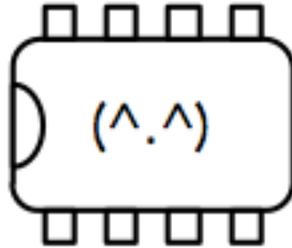
Spanda Revisited ch1



www.ZENMCU.com

Draft 1 2023-01-22

Copyright © 2023 by Trenton Henry, All rights reserved

Spanda

This is based on what seems to be the most suitable concept of all of the Spanda ideas I wrote circa 04/20. I went through a lot of concepts trying to maximize reliability and bus utilization with a single protocol on wired or infrared UART links. Having reviewed all of my notes 01/22/22 I have redesigned one of the earlier concepts slightly.

The main changes are switching from 8-bit to 9-bit bytes, and adding the repeated, inverted, payload byte. The former provides a control/data bit, and the latter is intended to improve reliability when used with infrared.

First, the maximum payload of a packet is a byte, or a character. The reason is that Spanda devices communicate via x4th. I.e, the base sends x4th 'commands' to the remotes, which respond in the standard x4th manner. Spanda cards are x4th devices, so they are cross compiled, and have no on chip dictionaries. So for the most part I want to send characters reliably.

SPANDA - A protocol for delivering ascii characters in a wired or wireless multidrop bus topology network.

Spanda is an experimental minimalist communications protocol intended to be used for both UART, and UART+I/R. Spanda is primarily intended to work for UART to UART communications for up to 16 MCUs, either connected via wires, or wirelessly via infrared. The goal is to employ the same protocol over wireless and wired transports, reasonably efficiently and reasonably reliably, while keeping the code as small as practical. It isn't intended for high throughput or large transfers. Instead it is intended for infrequent small transfers of ascii data, such as for a console/terminal/repl.

Spanda is a multidrop protocol. There is one base and multiple remotes on a common bus. The base polls the remotes, and remotes do not transmit unless polled. This reduces contention for the bus, and reduces the frame size since less control information is required.

The base periodically contacts each remote by transmitting a packet and awaiting a response. If the base has no payload pending for a given remote then it sends a null-poll. if the base has a payload pending for a given remote then it sends a data-poll. If the addressed remote has no payload pending then it responds with a null-response. if the addressed remote does have a payload pending then it responds with a data-response.

If the base delivers a poll to a remote but the remote does not respond within the 'turn around' time then the base will retry the last packet to that remote up to max-retry times. A receiver that has no buffer space does not reply. This can happen if the remote is busy with other activities and has not consumed a previously received payload.

Packet Format (New)

Packets consist of 1, 2 or 3 9-bit bytes. The first byte is called the header, the second byte (if present) is the payload, and the 3rd byte is the inverse of the payload.

(Should the header always be repeated inverted also? NP~NP DP~DP-P~P ?)

UART framing and parity are used to improve improve reliability and detect errors.

Payloads are 8 bits, and the 9th bit is the control/data bit.

Bit-9 is the control bit. Header bytes always have the control bit set, and payloads always have the control bit clear.

Bits 8 through 6 are the packet ID / pid. Bit-8 indicates the direction, where 0 is a poll from the base to a remote, and 1 is a response from a remote to the base. Bits 7 and 6 are the token. Null pid tokens have no payload, data

Bit-5 is the toggle bit, which is inverted each time the base transmits a new packet to a remote (the toggle is per remote, not global). The address is the identifier the of remotes, where address 0 is the discovery address, and address 15 is the broadcast address.

```
[ ctrl(1) | pid(3) | tog(1) | adr(4) ]  [ data(1) | pay(8) ]  [ data(1) | ~pay(8) ]

pids
null-poll 1 000 t aaaa
data-poll 1 001 t aaaa 0 pppppppp 0 ~pppppppp
?           010
?           011
null-resp 1 100 t aaaa
data-resp 1 101 t aaaa 0 pppppppp 0 ~pppppppp
nack-resp 1 110 t aaaa
?           111
```

(beacon pid? error pid?)

The base sends only null-poll and data-poll. The remotes send only null-resp and data-resp, and nak-resp. Each response contains the address and toggle of the poll that solicited it.

Synchronization

Each poll must be answered by a resp. If the base does not receive a resp after ***t-turnaround*** it retransmits the last packet up to **max-retry** times. If the remote fails to respond after ***max-retry*** tries the base considers the remote to be offline.

If the remote does not receive a well formed poll then it does not respond. If the remote does not receive a well formed poll for longer than ***t-keep-live*** then it considers the base offline. If the remote receives a well formed poll whose ***toggle*** bit has not changed since the previous poll then it retransmits its most recent response.

Flow Control

If the remote does not have space to land a payload byte then it responds with nack-resp to its own address. Negative acknowledgement is the indication of receiver-not-ready.

Base Memory Requirements

- Must store toggle bit per remote.
- Must keep last poll packet to each remote until the remote responds to it, or until the remote is declared offline.
- Must keep a timeout timer for the most recent packet sent (to any address; i.e., one timer).
- Count of timed-out responses, for analysis.
- Baudrate (possibly per remote if I/R uses rate adaptation)

Remote Memory Requirements

- Must store toggle bit.
- Must always respond to poll with matching toggle.
- Must keep last resp packet until it receives another poll (toggled).

Toggles

Normal sequence, no data.

NP0 NR0 NP1 NR1 ...

Normal sequence, data from remote, base polls until remote sends null response.

NP0 DR0-P~P NP1 DR1-P~P ... NPt NRt

Remote does not receive, or base does not receive response: base retransmits

NP0 (timeout) NP0 DR0-P~P ...

Example: dup

DP0-d~d NR0 DP1-u~u NR1 DP0-p~p NR0 DP1-\n~\n NR1 NP0 DR0-o~o NP1 DR1-k~k NP0 DR0-\n~\n

Discovery & Address Assignment * FIXME SOLVE THIS *****

All remotes power on at address 0, 300 baud?

All remotes power on at address 0, random (from table) baud? How to pick random?

All remotes are added one at a time, assigned an address which is stored in flash?

(Using pins for address bits is a waste of pins.)

(Daisy chaining won't work for I/R and slows communications.)

Tools

- IRShark Protocol analyzer: An infrared UART <--> USB listener. Receive only on

UART, send all received bytes up USB. display color coded trace.

- zterm looks like a chat room. remote 2 arrived. >2 .s >4 dup remote 2 departed. >all go. probably the logic to poll devices is managed on the host? it could be in the fw, but not sure there's any point.

Future

- Some type of beacon
- Concept of a frame?
- Method of adjusting baud based on comm fail
- Method of scanning for baud or autobaud
- Heuristic baud rate adaptation per node

Board Design

I've worked out the Spanda bus pinout and documented various connectors etc. I've created a Spanda Eagle device to quick reuse. And I've tentatively laid out base0001 to get a feel for it.

Tentatively, base0001a has:

- SAMD11C.
- HIF USB.
- LDO powered by USB VBUS.
- 3 Spanda slots, arranged for vertical cards.
- Does not have a reset button. All resets are connected so if a debugger on any MCU resets the MCU then all of the MCUs will reset.
- Bit bang SPI OLED display.

Notes:

- I made the spanda cards 4 pins (0.4") apart to give room between them in case some have bulky components.
- I use the 5 spare pins to bitbang a 64x48 SSD1306 OLED because I have some on hand.
- I don't have any pins left for buttons.

And I am considering:

- Some LEDs, red for RX, blue for TX?
- A board rev with a 20 pin MCU variant to have extra pins for buttons.
- Which way to orient the Spanda slots? For some projects like the social robots I

expect to want vertical spanda cards, possibly some below and some above the base board. In that case I will want the slots closer together, possibly alternating bottom, top, bottom, top ... But for other projects the Spanda cards may want to lie flat on top, like hats for RPi zero do. The search for the ultimate form factor goes on.

- Maybe the motor card pins should look like a 10 pin row, so motor pins can plug into dedicated motor card slot on robot base?

Tentatively spnd0002a has:

- SAMD10C
- 3 spare pins
- HIF UART-SPANDA
- 2 Si9986 motor drivers
- 1.27mm 5 pin SWD
- **ERROR spnd0002a need to move cap closer to MCU**

Questions:

- Do I want a reset button?
- Do I really need a reset signal?
- Do I really need VBAT or can boards that need it have their own header?
- Is the combined resets for motherboard and all slots the right thing?
- Should I use I2C GPIO expanders to get more pins for buttons and leds?
- Should I design a base board with a SAMD21E for more pins and memory?
- How to separate VBATT from VBUS? Select which one goes to which slots?

Appendix - Why not I2C?

I like I2C quite a bit. It is a fairly simple protocol, allows multi-master, each bit is acknowledged, its only two pins, you can have many devices on one bus, its easy to test out I2C devices by writing some Python code on RPi, or other platform, and putting it through its paces. All of that is great.

But if you want to have two or more identical devices on one bus, and those devices do not have selectable addresses, then you are out of luck. And I seem to encounter that problem frequently. So just use a different MCU with two I2C busses. Yes, that's an option. But for the SAMD1x family at least, that means a larger package, or a QFN-24 which, so far, I have failed to solder successfully by hand. (And yes, I have purchased a reflow hot-plate, which should arrive in a few days, and I will try again. But until I can get game with these types of packages the SOIC style package is all I can manage easily.)

Additionally, though this isn't any different than any other bus really, every I2C device

works slightly differently. Yes you read and write registers, but its slightly different for every IC. And that's fine, really. But it is a little tedious that for every device you want to play with you have to read the datasheet (and often they are terrible) in detail and write some firmware to get the job done. Sometimes someone already has reference code to consult, and that helps. But it is a factor that slows down the integration of new (to me) I2C devices.

But the main problem is that if you want to use multiple different I2C devices you need multiple drivers, each taking up code and data space. And since I'm still requiring myself to live with 8K instructions and 4K bytes of RAM, that can be costly.

So ... UART? Whu? Ok, I can explain. And if somehow I can't, then I need to rethink.

TTL level UART, if you only use software flow control, is just two pins. And, compared to I2C, it is fast. It is full duplex, frames and parity checks each byte, and most UARTs offer a 9 bit mode, which is ... like ... a free bit. And many, though not all, support IrDA modulation. And *that* means that your get wireless over UART for the modest cost of an IrDA transceiver.

The tricky requirement I've imposed on myself is that the exact same protocol is used for wired and wireless (infrared) communications. So, yeah, it's going to be half duplex. But I can live with that; I2C is also hdx. Since the UART does faming and parity, if I only wanted wired, I'd pretty much be done by defining a "packet" as a UART frame with a one byte payload. But since I/R needs some sense of collision detection and dropped packet detection, I need my "packets" to have some redundancy.

Using the same redundancy fore wired does reduce the utilization of the link. The simplest possible scheme is to repeat every byte, inverted, and bitwise AND them. If the result is 0 then the byte is good. And yes that is indeed 50% overhead. But ... I'm not sure I care.

I suppose that for trying to send a frame buffer to a remote display that might be limiting, so I might not repeat the bytes for the wired link. But that's a detail.

But ... I still want to use all those I2C devices out there! So there's always the option to just use them. For simple projects, why not? Problem solved. But the idea of spanda is to try to build a (possibly wacky) twist on the Apple II expansion bus design.

The Apple II expansion cards each had an assigned address range, and a small ROM that mapped to that range. So the main program could "call the driver" by jumping to a routine at a fixed address in a card's ROM, with parameters in registers. (Yeah, you had to trust the ROMs...)

Spanda is the same idea; the main computer runs code on the Spanda card. But instead of actually executing the card's code itself, it does a remote procedure call. Each Spanda card runs x4th, and implements standard Forth words, as well as card specific words. So the base computer just types words into the remotes, and they type their responses back. Well, I mean, sort of. Its like that, but the Spanda cards have no dictionaries, so they use the x4th mini-terpreter. Read about x4th if you care about the fiddly bits.

Anyhow a small, dedicated, MCU can implement the device specific I2C shenanigans, and preent that as a small set of Forth words over UART. I kind of wish that the SAMD parts came in an 8 pin package so I don't feel like this is overkill. But, oh well.

So, you want a summary, I can tell. Here it is:

Capability	Spanda (multidrop UART)	I2C
multi-master	no	yes
multi-device	yes (with diodes)	yes
full duplex	yes (wired) no (irda)	no
hardware flow control	no	yes
high baud rate	yes (wired) meh (irda)	no
2 pins + gnd (+ vcc)	yes (qwuuc)	yes (qwiic)

Also, for what it's worth, there are tons of USB to TTL UART bridges available, so you can talk Spanda to them from a PC if you want. And that means you can use picocom for simple things, or you can write termios code to do fancier things. It just seems more versatile. You may disagree; and I might change my mind eventually. But for now, I'm going to keep going.

Appendix - Random Whinging

The neat thing about the 57/58 series boards is that it is easy to make shields for them.

I'd like to reduce the size of the boards, and simplify standardize my pinout.

Going to 1.27 mm pins would help. But then I lose the ability to breadboard.

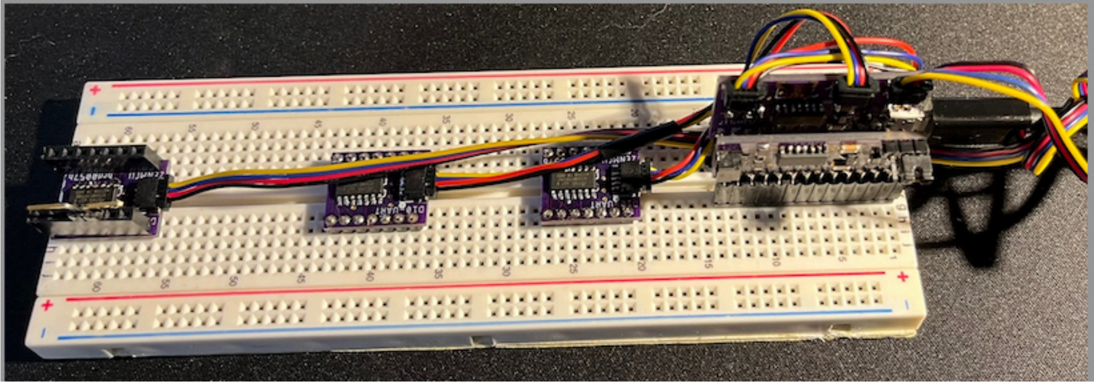
So I would need a 1.27 mm to 0.1 " dapta card.

Such cards could have various connectors or sensors etc.

So you solder a sensor to the bottom of the MCU card and get a sammich...

Update 01/29/23 So I spent way too much time designing possible board variants, thinking about what flavor of xforth should be on the Spanda cards, lots of thinking and writing notes about the right way to do all of this, etc. I experimented with the 0wire RGB LEDs from Sparkfun, but they need 5V so my 3V system couldn't light the blue LED well at all, etc. My solder reflow hot-plate arrived and I reorganized my "lab" to accommodate it. Then I designed Spanda cards again, and then I patched up a brd0035c (which had layout issues) to get 3 of the 4 downstream UART ports working. Then i built up 3 brd0057b's bit I bypassed the LDO and just routed VCC from the QWUUC connector to V3P3 since brd0035c feeds the remotes 3.3V, making the LDO pointless. And it simplifies the builds, gets closer to a real Spanda card, and lets me experiment with supplying 4 MCUs off one MCP1700. And, or course, start trying to work out the FW for the protocol.

(Yes that's a brd0034 in front of the brd0034. Its not hooked up to the Spanda bus though because one of the 0035's downstream ports was borked on the PCB ... because I was an idiot when I designed it...)



End