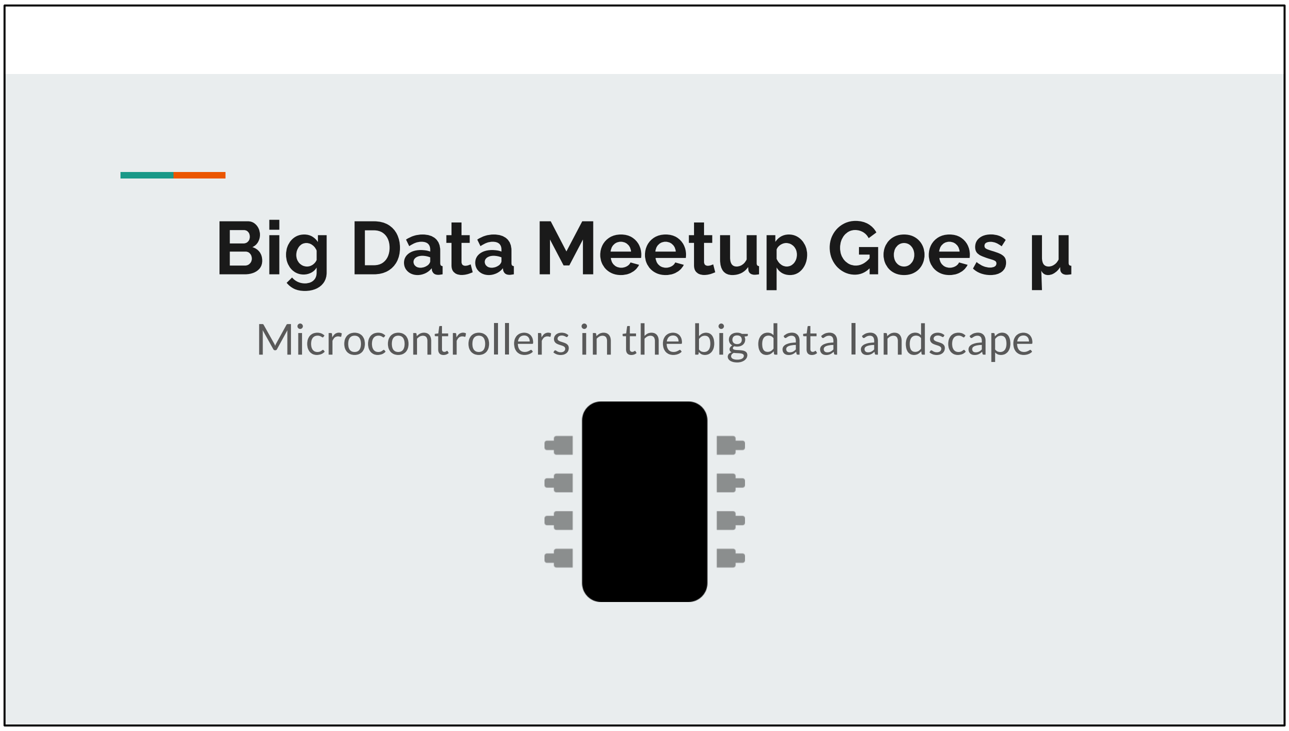
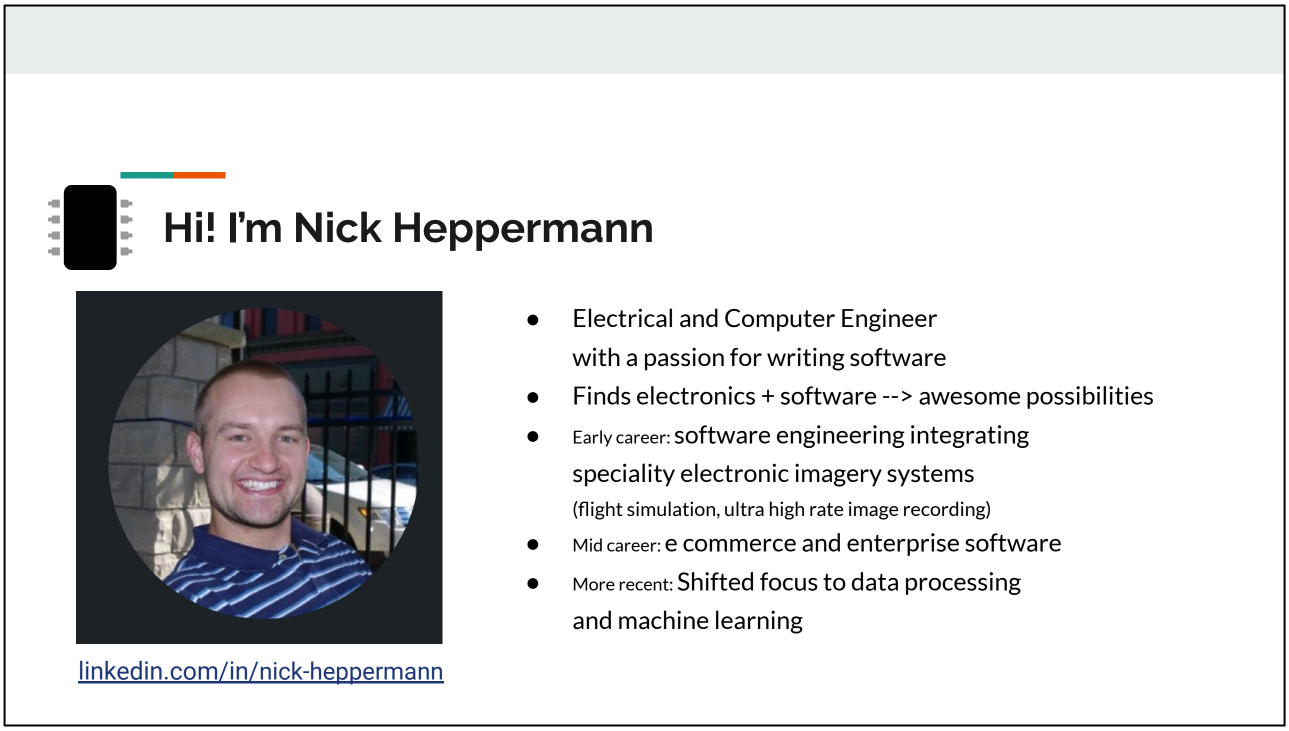


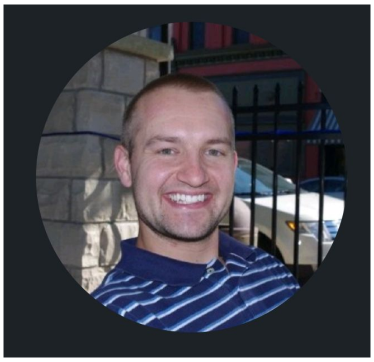
Big Data Meetup Goes µ

Microcontrollers in the big data landscape

Thank you everyone for being here and letting me share my ideas on microcontrollers in the big data landscape.



Hi! I’m Nick Heppermann



linkedin.com/in/nick-heppermann

- Electrical and Computer Engineer with a passion for writing software
- Finds electronics + software --> awesome possibilities
- Early career: software engineering integrating speciality electronic imagery systems (flight simulation, ultra high rate image recording)
- Mid career: e commerce and enterprise software
- More recent: Shifted focus to data processing and machine learning

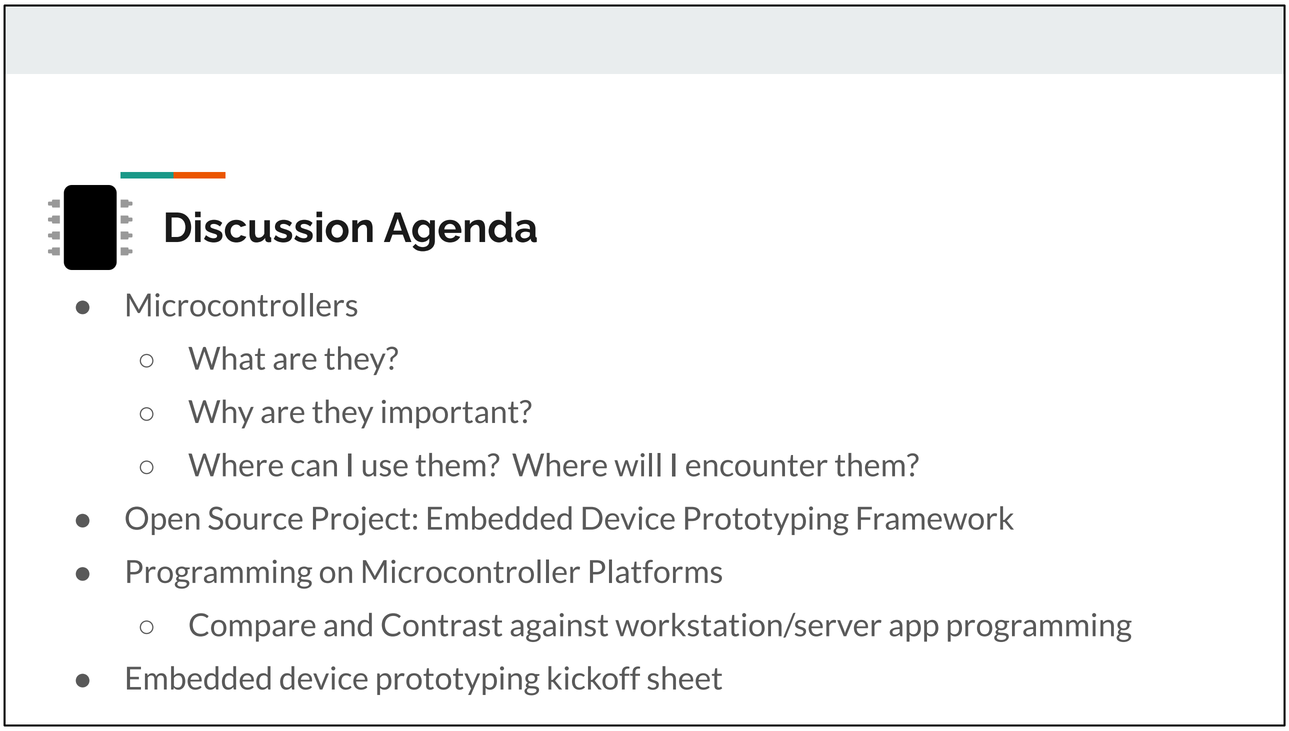
Hi, my name is Nick Heppermann, and I’m an Electrical and Computer Engineer with a passion for writing software.

I find the marriage of electronics and software creates some pretty awesome possibilities, and fills a vital role in many critical use cases and I hope to convey that this evening.

To give you some idea on my background, I started my career as a software engineer integrating electronic imagery and video systems for defense applications.

Then I joined an e commerce company based in the St. Louis with partwise focus on imagery and video processing, and also enterprise software in general.

And for the most recent past few years, I have shifted focus to data processing and machine learning technologies in the e commerce setting.



Discussion Agenda

- Microcontrollers
  - What are they?
  - Why are they important?
  - Where can I use them? Where will I encounter them?
- Open Source Project: Embedded Device Prototyping Framework
- Programming on Microcontroller Platforms
  - Compare and Contrast against workstation/server app programming
- Embedded device prototyping kickoff sheet

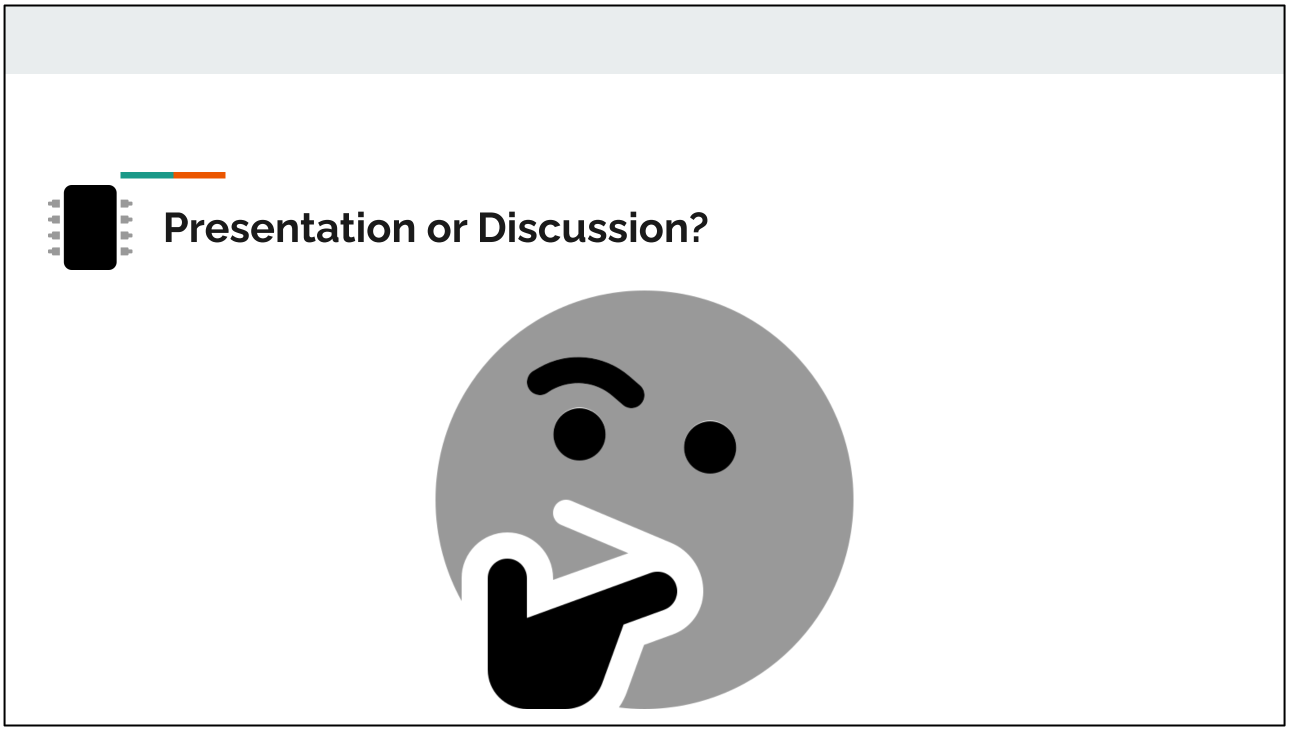
Here's what we have in store for tonight's discussion.

First we will go over microcontrollers in general: what are they, why they are important, where you can use them and expect to encounter them.

Then we will go over the public rollout of my open source project -- the Embedded Device Prototyping Framework -- which happens tonight

Also, we will compare and contrast programming on microcontroller platforms versus programming for computer workstations

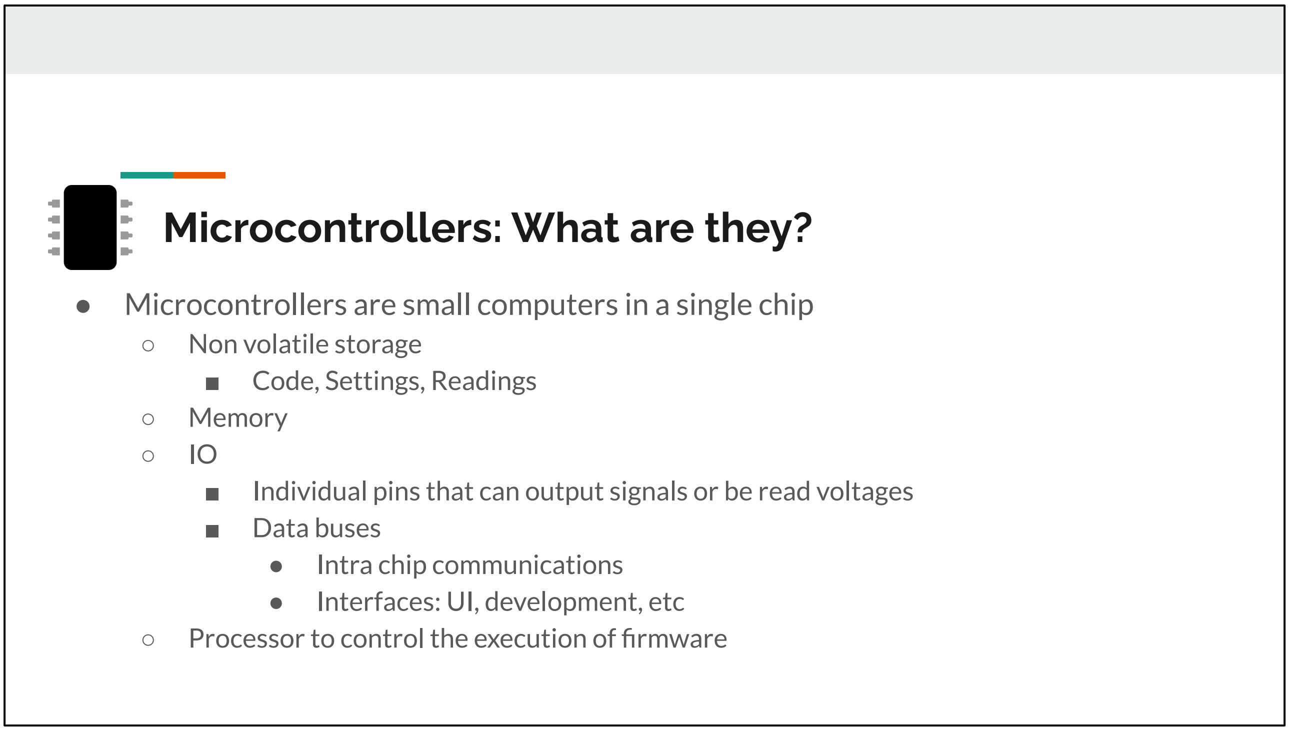
Lastly, I will share with you some embedded device prototyping kickoff tips and some closing thoughts.



Presentation or Discussion?

Before we dive in, let’s talk about whether this will be a presentation or discussion. I much more enjoy discussions over presentations. I want you all to be absolutely encouraged to interrupt me, share your takes on the matter, probe my shallow knowledge, and pitch in where you think you can.

With that said, I would like to ask some people to share -- Who here has worked on electronics projects? Past, currently working on one, or plans to undertake one in the near future? <what kind of project?> Are there some experienced engineers and hobbyists attending tonight? <what kind of systems have you worked on?>



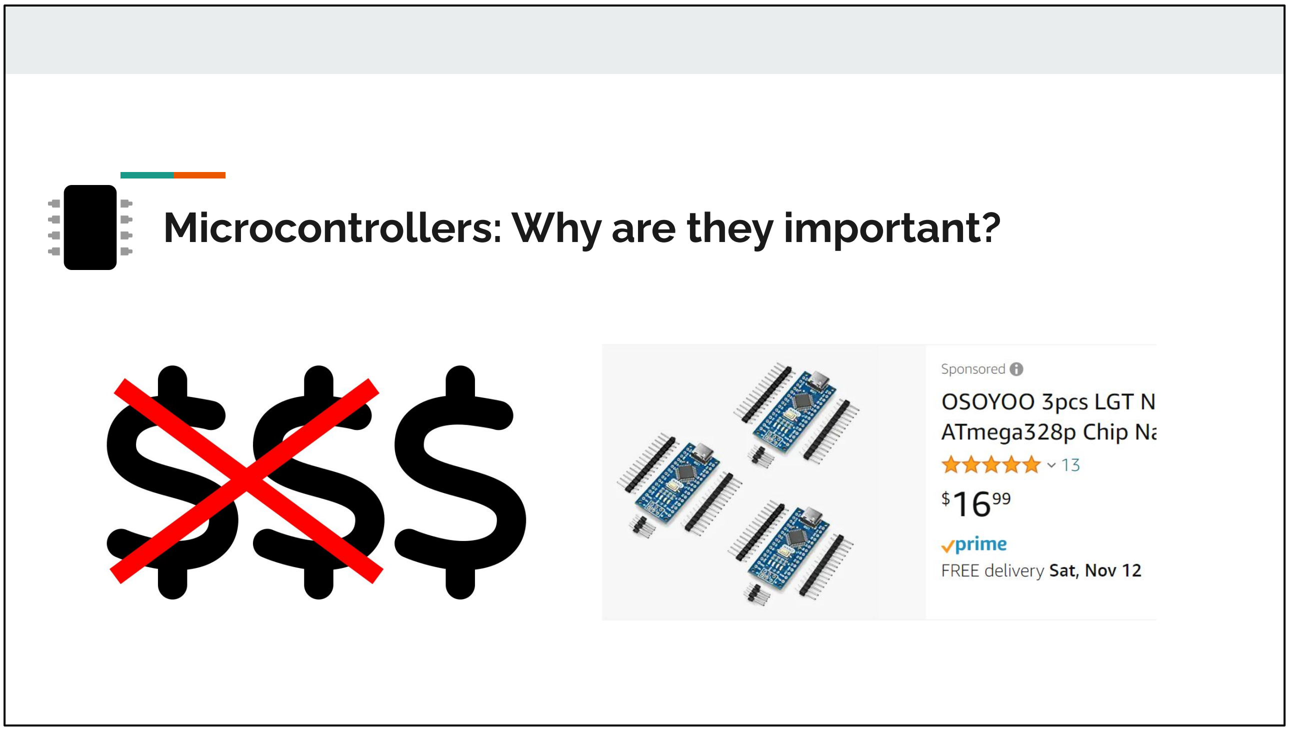
Microcontrollers: What are they?

- Microcontrollers are small computers in a single chip
  - Non volatile storage
    - Code, Settings, Readings
  - Memory
  - IO
    - Individual pins that can output signals or be read voltages
    - Data buses
      - Intra chip communications
      - Interfaces: UI, development, etc
  - Processor to control the execution of firmware

Microcontrollers: What are they?

Microcontrollers are small computers that come in the form factor of a single chip. Inside them you find the main components of normal workstations computers:

- Nonvolatile storage to hold the code that executes, settings that get remembered between power cycles, and sometimes readings from peripheral devices
- Memory to hold software instructions and volatile working storage
- IO components such as individual pins for outputting signals or reading voltages
- Data busses for communication between additional chip packages and UI elements
- And a processor to control the execution of firmware



Microcontrollers: Why are they important?

Sponsored

OSOYOO 3pcs LGT N
ATmega328p Chip Na

★★★★★ 13

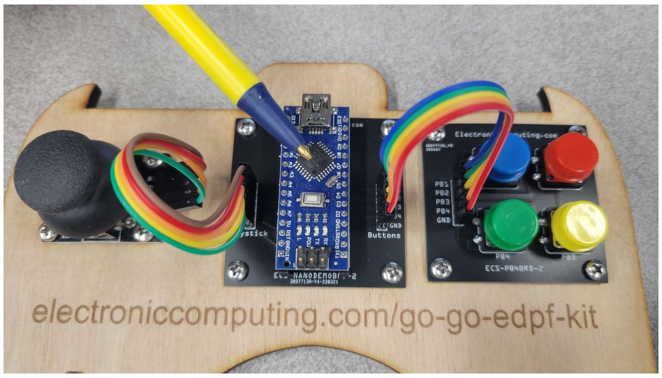
$\$16^{99}$

prime

FREE delivery **Sat, Nov 12**

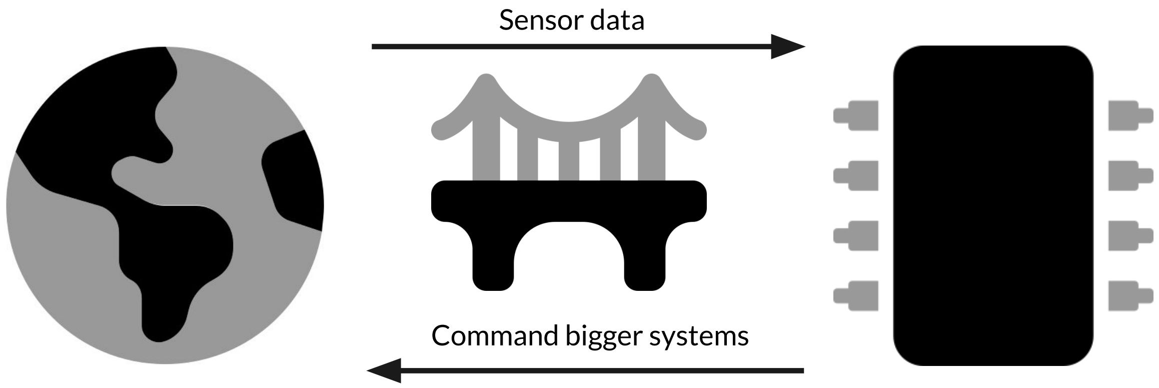
Microcontrollers have a few superpowers that help fulfill some very important needs in our data landscape. First off, they can be very cheap. Even in the pandemic times with chip shortage prices, a person can buy 3 microcontrollers with all the connectivity hardware in single package boards for under $20.

Microcontrollers: Why are they important?

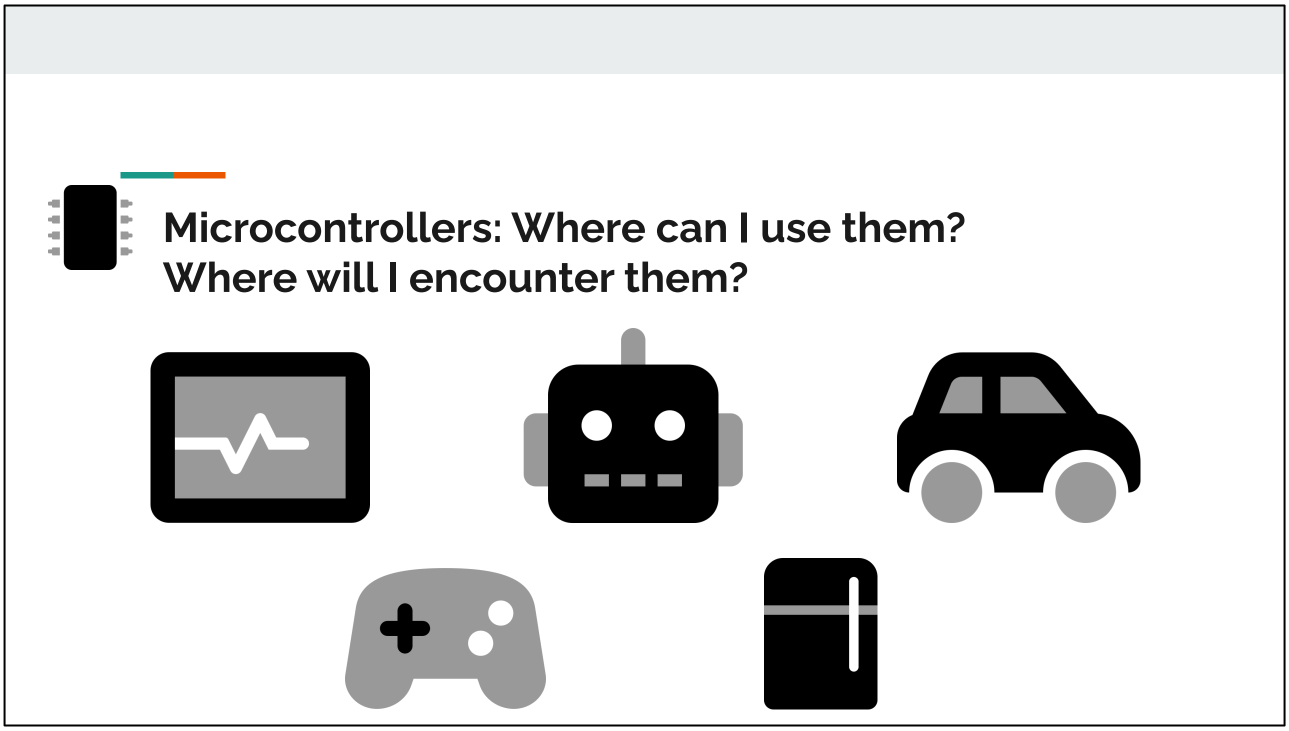


Microcontrollers are small -- you can see in this picture, the microcontroller is not much larger than the tip of the ink pen. Often, microcontrollers come mounted to boards that provides essential connectivity between the microcontroller and other components. As you can see the size of the microprocessor and and connectivity components are small, which means their form factor allows them to fit in many kinds of applications.

Microcontrollers: Why are they important?



And this is my favorite super power -- amongst their many uses they often act as bridges between the physical and digital worlds. They bridge the gap between the real world by providing access to sensors like in data capture applications, and also can be a main component in providing commands to larger systems that run physical machines, or bridging electro-mechanical human interfaces (push buttons, LED screens, joysticks and so on) that allow humans to interact with devices.

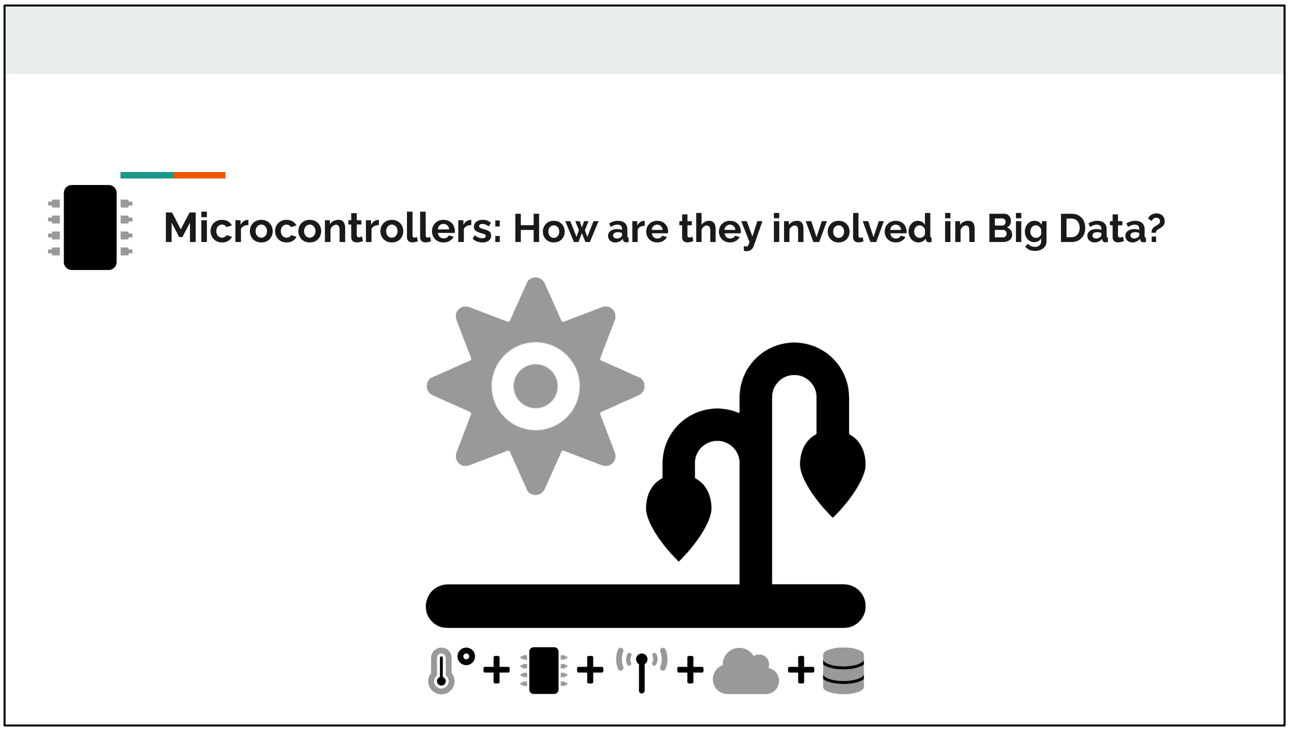


Microcontrollers: Where can I use them? Where will I encounter them?

Where will you encounter microcontrollers? I think the better question is “Where won’t you encounter microcontrollers?” With the benefits I mentioned earlier, they are used pervasively in many types of applications -- such as:

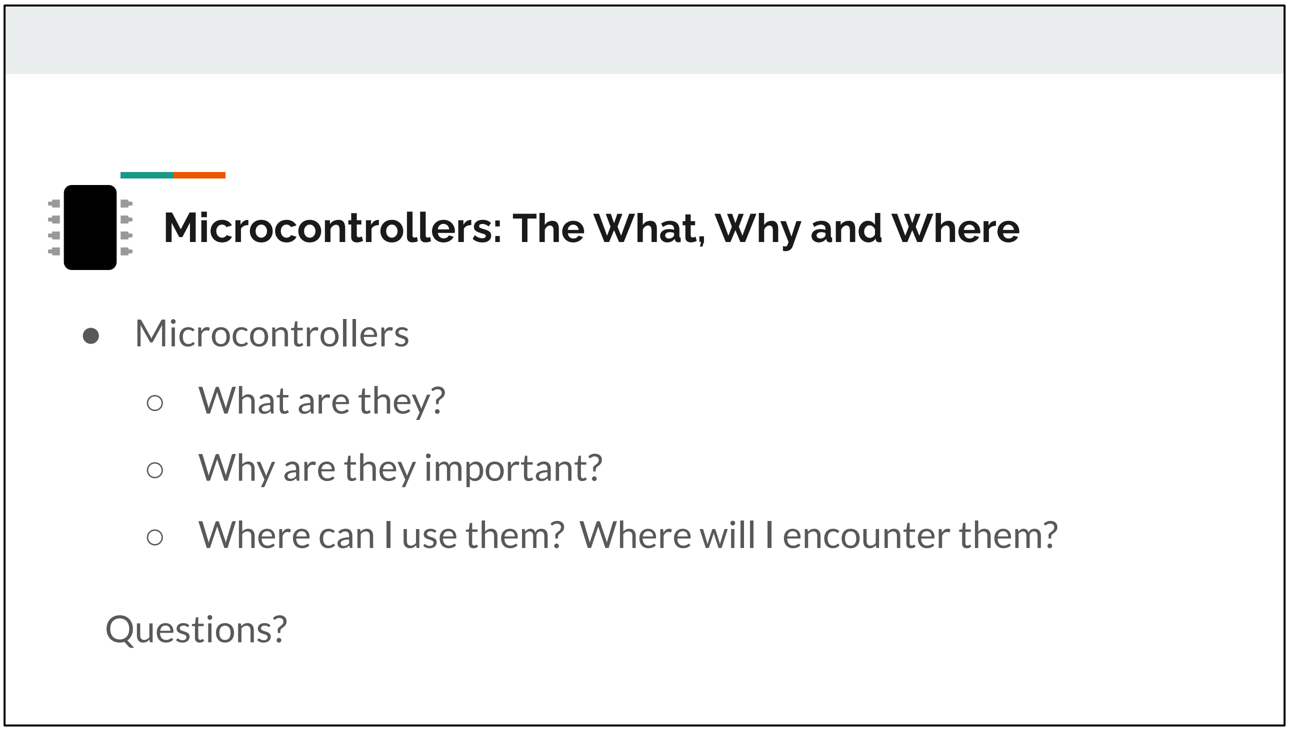
- Healthcare related devices
- Toys
- Automotive components
- Controllers
- Household appliances

And on, and on, and on



Microcontrollers: How are they involved in Big Data?

So how are microcontrollers involved in Big Data? In the Internet of Things there are numerous scenarios where microcontrollers come into place. Lets take this one example scenario where a temperature sensor exists in an agricultural setting. The microcontroller captures readings from the sensor, transmits the readings to the cloud, and finally the readings are persisted to a data store.



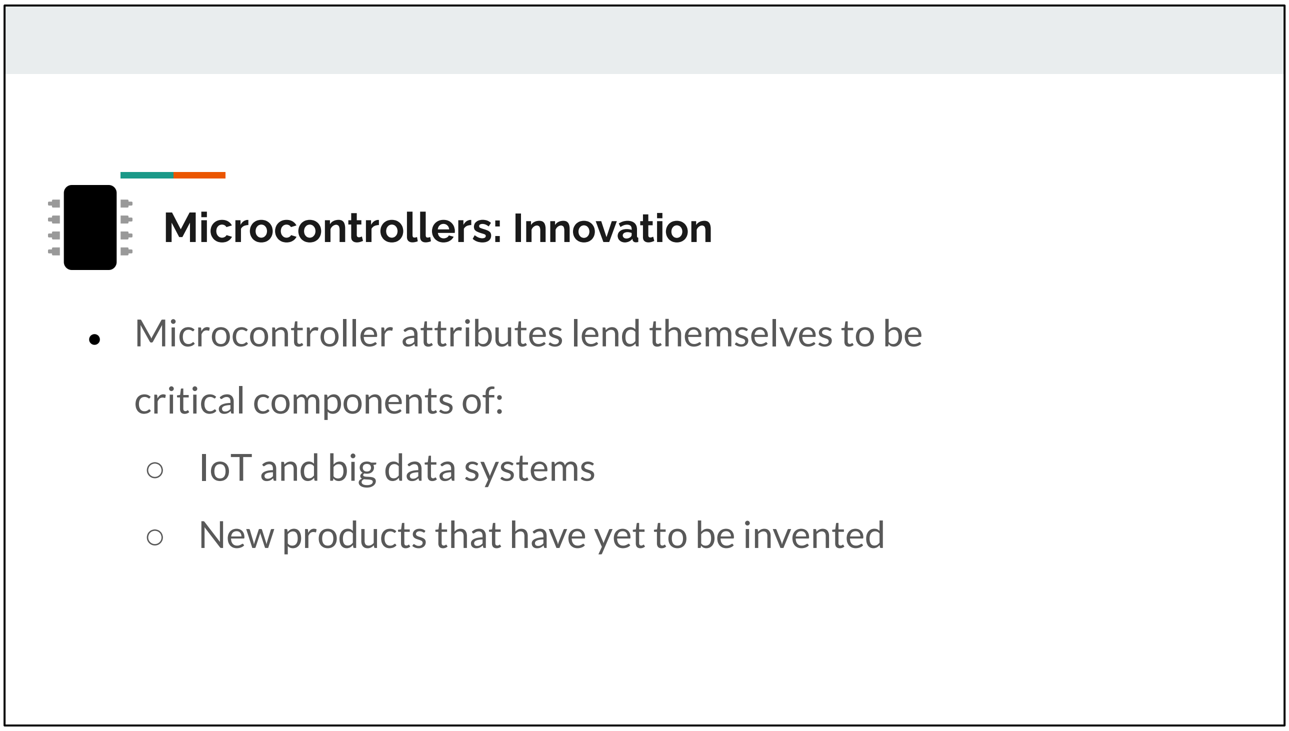
Microcontrollers: The What, Why and Where

- Microcontrollers
  - What are they?
  - Why are they important?
  - Where can I use them? Where will I encounter them?

Questions?

Alright, so that concludes the first part of the presentation -- the what, the why, and the where of microcontrollers.

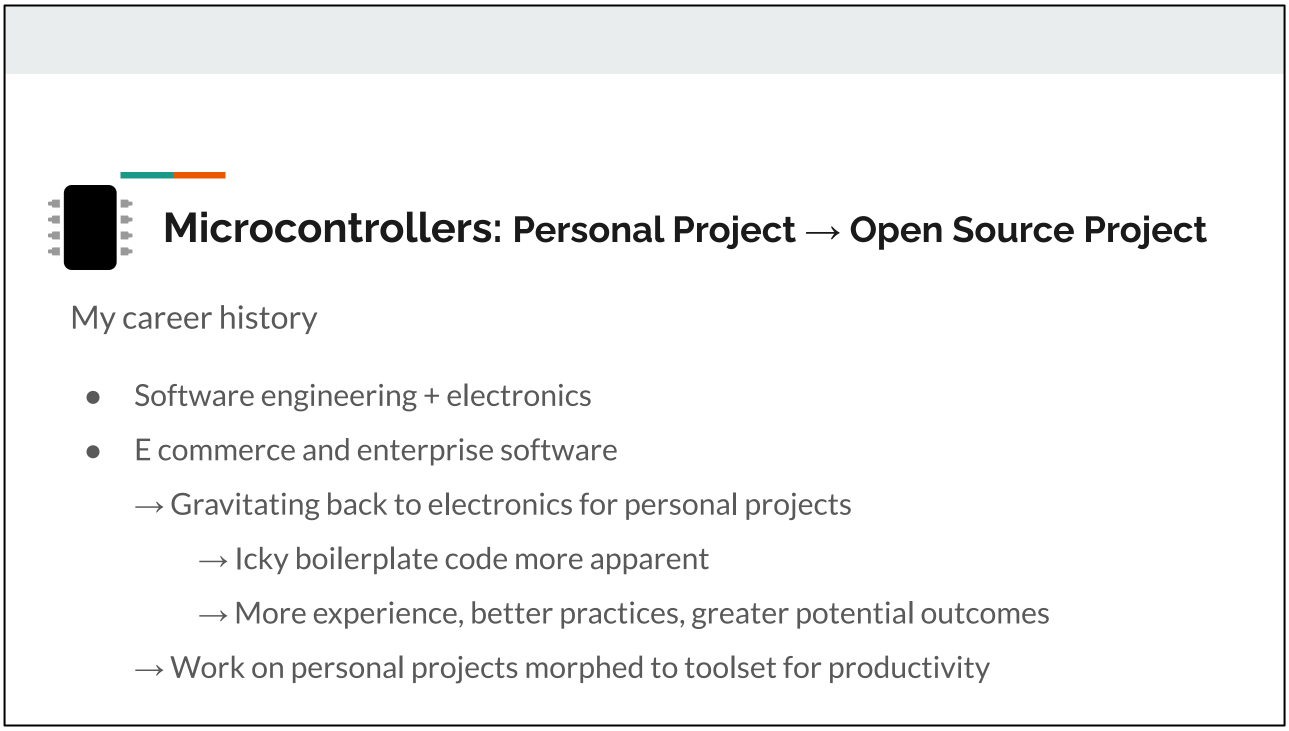
I want to take a pause from speaking a bit and open the floor. Any questions so far?



Microcontrollers: Innovation

- Microcontroller attributes lend themselves to be critical components of:
  - IoT and big data systems
  - New products that have yet to be invented

Now let’s tie microcontrollers and embedded systems into innovation. At a high level we touched on how microcontrollers fit into the digital landscape -- the attributes of the microcontroller make it a core component for many types of applications, and with their small size they can fit in almost any sort of form factor product and facilitate many use cases. The microcontroller has been a key component of many products, and will continue to be a key component for many new products to come.



Microcontrollers: Personal Project → Open Source Project

My career history

- Software engineering + electronics
- E commerce and enterprise software

  → Gravitating back to electronics for personal projects

    → Icky boilerplate code more apparent

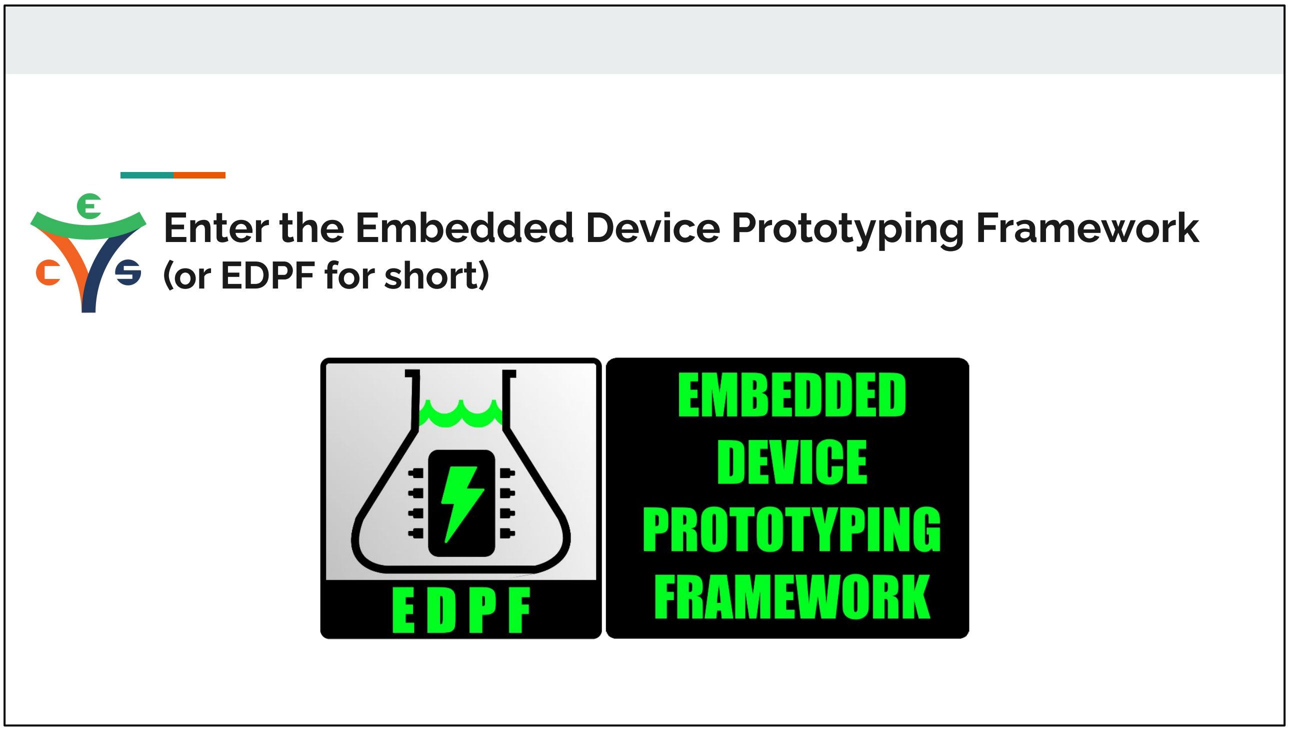
    → More experience, better practices, greater potential outcomes

  → Work on personal projects morphed to toolset for productivity

Now, lets focus on my exposure to microcontrollers and how it lead into the open source software we are discussing tonight. In my first 7 years out of school I worked daily as a software engineer integrating microcontrollers with computers, and then I shifted focus to apply software engineering to the e commerce space. Awhile later, I found myself dabbling back into electronics again in my free time doing a few personal projects.

After the additional years of software engineering and exposure to some really great minded engineers, habits, and concepts, I realized how much boilerplate code could be generated by the code of simply passing data back and forth between embedded devices and workstations.

With my additional learnings and experience, I realized I could use better software design, code less in my projects, and prototype my work much faster.



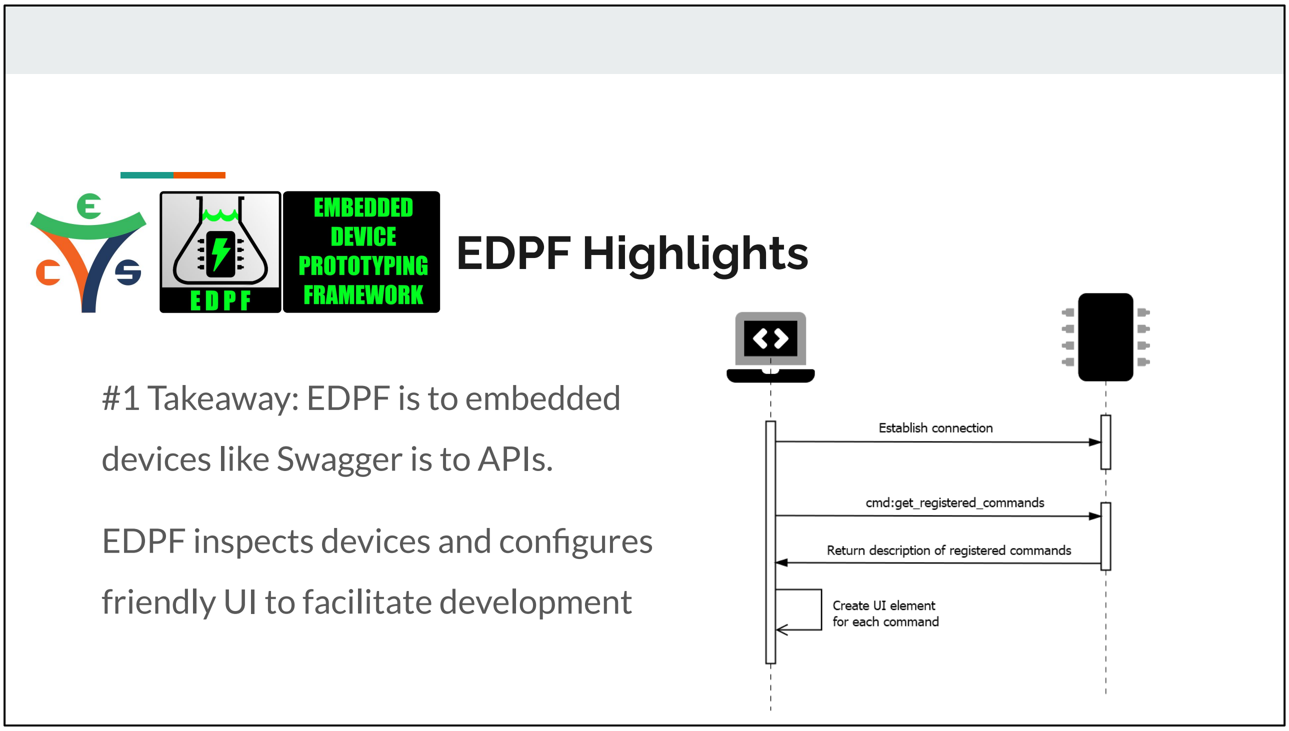
Enter the Embedded Device Prototyping Framework
(or EDPF for short)

EDPF

EMBEDDED DEVICE PROTOTYPING FRAMEWORK

Thus spawned the birth of the Embedded Device Prototyping Framework -- EDPF for short.

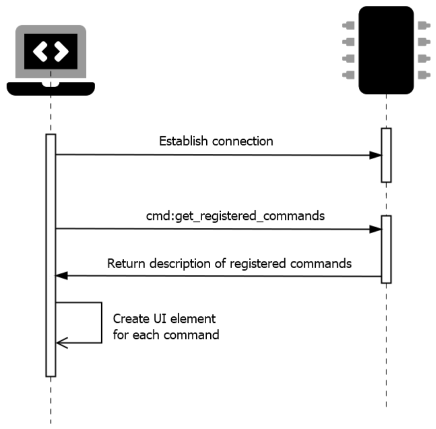
And there's two big takeaways I want everyone to leave with about the EDPF



EDPF Highlights

#1 Takeaway: EDPF is to embedded devices like Swagger is to APIs.

EDPF inspects devices and configures friendly UI to facilitate development

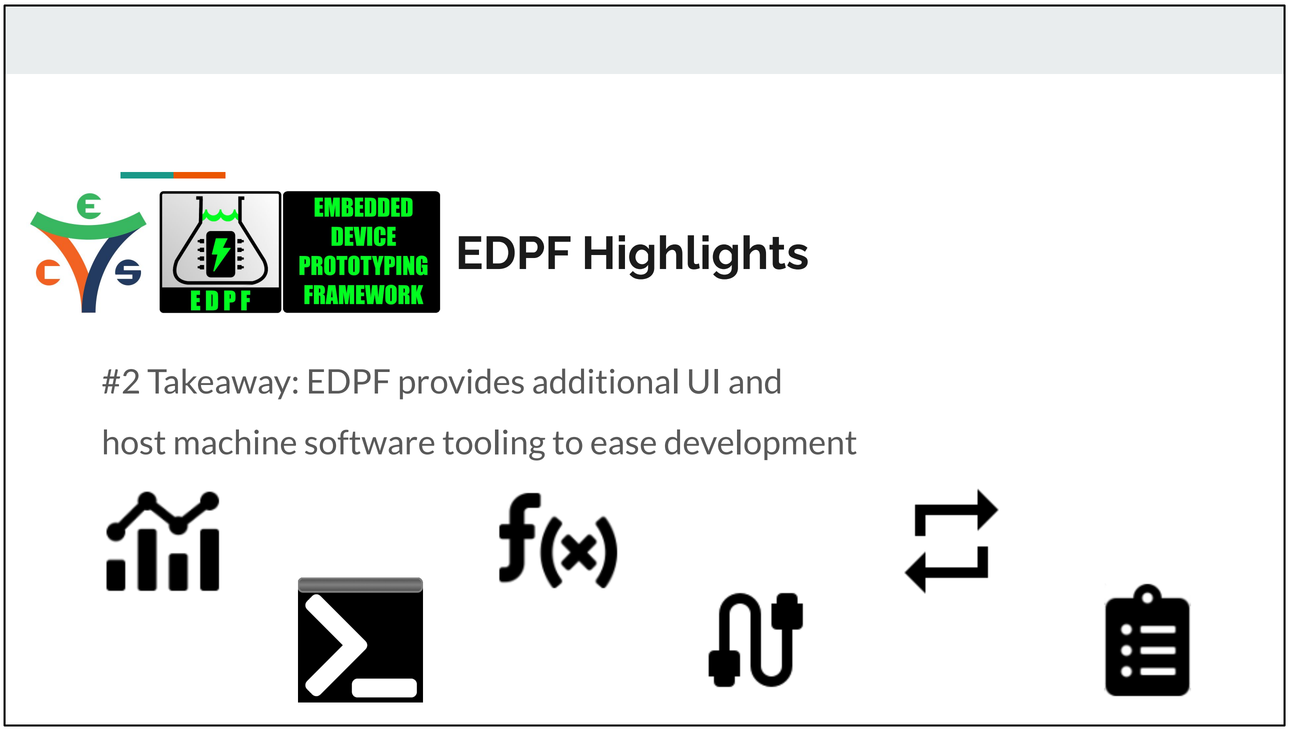

Establish connection
cmd:get_registered_commands
Return description of registered commands
Create UI element for each command

Number 1 takeaway, the EDPF is to embedded devices as Swagger is to APIs.

Q: Who here is familiar with Swagger? Could someone explain what Swagger does for APIs?

Right, Swagger is this great UI component for exploring APIs. Swagger provides this really convenient user interface which allows developers to learn a lot about the API endpoints with a minimal amount of effort.

Likewise, the EDPF provides a UI in the host application that inspects the microcontroller’s firmware, the firmware responds with its capabilities, and the host UI configures itself for easy interaction with the device.



EMBEDDED DEVICE PROTOTYPING FRAMEWORK

EDPF

EDPF Highlights

#2 Takeaway: EDPF provides additional UI and host machine software tooling to ease development

Number 2 takeaway -- the EDPF provides additional host UI software tooling to ease development of prototype devices.

This tooling includes charting

A console interface

A command interface which acts a lot like functions in programming

A connections tool

A macro tool to repeat sequences of device interactions

And logging

EMBEDDED DEVICE PROTOTYPING FRAMEWORK

EDPF Demo

<make video demo as backup>

And now for a quick demo of the UI. As you can see before any devices are connected the UI is pretty much disabled, but after a connection is established, all the magic happens. The host app queries the firmware for its supported functionality and the firmware responds with command descriptions.

For each command description, a UI component is created to easily execute each command. And if the command accepts arguments the UI provides named field input boxes that are strongly typed.

Click the Execute button and voila -- you have interacted with the device. You can see here the Console window shows the IO flow of the device and host machine interactions.

One of the commands that comes out of the box is the ‘edpf_kit_read’ command, which you compile into the firmware when you have an EDPF demo kit handy. The EDPF demo kit was made as an approachable real world sensor set to show off the features of the EDPF. Im going to press some buttons and move the joy stick around and execute the ‘edpf_kit_read’ command to show that different values are being read. Please keep this in mind as I move on to the other controls and show off how these tools can be tied together.

Moving on to the Macro Tool. The macro tool is useful for executing a sequence of commands either in a continual loop or in one shot. Lets put in a few of the 'edpf_kit_read' commands with some delays baked in, and we will kick it off to run continually.

You will see the stream of data flowing through on the Console view. We will just leave this running.

Now lets jump over the to the charting control. The charting control uses a really simple expression syntax to describe data coming from the device, and the Host UI uses the expression to parse device output and plots X number of data values on Y number data charts. For the 'edpf_kit_read' command we can use an expression string like this:

```
kit:{x|chart1},{y|chart1},{d|chart2},{pb1|chart3},{pb1|chart3},{pb2|cha
rt3},{pb3|chart3},{pb4|chart3}
```

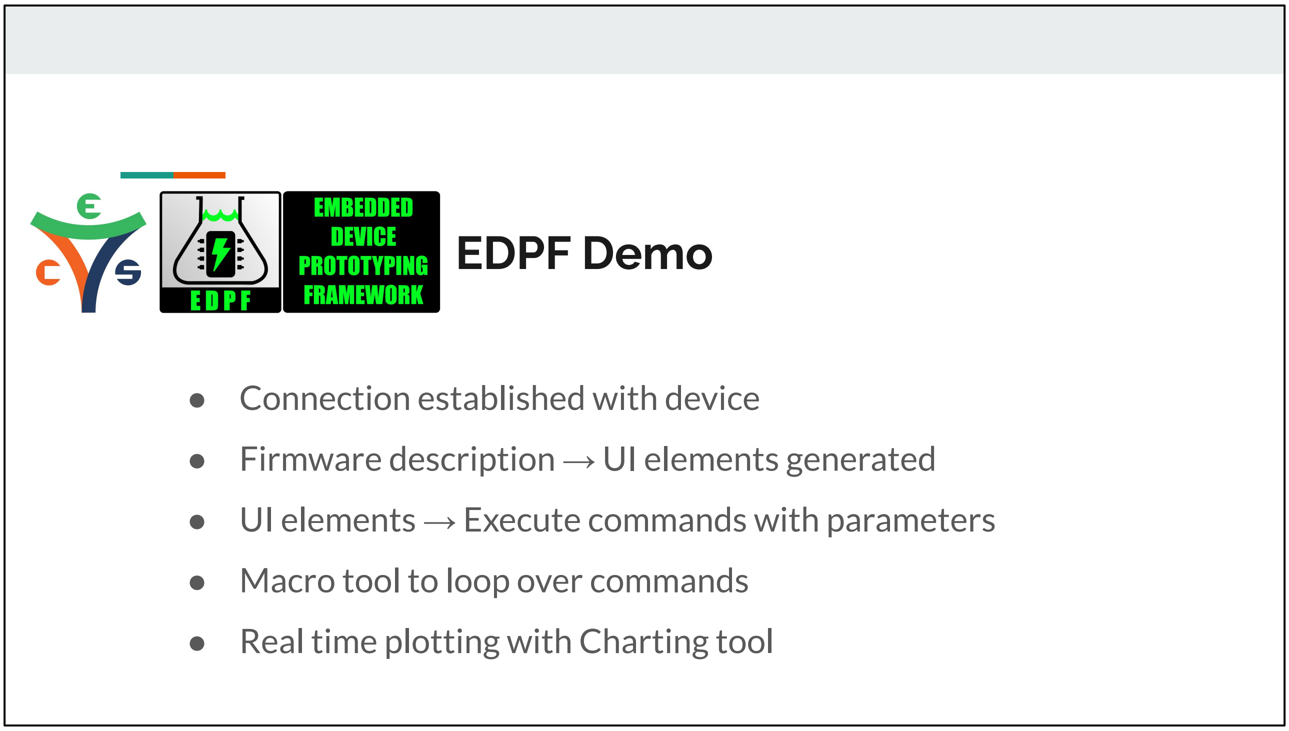
The expression string starts with the same prefix that was output by the "edpf_kit_read" command, and after the prefix comes a set of comma separated tuples with each tuple composed of a data variable name and a chart name to which the data will be plot.

It's really easy to make changes, such as renaming data values and combining or separating data values to different charts. Lets put a few more of these data values to the same chart:

```
kit:{x|chart1},{y|chart1},{d|chart2},{pb1|chart2},{pb1|chart2},{pb2|chart
2},{pb3|chart2},{pb4|chart2}
```

We will copy and paste this back to the Host UI

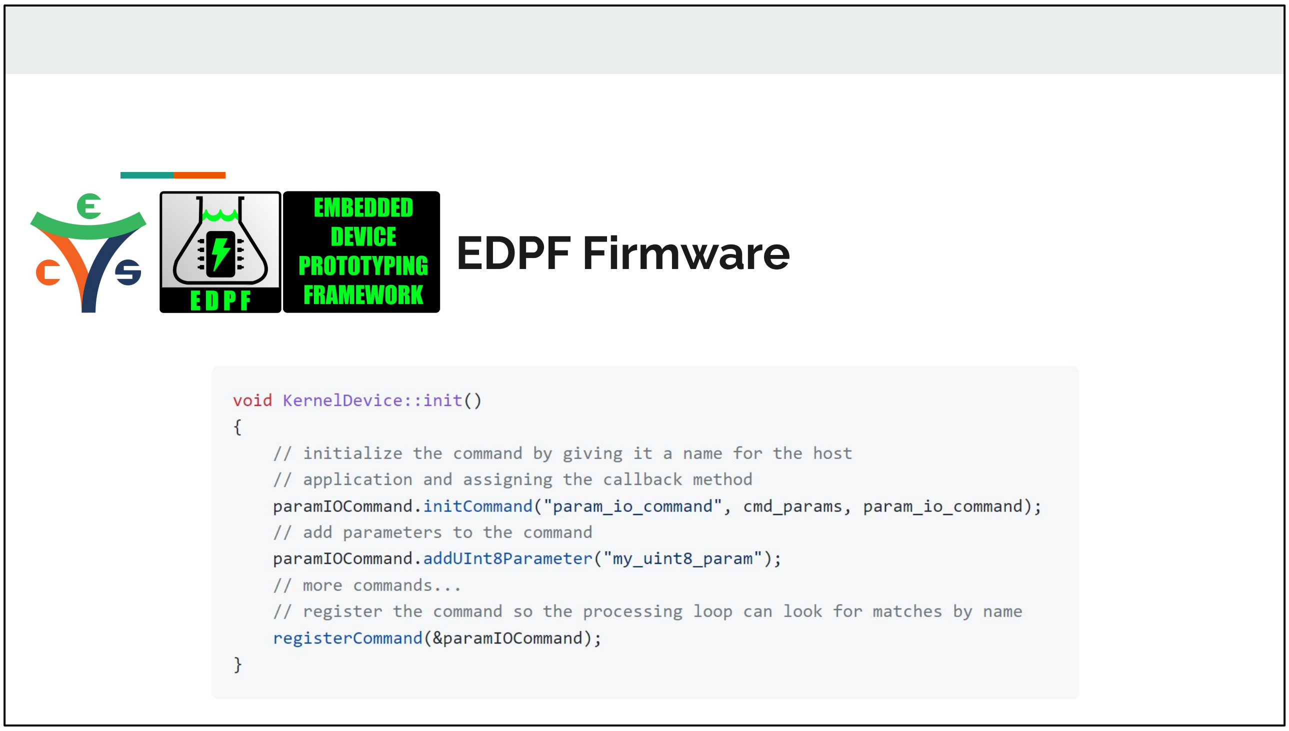
Now that we have collected the data samples we can save them to disk for further processing and analysis.



EDPF Demo

- Connection established with device
- Firmware description → UI elements generated
- UI elements → Execute commands with parameters
- Macro tool to loop over commands
- Real time plotting with Charting tool

And that's it for the EDPF Host UI demo. The Host UI established a connection with the device and configured UI elements to match the functionality of the firmware, and the functionality of the firmware is wrapped in commands which have optional parameters, we added some command calls to the macro tool and set the macro tool to loop repeating the commands, and last we used the charting tool to plot the data coming from the EDPF.



EDPF Firmware

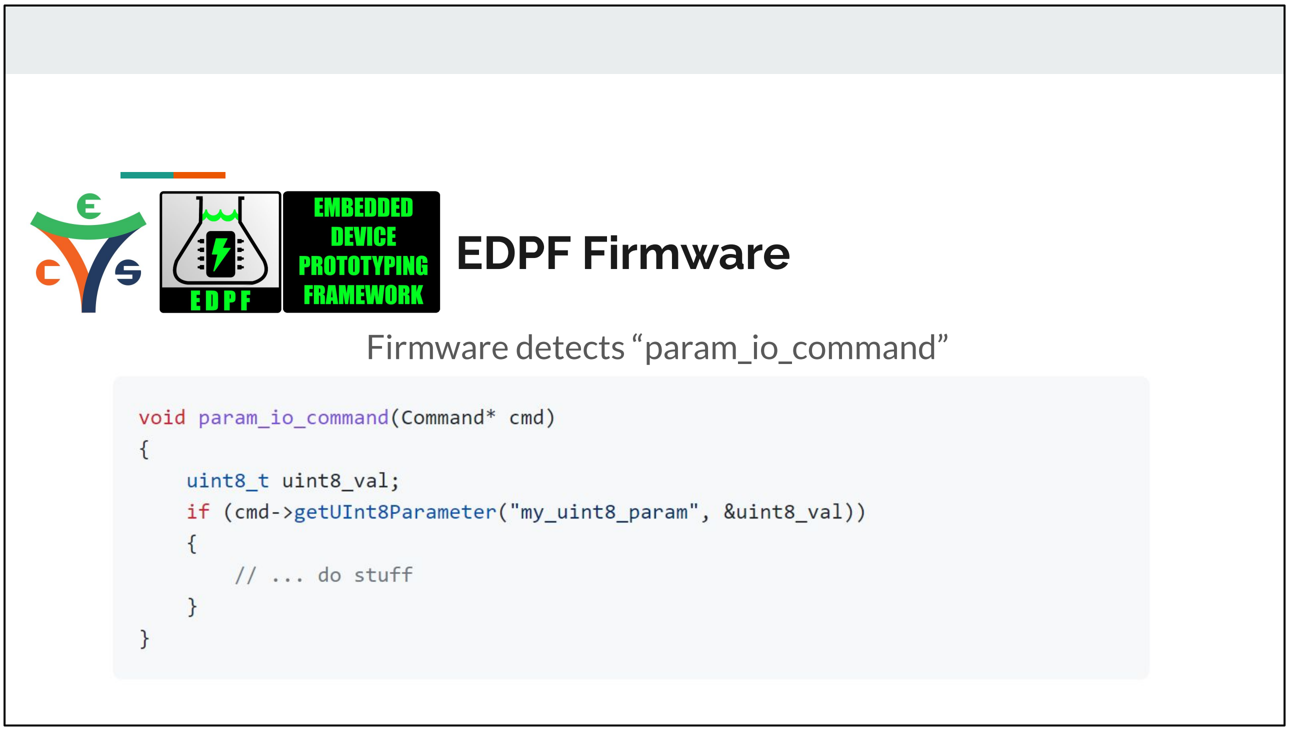
```
void KernelDevice::init()
{
    // initialize the command by giving it a name for the host
    // application and assigning the callback method
    paramIOCommand.initCommand("param_io_command", cmd_params, param_io_command);
    // add parameters to the command
    paramIOCommand.addUInt8Parameter("my_uint8_param");
    // more commands...
    // register the command so the processing loop can look for matches by name
    registerCommand(&paramIOCommand);
}
```

Now lets take a quick peek at the firmware that works in conjunction with the Host UI. The device needs to be able to describe its functionality, and the firmware does this through a registration process. Users of the EDPF firmware will write code to register commands and optional parameters to those commands. Think of EDPF commands and parameters much like software functions and function arguments, and the Host UI knows how to interrogate the firmware for these commands and provide a custom UI element for each command that it finds.

In addition, firmware commands have callback methods assigned to them that trigger when the host application sends the command name to the device.

Here is a simple example where a callback method is registered with a name, and one associated parameter that is an unsigned 8 bit integer.

With this bit of code in the firmware, the firmware knows how to describe itself to the host application. The host application will create a UI element and a named input field that is strongly typed, and the Host UI software does this with out of the box functionality so no UI development is needed to execute the callback method that includes data via the parameter.



EDPF Firmware

Firmware detects “param_io_command”

```
void param_io_command(Command* cmd)
{
    uint8_t uint8_val;
    if (cmd->getUInt8Parameter("my_uint8_param", &uint8_val))
    {
        // ... do stuff
    }
}
```

And here is the callback method. When the firmware detects the “param_io_command” token, it executes the command’s callback method. Inside the callback, the firmware retrieves the parameter by name and can make further use of the data with whatever custom firmware the developer coded.

Will My Innovation Facilitate Innovation in Others? Innovation squared???

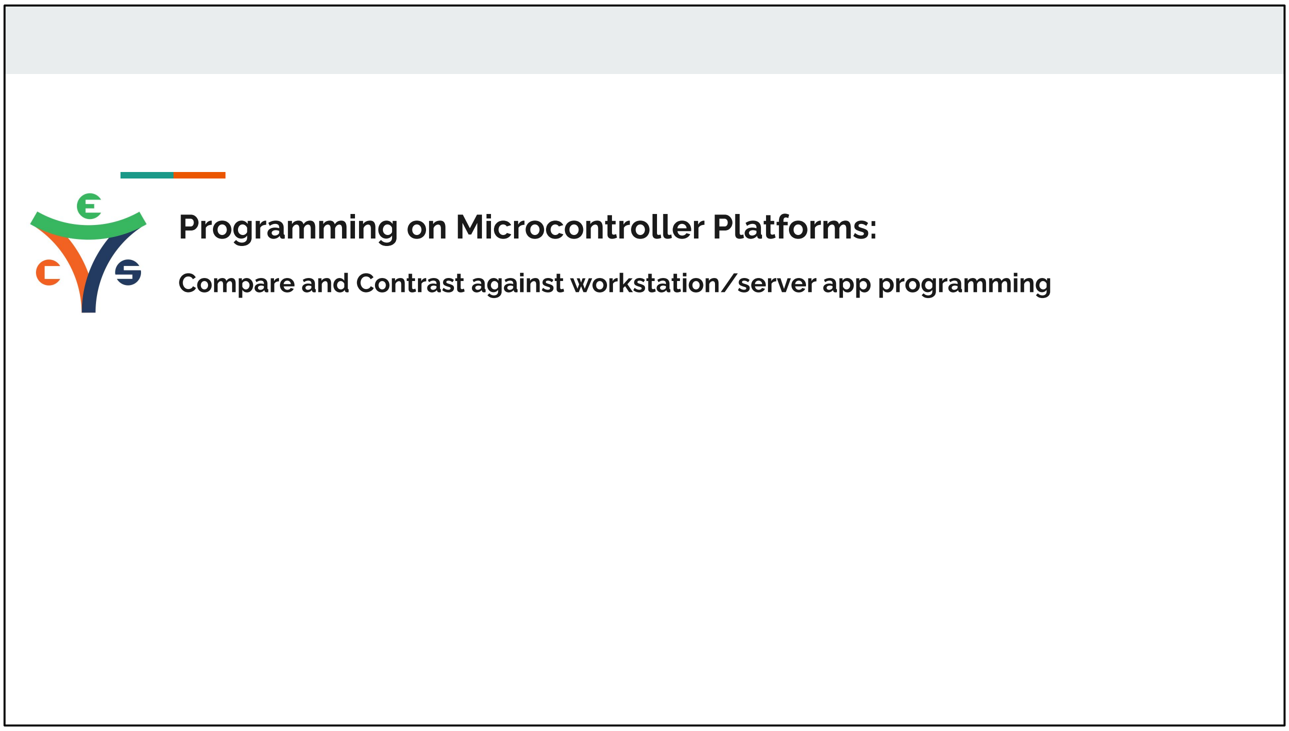
And that’s really about it for the EDPF -- the main focus is on reducing code needed to interact with an embedded device, and offering a great toolset to speed up development.

I’m using this project to cut down on my development time, and I’m hoping it will provide the same benefit to others. Since my project is geared towards prototypes and prototypes are by nature geared towards innovation, its almost as if my project is innovation with the goal of speeding up innovation.

Breather

OK, well that wraps up the part about the EDPF. Let’s take a breather to see what questions can I answer?

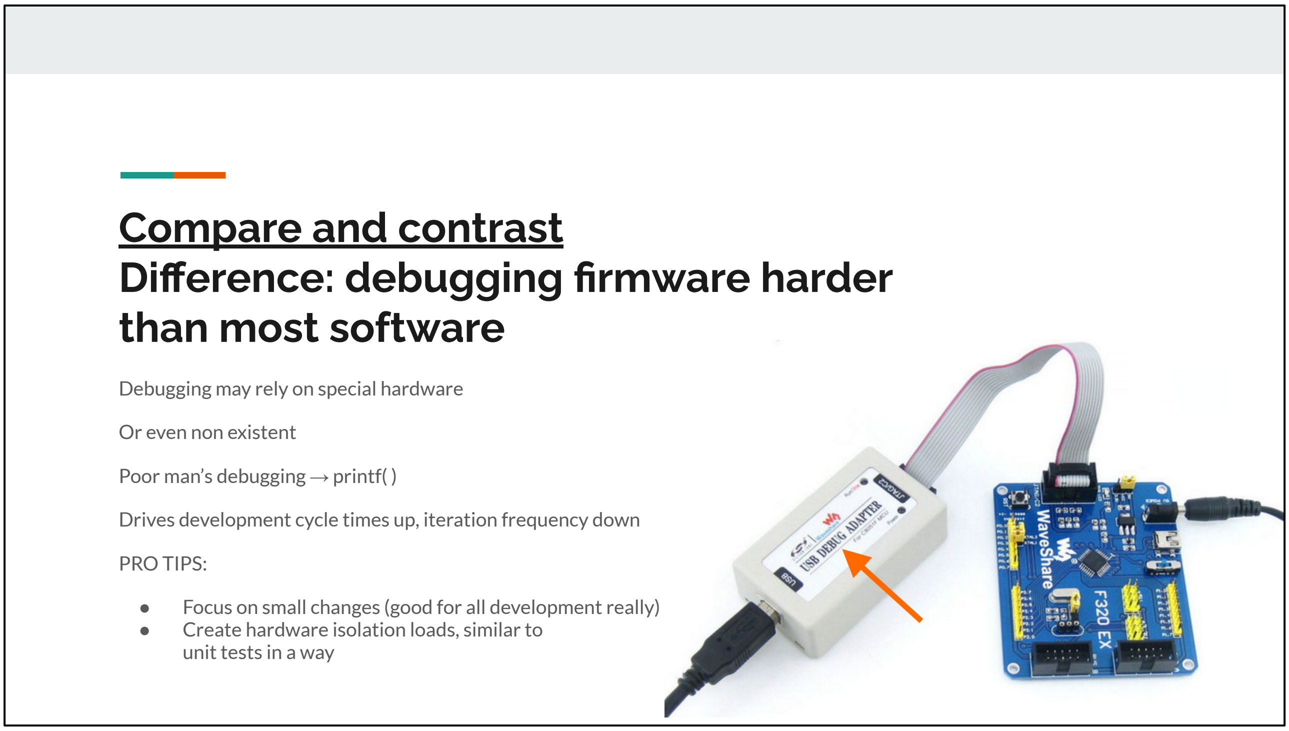
Alright, moving on now



Programming on Microcontroller Platforms:

Compare and Contrast against workstation/server app programming

Now, let’s go over a quick compare and contrast of embedded development versus more common workstation oriented development, and end with how to get started with electronics development on your own and some closing information.



Compare and contrast
Difference: debugging firmware harder than most software

Debugging may rely on special hardware

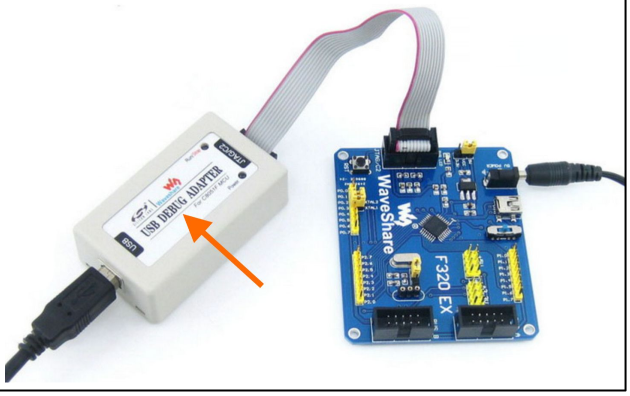
Or even non existent

Poor man's debugging → printf( )

Drives development cycle times up, iteration frequency down

PRO TIPS:

- Focus on small changes (good for all development really)
- Create hardware isolation loads, similar to unit tests in a way



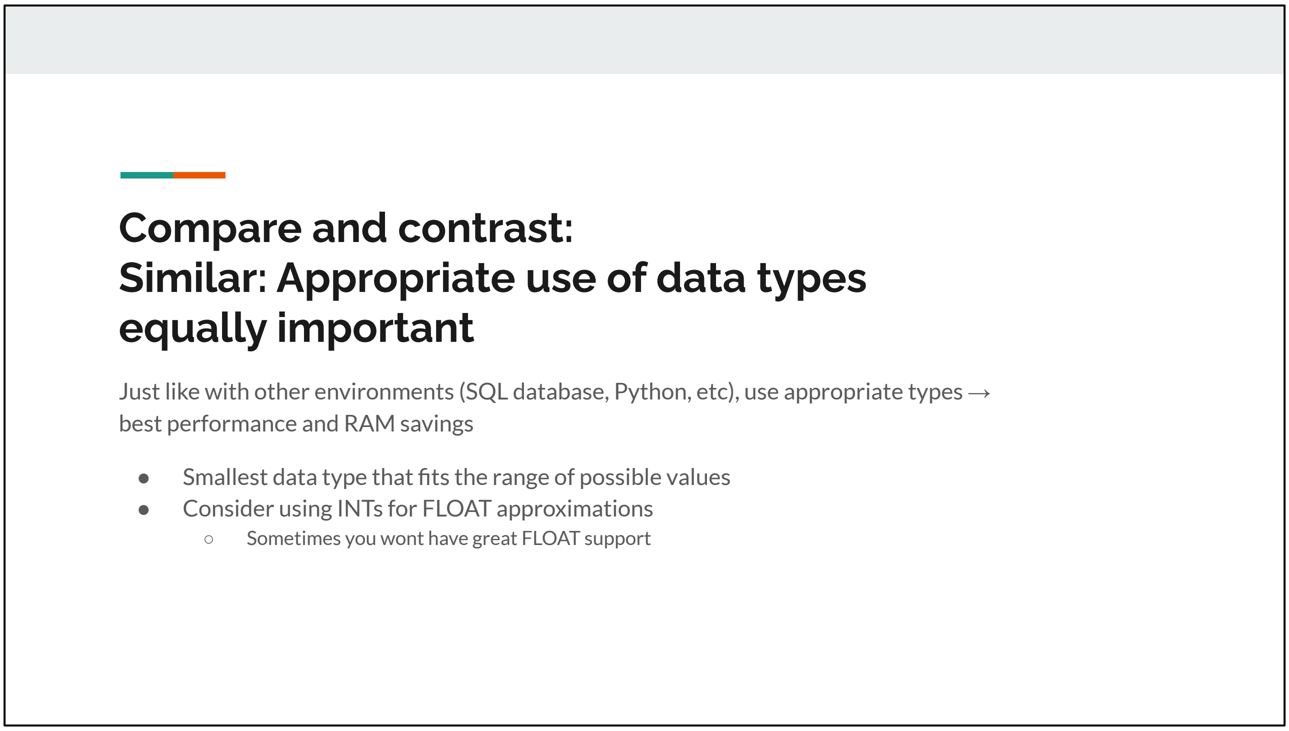
Debugging firmware tends to be harder than debugging software.

Some hardware platforms require the use of special hardware dongles to enable debugging, and this is usually on the higher end platforms. For the lower end platforms you may end up dedicating debugging through the only serial IO port, thus having to sacrifice a very important IO channel which you may need for your system operation. This often leads to the poor man's debugger, printf( ), as the only option which is not great. But sometimes its all you have. As you can imagine printf( ) debugging isnt ideal b/c you have pretty low visibility and it takes a comparatively long time to add print statements, redownload the code, and get to the code in question so this can drive development cycle time way up, which really slows down the iteration frequency and makes development time drastically increase. Avoid the printf( ) zone, if you can…but sometimes you cant.

Some other pro tips are:

- Create firmware isolation loads. Often when a piece of hardware is malfunctioning it can lock up the execution of your firmware and your device is bricked (bricked as in as useful as a brick). When your design becomes sufficiently complex, make firmware loads that perform self tests on different ancillary hardware components of your device. Think of these firmware loads like unit tests in that they isolate discrete functionality as much as possible and allow you to hone in on the problem.
- Another tip -- Focus on small changes -- with less visibility into what is going on with the code, keeping your deltas for each iteration minimized is best.

- Really this is a best practice of good software development anyways, but remember it even more so for firmware development.



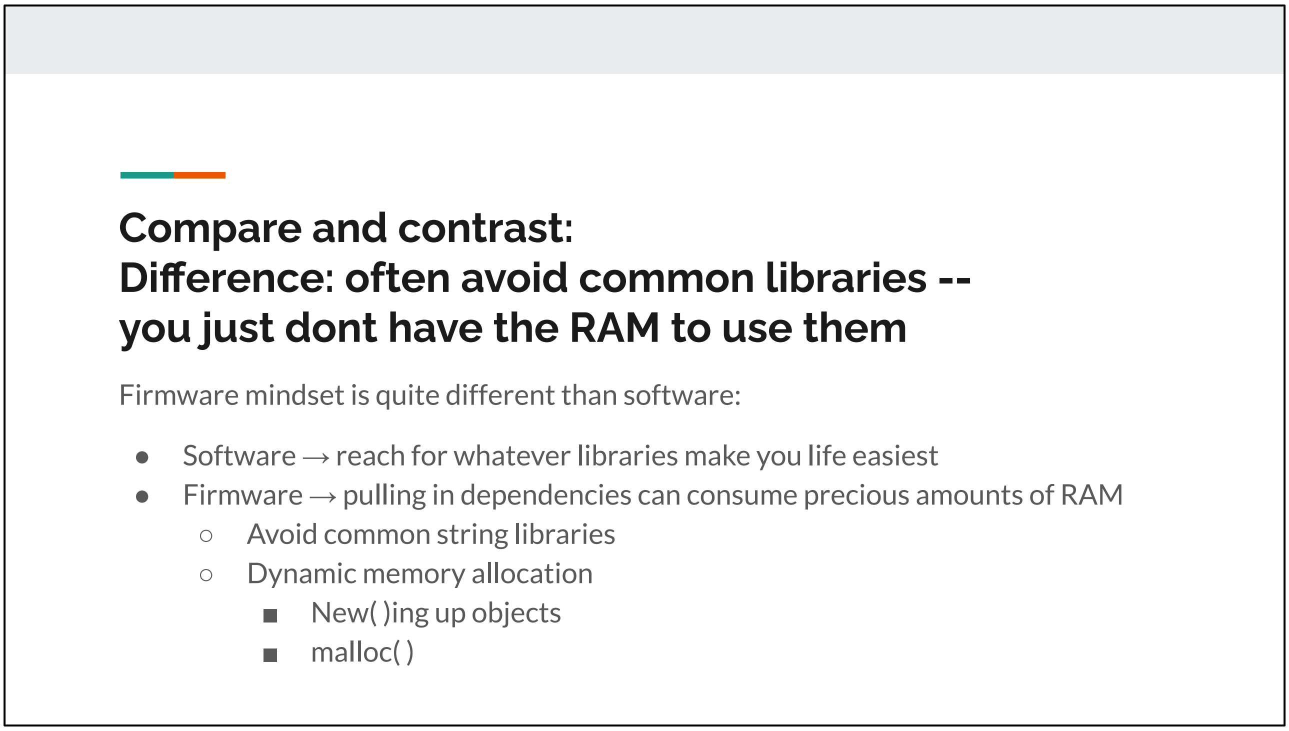
Compare and contrast: Similar: Appropriate use of data types equally important

Just like with other environments (SQL database, Python, etc), use appropriate types → best performance and RAM savings

- Smallest data type that fits the range of possible values
- Consider using INTs for FLOAT approximations
    - Sometimes you wont have great FLOAT support

As far as similarities, appropriate use of data types is something that benefits you really well in embedded development, just like other forms of development. Some things that might seem inconsequential on workstations like saving INTs to FLOATs and over using strings can have big performance impacts like gobbling up precious memory resources or causing needless delays that are a lot more painful on embedded devices. Some platforms won't have great FLOATing point data type support, and FLOAT operations can be orders of magnitude slower than INT operations.

Also, many data type principles of workstation or database programming apply to embedded programming, for example -- using the smallest data type possible to fit your data is usually the best approach.



Compare and contrast: Difference: often avoid common libraries -- you just dont have the RAM to use them

Firmware mindset is quite different than software:

- Software → reach for whatever libraries make you life easiest
- Firmware → pulling in dependencies can consume precious amounts of RAM
    - Avoid common string libraries
    - Dynamic memory allocation
        - New( )ing up objects
        - malloc( )

Differences again -- Firmware programming is quite different than most software programming, in that most often with software programming one usually reaches for whatever libraries you have available to cut down on the amount of code. Pulling in common software dependencies in firmware can have huge memory ramifications, and something as simple as pulling in different data types like ‘strings’ from a standard library or using dynamic memory allocation can rapidly consume constrained resources.

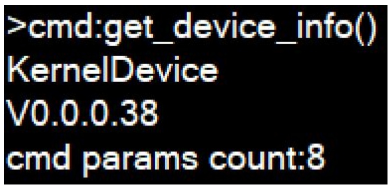
Instead of using dynamic memory allocation and strings, you often use raw character arrays and fixed sized memory buffers.

Compare and contrast:
Similar: versioning is key

Software → client software is keyed to the APIs it accesses

Firmware → host machine software is keyed to the device firmware

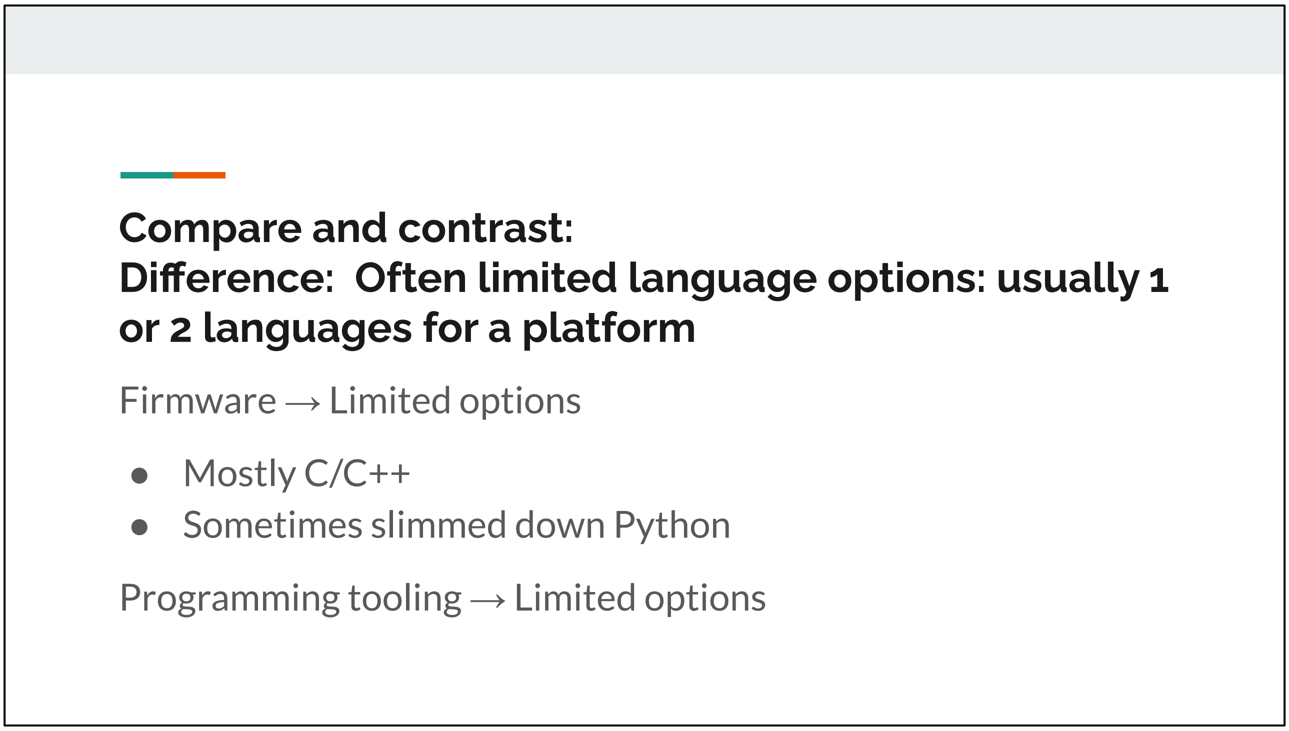
- Easy to forget to re-download the firmware after making changes



```
>cmd:get_device_info()
KernelDevice
V0.0.0.38
cmd params count:8
```

There is similarity in that versioning is key.

Just like software clients that access APIs and the compatibility aspects that apply, the general principle applies to host machine software that interfaces with firmware. I would go one step further and say knowing what firmware is running on your device is easy to mess up if you are not diligent, so creating a mechanism to know that you are running the absolute latest is key. Hardware can malfunction in funny ways, like the firmware download can silently fail or your storage device no longer accepts writes, that or you get in the habit of simply resetting the device without actually downloading firmware updates you just coded, and if you make this mistake enough times you create mechanisms to know that your changes are indeed executing on the device. For this reason, the EDPF reports the firmware version on command. My mantra, is when in doubt, bump the version number, redownload the firmware, and check the echoed back version number. I have gotten into the habit of changing the firmware version on almost every firmware change.



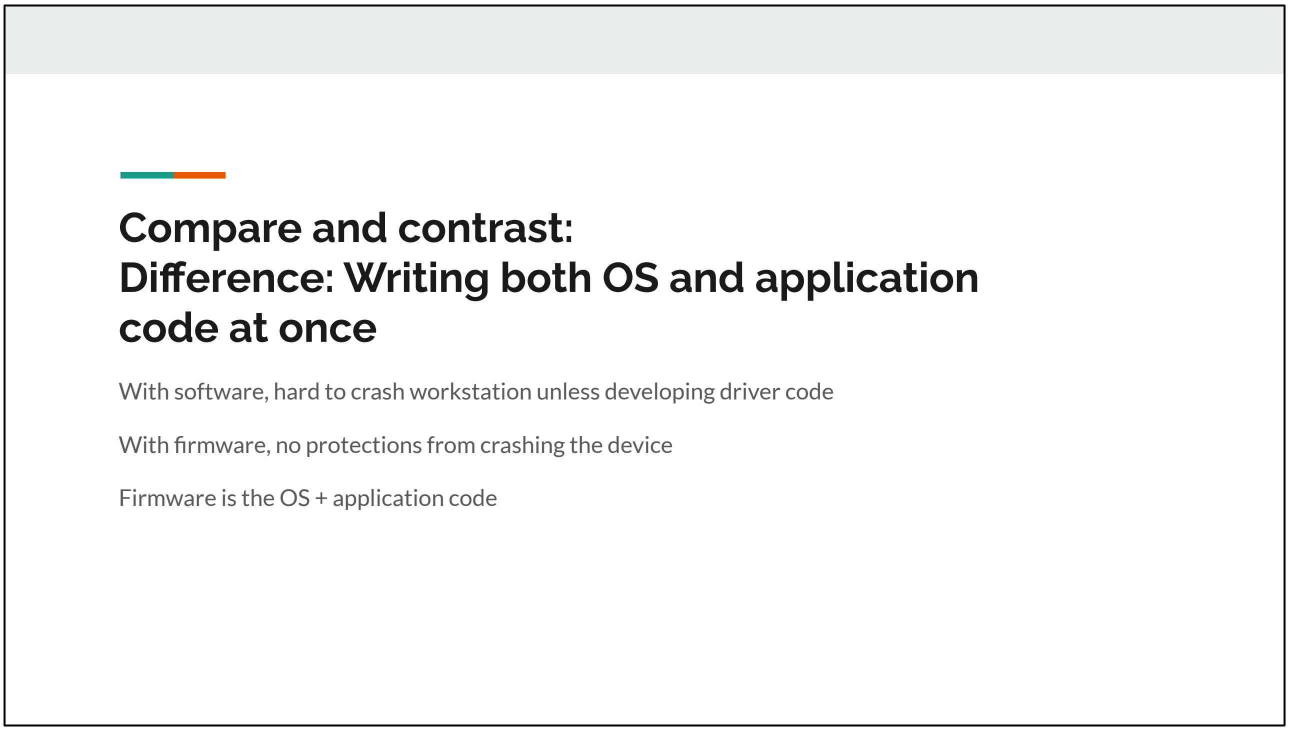
Compare and contrast:
Difference: Often limited language options: usually 1 or 2 languages for a platform

Firmware → Limited options

- Mostly C/C++
- Sometimes slimmed down Python

Programming tooling → Limited options

Another difference that you will find with programming against embedded devices versus other platforms is that your language and tool options are likely to be very constrained. C and C++ dominate, with a much smaller number of platforms supporting slimmed down versions of Python. As far as tooling, you may be left with only 1 or 2 practical choices provided by the hardware vendor for developing on the device platform. So tooling support is definitely key when picking an embedded platform.



Compare and contrast: Difference: Writing both OS and application code at once

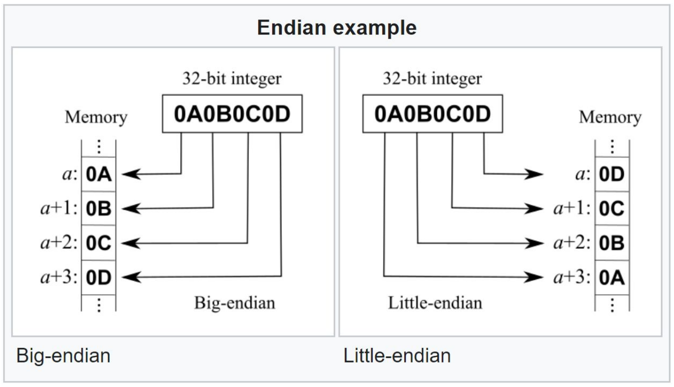
With software, hard to crash workstation unless developing driver code

With firmware, no protections from crashing the device

Firmware is the OS + application code

Unlike most other forms of development, writing code for embedded devices means you are writing a blend of application and OS code at the same time. With most forms of software development its hard to crash the device you are running on, and there are layers of protection in the OS that keep you from crashing the system. When developing firmware, you are very close to the bare metal and can pretty quickly get into spots where hard resets are your only option.

Compare and contrast:
Difference: Different processors, different data alignment

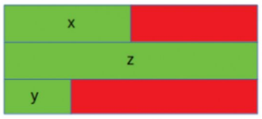


```
struct A {

    // sizeof(int) = 4
    int x;
    // Padding of 4 bytes

    // sizeof(double) = 8
    double z;

    // sizeof(short int) = 2
    short int y;
    // Padding of 6 bytes
};
        printf("Size of struct: %ld", sizeof(struct A));
```



```
Size of struct: 24
```

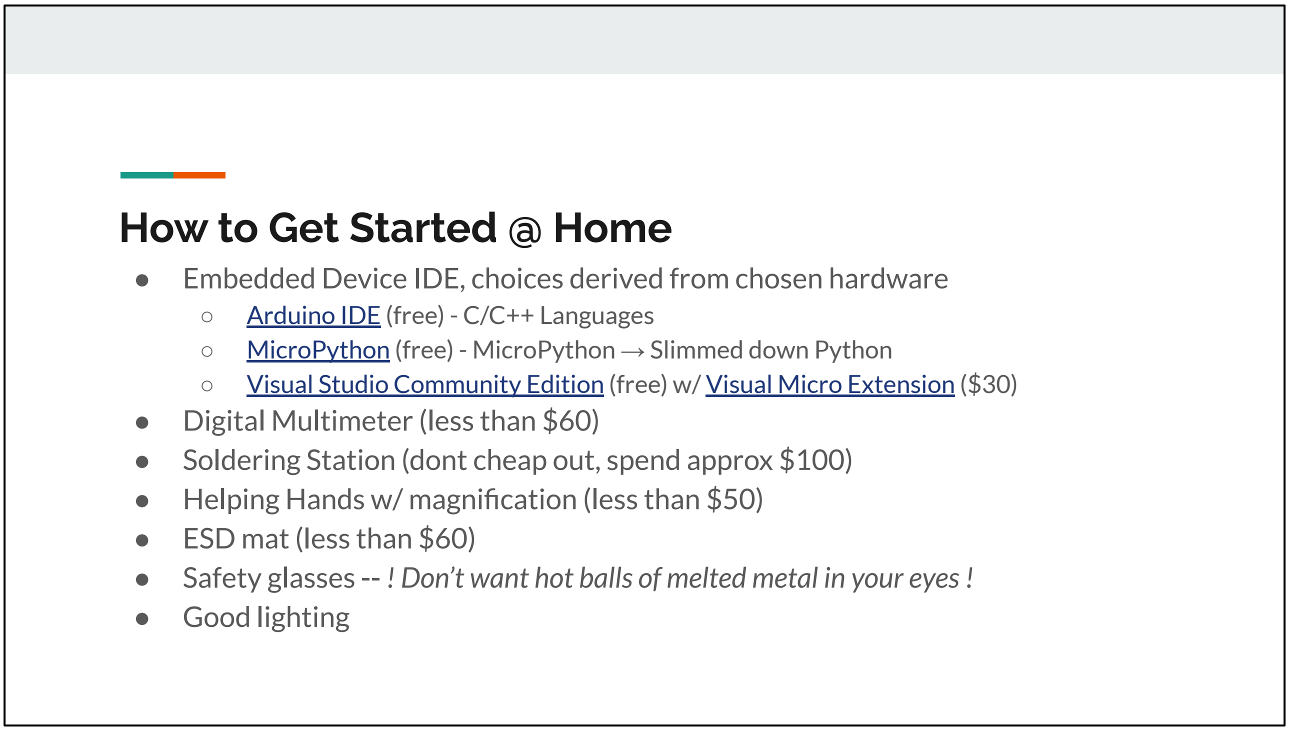
Q: Who here is familiar with endianness or structure member alignment? If you aren't I consider you lucky.

Endianness is the order of bytes of a digital word in computer memory. Said another way, the way a variable looks in memory could be completely flipped from one processor architecture to another. Looking at this example on the left, different endian processors will have the bytes of variables arranged backwards from each other.

Another surprising quirk is that different processors will arrange structure members differently in memory, injecting different amounts of padding in ways that can be unintuitive. So on the right you can see how a data structure with 14 bytes of storage consumes 24 bytes of memory for that processor, and it could be different with different processors.

Both of these attributes come into play when marshalling data back-and-forth between different processors in a system. I've run into these differences when a workstations communicates with USB or PCIe devices.

Be prepared to understand the impacts of bus width and endian type of your microprocessor



How to Get Started @ Home

- Embedded Device IDE, choices derived from chosen hardware
  - Arduino IDE (free) - C/C++ Languages
  - MicroPython (free) - MicroPython → Slimmed down Python
  - Visual Studio Community Edition (free) w/ Visual Micro Extension ($30)
- Digital Multimeter (less than $60)
- Soldering Station (dont cheap out, spend approx $100)
- Helping Hands w/ magnification (less than $50)
- ESD mat (less than $60)
- Safety glasses -- *! Don't want hot balls of melted metal in your eyes !*
- Good lighting

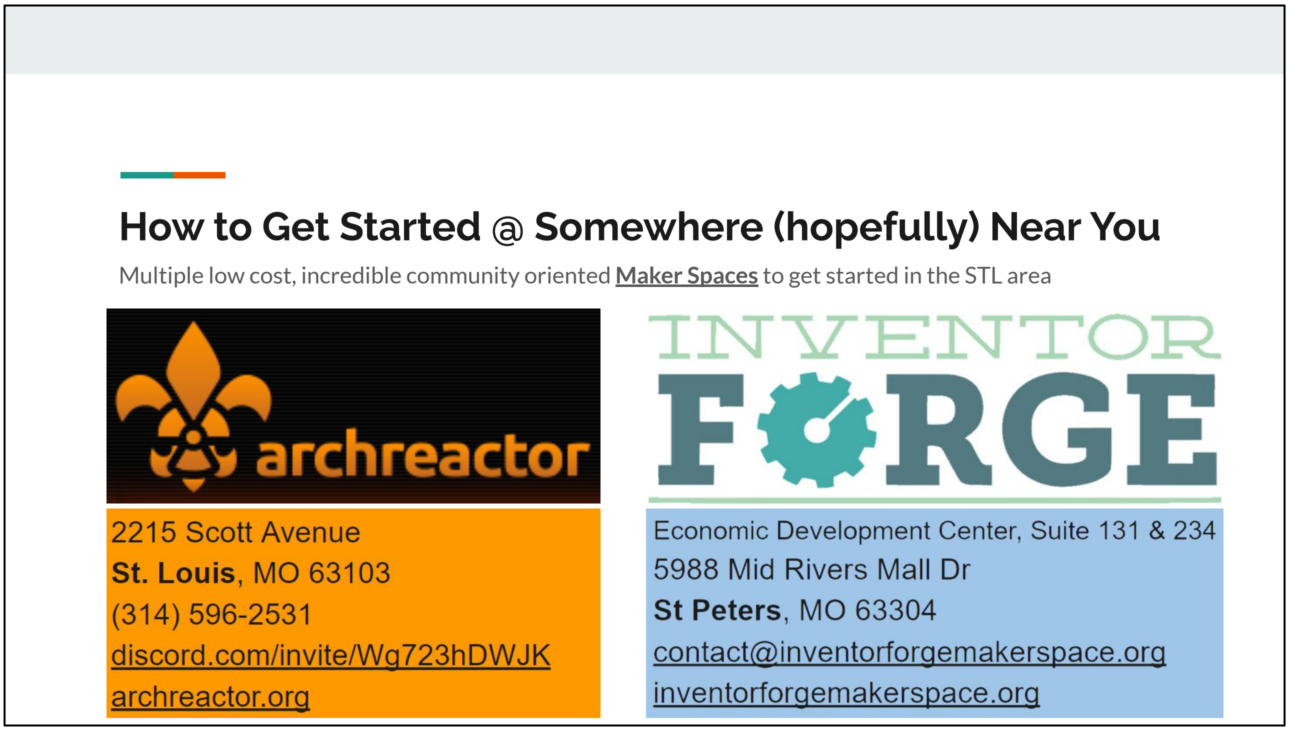
How to get started --

Don't let any of the differences with embedded device programming dismay you -- the materials, tooling, and devices have continued to get better and more and more approachable.

In addition, I put together a quick cheat sheet of supplies and software you could use to get started at home.

<go over components>

With just a few hundred dollars you can get yourself a pretty nice setup.



How to Get Started @ Somewhere (hopefully) Near You

Multiple low cost, incredible community oriented **Maker Spaces** to get started in the STL area

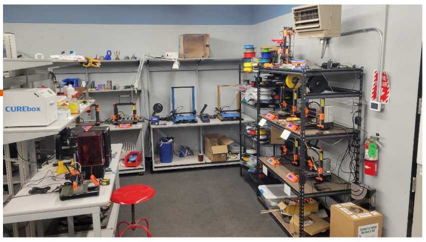
archreactor

2215 Scott Avenue
St. Louis, MO 63103
(314) 596-2531
discord.com/invite/Wg723hDWJK
archreactor.org

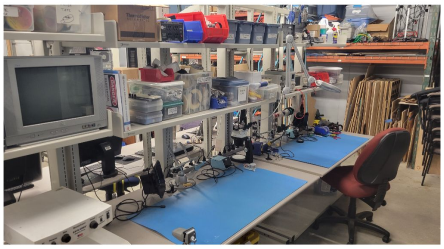
INVENTOR FORGE

Economic Development Center, Suite 131 & 234
5988 Mid Rivers Mall Dr
St Peters, MO 63304
contact@inventorforgemakerspace.org
inventorforgemakerspace.org

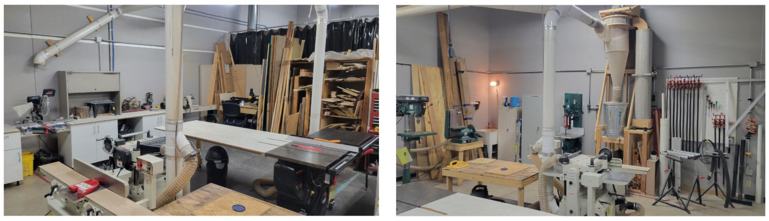
If you are looking for even lower startup costs and live towards the St. Louis area, there's a few makerspace groups I want to point out that have all of that equipment along with tons of additional tools, plus a really awesome member base with diverse interests and hobbies. There are at least two makerspaces in the St. Louis and St. Charles County area that provide 24 hour access to everything you would need.


3D Printers

Lasers


Electronics

They have 3D printers, lasers, electronics stations



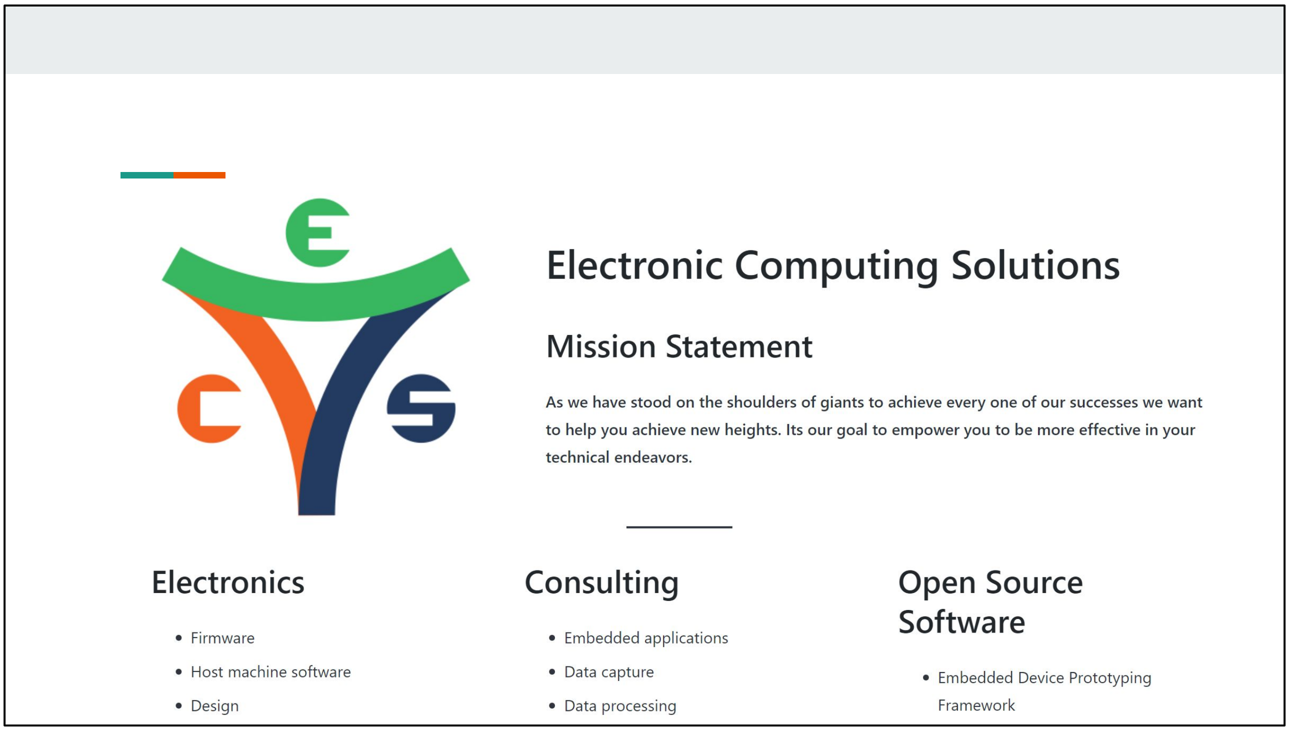
Wood Working

Create games, run you business, fix stuff, feed your passion

Tons of wood working and even metal working tools. And members use the equipment to create games, run businesses, fix stuff around the house, or just to have a hobby that feeds their passion.

They also do community based activities like promoting STEM to local youth, social activities for its members, and visit conferences together.

So I highly recommend checking out either of these groups -- they are great people and I think the benefits of the membership, which typically ranges from $40 to $60 a month, is well worth the cost.



Electronic Computing Solutions

Mission Statement

As we have stood on the shoulders of giants to achieve every one of our successes we want to help you achieve new heights. Its our goal to empower you to be more effective in your technical endeavors.

Electronics

- Firmware
- Host machine software
- Design

Consulting

- Embedded applications
- Data capture
- Data processing

Open Source Software

- Embedded Device Prototyping Framework

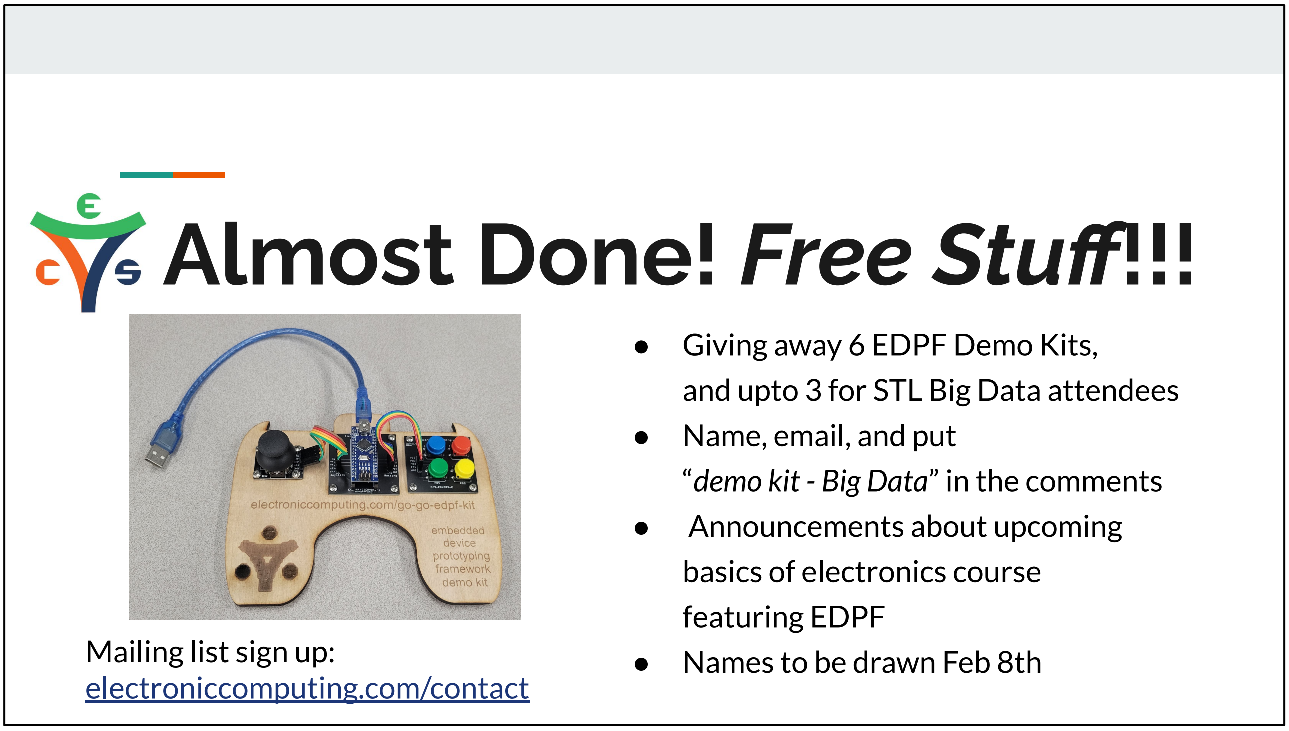
Now for the shameless plug -- in my day job I'm a hybrid of a software engineer, data engineer, and manager -- I've also started my own company, and it began with the idea of making development easier for electronic devices.

My company is Electronic Computing Solutions, and its mission statement is to empower you to be more effective in your technical endeavors. In addition to providing software and firmware for end users, I've also had the fortune to provide technology solutions for engineers -- boosting their productivity -- helping them complete their work faster. And that's where my company focuses its software development efforts -- helping others be more productive.

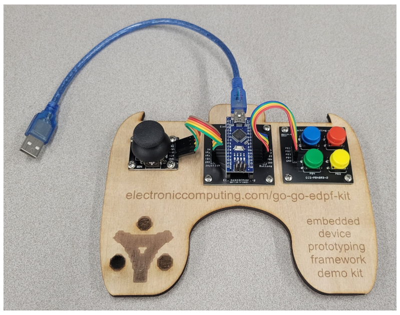
And to give you some insight as to how I operate as an engineer, I've been asked by others that see me work "Are you pumping your fists in the air in celebration?" and my answer is "Yes, I just automated this awful manual process we have been suffering through". When that new functionality works for the first time and I get to provide a boost to others, it gives me that engineering dopamine hit.

So if you know someone that is looking for help with their electronics and software project, please have them reach out to me and we'll talk. I'm most easily reached on LinkedIn.

Also, if you think the EDPF project can help you or your company, I'd be happy to share all that I can on that front and see what use cases the EDPF can facilitate.



Almost Done! *Free Stuff*!!!


electroniccomputing.com/go-go-edpf-kit
embedded device prototyping framework demo kit

Mailing list sign up:
electroniccomputing.com/contact

- Giving away 6 EDPF Demo Kits, and upto 3 for STL Big Data attendees
- Name, email, and put "*demo kit - Big Data*" in the comments
- Announcements about upcoming basics of electronics course featuring EDPF
- Names to be drawn Feb 8th

Last thing to go over tonight -- I'm doing a giveaway for anyone that is interested in electronics. I have a mailing list for an upcoming basics of electronics course which will be taught in the St. Peters makerspace around April. Anyone who signs up for that mailing list will be entered into a drawing where I will give away 6 of these EDPF Demo Kits -- and up to 3 will be given to tonight's attendees assuming I get 3 signups.

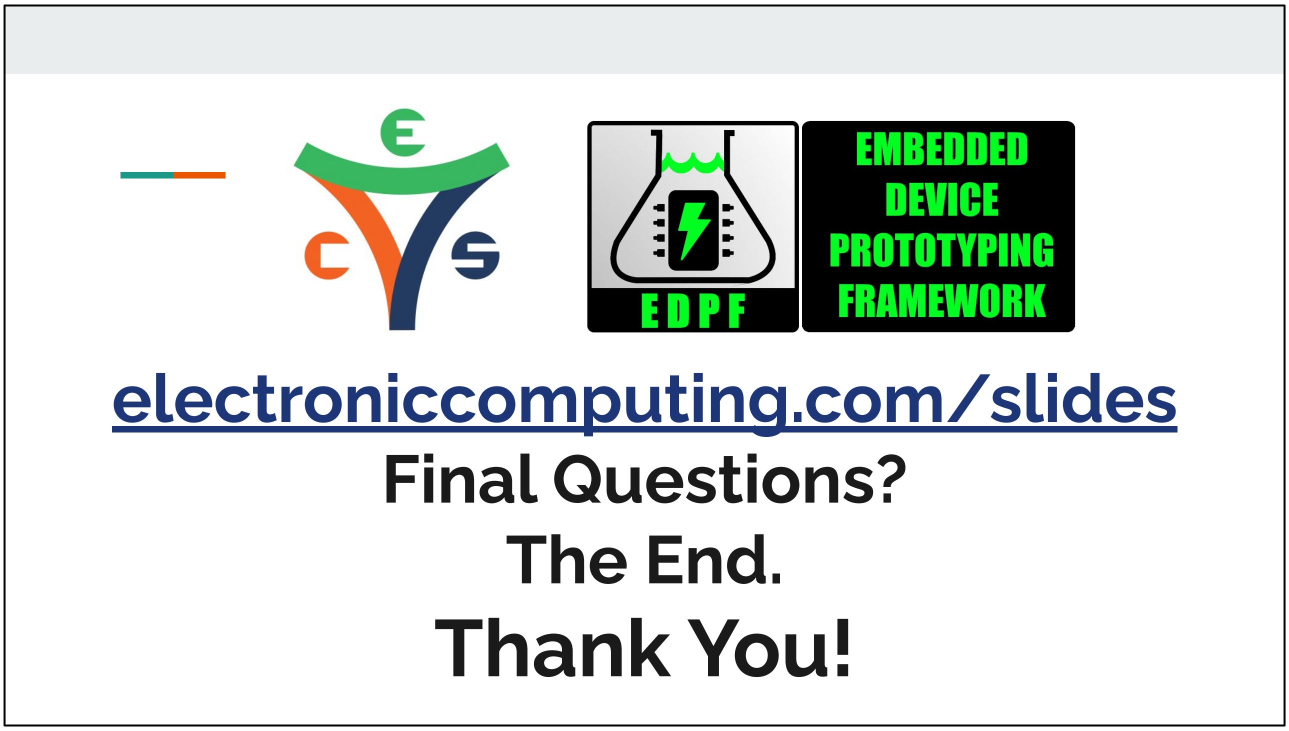
So please sign up if you are interested with the link at the bottom left: electroniccomputing.com/contact

Please put "demo kit - Big Data" in the comments so I know you attended tonight, and names will be drawn on February 8th.

Links

Mailing list sign up:
electroniccomputing.com/contact

- This slide deck - electroniccomputing.com/slides
- EDPF Project - github.com/nickhepp/embedded-device-prototyping-framework
- Giveaway sign up - electroniccomputing.com/contact
  "*demo kit - Big Data*" in the comments, names to be drawn Feb 8th
- Me at LinkedIn - linkedin.com/in/nick-heppermann
- STL area Makerspaces
  - St. Peters, MO - Inventor Forge Makerspace - inventorforgemakerspace.org
  - St. Louis, MO - Arch Reactor - archreactor.org



EDPF

EMBEDDED DEVICE PROTOTYPING FRAMEWORK

electroniccomputing.com/slides

Final Questions?

The End.

Thank You!

That’s all I had to cover this evening. What questions can I answer?