Hardware SPI (HSPI) Sending/Receiving Data Explained

esp8266.com/viewtopic.php

Hi ESP8266 Forum Members. I've dumped another lot of information I've gathered on the hardware SPI control on my blog:

http://d.av.id.au/blog/hardware-spi-hspi-command-data-registers/

I'll only be maintaining and editing the information on my blog, but I've copied it below for reference (and archive!).

The hardware SPI can be a bit unintuitive at first without knowing how it works. The Espressif example code really only shows how to send 8 and 9 bit packets when it can do much more than that. Also, I haven't seen anything on how to receive and store data, which I've covered below 😁

Enjoy, share, comment, and improve!

The following is what I've learnt from playing around with the ESP8266 SPI registers. This post is mainly going to be an information dump, and I'll write up something a bit clearer and easier to follow for beginners later. Hopefully this helps those wanting to use the hardware SPI functions of the ESP8266.

If you haven't already, have a look at the spi.c and spi_register.h files included with the Espressif SDK examples. Being familiar with the standard functions will help you understand where the following information is applied. The code examples given below are simply modifications or additions to the existing code.

There are three main registers that control the hardware SPI settings in terms of data that is sent and received.

SPI_USER - Controls which functions of the SPI controller will be used. Essentially used to toggle settings on and off.

SPI_USER1 - Defines Address, Data Out (MOSI), Data In (MISO), and Dummy Data lengths.

SPI_USER2 - Defines Command data length, and is also used to store the actual command data.

SPI_USER

This is the register you'll want to setup first. The register is a 32bit value like the others. You can see the full list of individual bit definitions in the spi_register.h file. I'll explain what each bit does in a second. To configure the register, you would use the following commands:

SET_PERI_REG_MASK(SPI_USER(spi_no), SPI_CS_SETUP|SPI_CS_HOLD|SPI_USR_COMMAND|SPI_USR_ADDR|SPI_USR_MOSI);

CLEAR_PERI_REG_MASK(SPI_USER(spi_no), SPI_FLASH_MODE|SPI_USR_MISO);

You need to SET all the functions you want to use by adding them to the second parameter using | (Logical OR). Above, I wanted CS Setup, CS Hold, Command data, Address data, and Output Data (MOSI).

You should CLEAR at least SPI_FLASH_MODE (this is included in the example code). It is good practice to CLEAR unneeded settings like SPI_USR_MISO just in case they have be SET by previous code.

SPI_CS_SETUP: Enabling this ensures that your chip select (CS) line is pulled low a couple of cpu cycles before the SPI clock starts, giving your SPI slave device some time to get ready if required. I don't see any harm in having this on by default.

SPI_CS_HOLD: As above, except it holds the CS line low for a few cpu cycles after the SPI clock stops. Again, good idea to enable this by default unless your SPI slave devices has specific requirements about when the CS line goes high.

SPI_USR_COMMAND | SPI_USR_ADDR | SPI_USR_DUMMY | SPI_USR_MISO | SPI_USR_MOSI:

Ok, the data clocked out during one entire SPI cycle is defined by these options. If a particular option is not set, the data for it is simply skipped and not clocked out. Just enable the parts you need depending on your application. If all options are enabled, the data is clocked out in the following order:

(NOTE: I haven't yet tested the DUMMY option, but I'm guessing it may come between MOSI and MISO data?)

[COMMAND DATA]-[ADDRess DATA]-[MOSI DATA]-[MISO DATA]

I have my code setup for an SPI EEPROM which uses COMMAND + ADDR + MOSI for writing data, and COMMAND + ADDR + MISO for reading in data.

For example, a write would be:

COMMAND = 0b101 (3 bits)

ADDRess = 0x14F (9 bits)

MOSI data = 0xAB (8 bits)

With the SPI_USER register setup like I showed before, I would get [101][101001111][10101101] on the SPI MOSI line.

If you just want to send 'x' bits of data and handle assembling the data packet yourself, you can just use the MOSI option without the command or address registers. However, this is a much cleaner way and advantageous when writing multiple data bytes in a row.

MISO DATA: During this period, the MOSI line simply outputs 0, while all the data on the MISO line is clocked in to the internal SPI_Wx registers. It is important to note that data in is only collected during this period. Communication is half-duplex. For an SPI EEPROM, I simply use SPI_USR_MISO instead of SPI_USR_MOSI. The COMMAND register holds the read command (100), and the ADDR register holds the address to be read.

When SPI_USR_MOSI is enabled, output data is sourced starting from the SPI_W0 register first, followed by W1, W2 etc. (more on controlling number of bytes sent later).

When SPI_USER_MISO is enabled, input data is stored into the SPI_W0 register first, followed by W1, W2 etc.

Since the MISO period comes after the MOSI output, you could use the entire 16x32bit space for output data, and then overwrite it all with the input data. However, if you want to keep the output and input data intact, you can split the SPI_Wx registers into two groups of 8.

SPI_USR_MOSI_HIGHPART: enabling this means all MOSI output data is sent from SPI_W8 through SPI_W15.

SPI_USR_MISO_HIGHPART: enabling this means all MISO input data is stored from SPI_W8 through SPI_W15.

You would only enable one of these to split the storage registers between input and output data.

SPI_USR_DUMMY: I assume this inserts 'x' bits of dummy 0's into the SPI output stream. I haven't had a chance to test this yet, so not sure exactly where the dummy bits get inserted. Could be useful for some SPI devices.

SPI_WR_BYTE_ORDER: By default, it sends out 8 bit (1 byte) chunks of the 32bit SPI_Wx registers at a time. By default, it will send the lowest byte first. Use this to swap the order from highest to low. Only affects the MOSI option.

For example, SPI_W0(HSPI) = 0xFEEDBEEF. This would clock out as 0xEFBEEDFE unless this byte order setting was set.

SPI_RD_BYTE_ORDER: Same as above, but for storing incoming data. Only affects the MISO option.

That's about it for SPI_USER register.

SPI_USER1

This is simply where you define how many bits long your MOSI, MISO, and ADDRess data are (also # of DUMMY cycles).

SPI_USR_ADDR_BITLEN: maximum value of 0x1F or 31 (although the spi_register.h file suggests 0x3F, the register that holds the address is only 32bits). The actual number of bits in your address is this value + 1. Set it to 7 if you have an 8bit address, 15 for a 16bit address etc.

SPI_USR_MOSI_BITLEN: maximum value of 0x1FF or 511. Again, this is one less than the actual number of bits. Maximum of 512bits of data is conveniently 16x 32bit registers (SPI_W0 to SPI_W15). 😀

SPI_USR_MISO_BITLEN: Exactly as per above, but for MISO input data length.

SPI_USR_DUMMY_CYCLELEN: Max value of 0xFF. One less than the number of dummy cycles needed. Max number of dummy cycles is 256.

Example code to setup lengths:

```
WRITE_PERI_REG(SPI_USER1(spi_no), ((7&SPI_USR_MOSI_BITLEN)<<SPI_USR_MOSI_BITLEN_S)| //8bits of
data out
((7&SPI_USR_MISO_BITLEN)<<SPI_USR_MISO_BITLEN_S)| //8bits of data in
((8&SPI_USR_ADDR_BITLEN)<<SPI_USR_ADDR_BITLEN_S)); //address is 9 bits A0-A8
```

If you don't turn on ADDR, MISO, or MOSI in SPI_USER, then these lengths are ignored and have no effect.

SPI_ADDR

This is the register where you store the address data you want to send out as part of the SPI data packet. Data is clocked out from the Most Significant Bits. If you set your address length to say, 9 bits, then only the 9 MSB bits are send!

SPI_ADDR = 0b00000000000000000000000123456789 where 123456789 is the 9 bit address.

Using this will not work, as it will simply clock out the 9 leftmost 0's instead! You need to shift your 9bit address to the topmost bits when storing it into the SPI_ADDR register. For example:

```
WRITE_PERI_REG(SPI_ADDR(spi_no), (uint32) 0b123456789<<(32-9)); //write 9-bit address
```

This shifts it (32-9=) 23 bits to the left. The actual data stored into SPI_ADDR is now 0b12345678900000000000000000000000 and the first 9 bits are shifted out into the SPI data packet correctly as required.

SPI_USER2

This register holds both the command bit length, and the actual command data itself.

SPI_USR_COMMAND_BITLEN: Max value of 0xF or 15. Thus max command length is 16 bits. Again, you set a value one less than the actual bit length. You cannot tell it that the command length is zero bits as it would be pointless (just don't enable SPI_USR_COMMAND!).

SPI_USR_COMMAND_VALUE: Actual command value. This is a 16bit value.

Now, just to confuse the hell out of you, the way data is clocked out from the command register is different again!

SPI_USR_COMMAND_VALUE = 0x24DF (for example)

The command value has a high byte and a low byte. 0x24 is the high byte, 0xDF is the low byte.

The LOW BYTE is always clocked out first, from MSB to LSB. Followed by the HIGH BYTE from MSB to LSB. If you set the command length to:

4 bits: you would get 0xD as the output

8 bits: you would get 0xDF as the output

12 bits: you would get 0xDF followed by 0x2

If you decode what's going on in the espressif 9bit LCD write code, they are essentially rearranging bits to make it work with the above logic.Bits 8 through Bit 2 get shifted down a bit. Bit 1 is discarded for now. Bit 9 becomes Bit 8. Finally, the original Bit 1 is moved to Bit 15! But in the final output stream you get Bit 9, then 8, etc as required. Tricky business!

For my 3 bit command, I simply shift it to the left 5 bits so it sits in the upper half of the low byte:

WRITE_PERI_REG(SPI_USER2(spi_no), (((2&SPI_USR_COMMAND_BITLEN)
<<SPI_USR_COMMAND_BITLEN_S) | 0b101<<5));

Once you have your everything setup for the SPI packet, simply set the SPI_USR bit in the SPI_CMD register!
SET_PERI_REG_MASK(SPI_CMD(spi_no), SPI_USR);

Hopefully the above is of some help. It's a lot to take in, and there are no pretty graphs to help you along (yet!). Some of this may not be entirely 100% correct as I didn't try every possible combination. I believe I tried enough cases to narrow it down to these rules though! Best thing to do is verify the output on a scope (preferably with SPI decoding) or on a Saleae Logic type device.

I'm working on a generic SPI driver which can be used for testing as the current driver code needs to be modified to suit each case. I might do a walkthrough of my SPI EEPROM code as well as an example. It needs two separate functions for read and write. For write, MOSI is enabled and MISO disabled. For read, MOSI is disabled, and MISO enabled. This is only setup in the spi_init function in the espressif driver, but can be easily added to each write/read function as needed without doing a whole new init process.

I am currently looking for information on how to invert the polarity of the HSPI CS line, as I can't seem to find a defined register for it. It just happens that my SPI EEPROM is one of the uncommon devices that requires CS active high, not active low like the ESP8266 default.

Any corrections or additions are more than welcome. Please post in the comments below.

http://d.av.id.au