Machina Speculatrix

Member-only story

MICROCONTROLLERS

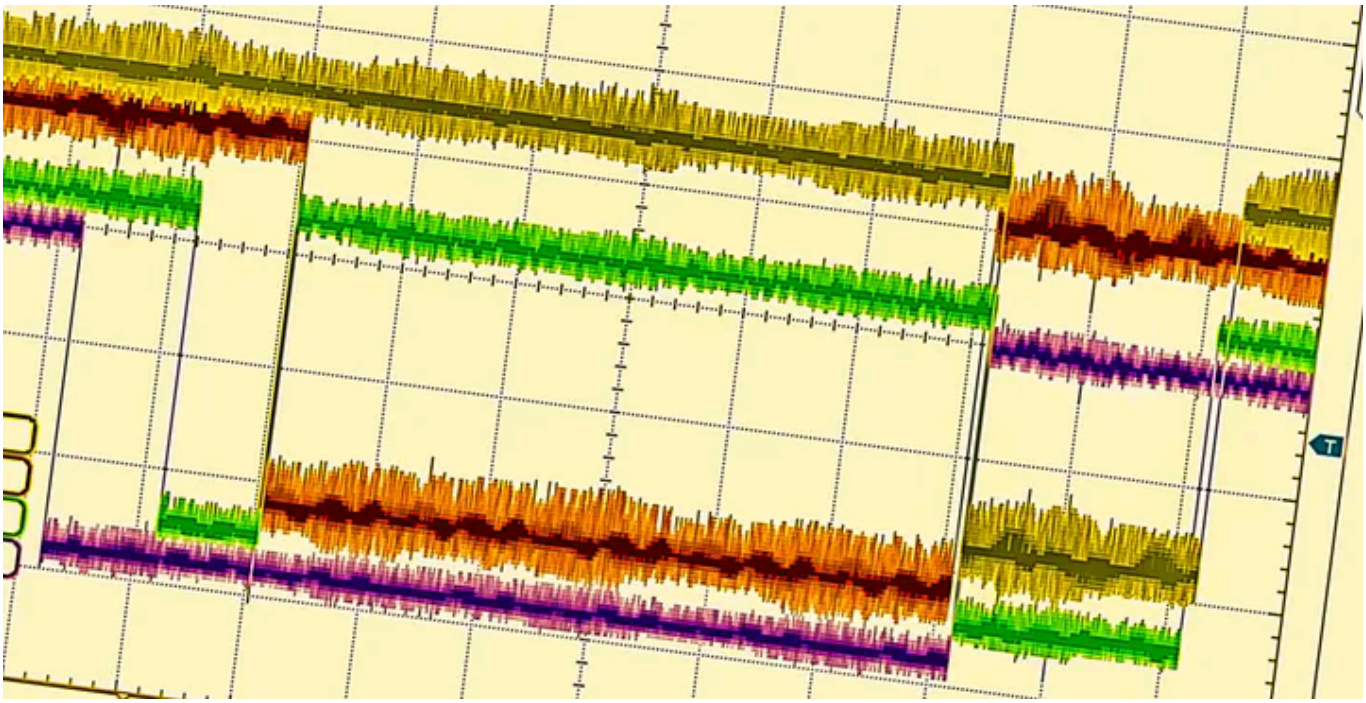
AVR Basics: Transmitting across I2C

The I2C protocol is all about communicating. So let's take a look at how you send data.

Mansfield-Devine · Following

Published in Machina Speculatrix · 10 min read · 3 days ago

3



Once you have set up the I2C interface on your AVR microcontroller, it's time to start sending some commands and data to the peripheral device.

The I2C (or I^2C, if you want to be pedantic) bus is quite complex — at least compared to bog-standard, UART-based serial or SPI. Fortunately, Atmel has done a lot of the hard work for you via a handful of registers. You just need to write the correct values to these and read the results, with the AVR hardware taking care of the nitty-gritty.

Sending a message

Here's a very simple outline of what happens when you send something over I2C. And remember that, for this series, we're talking about using the AVR chip, such as my beloved ATMEGA328P, as a bus controller and some other willing device — a sensor, an LCD display, a motor controller or whatever I2C-compatible gee-gaw your heart desires — as a peripheral.

All communications are initiated by the controller.

Communications are byte-oriented, which is just a fancy way of saying each command issued over the bus is usually a byte long and data is managed in byte-size chunks. When you read about I2C you will also see lots of talk about start conditions, stop conditions, acknowledge bits and no-acknowledge bits. These are all managed via single bits of data; but don't worry — you don't have to set or otherwise mess with these bits yourself. The AVR handles that for you when you use its registers. You just have to be aware that they're happening.

One concept that is useful to understand is that all information on the I2C bus is set up when the clock line is low. (Again, you don't have to worry about doing this, but it is handy to know it's happening.) In other words, each bit of information — whether it's for a command byte or data byte — is set high or low on the SDA line when the clock (SCL) is in the low position. That bit of data is then read when SCL is high.

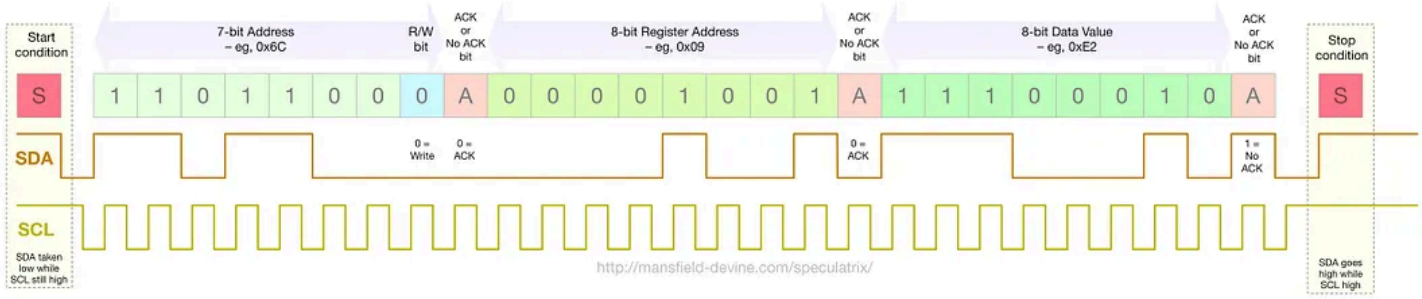
Why is this important? Because it explains how those start and stop conditions are handled. In those cases, the bit is set when the SCL clock line is high. So the fact that the SDA data line transitions from high to low or vice versa when the SCL line is high is a signal to all involved that this is a special kind of bit, not a data bit. Its actual significance depends on when it happens.

Bear in mind also that I2C lines, both SDA and SCL are held high — usually by pull-up resistors — when idle. So I2C lines are 'active low'. You will often see in the literature, such as the microcontroller's datasheet, that a register bit is 'cleared' by writing a 1 to it because 1 is the default condition.

The full package

Let's look at sending an entire message. This will be a fairly simple one consisting of writing a one-byte value to a register on the peripheral device, and in this example we'll write the value 0xE2 to a register at address 0x09.

First we need to get the I2C bus into the start condition. When the controller starts an I2C communication it brings the SDA line low. Because the SCL line is still in its idle position of high, this is a signal that things are about to kick off. The clock then starts pulsing and we are in what's known as a start condition.



Next we send a byte consisting of the sum of two values — the address of the peripheral device we want to talk to (known as `SLA`) and a one-bit value that tells the peripheral whether this is a read (`R`) or write (`W`) operation. This byte, then, is often referred to as `SLA+R` or `SLA+W`.

The address uses the top 7 bits of the byte. You'll see this referred to as a '7-bit address' and that can be confusing. Normally, if you think of something being a 7-bit value it means it's comprised of bits 0–6. But an I2C address isn't like that. It's actually comprised of bits 1–7. This leaves the least significant bit, bit 0, free for other things and also means that I2C addresses are always even numbers.

(I should add at this point that here we're only dealing with the most common I2C versions in which 7-bit addresses are used; there is also a version with 10-bit addresses spread over two bytes.)

Bit 0 is used to tell the peripheral whether this is a read (bit set to 1) or write (bit set to 0). So the value of this byte is: peripheral_address + mode. Let's say our I2C device has an address of 0x6C then the value of this byte will be:

```
For writing (SLA+W): 0x6C + 0 = 0x6C
For reading (SLA+R): 0x6C + 1 = 0x6D
```

So, that first byte consists of one of those values and means that only the device with that address will respond to what's about to come down the I2C bus — until, that is, a stop condition is encountered (we'll get to that).

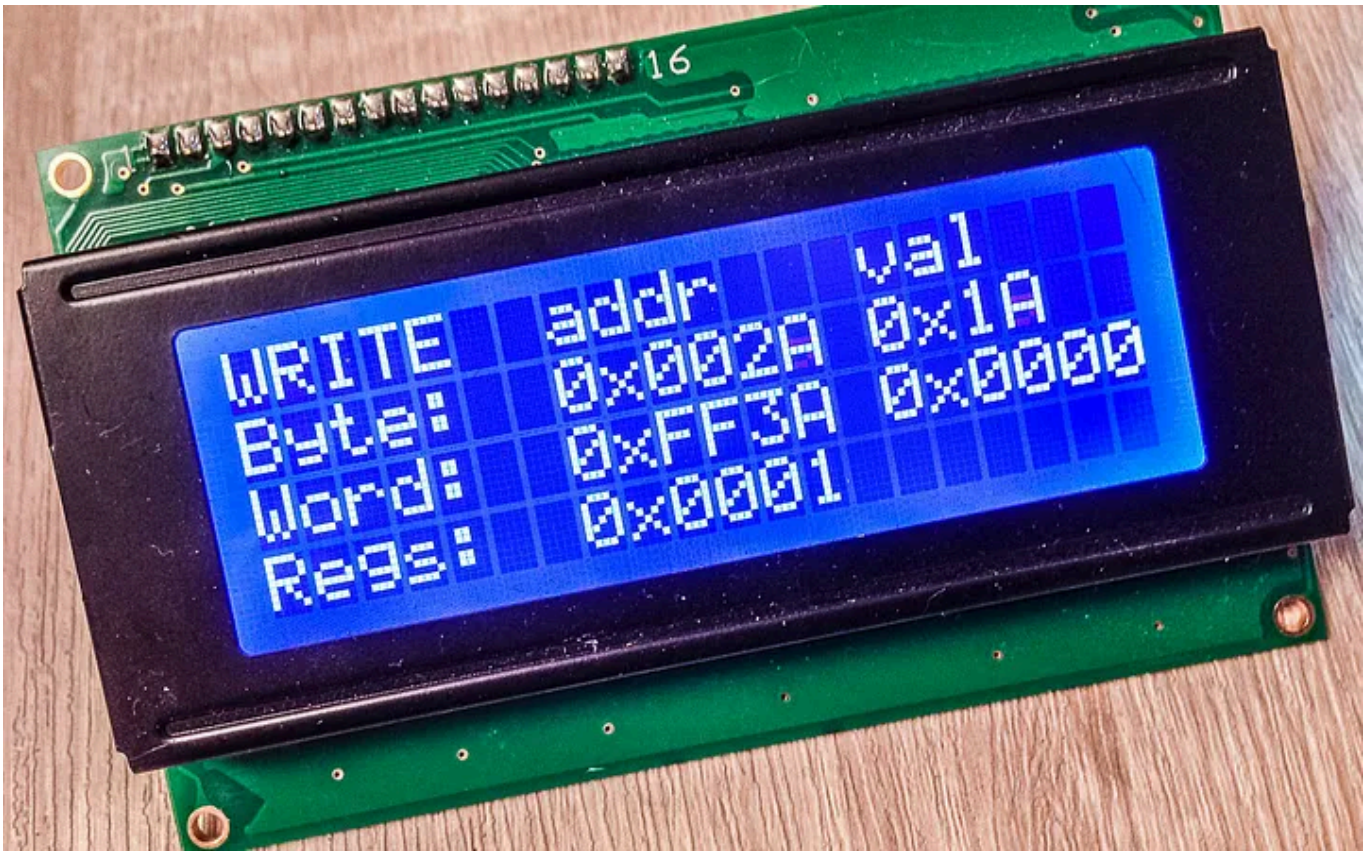
After each byte is sent from the controller, the controller and peripheral swap transmitter and receiver functions for one bit and the peripheral sends back one bit. It does this by taking the SDA line low or allowing it to return high. This is the acknowledge bit and is set to 0 for 'acknowledge' (`ACK`) or 1 for 'no acknowledge' (`NOT ACK`). It's up to your code how you handle this. We'll touch on this some more when we talk about the AVR's registers.

The next byte sent from the controller is typically a register value, also commonly known as a 'command' byte. Selecting a register is your way of telling the peripheral device what you want it to do. For example, with a real-time clock chip I'm playing with right now (the DS3107), sending the value 0x05 tells it that you want to write or read the month value. Again, this byte is followed by an acknowledge bit.

It could end there. With some devices, simply selecting a register achieves the effect you're after. Other times, you will send one or more data bytes. Let's take the example of a 24 x 4 LCD display I like to use. Sending the command (register) byte of 0 tells the display you want to write to the screen. You then send as many bytes as you want to write characters, control the backlighting, set the cursor position and so on. So a transmission might consist of several, even dozens of bytes all prefaced by just one address and one register byte.

Finally, the controller signals the end of the transmission — the 'stop condition' by holding SDA low, allowing SCL to go high where it stays, and then allowing SDA to go high.

Sounds complicated, doesn't it? Luckily, the implementation of all this is largely taken care of by the AVR microcontroller. All you need to do is make judicious use of its registers.



Writing to an I2C-compatible LCD display is surprisingly simple.

The registers

There are four main registers that concern us when it comes to the I2C bus. We met the bit-rate register, `TWBR`, in a previous article, so we don't need to mess with that again. But we'll now be using the other three. Well, two anyway.

TWDR — data register. This is the simplest of the lot. When transmitting, you just write the value of the byte you want to send to this register. When receiving, this is where you'll find the received byte.

TWCR — control register. Writing specific bits — each of which has its own name — to this register is what makes things happen on the I2C bus and also notes what has happened. The bits are:

```
  7       6       5       4       3       2       1       0
TWINT   TWEA    TWSTA   TWSTO   TWWC    TWEN    -       TWIE
int.    enable  start   stop    write   I2C     (not    int.
flag    ack.    cond.   cond.   coll.   enable  used)   enable
```

The `TWINT` interrupt flag is especially noteworthy here: it is set by the hardware whenever the two-wire interface (TWI, or I2C to us civilians) has finished its current task. So it's a way of flagging that reads or writes have completed. There's a lot more to it than that, but that'll do us for now.

In this article, we're only interested in the following bits, all of which are read/write.

TWSR — status register. We've already met two bits from this register: bits 0 and 1 (`TWPS0` and `TWPS1`) are used in setting the bit rate. Bit 2 isn't used and always returns 0 if read. The other bits, 3–7 (aka `TWS3` to `TWS7`) are read-only and return a value that corresponds to the result of the last operation. We'll come back to this in a later article, but to give you an idea, if you've issued the appropriate command to initiate a start condition, then the `TWCR` should contain the value 0x08. To get this value, you need to mask out bits 0 & 1. So, for example, one way of getting the value might be something like:

```
uint8_t i2c_status = TWSR & 0b11111100;
```

Making things happen

For anything to happen over the I2C interface, the `TWEN` bit of the control register needs to be set (to 1). This is what turns ordinary, innocent-looking

GPIO pins into the I2C interface.

You also need to clear the interrupt flag, `TWINT`, by writing a 1 to it. And so writing both of these bits to 1 is essentially our way of triggering the I2C bus, or prodding it into action.

And after each prod, we need to wait for the action to complete. We know this has happened because the `TWINT` bit gets set (to 0). Now, advanced users may use interrupt routines in clever ways that allow the microcontroller to get on with stuff while waiting for I2C actions to complete. But for our basic purposes here, we'll just hang around in a blocking loop. My preferred way of doing this is:

```
while ( !(TWCR & (1 << TWINT)) );
```

Some people like to define that as a macro, which we'll do here so it's obvious what's going on:

```
#define wait_for_completion while(!(TWCR & (1 << TWINT)));
```

Now we can just use `wait_for_completion` whenever we need to pause until the action is complete.

Set a register

As with the previous article, let's look at an example where we set the value of a register in the peripheral device — a common way of getting the device

to do something. In this example, we're going to write the value 0xE2 to register 9 (or 0x09, if you prefer). The process consists of:

- Setting a start condition.
- Sending the peripheral device's bus address plus the write bit down the bus.
- Sending the address for the peripheral device's register down the bus (eg, 0x09).
- Sending the value we want to write (eg, 0xE2) down the bus.
- Setting the stop condition.

Start condition

To set a start condition and basically get the whole thing rolling, you need to set the start bit, `TWSTA`, in the control register. This is accompanied by setting the `TWINT` bit (to clear the interrupt) and the `TWEN` bit (to enable the I2C bus). And after doing all that, we wait for this action to complete. So this is what we do:

```
// set the start condition
TWCR = ((1 << TWEN) | (1 << TWINT) | (1 << TWSTA));
```


Sending the address

Let's say the address of our peripheral device, typically referred to as `SLA`, is 0x6C. To this we need to add the appropriate bit for read (`R`) or write (`W`). As write mode is set using 0, then the next byte we send (`SLA+W`) is actually just the address. And we send a byte by first loading its value into the data

register, `TWDR`, and then setting `TWINT` and `TWEN` again. In this example, what we need to do is:

```
// send the address
TWDR = 0x6C;                          // data to send - ie, address + write bit
TWCR = ((1 << TWEN | (1 << TWINT));  // trigger I2C action
wait_for_completion;
```

Register address and data

Sending the register address is no different from sending any other byte of data. So now we're going to send the register address and the byte of data we want to store in it.

```
// send the register address
TWDR = 0x09;                          // register address
TWCR = ((1 << TWEN | (1 << TWINT));  // trigger I2C action
wait_for_completion;
// send the data byte
TWDR = 0xE2;                          // data byte
TWCR = ((1 << TWEN | (1 << TWINT));  // trigger I2C action
wait_for_completion;
```

Stopping

Now that we've done what we set out to do, we need to set a stop condition. This frees the I2C bus for other operations. This is nearly identical to setting the start condition, except that we use `TWSTO` in place of `TWSTA`.

```
// set the stop condition
TWCR = ((1 << TWEN) | (1 << TWINT) | (1 << TWSTO));
```

```
    wait_for_completion;
```

And that's pretty much it as far as sending a simple command is concerned.

You'll note that we haven't made any mention of the acknowledge bits that we talked about last time. Nor have we been paying attention to the status register `TWSR`. We'll talk a little more about those in the next article, which will deal with receiving data.

You can find all the AVR-related articles here.

I've created a GitHub repo for supporting files to accompany this AVR series of articles. You can find it here: https://github.com/mspeculatrix/AVR_8bit_Basics/

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

You can also buy Steve a coffee. He'd like that.

Programming | Avr | Microcontrollers | Technology | Electronics

Published in Machina Speculatrix

59 Followers · Last published 7 hours 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

89 Followers · 7 Following

Following

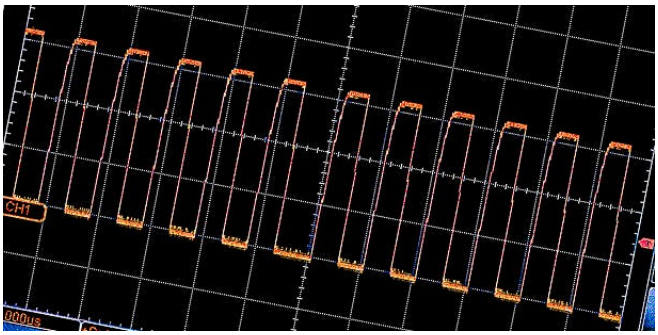
Freelance writer & photographer, tech journalist and electronics botherer.

No responses yet

Alex Mylnikov

What are your thoughts?

More from Mansfield-Devine and Machina Speculatrix

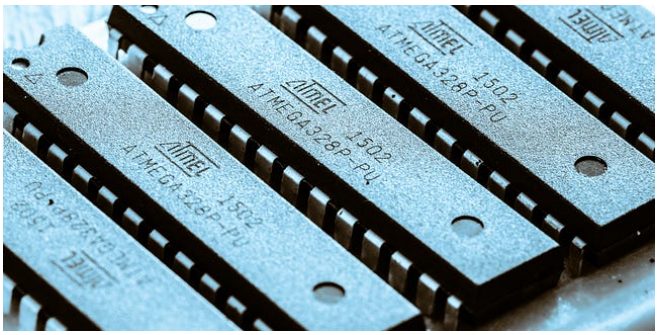


In Machina Speculatrix by Mansfield-Devine

AVR Basics: Getting started with I2C

Being able to communicate with other devices opens up a world of possibilities—if...

Apr 14 135 1

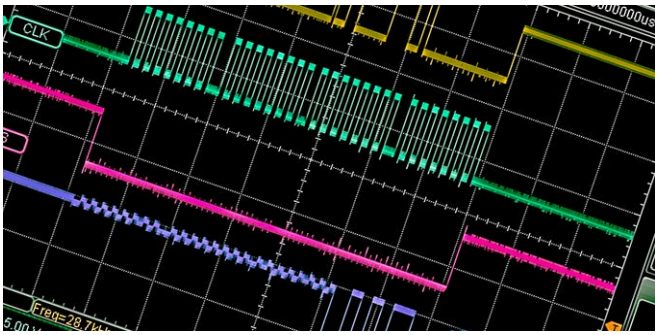


In Machina Speculatrix by Mansfield-Devine

AVR basics: pin change interrupts

Interrupts are extremely useful in micocontroller applications. And they're...

Mar 23 147 2

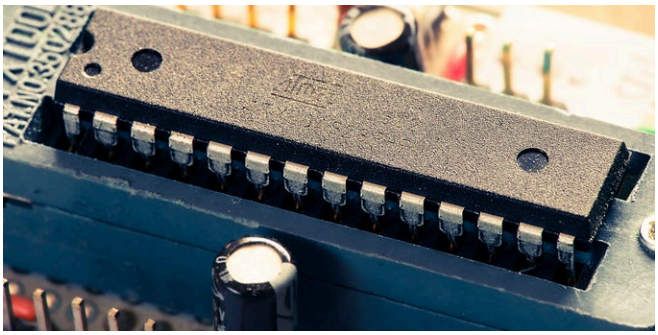


In Machina Speculatrix by Mansfield-Devine

AVR basics: SPI on the ATMEGA

The Serial Peripheral Interface (SPI) on AVR microcontrollers is very versatile, once you...

Mar 29 110 1



In Machina Speculatrix by Mansfield-Devine

AVR basics: reading analogue input

Turning a microcontroller's I/O ping on and off is all very clever, but what about when things...

Mar 16 84 2

See all from Mansfield-Devine

See all from Machina Speculatrix

Recommended from Medium

hiruthicSha

How I Set Up a Home Server That I Can Access from Anywhere

Ever since I started learning programming, I've had one consistent problem: I had to...

Apr 13 327 12

In Pythonic AF by Aysha R

I Tried Writing Python in VS Code and PyCharm—Here's What I...

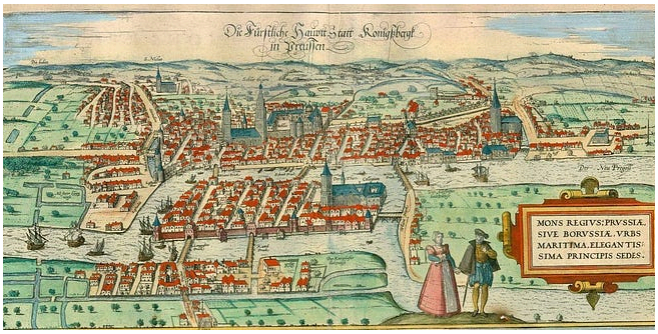
One felt like a smart coding companion. The other felt like assembling IKEA furniture...

Apr 15 806 69

In Machina Speculatrix by Mansfield-Devine

Getting to grips with the parallel interface

Putting an old printer back into use meant talking the language of its now (mostly)...



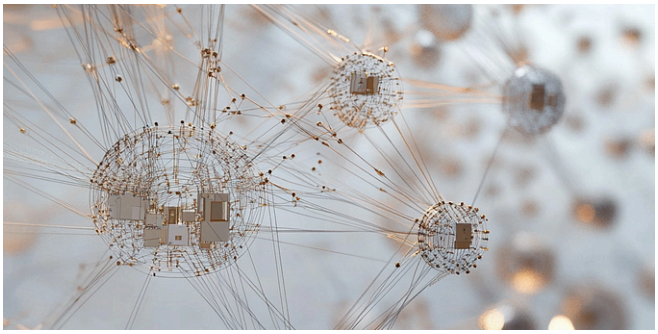
In EduCreate by A. Merriam

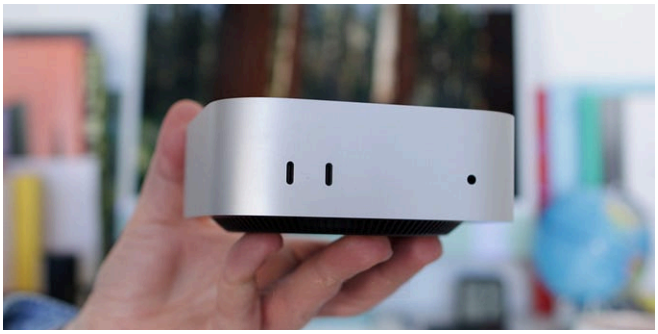
Euler and the Seven Bridges of Königsberg

How this classic problem in graph theory led to the development of topology

Feb 28 48

4d ago 151 1





In Radio Hackers by Simon Quellen Field

Python Radio 36: Mesh Networking

Get the word out to nodes you can't see.

Apr 15 233 2

In Mac O'Clock by David Lewis

How the M4 Pro Mac mini changed everything

One seemingly simple Mac purchase and the differences it's made

4d ago 179 6

See more recommendations