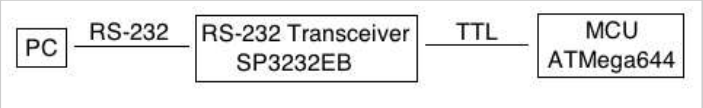
BLOG

MONDAY, MAY 16, 2011

RTS/CTS handshaking and waveforms

I decided to learn about RS-232 signalling and handshaking. I had a heck of a time finding a single page which summed up everything I would've needed to know, so I'm writing this post in the hopes that it'll help someone else in the same situation.

Here's what I'm trying to do: I have a DB-9 serial port hooked up to an ATMega644 microcontroller (MCU), with an SP3232EB RS-232 transceiver in the middle, as shown below:



Three-wire RS-232 (RX, TX, GND) wasn't sufficient for my purposes, so I'm going to do five (RX, TX, RTS, CTS, GND).

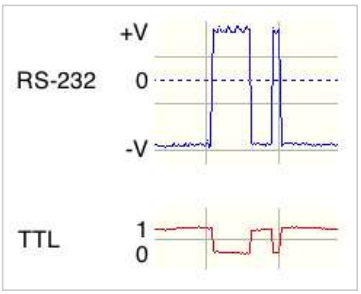
When we talk about RX and TX, we're going to do it from the MCU's perspective. That is, we read from the PC on RX, and we transmit to the PC on TX. RTS and CTS have non-reversible meaning, so no clarification needed.

How does this all work? RTS/CTS handshaking is used to ensure that the peripheral (the MCU, in our case) is ready to handle data from the host (the PC). It doesn't appear to apply in the other direction (or if it does, I haven't found the need to care). The idea is that the PC is powerful enough to handle just about anything you may care to throw at it, but the dinky MCU isn't so lucky. The PC can easily overflow any buffers the MCU may care to put in place, so the MCU needs a way to tell the PC when the PC is allowed to send new data. That's where the handshaking comes in.

There are two ways to do RTS/CTS handshaking. The first is request/acknowledgement. When the PC is ready to send data, it uses RTS (request to send) to tell the MCU that it (the PC) has data to send. I say "uses" because the logic levels are a bit funny, as described below, but are unimportant for now. The MCU sees that the PC is ready to send data. When the MCU is ready to receive it, it uses CTS to tell the PC that the data can be sent. The PC watches CTS to see a) when it can start sending data, and b) when it must stop. The second type of RTS/CTS handshaking does bidirectional metering (more details here). I've chosen to implement request/acknowledgement, but conversion to bidirectional would be straightforward.

When interfacing RS-232 and TTL, we have to deal with two different standards for logic levels. TTL is 0-to-not-zero. 0-to-5, in the case of my circuit. 5V is asserted; 0V is not asserted. RS-232, on the other hand, uses positive and negative voltages. A positive voltage is referred to as a space, while a negative voltage is a mark. The transceiver's job is to translate between RS-232 and TTL voltages. It does so by mapping RS-232 positive to TTL 0, and RS-232 negative to TTL 1. That is, the TTL voltages are roughly the inverse of the RS-232 voltages.

Here's the letter p (0x70), being sent across the wire at 9600 baud, 8N1:



Now that we know this, let's restate the operation of RTS and CTS using the TTL voltages generated by the transceiver. This is probably the most important paragraph of this post, and took a surprisingly long time to beat into my thick skull.

RTS and CTS are normally held high (TTL 1). When high, no request (RTS) has been made, and no permission

FOLLOWERS

Người theo dõi (6)

Theo dõi

BLOG ARCHIVE

- ▶ 2019 (1)
- ▶ 2012 (10)
- ▼ 2011 (31)
  - ▶ December (2)
  - ▶ October (4)
  - ▶ September (4)
  - ▶ July (3)
  - ▶ June (2)
  - ▼ May (3)
    - My own microprocessor
    - Gran Fondo NYC
    - RTS/CTS handshaking and waveforms
  - ▶ April (3)
  - ▶ March (8)
  - ▶ January (2)
- ▶ 2010 (13)

ABOUT ME

SIMMONMT

VIEW MY COMPLETE PROFILE

SUBSCRIBE VIA EMAIL

Enter your email address:

Subscribe

Delivered by FeedBurner

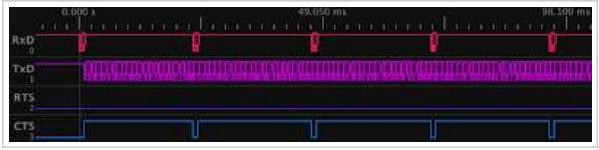
(CTS) has been granted. When the PC is ready to send data, it brings TTL RTS low. The MCU sees this. When the MCU is ready to receive data, it brings TTL CTS low. The PC sends data as long as TTL CTS is low, and stops when the MCU takes TTL CTS high. One imagines that the sender is allowed to finish the current symbol (byte), but I haven't verified that.

Now, to wire everything up. As stated above, we're using RX and TX from the perspective of the MCU, as that's how everything is labeled on the MCU's end (i.e. in the MCU itself and in the transceiver).

			SB3232EB (MAX232)		
Function	**Direction**	**DB-9 pin**	**RS-232 pin**	**TTL pin**	**AVR pin**
RX	PC to AVR	3	13 (R1IN)	12 (R1OUT)	14 (PD0/RX)
TX	AVR to PC	2	14 (T1OUT)	11 (T1IN)	15 (PD1/TX)
RTS	PC to AVR	7	8 (R2IN)	9 (R2OUT)	17 (PD3)
CTS	AVR to PC	8	7 (T2OUT)	10 (T2IN)	16 (PD2)

I messed up the wiring a couple of times until I drew up the above chart. Then everything went together smoothly.

Here's the end result, captured by an Open Bench Logic Sniffer via the OLS LogicSniffer client:



We're looking at an attempt by the PC to send a multi-character string as fast as it can, only to be metered by the MCU. Please excuse the bubbles -- when zoomed in, they show the byte values being transmitted. As we can see, the MCU idles with CTS low (ready to receive). A character comes in, and the MCU raises CTS (stopping further transmissions from the PC) while it processes the received byte. In this case, the program running on the MCU sends some data back to the PC describing the received byte. Once that's done, the MCU lowers CTS to receive another byte, raises CTS once that byte is received, and repeats the process. By doing this, we receive everything the PC wanted to send -- nothing gets lost due to MCU buffer overflows.

Here's some sample code, written for the ATMega644:

```
#include <avr/io.h>

#define UBRR_ASYNC_NORMAL(baud) \
  (((((F_CPU * 10) / (16L * baud)) + 5) / 10) - 1)

#define RTS PD3
#define CTS PD2

#define BAUD 9600

static int uart_haschar(void) {
  return (UCSR0A & (1 << RXC0));
}

static int uart_getchar(void) {
  while (!uart_haschar());
  return UDR0;
}

int main(void) {
  UBRR0H = UBRR_ASYNC_NORMAL(BAUD) >> 8;
  UBRR0L = UBRR_ASYNC_NORMAL(BAUD);
  UCSR0B = (1 << RXEN0) | (1 << TXEN0);
  UCSR0C = (0 << USBS0) | (3 << UCSZ00);

  DDRB = 0b11101111;  // PB4 = MISO
```

```
  DDRD = 0b11110110;  // PD0 = RX, PD3 = RTS

  // Inhibit CTS by setting it to TTL 1 (RS232 off)
  PORTD |= (1 << CTS);

  for (;;) {
    // Wait for host to set RTS (set it to TTL 0)
    while ((PIND & (1 << RTS)) != 0);

    // Set CTS (TTL 0 == RS232 on)
    PORTD &= ~(1 << CTS);

    char c = uart_getchar();

    // We have a character, so clear CTS (TTL 1 == RS232 off)
    PORTD |= (1 << CTS);

    // Do something with c
  }

  return 0;
}
```

There, of course, other ways to guard against buffer overflows. We could read from the UART using an interrupt handler, but that a) complicates the code considerably and b) only works if we can keep up with the uncontrolled data rate over the long run. Perhaps more realistically, we could structure our commands to the MCU such that they all have responses, offloading the responsibility for guarding against overflows onto the PC. That won't work, though, if we want the microprocessor to be able to generate its own messages asynchronously.

Updated: Clarified terminology for ATMega/AVR/MCU, added diagram, fixed T1IN pin number, described the other way to look at RTS/CTS handshaking for bidirectional use.
Updated: Changed initial setting of PORTD so it just sets the CTS bit. Not absolutely necessary, since there's nothing else in this code that'll be affected by it, but it'll help anyone who tries to cut'n'paste.

POSTED BY SIMMONMT AT 8:16 AM

LABELS: ELECTRONICS

17 COMMENTS:

ark May 16, 2011 at 3:49 PM

You lost me right at the start :(

"""Here's what I'm trying to do: I have a DB-9 serial port hooked up to an ATMega644, with an SP3232EB RS-232 transceiver in the middle. """

So is it: computer <-> SP3232EB <-> ATMega644?

and then you say:

"""When we talk about RX and TX, we're going to do it from the AVR's perspective."""

What's an AVR and where did this enter the mix?

ark

Reply

Replies

Owen DeLong March 9, 2012 at 1:22 AM

AVR is the initials of the engineers that developed the first ATMega chips and is a common nickname for the ATMega series of Microcontroller MCU chips.

Reply

simmonmt May 16, 2011 at 3:56 PM

The PC talks RS-232, with RS-232 levels. The microcontroller (the ATMega644) uses TTL levels. The SB3232EB lives between the two, and translates between RS-232 and TTL. See the picture below "Here's

the letter p" to see the difference between the two.

AVR is the name for the architecture used by Atmel for the line of microcontrollers that I'm using. So in this case, AVR and ATMega are roughly synonymous. I'll reword the text to make it clearer.

Reply

Maniac May 17, 2011 at 10:55 AM

I think there's a typo in the pinout table... you have pin 12 for both TTL in and TTL out.
I guess one of these should be pin 11

Reply

simmonmt May 17, 2011 at 11:04 AM

@Maniac: Fixed. Thanks!

Reply

CVBruce May 17, 2011 at 11:34 AM

"One imagines that the sender is allowed to finish the current symbol (byte), but I haven't verified that."

I agree with that statement, simply because of the nature of the protocol. For example 8N1 means one start bit, 8 data bits, no parity bits, and one stop bit. The start and stop bits are used to synchronize the transmitter and receiver. If you were to stop and start the bit stream, then that synchronization would be lost.

Secondly, and this may reveal my ignorance, but I've never heard of a UART that didn't have at least a one byte send and receiver buffer.

Reply

Ag May 17, 2011 at 11:42 AM

I'd like to make a clarification to your blog entry.

RTS/CTS are not request/acknowledge signals.

RTS is used to meter the flow of data from the DCE (the AVR) to the DTE (the PC).

CTS is used to meter the flow of data from the DTE (the PC) to the DCE (the AVR).

In your example, the PC is always ready to accept data, so RTS is always asserted. But the AVR is not always ready to accept data, so CTS will tell the PC to hold off sending data until the AVR is ready.

Reply

simmonmt May 17, 2011 at 12:09 PM

@CVBruce: The ATMega has at least one byte, but in my case one byte isn't enough. Say I'm writing something back to the PC in response to the command it just sent. Or perhaps some asynchronous event just happened, and I'm telling the PC about it. In either case, I'll be busy doing other things (writing to the PC), and won't be checking for incoming data. If a multi-byte command comes in during that time, I'll miss at least part of it. Hardware handshaking lets me avoid that.

@ag: RTS/CTS seems to be usable in both the request/ack mode I've described and the bidirectional metering you're describing. See the wikipedia page for RS232 (http://en.wikipedia.org/wiki/RS-232#RTS.2FCTS_handshaking) for a description.

Reply

Anonymous May 18, 2011 at 5:25 AM

Could you add details on the second figure (letter "p") to show where are the various bits ?

Reply

Owen DeLong March 9, 2012 at 2:43 AM

Multipart post due to 4k limit

One of the reasons it is hard to find a single page that sums up Serial Communications and RS-232 in particular in clear easy to understand terms is because it is a complex subject and there are about as many different ways to do RS-232 as there used to be formats for 5.25" floppy disks (hint, this included most of the permutations of {hard,soft} sectoring, {various numbers of sectors} , {various numbers of tracks, IIRC, 20, 35, 40, and 80}, {single,double} sided. Very common formats included 120k, 240k, 360k, 720k, and 1.2MB, but, that was late in the lifespan of the 5.25" floppy when things were roughly standardized on those formats, hard-sector disks were largely a thing of the past, and most drives could read/write all of those formats. In the earlier days, each manufacturer had their own idea of how to do things and you couldn't read a floppy written in one manufacturers drive in some other manufacturers drive. Enough of this digression... RS232.

First, I think we should talk about serial communications in general. Serial communications at its heart includes any signaling mechanism, whether bidirectional, unidirectional, or multipoint, full or half

duplex, where the number of parallel signaling paths is less than the number of bits contained in each encoded symbol. Usually, if you say serial, people assume a single bit data path or at most a single bit data path in each direction. However, the opposite of serial is parallel, which, when you use that term, people assume at least 8 and often 16, 32, 64 or even more bits of simultaneous signaling. There are many devices that use 2 or 4 bit-wide signaling paths and their behavior incorporates elements of both serial and parallel behaviors, but, is largely governed by the difficulties inherent in serial.

Interestingly, as data rates have climbed, crosstalk has become enough of an increasing problem that single-bit-datapath serial has become the more dominant fast-path for anything leaving the tightly controlled environment of the carefully designed multi-layer PCB. Again, I digress...

Serial comes in two principal flavors. Synchronous and Asynchronous. RS-232 is usually used to describe Asynchronous behavior and in fact most PC Serial ports are incapable of implementing truly synchronous communications even though the 16550 USART is actually capable of doing so. Few PC cards bring out all of the necessary pins from the 16550 to do so.

Reply

Owen DeLong March 9, 2012 at 2:44 AM

Part 2

In synchronous serial, the baud rate is nearly irrelevant because one of the systems provides a clock signal and all transmissions in either direction are synchronized to that clocking signal. The advantage of such a system is that it is not susceptible to clock drift as the sending system will simply wait for the clock to change state before changing state for the next bit and the receiving system won't start its read delay until it sees the clock change state.

In asynchronous serial, each system maintains its own internal "baud clock" and communication depends on the two systems having their clocks set to run at roughly the same speed. Note, this baud clock is not the same as the CPU clock. In order to synchronize framing, each symbol transmitted asynchronously is wrapped in a preamble and a suffix. The preamble is almost always a single Mark (logic 1) bit and the suffix is usually 1 or 2 (usually 1) Mark (logic 1) bit. (Logic 0 is called a Space).

Symbols are (usually) transmitted from lab to MSB using shift registers clocked by the baud clock. On the original teletypes, these shift registers were electromechanical devices and the start bit literally caused the register to reset to all zeros and the stop bit(s) caused the register to latch, position the printhead to the proper character, and fire the striker to make the impression on the paper. (these multiple operations are the reason that 2 stop bits used to be common.

RS-232 as a specification has not changed significantly since the days when it was first developed to drive teletypes over long distances using frequency shift keyed modulator/demodulator devices (modems). (In FSK, two frequencies are chosen for each direction of communication, e.g. 300/900 hz left to right and 600/1200hz B to A. a 300hz tone represents a space from A to B, a 900 hz tone a Mark from A to B, 600/1200 being mark and space from B to A, respectively).

RS-232 doesn't use FSK itself, however, it uses positive and negative voltages. Because it was designed to go fairly far (3,000 feet or more at lower speeds), fairly high voltages (relative to transistor logic) and wide voltage swings were used. An RS-232 Mark ranges from -5 to -15 volts at the transmitter and the receiver must accept anything between -3 and -15 (relative to signal ground (pin 7 on your typical DB-25 RS232 connector, pin5 on a DD-9 (often mistakenly called a DB-9))) . Conversely, spaces are +5 to +15 (+3 to +15 at the receive side). Notice that that's a maximum 30 volt swing, 6 times the Vcc level in your typical Arduino and nearly 10 times the logic level in most micro controllers and peripherals.

To translate TTL to RS-232 line levels, a variety of "line driver" chips were developed, including the 488/489 series, the 1488/1489s, and the authors SP3232EB (which is a low-power line driver with a fairly weak voltage swing at 13V differential V+ to V-). Some cheap USB<->RS-232 adapters are even worse and only generate +5/-0v RS-232 signaling. This has become so common that most UARTS will (contrary to the RS-232 specification) accept 0 volts (actually anything <+0.3v) as a valid mark and anything >2.2v as a valid space.

Reply

Owen DeLong March 9, 2012 at 2:45 AM

Part 3

Of course, especially back in the days of teletypes, it was relatively easy to overrun devices and many of them had very small (or no) buffers. This led to all kinds of entertaining mechanisms, including software and hardware flow control, RTS/CTS symbol metering, and inventions such as padding (a series of non-printable characters, usually nulls, sent after a carriage return to provide a delay while the mechanical printhead was moved from the right edge to the left edge of the paper. If you can find a n old teletype, send it a line of text followed by a and immediately more text and you will notice that the first 1 or 2 characters of the second line of text are printed randomly in reverse order as the printhead is returning to the left side of the page).

Basic minimum RS-232 with no flow control requires 3 pins. A signal ground, and 1 pin for data in each direction.

Software flow control also requires only 3 pins and manages flow control by sending specially designated characters (XON or Ctrl-Q for Transmit On or "Continue sending" and XOFF or Ctrl-S for Transmit Off or "Stop sending"). These flow control characters are sent from the receiver to the sender and the sender is expected to honor them as soon as the current symbol (and its suffix) are shifted out. However, the risk is that with longer cables and higher bit rates, there may be more than one symbol on the wire between the sender and the receiver at the same time. Also, since this is serial, there is a time delay while the XOFF symbol is loaded onto the wire and a time delay while it is shifted off of the wire as well.

This is why devices with buffers have high and low "water marks". At the high water mark (buffer (~75% full), the receiver transmits XOFF. As the buffer empties, when it reaches the low water mark (~25% full), XON will be sent. In this way, any extra symbols on the wire can be read into the 25% headroom in the buffer and not be lost to delays in flow control.

RTS/CTS symbol clocking is normally used where the data travels primarily in one direction with only acknowledgements or relatively small pieces of information on the return path (if anything). This signaling method (the one described by the author) requires at least 2 additional pins. One for RTS and one for CTS. In this configuration, when the sending unit has data ready, it informs the host that it would like to transmit (asserts RTS, hence the name "Request to Send"). Usually this is used to trigger an interrupt so that the host can suspend its normal processing and enter a routine to receive the data. When the host is ready to receive (back in the day, this might have included positioning the head in the correct place on magnetic storage for the buffer to receive the data to be recorded, so the delay could be significant in computer timing terms), the host says "Go ahead" (asserts CTS, hence the name "Clear to Send"). When the receiving host drops CTS, the sender is obligated to stop sending as soon as it finishes the current contents of the shift register. Again, this does not necessarily mean that the receiving host will not receive as many as 5 more symbols of data (though it's usually just 1 or 2). When the sender is finished, he will drop RTS. This signals the host that as soon as there is a gap in arriving symbols, he is free to return to normal processing. This flow control mechanism can also be used bi-directionally in a half-duplex environment where both sides monitor RTS as an input until one of them wants to send something. If they both assert RTS at the same time, neither will notice and CTS will not be asserted. After a timeout, both hosts will back off for a random amount of time and try again. This was rarely used as it is hard to implement in the hardware of the time compared to the next form...

Reply

Owen DeLong March 9, 2012 at 2:46 AM

Fourth Part

A slightly more refined form of hardware flow control that is useful in peer-to-peer environments where data flows may not be strongly asymmetrical and may be simultaneous in both directions is to use these same two pins (RTS/CTS) to indicate unit readiness. Generally each unit will assert CTS (Clear to Send) as an output to indicate that it is ready to receive additional symbols and will monitor RTS for permission to transmit. The exception is modems or "data communications equipment". You will often hear terms in RS-232 such as "DTE" or "Data Terminal Equipment", "Modem Cable", "Full Modem Cable", "Null Modem Cable", "DCE" or "Data Communications Equipment". These terms have to do with how the pins are arranged on the connectors. Generally, if you want to use cables where the pinout is identical at both ends, you need to be connecting something wired for DTE to something wired as DCE. DTE includes things like computers, terminals, printers, pretty much anything that terminates (hence "Data Terminal Equipment") a connection and does something with the data other than convert it for transmission over some other form of circuit. DCE generally means modems and other transmission-schem converters. (hence the term "Data Communications Equipment"). A modem cable means a cable that includes at least TX, RX, RTS, CTS, SG (Signal Ground) and CD (Carrier Detect) with the pinouts being identical on both sides. Usually a modem cable will also include at least DTR (Data Terminal Ready). A Full Modem Cable includes at least all of the above and additionally DSR (Data Set Ready) and RI (Ring Indicate) and will often include all 9 or 25 pins (depending on the connector), again wired straight through.

A null modem cable should include at least TX, RX, RTS, CTS, SG, CD, and DTR. It should be wired such that A-TX = B-RX, A-RX = B-TX, A-RTX = B-CTS, A-CTS = B-RTS, A-SG = B-SG, A-CD = B-DTR and A-DTR = B-CD. Effectively this reverses each set of complimentary signals between the two ends of the cable, swapping RX/TX, RTS/CTS, DTR/CD. Of course, reality isn't that simple and the actual specification of null modem is much less precise and varies widely in the handling of RTS, CTS, DSR, DTR, and CD with various combinations of pin mappings representing a wide array of differences in DCE being that is being substituted with a cable (originally RS-232 was designed with the assumption that it would be Device->Modem->phoneline->Modem->Device. All -> cables would be straight-thru (though the phone line cables are not subject to the RS-232 spec. and are actually expected to be flipped). The term null modem comes from the idea that the cable is taking the place of two modems and a phone line.

Reply

Owen DeLong March 9, 2012 at 2:47 AM

Final part

In the most common modem full signaling implementation for Asynchronous, the pins have the following meaning on a DB-25 connector:

1 Frame or Protective ground. Should be wired to the cable shield AT ONE END of the cable if a shield is present.
2 TXD (Data from the device to the modem)
3 RXD (Data to the device from the modem)
4 RTS (Device indicating it is ready to accept more symbols from the modem)
5 CTS (Modem indicating it is ready to accept more symbols from the device)
6 DSR (Modem indicating that it is ready to accept commands or data)[1]
7 SG (Signal Ground - The zero voltage reference for all signals)
8 CD (sometimes called DCD or Data Carrier Detect. Indicates that the modem is linked to a remote modem)
20 DTR (Data Terminal Ready - Host indicating to modem that the host is online)[2]
22 RI (Ring Indicate - Is asserted when ringer voltage is present on the phone line)

[1] DSR is used primarily during offline operations to indicate that the modem is willing to accept commands even though it is not connected to a remote modem. This is usually a concern for things like dialing scripts for making outbound connections.

[2] When a host is finished with a connection it will often De-Assert DTR for a period ≥250ms to indicate

to the modem that it should terminate any active call and reset to a known initial state.

For a project of the type CVBruce has described, I would use RTS/CTS flow control in the bidirectional mode. I would make sure that I held CTS low during any time where I wasn't ready to read the serial port. FWIW, if you use a PIC micro controller, the built-in USARTs are capable of managing RTS/CTS automatically.

Reply

Owen DeLong March 9, 2012 at 2:52 AM

Also, in your code, PORTD = (1 << CTS); should probably read PORTD |= (1 << CTS);

As you have written it, you are setting CTS to 1 and every other pin on PORT D to 0.

If you only want to change CTS, you may get different results than you want.
If you don't have anything else attached to the other 7 GPIOs on PORT D, it might not matter, but, since you've put it up as example code. it might be confounding to others.

Reply

Replies

simmonmt April 3, 2012 at 8:59 AM

Fixed. Thanks!

Reply

Axel July 23, 2012 at 3:42 AM

Wow Owen: thanks for the incredibly detailed and historical explanation. I never knew the reason for two stop-bits: now I know.

Reply

To leave a comment, click the button below to sign in with Blogger.

SIGN IN WITH BLOGGER

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)