Introduction to Building Embedded Devices

William R Cooke

April 27, 2015

Contents

1 Preface **5**

2 Introduction **7**
2.1 What is an Embedded System? 7
2.2 A Different World 7

3 Our Tool Set **9**
3.1 The Hardware 9
3.2 The Software 9

4 Hello, World **11**
4.1 11
4.2 11
4.3 11
4.4 11
4.4.1 11
4.5 11

5 History **13**
5.1 First Steps 15
5.2 The Cold War 15
5.3 Short Skirts and Small Computers 15
5.4 To the Moon 15
5.5 The Calculator 15

6 Programming 101 **17**
6.1 What is a Program? 17
6.2 language 18
6.2.1 Assembly Language and Machine Language 19
6.3 Formatting your Code 22
6.4 Name Calling 22
6.5 Data Types 22
6.6 Operators and Expressions 22

7 Architecture **23**
7.1 Overall . . . 23
7.2 CPU . . . 24
7.3 Memory . . . 24
7.4 I/O . . . 26

8 GPIO **27**

9 Timers **29**
9.1 . . . 29
9.2 PWM . . . 29

10 Interrupts **31**

11 DMA **33**

12 Serial Port **35**

13 Analog Input **37**

14 Analog Output **39**

15 What About USB? **41**

16 Real Time **43**

17 Appendix: Basic Electronics **45**

18 Appendix: Digial Electronics **47**

19 Appendix: Numbers and Stuff **49**
19.1 Decimal . . . 49
19.2 Binary . . . 50
19.2.1 Arithmetic . . . 50
19.3 Negative Numbers . . . 50
19.3.1 One's Complement . . . 51
19.3.2 Two's Complement . . . 51
19.4 Hexadecimal . . . 51
19.5 Floating Point . . . 51

20 What a Character **53**
20.1 ASCII . . . 53
20.2 Unicode . . . 53
20.2.1 UTF8 . . . 53

Chapter 1

Preface

Chapter 2

Introduction

You are about to start on a fantastic journey.

2.1 What is an Embedded System?

The short answer is that an embedded system is a computer that is "embedded" into another device where it performs its task, rather than being the center of attention.

Ask ten embedded system engineers what an embedded system is and you will probably get back at least eleven different answers. Part of the problem is that there are just so many of them. Another part is that the field is constantly changing, not staying still long enough to nail down a solid definition. Still another part is that such a wide variety of devices are used, from very small and simple to very large and powerful systems.

If you look around you right now, you are most likely surrounded by embedded systems. They are everywhere. Here is a short list of everyday things that have embedded processors: cell phones, MP3 players, microwave ovens, washers and dryers, home stereos, cameras, TVs, DVD players,

The PC you are reading this on probably has about a half dozen embedded processors in addition to the main processor. The keyboard, the mouse, the web cam, the hard disk drive, the printer all have them. On the PC motherboard itself there are probably one or more that handle simple tasks

If your car is less than 20 years old it probably has a large number of embedded processors.

toys

2.2 A Different World

Chapter 3

Our Tool Set

3.1 The Hardware

3.2 The Software

Chapter 4

Hello, World

Well, this thing won't program itself. Let's get started. It's traditional to start out programming, whether you are completely new or using a new language or system, with a simple "Hello, World" program that just prints "Hello, World" on the screen. But the embedded world, as we've seen, is a bit different. Typically you don't have a screen. The embedded equivalent of "Hello, World" is to blink an LED.

Wire up an LED and resistor to the LPC board as shown below.

Here is the program:

```
1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5     printf(``Hello World'');
6     return 0;
7  }
```

4.1

4.2

4.3

4.4

4.4.1

4.5

Chapter 5

History

Computer history is a fascinating subject. To me, it is amazing how far we have come in such a short time. In the span of an average lifetime we have gone from the first primitive but huge systems that barely worked, to having an incredible amount of computing power in tiny devices we carry in our pocket that run for weeks on a small battery.

But not only is the history interesting on its own, it is also useful to know how we got where we are today. Knowing how we got to where we are will help us understand why some things are the way they are.

Steam Computers

In the early 1800s the mathematician Charles Babbage faced a problem. Many tables of calculated numbers were needed for a wide array of uses. There were no pocket calculators. If one needed to know the sin or cosine of an angle it was looked up in a table. Many tables existed for different uses: navigation was an important use, as well as scientific research. But the tables were created by hand, by *human computers.* The process was error prone, and many of the critical tables were known to have incorrect numbers. Babbage conceived the idea of building a machine to perform these calculations automatically and remove the error prone humans from the process. He conceived and designed two different machines: first the difference engine, followed by the analytical engine.

Of course, this was well before electronics. These machines were completely mechanical. They were large and difficult to build. Babbage spent the rest of his life and large sums of money trying to complete them, with only partial success.

In recent years, a project to build the difference engine was carried out using processes and tolerances available in Babbage's time. The difference engine worked as expected, proving that it was possible Babbage could have built it. Another project to build an analytical engine is underway.

The analytical engine consisted of the same basic parts as modern computers. Babbage is usually considered to be the inventor of the digital computer. Had he been succesful we would have had computers about a hundred years sooner. The world might well have been quite a bit different.

There are plenty of good history books about computers. I encourage you to find and read some. I won't attempt to recreate one here. Instead, we will take a whirlwind tour of computer history, focusing mostly on the parts that are relevant to embedded systems. Even that will have to be too short to do it justice. Consider that the microprocessor (the computer on a chip) was invented specifically for embedded systems around 1970. There have been a huge number of advances in the last 45 years or so. Since we don't have a lot of time, let's get started.

5.1 First Steps

5.2 The Cold War

5.3 Short Skirts and Small Computers

5.4 To the Moon

5.5 The Calculator

Chapter 6

Programming 101

Now that we have our first tastes of success, it is time to dive into programming full force. Like so many things in life, basic programming isn't difficult, but becoming an expert will be a long road with many twists and turns. I have been programming for over 35 years and still learn new things every day.

Over the last 70 years or so, the true experts have come up with ways to make programming more productive and efficient. They have truly created the field of software engineering: placing programming on a level with other engineering. Engineering often means using standard best practices that are known to work well, rather than trying to reinvent the wheel every time one is needed. Writing software can be quite difficult, and using these best practices is important. Unfortuntely, even professional programmers often ignore what has been learned and do their own thing. I will try to encourage you to use good engineering practices from the start. It will make things a little more painful at first, but you will develop good habits that pay off huge in the long run.

Software engineering is a young field. What's best today may change next year. And if you ask five software engineers their idea of what is best in a given situation, you are likely to get back ten or more answers. One key is consistency. Pick what works for you and stick to it. I'm writing this book so it is full of what I think is best. If you prefer to do things differently, feel free. But make sure you understand the consequences of your choices.

6.1 What is a Program?

My wife sometimes asks me to do things for her that she would normally do, like make a lunch for the kids. But, as she says, I am just a dumb man, so she has to give me very explicit instructions in simple terms. That is the essence of what a program is: a set of very simple and very explicit instructions that a computer can follow. It is just a dumb computer and will do exactly what you tell it, not what you *meant* to tell it.

To write a succesful program, you either find or create an *algorithm* that

will do what you need and create *data structures* to hold the data the algorithm needs. Algorithm is just a fancy word for a simple method, broken down into steps, to achieve some goal. You know how those scientists are: they like to use a lot of fancy words. For instance, a recipe to bake a cake is an algorithm:

```
Set oven to 350 degrees
Pour cake mix into bowl
Add 1 egg to bowl
Add 1 cup milk to bowl
Stir until fully mixed
Pour mix into cake pan
Wait for oven to reach final temperature
Place pan in oven
Wait 20 minutes
Remove cake from oven
Let cool 10 minutes
```

That is the algorithm. The other part we need is data structures. Another fancy term that really just means an organized way to hold our information. Think of a file cabinet. A file cabinet holds our information, usually in some organized fashion, such as alphabetically organized by customer last name. A file cabinet is a data structure.

To make a program from an algorithm and data structures, we have to express them in a way the computer can understand. Unfortunately (or perhaps fortunately, as we shall see) the computer can't understand the language we speak or write. In my case, even people often can't understand me! It also will probably be a bit hard to shove a file cabinet into the tiny little chips we use. So, we need to find a way to create the algorithms and data structures into something we can get the computer to work with.

6.2 language

To write programs we need to give simple and explicit instructions to the computer. We can't use English or any other *natural language*. They are too complex. More importantly, they aren't defined well enough. Have you ever written instructions for someone that you thoght were perfectly clear, only to have that person interpret something different? It happens often with natural languages. The problem is that natural languages are *ambiguous*, with multiple meanings of many words and phrases. As humans, we can usually, but no always, determine what is meant by context. But the computer needs something much more precise. A computer language.

In a way, it is unfortunate that computer languages are called that. They aren't really languages. Although at first glance they may seem like a foreign language you don't understand. They are more accurately called a *notation*. Just as the way we write mathematics is a notation. You wouldn't write an autobiography using only mathematics notation (and most of us wouldn't want

to.) Computer languages are very similar to mathematical notation, and in fact were originally developed from it. But the name computer language has stuck, and we will use it. Just keep in mind there is a big difference.

The First Programmer

As we saw earlier, Charles Babbage was building computers in the early 1800s. Even then, he realized the computer was no good without a program to run. He enlisted the help of a young lady, Augusta Ada King, the Countess of Lovelace. Most people now refer to her simply as Ada Lovelace. She was the daughter of poet Lord Byron, and she was a talented mathematician.

She worked with Babbage to help develop the Analytical Engine, and in so doing wrote out algorithms in a notation meant for the machine. By doing so, she essentially became the first computer programmer nearly 200 years ago! A hundred years before the first working computer was finally built.

In the 1970s the US Department of Defense had a problem: they depended on computers but the programs they used (and had to maintain!) were written in hundreds of different computer languages. They began a program to develop a single language that would suit all uses. When the language was complete, they named it Ada in honor of the first programmer, Ada Lovelace.

There are hundreds, probably thousands of computer languages. Many have come and gone, or changed, or given birth to offspring over the years. Many more are created every year. Each one has some reason for existence, and people often get quite religious over their choice of language.

6.2.1 Assembly Language and Machine Language

Before we go on, we need to talk a little about assembly language. Perhaps you've heard of it. Assembly language is simply the very low level, simple instructions that the computer can actually understand. Any other language must eventually be translated to assembly language. Each type of computer, being built different from other types, has its own assembly language. We will see a little more about that when we talk about computer architecture later. Programming in assembly language isn't really that hard, but it is very tedious and error prone since you must think at a very low level, concerned with every detail. The other langauges we use are often called high-level languages: they allow us to think at a higher level and write our programs that way. Another big drawback to assembly language is that it is unique to each type of computer. If you write a program in assembly language for one type of system, say the LPC1114, then want to move it to a different type, say a PC with an Intel processor, you will have to completely rewrite the program. Most of the high-level languages are standardized, so can be moved between computers much easier.

Machine language is the actual ones and zeros representation of assembly language. It is normally a one for one translation of assembly language, which is human readable (really!) to the bits and bytes the hardware actually requires. It is possible to write a program directly in machine language, but you would have to be a masochist to do that by choice. We almost always use an *assembler* to convert from assembly language to machine language. But since the two languages are basically just different representations of the same thing, we often use the terms interchangeably.

We will have more to say about assembly language later. It is good to understand how it works and is used, but it is possible you may never need to write any assembly language.

The Turing Machine

The Turing Machine is a device originated by the British Mathematician Alan Turing.

For small embedded systems, one language has become somewhat of a standard: *C*. The C language was born with the *Unix* operating system (the grandparent of Linux.) It became very popular with programmers and has been in use for many years. It has lot of good attributes that make it popular, and quite a few drawbacks. Let's take a look at some of the good and bad.

The language is small and simple. It is easy to understand the entire language. Being small and simple also makes it easier to translate it to the assembly language of the small computers we tend to use.

The langauge is very powerful. It is easy to do many of the things, such as low-level input and output, that we need to do in embedded systems. It handles *bit twiddling* ver well: much better than most languages.

It is also easy to customize. The language itself does not have any input/output statements. That was rather unusual at the time C was created. Instead, any type of input and output, or most things that might differ between different types of computer, are handled by *libraries*. A library is simply some other code, often written in the same language, which a programmer can use within his own program without having to write it.

History of C

The C programming language has been around since the early 70s. There aren't very many languages from that time that are still as widely used, and fewer that have changed as little as C, or been as influential on other languages. The story of C is somewhat interesting, and knowing the background will shel a little bit of light on the warts of the language. So put on your leisure suit and let's ride the Soul Train back to about 1971.

But all is not rosy in the land of C. C is a teenager that never really grew up. It was created in the adolescent years of software engineering: we knew enough about writing good software just to be dangerous. Then it became so popular

so quick it wasn't possible to fix it as we learned more. Everyone who had been using it demanded backward compatibility so their old programs would still work. Besides, one of the reasons C became (and remains) so popular is that it lets you do just about anything you want, no matter how dangerous it is.

Let's go over some real examples. Throughout this book I will point out more areas where C can get you if you aren't careful. When it gets you, don't feel too bad: C gets even the experienced programmers fairly often. Just like a spouse, we learn to live with its faults rather than give it up.

We will soon be learning about *variables* and *data types*. When we write a program that handles data we have to tell the computer what kind of data we are using. The operations we can use on the data and what the effects of those operations are depend on the data type. One of the things computer scientists have learned well in recent years is that *strict typing* is important to writing correct programs that do what we intend. Strict typing means that once you declare some data to be one certain type, its type can't be changed behind your back. You must explicitly tell the computer if you want to use it a different way. That way, when the type does get changed it is because you made a concious decision to do so, and, one hopes, are aware of the consequences.

C is not a strictly typed language. It may appear to be at first, but then you find out it will go around changing types willy nilly behind your back. This causes a *lot* of problems. Imagine your pocket calculator changing the rules of how it calculates after you enter the numbers. We will have much more to say about data types and how they work in C.

Another area where C can bite you like a rattlesnake you just stepped on is with *operators* and *expressions*. Operators are those symbols that cause some action to take place, like the arithmetic operators (+, -, etc.) Expressions are the constructions we build out of operators and *operands*, the data that the operators operate on (that's a mouthful!) Here's an example: $3 - (a + 1)$ is an expression with "-" and "+" as operators and "3", "a", and "1" as the operands. C has a lot of powerful operatos, but sometimes they are confusing and sometimes do things we don't expect. And because of the rules of C and the C attitude of let the programmer have enough rope to shoot himself in the foot (I made that up) it often leads to programs that will "work" but not as we intended. Mix that in with the flexible data types and *bad things* happen!

But it isn't all bad. C is a great language. It is very easy to do many things with C that are very difficult in other languages. C puts the power, and responsibility, in the programmer's hands. Learning to watch out for the gotchas is part of learning to program with any language. I will try to point out as many as I can so you know what to watch for. And some things have gotten better over the years. The language continues to improve, and C compilers have gotten much better at warning us about dangerous things we do: it behooves you to pay attention to warnings from your compiler!

What is a Compiler?

It may not seem like it when you are first learning, but the C language is written to be easily understood (read) by humans. But to the computer it is just gibberish. Since the computer can only work with machine language directly, the C code has to be somehow translated into machine language before the program can be used. That translation is the job of a compiler.

The compiler reads through your program, breaking it down into small and simple steps. It then writes out an equivalent machine or assembly language bit of code that does what each portion of the C program intends. All the pieces are stitched together and written out as a program. If the compiler outputs assembly language, it must go through another step called an *assembler* before being used. There may also be other tools, such as a *linker* that combines different pieces into one single program. But often we refer to the entire set of tools as the compiler.

There is another way to translate a program written in some language other than machine language. Instead of converting the entire program at once before running it, a program can read through each piece of the program, execute a bit of machine language that matches that piece, and then get the next piece until finished. This type of converter is called an *interpreter*. It is common with many languages you may have heard of, like BASIC, Python, Perl, and javascript. We will not be discussing interpreters here.

6.3 Formatting your Code

6.4 Name Calling

6.5 Data Types

6.6 Operators and Expressions

Chapter 7

Architecture

To build embedded systems, it's important to know a bit about what pieces make up the hardware, how they are put together, and how they work. This is called the *architecture* of the system. This chapter won't turn you into a computer architect, but it will give you the basics to understand how the parts work together to build a functioning system.

Origin of Computer Architecture

In the 1960s software engineering was a very young but rapidly growing field. IBM had a grand plan for a new line of compatible computers, offering models for all uses. This was the Model 360 line. The systems needed a new operating system.

Computers are kind of like people: they are all similar on the surface, but the details vary between models. A lot of architectures have been tried over the years, but almost all computers now use the same basic archtecture. There are two variations of that architecture widely used in small microcontrollers, and we will cover that later.

7.1 Overall

A computer system, large or small, needs three main parts to be of any use. They are the CPU, the memory, and the Input/Output (I/O.) The details, size, and complexity of those three parts differentiates a small embedded controller from a large supercomputer. But in general they are all the same.

The three parts work together to form a working computer. The CPU does the actual processing, moving things(data) around and working on it in various ways. If you add two numbers or sort a list, it is the CPU that controls that.

The memory is where all the programs and data are held so that the CPU has access to them. The CPU will normally have a small amount of memory where it can store data it is currently working on, but the main memory (sometimes

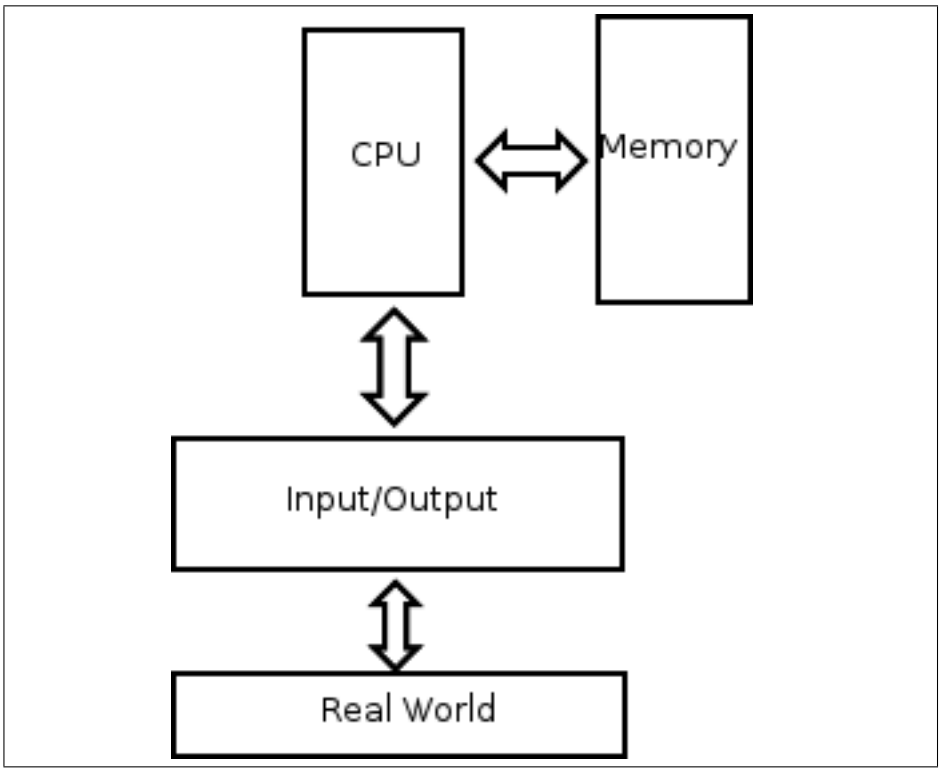


Figure 7.1: General Computer Architecture

referred to as core or core memory) holds the main working set of data as well as the program. One common mistake is to confuse storage memory, such as a hard disk drive, with core memory. They are two very different types of memory. Embedded systems will quite often have no storage memory at all.

Input gets information from outside the computer, and output sends information to the outside world. Input may come from a switch or a sensor or perhaps some type of disk drive or other device. Output may go to light an LED, torun on a device, send data to a display of some sort, or again perhaps a disk drive or other device.

Let's have a look at each one.

7.2 CPU

The

7.3 Memory

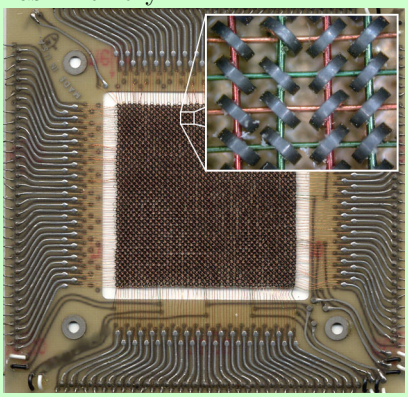
Core Dump

Today, the memory for our computers is usually made from transistors on an integrated circuit. But that was not always the case. In the early days of computers, various methods of storing data within the computer were used, including things like mercury delay lines and cathode ray tubes. But from about the mid 1950s until the mid to late 1970s, the primary storage technology was magnetic ferrite cores.

Ferrite is an iron-based material that is easily magnetized. It was formed into doughnut shapes so the magnetic field would form a loop. Thousands of these cores were strung together, one for each bit of memory, in a two dimensional array. Typically four wires were run through each core: an X select line, a Y select line, a sense line, and an inhibit line. The select lines allowed a single core in the array to be selected and written or read. The sense line provided the data output for reading. The inhibit line allowed a core that otherwise would be written from being selected.

Although the state of the art in making core memory progressed steadily during its twenty year reign, it was still much larger and slower than the then-new semiconductor memory. By the end of the 1970s it was mostly replaced by faster, smaller, cheaper semiconductor memory. Core memory did have one very attractive property: it held its data when the power was off.

In recent years, the technology has made somewhat of a comeback. Modern memory systems are building the rough equivalent of core memory on integrated circuits, called FerroElectric RAM(FeRAM.) FeRAM works like random access memory, but will hold its data without power like flash memory.



Ferrite core memory was so pervasive for so many years that we still refer to RAM as core memory. On many Unix and Linux systems, a crashed program will leave a copy of the crashed program as a file on disk for debugging. The name of the file is core. The term core dump is still in widespread use. Other uses are also common. Keep your eyes open for it when reading computer literature.

7.4 I/O

Chapter 8

GPIO

Chapter 9

Timers

9.1

9.2 PWM

Chapter 10

Interrupts

Chapter 11

DMA

Chapter 12

Serial Port

Chapter 13

Analog Input

Chapter 14

Analog Output

Chapter 15

What About USB?

Chapter 16

Real Time

Chapter 17

Appendix: Basic Electronics

Chapter 18

Appendix: Digial Electronics

Chapter 19

Appendix: Numbers and Stuff

To a computer, everything is a number.

There are a lot of ways to count numbers. The way we use most is called the "Decimal" number system. The prefix "deci" means ten. The decimal number system uses ten digits: 0 through 9 (count them!) It's generally assumed that we use ten digits because, well, we have ten digits: Our fingers. Picture a caveman telling his buddy how many wooly mammoths he saw when out gathering berries. He could hold up the same number of fingers as the number of mammoths he saw. But what if there were more than ten?

19.1 Decimal

So it is that we came to use the decimal system. Over the last few thousand years, we've gotten rather used to it. We prefer our numbers all be handled with decimal notation. So let's take a look at how it is constructed.

Suppose our caveman friend had not seen any mammoths. How would he indicate that? Interestingly, it was a very long time before the concept of a number for zero came about. You might find the http://en.wikipedia.org wikipedia article rather interesting. However it came to be, the zero was added to the digits. The "deci" in decimal means ten, and we have ten digits including zero. The highest is 9. What if we need to express ten, or eleven? What we do is add another digit, in the "tens" place.

10 11 12 Since decimal is based on ten digits, or a "base ten" system, each digit farther left is ten times as big as the one to its right. The right most digit has its own value. In the three examples above, the rightmost digits are 0, 1, and 2. But each has a digit to its left, which is multiplied by ten. So the numbers represented are one times ten plus zero (ten), one times ten plus one (eleven), and one times ten plus two (twelve.) With more digits we just extend the pattern: 257 2 times 100 = 200 5 times 10 = 50 7 times 1 = 7

We use the decimal system so much we tend to not even think about it. But there is nothing special about base ten. If our cavemen friends had only 8 fingers, we would likely use a base 8 number system. As it turns out, fourty years ago that was rather common. But we'll get back to that.

A couple of key points from all this. First, we can use ANY number base greater than 1. Second, the digits in that number base will range from 0 to one less than the base (ie 7 for a base 8 system.)

19.2 Binary

We came here to talk about computers, not cavemen and woolly mammoths. As we learned in the appendix on digital electronics, which is what computers are built out of, they have two states: high and low, or true and false. Or, how about 1 and 0? We can use digital electronics to make a base two number system!

If we use the low state to represent 0, and the high state to represent 1, we now have a base 2, or binary, number system. We could use low for 1 and high for 0 just as well, but the way we are doing it most common. From now on we will just call them 1 (high) and 0 (low.)

The binary number system works just like the decimal system, except that each digit can only be 0 or 1 and each column of digits is only two times the one to its right. It seems a bit odd at first and takes a lot of getting used to, but binary isn't hard. You will get more comfortable with it the more you use it. And working with embedded systems, you will use it quite a lot.

A single digit in binary, called a "bit" for Binary digIT, can be either a 0 or a 1. Two numbers. If we add digits (bits) we increase how many numbers we can represent. In decimal, each digit increases how many numbers we can represent by ten, the base of decimal. Same in binary. Each digit increases how many numbers we can represent by two, the base. That's true of all number systems. Instead of the one's place, ten's place, hundred's place, etc, we have the one's place, the two's place, the four's place, and so on. Let's see an example. Here is a list of all possible three digit (bit) numbers:

000 0 001 1 010 2 011 3 100 4 101 5 110 6 111 7

19.2.1 Arithmetic

How does arithmetic work with binary numbers? The same as with decimal numbers, only different.

I suppose that wasn't a very satisfying answer. So let's take a look. We will start with addition.

19.3 Negative Numbers

That covers the basics of binary numbers. With N bits we can represent numbers from 0 to $2^N - 1$. What about negative numbers? There are a couple of ways

to handle negative numbers, but most of the computing world has settled on a system called Two's Complement." We will get to that shortly. To better understand two's complement, we will first take a look at one's complement. It is rarely used today, but it is a bit easier to understand and leads us into two's complement.

19.3.1 One's Complement

If we have N bits (say 16) we saw that it can represent 2^N different numbers (65536.) If we use only zero and positive numbers, then they can range from 0 to $2^N - 1$ (65535.) If we want to also represent negative numbers, we must divide the range about in half, using half for positive and half for negative. How should we do that?

One's Complement on the Moon

One's complement was widely used in the past for various reasons. One reason was the hardware was simpler, making it popular when hardware was large and expensive. Another reason was the fact it is symmetrical: it has the same range for positive and negative numbers. One place it was used was the Apollo Guidance Computer (AGC) used to control the Apollo spacecraft that went to the moon. There were two AGCs on each mission: one in the Command Module and one in the Lunar Module that actually landed on the surface.

19.3.2 Two's Complement

19.4 Hexadecimal

Binary is great. It is simple and is perfectly matched for the hardware our computers are built from. But it can be a bit inconvenient for us poor humans. It takes a lot more bits to represent a number than it takes decimal digits for the same number: to represent a billion takes 30 bits! Remembering a long string of bits even long enough to type it in can be difficult. Usually, it is easier to work with larger number bases (like ten.) So the smart people who developed computers in the early days came up with a solution: break a binary number up into groups of bits and use those groups as a different number base.

19.5 Floating Point

Sometime in elementary school you learned about fractions. You may still remember the trauma. Or maybe you've blocked it out. But in day to day life fractions are important. Very often, although probably not as often as you think, it is important to have our computers deal with fractions. Here we will describe

how computers (normally) handle fractions. The method is called floating point and also handles very large and very small values.

Chapter 20

What a Character

20.1 ASCII

20.2 Unicode

20.2.1 UTF8

Ken Thompson

Appendix: Secrets of the Magic How do computers actually perform their magic?

Appendix: C Language

Appendix: What about Arduino?

Glossary