Smart Hardware Design

Conception de matériel intelligent

Diseño de hardware inteligente

スマートハードウェア設計

智能硬件设计

第五章 智能硬件的板内通信

تصميم الأجهزة الذكية

Slimme hardwareontwerpen

Σχεδίαση έξυπνου υλικού

大连理工大学-朱明

Progettazione di hardware intelligente

스마트 하드웨어 설계

Smart-Hardware-Design

Проектирование умного оборудования

5.0 思考回顾

- [5.0.0] 智能硬件的硬件
  - 智能硬件的六大硬件组成

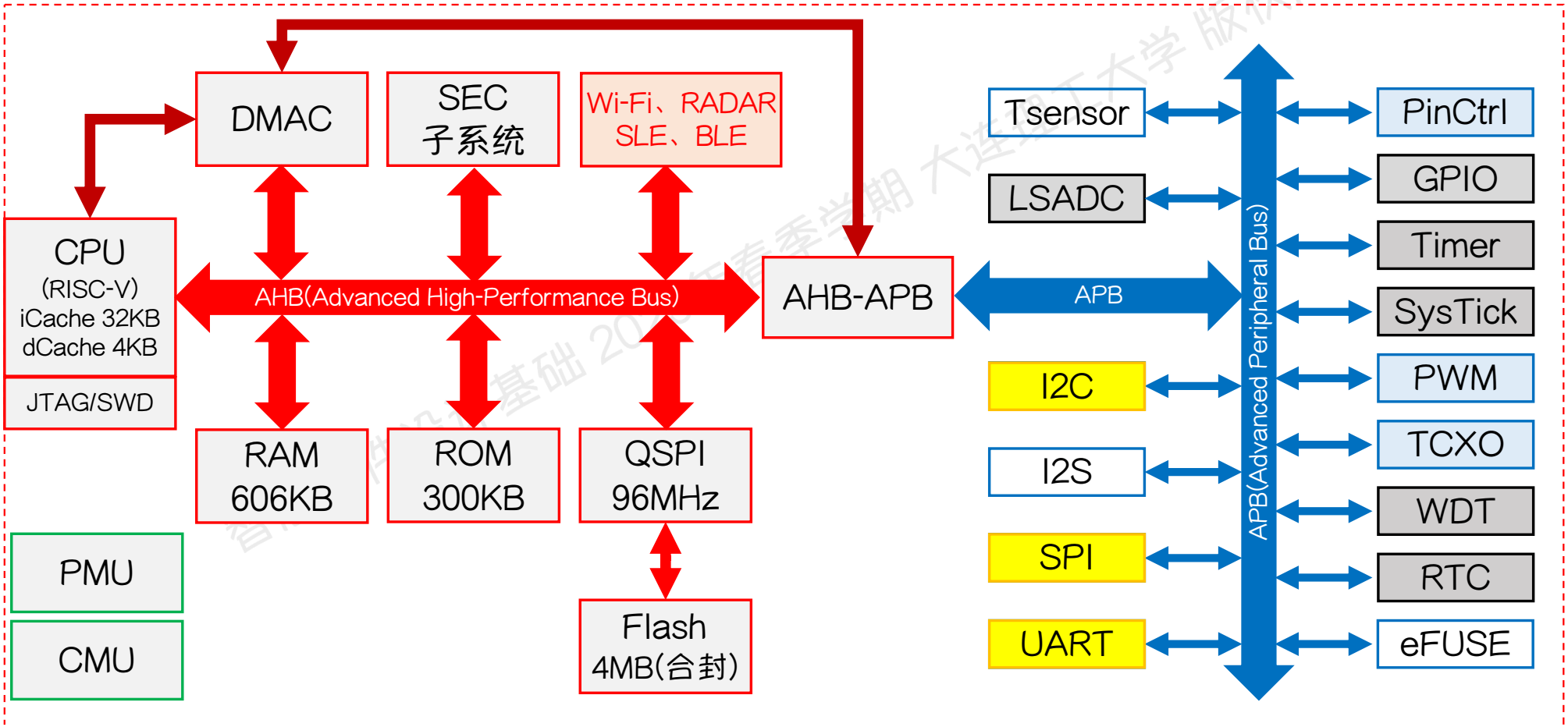
输入/感知	感知外部/内部状态，产生感知数据
通信	智能硬件之间或与外部系统的联网和数据通信
存储	保存操作系统、应用程序、数据和配置等内容
计算	进行运算和决策，控制智能硬件系统工作
输出/执行	输出计算结果，控制外部设备等

电源系统

智能硬件大多是电池供电设备，(电池)电源系统为所有系统设备供电

5.0 思考回顾

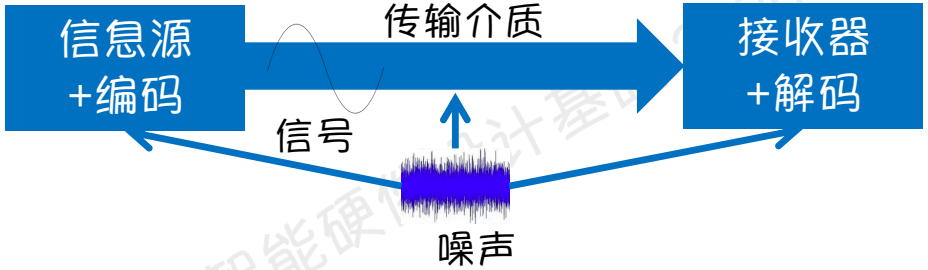
- [5.0.0] 智能硬件的SoC内部结构(华为海思WS63)



5.1 板级内部通信的构建

●[5.1.1] 通信的基础条件

- 信号层面
  - 各通信方之间，发送方发出信号能够被接收方正常接收并识别
- 信息层面
  - 各通信方之间，发送方发出信息能够被接收方正常理解和处理



协议(通信协议)
智能硬件内部的电子设备等之间进行数据交换时，各方都遵循的规则和约定。用于确保数据能够正确、有效地从发送方传输到接收方，并且通信各方能够理解和处理这些数据

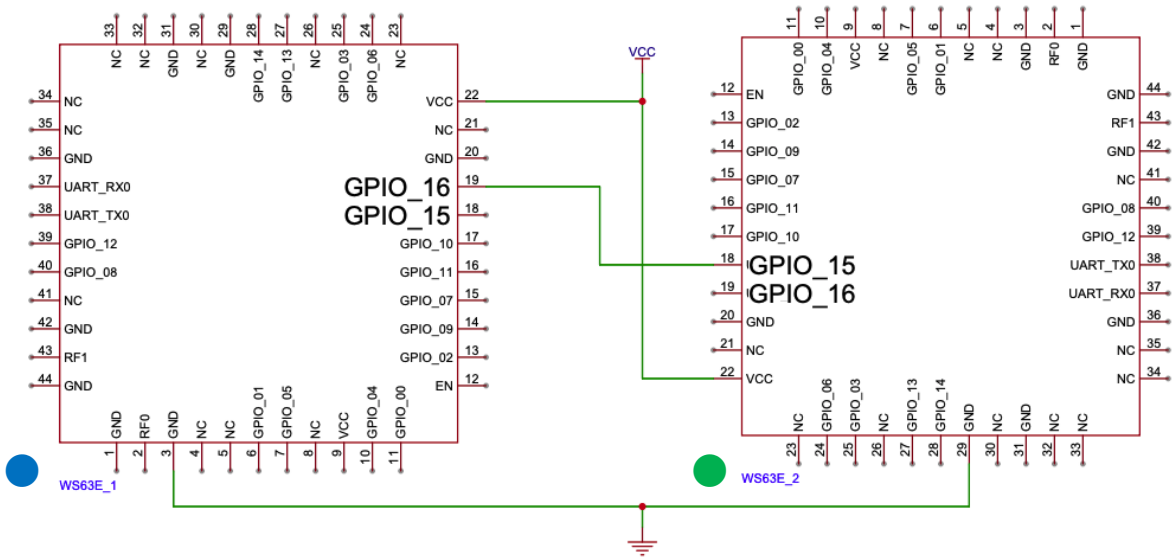
思考问题：如何以有线的形式从一个SoC向另一个SoC发送数据

通信的基础条件包括信息源、信号、传输介质、接收器、噪声、编码解码、协议和时序

5.1 板级内部通信的构建

●[5.1.2] SoC板内通信的基础

● 电气条件相同的板内通信

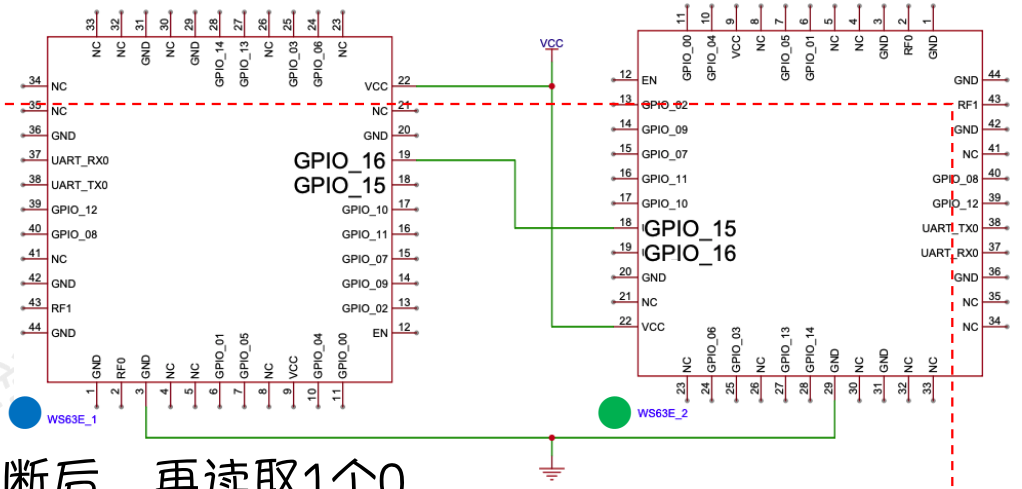


电气条件相同的说明
（板级基本通信的要求）
①共地：两片SoC的GND连接在一起并连接GND，GND=0V
②电压等级相同：两片SoC的VCC相同，WS63的VCC=3.3V
③信号电平相同：两片SoC的高电平对GND的电压相同；两片SoC的低电平对GND的电压相同

- WS63_1(简称●)向WS63_2(简称●)传递1 bit数据"1"的过程
  - ●的GPIO_16输出：uapi_gpio_set_val(GPIO_16, GPIO_LEVEL_HIGH)
  - ●的GPIO_15读取：val = uapi_gpio_get_val(GPIO_15)

5.1 板级内部通信的构建

- [5.1.2] SoC板内通信的基础
  - 电气条件相同的板内通信
    - 成功实现了1bit数据的传输
    - 如何实现2bits数据”10”的传输
      - 先输出1个1再输出一个0
      - 读取到1个1，然后有下降沿中断后，再读取1个0



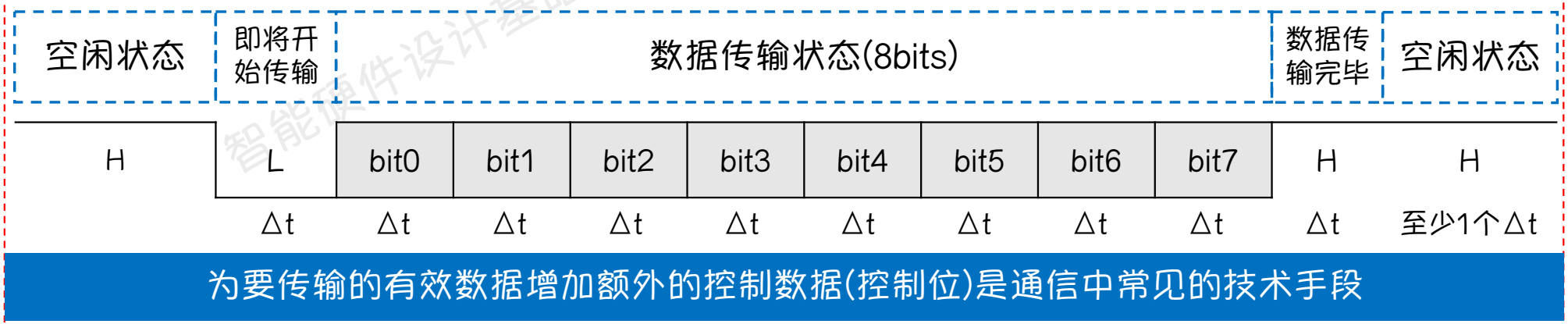
  - 如何实现2bits数据”11”的传输或者”00”的传输(不增加连线)
    - 利用前面讲过的知识点：GPIO、中断、ADC、定时器、看门狗、……
    - GPIO与中断相结合的方式传输数据
      - 利用定时器产生Δt时间的中断，每次中断发送1bit数据
      - 利用定时器产生Δt时间的中断，每次中断读取1bit数据
      - 上述过程是否会有新的问题出现，或是没有考虑周全之处

如何实现数据更长

5.2 异步串行通信技术

●[5.2.1] 异步串行通信方式设计

- 超过1bit数据的发送和接收设计
  - 如何实现1Byte任意数据传输（不增加连线）
    - GPIO与中断相结合的方式传输数据 ➤ 约定固定传输长度：1Byte
      - ●利用定时器产生Δt时间的中断，每次进入中断，发送1bit数据
      - ●利用定时器产生Δt时间的中断，每次进入中断，读取1bit数据
      - 不周全：问题在于●如何通知●何时开始传输数据



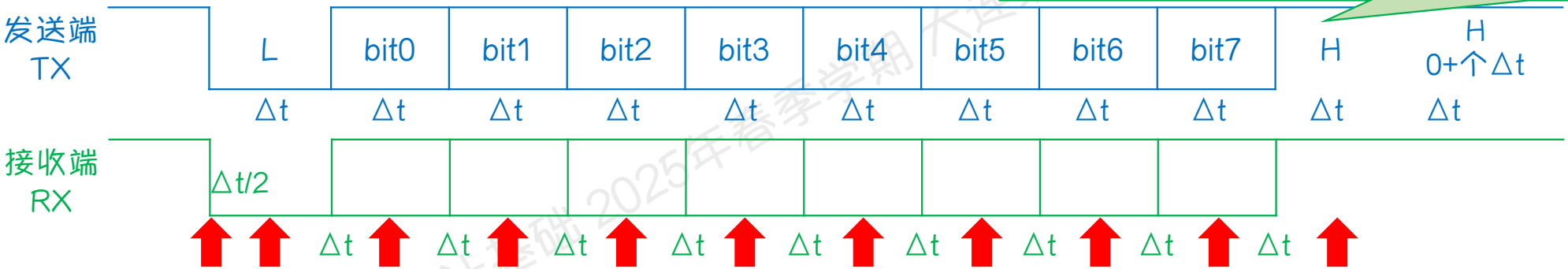
为要传输的有效数据增加额外的控制数据(控制位)是通信中常见的技术手段

5.2 异步串行通信技术

●[5.2.1] 异步串行通信方式设计

- 超过1bit数据的发送和接收设计
  - 如何实现1Byte任意数据传输（不增加连线）

时序图：描述电路中信号、事件、时钟等随时间变化行为，特别是多个信号或组件相互作用的时间和顺序。其被广泛用于描述信号的时序关系，例如时钟信号、控制信号、输入输出信号等



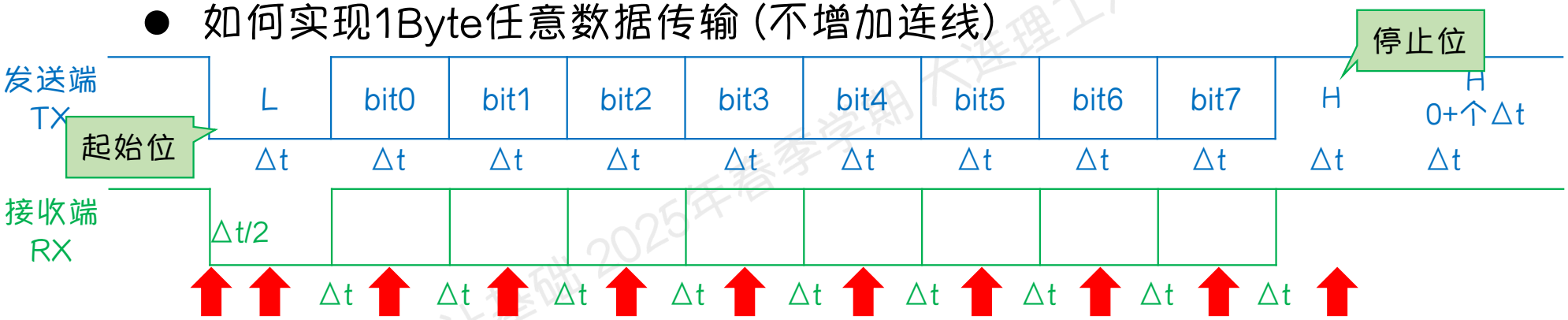
- 无传输时，检测到下降沿，延迟Δt/2后确认是L，开始接收
- 每隔Δt进行一次电平读取，8次后表示有效数据接收完毕
- 再读取一次，如果是H则表示数据接收完成且成功，否则数据无效
- 设计缺陷：每次只能传输一个字节，无法判断数据是否出错(无校验)

方法简单，只需要一根数据线，以及双方Δt等一致即可实现数据传输，且无需同步(硬件独立连接)

5.2 异步串行通信技术

[5.2.2] UART通信技术

- 超过1bit数据的发送和接收设计
  - 如何实现1Byte任意数据传输（不增加连线）



UART，Universal Asynchronous Receiver/Transmitter，通用异步收发传输器
信息系统最通用、最广泛、最经典的通信方式，常被称为串口(但实际不对)

- UART的传输速率灵活：Δt时间不易记忆，转换为速率进行表示
  - 波特率：每秒钟传输的位数(起始位+数据位+停止位+校验位)
- UART的数据格式灵活：数据位长度灵活、停止位灵活、校验位灵活

5.2 异步串行通信技术

[5.2.2] UART通信技术

- UART的基本参数
  - UART的传输速率灵活：Δt时间不易记忆，转换为速率进行表示
    - 波特率：每秒钟传输的位数(起始位+数据位+停止位+校验位)
      - 常用的波特率：9600、19200、38400、115200等
  - UART的数据格式灵活：数据位长度灵活、停止位灵活、校验位灵活
    - 起始位：固定长度，统一为1位
    - 数据位：多种长度设定，可5、6、7、8、9位
    - 停止位：多种长度设定，可1、2位
    - 校验位：多种校验方式：无0位、奇校验1位、偶校验1位
  - 常用的UART格式：9600 8N1，115200 8N1

校验位应排在何处

115200是波特率，8N1是什么

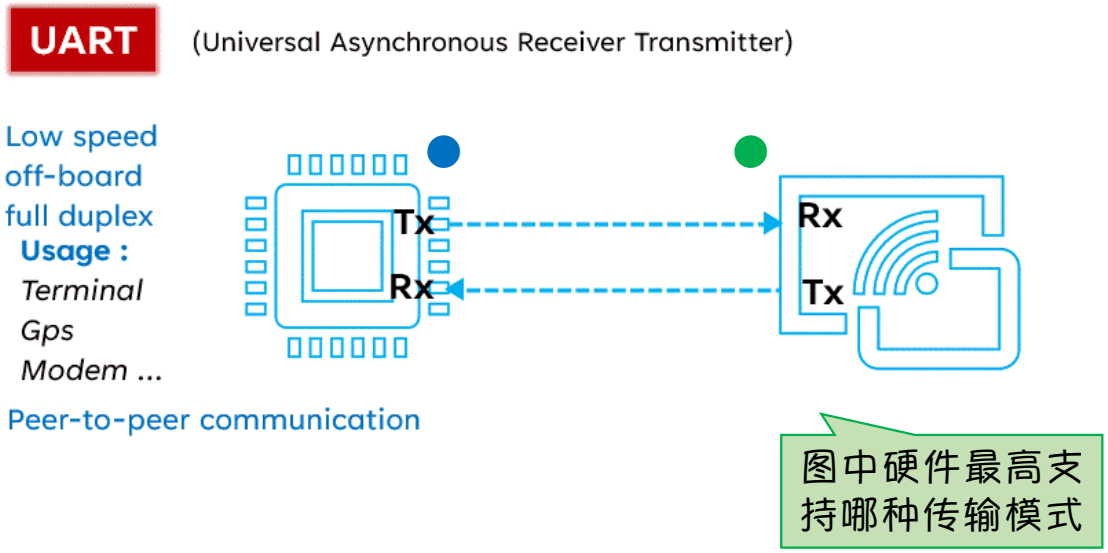
帧：一次连续完成传输的有效数据和控制信息

基于UART的双向通信实现简单，增加另一根连线即可实现

5.2 异步串行通信技术

[5.2.2] UART通信技术

- UART的连接形式
  - 蓝TX -> 绿RX
  - 绿TX -> 蓝RX
- 传输模式(不限于UART)
  - 单工(Simplex)
    - 只能由A->B
  - 半双工(Half-Duplex)
    - 可以由A->B，或者由B->A，但不能同时实现双向传输
  - 双工(全双工)(Full-Duplex)
    - 可以同时实现A->B和B->A的双向传输



5.2 异步串行通信技术

- [5.2.3] WS63的UART通信应用
  - WS63 UART的主要API
    - WS63的UART实际可控参数多于基本格式参数，API相对复杂
    - errcode_t uapi_uart_init(uart_bus_t bus, const uart_pin_config_t *pins,
      const uart_attr_t *attr,
      const uart_extra_attr_t *extra_attr,
      uart_buffer_config_t *uart_buffer_config)
      - uart_bus_t bus：设置使用的UART，WS63有三组UART硬件外设
        - UART_BUS_0、UART_BUS_1、UART_BUS_2
      - const uart_pin_config_t *pins：配置使用的UART组的引脚功能
        - UART除了RX和TX之外，还有用于流控的RTS和CTS(当前极少使用)
        - 以结构体的形式配置UART硬件中的四个功能的使用情况

- [5.2.3] WS63的UART通信应用
  - WS63 UART的主要API
    - WS63的UART实际可控参数多于基本格式参数，API相对复杂
    - errcode_t uapi_uart_init(uart_bus_t bus, const uart_pin_config_t *pins,
      const uart_attr_t *attr,

```
uart_pin_config_t pin_config = {
    .tx_pin = CONFIG_UART_TXD_PIN,          //使用TX硬件
    .rx_pin = CONFIG_UART_RXD_PIN,          //使用RX硬件
    .cts_pin = PIN_NONE,                    //不使用CTS
    .rts_pin = PIN_NONE                     //不使用RTS
};
```

      - uart_bus_t bus：设置使用
        - UART_BUS_0、UART_
      - const uart_pin_config_t *pins：配置使用的UART组的引脚功能
        - UART除了RX和TX之外，还有用于流控的RTS和CTS(流控极少使用)
        - 以结构体的形式配置UART硬件中的四个功能的使用情况

5.2 异步串行通信技术

- [5.2.3] WS63的UART通信应用
  - WS63 UART的主要API
    - errcode_t uapi_uart_init(uart_bus_t bus, const uart_pin_config_t *pins,
      const uart_attr_t *attr,
      const uart_extra_attr_t *extra_attr,
      uart_buffer_config_t *uart_buffer_config)
      - const uart_attr_t *attr：设置UART传输的基本参数格式：四个基本参数

```
uart_attr_t attr = {
    .baud_rate = UART_BAUDRATE,          //波特率
    .data_bits = UART_DATA_BIT_8,        //数据位8位
    .stop_bits = UART_STOP_BIT_1,        //停止位1位
    .parity = UART_PARITY_NONE           //无校验
};
```

5.2 异步串行通信技术

- [5.2.3] WS63的UART通信应用
  - WS63 UART的主要API
    - errcode_t uapi_uart_init(uart_bus_t bus, const uart_pin_config_t *pins,
      const uart_attr_t *attr,
      const uart_extra_attr_t *extra_attr,
      uart_buffer_config_t *uart_buffer_config)
    - const uart_extra_attr_t *extra_attr：设置UART传输的额外参数

```
uart_extra_attr_t extra_attr = {
        .tx_dma_enable = true,
        .tx_int_threshold = UART_FIFO_INT_TX_LEVEL_EQ_0_CHARACTER,
        .rx_dma_enable = true,
        .rx_int_threshold = UART_FIFO_INT_RX_LEVEL_1_CHARACTER
    };
```

这部分内容可以不设置，具体含义忽略

5.2 异步串行通信技术

[5.2.3] WS63的UART通信应用

- WS63 UART的主要API
  - errcode_t uapi_uart_init(uart_bus_t bus, const uart_pin_config_t *pins,
    const uart_attr_t *attr,
    const uart_extra_attr_t *extra_attr,
    uart_buffer_config_t *uart_buffer_config)
  - uart_buffer_config_t *uart_buffer_config：设置UART接收缓存参数
    - 接收批量数据时，建立接收缓存可以减少用户频繁中断接收保存数据的开销

```
static uart_buffer_config_t g_app_uart_buffer_config = {
    .rx_buffer = g_app_uart_rx_buff,                    //接收缓存
    .rx_buffer_size = CONFIG_UART_TRANSFER_SIZE         //缓存大小
};
```

WS63的UART初始化函数完成了UART全部通信参数的设置

5.2 异步串行通信技术

- [5.2.3] WS63的UART通信应用
  - WS63 UART的主要API
    - int32_t uapi_uart_write(uart_bus_t bus,
      const uint8_t *buffer,
      uint32_t length, uint32_t timeout)

      阻塞模式
      等待发送结束才有返回值
      返回值为发送完成的长度

      发送超时时间(最大执行时间)
    - errcode_t uapi_uart_write_int(uart_bus_t bus,
      const uint8_t *buffer,
      uint32_t length, void *params,
      uart_tx_callback_t finished_with_buffer_func)

      非阻塞模式
      函数执行后可以立即返回
      发送完成则触发中断
    - 尝试分析上述两种通过UART发送数据方法，有何相同和区别
      - 相同：控制特定UART连续发送特定长度的缓存中的数据(8位)
      - 不同：发送过程的系统响应方式不同

5.2 异步串行通信技术

- [5.2.3] WS63的UART通信应用
  - WS63 UART的主要API
    - int32_t uapi_uart_read(uart_bus_t bus,
      const uint8_t *buffer,
      uint32_t length, uint32_t timeout)

轮询(阻塞)
等待接收到足够数据才返回
返回值为完成接收的长度

接收超时时间(最大执行时间)

想要读取的数据长度

    - errcode_t uapi_uart_register_rx_callback(uart_bus_t bus,
      uart_rx_condition_t condition,
      uint32_t size, uart_rx_callback_t callback)

产生中断的条件
(与缓存状态和传输状态相关)
需求1：缓存满
需求2：到达接收数据量
需求3：不再收到新数据

组合1：UART_RX_CONDITION_FULL_OR_IDLE
组合2：UART_RX_CONDITION_FULL_OR_SUFFICIENT_DATA
组合3：UART_RX_CONDITION_FULL_OR_SUFFICIENT_DATA_OR_IDLE

5.2 异步串行通信技术

- [5.2.3] WS63的UART通信应用
  - WS63 UART的主要API
    - typedef void (*uart_tx_callback_t)(const void *buffer, uint32_t length, const void *params)

发送

```
uint8_t *buff = (void *)buffer;
for (uint8_t i = 0; i < length; i++) {
    osal_printk("uart TX data[%d] = %d\r\n", i, buff[i]);
}
```

    - typedef void(*uart_rx_callback_t)(const void *buffer, uint16_t length, bool error)

接收

```
uint8_t *buff = (uint8_t *)buffer;
if (memcpy_s(g_app_uart_rx_buff, length, buff, length) != EOK) {
    osal_printk("uart int mode data copy fail!\r\n");
    return;
}
```

5.2 异步串行通信技术

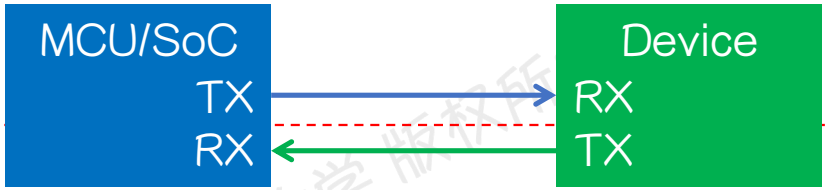
●[5.2.3] WS63的UART通信应用

● WS63 UART的其他API

①：errcode_t uapi_uart_deinit(uart_bus_t bus)	去初始化指定UART
②：errcode_t uapi_uart_set_attr(uart_bus_t bus, const uart_attr_t *attr)	设置指定UART的基本属性
③：errcode_t uapi_uart_get_attr(uart_bus_t bus, uart_attr_t *attr)	获取指定UART的基本属性
④：bool uapi_uart_has_pending_transmissions(uart_bus_t bus)	判断指定UART是否在传输
⑤：bool uapi_uart_rx_fifo_is_empty(uart_bus_t bus)	判断指定UART发送缓存是否为空
⑥：bool uapi_uart_tx_fifo_is_empty(uart_bus_t bus)	判断指定UART接收缓存是否为空
⑦：void uapi_uart_unregister_rx_callback(uart_bus_t bus)	去注册指定UART接收回调函数
⑧：errcode_t uapi_uart_suspend(uintptr_t arg)	挂起指定UART的传输(暂不生效)
⑨：errcode_t uapi_uart_resume(uintptr_t arg)	恢复指定UART的传输(暂不生效)

5.2 异步串行通信技术

[5.2.4] UART通信的局限性



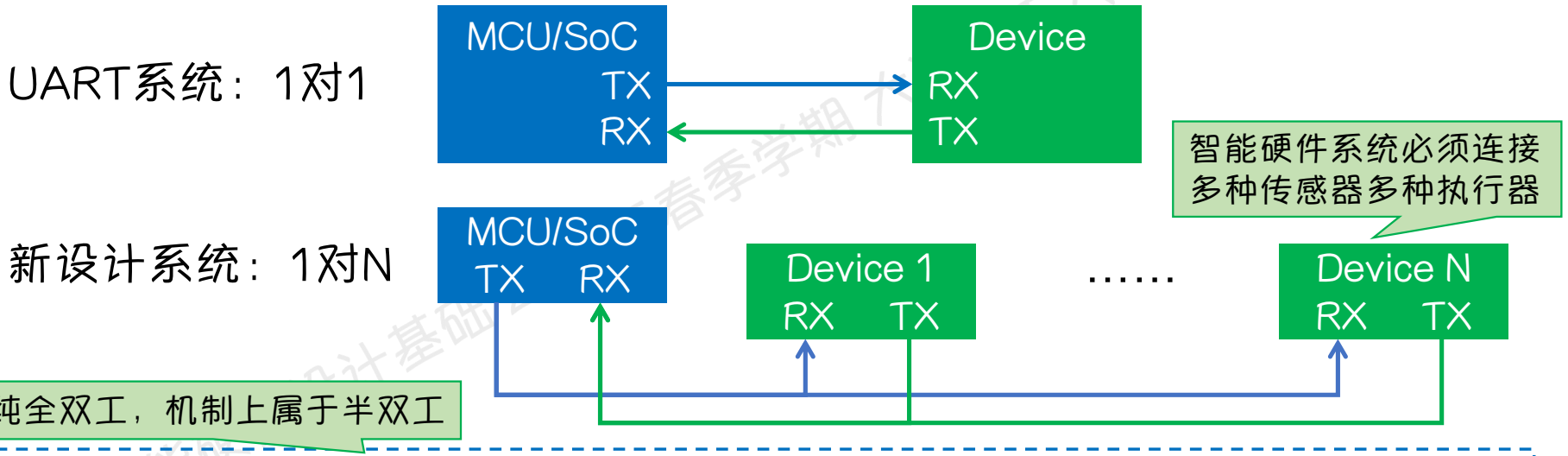
- UART机制主要问题
  - 通信速率低：UART常见最高速率只有115200bps(信号层面的异步问题)
  - 异步通信、缺乏流控：发送方和接收方需要通过软件来管理数据流，若接收方的缓冲区已满而发送方仍在发送数据，会引起数据丢失
  - 原生UART系统的设备数量受限：UART只能连接两个设备，若需要多个设备相互通信，只能使用多个UART接口
    - 可通过技术手段扩展成RS422/485，实现多个节点通信
    - RS485双线差分，最大电压差能达到±10V以上，且只能半双工传输
  - 原生UART系统的通信距离受限：高速下UART的安全通信距离不到1米
    - 但可通过技术手段扩展成RS232/485，实现数十米至公里级的通信

UART的通信设备数量受限，以及通信速率低，使MCU/SoC难以同时连接多个外设

5.3 同步串行通信技术

●[5.3.1] 协议改进与设计

● 如何实现两条线上连接多个设备高速通信



新系统工作流程约定

(1)所有通信均发生在MCU/SoC与某个Device之间的，MCU/SoC选择与哪个Device通信

(2)MCU/SoC通过■发出的指令/数据，都应该能够被Device识别是否是针对其自身

(3)Device仅识别并响应针对自身的指令/数据，再通过■返回响应MCU/SoC的状态/数据

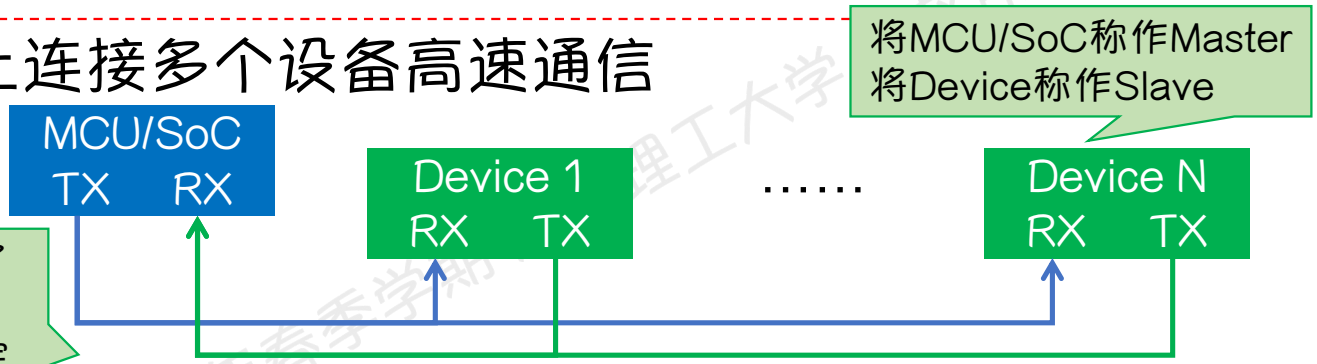
(4)Device只能响应MCU/SoC的查询，不能主动发起状态/数据的上报，不能主动占用■

5.3 同步串行通信技术

●[5.3.1] 协议改进与设计

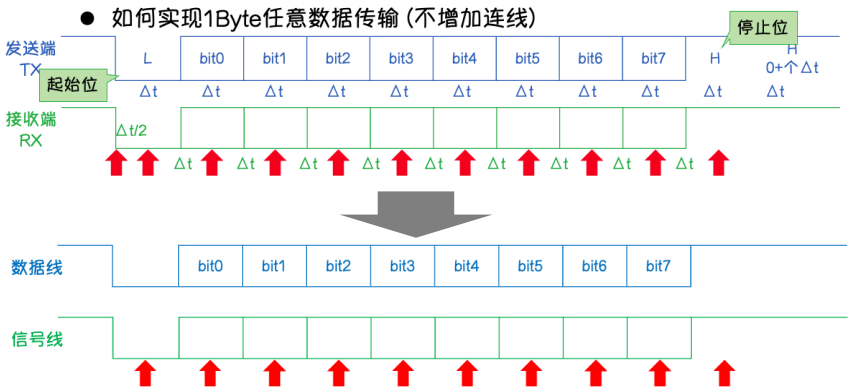
- 如何实现两条线上连接多个设备高速通信

新设计系统：1对N



上述工作流程的约定基本实现了MCU/SoC与Device的有序通信
但更具体如如何选择Device未定

- 信号层面(异步传输速率低的原因)：接收方自身控制接收(识别)的时间点
- 高速通信问题如何解决：同步
  - MCU/SoC控制接收(识别)的时间点
  - 用一根线作为控制接收(识别)信号线
  - 另一根线作为双向通信数据线(Data)

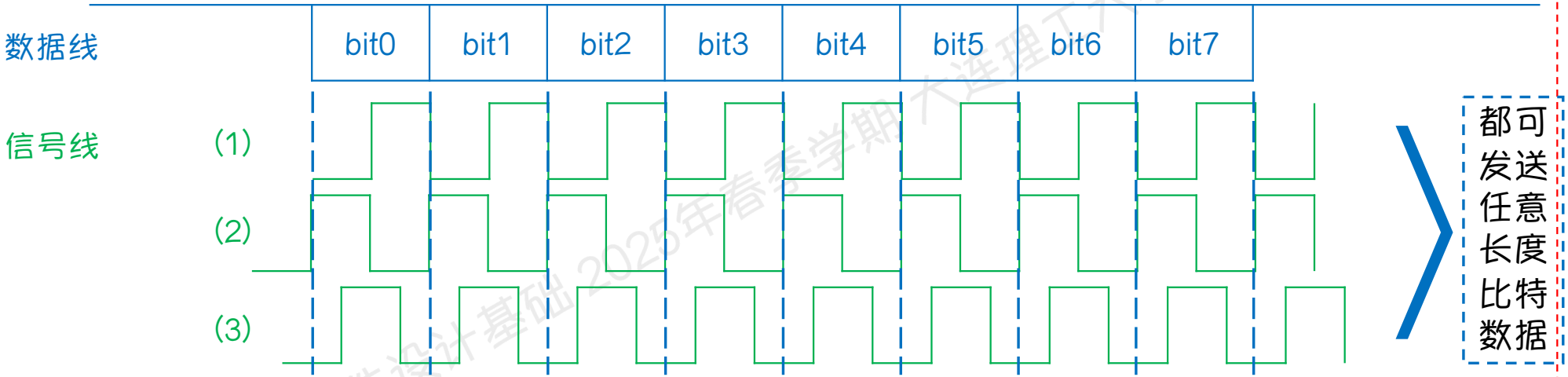


如何选择目标Device、在何种信号时识别数据等

5.3 同步串行通信技术

[5.3.1] 协议改进与设计

- 如何实现两条线上连接多个设备高速通信：同步采样



- 问题1：设定信号线上的何种信号控制Slave识别数据线电平
  - 方案(1)：在信号线的上升沿时进行采样
  - 方案(2)：在信号线的上升沿时进行采样
  - 方案(3)：在特定电平时进行采样

并无优劣区别

5.3 同步串行通信技术

[5.3.1] 协议改进与设计

- 如何实现两条线上连接多个设备高速通信：目标地址
  - Master -> Slave

帧格式

起始	地址	数据/指令	数据/指令	数据/指令	……	数据/指令	停止

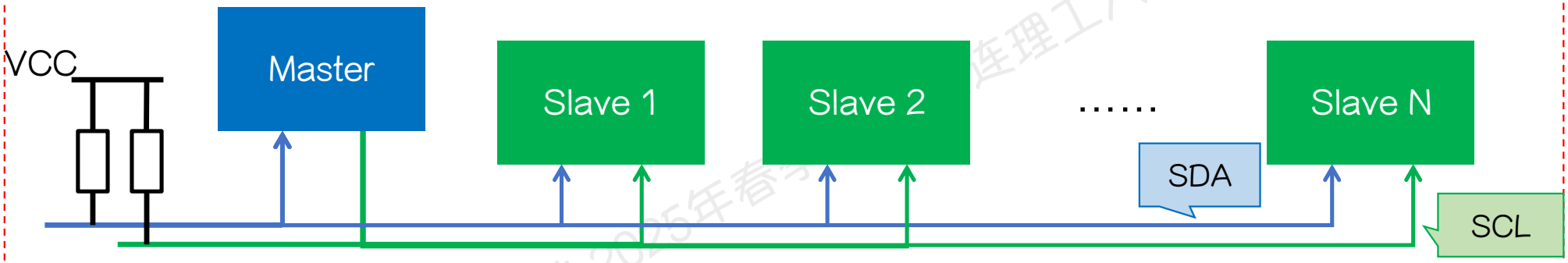
  - 很多问题需要解决
    - 空闲时信号线电平状态
    - 起始信号定义和停止信号定义
    - Master如何快速确认Slave是否存在、是否响应
    - Master如何快速实现读或者写这两种不同操作
  - 协议应该加强Master与Slave之间的传输与响应机制，提升系统效率

合理高效地解决上述问题，就构成了最高通信速率可达3.4Mbps的I^2C协议

5.3 同步串行通信技术

[5.3.2] I²C总线通信协议

I2C通信系统的结构与要点

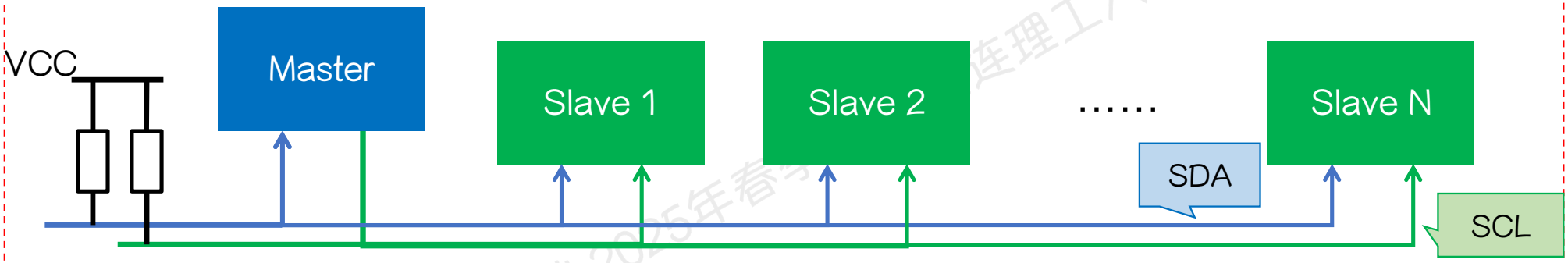


- 主设备Master：控制系统通信的设备
  - 生成时钟信号，发起数据传输，指定通信目标
- 从设备Slave：响应Master请求，接收或发送数据，没有时钟控制能力
- 数据线SDA：Serial Data Line，串行数据线，用于传输数据
- 时钟线SCL：Serial Clock Line，串行时钟线，用于同步数据传输

5.3 同步串行通信技术

●[5.3.2] I²C总线通信协议

- I2C通信系统的结构与要点



- 设备地址：每个I2C设备都有唯一的地址，一般是7位(居多)或10位
- 数据传输：同步传输协议，Master控制时钟信号，控制数据传输同步
  - 传输的数据以字节为最小单位，数据与控制信息共同构成I2C的帧结构
- 半双工通信：同一时刻只能有一个设备传输数据，但可分时双向传输
- 多设备支持：不仅支持多个Slave，也可以支持多个主设备

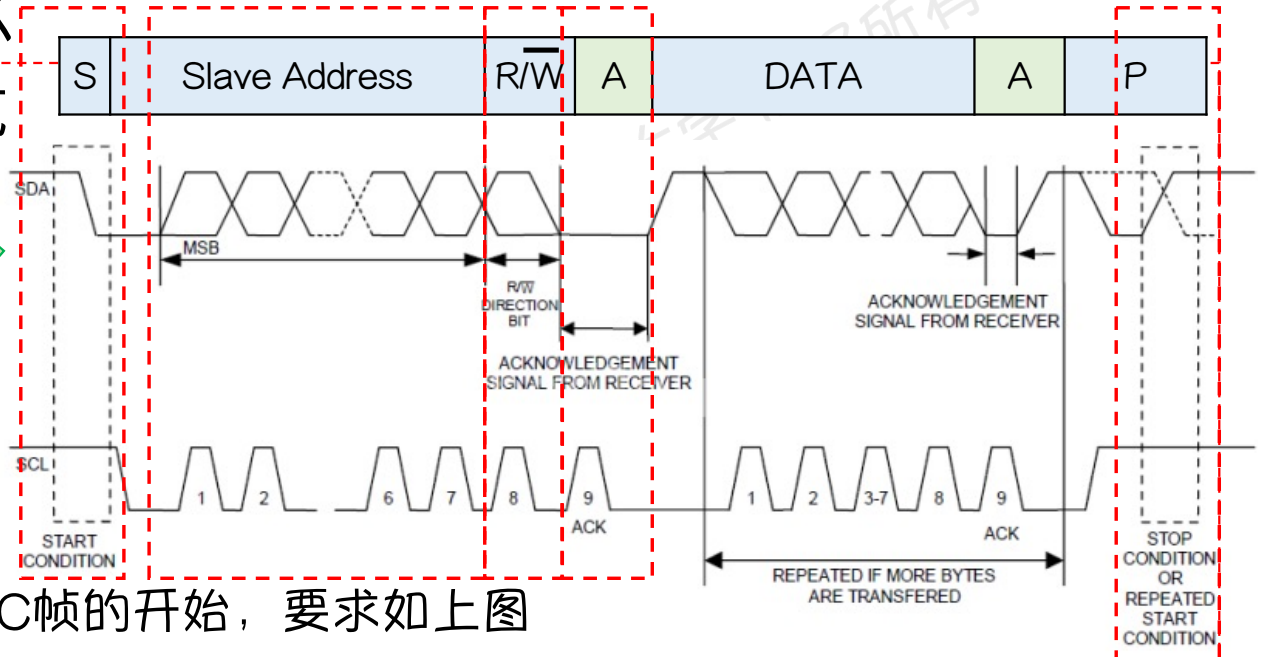
5.3 同步串行通信技术

[5.3.2] I²C总线通信协议

I2C通信系统的帧格式

时序图：描述电路中信号、事件、时钟等随时间变化行为，特别是多个信号或组件相互作用的时间和顺序。其被广泛用于描述信号的时序关系，例如时钟信号、控制信号、输入输出信号等

SDA与SCL之间的时间与顺序关系



- 起始位S(Start)：标识I2C帧的开始，要求如上图
- 地址Address：Slave的地址，上图所示为7位地址(最常见地址长度)
- 读写标志 R/$\overline{W}$：该位为0表示向Slave写入，1表示从Slave读取(问题：如何读取)
- 响应(A)ACK：Slave收到一个字节数据后，向SDA输出低电平并保持一个SCL
- 停止位P(STOP)：Master在ACK后输出，标识I2C帧的结束，要求如上图

5.3 同步串行通信技术

- [5.3.2] I²C总线通信协议
  - I2C通信系统的实现
    - 如果用GPIO实现I2C

(1)
```
#define  DEMO_SCL   GPIO_16
#define DEMO_SDA   GPIO_15
```

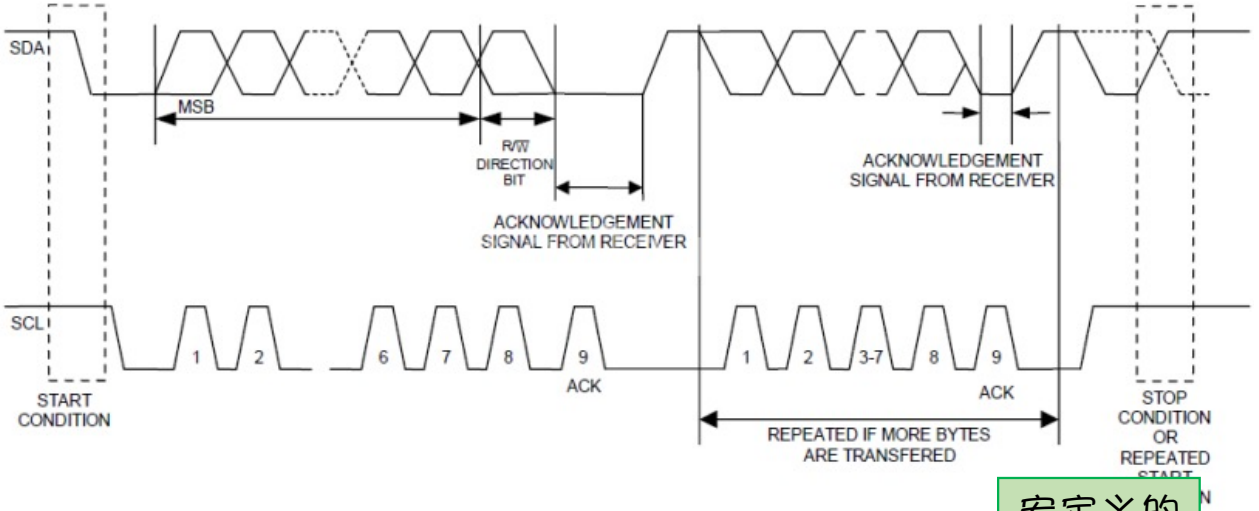
(2) 方向等初始化工作，略

(3)
```
#define DEMO_SCL_HIGH()    uapi_gpio_set_val(DEMO_SCL, GPIO_LEVEL_HIGH)
#define DEMO_SCL_LOW()     uapi_gpio_set_val(DEMO_SCL, GPIO_LEVEL_LOW)
#define DEMO_SDA_HIGH()    uapi_gpio_set_val(DEMO_SDA, GPIO_LEVEL_HIGH)
#define DEMO_SDA_LOW()     uapi_gpio_set_val(DEMO_SDA, GPIO_LEVEL_LOW)
```

宏定义的重要功能

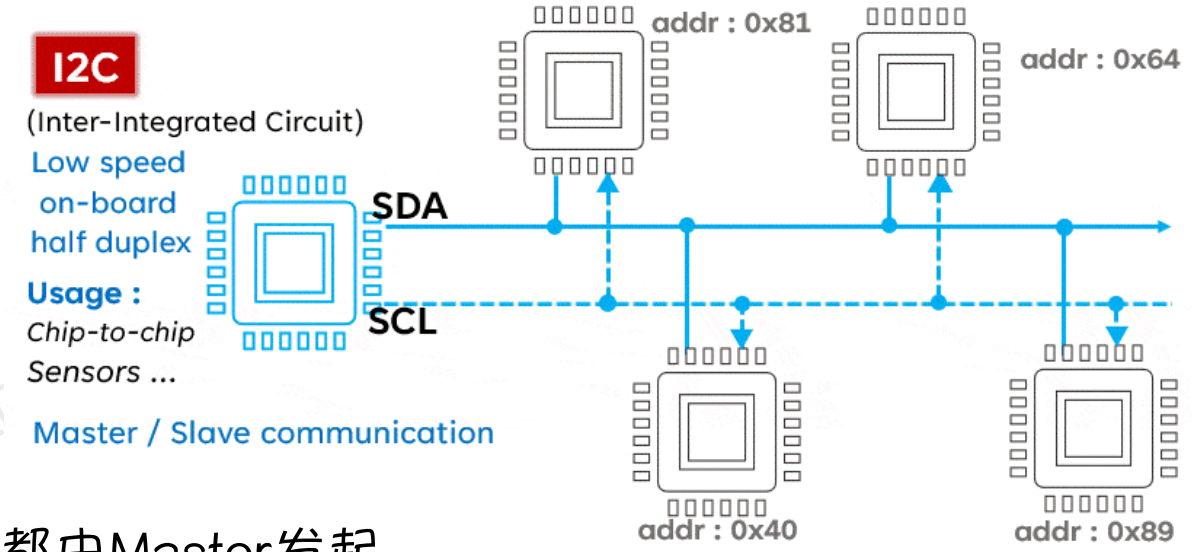
(4) 一起来完成：编写起始位S的函数demo_Start(void)和停止位P的函数demo_Stop(void)

(5) 一起来完成：编写起始位发送一个字节的函数demo_SendByte(unint8_t data)



5.3 同步串行通信技术

- [5.3.2] I²C总线通信协议
  - I2C的连接形式
    - 所有的SDA连在一起
    - 所有的SCL连在一起
  - I2C的传输形式
    - 半双工通信
    - 靠地址区分Slave
    - 所有的通信(包括读取)都由Master发起
  - I2C的传输速率
    - 标准模式(Standard Mode)100Kbps、快速模式(Fast Mode)400KBps、增强快速模式(Fast Mode+)1Mbps、高速模式(High Speed Mode)3.4Mbps



- [5.3.4] WS63的I2C通信应用
  - WS63 I2C的主要API - Master
    - I2C与UART的主要不同点：
      - UART双方通信上是对等的，没有主从之分
      - I2C在通信上，Master控制着I2C总线全部设备的接收和发送行为

为何UART没有被称作总线

    - Master与Slave工作模式的区别导致I2C的API出现很大差别
    - errcode_t uapi_i2c_master_init(i2c_bus_t bus,
      uint32_t baudrate, uint8_t hscode)
      - i2c_bus_t bus：设置使用的I2C，WS63有两组I2C硬件外设
        - I2C_BUS_0、I2C_BUS_1
      - uint32_t baudrate：I2C总线SCL频率，数值即可，如400000表示400Kbps
      - uint8_t hscode：I2C总线中Master的地址，可设定值为[0, 7]

5.3 同步串行通信技术

- [5.3.4] WS63的I2C通信应用
  - WS63 I2C的主要API - Master
    - errcode_t uapi_i2c_master_write(i2c_bus_t bus,
      uint16_t dev_addr, i2c_data_t *data)
      - uint16_t dev_addr：通信目标Slave的地址
      - i2c_data_t *data：待发送的数据的结构体的指针

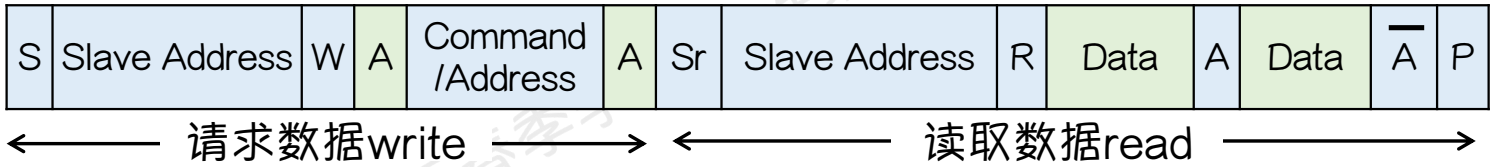
```
typedef struct i2c_data {
    uint8_t *send_buf;          //发送数据的buffer指针
    uint32_t send_len;          //发送数据的buffer长度
    uint8_t *receive_buf;       //接收数据的buffer指针
    uint32_t receive_len;       //接收数据的buffer长度
} i2c_data_t;
```

    - errcode_t uapi_i2c_master_read(i2c_bus_t bus,
      uint16_t dev_addr, i2c_data_t *data)

5.3 同步串行通信技术

[5.3.4] WS63的I2C通信应用

- WS63 I2C的主要API - Master
  - 从Slave读取(寄存器地址/指令)的流程(先写再读)



  - 注意ACK的响应原则：谁接收前面的字节(地址/数据/指令等)谁响应
  - errcode_t uapi_i2c_master_writeread(i2c_bus_t bus,
    uint16_t dev_addr, i2c_data_t *data)

```
typedef struct i2c_data {
    uint8_t *send_buf;          //发送数据的buffer指针，保存Command(待发送的指令)
    uint32_t send_len;          //发送数据的buffer长度，按照图示为1
    uint8_t *receive_buf;       //接收数据的buffer指针，保存Data
    uint32_t receive_len;       //接收数据的buffer长度，按照图示为2
} i2c_data_t;
```

5.3 同步串行通信技术

- [5.3.4] WS63的I2C通信应用
  - WS63 I2C的主要API - Slave
    - Slave模式下，WS63只能被动发送数据，不能主动发送数据
    - errcode_t uapi_i2c_slave_init(i2c_bus_t bus,
      uint32_t baudrate, uint16_t addr)
      - i2c_bus_t bus和uint32_t baudrate含义与Master相同
      - errcode_t uapi_i2c_slave：WS63在Slave模式下的地址
    - errcode_t uapi_i2c_slave_read(i2c_bus_t bus, i2c_data_t *data)
    - errcode_t uapi_i2c_slave_write(i2c_bus_t bus, i2c_data_t *data)

无论read还是write，Slave都应受Master的控制

Master/Slave的read/write/writeread都是密切交互的，需要等待数据或等待ACK

5.3 同步串行通信技术

[5.3.4] WS63的I2C通信应用

- WS63 I2C的主要API - 轮询模式
  - I2C的驱动层功能支持以轮询(POLL)的形式发送数据
  - 轮询模式的默认超时时间I2C_WAIT_CONDITION_TIMEOUT=1000ms
- WS63 I2C的主要API - 中断模式
  - I2C的驱动层功能支持以中断的形式发送和接收数据
    - OpenHarmony/LiteOS可以正常进行任务调度

    机制上不阻塞
  - 对用户而言，相关API依然是阻塞的
    - 需要等待API执行完成并返回执行的结果
    - ERRCODE_SUCC表示执行成功，否则表示执行失败

    形态上阻塞

对I2C而言，如果系统负载低(任务相对较少)，轮询模式和中断模式(以及DMA模式)性能差别很小

5.3 同步串行通信技术

[5.3.5] WS63的I2C通信示例

WS63 I2C与外部设备(RTC)通信示例

- 外部RTC类型：国产RTC芯片INS5699

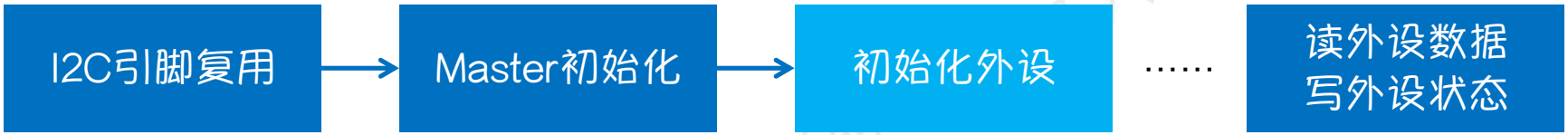
时间戳RTC，以年月日时分秒形式交互

Address	功能	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	可操作
0x00	SEC	无效	BCD码，秒十位，[0, 5]			BCD码，秒个位，[0, 9]				R/W
0x01	MIN	无效	BCD码，分十位，[0, 5]			BCD码，分个位，[0, 9]				R/W
0x02	HOUR	无效	无效	BCD码，时十位，[0, 2]		BCD码，时个位，[0, 9]				R/W
0x03	WEEK	无效	6	5	4	3	2	1	0	R/W
0x04	DAY	无效	无效	BCD码，日十位，[0, 3]		BCD码，日个位，[0, 9]				R/W
0x05	MONTH	无效	无效	无效	BCD码，月十位，[0, 1]	BCD码，月个位，[0, 9]				R/W
0x06	YEAR	BCD码，年十位，[0, 9]				BCD码，年个位，[0, 9]				R/W

什么是BCD码

不同类型的RTC芯片对于RTC数据的存储方式略有不同，应查阅厂商资料(PDF文件)确定

- [5.3.5] WS63的I2C通信示例
  - WS63 I2C与外部设备(RTC)通信示例



  - 设置WS63的引脚复用

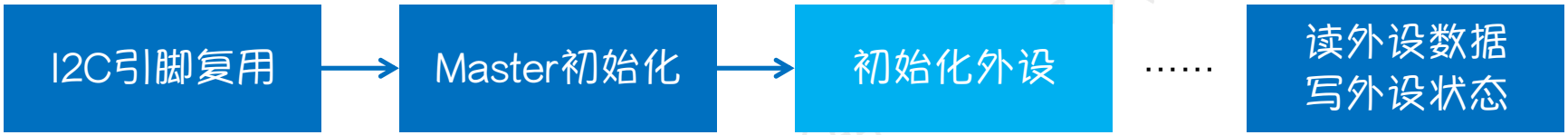
```
#define CONFIG_I2C_SCL_MASTER_PIN      15
#define CONFIG_I2C_SDA_MASTER_PIN      16
#define CONFIG_I2C_MASTER_PIN_MODE     2
//……
```

IO/MODE	0	1	2
GPIO_14	GPIO_14	UART1_RTS	RADAR_ANT1_S
UART1_TXD	GPIO_15	UART1_TXD	I2C1_SDA
UART1_RXD	GPIO_16	UART1_RXD	I2C1_SCL
UART0_TXD	GPIO_17	UART0_TXD	I2C0_SDA
UART0_RXD	GPIO_18	UART0_RXD	I2C0_SCL

```
uapi_pin_set_mode(CONFIG_I2C_SCL_MASTER_PIN, CONFIG_I2C_MASTER_PIN_MODE);
uapi_pin_set_mode(CONFIG_I2C_SDA_MASTER_PIN, CONFIG_I2C_MASTER_PIN_MODE);
```

5.3 同步串行通信技术

- [5.3.5] WS63的I2C通信示例
  - WS63 I2C与外部设备(RTC)通信示例



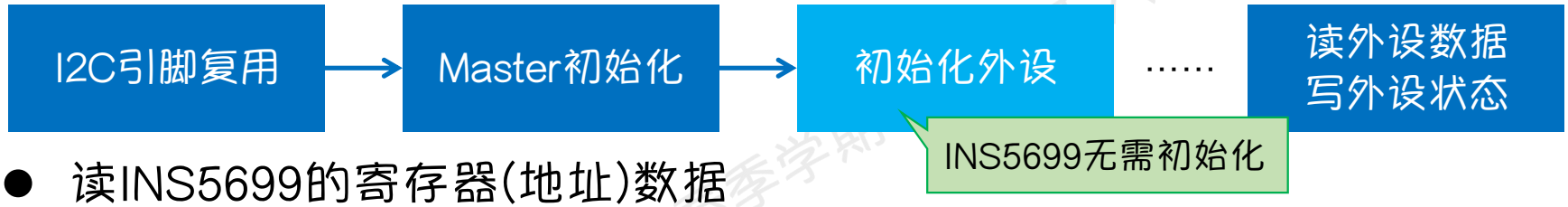
  - 设置WS63 I2C的速率和地址

```
#define I2C_MASTER_ADDR      0x0                                  //Master地址
#define I2C_SET_BANDRATE     100000                               //标准模式100Kbps
uint32_t baudrate = I2C_SET_BANDRATE;
uint32_t hscode = I2C_MASTER_ADDR;
errcode_t ret = uapi_i2c_master_init(1, baudrate, hscode);    //Master模式
if (ret != 0) {
    printf("i2c init failed, ret = %0x\r\n", ret);
}
```

5.3 同步串行通信技术

[5.3.5] WS63的I2C通信示例

WS63 I2C与外部设备(RTC)通信示例



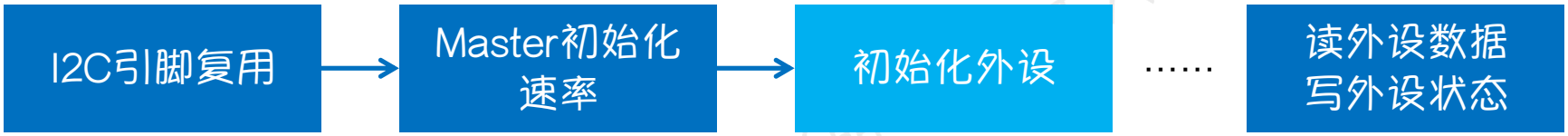
读INS5699的寄存器(地址)数据

```
uint8_t ins5699s_ReadREG(uint8_t reg){
    uint8_t send_buffer[] = {reg};                                  //发送缓存：寄存器地址
    uint8_t read_buffer[1] = {0};                                   //接收缓存：0
    i2c_data_t data = {0};                                          //结构体，四个元素
    data.send_buf = send_buffer;                                    //发送的数据
    data.send_len = sizeof(send_buffer);                            //发送数据的长度：1Byte
    data.receive_buf = read_buffer;                                 //接收的数据
    data.receive_len = 1;                                           //接收数据的长度：1Byte
    errcode_t ret = uapi_i2c_master_writeread(I2C_MASTER_BUS_ID, INS5699S_ADDR >> 1, &data);
    if (ret != 0) {    printf("INS5699S:I2cReadREG(%02X) failed, %0X!\n", reg, ret);    return 0;    }
    return data.receive_buf[0];                                     //返回接收到的数据
}
```

5.3 同步串行通信技术

[5.3.5] WS63的I2C通信示例

WS63 I2C与外部设备(RTC)通信示例



读INS5699的时间数据

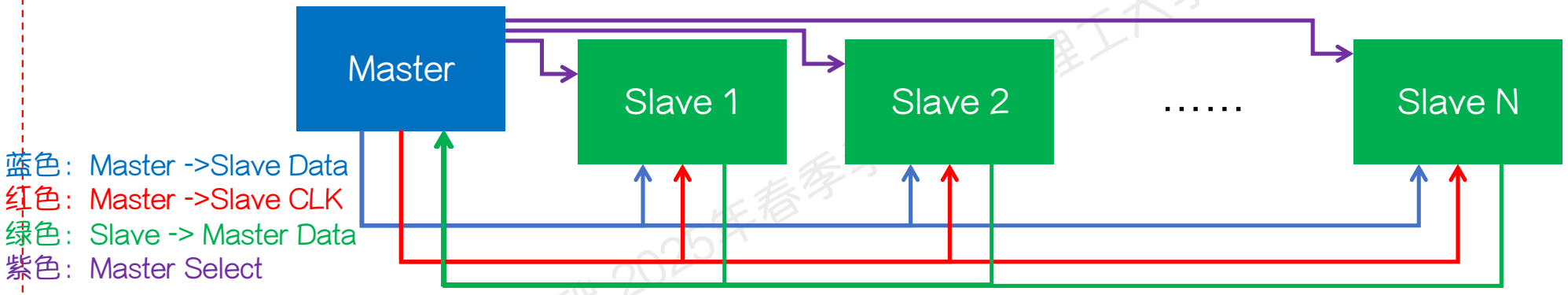
```
ins5699s_time ins5699s_GetTime(void){
    ins5699s_time time;
    uint8_t t_reg;
    t_reg = ins5699s_ReadREG(INS5699S_REG_SEC);
    time.sec = (((t_reg & 0x70) >> 4) * 10) + (t_reg & 0x0F);
    t_reg = ins5699s_ReadREG(INS5699S_REG_MIN);
    time.min = (((t_reg & 0x70) >> 4) * 10) + (t_reg & 0x0F);
    t_reg = ins5699s_ReadREG(INS5699S_REG_HOUR);
    time.hour = (((t_reg & 0x30) >> 4) * 10) + (t_reg & 0x0F);
    t_reg = ins5699s_ReadREG(INS5699S_REG_WEEK);
    //处理星期time.week、日time.day、月time.month、年time.year的数据
    return time;
}
```

```
typedef struct {
    uint8_t sec;
    uint8_t min;
    uint8_t hour;
    uint8_t week;
    uint8_t day;
    uint8_t month;
    uint8_t year;
} ins5699s_time;
```

课堂思考：如何处理四项数据

5.4 全双工同步串行通信技术

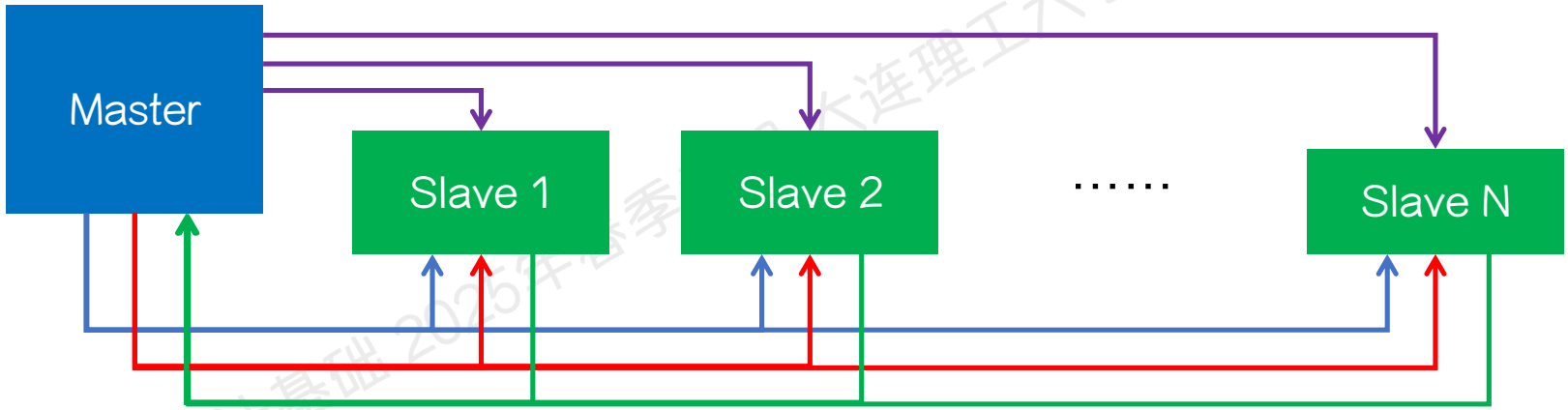
- [5.4.1] I2C协议的改进设计
  - I2C协议的缺点：半双工、通信选定Slave(慢)等



  - I2C协议适用领域：外部低速总线，传感器等数据量小的外设
    - 无法应对大数据量的双向传输 -> 尽可能少的扩展线的数量
    - 改进1：上图SDA从双向变为单向，只是实现从Master->Slave
      - 再增加一条线，实现从Slave -> Master
    - 改进2：每个Slave都是用单独的线进行选择

5.4 全双工同步串行通信技术

- [5.4.2] SPI总线通信协议
  - SPI通信系统的特点与结构

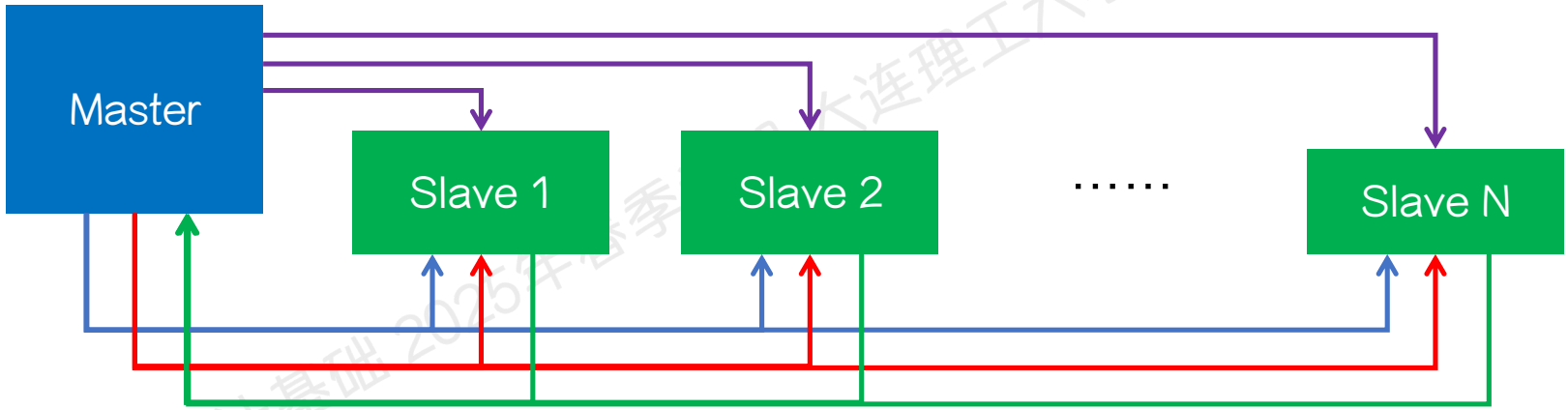


  - 全双工通信：同一个时钟周期内同时进行发送和接收
  - 同步通信：通过时钟进行与Slave的同步发送和接收
  - 传输速率高：速率可以达到几十Mbps以上
  - 无地址多设备：通过独立的信号线选择Slave进行通信

5.4 全双工同步串行通信技术

[5.4.2] SPI总线通信协议

- SPI通信系统的结构与要点

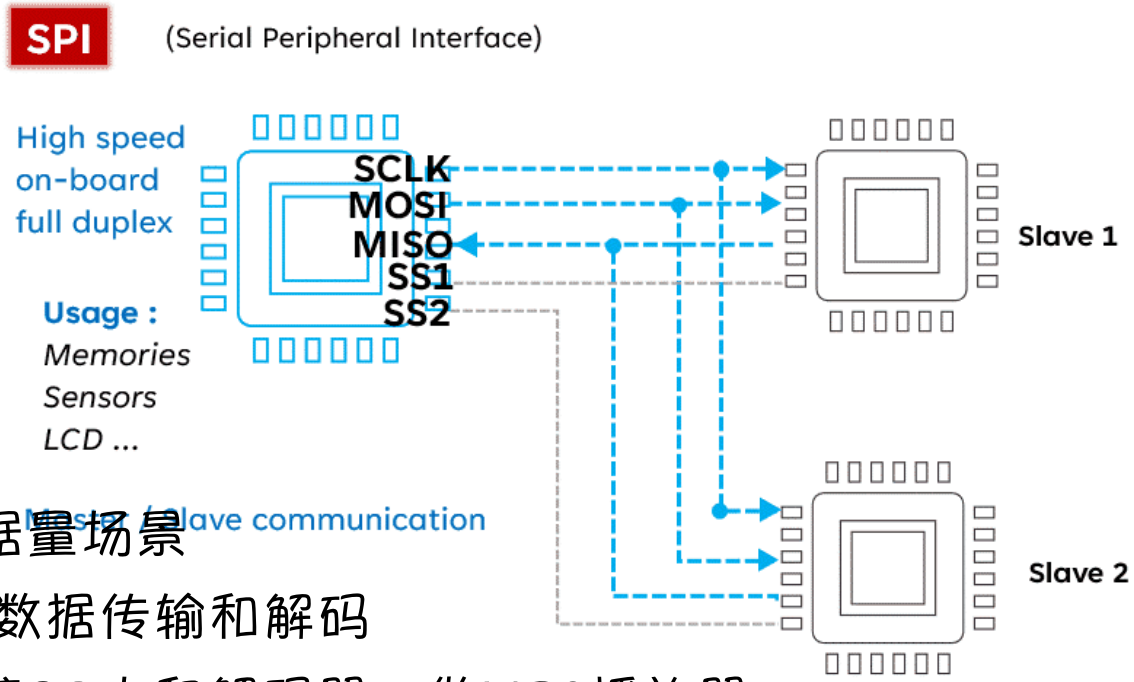


- MOSI(Master Out Slave In)：主设备发送，所有从设备均能接收
- MISO(Master In Slave Out)：某从设备发送，所有设备均能接收
- SCK(Serial Clock)：主设备发送，所有从设备均能接收
- CS(Chip Select)：主设备发送，只有特定的从设备被选中

5.4 全双工同步串行通信技术

[5.4.2] SPI总线通信协议

- SPI的连接形式
  - 所有的MOSI连在一起
  - 所有的MISO连在一起
  - 所有的SCK连在一起
  - 每个Slave有独立的CS
- SPI的应用场景
  - MCU/SoC的高速、大数据量场景
  - SD卡、Flash存储、音频数据传输和解码
  - 例如，利用MCU/SoC连接SD卡和解码器，做MP3播放器



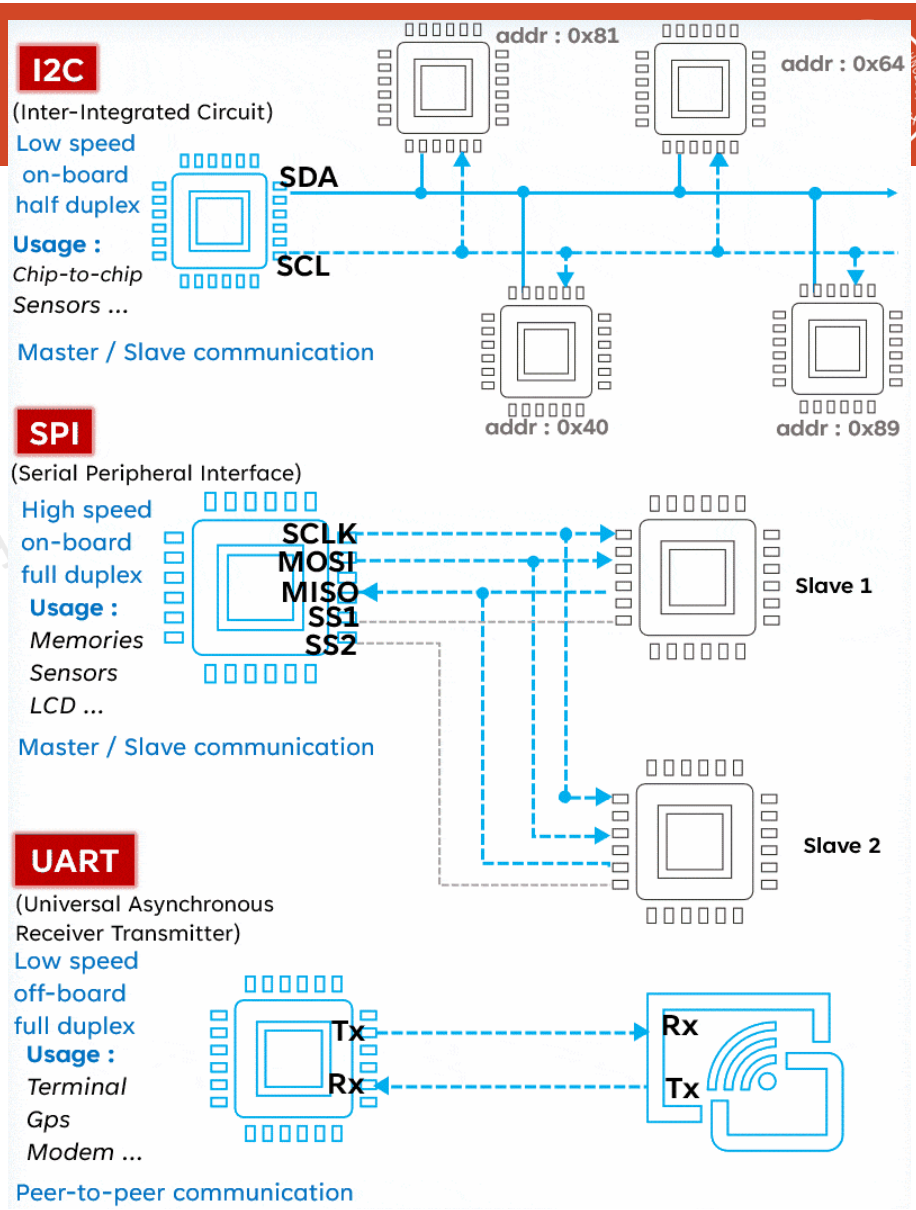
本课程中，暂不涉及SPI在WS63 SoC上的应用，后续内容不再讲授

5.5 通信方式总结

- [5.5.0] 三种板内通信对比

特性	UART	I2C	SPI
通信方式	全双工	半双工	全双工
引脚使用	2根 TX/RX	2根 SDA/SCL	4根 MISO/MOSISCK/CS
主从模式	无主从	多主机/多从机	单主机/多从机
通信速率	~100Kbps	~3.4Mbps	几十Mbps
设备数量	2	地址范围内	受CS引脚数量限制
协议复杂性	简单	较复杂	较简单
错误检测	校验位	无	无
设备地址	无	有	无
常见应用	设备调试、通信设备、卫星定位设备等	传感器、外扩存储、RTC、显示模块等	存储设备、音频设备、高速传感器等

在智能硬件系统中，没有最好，只有最适合



5.6 本章作业

- [5.6.0] 作业与思考

1. 智能硬件系统中板级通信的有何电气要求
2. 为何说UART是一种异步通信机制，发送方与接收方是靠什么机制来约定接收数据的一致性的
3. 针对UART的接收，来说明阻塞模式和非阻塞模式之间的差别
4. 为何说I2C是一种半双工通信，I2C通信系统中的Master是如何选择总线上Slave的
5. 简述I2C的Master从指定Slave中读取指定地址数据的流程
6. 截止目前内容，WS63唯一不需要设置引脚模式的外设是哪个
7. 简述UART通信的最基本参数，其中哪项是固定不可配置的
8. 对比UART、I2C和SPI的通信系统特性