Developing your own hardware

By admin
Published: 09/26/2007 - 13:37

Kristina Andersen , September 2007

Making your own technology can mean voiding the warranty of your iPod by "modifying to alter functionality or capability without the written permission of Apple" [8] , to straight up hacking and bending commercial products and on to building circuits from scratch. These techniques are all methods to graduate from user to owner in our everyday relationship with technology. If you cannot open it, you don't own it [9] and if you work with circuits as an artist and maker it becomes especially important to exercise ownership on the technological vehicles of the work.

B087 Lie-detector, Kemo-Electronic GmbH

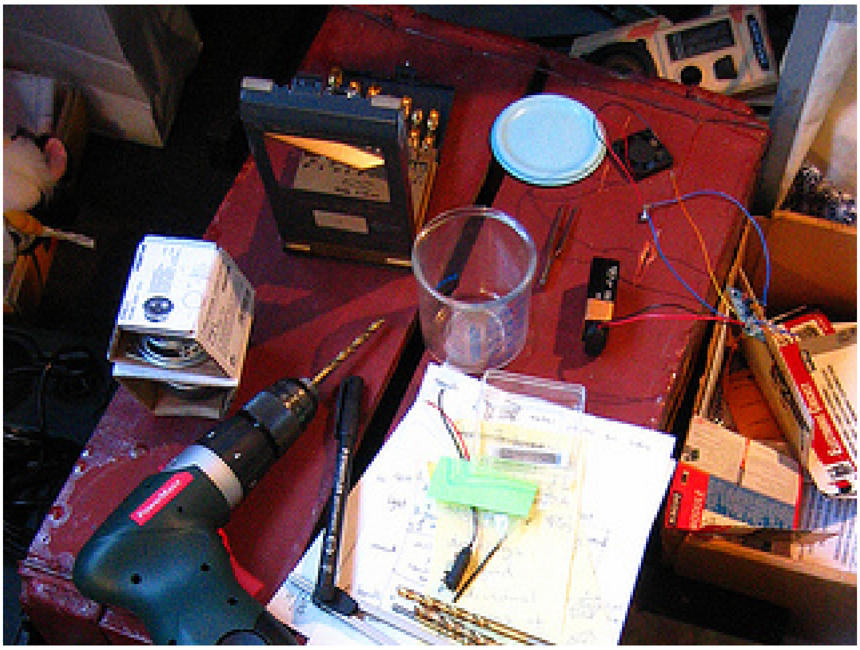
Since the 50's it has been possible to buy elaborate DIY electronic kits with farfetched uses and over-Xeroxed line drawings - made for father and son bonding sessions. When you first begin to solder and build circuits these are the kits you start out with. You order them from giant illegible catalogues and when they arrive they look as if they have been composed entirely out of reclaimed components from the soviet space programme. Which is why you keep ordering them: The beauty of the matte square green LED that sometimes comes with the KEMO lie detector kit [1] . Where else would you find that?

It is worth noting that almost nobody teaches their daughters to solder. The stupidity of the dog barking circuit kits, paired with a generally patriarchal electronics culture, has so far made electronics a largely non-female pursuit even as other areas like fashion and fabric crafts moves further towards the customised and the self-made. In fact the process of crafting fabrics is remarkably similar to the building of circuits. Like other forms of crafts, making your own technology can be seen as a sculptural process of allowing objects to be formed in your hands and as such it is an almost primal process of discovery and recognition. As an object slowly takes form on the table in front of us, we can begin to develop intuitions about its capabilities and behaviour. Maybe in an extrapolation of Hayes' 'naive physics' [6] we could say that it is through the actual making that we begin to comprehend the objects we are building.

Details of the artists' studio

Making and crafting can also be said to be a form of play as long as one keeps in mind that playing is not necessarily light-hearted or fun; but rather a way of approaching a particular design space and a set of rules and limitations [3] . We are making our home made technological objects to the best of our ability and the limitations we face are our own. Claiming this kind of independence is a way to retain some level of control over the work. This does not mean we do not need help and collaborators. It just means that when it fails, we stand a chance of asking better questions and questioning the advice we receive. In the crafters manifesto Ulla-Maaria Mutanen says that the things we make ourselves have magic powers and hidden meanings that other people can't see [11] , and this is true for the hand made circuit as well.

Things are changing in the world of making things. In the wake of the emergence of Interaction Design as a discipline, basic stamps and PIC boards are suddenly on every curriculum and there are enormous amounts of "make your own" tutorials on the web. Instructables, a website where users share step-by-step project instructions, has its origin at the MIT Media Lab as a platform for students to share projects [7] , Hack a Day publishes a new hack each day harvested from enthusiasts on the web [5] , popular websites like BoingBoing [2] and expensive paper publications like Make magazine from O'Reilly [10] have brought the hobbyist and part time makers out into some sort of mainstream.



Details of the artists' studio

For the artist and the maker DIY is an ongoing process of discovery and failure. You are trying to use your clumsy and limited abilities to create that which did not exist before: unusual behaviours, objects that exist in response to flimsy thoughts and desires. Along the way through the making you encounter the limitations of your hands and your patience, but you find something else as well: the accidental object, the technology that exist merely as a quirk, the successful misconstruction. These are all glimpses of the material speaking back to you, the maker. They are the gifts from working with your hands and making your own. Nick Collins says it in the 16th rule of hardware hacking: "If it sound good and doesn't smoke, don't worry if you don't understand it" [4] . This is perhaps why we do it. By using our crude and clumsy hands to make aspects of computational machinery we are re-inserting ourselves into a process that we are otherwise excluded from. We make technological objects in an attempt to glean understanding, to see if our naïve intuitions still functions in a world of current and solder.

Maybe we can say, we are making technology in order to understand it, and understanding technology in order to make our own.

Notes

[1] B087 Lie-detector, Kemo-Electronic GmbH, viewed 4 October 2007, http://www.kemo-electronic.de/en/bausaetze/b087/index.htm

[2] Boingboing, 2007, Happy Mutants LLC, viewed 4 October 2007, http://www.boingboing.net

[3] Caillois, R.1961, Man, Play, Games, University of Illinois Press. Illinois.

[4] Collins, N. 2006. `Laying of Hands in Handmade Electronic Music' in The Art of Hardware Hacking, Routledge, New York.

[5] Hack A Day, 2006, Hack A Day Beta, viewed 4 October 2007, http://www.hackaday.com

[6] Hayes, P. 1978, The naive physics manifesto Systems in the Micro-Electronic Age, Edinburgh University Press, Edinburgh.

[7] Instructables, 2007, Instructables, viewed 4 October 2007, http://www.instructables.com

[8] iPod and iSight Warranty, 2007, Apple's Limited Warranty document for iSight and iPod, Apple

Inc., viewed 4 October 2007, http://images.apple.com/legal/warranty/docs/ipodisight.pdf

[9] Jalopy, M. 2005. Owner's Manifesto, Make04, viewed 4 October 2007, http://makezine.com/04/ownyourown

[10] Make Magazine, 2007, O'Reilly Media, Inc. viewed 4 October 2007, http://www.makezine.com

[11] Mutanen, U. 2005, Crafter Manifesto, Make04, viewed 4 October 2007, http://www.makezine.com/04/manifesto

Images

[1] B087 Lie-detector, Kemo-Electronic GmbH, viewed 4 October 2007, http://www.kemo-electronic.de/en/bausaetze/b087/index.htm

[2] Studio details. Photographed by Suno

" >

span-->

- Hardware hacking: open hardware and stand alone objects

‹ Game art: theory, communities, resources up Hardware hacking: open hardware and stand alone objects ›

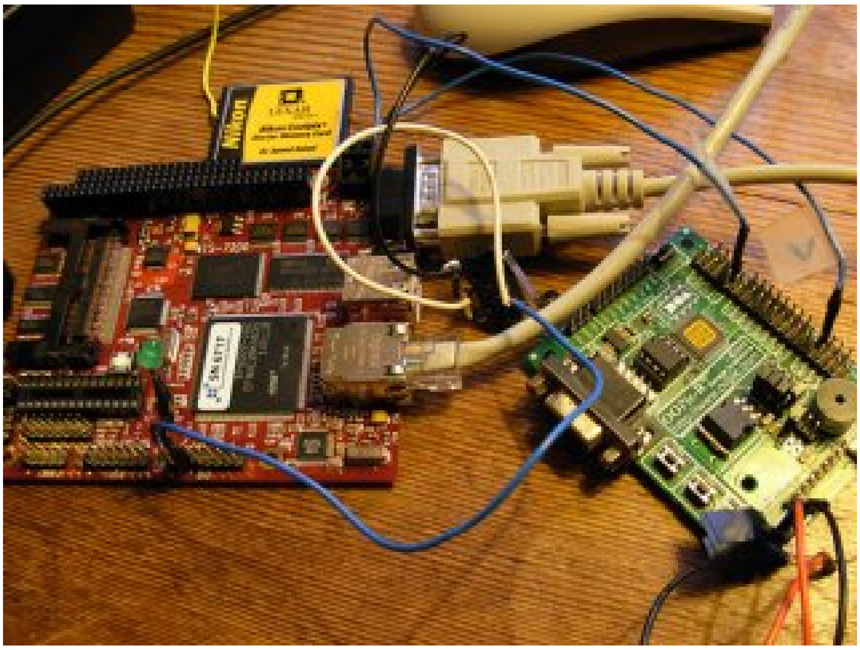
Hardware hacking: open hardware and stand alone objects

By marloes
Published: 09/18/2007 - 13:20

Tom Schouten , September 2007

That's a broad title. Building your own hardware can mean a lot of different things. To narrow the scope a bit, this article talks about embedded Single Board Computers (SBCs) and microcontrollers from birds eye view. An embedded SBC is anything that is complex enough to comfortably host a standard operating system, while a microcontroller is too small for that.

Physically an SBC consist of a single printed circuit board (PCB) with some RAM and Flash ROM memory chips, and usually a single highly integrated chip with CPU and peripheral devices. Often the smaller microcontrollers have RAM and Flash integrated together with CPU and peripherals on chip. For larger memory sizes, it is still (2007) more efficient to put memory in separate chips. To give you an idea, the minimal specifications of an SBC is something like a 32-bit CPU with about 8MB of RAM and about 4MB of permanent storage, usually in the form of flash memory.



TS-7200 ARM Single Board Computer

Anything simpler than an SBC we call a microcontroller. A microcontroller is typically a System on Chip (SoC). A SoC is a complete computer system in a single chip package, containing processor, RAM, permanent storage (Flash), and I/O peripherals. A microcontroller can run completely stand-alone, which makes them very interesting from a DIY perspective.

This article aims to give an overview of available tools without going into too much detail. The projects mentioned are well documented and most have and active community. Hopefully this will tickle your interest, so you can go on exploring for yourself.

In order to interact with the physical world, you need sensors and actuators. A sensor is a measurement device, and an actuator is a control device. Most embedded systems convert some kind of physical information (i.e. a button press, or a sudden acceleration) into a physical action (i.e. the switching of a light or the speed of a motor). What happens inbetween is handled by the computing system, and can be quite elaborate. Sensors and actuators are also called transducers: devices that convert one form of energy into another. A sensor is a low-energy transducer which converts any form of energy into electrical energy, while an actuator is a high-energy transducer which transforms electrical energy into any other form of energy. Getting these things to work is in the domain of analog electronics or popularly called physical computing.

It is assumed that you have some basic knowledge on the electronics and physical computing side, or know how to obtain this knowledge (see introduction). This article deals mostly with embedded software and tools: these days, doing hardware means also doing software. Some nostalgic and crazy hacks set aside, most problems are better, easier, cheaper and faster solved in software. Open up any electronic device; inside you'll find some kind of microprocessor.

A PIC 18F8720 microcontroller

Digital electronics systems can get quite complex, but it is not necessarily so that a such a highly complex system is difficult to develop for, or that a system of low complexity is easy. The main reason of course is that complexity can be hidden by modular design. If you can (re)use a lot of hardware and software designed (and made available!) by others, the perceived complexity of a system can be fairly low, even if it is extremely complex on the inside.

For example, a state of the art Single Board Computer (SBC) tends to be a very complex machine. However, due to the availability of a huge amount of reusable software tools, programming languages and code libraries, making it do something interesting and original does not have to be a complete nightmare. On the other hand, using a very simple and small microcontrollers can be quite difficult when you need to write some arcane low level code to squeeze out that last bit of performance. The middle road has a lot of wiggle room; for most projects that are not too sensitive to hardware cost, one can opt for a platform that makes the development more straightforward.

Also, let's assume it is not feasible to design your own embedded SBC circuit due to complexity and cost, and assume that such a system will have to be bought as a building block. On the other hand, we can assume it is feasible to design your own microcontroller circuits, and will briefly mention a collection free software tools that can be used to this end.

This article does not talk about sensor and actuator boards. Such a device is intended to be connected to a SBC or PC and used merely as an I/O extension, and cannot be programmed beyond simple configuration. If you're looking for something simple and want to avoid microcontroller programming, such an I/O board might be an ideal approach. A popular one seems to be the Muio [10] . It is a modular system and has extensions to hook up sensors to control Pure Data, MAX/MSP, SuperCollider and Processing running on a PC.

A note about vendors. The embedded electronics market is extremely diverse. For SBCs this is not such a big deal since the software can be made largely independent of platform. For smaller microcontrollers however, it can become a problem because there is usually not enough memory space to have a layer of software that abstracts away hardware differences. Details of the target bleed through to the software, making the software more low-level.

Instead of endorsing any specific vendor, the article points out how to look for vendors providing products that can be combined with free software tools, and indicates which vendors are most popular from the perspective of free software tools. Note that a lot of vendors provide their own tools, often free of charge. These tools are in general not free in the "libre" sense, and as such not the subject of this article. They are interesting to check out nonetheless. If you allow yourself to be locked to a single vendor, using the tools bundled with delivered hardware might be the most pragmatic solution.

Embedded SBC

We defined an embedded Single Board Computer (SBC) as a system that is powerful enough to comfortably run an operating system. The operating systems of interest here are those based on the Linux kernel. The Linux kernel is a widely used open source operating system kernel. An operating system kernel is the lowest level program in an operating system which handles the communication to the hardware through device drivers, and manages the communication between other programs and hardware, and programs themselves. As a result, programs that run on top of the kernel can talk to the hardware in a generic way, without having to know the particularities of the devices used. A kernel also allows programs on the same machine, or programs on different machines to communicate with each other. An example of this is the TCP/IP network protocol, which is a set of conventions used by machines to communicate through a packet switched computer network.

The Linux kernel is a fairly standard software platform that is largely independent of the underlying hardware platform. The most popular processor architectures that fall in this class are ARM9, MIPS and PowerSBC. Think wireless network router, network storage device, cell phone, PDA and handheld gaming consoles.

There is a tremendous amount of free software available to run on Linux. If you are in a position to use such a device, "building your own hardware" might be nothing more than finding the right software to use, compiling it for your embedded SBC and gluing together the parts. Most SBC vendors provide their own GNU Linux based tool chain. There are a lot of websites about embedded linux; it is a booming business. One that deals specificly with devices can be found here [9] .

Note that from a DIY or educational perspective it might even be more economical to gut a finished consumer product and put your own software on it! These products are mass produced, and thus cheap and ubiquitous. These properties usually cause the emergence of a community working on alternative software, enabling people to repurpose these locked down devices. The hard work of opening up a particular mass-produced device might already be done.

A good example of this is the OpenWrt [11] project. This is a complete operating system for embedded devices, originating on the Linksys WRT54G wireless router platform, but now available for a large amount of different hardware configurations, some of which are readily obtainable, general purpose embedded boards. The OpenWrt project is based on the Buildroot [3] suite and the ipkg binary package management system. Buildroot is a system that makes it easy generate a cross-compilation toolchain and root filesystem for your target Linux system using the uClibc C library.

```
[illegible]:~$ ssh root@fred
sh: /usr/X11R6/bin/xauth: not found

BusyBox v1.00 (2005.09.14-15:55+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

          W I R E L E S S   F R E E D O M
 WHITE RUSSIAN (RC3) ------------------------------
  * 2 oz Vodka   Mix the Vodka and Kahlua together
  * 1 oz Kahlua  over ice, then float the cream or
  * 1/2oz cream  milk on the top.
 ---------------------------------------------------
root@fred:~#
```

OpenWRT on a Linksys WRT54G

Now what does this all mean? A tool chain is a collection of programs that can transform software in source form into machine code your embedded SBC can understand. The source code language for Linux and the GNU system is mostly C. The component that performs the main part of this task is called a compiler, which in the case of Linux is the GNU GCC compiler collection [6] . For embedded development, the tool chain usually does not run on the embedded SBC, but on a different machine we call the host. In this case the compiler is called a cross-compiler.

A root file system is a directory tree that contains the collection of binary programs and data files that comprise the rest of the embedded operating system complementing the kernel. It also contains your application(s). Essential parts are the init process, which is the first program that runs after the Linux kernel is loaded, and a command shell, which can be used to interact with an embedded system once it is booted, by typing in commands at a console keyboard. Most embedded systems allow for the connection of an RS232 serial console to avoid the need of separate keyboard and monitor connections.

A shell is also used to write scripts: a sequence of commands executed without human intervention. The init process usually runs a collection of shell scripts that configure the hardware for its intended use. In embedded systems, an often used shell is Busybox [17] . This shell is optimized to consume little space; it includes stripped down versions of standard unix commands.

An embedded Linux system which contains the a binary package management system like ipkg can download and install programs and the libraries and programs they depend on while the system is live. This can be very convenient, giving the embedded system much of the feel of a Desktop GNU/Linux distribution, only smaller in size. The difference between OpenWrt and a raw image you create yourself with Buildroot, is that the binary package management system makes it possible to obtain OpenWrt in pre-compiled form. The system is split into a core system which you install as a whole, and a lot of optional packages managed with the ipkg system.

Systems like OpenWrt can make the learning curve for embedded development rather smooth. The system components are standard and most of them are very well documented, with a lot of community support available. Add to that the ability to start from a running system without having to do any real work except to figure out how to upload the first binary image, and you get an ideal framework for experimentation.

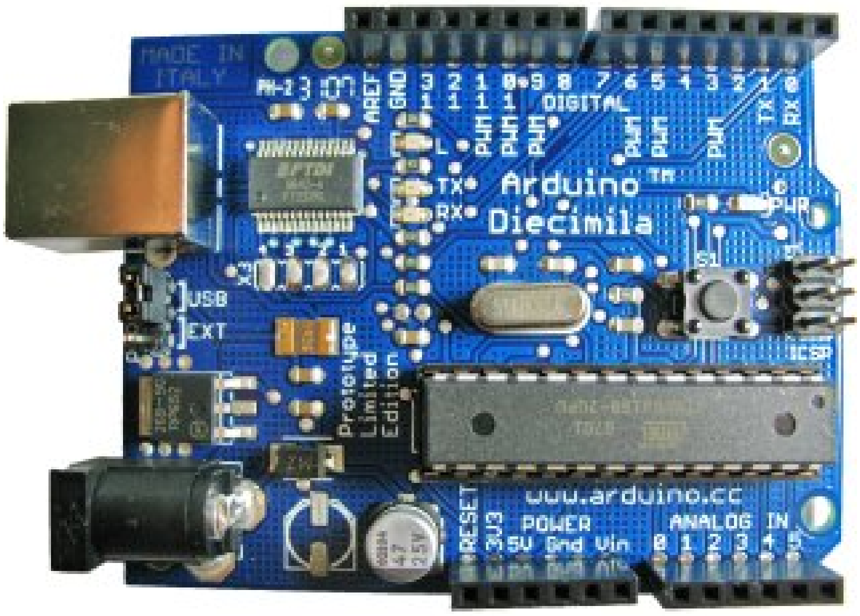
Microcontroller

Because of it's generality and still fairly high level of complexity, an embedded SBC solution might not be the best approach for simple sensor or actuator applications. While using an Embedded Linux SBC might make some complex tasks simpler, it can also make some simple tasks more complex. Some simple things are easier to do using a microcontroller.

It is mostly the absence of an external memory bus that makes it a lot simpler and cheaper. The embedded SBCs we talked about often consist of a highly integrated chip with external RAM and Flash memory, which creates the need for a lot of extra chip pins to be used as data and address busses for memory access. A SoC does not need a memory bus, so pins can be used for other things, or simply omitted. For example, there are microcontrollers with just 6 pins. A disadvantage of a SoC is that it is usually a lot smaller and offers less performance than a single board computer.

The main advantage of using microcontrollers is that they are simple from both an electronic circuit and a programming perspective. It is good to know that designing and building a circuit, and writing BASIC, C or assembly code is a task that can be done by a single individual in a matter of days or weeks, once the necessary skills are acquired.

If you do not want to design your own circuit, you can choose from a huge collection of simple controller boards. By far the most popular one seems to be the Arduino [15], which can be seen as a "sensor board on steroids" if used in its native environment. Arduino is targeted at artists who are looking for a low threshold entry in the world of physical computing, without giving up too much device performance. The Arduino language is based on C/C++, and is well documented. Underlying the Arduino project is the Atmel AVR 8-bit microcontroller, and the GNU GCC compiler. The Arduino design is open hardware, which means you can build the board yourself if you want to. The tools for Aruino are mostly free software, and run on OSX, Linux and Windows.

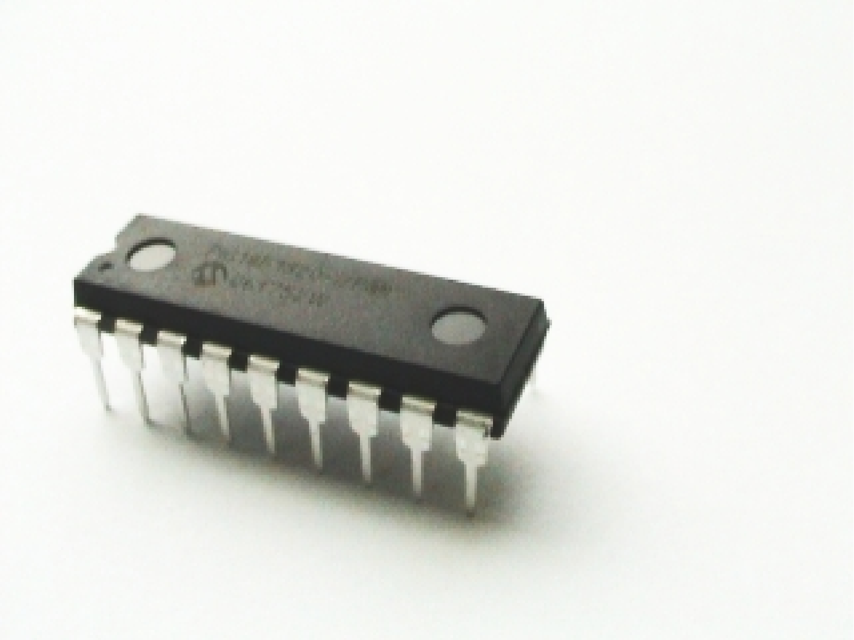


The arduino microcontroller board

Another off-the-shelf solution is the very popular Basic Stamp [22] . A Basic Stamp microcontroller is a single-board computer that has the Parallax PBASIC language interpreter in its permanent memory. Basic stamp software is provided by Parallax for the Windows, MAC and Linux platforms. Due to its popularity a lot of free embedded PBASIC software to run on the Basic Stamp is available. There is even an open source tool that lets you develop code for the Basic Stamp on the Linux platform [2] . Using the Basic Stamp is fairly straightforward. In addition a lot of community support is available, and like the Arduino, there are lots of places where you participate in a workshop for a hands-on introduction. The downsides of this platform are that it is fairly expensive and that it has limited performance compared to directly programmed chips.

The rest of the article will deal with the available open source tools to build your own custom hardware based on raw microcontroller chips. The first problem you encounter after deciding to use a microcontroller is to choose a chip and possibly board architecture. In the embedded SBC world this choice is less important due to the availability of the standard Linux platform which runs on almost anything. In the microcontroller world, there is an equally huge amount of different companies and architectures, but platform choices are more important due to the absence of a standard operating system layer.

For DIY projects, most people seem to go for the Atmel AVR [16] , or the Microchip PIC [21] . Which one is best? Unfortunately, the world is not 1-dimensional, and this includes the world of microcontrollers. One thing which seems to be a reality is that if you're not primarily cost-driven, you'll make the platform choice only once, and probably because of some arbitrary reason, insignificant in the long term. It takes some "getting to know", and can be hard to say goodbye to an old friend. Also note that per chip vendor, there is usually a very large set of architecture variants.



Another PIC 18F microcontroller

To be fair I have to switch to first person narrative here. Acknowledging my own bias, I'll tell you up front I chose the Microchip PIC18 architecture. What is more important than my choice, is the way I got to it. One often hears "just pick one", and there might be a lot of truth in that. However, I will do my best to give some arguments which might lead to an informed choice, because there are differences that can get important later on.

- Can you get direct support? Pick a platform that is used by an accessible expert in your direct surrounding.

- Programming languages, Tools and Libraries. This might be the most difficult one to know up front, but fortunately the easiest to change. Pick a platform with a lot of choice and diversity in this domain.

- Is there a large community surrounding the platform and its tools and libraries? Is there a forum where you can post questions? Do people seem helpful? Can you find indications of people getting stuck without anybody caring?

- Cost of hardware, including availability of free chip samples.

Free chip samples might seem an odd gift from the gods at first, but realize this is a form of vendor lock-in. Microchip has a very liberal sample policy. A good reason to choose for the Atmel AVR microcontroller is the existence of the Arduino board; it seems to have taken the world by storm. It provides an easier entry point to the world of microcontrollers, and uses a chip which is popular on its own. Once you are familiar with using the pre-fabbed board, the road to making your own microcontroller circuits becomes less steep. For more information, google for "pic vs avr". The first link that turns up for me is ladyada's comparison page [20] .

What do you need beside the actual hardware? You'll need to pick a programming language. Which language to choose is a topic of great debate, not only in the embedded software world. Some things you need to know about languages are that they tend force you think in a certain way. And in general, there is no silver bullet; the more power a language gives, the more time it usually takes to get to know it well, but the more interesting things you can do. Keep in mind that if you get stuck or your programs get too complicated, switching language might be an option. A language can be a dead end street: once you've exhausted its capabilities, it might pay off to back out and take a different road.

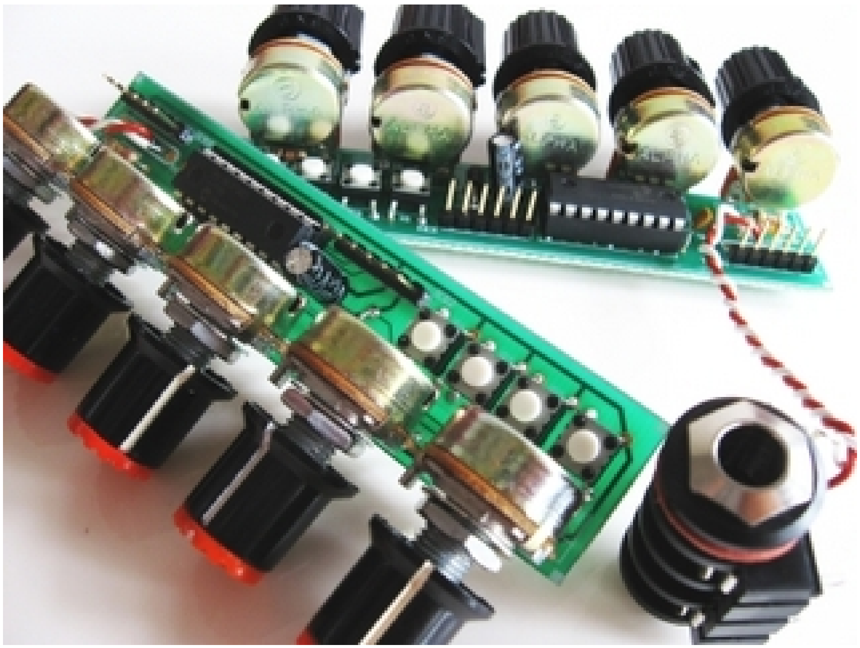
Traditionally, microcontrollers are programmed in assembly language. The reason for this is that they are often very resource constrained; the smallest versions have only a couple of bytes of ram, and putting the chip to good use can require some tricks specific to the platform. The advantage of using assembly language is that you are not limited in what you can do, but this is also a disadvantage since you have to take care of a lot of details yourself.

If you ever learn an assembly language just out of interest, a microcontroller might be the right choice due to its simplicity. Both AVR and PIC have very simple instruction sets, both being Reduced Instruction Set Computers (RISCs), although the PIC instruction set is a bit older and quirkier. Both chips have open source assembler tools [1] [7] . For PIC there is a very nice integrated development environment called PikLab [12] , similar to Microchip's MPLAB, which integrates with several compiler and assembler toolchains (like gputils, sdcc, c18) and with the GPSim simulator. It supports the most common programmers (serial, parallel, ICD2, Pickit2, PicStart+), the ICD2 debugger, and several bootloaders (Tiny, Pickit2, and Picdem).

Following the tradition, the next popular step up from assembly language is the C programming language. C is a bit more highlevel than assembly, in that it largely independent of the processor architecture. Remember that C is the language in which GNU/Linux is written. C is sometimes called portable assembler. The GNU/GCC compiler is available for AVR, and is used as the default compiler for the Arduino project. Also available is SDCC [14] , which supports Intel 8051, Maxim 80DS390, Zilog Z80 and the Motorola 68HC08 based microcontrollers. Work is in progress on supporting the Microchip PIC16 and PIC18 series.

In the DIY world, the BASIC language has always been fairly popular. Explicitly intended as a beginner language, it is a bit more limited and less expressive than C or assembly, but perceived as easier to learn and use. Due to the limitations and specific mindset it imposes, it is somewhat frowned upon in the professional world. Depending on your needs it might be the better choice nonetheless. There are several commercial solutions available. For PIC there is an open source compiler called Great Cow BASIC [5] .

Acting on the idea that popularity is not always a good indicator, I provide some pointers to more obscure language choices. For PIC there is a language called Jal [8] , loosely based on Pascal, which seems to have quite some followers. Then there are a couple of Forth dialects, the author's personal favorite. For PIC there is PicForth [23] , FlashForth [4] and (shameless plug) still in development, Purrr [13] .


The CATkit controller board for the Purrr language

Then finally, once you have selected a tool chain and a board or chip, you need a device called a programmer to dump the code generated by your tool chain onto the microcontroller's Flash memory. There are several ways of doing this. Both PIC and AVR provide their own in-circuit programmers. These devices can be used to program a chip while part of (soldered to) a circuit. In addition to this, there are a huge amount of DIY circuits you can build. Pick one you know somebody else uses!

To conclude the microcontroller section, I quickly mention board design. Commercial CAD packages are expensive for the simple reason that they are fairly complicated and highly specialized programs. In DIY circles, the most popular tool seems to be Eagle [18] . This program is free for noncommercial use and small board designs. For anything else you need to buy a licence. An alternative is GEDA [19] , an open source collection of design tools. Its main components are gschem, a schematics capture program, and PCB, a printed circuit board layout program. This tool set allows you to create CAD file, often called Gerber or RS-274X files. These files contain descriptions of the different board layers like copper, silk, solder resist and drill holes. To have a board built, you simply send the Gerber files to a PCB production or prototype house after you make sure your board respects their design rules.

Conclusion

Building your own hardware using open software and hardware tools can be done with a moderate amount of effort. By dividing the world into embedded SBCs and microcontrollers, I have showed how one might go about collecting the necessary tools. Especially for the embedded SBC approach there is a lot of free software available for reuse, and the GNU GCC tool chain is fairly standard. The microcontroller world is a bit more ad-hoc, not being as independent from hardware platform.

The trick in many cases is to take the best of both worlds; use a central general purpose embedded SBC connected to a couple of microcontroller based special purpose circuits, which you either build on perf board, or for which you design a circuit and have it fabricated.

Hope this helps. Good luck building!

Notes

[1] AVR assembler for Linux and others.

[2] Basic Stamp tools for Linux.

[3] Buildroot, a tool for building embedded Linux systems.

[4] FlashForth.

[5] Great Cow BASIC.

[6] The GNU Compiler Collection.

[7] GNU PIC Utilities.

[8] Jal, (not?) Just Another Language.

[9] Linux Devices.

[10] Muio, a modular system for sensing and controlling the Real World.

[11] OpenWrt a Linux distribution for embedded devices .

[12] PikLab.

[13] Purrr.

[14] Small Device C Compiler.

[15] Arduino, an open source electronics prototyping platform.

[16] Atmel AVR.

[17] Busybox.

[18] Eagle, a schematic capture and printed circuit board design package.

[19] Geda, a GPL suite of Electronic Design Automation tools.

[20] PIC vs. AVR.

[21] Microchip PIC.

[22] Basic Stamp by Parallax, a single-board computer for the PBASIC language.

[23] PicForth

Images

[1] TS-7200 ARM Single Board Computer

[2] A PIC 18F8720 microcontroller

[3] OpenWRT on a Linksys WRT54G. Photo by Tom Schouten

[4] The arduino microcontroller board

[5] Another PIC 18F microcontroller. Photo by Marloes de Valk

[6] The CATkit controller board for the Purrr language. Photo by Aymeric Mansoux

" >

span-->

‹ Developing your own hardware up