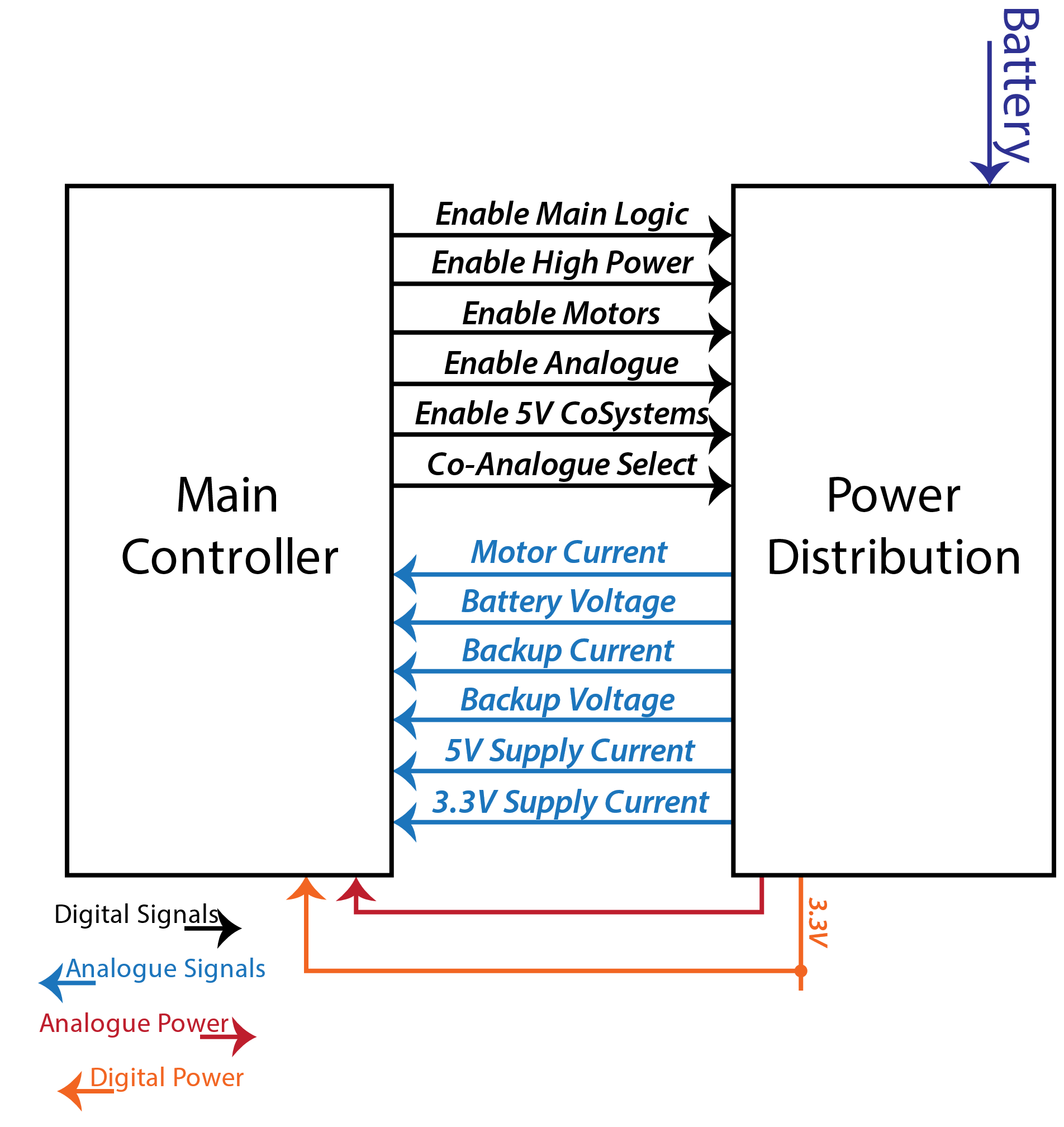
Off the Clock: Designing a Robot part 2

In my previous post, part one of this series, I ended on specifying the connections between the main controller and the power block. In this post I will finish that work, as it is the next logical step to verify if everything fits the processors I initially chose, or if things need to be moved around or even a processor needs to be replaced.

If you haven’t yet, I strongly suggest you read part one first, which lives here:

<https://medium.com/@asmyldof/off-the-clock-designing-a-robot-part-1-a3a37719c2c2>

To bridge the cliff-hanger neatly, I’ll recap the wiring description set at the end of the last post:

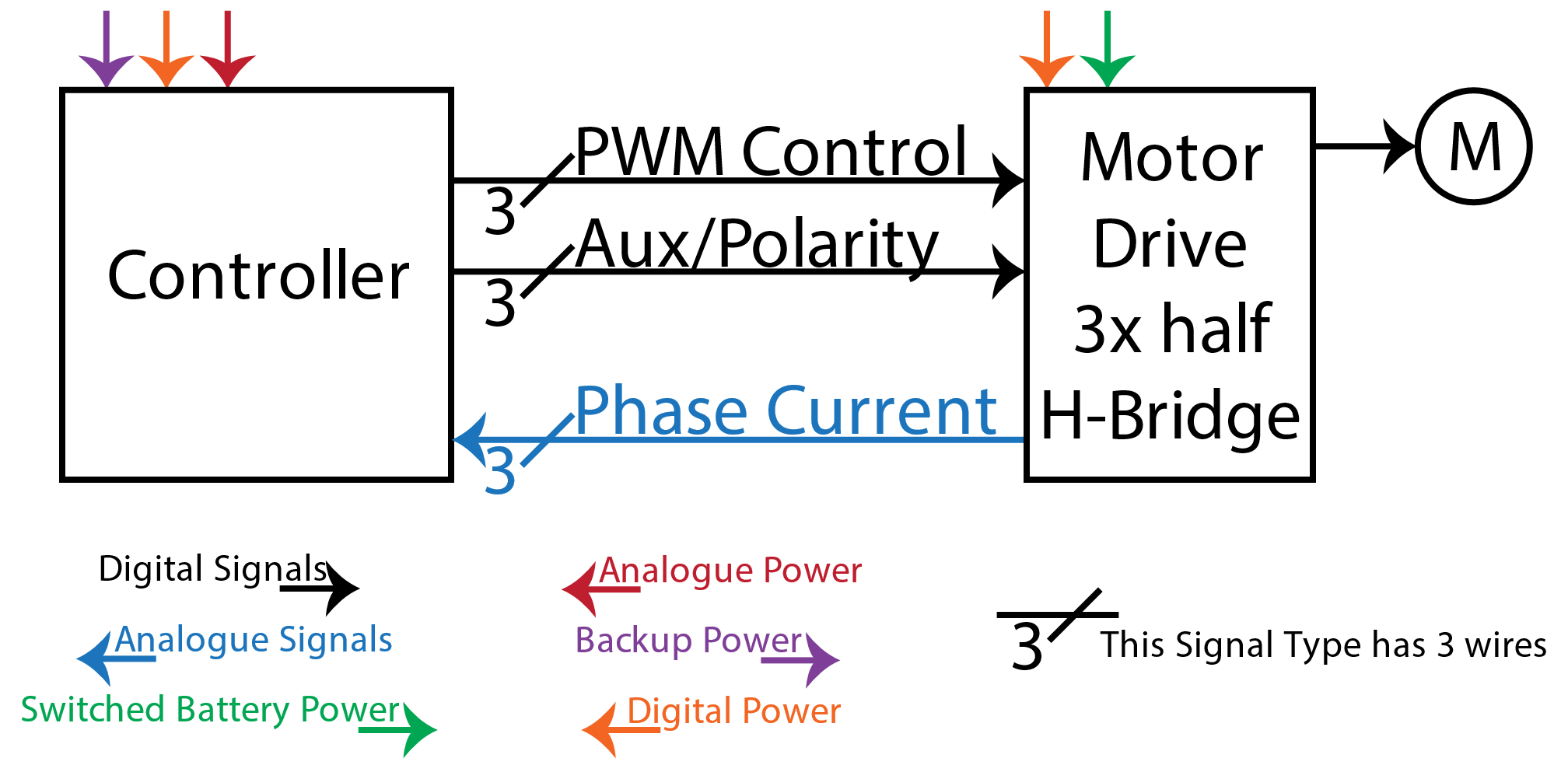


This image simply shows what kinds of control signals the Main Controller sends to the Power Distribution and what is measured back. Some of the analogue signals may be processed by hardware comparators or dedicated charge control chips at a later stage in the design, but for now we assume all that is done in Firmware, through the internal ADC. Since the backup voltage is also supplied to the expansion connection it is assumed for now we keep an eye on power consumption in that as well in some manner. Maybe the thing will trip a shut-off of the expansion at more than a few mA, or possibly there will be several alarm levels. This will also depend on the later choice of whether or not to add an actual backup battery or not. Right now, the backup power could also be a very-low-power linear regulator from the main battery, allowing the main processor to only “leach” a few dozen uA from it while sleeping. But I want expansions to also have the option to “stay ready” in a low power state when all main power is switched off, so the expansion will get a backup power wire, switched or not. The Co-Analogue Select is a provision I put in, in case I want to be able to select between two Analogue power and/or reference voltages for the Co-Processor.

## Controlling a Motor

To control a motor you need to decide what kind of motor you will be controlling. I choose to support three-phase and/or three-wire motors, which are normally controlled with on/off pulses to generate currents in the phases. Each successive set of currents puts the rotor at a known location, so by switching through given sets of currents at a specific speed, you can control the motor’s speed.

Instead of going for the normal approach of being able to actively control the high-side switch (the switch in the positive supply) and the low-side switch (the switch in the ground rail / negative supply), I’m doing it a bit differently. At least at first thought:



The idea is that the Motor Drive will be turned on and off at high frequencies – at least 10kHz, preferably more – while the polarity is switched by software at much lower speeds. This allows me to control the current more directly, hopefully allowing me to make sine-wave currents. There’s a half-H-bridge driver that seems to allow this behaviour, PWM’ing its “Output Disable” pin, while toggling the Polarity pin. But, since this is not normal use of the chip, I’d need to test this before committing to it. Luckily, I put a bunch of them on the Wurth Demo Board I mentioned in my first Off the Clock post:

<https://medium.com/@asmyldof/off-the-clock-3069af1cdd63>

Normally I’d test this by quickly manually etching a demo-set-up over two days or less and test it into destruction, but since I have a Demo Board with this set-up capable of driving up to 10A per channel (in theory), I can jump right to the testing right after this series of macro-design blogs. Don’t worry, I’ll write about that too.

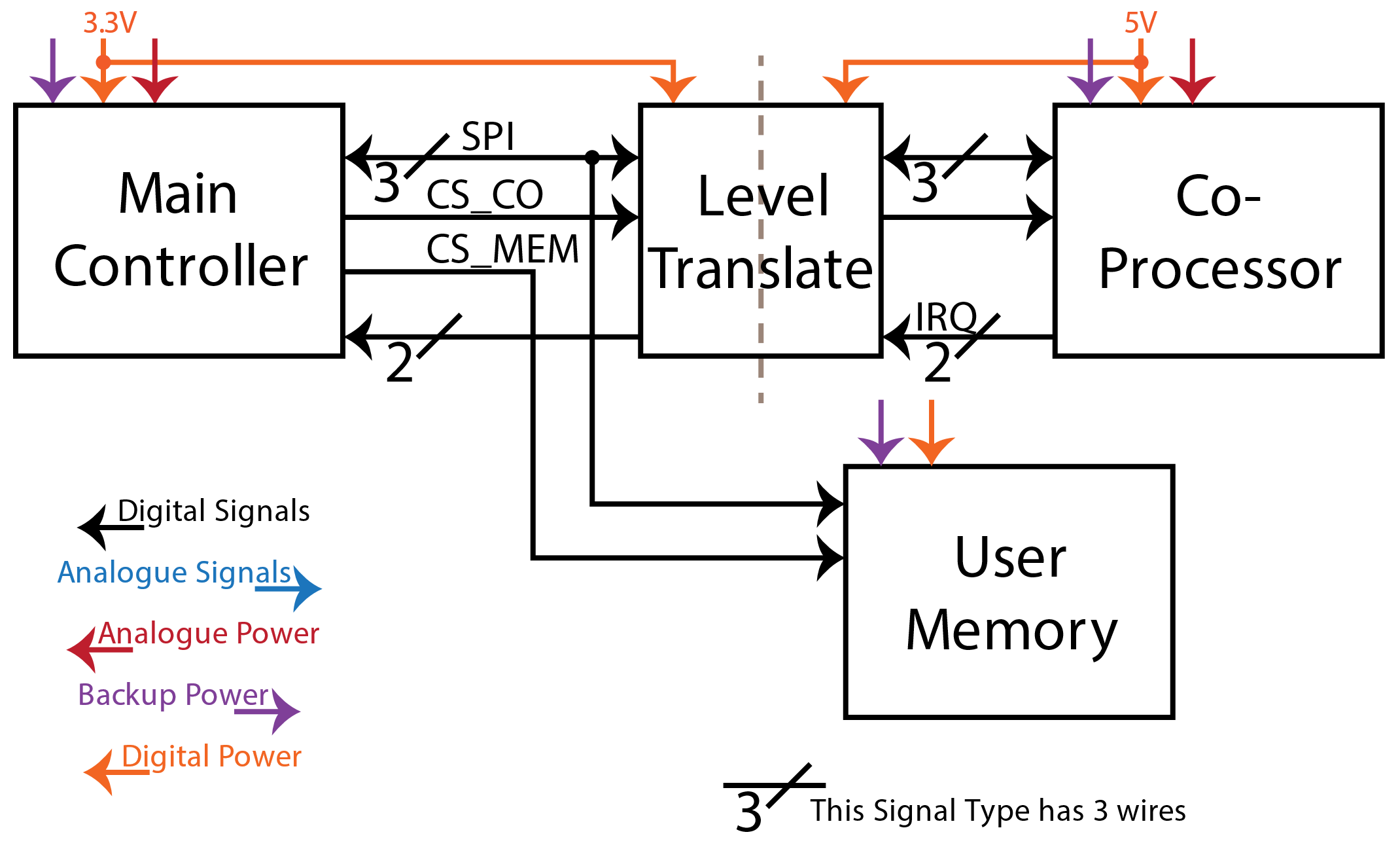
## Positional Sensors

I will need to add some sensors to the Robot so that it will be able to know just about where it is. I haven’t really figures out which kind of sensor to use there, but I intend to keep a dedicated SERCOM for those sensors. A SERCOM as a device in an Atmel that can be configured to be any of many kinds of Serial ports.

I have planned seven I/O pins, which allows me one SPI bus, one I2C bus and a spare line for interrupts or such. This may seem confusing, “One SERCOM” followed by “one SPI bus and one I2C”. This is one of the most fun things about the latest generation of Atmel chips. The internal devices, like the SERCOM, can be routed to several different groups of pins, or spread out over those pins without any penalty and they can also flip around the function of the pins willy-nilly. So with a bit of careful planning in the Firmware I can use the one SERCOM for both SPI and I2C on the same pins, or different pins. Of course, I hope all sensors will be available in SPI, or if there are many all in I2C, since I2C needs only two wires, even if you have many devices, so long as they can all have a different address. And even if that’s not the case it’s likely I will have two SERCOMs to spare for them, but I prefer to be prepared in my hardware design to stay flexible in my Firmware. Which is also why I will look to reserve a set of four pins that can connect to the same SERCOM as the other three, but also support another SERCOM. Knowing Atmels of this generation I expect I can do that fairly easily and keep the choice for one or two SERCOMs completely for later in the actual pin-planning and/or Firmware stage.

## Internal Bus

I was tempted to call this the Front-Side Bus (FSB) for a while, borrowing from main-stream computing, but I hate such blatant misnomers, since the FSB would be inside the main controller, along with the “North-Bridge” for the most part. The co-processor would be the South-Bridge for the most part. So this would be the un-named internal bus.



As you probably expected the internal bus is a bus between the Main Controller and the Co-Processor. I chose an SPI bus, because it doesn’t use a lot of wires, but still allows me to transport around one mega-byte per second. For mass storage systems that would be horrible, but communicating between two embedded processors only that’s not bad at all. Obviously, you also noted the level translation. As I touched on in the previous post I chose to power the Co-Processor with 5V, while the Main Controller is powered by 3.3V, this allows me to use both 5V and 3.3V devices everywhere I like, without having to pepper everything with level translations. This does mean that the two processors need to be connected through a level-translator.

The surprise here is the extra User Memory. Since either controller only has internal Flash and RAM, I’d have to count on the back-up power to keep highly variable things available, since writing to the flash a lot will damage it. There’s a cool “new” (it’s starting to get old by now) technology that may help me there: FRAM. It works like RAM to the outside world, but uses a special property of a material initially invented for use in capacitors: The insulation material can “remember” whether the positive charge was up or down for a very long time. So the bits in the FRAM will remember their data even if there is no power. So I plan on adding a chip of FRAM to the SPI bus, where configurations and even special user-programs can be stored and recalled with many billions or even trillions of write-cycles.

## Expansion Header

First off, I’ll mention I reserve the right to later decide to have two expansion slots. One “on board” to expand internal functionality with things like GPS boards or other such and the other one being the expansion I am planning now: A port at the front edge of the Robot, where it can easily pick-up small adapters that allow it to do jobs around the lab. I intend to find a mechanical solution and a connector that will allow the Robot to pick them up itself, since that’s the most fun.

For the front expansion (and likely the on-board one as well), I want to have one 3.3V and one 5V SERCOM, as well as two spare I/O pins from both Controllers. It’s likely any I/O pins left will be equally split between the on-board and the front-side connector, to keep all options open for as long as possible. Later we’ll work on a protocol to let expansions tell the controller what they expect from the connections and what the main bus is; the 3.3V one or the 5V one. We’ll make an attempt to design that protocol so that we can make new expansions without having to re-program the Main Controller, which may include the User Memory discussed above, as well as the Main Controller’s ability to run code from RAM.

## Debugging Headers

Now, a last, but very important thing to consider is the programming connection for both controllers. In principle in a production unit (if it ever comes to that) you would have a bootloader that programs the Main Controller and then the Main Controller can update the firmware on the Co-Processor through the Internal bus. Of course, during development of the code we want to be able to look into the controllers, so the Alpha version of the hardware needs a connector for a debug interface. For ARM controllers the available systems are JTAG and SWD. JTAG is a system with four or five wires, four being the most common and minimum number, but some controllers also implement a fifth wire. JTAG allows you to chain many devices in a row on one single connector, or “daisy-chain” devices as that’s called. JTAG is also used by many other types of chips, FPGAs (Field Programmable Gate-Array) for example. An FPGA is a chip that you can program (and re-program) to connect many logic gates in all kinds of different ways to do complex logic things.

These ARM controllers, however, use SWD and this protocol only uses two wires, but doesn’t allow you to connect many chips in a daisy-chain. So, we only have to consider two wires, other than the reset line, for each chip, but we do have to put two separate connectors on the board.