The cuckoo's nest

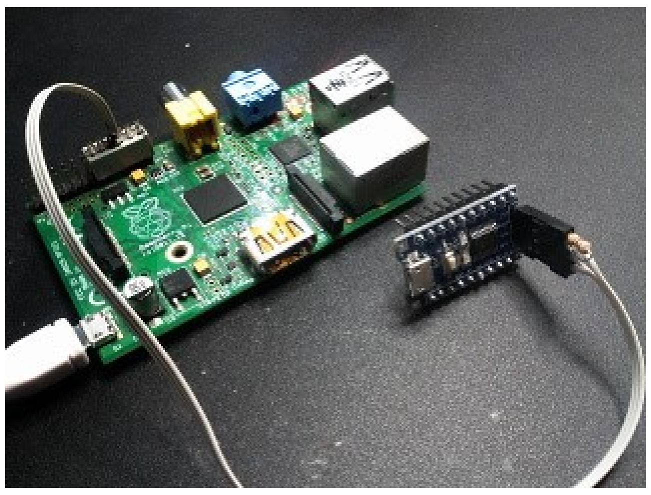
Welcome
Keyboard brain transplant
Hacking a Chinese IP camera: part 1
Hacking a Chinese IP camera: part 2
tiny SPI-based STM8 programmer «
Mysterious LED matrix drivers
What is a Warbird and why is it stealing my RAM?
coming soon: Hacking a USB webcam for fun (and profit?)

STM8 SWIM protocol
(ab)using SPI
The implementation
  Hardware
  Software
Source code

STM8 microprocessors utilize a proprietary (but well documented) programming protocol called "SWIM". The usual way to program these is with a ST-Link programmer. Without such luxury at hand, I managed to talk to the SWIM interface using a Raspberry Pi's SPI interface, one transistor and one resistor. Of course, it would be simpler to just buy an ST-Link, but that's not the point :)

Here's the source code if you're interested.

TL,DR: connect a pull-up resistor, MISO and MOSI to SWIM (MOSI through an open-collector transistor), ignore SCLK. Drive the SPI hardware at 8MHz, generate 8MHz-resolution output bit sequences in software, send out these buffers through SPI, parse the returned buffers as bit sequences in software. Implement SWIM on top of that.



STM8 SWIM protocol

Top

The programming (and debugging) protocol for STM8 is documented on the manufacturer's website. The flash programming manual is also relevant. If you are interested in the details, go ahead and read these. I will try to describe the essentials here.

To quote from the document, *"The SWIM is a single wire interface based on asynchronous, high sink (8mA), open-drain, bidirectional communication"*. In other words it uses one bidirectional pin with a pull-up resistor which is driven low by either the STM8 or the debugger. SWIM communications are synchronized to the STM8's SWIM clock, which defaults to 8MHz (regardless of the clock source of the microcontroller). Data bits are transmitted with a sequence of low/high pulses on the SWIM line. For example, in the "low speed bit format":

- transmitting a *0* means pulling the SWIM line low for 20 SWIM clock lengths (2.5us), followed by releasing the line for 2 SWIM clocks (0.25us)
- transmitting a *1* means pulling the SWIM line low for 2 SWIM clock lengths (0.25us), followed by releasing the line for 20 SWIM clocks (2.5us)

On the receiving side, the signal is decoded as a *0* bit if the low signal exceeds 9 SWIM clocks, or *1* otherwise.

data bit	idle	0	1	idle			
time [us]	0	1	2	3	4	5	6
SWIM line	1	0	1	0	1	1	

Most of the high-level operations are defined as sequences of these bits, there is also the "entry sequence" which consists of slower signalling.

Any device implementing this protocol needs to read/write the SWIM line quickly (8MHz resolution) and with predictable timing. A bit-banging implementation would not be practical on a Linux-based board (like the Raspberry Pi), because meeting the timing requirements could be difficult (feel free to prove me wrong :).

is not very interesting for our purposes). One of the devices, designated as the "master", generates the clock signal (SCLK). During each cycle of the clock signal, both of the devices send and receive one bit of data, master transmits on the MOSI (master out, slave in) line, slave transmits on the MISO (master in, slave out) line. Usually, the sequence of bits is interpreted in packs of 8 (bytes), with the most significant bit transmitted first.

bytes (master->slave)	idle	0x12								0x34								idle
bytes (slave->master)	idle	0xfe								0xdc								idle
SCLK	0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	1 0	0
MOSI	X	0	0	0	1	0	0	1	0	0	0	1	1	0	1	0	0	0
MISO	X	1	1	1	1	1	1	1	0	1	1	0	1	1	1	0	0	0

The master device is responsible for driving the clock signal whenever it wants to communicate with the slave. Obviously, it also determines the transmission speed by controlling the clock frequency. Most microcontrollers have dedicated SPI hardware which allows for precise clock generation (on the master side) or synchronization (on the slave side) and timing of the MISO/MOSI signals. More advanced devices support DMA, meaning that whole sequences of bytes can be streamed from the device's RAM to SPI (and back) while the CPU executes other code.

Note that the SPI master hardware can be thought of as a digital signal sampler/generator - whenever data is to be transmitted/received, the SPI controller generates the clock signal and outputs some bits in sequence on MOSI while sampling another sequence of bits on MISO. We can ignore SCLK, and use MISO/MOSI as a high-speed signal generator/analyzer. This allows us to implement a high-speed "bit-banging" protocol in software, with some limitations:

- The whole output bit sequence needs to be prepared ahead of time, there is no way to adjust it during transfer (for example in response to received data).
- So, the master needs to transmit some padding bits whenever it expects a response to be received.
- The master device needs to keep SPI running to receive any responses from the slave. Obviously, the method is not very suitable to slave-initiated communications.
- SPI hardware *can* choose to arbitrarily delay the clock, skewing our timings. With a real SPI slave this is not a problem, since the slave is synchronized to the clock signal. When the clock is ignored, this would lead to incorrect timing (output) and lost data (input). In practice, this rarely happens on the Pi, possibly thanks to its DMA-backed SPI hardware (however, I'm not sure if the linux spidev driver utilizes DMA).

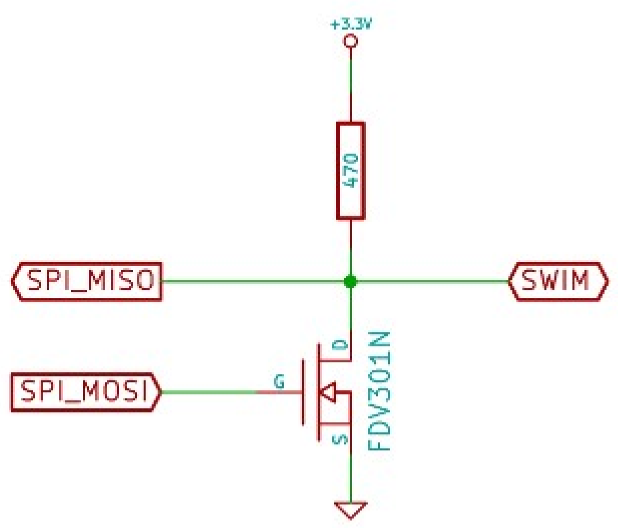
The implementation

Top

Hardware

Top

I've decided to use a Raspberry Pi (lying around) as the SWIM programmer. First, I've needed to combine the MISO and MOSI lines to operate on one wire. The schematic below shows a simple open-collector (or rather open-drain) configuration. Apart from that, the Raspberry Pi needs to drive the STM8's reset line with one of it's GPIO's, but that's not very interesting.



Welcome

Keyboard brain transplant

Hacking a Chinese IP camera: part 1

Hacking a Chinese IP camera: part 2

tiny SPI-based STM8 programmer «

Mysterious LED matrix drivers

What is a Warbird and why is it stealing my RAM?

coming soon: Hacking a USB webcam for fun (and profit?)

The cuckoo's nest

The transistor configuration works as an inverter, so that driving MOSI **high** pulls SWIM **low**. The MISO line monitors SWIM directly, so that it will receive any bits transmitted by the Pi or the STM8. Transmitting a sequence of bits on MOSI without the STM8 attached should in theory return the same sequence, inverted, on MISO. In reality, the transistor's turn-off time turned out to be skewing the results, extending each low-level period considerably. My software works with a simple work-around - the low periods are simply transmitted shorter than required. It sucks, but it works.

Software

Top

The programming software accesses SPI using the Linux spidev driver (/dev/spidev#.#). The driver allows programs to send out a reasonably long buffer of bytes and simultaneously receive the response into another buffer. The software fills the SPI buffer using three primitives:

- transmit low speed bit **0**
- transmit low speed bit **1**
- transmit nothing, making space for the STM8 to send one low speed bit (plus some margin)

Analogously, the received buffers are interpreted as low-speed bits 0/1, skipping the parts transmitted by us to recover data transmitted by the STM8.

Based on these primitives, SWIM operations such as ROTF (read-on-the-fly) and WOTF (write-on-the-fly) can be implemented. An example transmission is shown below. Abbreviations: **a** - ack, **X** - the host sends a "space" instead of 0/1 bit

operation	Pi: Read 1 byte from 0x123456							
SWIM (Pi->STM)	ROTF		1		0x12		0x34	
ls bits (output)	0 0 0 1 1	X	0 0 0 0 0 0 0 0 1 1	X	0 0 0 0 1 0 0 1 0 0	X	0 0 0 1 1 0 1 0 0 1	X
SWIM (STM->Pi)		a		a		a		a
ls bits (input)	0 0 0 1 1	1	0 0 0 0 0 0 0 0 1 1	1	0 0 0 0 1 0 0 1 0 0	1	0 0 0 1 1 0 1 0 0 1	1

The host prepares a SPI buffer with the following operations:

1. Send out a low-speed-bit sequence representing the "ROTF" (read-on-the-fly) operation.
2. Make some space to receive an acknowledgement from the STM (X)
3. Assume that the ack ('1' low-speed-bit from the STM) was received correctly. This will be verified only after the whole buffer is transferred and response is received.
4. Send subsequent data bytes (1, 0x12, 0x34, 0x56) making space for acks.
5. Make some more space to receive the response byte from the STM.
6. Preemptively send an acknowledgement confirming that the data was received correctly.

After the I/O is performed, the response buffer is scanned for ack's, and responses. Occasionally, STM doesn't acknowledge these operations properly, I assume that it's because of timing disruptions mentioned above. In such case, the operation is simply retried.

ROTF and WOTF operations, in turn, allow the whole flash programming procedure to be performed. See the manufacturer's documentation or source code for the boring details.

Source code

Top

I've used stm8flash as the base for my programming software. The original code supports SWIM, although only using the ST-LINK hardware. I've extended it to support SPI-based SWIM programming, and uploaded my version to github.

Add comment

Alfredo *2017-06-25 21:22:00*
Excellent tool and a nice tutorial. According to your experience, how hard would it be to implemented the debugging part of the SWIM protocol also?
respond