How do you handle large data transfers on very memory constrained, embedded systems?

Asked 16 years ago Modified 12 years, 2 months ago Viewed 5k times

6

I have a microcontroller that must download a large file from a PC serial port (115200 baud) and write it to serial flash memory over SPI (~2 MHz). The flash writes must be in 256 byte blocks preceded by a write command and page address. The total RAM available on the system is 1 kB with an 80 byte stack size.

This is currently working by filling a 256 byte buffer from the UART and then ping-ponging to a another 256 byte buffer being filled by an interrupt on the RX buffer ready signal while the flash is written to with busy writes. The buffer swapping is repeated until the operation is complete.

I would prefer to setup TX/RX interrupt handlers for both the SPI and UART ports that operate on seperate circular buffers. So, instead of polling for new bytes and waiting for operations to complete I can simply fill the TX buffers and enable the interrupt or check the buffers for incoming data. This would give a lot more clock cycles for real work instead of waiting on peripherals.

After implementing the IRQ's with 128 byte circular buffers, I poll the UART RX buffer for data and immediately place it in the SPI TX buffer to do the file transfer. The problem I am having with this approach is that I don't have sufficient RAM for the buffers and the PC receive buffer is filling up faster than I get the data over to the flash transmit buffer. Obviously, transmission speed is not the problem (115.2 kHz in and 2 MHz out), but there is a write cycle wait after each 256-byte page is transmitted.

It appears the frequent SPI interrupts were blocking some of the UART interrupts and causing bytes to be missed. The solution I chose was to use a ring buffer for the UART receive interrupt and feed the data into a 256 byte page buffer that is sent to the serial flash by polling for byte transfers and write completion. A 128 ring buffer is big enough to prevent overflows during the SPI write.

c embedded serial-port interrupt spi

Share

Improve this question

Follow

edited Oct 2, 2012 at 12:31

dsolimano

8,976 ● 3 ● 50 ● 64

asked Dec 11, 2008 at 15:37

Judge Maygarden

27.5k ● 9 ● 83 ● 100

4 Answers

Sorted by: Highest score (default)

4

Does the UART and the PC side of the application support RS-232 handshaking (flow control)? If so, when your receive buffer gets close to being full, have the ISR drop the CTS line - if the PC side is configured to respect hardware flow control it should stop sending when it sees this condition. Once you have drained (or nearly drained) the receive buffer, assert CTS again and the PC should start sending again.

Note that this makes the software on the embedded device considerably more complex - whether that's a trade-off you're willing to make would have to be analysis done by you and your manager & team.

Share Improve this answer

Follow

answered Dec 11, 2008 at 15:49

Michael Burr

340k ● 52 ● 548 ● 769

This actually happens through a USB interface and those signals are unavailable. It's good advice though.
– Judge Maygarden Dec 11, 2008 at 15:57

4

This is exactly what flow control was created for, I know its a huge pain set up but if you enable flow control on the serial line your problems would be history.

I'm assuming you're transferring a binary file so XON-XOFF isn't the best solution, which leaves hardware flow

control.

Another option is to use a protocol with built-in flow control such as XModem. I have a similar embedded project where the flash is written in 128byte pages. What a coincidence that XModem sends data in 128byte chunks then waits for an ACK before it sends the next.

Share Improve this answer

Follow

answered Dec 11, 2008 at 15:50

joshperry

42.2k ● 16 ● 92 ● 104

> I have used XModem in the past and have code laying around for it. However, I'm coming in on a contract and can't change the PC client (it just spews the whole file at once).
> – Judge Maygarden Dec 11, 2008 at 15:56

3

I'd do something like a scatter gather on a PC. Create a linked list of a struct like this:

```
typedef struct data_buffer {
    char flags;
    char[128] data;
}
```

Have one of the bits in the flag mean "ReadyToFlash" and one for "Flashing". You should be able to tune the number of buffers in your linked list to keep the flash from catching the UART as it writes or vice versa.

If the flash gets to a buffer block that isn't "ReadyToFlash" it would stall and you'd need to have your UART IRQ start it back up. If the UART gets to a block that is "ReadyToFlash" or "Flashing" it is filling too fast and you probably need another buffer, if you have dynamic memory you could do this tuning at runtime and add a buffer to the list on the fly, otherwise you'll just need to do some empirical testing.

Share Improve this answer

Follow

answered Dec 11, 2008 at 16:24

joshperry

42.2k ● 16 ● 92 ● 104

Note that 1 kB (1024 bytes) of RAM doesn't allow for very many buffers! Nonetheless it's good idea.
– Judge Maygarden Dec 11, 2008 at 16:28

True, though you could also try tuning the size of the buffers down (maybe 64b), but the overhead of the flags starts to impact your memory efficiency. – joshperry Dec 11, 2008 at 16:41

I would use a single flag variable with bites for each buffer.
– Judge Maygarden Dec 11, 2008 at 16:43

This also brings up the good point that I have been trying to do this without modifying the ISR to change behavior when in this transfer state. – Judge Maygarden Dec 11, 2008 at 16:45

1

Not sure what I'm missing here, but if the fact is that the average rate of data coming from the PC is higher than the average rate you can write it to the flash, then you're either going to need a lot of RAM, or you're going to need flow control.

But are you saying that it worked when you had block buffers, but now that you have byte buffers it doesn't?

Can you stick with the block buffers which are filled by the UART RX interrupt, and when each buffer is full, hand it off to the SPI/Flash code to empty that buffer using the SPI interrupt? That will save you copying each byte, and instead of having to do the circular buffer logic twice for each byte, you'll only have to do it for each block.

Share Improve this answer

Follow

answered Dec 11, 2008 at 16:05

Will Dean

39.5k ● 11 ● 92 ● 118

The wait-cycle for a page write after the transmission is the bottleneck. I added that information to the question.
– Judge Maygarden Dec 11, 2008 at 16:12

I can get it to work with a 128-byte circular buffer on the UART and a 256-byte intermediate buffer with busy write/poll to the SPI. 128-byte buffers without the 256 intermediary doesn't work. – Judge Maygarden Dec 11, 2008 at 16:13