(CCM0_BASE, (uint32_t *)QuaryFrame,
                                                   8, true);
```