THE JEE BOOK
Adventures in physical computing
Jean-Claude Wippler
The Jee Book
Adventures in Physical Computing

Jean-Claude Wippler

This book is for sale at http://leanpub.com/jeebook

This version was published on 2015-07-02

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2013 - 2015 Jean-Claude Wippler
To Liesbeth and Myra, with love and gratitude.
# Contents

Preface ................................................................. i
  About this book ..................................................... i
  Is it for me? .......................................................... i
  Organisation ........................................................ ii
  History ............................................................... iii
  A word about pills ................................................ iv

I  Explorations ......................................................... 1

1. Physical Computing ................................................. 2
   The world of Arduinos ............................................. 3
   Raspberry Pi’s and Beagle Bones ................................ 4

2. Getting Started ..................................................... 7
   Brutally Bare Basics .............................................. 7
   Welcome to ’32 ..................................................... 8
   Just an ARM and a LED .......................................... 10
   Long live the breadboard ....................................... 12
   Minimal code, the basics ....................................... 15
   The upload conundrum .......................................... 17
   Modifying the USB BUB .......................................... 20
   Uploading from Mac OSX ........................................ 22
   Uploading from Linux ........................................... 25
   Uploading from Windows ........................................ 29
   Life after uploads ................................................ 37
   Setting up the Virtual Machine ................................. 38
   Setting up a toolchain .......................................... 46
   More tools for your tool belt ................................... 49
   The need for speed ................................................. 53
   Relax and live longer ............................................ 55
Preface

About this book

This book is in its very early stages. The chapters I’m working on will be available separately as sample book and updated more often, to let you easily follow their progress.

I invite you to comment on the book in the discussion area¹ on the web. Right now, every bit of feedback will allow me to understand how to best meet your expectations, to see how the book is holding out in the real world, and to help weed out all those mistakes and omissions I so relentlessly seem to be adding in.

The Jee Book is produced and made available by Leanpub². This arrangement allows you to give a donation, if you so choose. To show my gratitude to Wikipedia, I will in turn donate 10% of the proceeds of this book to them. Never before has so much information and knowledge been made freely available to anyone with access to internet. They deserve all the support they can get.

This book is free. The software and hardware designs I create and describe here are released as Open Source³. Because open source leads to an open mind. You can use this material in any way you please, just be gentle and don’t hurt anyone.

Is it for me?

This is a hands-on book about technology, and more specifically Physical Computing - the intriguing combination of computer hardware and software with connections to the real world via sensors, actuators, and direct user interfaces.

This book is for children aged 8 to 88. Seriously, if you can read English (hey, look, you passed!), if you are interested in exploring technology, and if you want to understand why and how things work the way they do, then these pages should interest you. Technology is about man-made inventions, and this process as well as the background of all things invented so far offers a fascinating insight in the world of engineering and creative thought.

I started tinkering at age 8 when my French-Polish grandfather gave me a chemistry set, making all sorts of smokey, smelly, and other nasty stuff. Those experiments gave me enough insight that, later on, high-school chemistry was not only fun and easy - it was even intuitively logical.

¹http://jelabs.net/projects/cafe/boards/1
²http://leanpub.com/jeeknow
³https://en.wikipedia.org/wiki/Open_source
The same happened again when I started messing with electronics and seeing components go up in the strangest types of smoke (yep, smelly again). This went from transistors to microcomputer chips (couldn’t get my hand on an Intel 8080 as a kid here in the Netherlands, but I sure tried!).

Then people began fooling around with microcomputers, and I spent my time chasing magazine issues of Dr. Dobb’s, Byte, Practical Electronics, and Elektuur (the Dutch version of Elektor at the time). That pursuit has lasted me a lifetime and continues to this day (PDF datasheets? yummy!). As a “side effect”, it got me into university, and ultimately a maths and computer science degree. Which goes to show that all kids can get an education if they are lucky to have enough options, and if only they get hooked into the world of technology early enough in life, when exploration and invention are still called “playing”.

This book is for the playful child in you, if you like technology and are curious about software, hardware, or electronics. There are no barriers to entry, you do not need any certification or degree, and you do not need much, other than internet access for further study and obtaining some basic tools, components, and kits, depending on which area(s) you would like to explore.

One caveat: for people with a certain mindset⁴, the content that follows may lead to addiction. You have been warned…

**Organisation**

The chapters in this book are grouped into four parts:

**Part I - Explorations** is where it starts. By asking questions, we can investigate the field and look for answers. Some explorations will be practical, others will come as exercises for the mind. The deliverables are: new insight, increased skill, and/or more experience.

**Part II - Projects** is a diverse collection of hands-on articles, in a (hopefully) logical order. They are presented as step-by-step how-to guides, for you to replicate and build upon. In this part, the goal is to end up with concrete and working installations.

**Part III - Toolbox** is about materials, tools, components, supplies, equipment, kits, and instruments. In other words: all the “stuff” you need to buy or build for parts 1 & 2.

**Part IV - Concepts** goes into some of the science, theory, and knowledge behind all the explorations and projects in the other parts of the book.

Isn’t the order of these four parts completely backwards? Shouldn’t concepts be presented first, then tools introduced, then some real projects built with them, to finish with explorations into further what-if type of questions?

No - not when it’s about discovery. This book wants to take you along on an adventurous journey, a path which starts with an open mind and questions. From there, implicitly, we’ll go into trying

---

⁴https://slackprop.wordpress.com/2013/06/03/on-geek-versus-nerd/
things out, obtaining the stuff we need to do so, and learning the facts and theories to understand
the issues and solve the problems we run into.
You have to wonder why so many books start from the other end!

The order of these parts reflects my desire to make this first and foremost a hands-on guide, tempting
you on every page to ask and try out things for yourself. Which is why the explorations, with all
their questions, come first. It will cover topics such as: what physical computing is about, what stuff
you will need to get started, and why Ohm’s law should - will! - become second nature as you start
experimenting. Many of these questions will automatically lead you to the other three sections, in
the most natural way possible: when it’s relevant.

Adventures are about going places where you may never have gone before, and the best way to do
so is to ask your own questions and explore for yourself. Although you could easily slip into a comfy
chair and read through the text, which I hope will be a pleasant and interesting experience, there
are limits to where that can lead you. To go beyond the answers and experiences these pages can
give you, you will at some point need to get out of that chair and get physical.

There are many ways to use this book, which comes in several electronics formats: web, PDF,
ePub (for iPad, iPhone, and other ebook readers), and Mobi (for Kindle). There is no “right” way,
even though the material has been organised to allow reading it from the start onwards (intriguing
concept, eh?). Then again, if you’re like me, you may well single out some topics which pique your
interest and jump around like a madman to find and explore links as you come across them!

The table of contents is there to help you find your way and there are lots of cross-links inside the
book to refer to other relevant sections, as well as links to additional material on the web. Many web
links point to Wikipedia\(^5\) (that was such a link!) - a phenomenal source of information, even for deep
technical stuff. The cross-links in this book are under my control, so you should expect them to be
accurate. The web is a different beast, alas - links may break when sites drop out or get reorganised,
but I will do my best to fix broken URL’s whenever possible.

History

This book is a sequel to a project I started on the web, called the JeeLabs Daily Weblog\(^6\). This is a
collection of “posts” describing my early adventures in Physical Computing, based on various little
projects in and around the house to monitor and reduce our average energy consumption level.
Over the period of 2008 to 2013, it grew into a “brain dump style” collection of 1,400 articles with
the “JeeNode” as main building block (a wireless and ultra low-power variant of the Arduino open
source electronics prototyping platform).

\(^5\)https://en.wikipedia.org/wiki/Main_Page
\(^6\)http://jeelabs.org/
As announced⁷ at the end of 2013, I wanted to get away from the daily *post-something* rhythm and work towards a more approachable resource, where information could be rearranged over time to stay more logically organised, and with the ability to keep everything more up to date and more consistent. Hence this book.

The prefix “Jee” stands for JC’s (that’s me) Environmental Electronics. It doesn’t cover the whole field nowadays - feel free to reinterpret it as “JC’s Educational Experiments”, “JC’s Exploratory Engineering”, or even “JC’s Eccentric Elaborations”, if you prefer.

**A word about pills**

One central goal of this book is to illustrate that - *just as Alice⁸* in Lewis Caroll’s delightful story - nearly anyone interested has a choice: it’s a matter of going for the red or for the blue pill.

The *red pill* will help you become fluent in technology as a *maker* and maybe even as an engineer one day, whereas the *blue pill* will lead you to become a consumer of technology and to be left forever at its mercy. The choice - though not nearly as clear-cut - is yours to make.

All around the world, there is a bright future ahead for those willing to take control of the technology which is becoming so pervasive in our daily lives. The internet is making knowledge available to anyone, and recent hardware trends are now making the corresponding goods equally easy to obtain, at a very reasonable cost.

It’s a cliché, but this couldn’t come any closer from my heart: I hope you will have as much fun reading this book, as I am having researching and writing it (which remains an ongoing⁹ process).

– Jean-Claude Wippler

Dec 2013, Houten, The Netherlands

---

⁷http://jeelabs.org/2013/10/06/wrapping-up/
⁸https://en.wikipedia.org/wiki/Alice%27s_Adventures_in_Wonderland
⁹https://leanpub.com/lean
I Explorations
1. Physical Computing

Wikipedia defines physical computing as follows:

 [...] building interactive physical systems by the use of software and hardware that can sense and respond to the analog world.

We all know what computing is, to a certain degree, but *physical* is an essential ingredient in this context: while normal computers just connect to the internet via a network and to us via a display, keyboard, and mouse, the computers in this book are different. Much closer to a mobile phone in fact, than a desktop computer or even a laptop.

What makes the technology of physical computing special, is that the components are low-cost, easy to tinker with, and - as you will find out - easy to understand. You wouldn’t want to mess with your phone, but this is all about *intentionally* messing with the technology of physical computing!

As mentioned in the preface, for me physical computing represents an intriguing combination of computer hardware and software with connections to the real world via sensors, actuators, and direct user interfaces. Despite this chap’s prediction, embedded computers are now everywhere:

“I think there is a world market for maybe five computers.”
– Thomas Watson, chairman of IBM, 1943.

Digital computers have been around for well over half a century, and that includes devices which you could take apart and mess with. The Altair¹ computer was created in 1975, *by* and *for* hobbyists, and it lead the way to a new breed of tinkerers who called themselves “computer hobbyists” at a time when almost no one knew what a computer was, let alone *used* one:

“But what… is it good for?”
– Engineer at IBM, 1968, commenting on the microchip.

“There is no reason anyone would want a computer in their home.”
– Ken Olson, president, chairman, and founder of DEC, 1977.

(These delightful quotes come from the RinkWorks² web site, check it out for many more...)

¹https://en.wikipedia.org/wiki/Altair_8800
²http://rinkworks.com/said/predictions.shtml
We all know how that went. Keep in mind that the Personal Computer was born\(^3\) in 1981.

Unfortunately, with the industrialisation of the PC came the complexity which made it hard for hobbyists to mess with these machines (other than swapping expansion cards in slots, \textit{yawn}), and a lot of hardware tinkering potential was stifled as a result.

Another trend in the 1980’s came from what could now be described as the predecessors of game consoles: the TRS-80, ZX81, Commodore 64, and Amiga. Although fascinating for kids of all ages and highly educational, these too were not meant to be taken apart or hacked in any way. Computing? Yes! Physical? Nope.

Let’s fast-forward some three decades, to this century.

The world of Arduinos

In 2005, a student project in Italy created a little board and called it the “Arduino”. Self-contained, low-cost (about $30), and combined with a multi-platform software development environment

\(^3\)https://en.wikipedia.org/wiki/Influence_of_the_IBM_PC_on_the_personal_computer_market
which made experimentation easy. Above all, the hardware design and the software design were open source and... free!

This was at a time when some embedded boards were available, such as the Basic Stamp⁴, but with limited options, several times more expensive, with compiler vendors trying to make money on the compiler “toolchain” needed to build software for them.

The Arduino hardware is essentially an ATmega328 microcontroller, a USB interface, and probably most important of all by now: a set of conventions w.r.t. shape, pinouts, and the software for it.

After over half a million sold⁵, it’s safe to conclude that the Arduino has taken the world by storm.

Several years and revisions later, the Arduino universe⁶ has grown to an active forum, and the collaborative development of new IDE releases on GitHub. The latest trend appears to be to combine the original hardware with a small embedded Linux system, although it is not clear to me what the long term direction of all this is going to be.

Interesting times, for sure, and hats off to the original Arduino designers for creating a vibrant ecosystem around the concept of physical computing.

The Arduino is what got me started on the JeeLabs Daily Weblog⁷, back in 2008. Fond memories!

**Raspberry Pi’s and Beagle Bones**

The Raspberry Pi⁸ (“RPi”) and the BeagleBone Black⁹ (“BBB”) are recent developments in the wake of the Arduino to bring low-cost computing to the masses.

---

⁴https://en.wikipedia.org/wiki/BASIC_Stamp
⁵http://www.quora.com/Arduino/How-many-Arduinos-are-sold-each-year
⁶http://arduino.cc
⁷http://jeelabs.org
⁹http://beagleboard.org/Products/BeagleBone%20Black
With both products, the foundation is based on an embedded Linux processor, with considerably more memory and speed than an ATmega microcontroller. The reason these boards have become so affordable is no doubt largely due to the mass production of set-top boxes, routers, and mobile devices.

Working with these boards comes much closer to the way we work with laptops and desktop machines than with Arduinos: a mature and very extensive multi-tasking operating system, fully networked, with the ability to easily run every type of software we like, from compilers, to servers, all the way to HD video.
The Raspberry Pi only has limited I/O connection capability, and seems more geared to the educational experience of a full-fledged Linux system on such a small and low-cost computer, whereas the BeagleBone Black offers expandability through add-on boards which are called “capes”.

It’s not really meaningful to compare Arduinos with RPi’s and BBB’s, since their technical capabilities are so different. One should not consider them as competitors, but as mutually beneficial: together they offer an immense range of capabilities for exploring technology at every level, and for creating projects tied to the physical world. There are many examples of people implementing robots, home automation systems, and other projects with these building blocks.

The revolution comes from the fact that this has all become so affordable: for less than the price of even the simplest game console, it is now possible to build the basis of a fairly advanced home automation system with a base station as perfectly capable real-time web server, and wireless sensor nodes to collect environmental information around the house. The technologies exposed with such a set up have come in reach of the interested teenager, while also covering quite a substantial part of university-level Computer Science courses!
2. Getting Started

Brutally Bare Basics

There is a lot to be said for the Arduino and its Integrated Development Environment - which is why the original JeeNode and numerous other boards are designed to be highly compatible with it in terms of software. *Batteries included, hitting the ground running, a vibrant community* - it all helps to get people up to speed in no time.

But there’s also a lot going on under the hood which is at least as fascinating. When you peel away the layers from the hardware and software side, you end up with the key ingredient of physical computing: the **microcontroller** (often abbreviated as “µC”) which is driving today’s revolution behind all those smart products, from the coffee machine to the internet-enabled home thermostat.

This chapter aims to start with the simplest setup which could possibly work, and build up from there. By understanding what goes on inside (up to a point, clearly) we can gain a better understanding and appreciation of what all those extra layers of comfort and abstraction give us. And with open source, the whole point is not just that software is free and hardware designs can be replicated at very low cost, but that it’s all *intro)spectable*. Given the interest and the time, you can figure out anything - there are no barriers, the knowledge is yours.

There is complexity, of course, and there are many trade-offs in the creation of any project or product, whether hardware or software, but open source is fundamentally different from commercial “closed source”, where use and consumption are usually the only options. With open source physical computing, you can discover, explore, re-purpose, combine, invent, and build new solutions as much as you like. Whether for fun or for profit is up to you.

Here is the definition of the term “physical computing” from Wikipedia¹:

> [...] a creative framework for understanding human beings’ relationship to the digital world. In practical use, the term most often describes handmade art, design or DIY hobby projects that use sensors and microcontrollers to translate analog input to a software system, and/or control electro-mechanical devices such as motors, servos, lighting or other hardware.

The “digital world” in this context really means “computing”, so we’re going to have to dive into a lot of inter-related topics: 1) the µC itself, which is an integrated circuit, 2) how to connect it to a power source, 3) the way to make it do something intelligent, i.e. its software, 4) how to get

---

¹[https://en.wikipedia.org/wiki/Physical_computing](https://en.wikipedia.org/wiki/Physical_computing)
that software into the µC, and 5) how to connect the µC to the outside world of sensors and other 
“electro-mechanical devices”.

And that’s just a first iteration, really. As you’ll see, physical computing is an endless world, bound 
only by (your!) imagination, inspiration, and yes... also a smudgeon of perseverance.

Again, courtesy of Wikipedia², a quick overview of what physical computing is about:

![Bi-directional system interaction with the real world](https://commons.wikimedia.org/wiki/File:Physical_computing.svg)

Let’s start from scratch. No sketches. No IDE’s. No kits. Just a µC plus some components.

**Welcome to ‘32**

There are thousands of different microcontrollers (µC’s) to choose from. In this context, we’re going 
to use one which is easy to handle yet powerful enough to take us deeply into the modern world of 
embedded µC chips.

Many choices ahead. First, the architecture: 8-bit, 16-bit, or 32-bit? The lowest-cost option used to be 
8-bit: all the machine instructions operate on 8 bits at a time (although the program counter quickly 
outgrew that, and went to 16 or more bits). What this means is that the µC can deal directly with 
numbers in the range 0..255 (uint8_t) or -128..+127 (int8_t). Not a lot you can do with that really, 
but fortunately every C compiler can easily sidestep that by adding a layer of abstraction, so that 
larger values are handled by generating multiple instructions at once for each requested operation.

In an 8-bit µC, such as the Arduino’s ATmega328, the expression 1234 + 5678 simply takes a bit 
more work for the compiler and the µC. Even more so with 1234 * 5678, which produces a 32-bit 
result (well, 23 bits to be exact, i.e. 7006652). When programming in C or C++, we can effectively 
ignore the issues and just declare our variables to have the proper 8-, 16-, or 32-bit width. Except for 
one aspect that is: all those extra instruction do end up getting stored in memory, so the program that 
ends up running in that 8-bit µC will be a bit larger than we’d expect by just counting the operations 
in our original program.

With 16-bits µC’s, such as the popular MSP430, life is a bit simpler. There’s quite a bit you can do 
with numbers in the range -32,768..+32,767 (int16_t), and a program counter which can address at 
least 64 KB of flash memory, so it’s a pretty sweet spot.

---

²https://commons.wikimedia.org/wiki/File:Physical_computing.svg
But why stop there. With 32-bits, we can count microseconds in an hour without overflowing the default numeric variable, or even seconds in a lifetime. Perhaps more importantly these days, as the memory sizes of many low-cost µC’s are increasing well beyond 64 KB, is the fact that a 32-bit architecture has some very practical benefits:

- all memory can easily be addressed with a 32-bit pointer, both flash and RAM
- we rarely need to think about overflow when the basic arithmetic size is 32 bits
- all basic integer, floating point, and pointer operations use the same size values

Unlike an 8-bit ATmega328, a 32-bit µC does not need to play tricks to access flash memory or RAM - it’s all one pointer away. As a consequence, constants and static strings can be kept in flash memory, and don’t need to be copied to RAM. On very low-end µC’s with very limited RAM space, this can be an important benefit.

Note that being 32-bit on the arithmetic and pointer side of things does not mean that every variable needs to be stored in 32 bits. Even with 32-bit architectures, the standard is to keep memory byte-addressable, i.e. each 8-bit memory unit can be accessed individually. An array of 8-bit ints is as compact on a 32-bit µC as it is on an 8-bit one, the 32-bit version simply “widens” the 8-bit value to fill the 32-bit register when fetched, and vice versa.

Even on the code size, 32-bit architectures are not necessarily more wasteful than 8-bit designs, for a couple of reasons. For one, accesses to memory can be “indexed”, i.e. offset from another register value by a small value. Such instructions do not waste 32-bits to store small offsets. Another factor is that every arithmetic and pointer operation will act on the full 32 bits, instead of requiring multi-precision (and hence multi-instruction) trickery to make 8- and 16-bit operations simulate a 32-bit variant.

In fact, it’s not uncommon for 32-bit chips to generate less machine code for the same application as their 8- and 16-bit counterparts. One reason is perhaps that C and C++ compilers tend to focus on 16- and 32-bit operations (because 8-bit ints are often too limiting). A hand-assembled program for an 8-bit chip can probably still be made smaller than a hand-assembled one for ARM, but the reality is that assembly code is very rarely written these days. It’s simply too labour intensive, compared to C and C++.

So much for the architecture. Let’s see what options there are with ARM, the 32-bit de-facto standard today. Interestingly, ARM is not a single manufacturer of chips, but a supplier of designs licensed to a large number of different chip manufacturer. Unlike the AVR, PIC, or MSP series, choosing ARM does not limit one’s options to a single supplier. The difference is that each chip “vendor” makes different feature trade-offs and adds their own set of hardware peripherals onto that common µC architecture.

This is a big deal. It means we can pick ARM as architecture, and get all our software tools set up in just the way we like it, yet still shop around to pick a particular chip to work with. If that chip later turns out to stagnate in development, or is missing features we’d like to start using, we can switch to a different vendor and retain the investment in time and effort spent in setting up our tools. With
luck we can keep most of our code - or at least port it without having to start from scratch, since the µC core architecture is still the same.

In short: let’s commit to ARM, and let’s pick one specific chip for now, just to get started. Here is a very nice one, since it’s small yet easily handled - as 8-pins DIP package:

This is the **LPC810** from NXP (used to be part of Philips). Make no mistake, it *is* a small and very low-end ARM chip. After all, with 2 pins used for power, this leaves just 6 I/O pins, and four of them need to be used in a specific way to load software into that chip!

Some specs: 4 KB flash, 1 KB RAM, 2x USART, 1x I2C, 1x SPI, 4x 32-bit timer, 4x PWM.

The chip runs on 1.8 .. 3.6 V, consumes at most 3.3 mA, and powers down to 1 µA.

As you’ll see, this little 8-pin chip is filled with oodles of 32-bit fun, and it’s surprisingly easy to fool around with. It even includes some *magic*, in the form of a “switch matrix”.

**Just an ARM and a LED**

So with the LPC810, what’s the *minimal* support circuitry needed to be able to use it?

It obviously needs power, i.e. a supply voltage between 1.8 and 3.6V, so a pair of AA batteries would work. In *theory*, a minimal setup can be created with a battery, the LPC810 chip, and some wires. But in *practice*, due to the electrical properties of the battery and the connecting wires, we’ll also need a 0.1 µF *decoupling* capacitor.

To summarise, this is the absolute minimal working setup for an LPC810 µC chip:
But there’s a problem here, because this chip has no way to communicate with the outside world. We could add an LED, but the bigger problem is much more fundamental: there is no way to instruct the µC what to do. What use is a micro-controller you can’t control?

We’re going to have to add a serial communication link for “uploading”, and one way to do this is with an “FTDI interface” plus a USB cable: one side connects to a host computer via USB, the other side connects to the target µC as a “logic-level” serial port.

For uploading software to the LPC810 µC, four I/O pins need to be connected to it:

- pin 8: USART receive (RXD) - tied to FTDI’s TX
- pin 2: USART send (TXD) - tied to FTDI’s RX
- pin 5: the ISP pin - tied to FTDI’s RTS
- pin 1: the RESET pin - tied to FTDI’s DTR

Add to that the supply voltage (VDD, pin 6) and the return ground line (VSS, pin 7), and you can see that we’re in fact connecting 6 pins out of the total 8 to be able to do even the slightest meaningful thing with the LPC810. But no worries: we’ll get some pins back later.

There are two potential show-stoppers with a standard FTDI interface board such as the USB BUB by Modern Device (or equivalent ones by SparkFun and AdaFruit):

1. it normally supplies 5V, which is too high for the LPC810
2. it doesn’t offer an easy way to access the RTS output signal

There are a number of ways to address this. Here is the approach we’re going to use:

1. insert a red LED between the FTDI’s 5V and the LPC810’s VDD pin
2. perform a small modification on the USB BUB to get at its RTS signal

The change to the USB BUB is essential for simple uploading to ARM chips, and will come in real handy time after time, later on. Luckily, this modification should have no ill effect when used with JeeNodes or other Arduino-compatible boards. We’re simply extending the FTDI pinout a bit by allocating an unused pin on the 6-pin FTDI header.

This is the complete - minimal! - setup, needed to experiment with a bare LPC810 chip:

Lots of wires, but just 3 components: µC + LED + capacitor. And the FTDI BUB, of course.

It’ll be enough to make the LED blink, but also to perform various experiments using the USB connection as serial interface. There are still two free I/O pins for us to play with, and there are in fact up to 6 if we’re careful. Plenty for lots of fun uses!

But first things first. The immediate task ahead is to implement the famous “Hello world” litmus test of physical computing: a blinking LED. And as you’ll see, ours will add a unique ultra low-power twist to it.

**Long live the breadboard**

Breadboards are the electronic hobbyists editor, so to speak. They let you connect stuff together and try out different circuit ideas without soldering. Here’s a mini breadboard, with on the right the bottom view when its protective adhesive layer is peeled off:
As you can see, all the pin holes on the top are connected together in groups of five. Keep this in mind while looking at the complete circuit we’re now going to assemble:

(The above image was created with Fritzing³, a very nice entry-level electronics design tool)

You can deduce how everything connects together from the above, but to get a real insight on how things tie together exactly, it’s better to look at the corresponding “schematic”:

The components here are not drawn to scale, but all the components are listed, as well as all the pins and all the wire connections. This is the definitive reference for this circuit.

It’s easy to make mistakes, in which case this thing won’t work, or may even get damaged. Here is

³http://fritzing.org/home/
a first step, with the main components, and the wiring needed for supplying power:

Note the tiny orange capacitor between the µC’s pin 6 and 7 (it’s not a wire!) and the red LED, which needs to be inserted with its longest lead pointing to the right, i.e. connecting to the white wire. When reversed, things won’t work. The µC’s pin 1 is on the bottom left.

You can check proper operation so far by connecting this to a USB BUB and powering it up - which should cause the red LED to light up. Here’ is the complete circuit, with the rest of the connections built up using jumper wires, a messy but very flexible technique:
Note that the placement of components and wires is slightly different from the Fritzing diagram. The result is the same, these circuits are equivalent with respect to the schematic.

That’s it. Our µC circuit is ready to go. Now we need to figure out how to make it blink.

**Minimal code, the basics**

The power of µC’s is their immense flexibility, because they are completely controlled through *software*, most often nowadays in the form of program written in C or C++.

If you’re familiar with Arduinos (or JeeNodes), then brace yourself: we’ll be going a lot deeper here, and we’re going to set up our own “runtime startup code”. We’re going to completely take control of the µC from the microsecond it starts up. The benefits are a deeper understanding and a much lower code overhead, the main drawback is that it will take somewhat more effort to get there than downloading the Arduino IDE and firing it up.

But no worries. Here too, our minimalist starting point will pay off nicely: we only need to set up a few things to get our blinking LED going.

So, here’s the circuit again, now stripped to its bare essentials. How can we make it blink?
There’s something odd going on here, and it’s very much on purpose: the LED is inserted \textit{in series} with the µC, while the whole thing is powered from the FTDI’s 5V supply pin. As you may remember, that’s more than the maximum allowed for an LPC810 chip, but here’s the trick: a red LED tends to have a \textit{forward} voltage drop of about 1.7V, i.e. when conducting. So with 1.7V less, the µC is in fact getting power as it should: with a supply voltage of ≈ 3.3V (anything between 1.8 and 3.6 V is fine for the LPC810). Note that the LED’s primary role here is not to light up, but to act as a basic \textit{voltage reducing} component!

Anyway, the challenge is to make that LED blink. And the solution is a bit unusual: we’re going to make the µC consume more or less power. While drawing current, the LED will be on. When asleep, the LED will be off. By \textit{modulating} power use, we can control the LED.

So the \textit{goal} is to make the LED blink, but the \textit{task} of the code will be to power up and put the LPC810 into an ultra low-power sleep mode. If we can do this periodically, we’re home.

Let’s do it:

```c
#include "LPC8xx.h"

int main () {
    LPC_SYSCON->SYSAHBCLKCTRL |= 1<<9;  // SYSCTL_CLOCK_WKT
    LPC_PMU->DPDCTRL |= (1<<2)|(1<<3);  // LPOSCEN and LPOSCDPDEN
    LPC_WKT->CTRL = 1<<0;               // WKT_CTRL_CLKSEL

    NVIC_EnableIRQ(WKT_IRQn);

    LPC_SYSCON->STARTERP1 = 1<<15;     // wake up from alarm/wake timer
    SCB->SCR |= 1<<2;                   // enable SLEEPDEEP mode
    LPC_PMU->PCON = 3;                  // enter deep power-down mode

    for (int count = 0; count < 900000; ++count)
        __ASM("\n        // waste time while drawing current
        ");

    LPC_WKT->COUNT = 5000;             // 10 KHz / 5000 -> wakeup in 500 ms
    __WFI();                          // wait for interrupt, powers down

    // waking up from deep power-down leads to a full reset, no need to loop
    while (true);                     // yak shaving: 4 bytes less ;) 
}

extern "C" void WKT_IRQHandler () {
    LPC_WKT->CTRL = LPC_WKT->CTRL;    // clear the alarm interrupt
}
```
The first point to make about this code is that you’re not expected to read through this and nod in complete agreement and understanding. There’s really a lot going on here, and it’s happening at a frightfully primitive chip level. Lines such as `LPC_PMU->PCON = 3;` are a way to directly manipulate the hardware of the LPC810 chip in C. And the documentation for all this is buried in a 370-page datasheet (PDF: UM10601⁴) - all for a tiny 8-pin µC chip.

Some notes about the above code, to give you an idea of what’s going on:

- **lines 7..15⁵** are required to prepare the chip for ultra low-power sleeping
- **lines 17..18⁶** spend 0.5 seconds doing “nothing” to keep the LED on
- **lines 20..21⁷** set up a 0.5 s watchdog timeout and enter “deep power down” mode
- **line 24⁸** is never reached, because the µC resets itself when it wakes up again
- **lines 27..29⁹** define a wake-up time interrupt handler which clears the interrupt

That’s not the whole story. You’re looking at the main() code, but there’s another 180-line C file which is essential to make the µC call our main code. Not to mention 5 header files with 3,000 lines of definitions from ARM and NXP, as needed for all this.

If you’re curious, it’s all on the web for you to look at. This example and everything needed to build and upload it is in a new GitHub repository from JeeLabs called embello¹⁰. The files for this minimal example can be found in this directory¹¹.

There are two steps missing to actually make our LED blink: (cross-) compiling the above software and uploading the resulting binary file into into our LPC810 chip. We can skip the compile step for now, because the compiler output file is also on GitHub as “firmware.bin”.

We just need to figure out how to load our carefully crafted code into the chip.

### The upload conundrum

*Uploading* plays a central role in embedded computing. Unlike the processor in your laptop or mobile device, where you can simply install an app, or - if you’re a software developer - compile from source code and launch the executable, embedded µC’s have extremely limited resources. No disks, no Ethernet or USB even on the simpler chips, and definitely no room for a software development environment, or even a code editor or command shell.

---

⁵https://github.com/jeelabs/embello/blob/master/explore/1446-lpc810/minimal/main.cpp#L7-L15
¹⁰https://github.com/jeelabs/embello
¹¹https://github.com/jeelabs/embello/tree/master/explore/1446-lpc810/minimal
Instead, we have to use our laptops (or desktops) as “host” to “cross-compile” the code which is to be run on the “target” µC. Going from here to there requires special dance steps.

Let’s assume that the code has been cross-compiled for now, and is ready as a binary “firmware image” to be sent to the µC to try it out.

On AVR chips like the ATmega328 or ATtiny85, we can use an *In-System Programming* (ISP) connection, which requires 6-pins: 2 for power, plus 4 for RESET and an SPI link (don’t confuse “SPI” and “ISP”). The standard ISP hookup uses a 2×3-pin 0.1” header:

All ARM-based chips have either a JTAG or SWD interface. The latter is more popular with small chips, because it requires only two I/O pins. This is usually a 2×5-pin 0.05” header:

JTAG and SWD are considerably more than just an upload interface: you can start and stop the µC, and set “tracepoints” and “watchpoints”. These hardware debugging features, which also include single-stepping through the code, can be incredibly useful when chasing some complex bug or trying to understand what the hardware is really doing.

The convention, also in the LPC810, is that ARM µC’s power up with the two SWD I/O pins enabled, but this feature can then be turned off in the code and the pins re-used otherwise. So out of the box, every ARM chip is controllable and debuggable through its SWD pins.

The drawback of using AVR’s ISP or ARM’s SWD pins is that we need special hardware to connect things to our laptop. With working code, all we really want is a way to upload it.

In a way, all this is unavoidable: to power up a chip, you need power pins, and to send data to a chip, you need I/O pins. But power pins get re-used once debugging is over, since power has to be applied anyway. And a few I/O pins are always going to be useful for debugging and testing, as a way for the code in the chip to report what’s going on.

The solution is to re-use the same two I/O pins for uploading code *and* for communicating serially...
with a chip when it’s running that code. We might need those pins in our projects, but at least while debugging we really have to have a solid communication mechanism.

It all depends on how many spare pins there are on the chip (as Murphy keeps telling us: never enough!). From the above, the minimum appears to require 4 pins, including power.

Unfortunately, that’s not enough. For control, we also need to be able to reset the chip, if the code is not working properly (and as Murhpy gets his way, that’s often the case!).

On Arduinos, JeeNodes, and other AVR-based systems the whole uploading dance is indeed based on these three I/O pins. But it requires a trick, in the form of a “boot loader” which is given control whenever the µC starts up after a reset. This boot loader then takes care of the communication and of storing new firmware into permanent flash memory.

But how did that boot loader get there?

Easy: through ISP. Except that now we’re back to square one. On most AVR chips, someone has to install a boot loader on the µC, and then we get the convenience of uploads and serial comms. The boot loader will wait a short time to see if uploading is requested, and otherwise jump to the code currently stored in the remaining flash memory.

With ARM chips, we have one more trick up our sleeve, but it requires the use of one extra I/O pin. Ok, nothing comes for free - but as you’ll see further on, it’s not such a big deal.

The trick is that ARM µC’s include a boot loader in ROM, i.e. built into the chip and delivered ready to go from the factory. It’s always included, and it can’t be broken or lost.

In the case of the LPC810, there’s a “serial” boot loader on the chip, which (surprise!) starts up after a reset, and (surprise!) uses a simple serial protocol on two I/O pins. It depends on one additional convention: a specific “ISP” I/O pin on the chip needs to be low (“0”) when the chip comes out of reset, to activate that serial boot loader. When high (“1”), the code already in flash memory will be started.

So what we need, is a way to control both the RESET pin and the ISP pin.

We’ve now lost 4 I/O pins to get code upload support. The good news is that all those I/O pins can be re-used and re-allocated at will once the µC has started running. The only requirement is that the RESET and ISP pins must start as high during normal power up.

Note that the two SWD pins are still available for hardware debugging if we want ‘em. All of the above can be squeezed into an 8-pin DIP chip. Which is exactly what the LPC810 did:
Pins 3 + 4 are easy to use in a project (disable SWD), then 2 + 8 (no in-circuit uploads), and lastly 1 + 5 (as long as they are high on power-up, perhaps by using them as outputs).

### Modifying the USB BUB

The upload puzzle is solved if we can find a device which connects to USB, has power and ground, has a serial link (i.e. send and receive lines), and has two output pins which can be controlled from the USB side.

We’re *almost* there with what has got to be one of the most common interfaces in embed-land by now: the “FTDI interface”. It meets all of the above requirements, except that it only has one output pin, tied to DTR on the USB side.

There are tons of these on the market, some of questionable origin perhaps, but all extremely widely used and highly popular (as FTDI Ltd [found out]¹²).

The solution is to make one or two changes, depending on which type of FTDI board you’re using. This requires some soldering skill, and you have to do things calmly and carefully...

Note that this applies to any FTDI board with the following properties and jumper settings:

- supply voltage on the µC side is 5V, not 3.3V
- logic signal level is 3.3V, not 5V
- has a 6-pin “FTDI header”
- based on the FT323RL chip, as most boards are

The first two requirements are not essential, in that other combinations can be made to work with ARM chips, but it’s harder to get this right and you may get burned one day when you mix & match, and hook things up to another FTDI board with different settings.

The “5V power with 3.3V logic” setting is the default for all the boards from JeeLabs, and probably also most of the other Arduino-compatible boards out there.

Let’s get started. The worst that can happen is a “bricked” FTDI board - take your time!

#### Modification 1 - connecting RTS

There is usually a pin on the FTDI header side of the board which is marked CTS (Clear To Send). This pins is not used with Arduinos. It’s of very little use, being an input pin.

The modification needed is to connect it to the RTS (Request To Send) pin of the FTDI chip, which is an output. There’s no need to disconnect the CTS input, since no one uses it.

The *bad* news is that on many FTDI boards, the RTS pin on the chip is not connected to anything. That means you have to have steady hand, a fine-tipped soldering iron, and soldering wick to correct any problems - always a good idea for your toolset, by the way!

Here is the wire you’ll need to add in this case, very carefully:

![Image of USB FTDI](image1.jpg)

The RTS signal is on pin 3 of this 28-TSSOP package - those pins are only 0.5 mm apart!

The good news is that the “USB BUB II” available from Modern Device\(^ {13}\) and resold in the JeeLabs shop\(^ {14}\) has RTS exposed via an internal pad. This makes modification a lot easier.

Connect a wire between the CTS header pin and RTS, as shown below:

![Image of USB BUB II with wire](image2.jpg)

Make sure that the wire doesn’t touch anything but the RTS pad and the CTS header pin.

**Modification 2 - disable the reset cap**

This modification only applies to the BUB II, which inserts a capacitor between the chip’s DTR output signal and the FTDI header. This is harmless for JeeNodes, but will interfere with proper reset operation on ARM chips.

Locate the tiny “C4” SMD capacitor on the board and either replace it with a short wire, or simply solder a wire on top to short out the capacitor, as shown in the image above.

\(^ {13}\)http://moderndevice.com/product/usb-bub-ii/
\(^ {14}\)http://www.digitalsmarties.net/products/usb-bub
That’s it. This modified BUB will continue to work as before with JeeNodes, but it now has that extra RTS output pin available to also properly perform uploads on ARM chips.

If you absolutely totally can’t wait to upload that blink demo to your own LPC810 chip…

On MacOSX, using Homebrew:\footnote{http://brew.sh}

\begin{itemize}
\item install the open source \textit{lpc21isp} utility: \texttt{brew install lpc21isp}
\item type: \texttt{lpc21isp -control -bin firmware.bin /dev/tty.usb* 115200 0}
\end{itemize}

On Linux (Ubuntu), using \textit{apt-get}:

\begin{itemize}
\item install the open source \textit{lpc21isp} utility: \texttt{apt-get install lpc21isp}
\item type: \texttt{lpc21isp -control -bin firmware.bin /dev/ttyUSB* 115200 0}
\end{itemize}

Otherwise, just hang in there and it’ll all be described in a lot more detail Real Soon Now.

\section*{Uploading from Mac OSX}

In a way, Apple Macintosh computers running OSX are a curious - some might say \textit{schizophrenic} - mix of technologies: out of the box, you get an appliance full of icons and windows, ready to go and operated primarily through a mouse or trackpad:

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{macosx.png}
\caption{MacOSX Interface}
\end{figure}

But underneath sits a \textbf{POSIX}\footnote{https://en.wikipedia.org/wiki/POSIX}-compliant operating system which is every bit as much like \textbf{Unix}\footnote{https://en.wikipedia.org/wiki/Unix} as Linux and FreeBSD - you know, that invention from the 1970’s, \textit{decades} ago. Some things are so
timeless and able to evolve to meet new needs, that they tend to last virtually forever. Like wheels, the written word, transistors, the C programming language, Unix?

The gateway to this hidden world is the Terminal app (it’s buried in the Utilities folder):

Transitioning from that first screen to the second is like moving to the next game level - turning from a consumer to a creator of software. The terminal, or more exactly the “command shell” inside it, unlocks great powers. With this power comes responsibility, but it’s actually a very safe and robust playground, as long as you take some precautions:

• under the hood, a Mac OS X disk often has over a million files: yes, it’s complex
• don’t mess with it, try not to change anything outside your home directory
• with one exception: install Homebrew to manage all the files in /usr/local/…
• and also: try not to mess with the Library folder inside your home directory
• only use “sudo” (to get superuser privileges) if you know what you’re doing

This way it’s very unlikely that you’ll interfere with what the Mac and its “apps” are doing.

**Homebrew**

Homebrew¹⁸ is a package manager. It lets you install (and uninstall!) packages prepared by others and made freely available. It’s actually much more than that: in the long run, it also takes care of your sanity. With brew, as the command is called once installed, you gain access to a wide range of software tools (some 3,000 at last count) - to install, try out, update, or remove at will. The key is that it also manages all package dependencies, and that it goes out of its way to let know know how to solve problems if anything goes wrong.

¹⁸https://github.com/Homebrew/homebrew
With Homebrew, life is good. Without, your Mac is a ticking bit-rotting time-bomb.

The installation instructions are on brew’s home page¹⁹. You will be asked to install Xcode as well, which is Apple’s development environment. The reason for this is that it includes a base collection of command-line utilities, most of which are essential during development.

Once installed, life is easy. These commands are really all you ever need to deal with:

- `brew search ...` - find out what packages Homebrew knows about
- `brew home ...` - open the homepage associated with a specific package
- `brew install ...` - install the specified package and all its dependencies
- `brew uninstall ...` - uninstall the specified package and its dependencies
- `brew update` - get brew’s latest list of packages and versions from the web
- `brew upgrade` - download updated versions of the packages you’re using
- `brew cleanup` - remove files belonging to obsolete or outdated packages
- `brew doctor` - verify that all is well, report any potential troublespots

That’s a bunch of commands to get used to, but you can always type `brew` for a summary.

Packages are added to Homebrew all the time. Most packages get updated shortly after a new release is out, so Homebrew is also a fine way to keep your tools up to date. With Homebrew, it’s easy to keep a Mac running well, up to date, and in good shape for years. In fact, the adage “nothing gets installed, other than as Apps or via Homebrew” is a fine one.

For anything experimental, be it the code you are developing or code from others which is not production-ready, your safest bet is to keep all that inside your own home directory.

**lpc21isp**

One reason for that whole `brew` stuff, was to get access to an open source utility called `lpc21isp²⁰`. This tool knows about the protocol used by the serial bootloader of the LPC810, and is able to upload to it as well as manage a serial connection for debugging.

Getting this tool on your Mac couldn’t be easier now - `brew install lpc21isp`

After that, you can type `lpc21isp` as command to see a brief summary of how to use it.

In our case, assuming you have a copy of the `firmware.bin²¹` code you want to upload to the LPC810, the steps to send a firmware image to the LPC810 are as follows:

- plug in the (modified) FTDI board, with the breadboard setup described earlier
- type: `lpc21isp -control -bin firmware.bin /dev/tty.usb* 115200 0`

Here is what a successful upload looks like:

---

¹⁹[http://brew.sh](http://brew.sh)
²¹[https://github.com/jeelabs/embello/tree/master/explore/1446-lpc810/minimal](https://github.com/jeelabs/embello/tree/master/explore/1446-lpc810/minimal)
So much for uploading on the Mac - it’s all Unix with a lot of 21st century lipstick, really.

**Uploading from Linux**

*If the “Mac” is not your cup of tea, how about a tasty sip of Linux?*

The Linux\(^\text{22}\) story has been told many times. If anything can represent the success of open source software, then surely it has to be Linux. It’s free, it’s extensible, and it’s everywhere. Anything\(^\text{23}\) with a CPU chip, a modest amount of RAM, and some storage can run Linux.

While some people run Linux as the main operating system on their main computer, this is still a fairly small group. You need to get into chipsets and all the details of the attached peripherals and screens to get things working as consistently as Windows or Mac OS X.

But there are in fact much simpler ways to get started with Linux and use it for software development. A task for which Linux is eminently suited, due to the huge amount of available software and the extreme degree to which it can be customised and automated:

- set up a virtual machine (VM) running Linux under/next to your main work setup
- get a low-cost Raspberry Pi, BeagleBone Black, or Odroid embedded board
- set up a remote server in the cloud running Linux

A remote server is usually not very practical for physical computing projects, which need to connect to the outside environment through USB or direct I/O pin hookups, though Ethernet is definitely a

---

\(^{22}\)https://en.wikipedia.org/wiki/Linux

\(^{23}\)http://hackaday.com/2014/11/18/running-debian-on-a-graphing-calculator/
useful option for projects which connect to a LAN or internet. Still, if you just want to get familiar with Linux, setting up a server with say, DigitalCloud²⁴ is a matter of minutes and will cost you no more than €5 per month (and no minimum).

The choice for a Linux setup need not be restricted to a single one - with low-end Linux boards available for under €25, there’s really little point agonising over what is best. It’s mostly a matter of convenience and long-term stability.

**Virtual machines**

In terms of convenience, it’s hard to beat the use of a VM: it provides a complete context right next to the setup you’re working with anyway, it’s easy to integrate into your back-up procedures (you do have one, right?), and it’s as fast as what you’re already used to.

All you need is the software to create such a virtualised environment. The big three are VMware, Parallels (on Mac OSX), and VirtualBox. They’re all fairly similar - VBox is free:

![Virtual Machine Screenshot]

The mechanism is always the same: you start up the app (VBox needs registration to add USB support), and you give it a Linux installation CDROM or DVD, or a disk image. From then on, it’s as if you’re running on a separate machine - and in a sense, you are indeed.

There are some important implications:

- it takes extra disk space, as much as you’d need on a real machine (some 4 .. 20 GB)
- it needs extra RAM, since the VM is running next to your OS (for Linux, 1 GB is fine)
- you need to set up a network connection, the VM has to “go through yours” to get out
- you have to decide and manage which USB devices connect to the VM (exclusively)

If you want to set up a complete graphical user interface under Linux, you can, but it’ll require more RAM and more disk space. The alternative is to set up a minimal VM, very much like you would on a remote server: command-line only, with network access via SSH, the “secure shell”. A basic setup can be done with 2 GB of disk space and 512 KB of RAM.

The benefit of a locally-installed VM is that it becomes part of your work environment. No need to hook up to anything external, no extra hardware (assuming that your setup has enough resources).

²⁴https://www.digitalocean.com
All while not in any way affecting your own system - Linux lives on the side, it won’t (and cannot) interfere with anything else on your machine, and vice-versa.

A second benefit is that a VM can be suspended - frozen in mid-air, and resumed in exactly the same state later on. No need to shut down, no need to stop processes running on Linux.

One drawback is that you do need a setup with a decent amount of RAM (4 GB at least, probably), and some spare disk space (2 .. 4 GB would be a minimum to have available).

Another drawback is that USB connectivity may be tricky and finicky at times. The VM software is playing nasty tricks at the operating system level to make USB devices available to the Linux VM, and sometimes a restart or unplug shuffle is needed to make it all work.

**Dedicated hardware**

The other option is to get one of these, or any of their ever-more abundant alternatives:

[Images of various ARM-based boards]

These are all ARM-based boards, from left to right:

- **Raspberry Pi B+** [25] from the Raspberry Pi Foundation
- **BeagleBone Black** [26] from Texas Instruments
- **Odroid U3** [27] from Hardkernel
- **AM3352-SOM-EVB** [28] from Olimex

All of these have some particular mix of: 1 .. 4 CPU cores, 0.5 .. 2 GB RAM, either 2 .. 4 GB on-board flash or a µSD interface, an Ethernet port, 1 .. 4 USB host ports, a HDMI or similar video interface, and prices ranging from €40 to €80.

For each of these there is either a Debian- or Ubuntu-based distribution (as well as others).

What’s so great about these boards:

- choice: you pick your mix and options, with corresponding modest price trade-offs
- availability: some of these boards are selling like, ehm, well, hot pie’s [29], sort of ...

---

28 [https://www.olimex.com/Products/SOM/AM3352/AM3352-SOM-EVB/](https://www.olimex.com/Products/SOM/AM3352/AM3352-SOM-EVB/)
• interchangeability: at a fairly broad level, all of these boards are very similar
• documentation: plenty of documentation and forums for these boards, and for Linux
• truly low-power, usually 3.5 Watt via a 5V adapter, perfect for always-on use
• simple USB and LAN connectivity, and in some cases tons of freely usable I/O pins

An embedded Linux board does have some issues which you need to be aware of:

• with an active system running off µSD cards, flash will eventually wear out and fail
• you have to think carefully how to set things up for backups, especially if always-on
• when messing with those I/O pins, you can damage and even kill the whole unit
• each of these boards is different, figuring out all the details can eat up a lot of time
• sometimes new revisions are released, with pesky little differences to trip you up

One practical strategy, if you’re willing to invest a bit more, is to purchase two identical setups right from the start. Then you can use your laptop to regularly copy the main µSD card to an extra copy, and use the second system as development c.q. staging backup, and as comparison in case you ever suspect any hardware damage.

Compared to a virtual machine, an embedded board will affect how you work:

• you need to either attach a keyboard + monitor, or connect via SSH (preferable)
• you now have two systems to deal with: your own laptop and this extra computer
• this also means you’re probably going to transfer files between them (regularly!)
• these boards are a lot slower than your laptop, i.e. longer compile & build times

But it’s definitely a very nice low-impact way to get your feet wet in the Linux world. And if you’re planning to set up a home automation server, or some other project requiring a permanent “server”, then chances are that you’re going to need one if these boards anyway.

Choosing a “distribution”

Linux is available in many flavours. Tastes differ. This article will use Ubuntu³⁰, which is based on Debian³¹ - a solid configuration using “apt” and “aptitude” as package managers.

The nice thing about Ubuntu (at least 14.04 and later), is that it knows about the “lpc21isp” package. Installing it is as easy as typing:

³⁰http://www.ubuntu.com
³¹http://www.debian.org
1  sudo apt-get install lpc21isp

As with Homebrew on Mac, the “lpc21isp” command can then be used on your system.
Here’s how to upload a file to the LPC810, almost exactly the same as on Mac OS X:

- plug in the (modified) FTDI board, with the breadboard setup described earlier
- type: lpc21isp -control -bin firmware.bin /dev/ttyUSB* 115200 0

If you get an error, there may be a problem with access permissions. Usually, the way to solve this
is to add yourself in the “dialout” permission group, using this command:

1  sudo usermod -aG dialout <your-linux-username>

Then logout and log back in to make these changes work (or restart Linux).
Keep in mind that with a virtual machine environment, you will have to “attach” the USB device
to the VM, preferably in a permanent way, so that later unplugging and re-plugging will still work.
Use the following command to look at the end of the system log, to see whether the inserted USB
device is indeed being recognised by Linux:

1  dmesg | tail

Welcome to Linux: there’s a command for just about anything you can imagine, and more. A good
one to remember is “apropos …”, which lists all installed packages related to “…” It does this by
going through the list of installed manual pages. Ah, yes, that’s another useful one to remember -
try “apropos nxp” and “man lpc21isp”, for example!

**Uploading from Windows**

Which leaves us with Windows, the operating system most readers are probably using:
While setting up a development environment for Windows is definitely feasible, this story is not about developing under Windows, but developing under Linux under Windows, i.e. running Linux in a virtual machine and handling compilation & uploading from there.

The main reason is that Linux is highly geared towards automation - of the development process itself that is. Which is not so surprising, when you consider the fact that it was built by and for software developers. With Linux, you can augment your own developer skills.

**Soapbox time**

With a little setup, hitting a few keys in your editing environment is all you need to save your changes, perform a cross-compilation, build the firmware image, report some statistics, and upload the result to the LPC810, or any other µC for that matter (including ATmega’s). This is not a gimmick - each of those little conventions, time savings, and habits you collect add up to becoming a major productivity boost over time.

Note that this is *not* about “everyone must learn editor X”. You can in fact continue to write and edit your source code in whatever tool you like: it’s quite simple to have a VM share an area on the disk and pick up changes so the same files are also visible within the Linux VM.

This *is*, however, a gentle way to nudge you into using the command line, “make” (a terrific workhorse for automation), and command-line based compilers, linkers, debuggers, uploaders, and terminal emulators. The Arduino IDE is very nice, but it’s single purpose. No matter how good it
might be at editing and compiling and uploading, it won’t let you streamline your own workflow in ways which were not envisioned by the Arduino team.

There is much more to embedded software development than editing and compiling and uploading firmware: for example, you might wish to implement or integrate some “host-side” software, such as a central home automation server driving a network of remote Wireless Sensor Nodes. With a Unix / Linux mindset, the whole setup can become a single (but evolving) environment. How about securely downloading new code, compiling it, and uploading it to remote nodes via that same central server? Can do - it’s just a matter of designing the right workflow and automating it. Linux is good at long-term automation.

This does mean you’ll need to become familiar - and even proficient - at working “on” the command line, typing in commands at the “shell”, and setting up little routines for testing, comparing results, repeating some process, or launching debugging sessions. Even brief tasks such as diving in deep to chase a bug, can benefit from setting up some quick-and-dirty helpers, to avoid you from pressing the same buttons over and over again while going through an “edit-run-debug” cycle.

Mice are great. IDE’s are great. But in repetitive cycles with some variability, such as long-term development, or intense testing or debugging sessions, every step saved helps. Your muscles will thank you. So will your co-developers. Even your brain will thank you, as all the routines and repetitions end up in shell scripts, makefiles, and muscle memory.

Getting sources from GitHub, creating snapshots, comparing changes, installing software... once every action can be performed through the command line, it can all be automated. Anything done more than a few times, and anything you might have to do again sometime in the future is worth casting into a few lines of a “shell script”, or as part of a “makefile”.

Imagine being able to type “make help” in any project you’ve ever worked on, to get a list of actions you can perform in that project. Regardless of tools used, platforms used, or even the problem domain. All you need is one simple text file in each project directory.

Even an elaborate programmer’s environment such as the Eclipse IDE can’t quite cover the breadth of such an approach, and it’s considerably more complex to manage Eclipse than to verify that you have the “make” command installed on your (virtual) machine.

The fact is that “large” environments such as IDE’s are almost always constructed on top of command-line tools anyway. But we can look under the hood, and ignore the shiny veneer layer added on top. Sure, veneer is great and beautiful, but real power lies underneath.

**Linux in a VM**

So the task ahead is to create a virtual machine running Linux under Windows, and then to integrate things in such a way that they can both be used conveniently together.

We’ll use VirtualBox³² for this, the same product which is also available for Mac OS X (and Solaris, and Linux). Getting started is easy:

³²https://www.virtualbox.org
• download and install the latest version from their download page\textsuperscript{33}
• you also need to install the “VirtualBox Oracle VM VirtualBox Extension Pack”, to add support for USB ports within the VM (this step requires registration)

VirtualBox is open source software (originally from Sun, now Oracle), but the extension pack isn’t, even though both are free for personal use.

You now have the ability to create virtual machines. The next step is to obtain a server version of Linux. Ubuntu is solid, popular, and well-supported, so we’ll use that, as before:

• download a copy of the server edition image, preferably a 64-version - listed here\textsuperscript{34}

Using the server edition means that the system installs without GUI or desktop (which can be added later on, if needed). We’re going after the command-line, remember?

A server install is a fraction of the size of a full install (under 1 GB versus ≈ 3 GB disk use).

**VirtualBox setup**

Click on the “New” button in VirtualBox and enter a name (“minnie” in this example) and adjust the other settings to match what is shown below:

\textsuperscript{33}\url{https://www.virtualbox.org/wiki/Downloads}

\textsuperscript{34}\url{http://www.ubuntu.com/download/server}
Click on “Create”, which leads you to the main window again:
You need to “attach” the downloaded Ubuntu disk image to this VM. Click on “Settings”:

Select “Storage”, then click on the “empty” IDE controller, and then click on the CD icon next to the
text marked “IDE Secondary Ma(ster)” Select the *.iso file you downloaded.

That’s it. Your new VM is ready to launch. At this point it is still empty, but it will boot from the virtual CDROM drive, and then go through the standard procedure of setting up Linux. Be sure to select “Install to hard disk” (which is virtual, but Ubuntu doesn’t know that).

The whole installation process of Linux itself will consist of about a dozen questions and should take less than half an hour. It’s a once-only process.

At this point, you have a working Linux environment running inside Windows, although neither of them knows much (if anything) about the other.

**Bridging two worlds**

To complete the process, you need one more tool: an terminal emulator called Putty\(^{35}\). Please download and install it according to the instructions on its homepage.

Just to regain our perspective for a moment: the goal of this whole exercise is to run Linux under Windows, and then hook it up to the network and USB. Once configured (in VBox), the Linux part is essentially ready. We can then start / suspend / resume Linux at will.

*Hang in there... there is still quite a bit of nasty (one-time) configuration left!*

The second part of the equation is Putty: it can be used as a (secure) terminal window into any SSH server on the network, including the Linux VM on this machine itself. It needs to be configured to connect to the VM (which must be running).

The details of this VBox + Putty configuration are a bit too complex to include here - they are described in a separate article, see Setting up the Virtual Machine\(^{36}\).

For now, here’s what the result of all this fiddling will look like, i.e. connecting Putty to the VM, entering a Linux login password, and typing a few standard Linux commands:

\(^{35}\text{http://www.putty.org}\)

\(^{36}\text{http://jeelabs.org/book/1447c2/}\)
This is an application on Windows (Putty) logging into another application (VBox) running on the same Windows machine. *Inside* Putty, you’re in Linux - *outside*, you’re in Windows!

But there is one more trick which really helps keep these apart (as if the difference in appearance weren’t enough) - press ALT+ENTER and the Putty window goes fullscreen. You now have two operating systems running side by side on the same hardware.

## Uploads

Even though this setup goes much further, and took quite some effort, the original goal was to upload a firmware image to the LPC810. The good news is that once the USB ports have been set up properly in Vbox, this is now the same as on a Linux or Mac OS X machine:

- plug in the (modified) FTDI board, with the breadboard setup described earlier
- `type: lpc21isp -control -bin firmware.bin /dev/ttyUSB* 115200 0`

Everything is back in sync, whether you’re a Windows, a Mac, or a Linux person.

Some will see a crude text-only window and regard it as the lowest common denominator, but that’s not really so: you now have the foundation on which to build any embedded software, with a world of software and automation tools within reach of a few keys.

Yes, “keys”: mice and trackpads can still be used (really!), but they play a considerably smaller role in this command-line environment.
Life after uploads

Ok, the LPC810 is making an LED blink... now what?

Ignoring for a moment how you can modify and compile your own (variations of the) software, this really isn’t about blinking LEDs, of course. That’s just an appetiser.

The real value of the LPC810 is that it’s so small yet easy to use in different projects, and extremely flexible due to its extensive hardware peripherals and total programmability. Or to put it differently: six pins of 32-bit µC goodness in a tiny 8-DIP package, powered from 1.8 .. 3.6V (yet 5V-tolerant), capable enough to run at up to 30 Mhz, yet efficient enough to go to sleep using less than 1 µA of current (i.e. over a decade on a coin cell).

If you use a DIP socket in your project, then programming these chips need not even be done “in-system” (the first two letters of the ISP acronym). In fact, the circuit described so far is a perfect programmer for chips - useful enough to be placed on a dedicated board:

![Image of LPC810 programmer circuit](image)

The socket allows quick programming and removal of an LPC810, so that you can fully use all the 6 I/O pins of the LPC810 for dedicated purposes, including serial, I2C, or SPI communication - or even a combination thereof.

The LED can be omitted when the power source is between 1.8 and 3.6V, such as a coin cell or 2 AA cells, meaning that all you need is the LPC810 and the 0.1 µF decoupling capacitor.

The fader³⁷ demo on GitHub illustrates a first refinement of the minimal³⁸ Bare ARM Blinker, and shows how to do a bit more between power down cycles - the result is a periodically fading LED. Not very useful, but the techniques used there can be applied elsewhere.

A more sophisticated example of things an LPC810 can be made to do is the 30-line sine³⁹ demo, also included in the jeelabs/embello⁴⁰ repository on GitHub. Keep in mind that “officially” the LPC810 only has a very limited analog-to-digital conversion capability, in the form of a 5-bit comparator, and no digital-to-analog capability at all.

⁴⁰[https://github.com/jeelabs/embello/](https://github.com/jeelabs/embello/)
Until we start pulling old tricks out of the new physical computing hat, that is:

The *sine* demo generates a 50 Hz *analog* sine wave on pin 3 of the LPC810. Or rather: it generates a sequence of PWM pulses, which - *after being passed through a 1 kΩ + 10 µF low-pass filter*\(^1\) - produces a fairly pure sine wave of approx 1 Volt peak-to-peak amplitude.

How’s that for a tiny digital chip running some bits-and-bytes software, eh?

And that’s just using 1 pin and under 1 KB of flash memory (half of which is for sine wave coefficients). What will *you* do with the remaining 5 pins and 3 KB of flash memory?

All the articles so far have been about how to set things up, and a few more are still needed about how to install and use the ARM cross-compiler toolchain. It’s all a bit tedious, alas.

But this is where the fun starts: with the proper tools in place, we can now begin to explore the essence of physical computing: working on useful and entertaining projects, using µC’s such as the LPC810 to implement all sorts of intelligent behaviour, or perhaps merely creating solutions which used to be much more complicated or expensive before, and tying sensors, displays, actuators, and various forms of communication together.

This is the playground where programming, electronics, perception, and motion meet. Where anyone can participate, learn, combine, and share. *And where curiosity is king!*

### Setting up the Virtual Machine

Setting up a VM with Linux for embedded software development takes some configuration. This is a sequel to *another article*\(^2\). Here is the *big picture* of the setup we’re aiming for:

---

\(^1\)https://en.wikipedia.org/wiki/RC_circuit

\(^2\)http://jeelabs.org/book/1447c/
When you install Linux and start the VM for the first time, you’ll get a console window attached to the VM. Inside this window it’s all Linux, but there are still some issues:

- the VM is (probably) not yet connected to the network, so it can’t reach internet
- the VM does not see any attached USB devices either, i.e. no FTDI interfaces
- there is no way to exchange data or files between the host system and the Linux VM
- the console is a bit limited, with Linux system messages interfering with normal use

Each of these should be relatively easy to solve and configure. So let’s get on with it, eh?

### Internet access

For networking, we need to define a (virtual) network interface in VirtualBox. This can be done in settings - here is an example of what it should look like:
You can test for network connectivity and internet access using this command in Linux:

```
$ ping -c 3 google.com
```

If no errors are reported, you’re good. One of the first things to do when connecting to internet for the first time, is to request the latest updates for Linux:

```
$ sudo apt-get update && sudo apt-get upgrade
```

(press return to accept the latest updates, if you get a question about this)

**Ignore the console**

The console is a bit limited, which is why the Getting Started page included the advice to download and install Putty. Using Putty, we can set up a terminal session to Linux, using a normal window-based application, which can be resized, set to full-screen, and connect to other systems as well - using the encrypted SSH support available in most Linux setups.

One approach would be to simply connect into the VM using that network connection we just set up. It would work, but it’s a bit inconvenient: if there is no network access, or if you switch between LAN and WiFi connections, then you may not be able to connect.

---

A far better option is to create a second network interface in the VM, which remains local inside Windows, i.e. no actual network interface involved at all. Here is the VBox setup:

![VBox Network Settings](image)

Note that this is a (virtual) “Host-only Adapter”. It’s not tied to a real network interface.

We need to set up networking on the Linux side, and find out the assigned IP address, so in the console, enter the following:
If `/etc/network/interfaces` shows something different from this, you’ll need to edit it. One way to do so is with the following command:

```bash
sudo nano /etc/network/interfaces
```

Double-check all the values, save the changes, and reboot Linux (type: “sudo reboot”).

The two “ifconfig ...” commands shown above are a quick way to find out if both network interfaces have been properly set up in Linux, and what IP addresses they were given.

Now we can configure Putty to connect to the VM that way, and save the configuration for easy re-use later. Here’s a configuration setup called “minnie”:
Note that the IP address matches the one assigned to the 2nd interface by VBox.

Save the changes first, then click on the “Open” button. You’ll see this after logging in:

Congratulations, you have connected Putty as terminal window to the Linux VM over SSH.

A benefit of this setup is that it works even when you are off-line, or running Windows’ firewall to constrain outside network access, since this uses a direct connection to VBox.

**Sharing files**

If you intend to edit your source code from Windows, then a shared disk is the way to go: we can create a shared area in VBox, which is placed in Windows, but which the Linux VM can access as if it were an attached USB drive. It’s well worth the effort to get this going:
In Linux, we can streamline things further, so that this USB drive gets *auto-mounted* every time Linux boots up. To do this, install the "usbmount" package:

```bash
sudo apt-get install usbmount
```

Then, add yourself to the "vboxsf" group to get the proper access permissions:

```bash
sudo usermod -aG vboxsf <your-user-name>
```

And lastly, log out and log back into Linux to refresh to these new group permissions.

Now, the same files on Windows will be shared with Linux as `/media/sf_xfer/....`

**USB Devices**

Lastly, we should make it really easy to plug in an FTDI interface (or JTAG debugger, or whatever) and have it appear as device in *Linux*, not Windows. Again, in VBox:
The way to add an entry, is to insert the FTDI board and click on the “USB/+” icon. You’ll get a pop-up to select the device. Then simply accept the suggested values and it’ll be saved.

Now every time you plug the device in, it will be attached to the Linux VM, thus completely bypassing the Windows host. No need to install FTDI drivers in Windows.

**Security**

The configuration described here should be as secure as before. Linux is reachable from the outside, but only through SSH if you haven’t opened up more ports. SSH is secure if you choose your login passwords properly - and even if you don’t, any damage will be confined to Linux, which has the same access rights as any other system on your (local!) network.

However, if you want to lock down things 100% tight, you’ll need to do two more things: 1) set up Putty to use a public/private key pair for logging into your own account on Linux, and 2) disable all logins over SSH using only a password. That way, there’s no way into Linux other than via Putty (which requires physical access to your machine), or if someone gets hold of your public *plus* private key files, stored in Putty.

To conclude this setup guide: there is unfortunately some tinkering involved in getting all of the above steps just right, but the pay-off is a well-integrated setup, whereby Windows remains as is, with VBox and Putty running inside, and Linux is fully tied into the network and any USB devices you wish to use from there.

With this approach, there is no need to ever install USB drivers in Windows, if all you want to do is use that particular USB device only from the Linux VM. There *is* of course a need to set things up on Linux, but in general Ubuntu is pretty good at recognising USB hardware.
So there you have it: total isolation, plus controlled screen, USB, and network access.

Setting up a toolchain

In software development, a “toolchain” is a set of tools to generate machine-executable code, i.e. compilers, linkers, and all the other utilities needed to make them work.

Linux

On Linux (including Linux running on a VM), there are two pieces to this puzzle:

- all the standard tools: “make”, “gcc”, and “g++” essentially, and everything they depend on - these are available as a single super-package called “build-essential”
- the cross-compiler for ARM: on Ubuntu, it’s available as “gcc-arm-none-eabi”

So all you need to do if you’ve been following along and are using Ubuntu 14.04 or later, is:

1. $ sudo apt-get install build-essential gcc-arm-none-eabi

Don’t include the “$” when typing, that’s the prompt displayed by the shell. In other words, in all these instructions: start typing the text from the point after the “$” prompt.

Hit return to accept the installation suggestions, and let things run for a few moments.

Here are some quick checks to verify that the toolchain has been installed:

1. $ make -v
2. GNU Make 3.81
3. Copyright (C) 2006 Free Software Foundation, Inc.
4. 
5. 
6. $ gcc -v
8. COLLECT_GCC=gcc
9. COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/4.8/lto-wrapper
10. Target: x86_64-linux-gnu
11. Configured with: ../src/configure -v 
12. Thread model: posix
13. gcc version 4.8.2 (Ubuntu 4.8.2-19ubuntu1)
14. 
15. $ arm-none-eabi-gcc -v
Getting Started

16 Using built-in specs.
17 COLLECT_GCC=arm-none-eabi-gcc
18 COLLECT_LTO_WRAPPER=/usr/lib/gcc/arm-none-eabi/4.8.2/lto-wrapper
19 Target: arm-none-eabi
20 Configured with: ../gcc-4.8.2/configure [...]
21 Thread model: single
22 gcc version 4.8.2 (4.8.2-14ubuntu1+6)

You’re done. This is all you need to recompile code for the minimal⁴⁵ demo and others.

Here is a sample compile session, again with some output truncated for brevity:

$ cd embello/explore/1446-lpc810/minimal
$ make
3 arm-none-eabi-g++ -mcpu=cortex-m0plus -mthumb [...] -c -o main.o main.cpp
4 arm-none-eabi-gcc [...] -c -o gcc_startup_lpc8xx.o gcc_startup_lpc8xx.c
5 arm-none-eabi-gcc -o firmware.elf -Wl,--script=LPC810.ld [...]
6 arm-none-eabi-objcopy -O binary firmware.elf firmware.bin
7 lpc21isp -control -bin firmware.bin /dev/ttyUSB* 115200 0
8 lpc21isp version 1.94
9 File firmware.bin:
10 
11 image size : 396
12 Image size : 396
13 Can't open COM-Port /dev/ttyUSB* ! (Error: 2d (0x2))
14 make: *** [upload] Error 2

That’s two source files compiled, linking everything together into the firmware.elf file, converting that to firmware.bin, and starting the lpc21isp uploader (which then fails because the FTDI converter was not present).

Make is a really nifty little utility: it processes the rules specified in the “Makefile”, doing only as much work as minimally needed. If a source file has not changed since the last time, then make knows that it does not need to run the gcc compiler again.

Here’s the output when calling make a second time:

$ make
2 lpc21isp -control -bin firmware.bin /dev/ttyUSB* 115200 0
3 lpc21isp version 1.94
4 [...] [https://github.com/jeelabs/embello/tree/master/explore/1446-lpc810/minimal]
As you can see, only the upload step is performed. In larger projects, “make” can have an immense impact on the speed of development. Just type “make” and it’ll only perform the steps which are needed, based on file modification dates. For example: if the main.o file is newer than the main.cpp source code, then there is no need to compile it again.

(there’s more to it due to include files, but make and gcc can be told to handle all that)

**Mac OSX**

On the Mac, all the standard development tools will have all been installed when you went through the installation of “Homebrew”, as previously described. But it takes slightly more work to install the cross-compiler toolchain for ARM:

- download the latest official build from this page on launchpad.net
- be sure to pick the “Mac installation tarball”, not some source code or zip file
- unpack its contents by double-clicking the downloaded results
- pick a location to store these files, perhaps “Tools” in your home folder

One last step is to make sure that the system can find these tools from the command line. So if you moved the files to "$HOME/Tools/gcc-arm-none-eabi-4_8-2014q3/", then you’ll need to add one line to a file called ".bash_profile", also in your home directory. Enter these three lines, making sure that they match your setup and that there are no typo’s:

```
1 gcc=$HOME/Tools/gcc-arm-none-eabi-4_8-2014q3
2 PATH=$PATH:$gcc/bin
3 echo "PATH=$PATH:$gcc/bin" >> $HOME/.bash_profile
```

The third line makes sure that the PATH environment also gets extended in all future logins. Be very careful to type “>>” and not “>” or you could wipe out other settings.

Finally, verify that you can indeed launch the compiler from the command line:

```
1 $ arm-none-eabi-gcc -v
2 Using built-in specs.
3 COLLECT_GCC=/arm-none-eabi-gcc
4 [...] 5 gcc version 4.8.4 20140725 (release) [...]  
```

That’s it. Compilation for ARM µC’s should now work on Mac OS X as well.

**Alternative setup** - if you already have Arduino IDE version 1.5.x installed, then there is another way to compile for ARM, since that IDE includes a copy of the ARM gcc toolchain. Exact details will depend on the IDE version, but something like this should do the trick:

---

⁴⁶https://launchpad.net/gcc-arm-embedded
Getting Started

Having said that, it’s easy to use the toolchain from a VM on the Mac, so examples from now on will use Linux. This way, everything works the same on Linux, Windows, and Mac!

**More tools for your tool belt**

Linux is like a human language: at first nothing makes any sense, then you start to see patterns, then you try a few commands, often making mistakes and not getting anywhere initially. From then on, you’re on a fast upward trajectory: learning more commands every day, combining them to get even more done, looking up the finer details, all the way to becoming really proficient and creative with these new-found language skills.

The alternative is the GUI, with lots of flashy visual effects, lots of buttons to press, and never-ending hunting and clicking. But no matter how you look at it, it’s still all about a person giving commands to a machine.

*A GUI is like a dictionary: everything is in there, ya’ just gotta find it…*

A command-line interface is like Meccano - i.e. erector set⁴⁷ for US people. The pieces are all there to build anything, but you have to plan and plot your path or you won’t get anywhere.

With Linux’s command line, we’re going to construct, extend, and streamline - day in day out. No, not just the project we’re working on, but the *environment* we’re working in.

Just as Lisp⁴⁸ has been called the *programmable programming language*, one could call the command line in Linux the *sophisticated developer’s self-automation setup*. If you need to do something more than once, you can usually find a way to create a tool or shortcut which will help you do it faster (including the time to make that tool or shortcut!).

There are several implications with this approach:

- you need to learn commands, plus their features and options - *lots* of ‘em
- you’ll forget them again, so above all you need to learn how to *find* commands
- some things you’ll do so often that they become second nature and reflex-like
- the rabbit hole is real: anything is possible, but watch the time you spend on it
- lastly, a warning regarding all this: there plenty of rope to hang yourself!

But don’t worry, the journey is an exciting one. A long and winding road, as they say...

---

⁴⁸[https://en.wikipedia.org/wiki/Lisp_%28programming_language%29](https://en.wikipedia.org/wiki/Lisp_%28programming_language%29)
Essentials

Some commands in Linux are so basic, they won’t be covered here: cd, ls, cat, rm, grep, etc.

Getting and sharing source code - you’re going to need git right from day one:

```bash
1   sudo apt-get install git
```

Then type “git --help” to get a quick summary, or “man git” to view over a dozen pages of information. To grab a copy of the embello⁴⁹ sources, for example, you could type:

```bash
1   cd $HOME               # go to your home directory
2   mkdir github           # create a subdirectory called "github"
3   cd github              # go into that new subdirectory
4   git clone https://github.com/jeelabs/embello.git
```

This creates a directory called embello will all the latest files downloaded from GitHub.

Everything⁵⁰ in open-source land lives on GitHub nowadays. Or nearly⁵¹ everything.

Git is a revision control⁵² system. Put your (developer’s) life in it, and you’ll never ever lose a change again. All changes, deletions, even complete rewrites, are tracked. Git never forgets.

But the big thing about git is something else: when you edit files in your “clone” made by git, you can still track updates made in the original by typing “git pull”. If the changes don’t interfere with the ones you made, they’ll be merged into your copy, without losing your own changes - even within the same source file. Which, dear reader, is a huge deal.

(if the changes do interfere, your files will contain both versions, very clearly marked)

Warning: git can be incredibly complex. And it always wins. If you try to fight the way it’s intended to be used, you will lose. Getting started with git is easy⁵³, learning it well is not.

Perhaps the best way to get started is to set up an account on GitHub via its help site⁵⁴.

The shell - this is Unix/Linux-speak for the software which presents itself as a command line. You type commands, it interprets them and performs the requested action.

No need to download or install anything, [bash]/bash⁵⁵ is present as default shell in most Linux distributions, as well as in the Terminal on Mac OSX. There are dozens of other shells.

---

⁴⁹https://github.com/jeelabs/embello
⁵⁰https://github.com/explore
⁵¹http://sourceforge.net
⁵²https://en.wikipedia.org/wiki/Revision_control
⁵³http://git-scm.com
⁵⁴https://help.github.com
⁵⁵https://en.wikipedia.org/wiki/Bash_%28Unix_shell%29
It’s important to learn the basics: commands, “pipes”, redirection, variables, scripts. Numerous tutorials\textsuperscript{56} can be found online. Pick one, so you know what it’s all about.

**Make** - the Make\textsuperscript{57} command was installed as part of the setup information given earlier. It should be the second tool after the shell to get quite familiar with, because it can do all the repetitive work for you.

**Bash** is about commands, options, input / output, and about how to combine it all.

**Make** is about rules, containing dependencies and commands, and when to execute them.

There’s a manual\textsuperscript{58} online, but that’s a long read. Here’s an old but useful quick intro\textsuperscript{59}.

### Very useful

**Diff** is a tool to compare two text files. Extremely useful for source code, if you want to figure out differences between an old version and a new one, for example.

With git, there’s also “git diff ...”, which performs a similar task but now comparing versions stored in git. For example, with “git diff HEAD HEAD” you get an overview of everything that’s changed between the current version and the previous one.

**Lua, Python, Ruby** - these scripting languages are great to glue anything to anything. Need to massage a text file, extract specific data, launch a complex set steps, and much, much more? Lots of power under the hood of each of these languages - even a little experimentation goes a long way. Many people do all their programming with these tools.

**All two-letter commands** - at, bg, cd, df, du, fg, if, ls, mv, nc, ps, rm, su, tr, vi, wc, ...

Many of these are classics from the Unix era. Most of them are very useful, and of course there are tons more - wait till you get to the 3-letter and 4-letter commands out there!

### Finding your way around

With so many commands pre-installed, and tens of thousands more not installed by default but available by simply typing “sudo apt-get install <package-name>”, it can be hard to get to grips with it all. These commands will get you going:

**Apropos** - type “apropos blah” and you’ll get a list of all the installed packages with “blah” in their name or brief description (spoiler alert: no matches). The search term is actually a regular expression, so to get a list of all the package names or descriptions starting with “make”, type “apropos ’’make’’” to get over two dozen packages.

Try “apropos sort” for example.

\textsuperscript{56}http://wiki.bash-hackers.org/scripting/tutoriallist
\textsuperscript{57}https://en.wikipedia.org/wiki/Make_%28software%29
\textsuperscript{58}https://www.gnu.org/software/make/manual/
\textsuperscript{59}http://www.linuxdevcenter.com/pub/a/linux/2002/01/31/make_intro.html
Manual pages - type “man ls” to get 3 pages of explanation about the “ls” command. Try the same for “bash”, and you get 60+ pages of a well-written manual to dive into.

Try “man sort” for example.

Note: sometimes you get better information using the “info” command instead of “man”.

Keep in mind that both apropos and man only report what’s currently installed on your system. What if you need to find out about other packages? More commands, of course...

Apt/dpkg - the “apt” package system in Ubuntu comes from Debian Linux. It’s easy to use from the command line. Here are a few examples:

1. `apt-cache search blah`  # list all packages mentioning "blah"
2. `sudo apt-get install blah`  # this will fail, but you get the idea
3. `sudo apt-get remove blah`  # not hard to guess what this does...

Warning: “apt-cache search sort” will return hundreds of matching packages.

Aptitude - this is a more advanced tool built around the apt/dpkg system. When run as “sudo aptitude”, you get a text-mode screen display, with a menu at the top - and some instructions on how to use it. Takes some getting used to, but it’s a quick way to install anything available in Ubuntu and Debian - all dependencies are automatically handled.

Aptitude is a nice way to look around and prepare for installing larger packages and combinations, because you can browse and select what to install and uninstall, and then view what aptitude is about to do (type “g”) before it starts doing it (type “g” again).

To quit aptitude, type “q” (sometimes more than once), and then “y” to confirm it.

Built-in help - last but not least: almost every single command in Linux will understand the “--help” (or sometimes “-h”) option to show a brief summary of the command.

Compare “make --help” with “man make” for example.

What’s this “sudo” thing?

The “sudo” command is a way to briefly gain “superuser” privileges. Anything potentially dangerous or harmful in Linux is protected from accidental use, and requires extra super powers. Things like formatting a disk, changing key system files, configuring hardware.

If you’re not allowed to run command “foo bar”, then you can force your way through by typing “sudo foo bar” instead, but you better be careful and know what you’re doing.

Sudo is a way of getting “administrator permissions” for the duration of that command. Tip: look up “sudo -i” if you need to be superuser for a while, but again be careful.

This brief Linux intro should be enough to get started. Welcome to the command line!
The need for speed

Let’s find out what our little LPC810 is capable of in terms of processing speed...

One of the interesting features of ARM chips, is that they have a phase-locked loop (PLL\(^{60}\)), which is a way to run at a higher clock frequency than the clock electronics itself provides. This is why the LPC8xx series can run at up to 30 Mhz without any external components, even though the internal clock runs at 12 MHz. That’s a 2.5x speedup, and all we need to get there, is to execute a few extra statements after power-up.

To verify this, and to find out how fast the LPC810 will actually run, we can make it toggle an I/O pin and then use a logic analyser or oscilloscope to look at the resulting signal.

The 50-line toggle demo\(^{61}\) in the jeelabs/embello\(^{62}\) area on GitHub uses the following code:

```c
int main () {
  setMaxSpeed();
  ...
  SysTick_Config(12000000/50000);
  while (true)
    LPC_GPIO_PORT->NOT0 = 1<<2;
}

extern "C" void SysTick_Handler () {
  LPC_GPIO_PORT->B0[3] = 1;
  LPC_GPIO_PORT->B0[3] = 0;
  LPC_GPIO_PORT->B0[3] = 1;
  LPC_GPIO_PORT->B0[3] = 0;
}
```

It does two things:

- toggle PIO0_2 in a very tight loop (this is pin 4 on the 8-DIP LPC810 package)
- generate SysTick interrupts at 50 KHz, which toggles PIO0_3 (pin 3) four times

Here is the result, as seen on an oscilloscope - note that we’re running at 30 MHz:

\(^{60}\)https://en.wikipedia.org/wiki/Phase-locked_loop
\(^{62}\)https://github.com/jeelabs/embello
Theyellowtraceistheperiodicinterrupt(pin3),thebluetraceisthetightloop(pin4).Thisscreen
wascapturedbytriggeringontheinterruptpulsingpin3every20µs.

We can make several observations:

- the tight loop takes 0.3 µs per iteration, generating a 1.67 MHz square wave
- the interrupt suspends the tight loop for an additional 1.71 - 0.3 = 1.41 µs
- fast toggling without the loop overhead can generate pulses at 10 MHz

Looking back at the source code, we can deduce that those 4 lines inside SysTick_Handler require
at least 0.2 µs (it’s in fact a bit more, due to some register setup), so the interrupt overhead of the
LPC810 is at most 1.21 µs: i.e. the total interrupt routine entry + exit time.

One more factoid: the LPC810 draws 4.5 mA of current while running this test @ 30 MHz.

But there’s a puzzle hiding in these timings: when running at 30 MHz, each instruction requires 33
ns. How then can four instructions require six clock cycles? There’s no such thing as an instruction
taking 1.5 cycles, it has to be an integral number of clock cycles!

The answer is quite instructive of how an ARM chip such as the LPC810 operates. Keep in mind that
in this example, the µC fetches instructions from flash memory. In the LPC810 µC, flash memory is
probably organised in units of 4 bytes, but on ARM Cortex chips, each instruction uses 2 bytes, so
each flash memory fetch reads two instructions at a time.

On the LPC810, flash memory is configured with 2 “wait cycles”, so reading a word from flash
requires three clock cycles. Which explains it all: 2 instructions per 3 clock cycles!

According to the LPC8xx data sheet, flash memory can be set to a single clock cycle, but only if
the master clock is 20 MHz or less. Let’s ignore those limits for a moment and over-clock the flash
memory access by reducing the wait cycles anyway, even at 30 MHz:
Bingo: now we are toggling pin 3 every 33 ns, i.e. one pin change for each instruction. That darn little 8-DIP chip is generating a 15 MHz square wave output ... in software!

Don’t get too carried away with this, though: normal code will add loop overhead, interrupts occurring once in a while, and complex paths through the program logic. Generating a 1 MHz’ish signal in software is possible, but there will be glitches.

Luckily, the LPC810 also has hardware peripherals which can generate a clean 15 MHz.

Relax and live longer

Now let’s see how little power the LPC810 can consume when in “deep power-down” mode.

All it takes to fully power down is a few lines of C code:

```c
#include "LPC8xx.h"

int main () {
    SCB->SCR |= 1<<2;  // enable SLEEPDEEP mode
    LPC_PMU->PCON = 3; // enter deep power-down mode
    __WFI();         // wait for interrupt, powers down
}
```

The problem here is how to measure the ridiculously tiny current involved. But it’s actually quite simple, thanks to Ohm’s law: if we place a 10 kΩ resistor in series with the μC, then the voltage drop will be proportional to the current, with 1 µA of current leading to a 10 mV voltage drop: enough for a modern multimeter to measure accurately in its lowest range.

The only remaining problem is the initial startup current surge, which is a few milliamps. What we need to do is put the resistor in series, with a voltmeter across it, but short out the resistor while powering up. Once it is in deep sleep, we remove the short and measure.
Result: the LPC810 consumes a mere 0.266 µA (266 nA!) in deep sleep mode.

That’s less than a *millionth* of a Watt of power (Watt = Volts x Amps).

Unfortunately, this setup is not very useful. The *only* way to wake up the µC from this state is to pull its WAKEUP pin low (pin 2). It has no way to wake up on its own anymore.

A considerably more useful configuration is to activate the low-power watchdog timer, and use it to periodically wake up from deep power-down. This was used in the minimal⁶³ and fader⁶⁴ demos in *jeelabs/embello*⁶⁵ to make the LED blink.

There’s another demo on GitHub called relax⁶⁶, which goes to sleep for an entire minute between blinks. It’s really just a small variation of the original minimal demo. With a timeout of a minute, there’s enough time to measure the sleep mode current consumption, with an occasional blink to let us know we haven’t lost the ability to wake up again.

New result: 1.125 µA current consumption with the watchdog running.

Let’s estimate how long the “relax” demo would run on a standard CR2032 coin cell:

- we’ll keep the LED on for 0.5s once every 60s, i.e. less than 1 % duty cycle
- assuming a 4 mA current draw, this translates to 40 µA on average
- the 1.125 µA current consumption can more or less be ignored in this case
- a fresh CR2032 coin cell normally has a capacity of at least 200 mAh
- with 40 µA, that’s 5000 hours of runtime, i.e. over half a year
- it’d be trivial to increase this to a year: simply reduce the ON time to 0.25s

Note that on a coin cell, the LED can’t be placed in series with the µC - we need to modify the code slightly to toggle an I/O pin and connect the LED with a series resistor to it.

Now let’s leave the LED out, and assume this setup has something else useful to do which takes 10 ms once a minute. While running, let’s also assume that the µC + attached circuit draws a whopping 25 mA. Then we can adjust the above estimate as follows:

- 10 ms every 60 is a 1:6000 duty cycle, so 25 mA averages out to only about 4 µA
- add to that the 1.125 µA needed while the circuit is asleep, i.e. most of the time
- if we round off a bit, this circuit will draw 5 µA on average
- again using the same coin cell, we get a 200 mAH / 5 µA = 40,000 hours of run time
- that’s four years on a coin cell, for a circuit which wakes up once per minute

---

⁶⁵https://github.com/jeelabs/embello
Now we’re getting somewhere: think of a wireless sensor node. *Using a coin cell!*

In conclusion: with everything covered in this Getting Started series, you should now be able to program LPC810 chips and make them do all sorts of things. From generating a 15 MHz square wave (or anything else that needs this sort of processing “power”) to running off a coin cell without having to replace it for several years. How’s that for flexibility, eh?

*It’s all just a matter of software plus electronics - and anyone can get into the game.*