AN4655
应用笔记

在 STM32 应用中虚拟增加串行通信外设数目

前言

应用工程师经常面临的问题是，微控制器在功能和性能上满足应用的所有其它要求，只是串行通信外设数量有限。有时，他们只能替换为具有充分数量通信外设的高级微控制器以规避此问题。这种移植可带来额外的（经常是用不到的）性能和功能，在多数情况下应用都不需要或用不到，但却增加了成本和 PCB 复杂度。

经常发生的情况是当并不是每个通道都需要全部（或特定）功能时，通信流及其控制可极大简化（例如仅在特定的模式或时隙需要通信、通信速度可以更低、并不是所有信号都严格需要正确的时间、可接受简化的协议或流程）。在这些特定情况下，如果有方法能用当前硬件补充缺失的信道，避免移植，则用户可真正受益。

另一方面，要在复用通道上与所有原始硬件特性完全兼容，代码和性能方面的代价太大，也很难实现。通常，会倾向于放弃一些特定需要，以达到简化复用外设通道概念的目的。这里，关键一点是识别那些应用中不需要的、可使用低工作量、降低性能的方法变更的通道。

本应用笔记提供了所述问题的基本概述，可帮助应用工程师在实现缺失的通信通道时，发现可能的替代方法。因为所讨论的外设存在于所有该类产品，所以它适用于所有 STM32 微控制器。

在下面的参考文档中有更多的信息和样例，说明了实际的用例和解决方案：

- TN0072，软件工具链和 STM32 特性；
- UM0892， STM32 ST-LINK 工具软件说明；
- AN4457， STM32F4 全双工 UART 模拟。

最后引用的文档与相同主题的其它应用笔记共同开发，用户可在 *www.st.com* 上查看。

目录

表格索引

图片索引

1 硬件方法

如果不能接受增加原始硬件外设 （例如，增加专用通信协处理器单元等外部组件），那么可使用下述基本方法补偿外设数量的限制：

1. 使用内部专用外设实例的重映射选项，更改不同端口处的输入和输出，以此建立微控制器与不同通道的临时连接 （用户可在数据手册的引脚说明中，针对每种产品及其实现的外设找到复用功能能力的信息）。
2. 使用复用逻辑，将多个通信通道复用为一个通信端口。
3. 使用 SWO 特性。

前两种方法提供了除专用时隙之外所选通道的服务，仅当它发生时不严格要求复用通道的同时操作和永久监测才可使用，例如：

- 服务通道；
- 偶尔与远程设备通信；
- 监测数值变化缓慢的传感器；
- 用于周期性备份的通道。

当从设备的活动等待主请求，或与从设备的通信可丢失，当主设备重新连至服务通道时可容易地重新同步时 （监听并通话），这些方法适合于主设备通信。当从设备和主设备之间提供额外控制信号来告知从设备连接建立时 （例如，当从设备能 / 不能在通道工作时的片选或通道就绪 / 繁忙信号），它也可用于从设备侧。

总之，当复用功能重新映射或复用到另一通道时，用户应注意与外设实例总线信号相关的 GPIO 处于非活动状态。正确设置可确保空闲条件的模拟，防止阻塞于当前未选择的或未用于通信的通道。可通过正确配置 GPIO 端口寄存器达到这一目的，增加外部组件 （上拉 / 下拉电阻）也有部分作用。请注意，当从端口移除 （重映射）复用功能时， GPIO 逻辑一般会覆盖相关引脚的控制，用户必须防止这些临时冻结通道上的任何总线虚假状态检测。

对于复用情况，如可用于产品，用户可使用路由接口专用组处的 I/O 开关专用控制。用户必须确保单个组内的适当开关组合 （配对）及时关闭，以令信号通过需要的路径。用户必须为这种开关指定额外的 IO，它们不能用于其它用途，而且还会有额外的 PCB 走线。

该原理示于*图 1* 和*图 2* 中。

图 1. 重映射选项原理

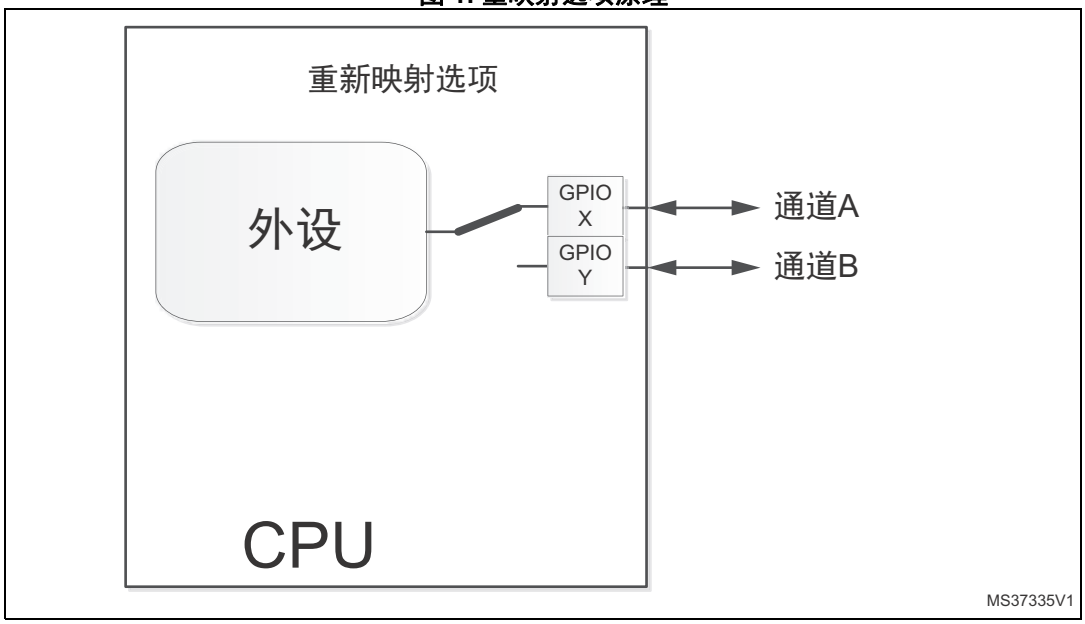
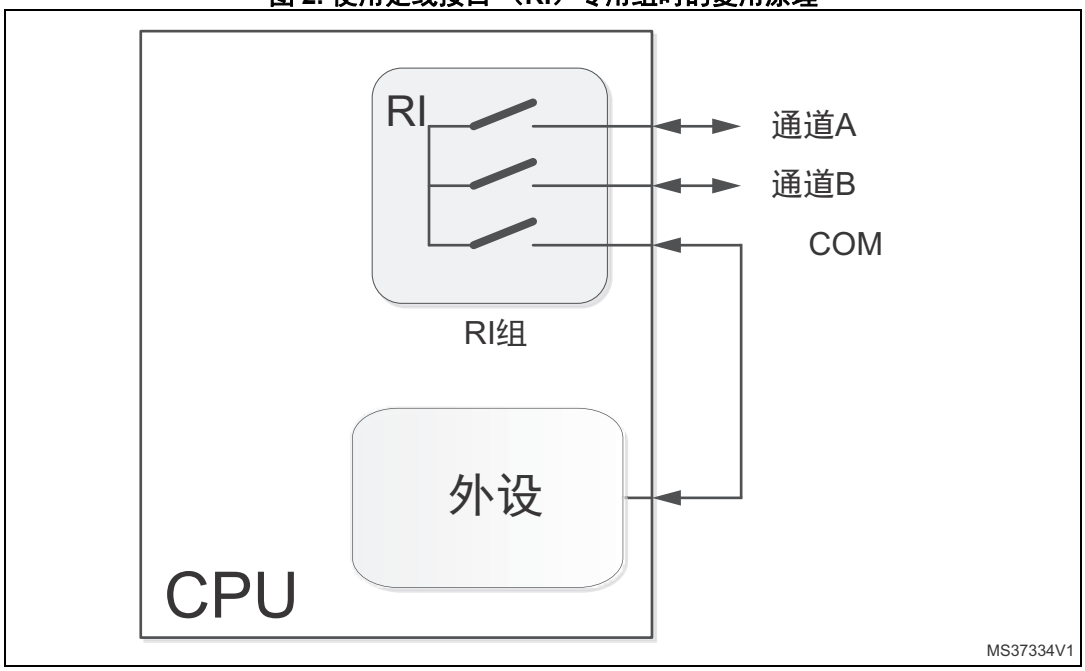


图 2. 使用走线接口 （RI）专用组时的复用原理



在调试期间，经常需要额外的通信端口，以传递应用专有的诊断信息和实时运行期间的系统监测。使用专用的指令跟踪宏小区 （ITM）和串行线浏览器 （SWV）接口期间，用户可在 SWO 引脚设置虚拟 COM 通信端口。

ITM 是应用驱动的跟踪源，支持 printf 类的调试以跟踪操作系统 (OS) 和应用程序事件，并通过 STLink 工具发布诊断系统信息。一些最新产品有 ITM 模块，可节省标准的通信外设用于应用目的。一些 IDE 和 ST HAL 固件支持它。要在接口发送一字节，在 *fputc()* 内调用 *ITM_SendChar()* 函数就足够了，通过分配 TRACE 引脚为异步跟踪模式可配置调试模块。

2 软件方法

2.1 兼容因子

当一个开发者开始考虑通过使用 GPIO 软件控制，模拟外设硬件接口功能将其替代时，他应该仔细考虑兼容因子及真正需要的等级。

2.1.1 功能性 （与时序无关）

这通常是由配置和状态寄存器控制的 （例如数据或协议格式、通信速率、奇偶校验、相关信号控制、 CRC 计算等）。在这里，通用性是一个关键问题。当需求仅为匹配应用需要时（固定时钟相位或极性、位的顺序和数量、简化流、无奇偶校验、不支持 CRC），开发可相当程度地简化。

当目标是用通用方法使用固件，尝试达到更大的灵活性和能力时，开发复杂度会极大增加，包括测试和验证阶段的要求。基本上说，在初始计划阶段，画一张带有所有状态转移及其条件的完整状态图是一个好的开始，因为它能帮助理解所有需求以及需要模拟的复杂度。

2.1.2 时序

这里，必须考虑信号的正确形状 （边沿、占空比）、时钟和数据访问间的时序 （例如数据建立或保持时间）以及正确的信号采样 （位置和采样数）。这些需求通常受限于缺少性能能力，以及当由软件实现时的任何快速 IO 控制的限制。

一些内核提供专用的内部总线用于 IO 控制，以执行非常快速的 IO 访问和驱动能力 （例如，Cortex®-M0+ 内核的专用 IOPORT）。

当由硬件做输出控制时 （例如输出比较特性），当提供 GPIO 信号时可取得很好的结果。用户可使用广泛的 STM32 架构能力来内部同步外设，但在此情况下，必须由中断服务内部的软件来监控模拟控制流逻辑，尤其是对于所有的负面影响和确定的时序限制。

2.1.3 HW 接口

关于特定的引脚需求 （例如真正的漏极开路驱动、边沿斜率），（通过设计）专用于某功能的那些引脚可具有其原始硬件支持特性。这意味着当在不支持这种原始功能的其它引脚模拟相同的功能时，几乎不可能复制这类特性。

至少，正确的 GPIO 配置 （速率限制、输出驱动）不会使问题更糟，只会尽可能地符合给定目的。

2.1.4 API 接口

构建于应用软件底层驱动结构的 API 接口能带来很好的灵活性，但它会增加系统性能负担，这是因为解决方案的复杂度会随着目标范围快速增长。用户通常定义 API 函数以配置一个专用外设、执行单个读写操作、打开或关闭通信会话。过程事件通常由回调处理，执行正确数据流或偶尔的错误状态。这里，又出现了通用性问题。当可以限制、移除或不支持一些功能时，能显著简化任务。此方法的最大优势为更好的代码阅读性，代码可保持很好的结构。缺点是时间和性能约束。

2.2 可用的方法

当用软件模拟通信外设时，可使用两个基本方法：

1. 位模拟；
2. 硬件和软件组合控制 GPIO。

2.2.1 位模拟

纯粹通过软件直接控制 IO 端口，模拟信号。此方法尤其是可用于同步协议 （SPI、 I2C），通常在这些协议中信号时序并不关键。 *SPI 位模拟序列*和 *I2C 主设备位模拟序列*指出了可能的解决方案。

2.2.2 硬件和软件组合控制 GPIO

通过使用专用硬件能力在需要的时间控制 IO，间接模拟信号。之后在软件级别实现通信逻辑，主要是从内部硬件事件驱动中断服务。用户可研究*基于硬件和软件组合控制 GPIO 的 UART 双工通道*中的原理，它示范了在实际情况下应用此方法。

用户还可在已经引用的 AN4457 中找到另一可能的解决方案。

本文详细说明了定时器和 DMA 之间的硬件同步：专用定时器的事件同步并触发专用数据缓冲和 GPIO 寄存器间的 DMA 传输序列。数据缓冲内容可为事先由软件储存在专用存储区的 GPIO 输出控制序列源，或者是从 GPIO 输入读取并储存的采样序列，在交互结束后分析。在通信交互期间的确切周期提供这些传输，双向同时的流可由独立的控制执行。

2.3 通用方法论

2.3.1 时钟信号方面，主从的对比

理论上，主设备模拟是更容易的情况。它能决定时隙流通信，可线下处理数据流 （例如，可预先准备 CRC 计算，之后对数据包检查），所以能以相对快速的方法处理物理数据流。

主设备提供同步时钟是最简单的情况。主设备启动流，处理时钟和数据信号，这里时序并不关键，除了从设备侧 （通常更慢）需考虑一些特定限制的情况 （例如建立或保持时间，通信超时）外，时序几乎是自由的。

异步主设备要求更高，因为性能不够会产生错误的效果，损毁数据流。在此情况下，主设备必须能够提供永远正确的数据输出和 / 或输入时序。

从设备模拟通常是最难的情况 （不管时钟信号是否同步），因为从设备必须：

- 永远为交互时隙做好准备；
- 与时隙起始保持良好同步 （尤其是异步时钟的情况）；
- 在时隙内保持正确的时序 （由时间计算或时钟信号改变得到）；
- 处理正确的数据流 （输入、输出，或者甚至是两者都有）。

2.3.2 中断和 DMA vs. 位模拟

从 GPIO、定时器或 DMA 事件得到的中断更容易处理，在固件级别更加系统化。若系统具有足够的性能容量来处理所有需要的软件过程，则要求的协议能很好地与数据流同步。用户能控制所有流，直接在中断服务内部或通过 API 提供的专用回调为事件服务。第二个方法给出了确定级别的通用性，保持软件结构良好。但此方法要求更宽地预留微控制器性能，以能实时充分管理所有通信事件。当专门处理双工数据流、支持更高通信速率时，这会是一个限制。

当使用 DMA 时，用户应考虑内部总线的容量，尤其是当并行处理额外的 DMA 进程、考虑它们的优先级时。每次 DMA 传输都会将相关的总线阻塞几个周期，当系统处理中断服务并返回时，或当执行请求总线通信的特定指令时， DMA 服务会出现显著的时延。

让我们想象一个特定的情况：信号边沿由外部 GPIO 中断处理。固件开发者必须计算：

- 中断服务时延 （几个 CPU 周期，还取决于其它待处理的中断及其优先级）；
- 进程进入中断服务 （储存上下文的时延）；
- 状态逻辑 （检测通信阶段所需的时间、软件执行的行动）；
- 专用于流的行动 （设置数据输出、采样数据输入、处理数据存储、更改时钟信号、设置下一事件的时序和专用控制、设置内部标志、在流的关键点调用并提供处理回调）；
- 保持及跟随流的必要硬件设置 （清除中断事件标志、启用禁用相关硬件事件）；
- 从中断服务返回 （恢复上下文）。

位模拟会快很多，但应用此控制时不容易保持任何适当的时序。它通常有软件开销，消耗 CPU 周期，这些周期本来可被其它进程利用。这对系统响应其它事件可能有显著影响，在一个硬实时系统中，可能会显著影响系统满足实时要求的能力。

若位模拟接口对实时性能没有负面影响，则必须用低优先级执行它，令它不能影响数据吞吐及时延。

当相同的代码在不同的起始地址放置并执行时，由于代码对齐，可能发生一个特别的问题。这会产生所提供信号的不同时序。当预先从存储器取得代码，或从管道清理代码时，不同的管道机制会导致这个问题。用户必须考虑指令预取设置，以及指令和数据缓存。

使用硬件接口和 DMA 传输，最小化软件负荷及 cpu 负载，可达到最有效的 CPU 传输。位模拟是它的另一个极端。它既不好移植，也不够灵活，不能处理所有外设的实例。产生的信号通常有更多的抖动及失真，尤其是当控制器还在执行其它任务时。这就是为什么尤其在实时关键环境下，要避免这种通信。

使用专门的宏和条件代码可显著增加代码的可读性，即使在这种底层编程时也能带来很高的灵活性 （参见代码样例）。

位模拟和中断组合使用，可以给出很好的自动及中断驱动位模拟代码，代价仅为使用一个专用定时器。但是总的来说，在高速、多任务环境下，很难做到可靠的位冲击。

2.4 软件模拟的外设方面

2.4.1 SPI

软件模拟 SPI 是最简单、最容易适配的，因为它是同步的，对速率、信号时序或物理接口特性也没有特别限制要求。

数据收发由两根独立的数据双向线 （MOSI、 MISO）同时提供，由主设备提供的通用时钟信号线同步。不实现寻址或回应控制。

关于 SCK 时钟信号，用户必须考虑两个通信节点间的固定时钟相位和极性，即使在简单网络内一组节点可使用不同的设置，若其通信独立，就必须考虑。对于从设备的选择信号，通常希望负极性。当通信是在一对节点间、主从角色固定时，通常不实现 SS 信号。也可以支持多主设备环境，但除了 SS 信号电平，没有特别的仲裁。当 SS 输入保持非激活时 SPI 节点可提供交互，否则会识别多主冲突。虽然全双工通信被认为是标准情况，很多解决方案使用简化的单工通信 （SPI 节点仅能发送或接收）。

时钟信号可为连续的或不连续的，支持无时钟信号拉伸。这尤其对于从设备施加了限制，因为不管任何数据在从设备侧就绪或接受，交互都可开始或继续。因此，从设备很容易进入数据下溢或上溢情况。

在节点间交互的比特数固定，但可任意配置。

可选的 API 可对于发送和接收集成单数据缓冲或更复杂的 FIFO 结构。

2.4.2 UART

软件模拟 UART 通道需要适当的时序控制，因为数据流是异步的。数据发送和接收是两个非常独立且非同步的进程，由两条独立的双向数据线 （Tx 和 Rx）提供，专由发送端驱动。

通常，任何同步或时序问题都不应在每个交互的比特率周期内有超过 30% 的时序错误。当字符帧内交互了 10 比特时，发送和接收间的内部时钟差值比不应超过 3%，才能正确处理最后交互的比特。定义的速率和比特协议可以配置。

物理接口多数由另外的通信标准定义，如 RS232/422/485。

调制解调器工作能力可由额外的状态信号支持，如 CTS/RTS。不实现寻址或回应控制。

字符帧部分可配，使用 NRZ 格式。帧的开始是一个起始比特，后面是 5-9 个数据比特，然后是可选的奇偶校验比特，最后是 1-2 个停止比特。若数据线保持在空闲的时间超过一个字符时间，则会检测到中断条件。位采样一般在比特周期的中间提供，在同一位中可包括若干采样。它会升高噪声、帧 （停止）或奇偶校验帧错误。接收端也会面临两种上溢错误。

可选的 API 可对于发送和接收集成单数据缓冲或更复杂的 FIFO 结构。

2.4.3 I2C

软件模拟 I2C 通道是最复杂的情况，因为必须考虑很多方面。

即使使用的是同步协议，根据所选的模式，需要特定的时序。这尤其适用于标准模式 （其速率更低，最高为 100 kbit/s），其中 SCL 时钟信号的低周期和高周期长度、在开始和停止条件期间数据 SDA 和时钟 SCL 信号变更之间的长度必须保持为约 5 µs 的最小值。

双向 SDA 和 SCL 信号的输出级应具有开漏或开集电极，以执行线与功能。标准允许用不同的供电电压连接源。

协议包括 10 位或 7 位寻址读写模式，总线仲裁支持多主通信、数据回应、时钟拉伸及时钟同步过程。当应用不需要支持这些需求时，模拟任务可极大简化。

若应用 API，则它主要基于更复杂的接口，能收集所有必要信息来处理协议、数据传输及发生的事件。错误设置的开始或停止条件 （总线错误）、回应失败及仲裁丢失 （多主系统）是 API 会通常发出的通信错误信号。当没有使用时钟拉伸时，从设备接收端也会面临数据上溢。

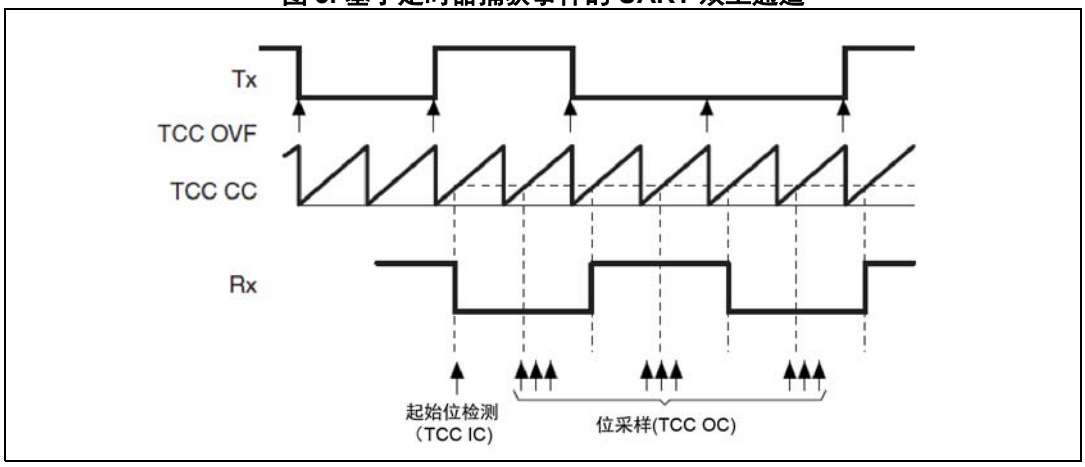
2.5 示例

2.5.1 基于硬件和软件组合控制 GPIO 的 UART 双工通道

本例显示了如何基于硬件事件，由软件控制发送和接收过程。

使用溢出及单捕获 & 比较定时器的 IO 捕获事件，控制 UART 输入采样以及 Rx 和 Tx 信号输出处理的模拟时序。相关的输入捕获引脚专用于接收数据。任何通用输出引脚或相关的输出比较引脚都可用于数据发送。通过设置定时器自动重载寄存器，将定时器溢出周期设为通道半比特率。*图 3* 说明了时序图。

图 3. 基于定时器捕获事件的 UART 双工通道



发送过程控制基于定时器溢出事件的常规中断。当用于发送的数据储存在专用变量之后 （软件模拟发送缓冲），会出现内部发送请求，使用最近的溢出中断开始新的发送 （发送开始的最大时延可为半比特时间）。这是因为没有正常同步的收发过程使用的是同一定时器控制。每偶数个定时器溢出事件中断都会控制专用输出引脚上发送信号的连续边沿改变，直到帧发送结束。奇数的溢出中断被丢弃。

接收过程使用同一定时器及其专用引脚的输入捕获和输出比较特性。一开始，在引脚处执行输入捕获。检测到第一个下降沿后，因为输入引脚功能切换至输出比较模式，所以捕获寄存器所捕获的值用于比较目的 （因为引脚仍停留在输入模式，所以任何输出 GPIO 能力不受影响）。由于定时器的半比特溢出周期，所以最近的输出比较事件指向第一个接收比特 （起始比特）中间。可通过在此时刻连续三次读取输入引脚电平模拟采样，对其中的每一个采样，若检测到低电平，则判断为正确接收了开始比特。然后继续接收过程，观察每隔一个的奇数输出比较事件。可执行同样的三点采样方法加上噪声检测逻辑，用于这里以及所有其它接收到的数据比特，直到当前接收帧结束。丢弃所有偶数比较中断。当采样到帧停止比特后， Rx 引脚切换回输入捕获模式，等待下一个帧开始条件。若正在采样停止比特时检测到噪声，则会导致帧错误。若在起始比特检测到噪声，则应终止接收过程，在等待下一个下降沿捕获时将 Rx 引脚切换回输入捕获模式。

用户可在此底层硬件抽象级别构建 API 接口。它可包括 UART 通道初始化、启用和禁用 Rx 和 Tx 通道、读写数据流 （例如控制数据缓冲）、检查交互状态。数据尺寸、停止比特的奇偶校验控制数目或其它控制可在预编译级别解决，方法是通过条件编译加速这些特性不被使用时的代码。

2.5.2 SPI 位模拟序列

这是一个控制 SPI 主设备全双工 8 比特数据帧交互的程序样例 （MSB 在前格式）。该程序可基于使用一系列宏来达到最高速度，即使它没有完全考虑好的通用软件实践来做结构化的代码整理 （例如，将重复的部分组织为循环或子程序）。

```
/**
 * @ 说明：SPI 主设备读写位模拟
 * @ 参数：unsigned char wb - 要发送的数据
 * @ 返回值：unsigned char rb – 接收的数据
 */
unsigned char SPI_master_8_bit_data_RW (unsigned char wb)
{
 unsigned char msk = 0x80;
 unsigned char rb = 0;
 do
 {
 #if CPHA=0
  miso_control(rb);
  mosi_control(wb);
  setup_dly();
  sck_odd_edge_control();
  half_bit_dly();
  sck_even_edge_control();
  setup_dly();
 #else /* CPHA=1 */
  sck_odd_edge_control();
  setup_dly();
  miso_control(rb);
  mosi_control(wb);
  setup_dly();
  sck_even_edge_control();
  half_bit_dly();
```

```
  #endif /* CPHA */
  }
  while ((msk>>=1) != 0);
  return(rb)
}
```

应用的条件代码排除了任何循环内的剩余测试，同时仍然保持了确定等级的通用性和可读性。若不能接受条件编译，则相对于使用集成的逻辑解决多种内部配置的单个通用程序，专用于具体设置组合的一组独立程序提供了更快的代码。

所有 SCK、 MOSI 和 MISO 控制信号都应该使用类似宏定义，在代码间插入或不使用时没有代码。如果通信速率限制是至关重要的，主要工作是防止这段代码用函数调用的方式来写。如果必要且主设备运行太快，则可加入定义 SCK 时延的宏，以考虑输入和输出建立时间或者时钟信号占空比波形。当提供位模拟逻辑的指令周期执行时间与数据建立及保持时间可比较或者比它更长时，它们一般可以保持为零。

当使用条件编译时，可如下定义相关的宏：

```
#define spi_set_mosi() { SPI_DR |= (1 << SPI_MOSI_PIN); }
#define spi_clear_mosi() { SPI_DR &= ~(1 << SPI_MOSI_PIN); }
#define spi_get_miso()  (SPI_DR & (1 << SPI_MISO_PIN))

#if CPOL=0
  #define sck_odd_edge_control() { SPI_DR |= (1 << SPI_SCK_PIN); }
  #define sck_even_edge_control() { SPI_DR &= ~(1 << SPI_SCK_PIN); }
#else /* CPOL=1 */
  #define sck_odd_edge_control() { SPI_DR &= ~ (1 << SPI_SCK_PIN); }
  #define sck_even_edge_control() { SPI_DR |= (1 << SPI_SCK_PIN); }
#endif /* CPOL */

#ifdef SPI_MOSI
  #define mosi_control(b) { (b & msk) ?spi_set_mosi() : spi_clear_mosi(); }
#else
  #define mosi_control(b) {} /* 未提供代码 */
#endif

#ifdef SPI_MISO
  #define miso_control(b) { if( spi_get_miso() != 0)  b |= msk;  }
#else
  #define miso_control(b) {} /* 未提供代码 */
#endif
```

```
#define setup_dly() {};/* 未提供代码 */
#define half_bit_dly() {};
```

2.5.3 I2C 主设备位模拟序列

本例为控制 I2C 主设备 8 比特数据帧读写交互的程序。

```
#define i2c_bus_init() { I2C_DR |= ((1 << I2C_SDA_PIN) | (1 << I2C_SCL_PIN)); }
#define i2c_set_sda() { I2C_DR |= (1 << I2C_SDA_PIN); }
#define i2c_clear_sda() { I2C_DR &= ~(1 << I2C_SDA_PIN); }
#define i2c_get_sda()  (I2C_DR & I2C_SDA_PIN)
#define i2c_set_scl() { I2C_DR |= (1 << I2C_SCL_PIN); }
#define i2c_clear_scl() { I2C_DR &= ~(1 << I2C_SCL_PIN); }GE

#define sda_wr_control(b) { (b & msk) ?i2c_set_sda() : i2c_clear_sda(); }
#define sda_rd_control(b) { I f(i2c_get_sda() != 0)  b |= msk;  }

/**
 * @ 说明：I2C 主设备写入 8 比特数据位模拟
 * @ 参数：unsigned char b - 要发送的数据
 * @ 返回值：unsigned char ack – 接收的回应
 */
void I2C_master_write (unsigned char b)
{
 unsigned char msk = 0x80;
 unsigned char ack;
 do
 {
  sda_wr_control(b);
  setup_dly();
  i2c_set_scl();
  half_bit_dly();
  i2c_clear_scl();
  setup_dly();
 }
 while ((msk>>=1) != 0);

 i2c_set_sda();/* ACK 时隙检查 */
 i2c_set_scl();
 half_bit_dly();
 ack = i2c_get_sda();
 i2c_clear_scl();
 return (ack);
}
```

```
/**
 * @ 说明：I2C 主设备读取 8 比特位模拟
 * @ 参数：unsigned char ack – 回应控制
 * @ 返回值：unsigned char b – 接收的数据
 */
unsigned char I2C_master_read (unsigned char ack)
{
 unsigned char msk = 0x80;
 unsigned char b = 0;
 do
 {
  i2c_set_scl();
  half_bit_dly();
  sda_rd_control(b);
  i2c_clear_scl();
  half_bit_dly();
 }
 while ((msk>>=1) != 0);

 if(ack != 0)
 {
  i2c_clear_sda();/* ACK 时隙控制 */
 }
 setup_dly();
 i2c_set_scl();
 half_bit_dly();
 i2c_clear_scl();
 half_bit_dly();
 return (b);
}

/**
 * @ 说明：I2C 开始
 * @ 参数：无
 * @ 返回值：无
 */
void I2C_Start_condition(void)
{
 i2c_bus_init();
 half_bit_dly();
 i2c_clear_sda();
 half_bit_dly();
 i2c_clear_scl()
```

```
  half_bit_dly();
}

/**
  * @ 说明：I2C 停止
  * @ 参数：无
  * @ 返回值：无
  */
void I2C_Stop_condition(void)
{
  i2c_clear_sda();
  i2c_clear_scl()
  half_bit_dly();
  i2c_set_scl();
  half_bit_dly();
  i2c_set_sda()
  half_bit_dly();
}
```

3 结论

对于 STM32 微控制器，使用本文所述的方法，理论上能够有效模拟通信外设。

若用户了解限制并考虑了合理的性能带宽，则他可受益于这一附加特性。若用户知道适合这种模拟的通道，且放弃所有不严格需要的需求，则可显著简化任务。

应仔细考虑这里的任何通用性，以防止解决方案从性能角度变得太复杂或繁重。

4 版本历史

表 1. 文档版本历史

日期	版本	变更
2015 年 3 月 5 日	1	初始版本

表 2. 中文文档版本历史

日期	版本	变更
2016 年 8 月 4 日	1	中文初始版本

重要通知 - 请仔细阅读

意法半导体公司及其子公司 （“ST”）保留随时对 ST 产品和 / 或本文档进行变更、更正、增强、修改和改进的权利，恕不另行通知。买方在订货之前应获取关于 ST 产品的最新信息。 ST 产品的销售依照订单确认时的相关 ST 销售条款。

买方自行负责对 ST 产品的选择和使用， ST 概不承担与应用协助或买方产品设计相关的任何责任。

ST 不对任何知识产权进行任何明示或默示的授权或许可。

转售的 ST 产品如有不同于此处提供的信息的规定，将导致 ST 针对该产品授予的任何保证失效。

ST 和 ST 徽标是 ST 的商标。所有其他产品或服务名称均为其各自所有者的财产。

本文档中的信息取代本文档所有早期版本中提供的信息。