PART I

The Basics

This first section of the book covers the material you'll need to know for most AVR projects. These chapters build directly on one another, and you're probably going to want to work through them in order. Chapter 1 starts out with an overview of the chip and what it can do for you, then we move on to doing it.

The first task is to learn how to write and compile code for the AVR, and then get that code written into the chip's flash program memory. By the end of Chapter 2, you'll have an LED blinking back at you from your breadboard. Chapter 3 introduces the topic of digital output in general, and we'll build a POV illusion gadget that you can program yourself. Chapter 4 is an introduction to bit-level manipulations using bitwise logic functions. Though not a particularly sexy chapter, it's fundamentally important.

Chapter 5 connects your AVR to the outside world: in particular, your desktop computer. Bridging the computer world and the real world is where microcontrollers excel, and the serial port is the easiest way to do so. To show off a little, we'll make an organ that you can play from your desktop's keyboard.

Chapter 6 introduces you to the world of button pressing. We'll make a standalone AVR music box where you control the tempo and length of the notes that are preprogrammed into the chip and leverage the serial connection from the previous chapter to make a dedicated web page–launching button.

Chapter 7 brings the outside world of analog voltages into your AVR, by introducing the built-in analog-to-digital converter (ADC) hardware. Knowing how to use the ADC opens up the world of sensors. We'll build a light meter, expand on this to build a knob-controllable night light, and finally combine the ADC with serial output and your desktop to implement a simple and slow, but still incredibly useful, oscilloscope.

Introduction 1

The first question to ponder is what, exactly, is a microcontroller? Clearly it's a chunk of silicon, but what's inside of it?

What Is a Microcontroller? The Big Picture

Rhetorical questions aside, it's well worth getting the big-picture overview before we dive headfirst into flipping bits, flashing program memory, and beyond.

A Computer on a Chip...

Microcontrollers are often defined as being complete computers on a single chip, and this is certainly true.

At their core, microcontrollers have a processor that is similar to the CPU on your computer. The processor reads instructions from a memory space (in flash memory rather than on a hard drive), sends math off to an arithmetic logic unit (instead of a math coprocessor), and stores variables in RAM while your program is running.

Many of the chips have dedicated serial hardware that enables them to communicate to the outside world. For instance, you'll be able to send and receive data from your desktop computer in Chapter 5. OK, it's not gigabit Ethernet, but your microcontroller won't have to live in isolation.

Like any computer, you have the option of programming the microcontroller using a variety of languages. Here we use C, and if you're a software type, the code examples you see in this book will be an easy read. It'll contain things like `for` loops and assigning variables. If you're used to the design-code-compile-run-debug cycle, or you've got your favorite IDE, you'll feel at home with the software side of things.

So on one hand, microcontrollers are just tiny little computers on a chip.

...But a Very Small Computer

On the other hand, the AVR microcontrollers are *tiny little* computers on a chip, and their small scale makes development for microcontrollers substantially different from development for "normal" computers.

One thing to notice is that the chips in the AVR product line, from ATtiny15 to ATmega328, include the flash program memory space in kilobytes in the chip's name. Yeah, you read that right: we're talking about 1 KB to 32 KB of room for your code. Because of this limited program memory space, the scope of your program running on a single chip is necessarily smaller than, for example, that Java enterprise banking system you work on in your day job.

Microcontrollers have limited RAM as well. The ATmega168 chips that we'll be focusing on here have a nice, round 1 KB. Although it's entirely possible to interface with external RAM to get around this limitation, most of the time, the limited working memory is just something you'll have to live with. On the other hand, 1,024 bytes isn't that limiting most of the time. (How many things do *you* need 1,024 of?) The typical microcontroller application takes an input data stream, processes it relatively quickly, and shuttles it along as soon as possible with comparatively little buffering.

And while we're talking specs, the CPU core clocks of the AVR microprocessors run from 1 to 20 megahertz (when used with an external crystal), rather than the handful of gigahertz you're probably used to. Even with the AVR's RISC design, which gets close to one instruction per cycle, the raw processing speed of a microcontroller doesn't hold a candle to a modern PC. (On the other hand, you'll be surprised how much you can do with *a few million* operations per second.)

Finally, the AVR family of microcontrollers have 8-bit CPUs without a floating-point math coprocessor inside. This means that most of the math and computation you do will involve 8-bit or 16-bit numbers. You *can* use 32-bit integers, but higher precision comes with a slight speed penalty. There is a floating-point math library for the AVRs, but it eats up a large chunk of program memory, so you'll often end up redesigning your software to use integers cleverly. (On the other hand, when you have memory sitting unused, go for it if it helps make your life easier.)

Because the computer that's inside the microcontrollers is truly *micro*, some more of the niceties that you're probably used to on your PC aren't present. For instance, you'll find no built-in video, sound, keyboard, mouse, or hard drives. There's no operating system, which means that there's no built-in provision for multitasking. In Part II, I'll show you how the built-in hardware interrupt, clock, and timer peripherals help you get around this limitation.

On the other hand, microcontrollers have a range of hardware peripherals built in that make many of the common jobs much easier. For instance, the built-in hardware serial interface means you don't have to write serial drivers, but merely put your byte in the right place and wait for it to get transmitted. Built-in pulse-width modulation hardware allows you to just write a byte in memory and then the AVR will toggle a voltage output accordingly with fractional microsecond precision.

What Can Microcontrollers Do?

Consumer examples of microcontrollers include the brains behind your microwave oven that detect your fingers pressing on the digit buttons, turn that input into a series of programmed on-times, and display it all on a screen for you to read. The microcontroller in your universal remote control translates your key presses into a precise series of pulses for an infrared LED that tells the microcontroller inside your television to change the channel or increase the volume.

On the other end of the cost spectrum, microcontrollers also run braking and acceleration code in streetcars in Norway and provide part of the brains for satellites.

Hacker projects that use microcontrollers basically span everything that's cool these days, from the RepRap motor-control and planning electronics, to quadcopter inertial management units, to high-altitude balloon data-loggers; Twittering toilets and small-scale robotics; controls for MAME cabinets and disk-drive emulators for C64s. If you're reading this book, you've probably got a couple applications in mind already; and if you don't, it'll only take one look at Hack-a-day or the Make blog to get your creative juices flowing.

(If you want to know *why* you'd ever want to get your toilet to tweet each time you flush, I'm afraid I can't help you. I'm just hear to show you *how*.)

Hardware: The Big Picture

So a microcontroller is a self-contained, but very limited computer—halfway between a *computer* and a *component*. I've been talking a lot about the computer side. What about the AVR chips as components? Where can you hook stuff up? And how exactly do they do all that they do? Figure 1-1 lays out all of the chip's pins along with the mnemonics that describe their main functions.

If you're coming at this from no background, you're probably wondering how a microcontroller does all this marvellous stuff. The short answer is by reading voltages applied to its various pins or by setting up output voltages to these very same pins. Blinking an LED is an obvious example—when the output voltage is high, the LED lights up, and when the voltage is low, it doesn't. More complicated examples include the serial ports that communicate numbers by encoding them in binary, with high voltage standing in for a 1 and low voltage standing in for a 0, and changing the voltage on the pins over time to convey arbitrary messages.