PSX/PS2 手柄与单片机间的通信

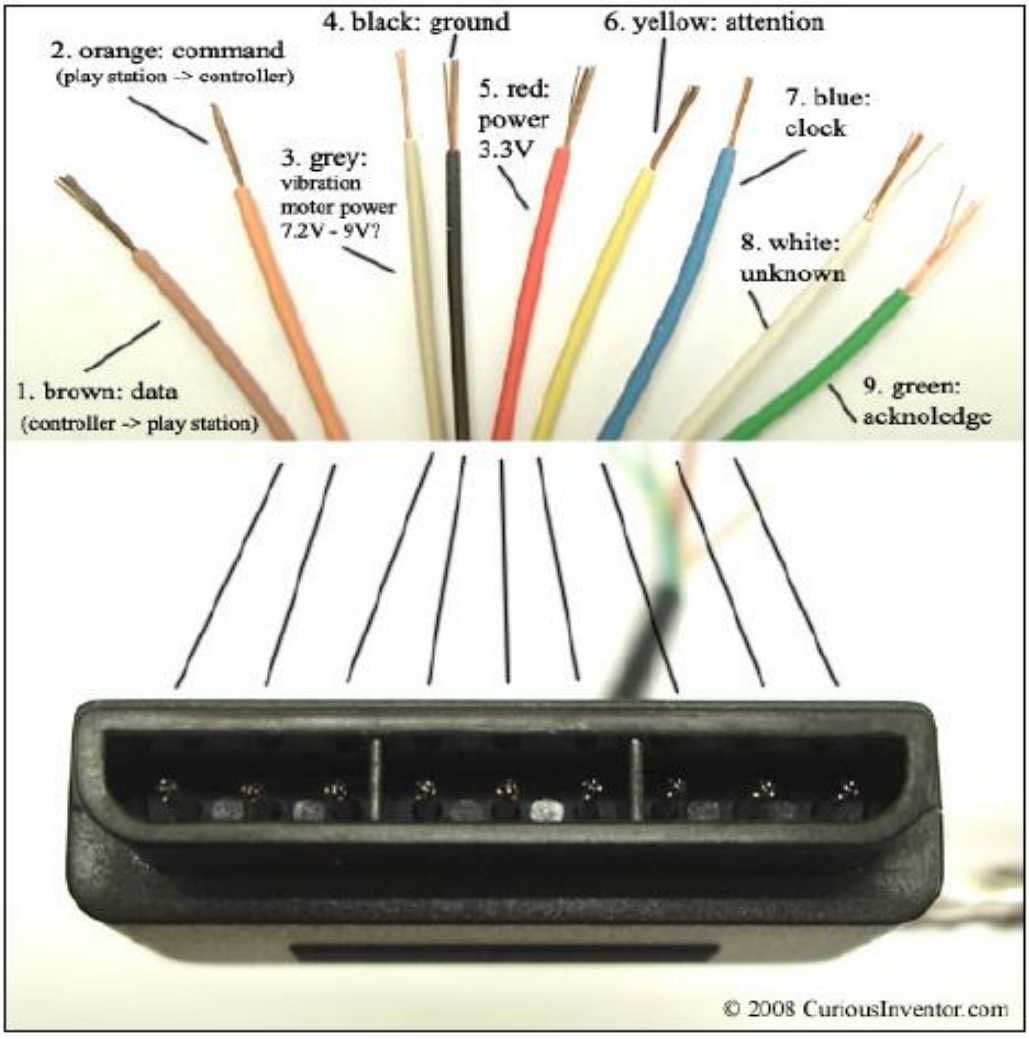
摘要：数字摇杆手柄使用 SPI 串行通信与上位机进行通讯（在通讯过程中的角色为：下位机），而 AVR 单片机集成了 SPI 串行总线。这可以很方便地使用单片机的硬件资源实现单片机与数据摇杆手柄之间的通讯，而非使用软件模拟来实现单片机读取数字摇杆手柄。这不仅提高了程序运行的效率、运行的稳定，并且也使得程序更简短。

关键字：PSX PS2 数字摇杆手柄 SPI 串行通讯 AVR 单片机

DAT-P00 CMD-P01 MVCC-PP GND-PP 3v3-PP ATT-P02 CLK-P03 none-PP ACK-P04

引言

对于手动机器人首选的人机界面之一就是数字摇杆手柄，作为游戏专用的手柄，它具备了舒适度高、灵敏度高等等许多的特质，因此颇受青睐。本文主要阐述数字摇杆手柄所采用的通讯协议、SPI 串行接口，并实现单片机与数字摇杆手柄之间的通讯。

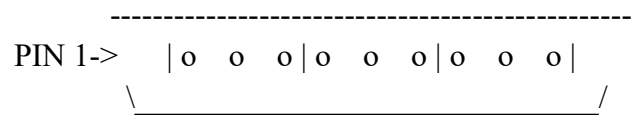


注：PSX/PS2 游戏手柄引脚定义

第一部分：PSX/PS2 通讯协议

一、各引脚定义：（引脚号/定义/用途)

①PS 手柄针脚输出(面对插头)



DATA

信号流向从手柄到主机。

此信号是一个 8 bit 的串行数据，同步传送于时钟下降沿(输入输出信号在时钟信号由高到低时变化，所有信号的读取在时钟前沿到电平变化之前完成。)

COMMAND

信号流向从主机到手柄。

此信号和 DATA 相对，同样是一个 8 bit 的串行数据，同步传送于时钟下降沿。

VCC

电源电压从 5V 到 3V 原装的索尼手柄都可以工作。

主机主板上装有表面安装的 750mA 保险丝 ，用于防止外设过载(750mA 是包括左右手柄和记忆卡)。

ATT

ATT 用于提供手柄触发信号。

信号在通信期间处于低电平。又有人将此针脚叫做 Select, DTR 和 Command。

CLOCK

信号流向从主机到手柄。

用于保持数据同步。

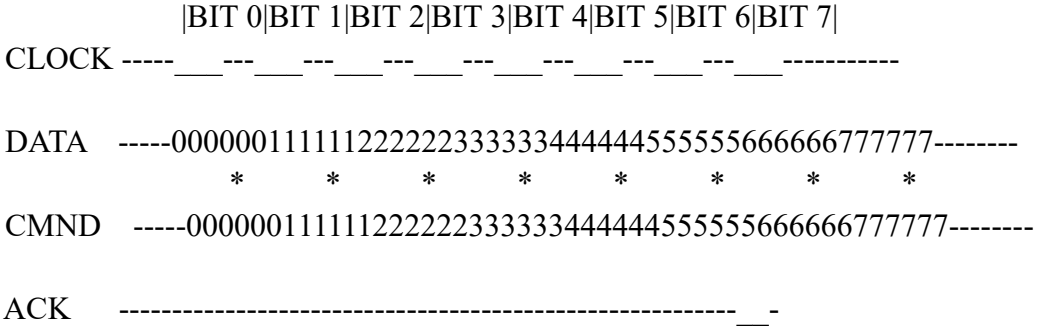
ACK

从手柄到主机的应答信号。

此信号在每个 8 bits 数据发送之后的最后一个时钟周期变低，并且 ATT 一直保低电平。如果 ACK 信号不变低约 60 微秒 PS 主机会试另一个外设。

②PS 手柄信号

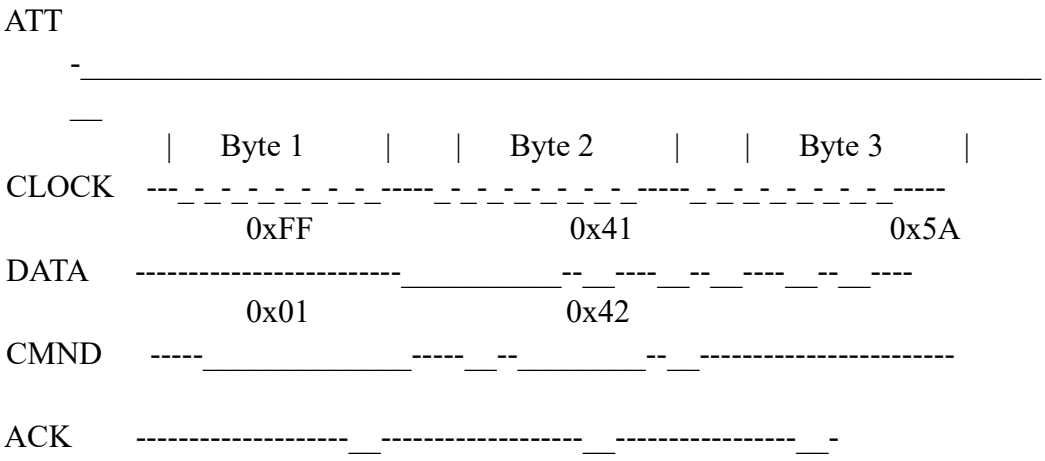
所有通讯都是 8 bit 串行数据最低有效位先行。在 PS 手柄总线的所有时码在时钟下降沿都是同步的。传送一个字节的情况如下所示:



注：

数据线的逻辑电平在时钟下降沿驱动下触发改变。数据的接收读取在时钟的前沿（在记号*处）到电平变化之前完成。 在被选手柄接收每个 COMMAND 信号之后，手柄需拉低 ACK 电平在最后一个时钟。如果被选手柄没 ACK 应答主机将假定没手柄接入。 当 PS 主机想读一个手柄的数据时，将会拉低 ATT 线电平并发出一个开始命令 (0x01)。手柄将会回复它的 ID (0x41=数字, 0x23=NegCon, 0x73=模拟红灯, 0x53=模拟绿灯). 在手柄发送 ID 字节的同时主机将传送 0x42 请求数据。随后命令线将空闲和手柄送出 0x5A 意思说：“数据来了”。

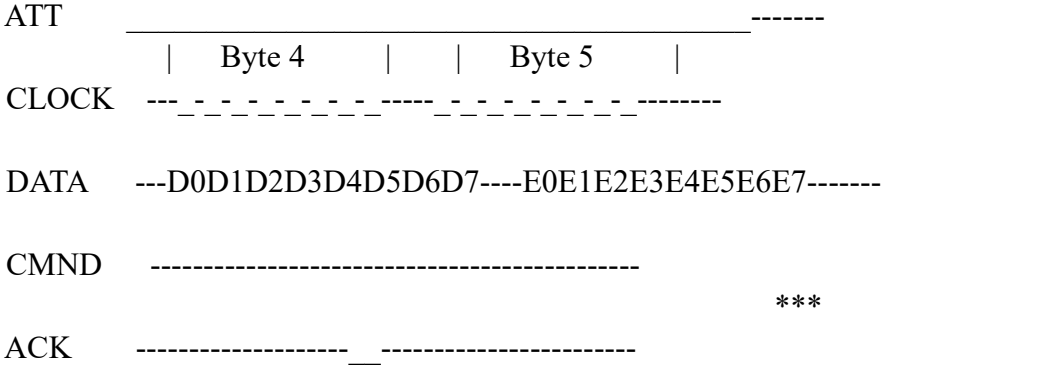
③时钟信号



注：

在手柄执行初始化命令之后将发送它所有的数据字节（数字手柄只有两个字节）。在最后字节发送之后使 ATT 高电平，手柄无需 ACK 应答。

数字手柄的数据传送如下所示(这里 A0,A1,A2...B6,B7 是两个字节的数据比特)。



注意: 没 ACK.

二、主要通信协议：

当主机想读取手柄的数据时，将会拉低 ATT 电平，并发出一个开始命令 0x01。手柄将会回复它的 ID：0x41（数字）、0x73（模拟红灯）等等。在手柄发送 ID 的同时，主机将发送 0x42 命令请求数据，随后将命令线空闲并等待手柄发

送“准备好了”0x5A 的命令。以上是主机读取手柄键码数据的初始化阶段，接着手柄就会发送它所有的数据字节（数字手柄只有两个字节）。在最后字节发送后将 ATT 电平拉高，手柄无须 ACK 应答。

PSX 手柄数据(下面显示手柄的实际发送字节)：

①标准数字手柄

Standard Digital Pad

BYTE	CMND	DATA							
01	0x01	idle							
02	0x42	0x41							
03	idle	0x5A	Bit0	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7
04	idle	data	SLCT			STRT	UP	RGHT	DOWN	LEFT
05	idle	data	L2	R2	L1	R1	/\	O	X	_|

All Buttons active low.

*所有按键按下有效

②NegCon

NegCon

BYTE	CMND	DATA							
01	0x01	idle							
02	0x42	0x23							
03	idle	0x5A	Bit0	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7
04	idle	data				STRT	UP	RGHT	DOWN	LEFT
05	idle	data				R1	A	B		
06	idle	data	Steering		0x00 = Right			0xFF = Left		
07	idle	data	I	Button	0x00 = Out			0xFF = In		
08	idle	data	II	Button	0x00 = Out			0xFF = In		
09	idle	data	L1	Button	0x00 = Out			0xFF = In		

All Buttons active low.

*所有按键按下有效

③模拟手柄绿灯模式

Analogue Controller in Green Mode

BYTE	CMND	DATA							
01	0x01	idle							
02	0x42	0x53							
03	idle	0x5A	Bit0	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7
04	idle	data				STRT	UP	RGHT	DOWN	LEFT
05	idle	data	L2	L1	_|	/\	R1	O	X	R2
06	idle	data	Right Joy 0x00 = Left					0xFF = Right		
07	idle	data	Right Joy 0x00 = Up					0xFF = Down		
08	idle	data	Left Joy 0x00 = Left					0xFF = Right		
09	idle	data	Left Joy 0x00 = Up					0xFF = Down		

All Buttons active low.

*所有按键按下有效

④模拟手柄红灯模式（此为本文所使用模式）

Analogue Controller in Red Mode

BYTE	CMND	DATA							
01	0x01	idle							
02	0x42	0x73							
03	idle	0x5A	Bit0	Bit1	Bit2	Bit3	Bit4	Bit5	Bit6	Bit7
04	idle	data	SLCT	JOYR	JOYL	STRT	UP	RGHT	DOWN	LEFT
05	idle	data	L2	R2	L1	R1	/\	O	X	_|
06	idle	data	Right Joy 0x00 = Left					0xFF = Right		
07	idle	data	Right Joy 0x00 = Up					0xFF = Down		
08	idle	data	Left Joy 0x00 = Left					0xFF = Right		
09	idle	data	Left Joy 0x00 = Up					0xFF = Down		

All Buttons active low.

*所有按键按下有效

⑤PSX 鼠标

```
PSX Mouse
(credit to T.Fujita
 http://www.keisei.tsukuba.ac.jp/~kashima/games/ps-e.txt)

BYTE    CMND    DATA

 01     0x01    idle
 02     0x42    0x12
 03     idle    0x5A    Bit0 Bit1 Bit2 Bit3 Bit4 Bit5 Bit6 Bit7
 04     idle    0xFF
 05     idle    data         L    R
 06     idle    data    Delta Vertical
 07     idle    data    Delta Horizontal

All Buttons active low.
```

*所有按键按下有效。

三、用单片机模拟主机向手柄发送 CMND 并读取手柄的数据：

我们可以利用单片机的可位寻址的 I/O 口分别作为 DATA、CMND、CLCK、ATT 与手柄进行通信，我们只先将 ATT 电平拉低，然后将要发送的命令低位在前放到 I/O 口的锁存器，然后控制 CLCK 制造同步下降沿信号，就可以完成命令数据串行传送了。数据的接受亦同理。

程序：

```
#include <mega16.h>

unsigned char ucDat[9];          //存放遥杆键值
#define START_CMD           0x01
#define ASK_DAT_CMD             0x42

//遥杆 IO 口初始化程序
void PSXControlInit(void){
 DDRC.1 = 1;     //ATT
 PORTC.1 = 1;
 DDRC.0 = 1;     //CMD
 PORTC.0 = 1;
 DDRD.7 = 1;         //CLOCK
 PORTD.7 = 1;
 DDRD.6    = 0;       //DATA
 PORTD.6 = 1;
}

//遥杆字节通信程序  *ucCMD:  命令字
                        //*返回值:读回的数据
unsigned char ReadWriteControlByte(unsigned char ucCMD)
{
```

```
 unsigned char ucReDat = 0,ucTmp,i;
 ucTmp = ucCMD;
 for(i = 0;i < 8;i++){
       ucReDat >>= 1;
       PORTD.7 = 1;
       PORTD.7 = 0;
       if(ucTmp & 0x01){
            PORTC.0 = 1;
       }else{
            PORTC.0 = 0;
       }
       PORTD.7 = 1;
             if(PIND.6){
            ucReDat |= 0x80;
       }
       ucTmp      >>= 1;
 }
 return ucReDat;
}

//读取遥杆键值
void ReadWriteControl(void){
 unsigned char ucCount = 0;
 PORTC.1 = 0;
 ucDat[ucCount++] = ReadWriteControlByte(START_CMD);
 ucDat[ucCount++] = ReadWriteControlByte(ASK_DAT_CMD);
 ucDat[ucCount++] = ReadWriteControlByte(0);
 ucDat[ucCount++] = ReadWriteControlByte(0);
 ucDat[ucCount++] = ReadWriteControlByte(0);
 ucDat[ucCount++] = ReadWriteControlByte(0);
 ucDat[ucCount++] = ReadWriteControlByte(0);
 ucDat[ucCount++] = ReadWriteControlByte(0);
 ucDat[ucCount++] = ReadWriteControlByte(0);
 PORTC.1 = 1;
}

void main(void)
{
PSXControlInit();
while (1)
          {
             ReadWriteControl();
          };
}
```

第二部分：串行 SPI 接口

一、SPI 串行总线介绍

①SPI 总线的组成

串行外设接口 SPI 是由 Freescale 公司提出的一种采用串行同步方式的 3 线或 4 线通信接口，使用信号有使能信号、同步时钟、同步数据输入和输出。典型的 SPI 总线系统包括一个主机和一个从机，双方通过 4 根信号线相连，分别是：

ⅰ主机输出/从机输入（MOSI）。主机的数据传入从机的通道。

ⅱ主机输入/从机输出（MISO）。从机的数据传入主机的通道。

ⅲ同步时钟信号（SCLK）。同步时钟是由 SPI 主机产生的，并通过该信号线传送给从机。主机与从机之间的数据接收和发送都以该同步时钟信号为基准进行。

ⅳ从机选择（$\overline{SS}$）。该信号由主机发出，从机只有在该信号有效时才响应 SCLK 上的时钟信号，参与通信。主机通过这一信号控制通信的起始和结束。

②SPI 通信地特点

SPI 通信过程是一个串行移位的过程。当主机需要发起一次传输时，它首先拉低$\overline{SS}$，然后在内部产生的 SCLK 时钟作用下，将 SPI 数据寄存器的内容逐位移出，并通过 MOSI 信号线传送至从机。而在从机一侧，一旦检测到$\overline{SS}$有效之后，在主机的 SCLK 时钟作用下，也将自己寄存器中的内容通过 MISO 信号线逐位移入主机寄存器中。当移位进行到双方寄存器内容交换完毕时，一次通信完成。如果没有其他数据需要传输，则主机便抬高$\overline{SS}$，停止 SCLK 时钟，结束 SPI 通信。

可以得出 SPI 通信的特点如下：

★主机控制具有完全的主导地位

★SPI 通信时一种全双工高速的通信方式

★SPI 的传输始终是在主机控制之下，进行双向同步的数据交换

二、AVR 的 SPI 接口原理

①SPI 接口的主要特征

★全双工、3 线同步数据传输

★可选择的主/从操作模式

★数据传送时，可选择 LSB（低位在前）方式或 MSB（高位在前）方式

★7 种可编程的位传送速率

★数据传送结束中断标志

★写冲突标志保护

★从闲置模式下被唤醒（从机模式下）

★倍速（CK/2）SPI 传送（主机模式下）

②SPI 接口硬件

⑴数据寄存器

SPI 接口的核心是一个 8 位移位寄存器，这个寄存器在时钟信号的作用下，实现数据从低位移入，高位移出。一旦程序将需要发送的字节写入该寄存器后，硬件就自动开始一次 SPI 通信的过程。通信结束之后，该寄存器中的内容就被更新为收到的从机串出的字节，供程序读取。写入发送的字节的操作需要在一字节传输结束之后进行。

⑵时钟逻辑

★当配置为 SPI 主机时，时钟信号由内部分频器对系统时钟分频产生。这个时钟信号一方面被引入到移位寄存器，作为本机的移位时钟；另一方面还被输出到 SCK 引脚，以提供给从机使用。

★当配备为 SPI 从机时，时钟信号由 SCK 引脚引入到移位寄存器，与内部时钟无关，此时 SPI 时钟配置位无效。

⑶引脚逻辑

SPI 引脚方向定义：

引脚	主机方式	从机方式
MOSI(PB5)	用户定义	输入
MISO(PB6)	输入	用户定义
SCK(PB7)	用户定义	输入
$\overline{SS}$(PB4)	用户定义	输入

注：对输入引脚应通过设置相应位使能内部的上拉电阻，以节省总线上外接的上拉电阻。

⑷控制逻辑

控制逻辑单元主要完成以下功能：

★SPI 接口各参数的设定，包括主、从模式、通信速率、数据格式等

★传输过程的控制

★SPI 状态标志，包括中断标志(SPIF)的设置、写冲突标志（WCOL）的置位等

三、SPI 接口的设计应用要点

①初始化

★正确选择 SPI 的主/从机方式

★正确设置通信参数

★正确设置数据串出的顺序

②$\overline{SS}$引脚的处理

★在主机模式下，$\overline{SS}$引脚方向的设置（PB4）会影响 SPI 接口的工作方式，尽量设置成输出方式

★在 SPI（主机模式）操作过程中，$\overline{SS}$并不会自动产生任何的控制信号，所有需要从$\overline{SS}$输出地控制信号均需通过用户程序来完成

③总线竞争的处理

产生总线竞争是当 SPI 总线上存在多主机情况下产生的，处理总线竞争不仅需要硬件具备相应的功能，同时在 SPI 中断程序中也需要包含对总线竞争的处理过程。为了防止总线冲突，本机的 SPI 接口将自动产生以下操作：

★清除 SPCR 寄存器的 MSTR（主机选择）位，将自己设置为从机。MOSI 和 SCK 引脚自动设为输入

★SPSR 寄存器的 SPIF 置位，申请中断

注：在不需要处理总线竞争的简单 SPI 系统中，位保证本机作为 SPI 主机正常工作，应将$\overline{SS}$设置为输出。如果将$\overline{SS}$设置为输入，则应保证该引脚始终为高电平。

四、SPI 接口与手柄之间通信的软件设计

①SPI 的设置：

1. SPI Mode：Mode3
2. SPI Clock Rate：115.200kHz（Fosc 为 7.372800MHz）
3. SPI Type：Master
4. Data Order：LSB First

② 程序：

```
#include <mega16.h>
#include <spi.h>
#define SPIF                    7
#define START_CMD               0x01
#define ASK_DAT_CMD             0x42

unsigned char ucDat[9];

char spi_transfer(volatile char data)
{
  SPDR = data;
  while (!(SPSR & (1 << SPIF)))          //等待完成
  {};
  return SPDR;
}

void ReadWriteControl(void)
{
    PORTB.4 = 0;
    ucDat[0] = spi_transfer(START_CMD);          //无用数据
    ucDat[1] = spi_transfer(ASK_DAT_CMD);        //ID
    ucDat[2] = spi_transfer(0);                  //准备完成标志
    ucDat[3] = spi_transfer(0);                  //第四帧、第五帧为数字模式（按
```

```
    ucDat[4] = spi_transfer(0);     //键）及模拟红灯模式下（按键及摇杆）的数据
    ucDat[5] = spi_transfer(0);     //第六帧、第七帧为右摇杆的数据
    ucDat[6] = spi_transfer(0);
    ucDat[7] = spi_transfer(0);     //第八帧、第九帧为左摇杆的数据
    ucDat[8] = spi_transfer(0);
    PORTB.4 = 1;
}

void spi_init(void)
{
    unsigned char temp;
    DDRB.4 = 1;     //ATT
    PORTB.4 = 1;
    DDRB.5 = 1;     //CMD
    PORTB.5 = 1;
    DDRB.7 = 1;         //CLOCK
    PORTB.7 = 1;
    DDRB.6    = 0;       //DATA
    PORTB.6 = 1;
    SPCR=0x7E;
    SPSR=0x00;
    temp = SPSR;
    temp = SPDR;
}

void main(void)
{
    #asm("cli")
    spi_init();
    #asm("sei")

    while (1)
            {
                ReadWriteControl();
            };
}
```

总结：

实验结果表明，时钟频率的设置时影响通信结果的主要因数之一。时钟频率设置不同，也能够实现单片机对手柄数据的读取，但影响读取数据的正确率。经过实验，时钟频率设置在 115KHz 左右时读取手柄数据是最稳定。过高和过低都会影响数据的读取，因此在编写程序时要注意时钟频率的设置。读取的数据分析如下：

总共接收 9 帧 8 位数据帧。第一帧始终为 0xFF；第二帧为手柄回复的 ID 号，0x41 为数字模式，0x73 为模拟红灯模式；第三帧为手柄准备完毕信号，始终为 0x5A；第四帧、第五帧为数字模式（按键）及模拟红灯模式下（按键及摇杆）的数据，无按键按下时都始终为 0xFF；第六帧、第七帧为右摇杆的数据，无摇杆按下，数据都始终为 0x80；第八帧、第九帧为左摇杆的数据，无摇杆按下，数据都始终为 0x80。（注：摇杆在数字模式下影响第四、第五帧数据帧，而不影响第六至第九帧数据帧，功能相当于按键；而在模拟红灯模式下不影响第四、第五帧数据帧，影响第六至第九帧数据帧）以下为各按键或摇杆的值：

第四帧：

SELECT:0xFE

START: 0xF7

UP:0xEF

DOWN:0xBF

LEFT:0x7F

RIGHT:0xDF

第五帧：

L1:0xFB

L2:0xFE

R1:0xF7

R2:0xFD

△:0xEF

×:0xBF

□:7F

○:DF

第六帧：

JOYR LEFT:0x00

JOYR RIGHT:0xFF

第七帧：

JOYR UP:0x00

JOYR DOWN:0xFF

第八帧：

JOYL LEFT:0x00

JOYL RIGHT:0xFF

第九帧：

JOYL UP:0x00

JOYL DOWN:0xFF

注：摇杆在所有方向都有相应的值，这里只是列其一二