Nerd Ralph

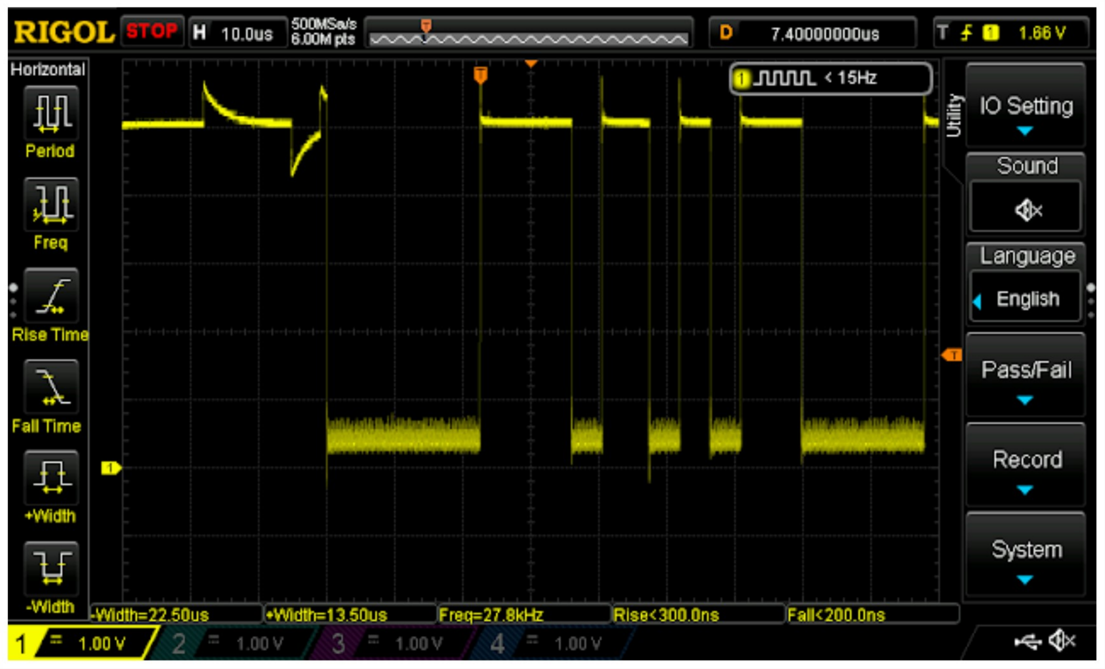
science and technology stuff

Wednesday, February 12, 2020

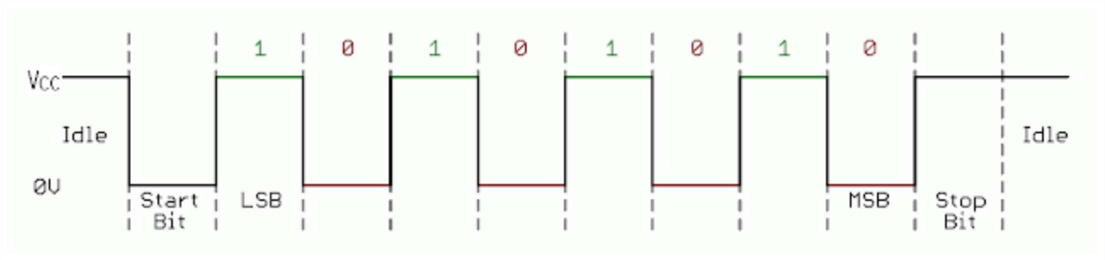
Building a better bit-bang UART - picoUART

Over the past years, one of my most popular blog posts has been a soft UART for AVR MCUs. I've seen variations of my soft UART c other projects. When MicroCore recently integrated a modified version of my old bit-bang UART code, it got me thinking about h improve it.

There were a few limitations to my earlier UART code. One was that it didn't support baud rates below 19.2kbps at 8Mhz or baud 38.4kbps at 16Mhz. It was also problematic for people that tried to integrate it into C/C++ libraries, as the code was written in A assembler. Another problem that was recently brought to my attention by James Sleeman, was that the UART receive didn't work moderately high baud rates such as 57.6kbps. Since my AVR skills had improved over time, I was confident I could make tangible to the code I wrote in 2014.



The screen shot above is from picoUART running on an ATtiny13, at a baud rate of 230.4kbps. The new UART has several improve my old code. To understand the improvements, it helps to understand how an asynchronous serial TTL UART works first.



Most embedded systems use 81N communication, which means 8 data bits, 1 stop bit, and no parity. Each frame begins with a lov the total frame is 1 start bit + 8 data bits + 1 stop bit for a total of 10 bits. Frames can be sent back-to-back with no idle time be The data is sent at a fixed baud rate, and when either the receiver or transmitter varies from the chosen baud rate, errors can oc

When it comes to the timing error tolerance of asynchronous serial communications, I've often read that somewhere between 2% timing error is acceptable. I've also read many "experts" claim that a micro-controller needs an accurate external crystal oscillat avoid UART timing errors. The truth is that UART timing can be off by a total of over 5% without encountering errors. By total, I of the errors for both ends, so if a transmitter is 2% fast, and the receiver is 2% slow, the 81N data frames can still be received er timing on a USB-TTL UART adapter is usually accurate to within 0.1%, so if I am sending data from an AVR that is running 3% slow, adapter still receives it error-free.

If a frame is being transmitted at 57.6kbps, each bit needs to last 1000/57.6 = 17.36us. That means 17.36us after bringing the lir start bit, the least significant bit needs to be sent. A receiver will wait for the start bit to begin, wait another 17.36, and then w middle of the first bit to sample the line. If the line is high, the bit is a 1, and it it is low, the bit is a zero. So the receiver will s first bit 1.5 * 17.36 = 26.04us after the line goes low to signal the start bit. The last(8th) bit will be sampled after 8.5 *17.36 = 14 the transmitter is to slow, and is still transmitting the 7th bit, it will cause a communication error, as the receiver will interpret t actually being the 8th bit. If the transmitter is still sending the 7th bit after 147.56us, then it is sending at 8/8.5 or 0.941 * 57.6 Since many UARTs check for a valid stop bit, the maximum timing error is usually 9/9.5 or 94.7% of the baud rate.

The transmit timing of my earlier soft UART implementations is accurate to within 3 clock cycles. This was each iteration of the takes 3 clock cycles - one for decrement and two for the branch:

```
    ldi delayArg, TXDELAY
TxDelay:
    dec delayArg
   brne TxDelay
```

And since delayArg is an 8-bit register, the maximum delay added to the transmission of each bit is 2^8 * 3 = 768 cycles. On a MC 8Mhz, that limited the lowest baud rate to around 8000/768 or 10.4kbps. To allow for lower bit rates, picoUART needed to suppo delays. I also wanted to support more accurate timing, so picoUART uses __builtin_avr_delay_cycles during the transmission of e exact number of cycles to wait is calculated by some inline functions, which is a better way of doing the calculations than the ma used before. Writing picoUART in C made the timing calculations more difficult, since compiler has some flexibility in how the co compiled to AVR machine instructions. In order to get avr-gcc to generate the exact sequence of instructions that I wanted, I hac inline asm statement. When I used a C "while" loop instead of the asm goto "brne" instruction, the loop was one cycle longer due superfluous compare instruction. Future versions of the compiler may have improved optimization and omit the compare, which impact the timing.

As with the transmit code, picoUART's receive code is accurate to within one cycle. Unlike my earlier UART code, picoUART retur reading the 8th bit instead of waiting for the stop bit. Because of this change, picoUART begins by waiting for the line to be high waiting for the start bit. Without the initial wait for high, back-to-back calls to purx() could lead an error when the 8th bit of on 0(low) and gets interpreted as the start bit of the next frame. This change approximately triples the amount of time for the AVR each byte in a continuous stream of data.

My earlier UART code had two incompatible versions. One version used open-drain communication, where the transmit line is pul external resistor, and pulled low by the AVR. This version supported using a single wire for both receive and transmit. While it al with separate pins, some users found it inconvenient to add the pull-up resistor. Instead they would choose the "push-pull" versio AVR drives the line high and pulls it low. With picoUART a single version works for both use cases, because it works in "push-pull" during transmit. When not actively transmitting, the IO pin is set to input mode with the internal pull-up activated.

I've tried to help both the noobs and experienced AVR developers. The noob can download a release zip file to add as an Arduino you are an old AVR developer like me that prefers a keyboard over a mouse, you'll find a basic Makefile with the echo example. T baud rate is 115.2kbps, although it is capable of accurate timing at much higher speeds such as 1mbps for an AVR running at 8Mhz transmit is on PB0, with PB1 for receive. The defaults can be changed in pu_config.h, or with build flags like "-DPU_BAUD_RATE=

Posted by Ralph Doncaster at 6:05 PM

13 comments:

Unknown February 18, 2020 at 8:46 PM

Why "81N"?
That always was being written as "8N1"

Reply

Replies

Ralph Doncaster February 24, 2020 at 2:54 PM

Does it matter? The meaning is unambiguous whether it is stated as 81N or 8N1.

Reply

asdf February 24, 2020 at 2:30 PM

Why does the start of the byte look so bad on the scope shot?

Reply

Replies

Ralph Doncaster February 24, 2020 at 2:56 PM

I was doing an echo test. The incoming signal on the Rx line induces a signal on the adjacent Tx line.

Reply

sebi April 6, 2020 at 11:39 AM

I have tried your picoUART and it works great! Thank you so much for this great piece of software!!
- I have noticed that it works fine when selecting PB0 as the TX/RX pin, but not PB5. Any reason why?
- I have used your 2014 electronic scheme to avoid local echo and it works great. Why didn't you had a resistor in series with the dio
current flow if TX/RX is high and TX is low?
Thanks :-)

Reply

Replies

Ralph Doncaster April 6, 2020 at 4:44 PM

I started it as a header-only lib, so defining the port/pin before including picoUART.h would allow changing the PORT & bit.
inconsistencies with gcc between c++11 mode and c11 mode, I decided to split it into separate files. Since the release ve
code in a separate .c file, to change the pin, you'll have to change pu_config.h.
I'll think about changing it back to a header-only lib, which would permit setting the pin before the #include. Another op
global definitions for the port & pin which should get optimized away by gcc as long as LTO is used.

As for the circuit, for the combined Tx/Rx to work properly, both ends need to know when it is safe to transmit.
communicating with a device that might transmit at the same time as the AVR, using a single pin for Tx/Rx is not recommenc

sebi April 7, 2020 at 9:24 AM

This comment has been removed by the author.

sebi April 7, 2020 at 9:27 AM

> to change the pin, you'll have to change pu_config.h.
Indeed that is what I did, but I have noticed that the half duplex serial communication worked for TX/RX = 0, 1, 2, 3, 4 but
is that?

Ralph Doncaster April 8, 2020 at 8:06 PM

I tested the code on a few different pins with a t13 and a t85, but they don't have PB5 as IO. I tested a build with TX/RX =
compiled code looks good. What MCU are you trying it on? Besides the t13 and t85, I've got some t84s, t88s, a m168 & some 3

sebi April 8, 2020 at 9:58 PM

My bad: I have a t85 and PB5 is /Reset. This must be the reason why it cannot work. Sorry.

Reply

sebi April 8, 2020 at 9:57 PM

This comment has been removed by the author.

Reply

Unknown November 26, 2020 at 12:22 PM

80 bytes TRx is very impressive!
I've written my own SoftwareUART back in 2017 [1] using clobbers and assembler bindings from C, needs 120bytes for both TRx.

[1] https://github.com/mihaigalos/Drivers/tree/master/AVR/SoftwareUart

Reply

Unknown February 3, 2022 at 10:43 AM

Wanted to share with you a 64 byte implementation for Software UART using clock scaling.
I guess this can be improved, drop me a line if you like or disapprove of it!

https://github.com/mihaigalos/Drivers/tree/master/AVR/SoftwareUart_v2

Reply

Newer Post Home

Subscribe to: Post Comments (Atom)

Simple theme. Powered by Blogger.