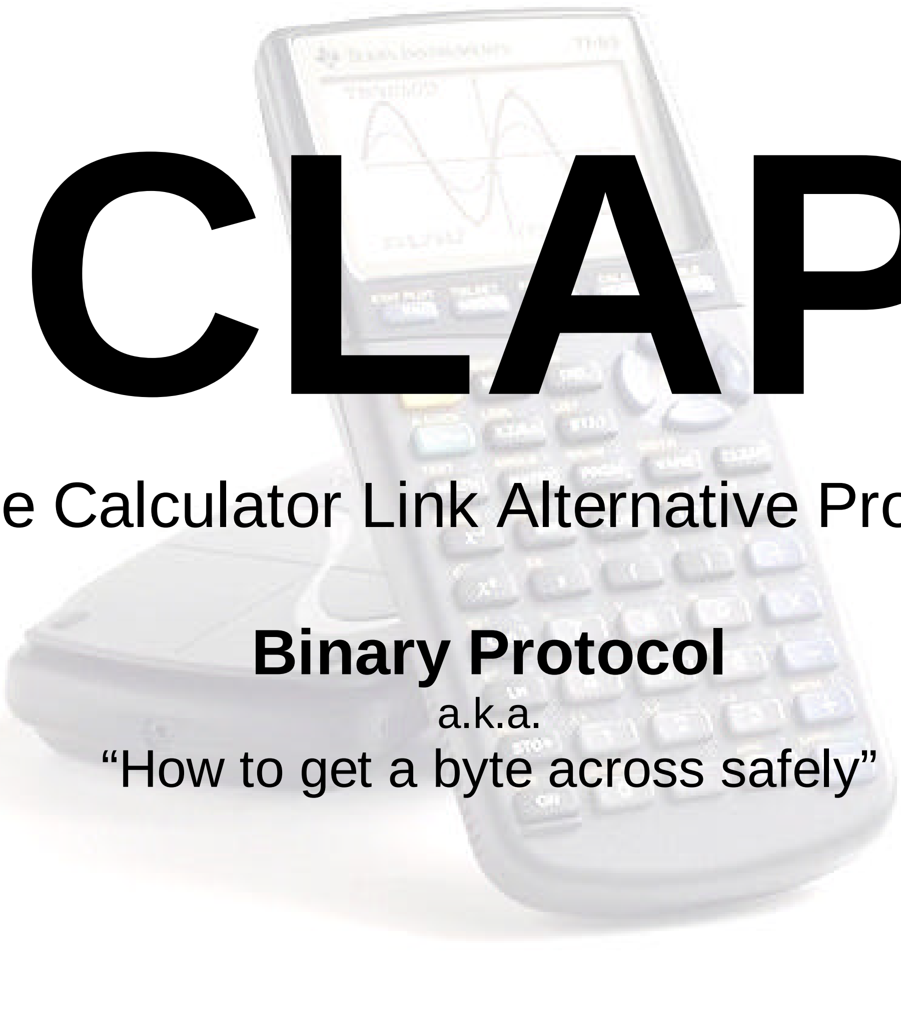


CLAP

The Calculator Link Alternative Protocol

Binary Protocol

a.k.a.

“How to get a byte across safely”

Basic Knowledge

Texas Instruments uses an interrupt routine to check for transfers in the background. This is what enables silent transfers on those calculators that support it. Knowing this is very important, because it influences the way the link port behaves significantly. And if we ever get to background processes taking care of the dataflow, then we need to keep an eye on the interrupt part of this as well.

```
"The link port normally operates in a half-duplex mode where a bit is
sent by activating the corresponding line ("ring" or "tip") and the
receiver acknowledges by activating the other line. The sender now
releases its line and finally the receiver releases the acknowledge."
```

This means that if I pull one line down, the calculator will – what is known as – 'freeze'. The interrupt will think it's receiving data from somewhere, acknowledge the received bit, and start waiting for more to come.

Something else that's really important to know about the link port is that it's a logical or gate. If two calculators are connected, then either one of them can pull a line low, and it will remain low no matter what the other calculator does. Both calculators have to keep a line set as high for it to be high (and to be read as such by both).
I expect this to work for more than two calculators as well. It should logically become "A or B or C", so any one of the three calculators can pull a line low, and all three have to release it for it to be high. This would mean that with send and receive routines that do not use acknowledgement or anything else going from receiver to sender, it should be possible to send a byte to more than one receiver at a time. The hardware to link those calculators could be very simple; connect all the tips, all the rings and all the bases, and you'd theoretically have a calculator network, if it wasn't for the lack of software :).

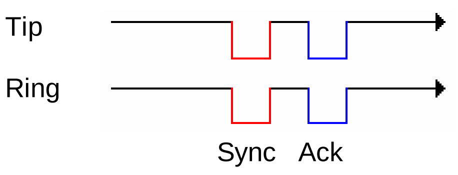
Bit Protocol

With this in mind, I started devising a binary protocol with the sole purpose of enabling two calculators to send over a byte of data, without triggering the Ti-OS interrupt and with quite a lot of fail-safes. I plan on making two versions of the software; one for linking two calculators, and one for the final networking software, if I ever get that far. I want this because the network routines will have less error checking (since receivers cannot reply to a sender) and I don't want the calc to calc routines, that will probably be used more often, to suffer under those restrictions.

I want the two calculators to be able to wait for each other if necessary and if desired, and to check if the other is listening at all. I want a simple form of error checking in the bit stream, and the routines should be able to be used to send over a stream of bytes.

So the first thing we'll need is a synchronize routine. This routine should return an error when it is not linked to another calculator, or when it is linked to another calculator, but this calculator is not able to participate in the byte transfer. Since we do not want the Ti-OS to interfere, there's only one affirmative sync response possible:

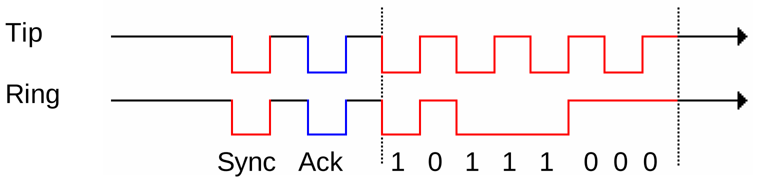
pulling both lines low. It's only natural to make the sync request the same. So a proper synchronization between two calculators, as seen by an observer, should look like this:



Where one calculator is represented with the colour red, and the other with blue.

Since the Ti-OS only responds when one of both lines is pulled low, and never responds by pulling both lines low anyway, that shouldn't be able to interfere. And when there is no calculator connected, there will also be no response. So when the synchronizing calculator (red) reads an acknowledge (blue), it knows that there's a proper program on the other end of the line.

Then, the next thing we want to do is send over a byte. However we do this, we must not forget to turn off interrupts first, to keep the Ti-OS from getting a say in it after all. There are various ways to get a byte across, of course. The fastest way to do this with two lines is probably to use one as a clock, and the other as a data line. The lines would then have to do this:

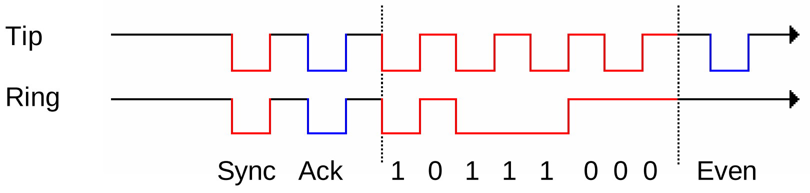


The first calc, indicated with red, tries to synchronize, gets an acknowledge, and sends out the byte 29 (which is %00011101 in binary). One bit of the byte is sent out on every change of the clock, so after pulling the clock (the tip) low for four times, we have sent a byte over.

But we want these routines to be secure, so they have to give an acknowledgement if the byte was received alright. And if we have to give an acknowledge anyway, why not use it for a simple checksum?

TR
10 = Even number of ones in the bitstream
01 = Odd number of ones in the bitstream

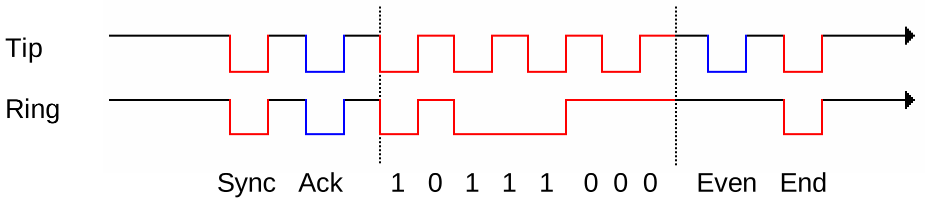
We add this to the picture:



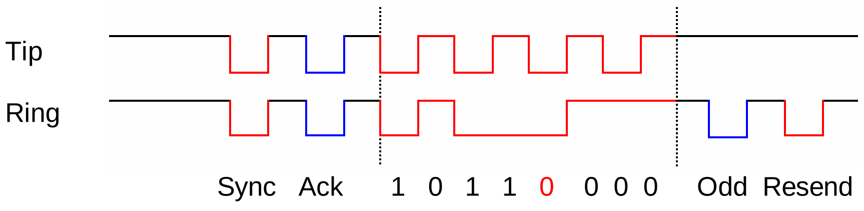
Okay, we're almost there, just a few more problems. If the sender sends out 29, and gets an "Odd" acknowledgment or if it doesn't get an acknowledge at all, it can see that something went wrong, but it can't do anything about it. Another problem is that we have to synchronize for every byte we want to send this way, which is a waste of precious time. Let's solve both issues at once.

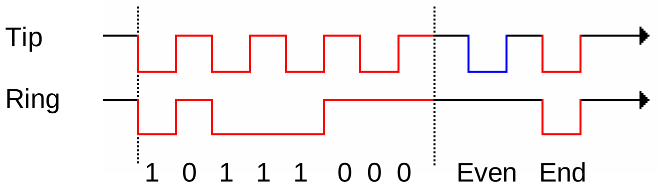
TR
11 = Send went Ok, end of stream
10 = Send went Ok, next byte
01 = Send went wrong, resending
00 = Send went wrong, aborting

Now, let's send over 29 again:

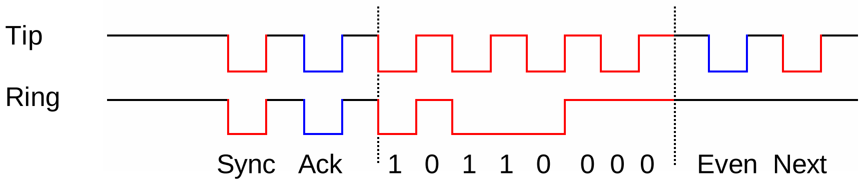


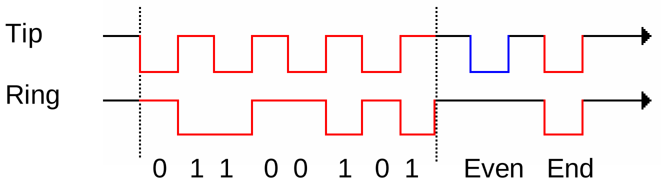
Or, when bit 4 goes wrong the first time, for whatever reason:





Now we can also send over the numbers 29 and 166 at once:

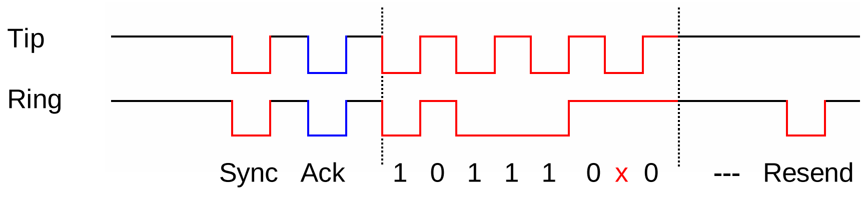


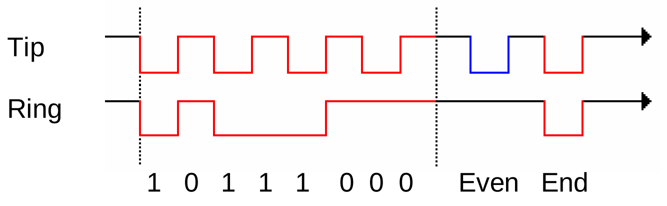


With all this, we've got a pretty stable protocol for sending over bytes or streams of bytes. Now we just have to make sure we implement this in a 'safe' way. For instance we have to make sure that there are no endless loops in the software that a program can get stuck in. Every loop that waits for input from outside **must** have a timeout mechanism.

When some bits are lost completely, the receive routine must timeout on the missing bits, and start waiting for a "Resend". If this doesn't come either, it has to have another timeout and go back to wait for a new "Sync".

The send routine shouldn't wait for the acknowledgement of the byte indefinitely either. If it doesn't come quickly enough, it should resend it, and wait for a response. If there is still no reply from the other calculator, it goes back to trying to synchronize.


Tip
Ring
Sync
Ack
1 0 1 1 1 0 x 0
--- Resend


Tip
Ring
1 0 1 1 1 0 0 0
Even
End