To select the microcontroller for your application there are several routes to take. Most, if not all, of those routes become easier every time you walk them, as you build up an array of experience with different vendors and what they have to offer.

To shortly digress on the subject of brands:

I personally built up a preference for Atmel controllers over the period of 15 years, in those cases that their portfolio has something fitting. My preference comes mainly because of their custom of keeping the hardware and instruction-set easy to use, which they started with the AVR series. They offer(ed) one of the most flexible collections of peripherals in any of the markets they operate(d) in, in my personal experience. This is in no way to say that I haven’t used TI, Freescale, NXP or ST controllers in projects or that I would avoid specifying them if they best fit a project. Although, if I have to be honest, which I’m trying to be in this blog series – no matter what – I’d have to admit that I do avoid specifying Texas Instruments (TI) where I can. I have a long history of aggravation with TI’s own designs from incomplete or erroneous claims in datasheets and failure to resolve hard problems all the way to the Tiva TM4C12xx’s 130 page long Silicon Errata, even in Silicon Revision 7, deemed “mature”. In the past 20 years it has seemed to me TI is taking non-disclosure more serious than proper design and verification or levelling with their customers, large or small. In large contrast to National Semiconductor, which they bought quite some time ago. NS was a company that nearly always including an equivalent internal schematic on every datasheet and made sure specifications given were accurate and as complete as possible. In fact, large parts of my early (age 6 to 15) education came from Nat-Semi and Linear-Tech databooks and if you can find one, I’d suggest you leaf through it, doesn’t matter what vintage. My personal suggestion; if you buy TI, try to only buy parts designed by others, such as the LM series of components.

Back to selecting a microcontroller

Step one in finding the right “brain” or “brains” for your application usually is trying to estimate what kinds of thinking you will need and how quickly it needs to be able to do that. Over time you may find you get better intuition for guesstimating it, which is what I did, rather than formally calculating it. Had this been a privately held project it’s quite possible I would have selected an AVR for the co-processor, if one availed itself for my purpose. Since AVRs can be more power efficient, have lower power usage in sleep and come back from sleep more efficiently. But, since many more people understand ARM cores than do AT(X)MEGAs, this being a fully open-source project, I decided early on to lock myself into only using ARM cores from the Cortex Families.

So that’s the first elimination right there, that excludes the vast majority of options: I will only use ARM cores, because more people know them.

Next I had the option to buy one massive processor with one or two cores inside it to get all the pins and peripheral options I expected I would need:

* Flexible ADC
* One or two DAC channels just in case
* A large number of PWM outputs (really, here is where some chips in the XMEGA series shine: up to 48 channels of PWM with a source clock up to 48MHz and 16bit accuracy on each channel at a mere few dollars per chip.)
* Easy and quick configuration of functions on I/O pins.
* Battery Back-Up option

With that list in hand it seemed to me quite quickly I’d be most flexible if I would use two separate chips, rather than one massive one. Even if you buy a $15 ARM chip, usually the functionality of peripherals is restricted to a small group of pins for each type of interface, where on smaller chips you much more often see peripherals that can be routed “all over the place”. With one last criterium for this specific design I quite quickly ended up with low to midscale Atmels: In this contest, there’s a deadline for most fully functional I can get it and I know Atmels, and Atmel made a free development environment and support library supporting all the functions in even their latest chips. Something almost all other brands – even KEIL/ARM itself – are still struggling with. So, back to Atmel I went, but now they are made and sold under the name of Microchip.

Finding the Co-Processor

Because I have recently been working quite a lot with the Cortex-M0+ offerings in the ATSAM series, I settled on using one of those as co-processor, leaving me to choose from ATSAMD, ATSAMC and ATSAML. The SAMD series is the broad base on which the more specifically aimed L and C series were built, so the D is more “centrist” in its functionalities, even though it can do quite a few impressive tricks as well. The L series features several blocks that are more aimed at USB performance and direct LCD drive, so since it’s the co-processor this felt like wasted money (if it gets a display, it’ll be an expansion later). You pay for everything you buy in a controller and 25% of its silicon surface is USB and other stuff you won’t use, that means you’re getting 25% less bang for your buck. So I looked at the recently released C series and was happy to see another option opened up right there: These chips can be powered from 2.7V to 5.5V, which was an immediate check in the “pro” column. Then, looking further, it turned out to also have a Hardware Acceleration for division and square root, which will be a great help if I need to calculate RMS values of analogue signals. It also has a DAC, configurable logic block, more options in the SERCOM module and a higher speed ADC with the offset and gain compensation. Probably all because it has not wasted any room on having a USB interface. So, that’s it, Co-Processor chosen: ATSAMC21, because the C21 has more of the peripherals at a marginal cost increase over the C20. I chose the package with the most pins, since again it only costs a smidge more than the smaller ones and for the first version, of course, the one with the most memory, so I can play all I want without fear of running out. If it comes to production, the memory content of each processor would be re-evaluated.

Deciding on the Main Controller

For the Main Controller, again considering the design speed required, I decided I’d like to be able to use some floating points where necessary, at least in the alpha and beta firmware. So I would prefer having a core with a Floating Point Unit (FPU). I also didn’t mind if it was a bit quicker than the co-processor, since the idea of short bits flight is still buzzing around in my head and ARMs aren’t always the most efficient with the clock cycles. I wanted it to be somewhat new as well, so that I had a chance of using the same type of SERCOM and ADC interfaces for both controllers. This quickly brought me to the SAMG series. This controller is a bigger cousin to the D/C/L series of about the same vintage as the SAMD20-series. This means in broad terms all the peripherals come from the same cookbook, meaning I’ll be wasting less time developing for those. It also has a USB2.0 connection, which is more than fast enough for this use, with a built-in hardware Bootloader. It runs a Cortex-M4 core with FPU, running at up to 120MHz. So that works out pretty well.

Of course, I’m not convinced of a processor on just the cover page of a datasheet and while I had worked with SAMD09, SAMD20, SAMD21 and SAML21 before, the SAMG series was completely new to me, so I had a careful read of the first few chapters of the full datasheet. This lead me to believe the SAMG55 would be the ideal choice as a Main Controller, but I wanted to be certain, so first I looked at the SAM4S and SAM4N devices. While those series have some impressive chips, some with more pins, with more peripherals and even more impressive clocking options, it quickly occurred to me that to get the same amount of SERCOM functionality I’d have to spend twice as much. And none of those chips would have the 16kByte SRAM from which the core can run code directly. That last bit is especially interesting to me, because FLASH is a slow type of memory and Atmel uses wait-states to synchronise loading of commands from FLASH, rather than making a chip with expensive 120MHz FLASH. They do this mainly because many operations on an ARM need two or three clock-cycles of the core to complete anyway and with pre-fetching the wait-states would become nearly unnoticeable. But nearly is not fully. And the special chunk of RAM can operate right at the maximum clock speed, so there’s no delays, ever. This means that if my robot needs a control loop to be incredibly responsive, for example for a fine-mechanical tool on an extension, I can run that from the RAM and be sure it always executes at calculated processing speeds. While some of the other Cortex-M4’s have higher internal clocks available, I don’t expect that I will use that as quickly as the RAM-code option.

Because designing good hardware is just as much a scientific process as any other thing in research and development, I will never make the mistake of thinking this is the definite choice for all bots to come. Because I did a lot of thinking, searching and reading, which is also part of a scientific method, before making this choice, I do feel the chances are very good for this being the way the hardware will be for a good few versions. In engineering you need to do good research, maths, and think a while whether you’re not forgetting something, but once all your calculations have finished and all seems good you also need to just go for it and stop worrying, or you’ll end up stuck behind your desk for years before the first hardware comes.