Machina Speculatrix

Member-only story Featured

ROBOTICS

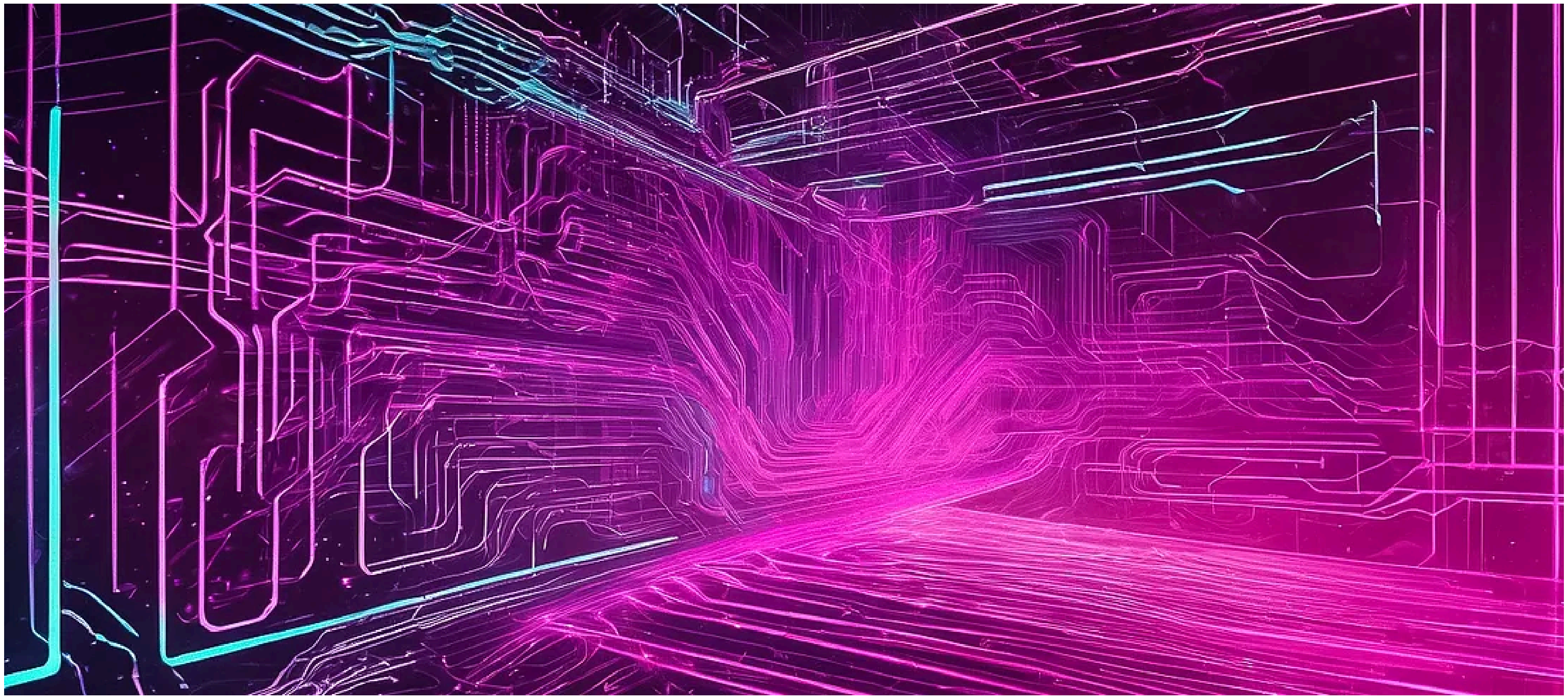
Inventing a serial protocol for smart sensors

We need to talk. And when I say 'we', I mean smart sensor devices and their microcontroller overlords.

Mansfield-Devine Following

14 min read · 6 days ago

270 5


AI-generated conceptual illustration. All a bit silly, really.

There's a saying in the cyber-security world: 'Don't roll your own crypto'. It's actually more than a saying, it's a warning. And the underlying message is simple: there are existing encryption algorithms and libraries that were created by people much smarter than you and that have been tested and proven to work. If you do it yourself, you'll just screw it up in ways you won't be able to predict and probably won't understand.

We'll quickly skate over things like MD5 and SHA-1 that later turned out to be flawed because the principle is a sound one. Creating your own crypto solutions is unnecessary, dangerous, wasteful and likely to lead you into a world of pain.

And much the same goes for less critical technologies. With serial communications protocols, for example, we're spoiled for choice. Serial signals of the RS-232/UART variety are everywhere, and have been for a long time. There's I2C, SPI (so easy to bit-bang), USB and more. All you have to do is pick the right one.

So why did I roll my own? Simple.

Unreasonable reasons

The main reason I wanted to invent my own protocol (with both 'invent' and 'protocol' overstating the case somewhat) is that I wanted to.

Having decided to imbue simple sensors with an element of intelligence and autonomy as part of a nascent robotics project, this raised the question of how the 'smart' sensors would talk to other devices. And so I came up with a method that can be implemented on anything with GPIO pins (because it's bit-banged) and doesn't require exotic hardware. I'm calling the resulting protocol 'SensorBus'.

Heaven knows it’s not about speed. Nothing I can build is likely to be anywhere near as fast as existing technologies.

And it’s not about being clever. There’s nothing ingenious or cunning here.

So why? Because it’s mine. And because it’s fun. And along the way it taught me how tricky a task this can be and gave some insight into why certain existing protocols work the way they do.

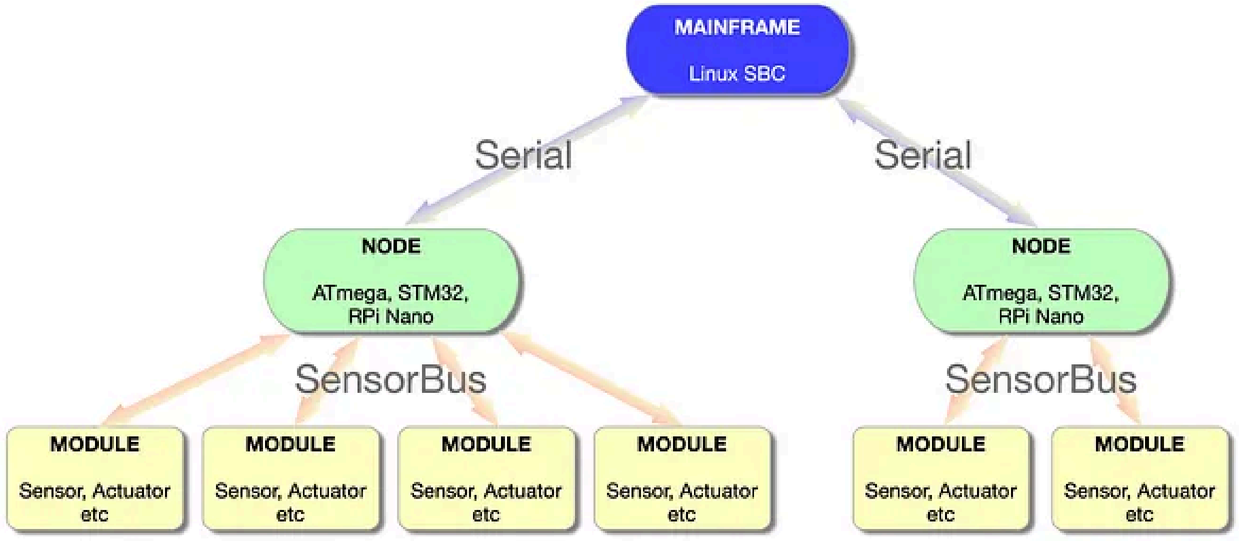
The big picture

So here’s the big idea. A sensor or subsystem module of some kind (let’s just say ‘module’ from now on the keep things simple) has its own microcontroller.

Most of the functions of the module are managed by the MCU, including decisions based on what it’s sensing or doing. But every now and then, according to certain criteria, the module needs to kick some information up the chain of command to a higher level system, known as a node, which may be running on a more sophisticated microcontroller or a single-board computer. That, in turn, might escalate to a higher power. (I’m calling this, in a fit of grandiosity, the ‘mainframe’. Indulge me.)

In addition, the node computer might want to issue instructions to the module. These could include commands like 'go forward' or 'stop' sent to a motor controller module, or it might be a new set of parameters or configuration settings sent to a sensor package.

The point is, either side needs to be able to start a conversation.



The overall architecture.

How many wires?

A key question in designing any protocol like this is how many signals do you need? (You'll have to have a common ground connection between both ends, but we won't count that as a signal.)

Serial comms (RS232 and its ilk) has just two — TX and RX. But there's also significant complication involving setting baud rates, initiating comms etc. So it's not as simple as it looks.

I2C (sometimes known as a 'two-wire' protocol) has, well, two. But it uses a controller/peripheral architecture in which it's always the controller that initiates actions. SPI has four wires (three common to all peripherals, one dedicated to a specific device). So how many — or more pertinently, how few — could I get away with?

I went through many iterations of this and eventually landed on a three-wire solution. The signals are:

- `SB_CLK` — a clock.
- `SB_ACT` — bus active.
- `SB_DAT` — data (also referred to as `DAT` sometimes in my code because I went back and forth on this. As we know, naming things is the hardest part of programming).

The first two signals, `SB_CLK` and `SB_ACT` are shared between all modules connected to a node. But every module has a dedicated `SB_DAT` line, which means that the node might have eight or perhaps 16 such lines.

The `SB_DAT` and `SB_ACT` lines are active low and require pull-ups.

Power supply

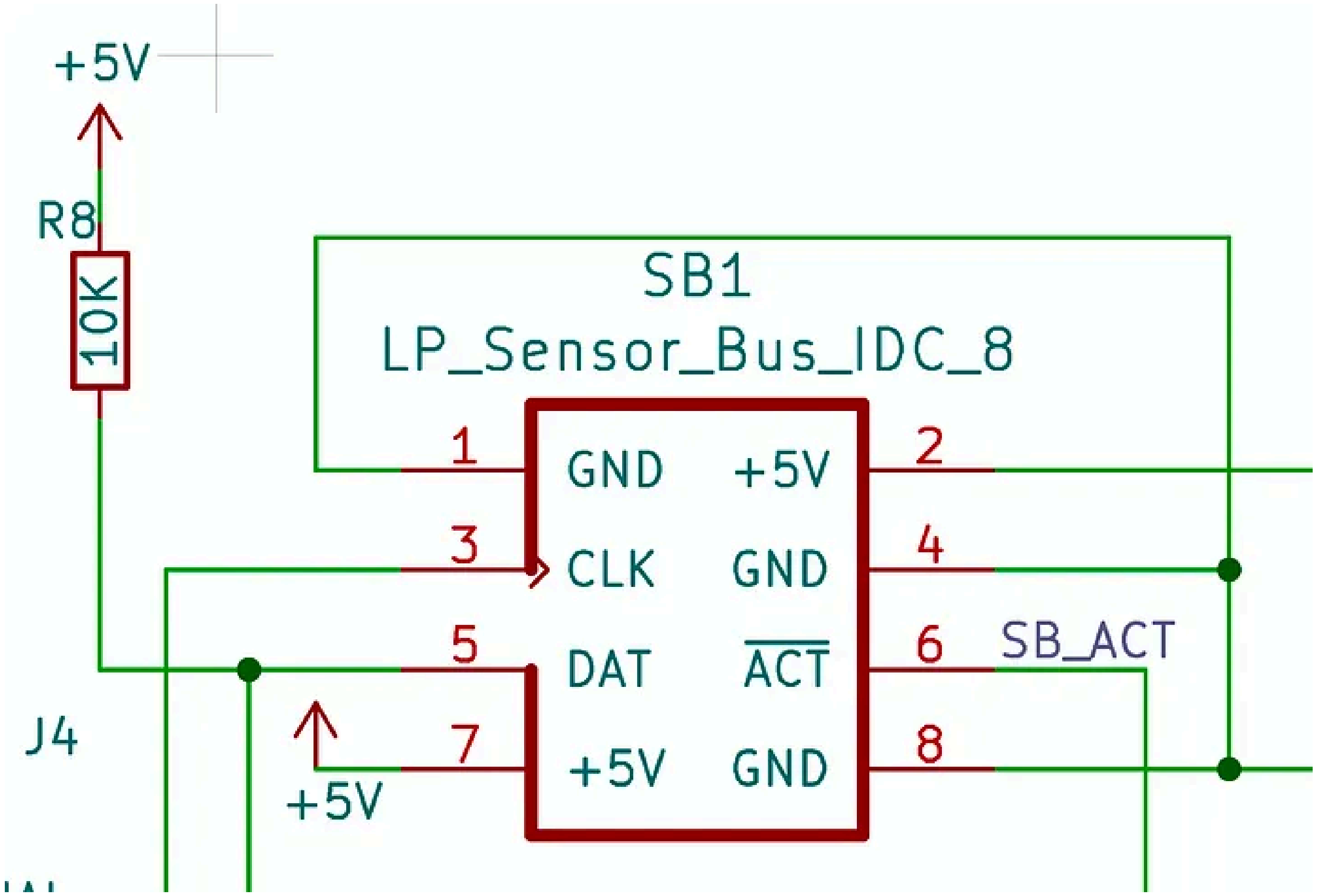
The SensorBus signals will operate at 5V and any module that has components that require other voltages will have to take care of that locally. So when considering the SensorBus cable and connector I chose to throw in power supply, too.

There are two +5V lines to ensure adequate power capacity (I think. I mean, it can’t hurt, right?). Each has its own GND return. On the connector, the +5V and GND pins are arranged so that if you (ok, I) plug in the cable backwards, the power connections will still be fine although the signals will be messed up. I did this because there are likely to be occasions when space on a PCB is at a premium and I don’t want to use a shrouded connector.

The three signal lines, plus two pairs of power lines adds up to seven, according to my calculator. So I threw in an extra GND line to make a even eight for the IDC-style 2x4 pin header on which I’d already set my heart.

When attached to a ribbon cable, you have power lines on the outer four wires and the extra ground in the middle.

```
IDC HEADER               CABLE
 +------------------+      1. GND
 | •1 GND   +5V 2• |      2. +5V
   •3 CLK   GND 4• |      3. CLK
   •5 /DAT /ACT 6• |      4. GND
 | •7 +5V   GND 8• |      5. /DAT
 +------------------+      6. /ACT
                           7. +5V
                           8. GND
```



Pinout of the SensorBus header as seen in a schematic.

Theory of operation

Essentially, the process is the same regardless of whether the device sending a message is a module or a node. So, in this context, we'll refer instead to a 'Sender' and a 'Receiver'.

The basic process is this.

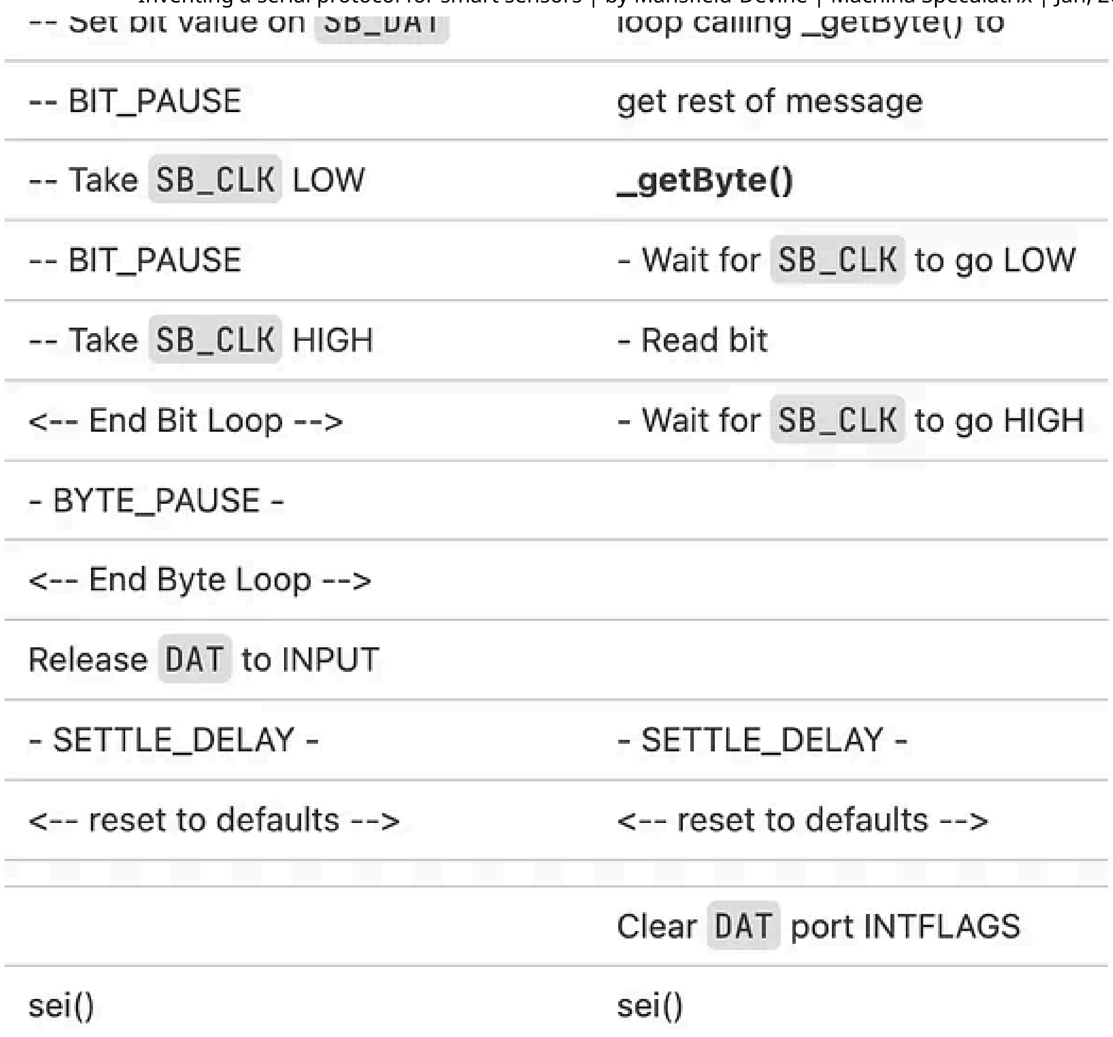
By default (and in the idle state) all SensorBus pins on both Sender and Receiver are configured as inputs, so that they are effectively high impedance. And all `SB_DAT` pins are configured to generate interrupts on a falling edge.

- The Sender checks the `SB_ACT` line to see if the bus is busy. If not, we proceed.
- The Sender takes the `SB_ACT` low to take control of the bus. (And yes, I'm aware of the possibility of race conditions here. We'll pretend those don't exist. And I've already implemented one mitigation, as we'll see later.)
- The Sender strobes the `SB_DAT` line low. This wakes up the Receiver, which goes into receive mode.
- Once the Sender's strobe is finished, the Receiver also strobes the `SB_DAT` line low. As there is a unique link between the two devices over this line, this confirms to the Sender that the Receiver has seen the request and (in the case of the Receiver being a node) correctly identified which device is about to send a message.
- The message bytes get sent, using the falling edge of the `SB_CLK` line to signal when each bit is stable and ready to be read.

The first byte contains the full length of the message, so the receiver knows how many bytes it needs to clock in. The second byte I'm reserving for a 'message type' identifier. So a message always has a minimum length of two bytes. The rest is variable depending on the activity.

Here's a more complicated breakdown for those who have nothing better to do.

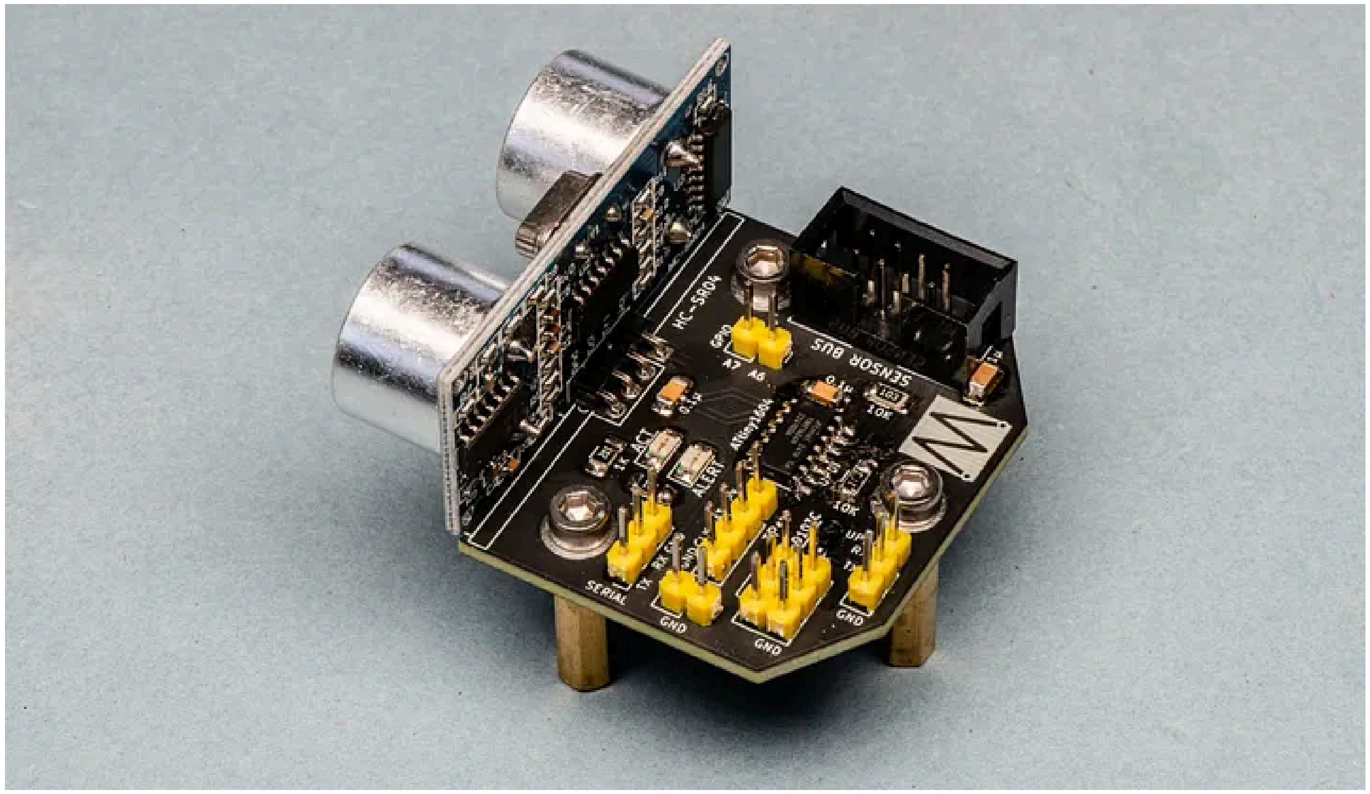
SENDER	RECEIVER
cli()	
_setSendMode()	
Wait for `SB_ACT` to be high	
Set `SB_CLK` to OUTPUT, HIGH	
Set `SB_ACT` to OUTPUT, LOW	cli()
Set `DAT` to OUTPUT, HIGH	
Strobe `DAT` LOW	**_setReceiveMode()**
Set `DAT` to INPUT	- Wait for `DAT` to be HIGH
Wait for `DAT` ack strobe	-Set `DAT` OUTPUT, HIGH
Set `DAT` to OUTPUT, HIGH	- ACK_PAUSE -
	-Strobe `DAT` LOW
	-Set `DAT` to INPUT
sendMessage()	**recvMessage()**
- START_TRANSMISSION_PAUSE -	
<-- Byte Loop -->	
<-- Bit loop for each byte -->	run _getByte() to get first
- For each bit in byte (8 times)	byte containing msglen, then
- Set bit value on `SB_DAT`	loop calling _getByte() to



-- Set bit value on `SB_DAT`	loop calling _getByte() to
-- BIT_PAUSE	get rest of message
-- Take `SB_CLK` LOW	**_getByte()**
-- BIT_PAUSE	- Wait for `SB_CLK` to go LOW
-- Take `SB_CLK` HIGH	- Read bit
<-- End Bit Loop -->	- Wait for `SB_CLK` to go HIGH
- BYTE_PAUSE -	
<-- End Byte Loop -->	
Release `DAT` to INPUT	
- SETTLE_DELAY -	- SETTLE_DELAY -
<-- reset to defaults -->	<-- reset to defaults -->
	Clear `DAT` port INTFLAGS
sei()	sei()

Implementation details

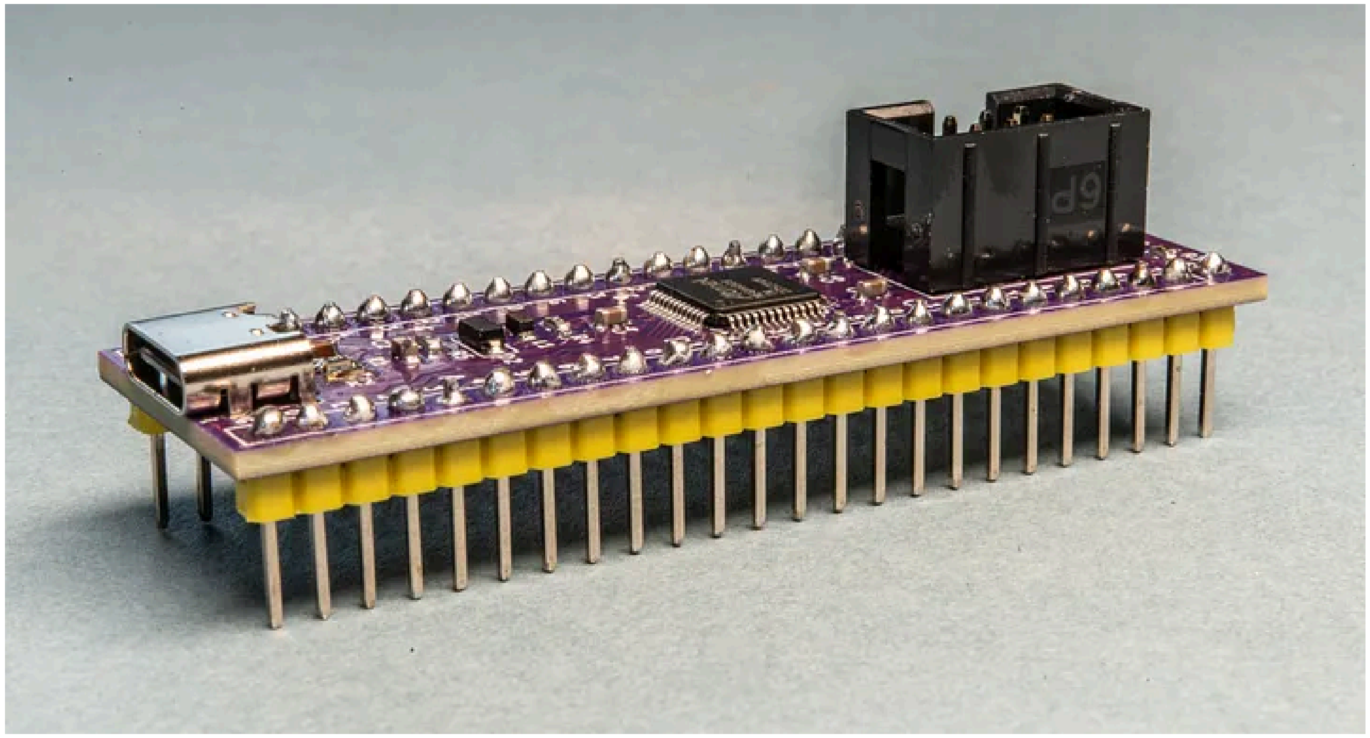
Currently, the code is all based on AVR 0-Series microcontrollers. My test rig consists of a single module and a generic node both using MCUs running at 20MHz (their default speeds).



The ultrasonic rangefinder module.

Both of these are using PCBs I designed and which were kindly provided for free by PCBWay. The module, built around an ATtiny1604, is a custom board for using the HC-SR04 ultrasonic rangefinder. The node uses a simple breakout board for the ATmega4809. For this, I had PCBWay do all the assembly of surface-mount parts, and am mightily glad I did so.

I can easily envision nodes being based on the STM32, too. Although that family of microcontrollers runs at 3.3V, many of the pins are 5V-tolerant, so that shouldn't be too much of a complication.


The ATmega4809 breakout board.

A matter of class

As an unrepentant fan of object-oriented programming, the code uses a number of classes.

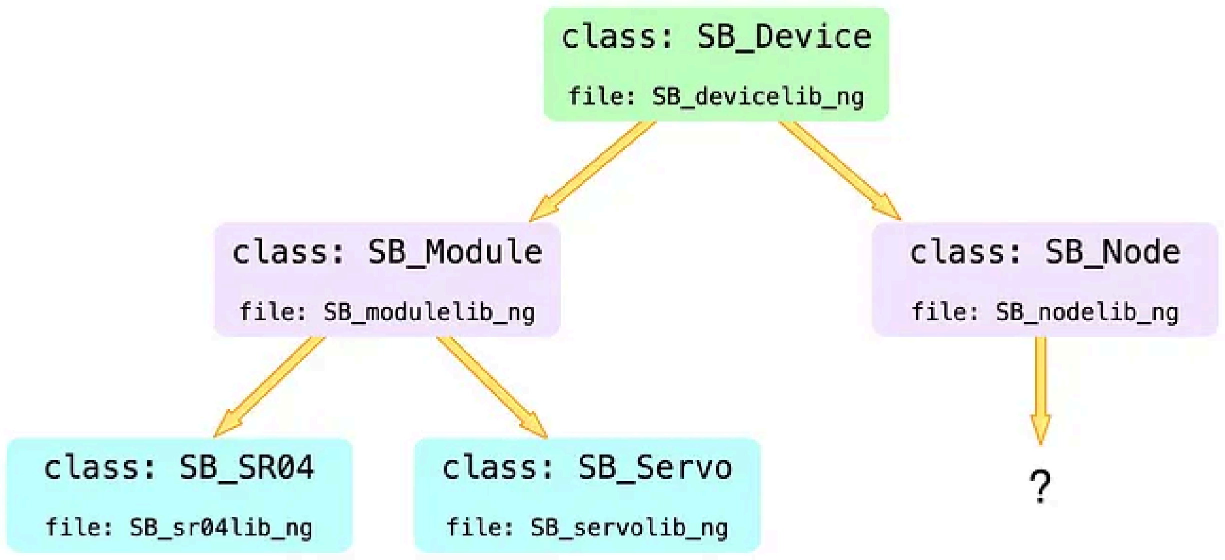
Most of the functionality is included in a base class `SB_Device` as defined in the `SB_devicelib_ng` library, if you care to look at the source code (pro tip:

select the `dev` branch for the most up-to-date code). This class works for both nodes and modules.

Two (so far) very simple child classes — `SB_Module` and `SB_Node` —add some tweaks for the specific use cases. And I have also created child classes of `SB_Module` for particular devices. So far, these are `SB_SR04` for the ultrasonic rangefinder (working) and `SB_Servo` for servo motors (in progress).

In this kind of bit-banged approach, devices spend a lot of time waiting for a signal to be in a desired state. For example, during the message transfer, the Receiver waits around for the `SB_CLK` signal to go low as a sign that it can read the bit on the `SB_DAT` line. For that reason, I included a `_waitForState()` function as part of the base class. But if all the device did was wait, things could get a little hung up. And a program freezing because it never gets the signal it's expecting leads to bugs that are hard to diagnose. So the `_waitForState()` function includes timeouts. It employs the timer `TCB0` which gets set at the beginning of the function, and then we just check regularly to see if it has overflowed.

You can find the source code for the modules and the node in the Hestia repo on GitHub.



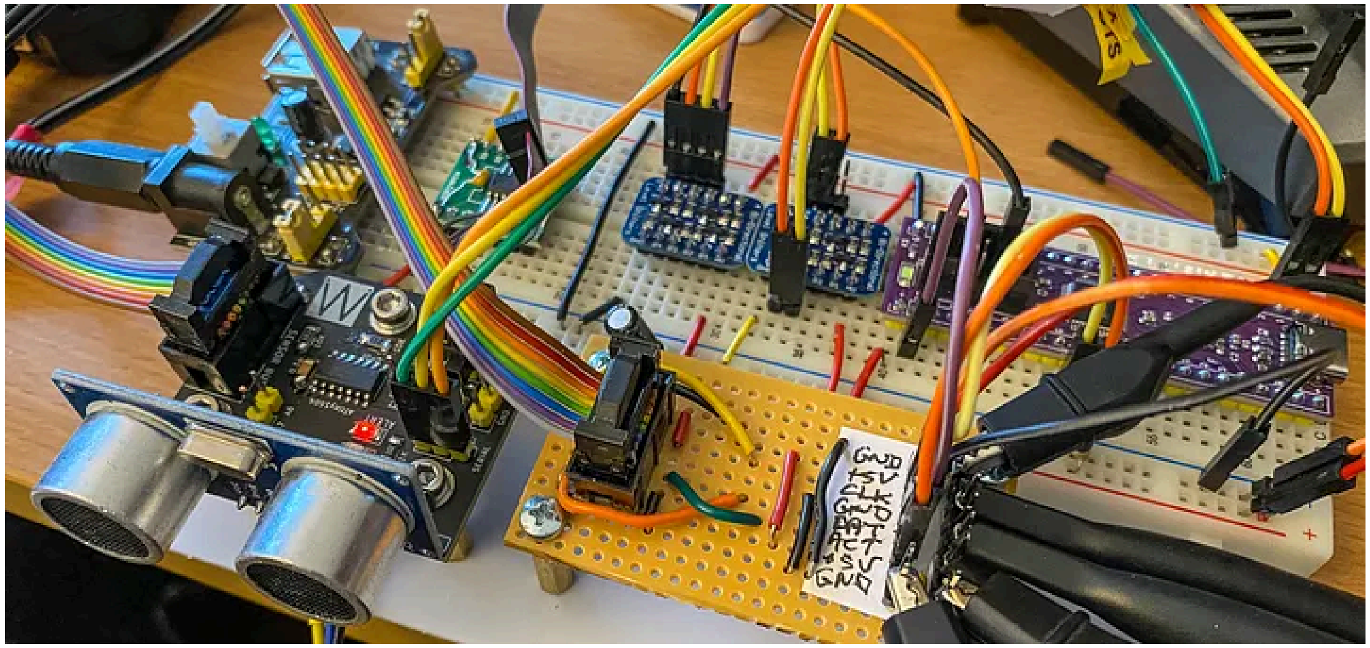
The classes used in my code. I don't yet know if there will be a reason to sub-class SB_Node.

Nagging problem

Sounds easy, doesn't it? I thought so too, until the process of getting it working changed my mind.

As a first step, I configured the module to send data once a second to the node. The data was getting through, but I had a persistent bug. The code on the node received the message fine, but then it seemed to think there was another one to be collected. There wasn't, so it would error out.

It was obvious that this had to be some form of timing error and I spent hours staring in incomprehension at the oscilloscope screen. It revealed that, for some reason, the `SB_DAT` line was remaining low.



The test rig. What could possibly go wrong? At the front, we have the SR04 module (left) connected via a SensorBus ribbon cable to stripboard where I break out the signals and have connected scope probes. In the background, on the breadboard, are (left to right): power supply; a board with an ATtiny1604 that I'm using to prototype the servo module; two blue level converter boards allowing me to connect to a Raspberry Pi (just seen creeping into the top right of the picture); and the purple ATmega4809 breakout board playing the role of a node. Photo by the author.

After every transaction, the code for sending and receiving calls a `_setDefaults()` function that, among other things, puts all the pins into input mode and sets pull-ups on the `SB_DAT` lines. But *something* was driving that line low.

In the end, the culprit turned out not to be timing but interrupts. Or to be more precise, timing *and* interrupts.

When the Sender released the `SB_DAT` at the end of the sending process, it's possible it didn't do this quite fast enough. The last bit of data sent was often a `0`, meaning that the `SB_DAT` was low and hadn't yet been pulled up when the line was released. The Receiver was seeing this as a 'tail-end' trigger and interpreting it as another communication request.

The solution was to clear interrupt flags at the end of the read process in the main app. For example, on the test node, I have `PORTD` configured as data lines. In the node app code, right at the end of the block where it reads an incoming message, and just before turning interrupts back on, I reset the interrupt flags.

Let’s run through this briefly.

The config file for the app sets up a bunch of macros that include:

```
#define MOD_SR04        PIN0_bm
#define SB_DATPORT      PORTD
#define SB_DAT_ISR_VEC  PORTD_PORT_vect
```

The `SB_Device` class has a public property used to flag when interrupts have been triggered on data lines.

```
volatile int8_t commRequestRcvd = -1;
```

In the node app's interrupt service routine for dealing with interrupts from the data lines, we set this to the value of the interrupt flags.

```
// ISR invoked when any DAT line is pulled low.
ISR(SB_DAT_ISR_VEC) {
    node.commRequestRcvd = SB_DATPORT.INTFLAGS; // set to bits triggering int
    SB_DATPORT.INTFLAGS = node.commRequestRcvd; // clear flags
}
```

And then in the main loop of the node app, we have:

```
if (node.commRequestRcvd >= 0) {
    // A module is requesting comms
    cli();    // Disable interrupts while dealing with this.
    // Get the incoming message
    SensorBus::err_code err = node.recvMessage(node.commRequestRcvd);
    if (err == ERR_NONE) {
        uint8_t device = 0;
        if ((node.commRequestRcvd > 0)
            && ((node.commRequestRcvd & (node.commRequestRcvd - 1)) == 0)) {
            device = __builtin_ctz(node.commRequestRcvd);

            // ... do stuff here ...

        } else {
            // deal with multiple flag error
    } else {
        // deal with error reading message
    }

    // When done...
    node.commRequestRcvd = -1;  // reset
    // Clear the interrupt flag one last time to remove any
    // 'tail-end' triggers
    SB_DATPORT.INTFLAGS = 0xFF;
    sei();                          // Re-enable interrupts
}
```

Note that bit at the end: `SB_DATPORT.INTFLAGS = 0xFF`. That sorted all my problems.

There are two other notable lines in that code.

Checking the bit

```
if ((node.commRequestRcvd > 0)
    && ((node.commRequestRcvd & (node.commRequestRcvd - 1)) == 0))
```

This tests to ensure that one, and only one bit is set in `node.commRequestRcvd`. If more than one is set then two interrupts fired at the same time, which I'm treating as an error.

This technique is known as the Power of Two check. Since each bit position represents a power of two (1, 2, 4, 8, etc), a byte with exactly one bit set is, by definition, a power of two. And the trick relies on how binary numbers ‘roll over’ when you subtract 1.

Let’s say only one bit is set and it’s the value 16, which in binary is `00010000`. Subtracting 1 gives you 15, or '00001111'. When you perform a bitwise AND

(&), no bits overlap, resulting in 0.

Now let's assume you have multiple bits set. For example, if the original byte value is 20 (`00010100`), then subtracting 1 gives 19 (`00010011`). The higher bits (`0001`) remain unchanged ANDing the two values results in `00010000` (16), which is not 0.

If the original value is 0, the effect of subtracting 1 could have various results depending on how the C++ you're using. So we check that this isn't the case first.

If you are using C++20 or later, there's a much easier and more efficient option. The standard library provides a function in the header:

```
#include <bit>
if (std::has_single_bit(node.commRequestRcvd)) {
    // Exactly one device is active
}
```

Alas, avr-g++ doesn't have this.

Bit value

The other handle function we use here is:

```
device = __builtin_ctz(node.commRequestRcvd);
```

I’ve defined the various devices as bit masks. Let’s take a theoretical example because it shows the effect better. We might create the macro `#define MOD_WIDGET PIN4_bm`. I can then pass these macros to instructions such as: `PORTA.DIRSET = MOD_WIDGET`. But what's the actual value of `MOD_WIDGET`? It's a byte with just bit 4 set, like this: `00010000`. The decimal value is 16.

But what if want not the value but the ‘number’ of the bit set? In other words, I want a function that will take `00010000` and return the decimal value 4. That's what `__builtin_ctz()` does. And it does it by finding the lowest set bit and then counting the number of zeros after it (hence `ctz`).

Speed. Or not.

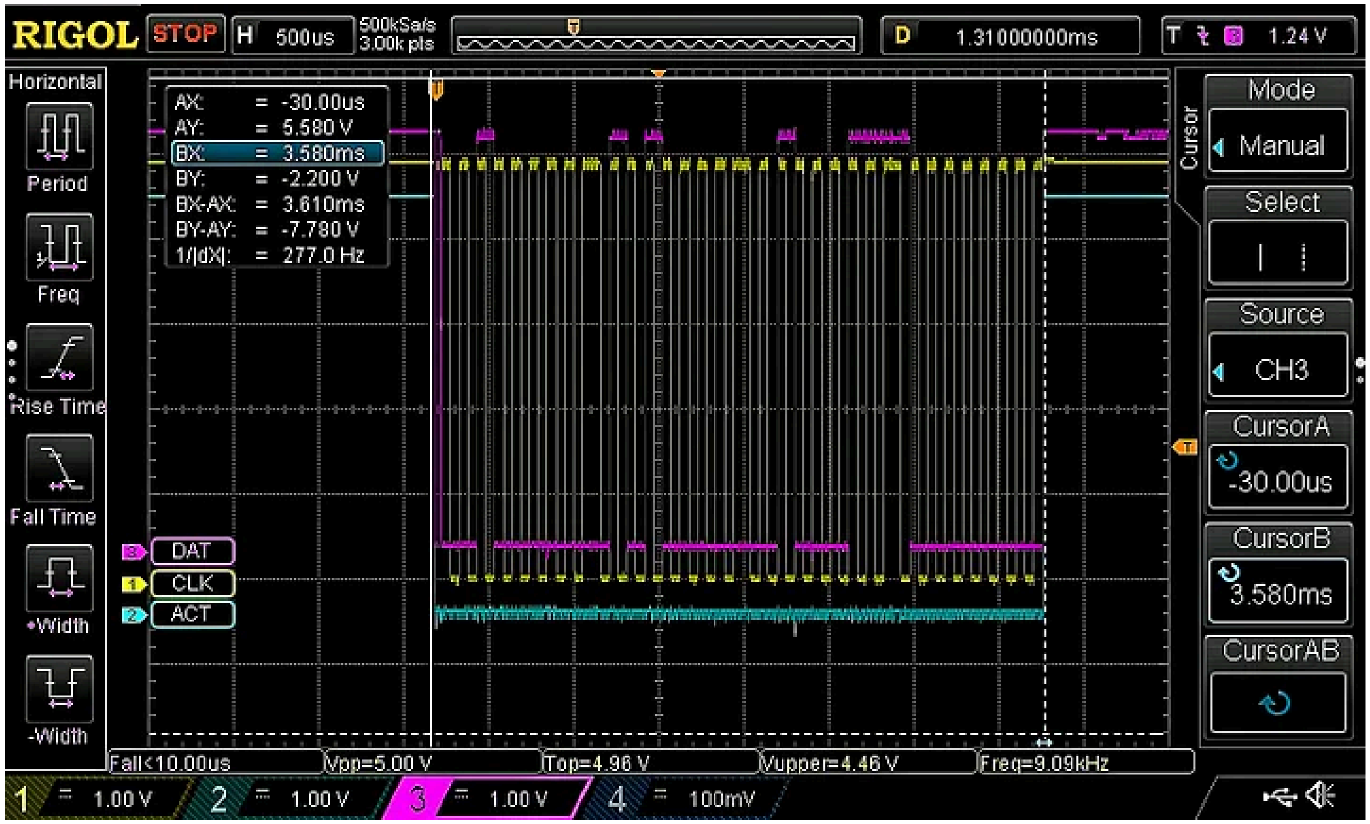
But anyway, back to SensorBus. How fast is it?

Do I have to answer that?

Oh, okay. It's not as fast as you'd think. Or quite fast, if your expectations were understandably low.

At first, I thought that sending a four-byte transaction was taking 8.6ms. But there was a weirdly long gap between the sender (the module) transmitting the initial strobe on the `DAT` line and the receiver (the node) responding in like manner. Then I realised that the node was printing a debugging message over serial between those two stages. With that action shifted to a less intrusive place, the message took just 3.6ms.

Let's do some maths. Four bytes is 32 bits. Let's say, for the sake of argument, that the initial strobe and its acknowledgement count as two more bits. That's a total of 34. Divided by 3.6, that's around 9.4 bits per millisecond or 9,400 bits per second. That'll do.



Four bytes of data in 3.6ms. Good enough for me.

The lack of any synchronisation after the first handshake strobes and before the subsequent sending/receiving of data is a simplification that could come back to bite me. Once the message exchange is set in motion on the sending side, the receiver just has to keep up. There are timeouts and error messages and, if there is a problem, the receiver will probably know about it. The sender? Not so much. So the message sending is somewhat on a UDP-like ‘if it gets there it gets there’ basis. But that does at least reduce message overhead.

What did we learn?

Sending bits back and forth across a wire is more complex than you might think. Getting a receiver to understand what a sender is sending involves a mixture of well-defined expectations and great timing.

The biggest challenge with SensorBus (crude as it is) was letting each end of a conversation know when it should assume command of a signal — particularly the `SB_DAT` line. For example, when a Receiver is in idle mode, it has that line set as an input. After seeing the initial strobe coming from the Sender, it must switch the pin into output mode, perform its own strobe, and then switch back to input mode. The Sender, in the meantime, has a similar journey. And these actions must be synchronised.

Relying purely on timing and delays, hoping the processes will interleave in perfect synchronisation, is a recipe for heartbreak. For the most part, bit-banging of this kind means being able to watch for the state of a line, and having a fallback plan if you don't get what you expected.

Something I already knew, but which this project heavily underlined, is that this kind of challenge would be insuperable without an oscilloscope. A cheap logic analyser might also have done the job, but there were times when I wanted to see actual voltage levels, not just logic levels.

Where next?

So far I have short messages being passed successfully between a node and a module, in both directions. The next step will be to increase the pressure.

I’m going to play with shortening delays in the code to speed things up. And I’m working on a second module, currently at the prototype stage, that is intended to drive a servo motor. At the moment it is sending dummy data to and receiving dummy data from the node.

With both the servo module and the SR04 module running, the node appears to be handling things fine. But there are bound to be collisions between messages. The question is how to deal with these. I could use an Ethernet-like technique of having the sender wait a short, random amount of time before trying again, and having a limit on the number of retries before erroring out.

One question is how much of this to bake into the base class, `SB_Device`, to have consistent behaviour across all devices and how much should be implementation dependent. For example. I might decide that a certain sensor *must* get its message through ('you're about to fall off a cliff!') while others couldn't care less.

There's much scope for experimentation, which is kind of the whole point.

Steve Mansfield-Devine is a freelance writer, tech journalist and photographer. You can find photography portfolio at Zolachrome, buy his books and e-books, or follow him on Bluesky or Mastodon.

Or you can buy Steve a coffee — it helps keep these projects going.

Electronics Technology Robotics Makers Coding

Published in Machina Speculatrix

331 followers · Last published 6 days ago

Following

Electronics, robotics, home automation, hacking and more. The lab notebook of an amateur meddler who likes playing with things until they work—or blow up.

Written by Mansfield-Devine

506 followers · 16 following

Following

Freelance writer & photographer, tech journalist and electronics botherer.

Responses (5)

Alex Mylnikov

What are your thoughts?

Chris Hornberger
3 days ago

> One question is how much of this to bake into the base class, SB_Device, to have consistent behaviour across all devices and how much should be implementation dependent. For example. I ...

IMO, handle this in as much of a 'round robin' way as you think you can manage and yes, include it as base functionality, with ACKs on the message type (and/or content).

When I wrote a queuing system for fund, real-time-ish trades when at BlackRock... more

1 1 reply Reply

Neil Tragham
4 days ago

I took the easy approach...the ESP-NOW lib between sensors back to a main node, back to a WiFi node, through HaLow, through to a WiFi router and onto my servers on the internet. "Intelligence" is back at the server end which sends alerts to people... more

1 Reply

Ernst Reidinga he
4 days ago

Very cool!! 😎 I have been wondering on doing something like it, but more in theory. Very cool to see this in action - shows real experience and knowledge. 🤩

1 Reply

See all responses