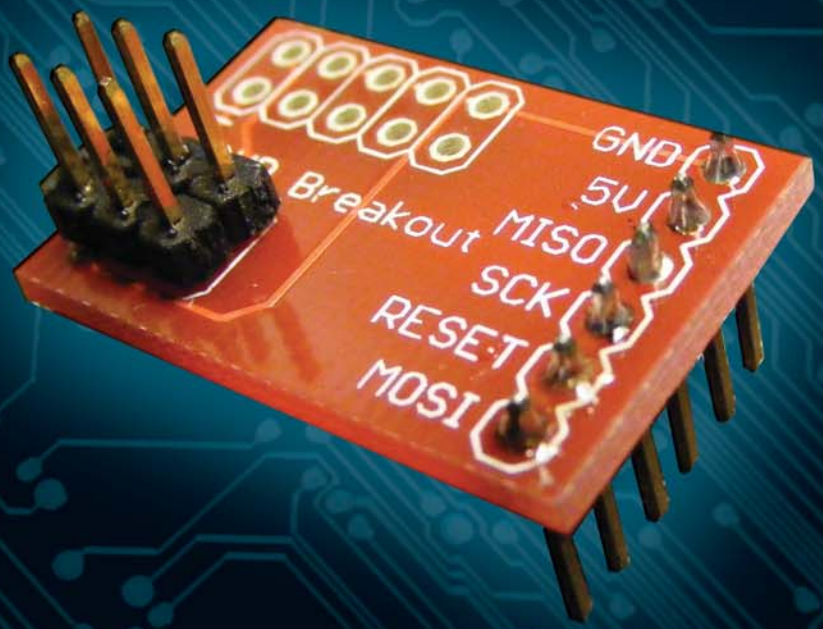
TECHNOLOGY IN ACTION™

Practical AVR Microcontrollers

Games, Gadgets, and Home Automation with the Microcontroller Used in the Arduino

FLASH, SENSE, SPIN, AND ROLL WITH THE AVR MICROCONTROLLER



Alan Trevennor

Contents at a Glance

Foreword xv

About the Author xvii

About the Technical Reviewer xix

Acknowledgments xxi

Introduction xxiii

Part 1: The Basics 1

Chapter 1: A Brief History of Microcontrollers 3

Chapter 2: Building Our AVR Test Bed 13

Chapter 3: Arduino and the Naked AVR 49

Chapter 4: Moving On! 85

Chapter 5: Smarten Up! 155

Chapter 6: Digitally Speaking 167

Part 2: The Projects 189

Chapter 7: Introduction to the Projects Section 191

Chapter 8: Project 1: Good Evening, Mr. Bond: Your Secret Panel 195

Chapter 9: Project 2: Crazy Beams—Exercise Your Pet! 213

Chapter 10: Project 3: WordDune 229

Chapter 11: Project 4: The Lighting Waterfall 249

Chapter 12: Moving to Mesmerize 277

Chapter 13: Smart Home Enablers 305

Appendix A: Common Components 333
Appendix B: A Digital Electronics Primer 347
Appendix C: Breadboards 359
Appendix D: Serial Communications 365
Index 377

PART 1

The Basics

CHAPTER 1

A Brief History of Microcontrollers

Although it's not essential that you understand how microcontrollers developed to the point where they are today, it's an interesting story, which can help you understand where an AVR microcontroller fits into the overall hierarchy of information technology (IT) and electronics products. More important, by having such an understanding you can make better choices and decisions about when and where to use a microcontroller, in preference to other alternatives.

If you open up a CD player or a VCR from the 1980s (perhaps you have one in the attic, or in your garage, I know I do!) you will find that they are absolutely stuffed with circuit boards, and that each circuit board is densely populated with integrated circuits (chips) and components that made the thing work.

By contrast, open up a DVD player made in the last few years and you are likely to find quite a lot of empty space, and just one quite small circuit board that contains perhaps two or three quite large chips and a handful of other components. Yet, it's probable that the modern device offers far better quality and robustness. It will certainly offer massively more product features and options than its 1980s predecessor.

This transformation is due to two main factors:

- The increasing miniaturization of electronics and components, which has enabled more and more circuitry to be put onto single chips, reducing the chip count needed for any given function.

  The transistors in the first family of logic chips (launched in the early 1970s) each measured about 10 microns[1] across. Just to give you some idea of scale, a human hair averages about 100 microns in width. At the time of writing this, in 2012, the size of transistors on current generation chips can be as small as 22 nanometers. That's just 22 billionths of a meter! That gives you some idea of the pace of miniaturization that has gone on inside integrated circuits since the 1970s.

- The progressive transition from implementing device functions in hardware to implementing them in software running on microcontrollers.

Let's start with a quick timeline before getting into the whys and wherefores of microcontrollers and AVR.

[1] A micron (now more often called a micro-meter) is one millionth part of a meter, or about 0.0000394 inches.

A Microcontroller Timeline

Until the mid-1980s most electronic products were still built using extremely intricate and clever combinatory logic[2] circuits, implemented with an awful lot of chips! Starting in the early 1980s, a minority of manufacturers started to build in microprocessors to their products in order to reduce chip count, which brought down manufacturing costs and thus reduced end-user prices.

The earliest 8-bit microprocessors such as the Intel 8080 or the Zilog Z80 first appeared toward the late 1970s and were a significant advance on what had gone before. Engineers and designers soon realized that once you put a microprocessor into a device, you could not only make it do much more, but you could also update it much more cheaply if defects or flaws in the original design came to light. Many product defects could now be addressed by using semiskilled labor to plug in a replacement firmware ROM (read-only memory) (this was in the days before programmable flash memory) rather than having to use skilled labor to expensively rework or replace thousands of complete circuit boards. As the 1980s wore on, more and more products had a microprocessor at their core.

Even though microprocessors were a huge improvement on what they replaced, they weren't a complete magic bullet for bringing down costs and complexity of product design. The problem was that, to make a microprocessor do anything useful, it had to be surrounded by a large number of additional chips for input output (I/O) and it usually needed other support chips too–such as real-time clock chips and address decoders.

By the 1990s, improved silicon processing and chip manufacturing techniques resulted in the ability to put ever more circuitry on one chip. One of the ways this was used was to augment the microprocessor chip with additional functions and features that had previously been implemented by separate external chips. To differentiate these new super-micro chips from their simpler forebears, these came to be called microcontrollers. Some examples of functions that moved from being external chips to being part of the microcontroller are

- Serial ports to enable the subsystem to talk to a desktop computer or other RS232 port-equipped devices.
- Timers to enable the microcontroller to have an accurate time reference on chip and to carry out events at accurate preset intervals. These timers also enabled microcontrollers to generate music and sounds, since interval accuracy could be assured.
- Serial digital channels to enable microcontrollers to chat with one another, over just two linking wires.
- Analog to digital convertors allowing a microcontroller system to sense analog signals and store or process them as digital data.
- Digital to analog convertors that allowed microcontrollers to interface with external devices like motors that need a continuously variable voltage.
- Input ports for sensing on/off states of things in the outside world.
- Output ports for switching on/off things in the outside world.

[2]Combinatory logic circuits use individual chips in combination to provide each function. For example, in a microcontroller project that controls ten motors for an industrial process, we would use a software counter for each motor to count how many times it had turned. In the combinatory logic implementation of the same thing, there would have to be an actual counter chip for each motor sensor. So, in a microcontroller approach to this function, a whole board full of counter chips could be replaced by perhaps 20 lines of software. This would reduce cost, power consumption, heat generation, and size. Furthermore, if the design were updated, in the combinatory approach, rewiring and very likely redesigning of the board would be needed. In the microcontroller approach a simple software update would attain the same result.

Once microcontrollers started to be designed into consumer goods and control systems during the 1990s, the already impressive electronic miniaturizations of the previous two decades took another big jump, in terms of both size reductions and the ability to sport more options and features than ever before.

By the first decade of the 2000s, nobody would seriously consider designing anything other than the very simplest consumer electrical device without the use of some kind of microcontroller. They are everywhere; they get more capable and more complex as time goes on. As a technical person, unless you understand microcontrollers at some level, you will be at a considerable disadvantage compared to those who do.

Why Microcontrollers?

The ubiquity of microcontrollers is the main reason you should know something about them. However, it's also very satisfying to use and design with microcontrollers. You can get things running very quickly that previously would have taken very much longer to complete. You can also have a considerable amount of fun in the process, and what's life without a little fun? The AVR family of microcontrollers is a wide ranging and cost-effective way to implement your projects. The ever-growing popularity of AVRs means that there is an enormous and very active online support community to help you out if you get stuck. It also means that there is a massive amount of free AVR software available that makes your projects far easier and faster to complete.

Why Should You Learn About Microcontrollers?

To answer this question simply: Because they are fun! The fascination of what you can do and what you can make with them is never ending. It has been rightly said that the computer is the ultimate tinker toy: you can use it an infinite number of ways to enhance your job, your learning, your hobby, or your social life. The value of microcontrollers is that they allow you to extend the benefits of computing into the real world.

You probably already own quite a few microcontrollers without knowing it. They are embedded in most of the appliances and devices around you.[3] Anyone who wants a real understanding of how modern products work–from cars to mobile phones to toys–needs to have at least a rudimentary understanding of microcontrollers.

What Can You Do with a Microcontroller?

Okay, well here's the heart of the matter. Desktop computers (PCs and Macs) are excellent–they are truly a wonder of the age. In concert with the Internet, the desktop computer you buy from the store can do just about anything you want with digital information.

The desktop computer is essentially a resource-rich computer for reliably processing and storing information in a networked world. It can do many things at once (e.g., check your e-mails, and do virus checking while you are browsing the news online) because it is running a complex operating system that is capable of multitasking on a scale that is *truly* (to use that overused word accurately for a change) awesome[4]—what we see on the screen is only the tip of the iceberg of the work going on inside the machine.

Having said that, a modern desktop computer has a central processor running at something around an unimaginable 3 billion cycles per second, and many processors have a multicore architecture, meaning that they are capable of executing two, or even four, sets of instructions streams at this speed, simultaneously! Your modern desktop machine is likely to have a hard drive inside offering at the very least 500 gigabytes (that's

---

[3] Thus, you will often find microcontrollers referred to as embedded computers, and the software they run is often generically called embedded software.

[4] Did you know, for example, that both Linux and Windows update their time clock and the statistics for all the running tasks and certain other internal data at least 100 times . . . per second?

500,000,000,000 bytes) of storage, and it probably has a RAM (random-access memory) of 2 gigabytes or more. So, in computing terms, it is a resource-rich machine. Alan Turing himself could not have wished for more.

Want to edit a video? No problem. Find out who your great, great, great, great, great grandfather was? Yes, can do! Want to send an e-mail to the other side of the world in just a few minutes? Yes of course, tell you what, let's make it one minute! Want to index your family photo collection? Shazam! It's done. Want to play hi-def movies? Let's do it!

The reason desktop machines have evolved into these monstrous–and comparatively expensive–computers is that they are general-purpose machines. The capabilities of even a low-end desktop machine are now so high that you could use it for any of the previously mentioned information management tasks without any problem, but in its default state it's actually quite poor at interfacing with the real world.

But: Do you want to be notified when your freezer fails? Want to intelligently control the speed of a fast running motor? Want to implement a control system for deriving electricity from the rainwater running down from your roof gutters? Hmmm, no, that's a bit trickier–your out-of-the-box desktop computer can't do that without adding on quite a lot of extras.

The dirty little secret about modern desktop machines is that most of them barely ever break into a canter. Graphical compilation (compressing and encoding videos, 3D game compiling, etc.) are among the most demanding tasks that a desktop PC can be set to do, and comparatively few of them are ever used for these things. Playing full-screen video is probably the most challenging thing that most desktops are asked to do, and almost any modern machine can do that and still have processing power to spare. So, it's fairly clear that much more mundane computing tasks really don't need that huge amount of processing power.

The desktop computer in your house (statistically you are likely to have more than one, by the way) is the de facto "home hub" for IT in your house. But, as we saw, there are "real world" tasks that are actually beyond this general-purpose behemoth. It's very good at processing information, but in its default state it's useless at interfacing with other devices in the real world. Enter microcontrollers.

On paper, a microcontroller looks like a very poor relation to that desktop machine of yours. It will have a processor that runs at only small a fraction (perhaps a 300th) of the one on your desktop, and it's unlikely to have more than a fraction of the memory capacity. It doesn't have inbuilt support for interfacing to hard drives and you can't just plug it in to the Internet.

On the other hand, you can get a midrange microcontroller chip for the price of a Skinny Latte and you can build a complete microcontroller system for rather less than the price of a business lunch. The microcontroller will have inputs and outputs suitable for use with real-world devices, and with a little effort, it can talk to your desktop's serial or USB ports.

So, here's how it pans out:

- Use your desktop computer for general-purpose big-world stuff, Internet, e-mail, downloading and playing video, word processing, printing stuff, instant messaging, social networking, building photo libraries, editing photos, and . . . you get the picture. The standard USB and serial ports on your desktop machine can also be used to talk to external microcontroller systems, to allow it access to real-world data, and to be the brain that controls real-world stuff like heaters and motors and lights.

- Use a microcontroller as a single-purpose stand-alone computer that performs a particular small-world task, like controlling some lights, measuring the temperature, and passing the results on to your desktop machine. Microcontroller systems can take orders from your desktop machine, "Switch that heater on, Put that light out." But a microcontroller system doesn't have to be connected to a desktop machine; it can happily work as a complete single-purpose, simple, but still intelligent, stand-alone computer.

In summary: The desktop computer is built, sold, and operated as a general-purpose computer. It is intended to find and manipulate any digital information, in any way you want. A microcontroller is a much simpler, scaled-down computer that is far cheaper than a desktop machine but is suitable to be programmed to do just one task very well.

A microcontroller system can be the interface to real-world devices (freezers, temperature sensors, fans, heaters, lights) for a desktop machine,[5] or it can just be used built into a stand-alone system. Such applications are often called Smart Appliances because, thanks to the fact that they are software controlled, they can allow an appliance to exhibit a limited range of adaptive behaviors. Modern cars usually feature several microcontrollers embedded in their various systems.

In this book we look at a variety of microcontroller-based projects. Some are interfacing projects that benefit from connection to a desktop computer, while others are stand-alone, independent systems.

Why AVR?

There are, in fact, a large number of different (and software-incompatible) microcontroller families on the market, of which AVR is one. Probably the market leader in this field is the PIC (Programmable Intelligent Computer[6]). PIC was gradually developed as an upgrade to a previous generation of microprocessors by General Instruments in the early 1980s. The product line was inherited by Microchip Technologies–the commercial successors to GI–which by the mid-1990s had added additional refinements such as on-chip user-writable program memory.

PIC chips offer excellent value and there is a lot of support software and hardware available for them–they deserve their success. However, PIC chips are not especially clock efficient. That is, a PIC chip driven at a certain clock rate will not achieve as much useful work as other microcontrollers, due to certain inefficiencies inherent in the PIC architecture.[7] The PIC was not originally designed around a RISC methodology (see the following section)–whereas the AVR family of microcontrollers has a more recent design and is RISC to the core.[8] To answer the preceding question, "Why AVR?,"I am a fan of AVR because it is fast, well designed, easy to use, well supported, and cheap to buy.

Some History: The Computer Industry Takes a RISC

To understand what RISC (Reduced Instruction Set Computer) really means, and how today's computing benefits from it, we need to look briefly at how computer processors developed in the 1970s and 1980s and how computers were used back then.

Early electronic computers offered the programmer very few machine instructions. Adding or subtracting two numbers, moving a value from one storage register to another, loading and saving registers to main memory–and that was about it. The people who programmed these pioneering machines in the late 1940s and into the 1950s were working in pure machine code: They had to learn the binary values for each instruction that the machine understood and construct great slews of binary codes for the processor to execute. No screens, no hard drives or floppy drives, everything keyed in on a large bank of switches and lights. Very hard work!

However, as technology advanced into the 1960s and 1970s, increased machine capabilities made it possible for higher-level programming languages to appear. These languages were implemented by compilers: a compiler is a program that converts human-readable programs–written in languages like FORTRAN, BASIC, or C—into

---

[5] If you look closely at any USB accessories you may have for your desktop computer you may be able to see that they do in fact have a microcontroller at their heart. Many toys, novelties, and domestic appliances are similarly built around a microcontroller. If you have a USB memory stick or pen drive, you will find it almost certainly has a microcontroller inside.

[6] Originally it stood for Peripheral Interface Controller.

[7] Some of these have been addressed in more recent PIC chips, but some still remain.

[8] Various tests have concluded that if an AVR and a PIC are clocked at the same speed and set to do an identical task, the AVR will be around four times faster than the PIC. I have never tried this, but I have observed notable speed differences in strong favor of the AVR.

actual machine instructions. The advent of compilers (and their increasing importance and scope during the period 1960–1980) meant that human programmers gradually became insulated from the need to know intricate details of the computer processors for which they were creating software.

CISC: The Computer Industry Gets a Complex!

By the early 1980s, huge improvements in semiconductor manufacture made it possible to implement ever more complex computer processors, and there was a kind of a gold rush. Each major manufacturer of the time (IBM, Sperry, ICL, Boroughs, DEC, etc.) vied to give successive generations of processors ever more complex and comprehensive instruction sets. The theory was that if you implemented commonly used functions such as string searches or list processing as a single machine instruction, then your machine would out perform its competition. Equally important, if your processor achieved more useful work with each instruction, you needed fewer instructions for any given program task, and the program size would be smaller. Minimizing program size was a very important consideration in 1980, when a computer that had 512 kilobytes of RAM (half a megabyte) was a top-of-the-range machine!

This gold rush went on until the mid-1980s with processors getting faster, but more complex, with each passing year. By the mid-1980s, a processor like Digital Equipment Corp.'s VAX boasted an instruction set totaling about 160 instructions, further subdivided into more than 400 variants.

By the 1980s, almost nobody was programming large computers in machine language any more. Almost all software was being written in high-level languages like C, PASCAL, BASIC, and FORTRAN. A very lucrative software industry had grown up writing compilers.

A compiler is not an easy or cheap piece of software to create. A compiler has two major headline functions. The first function (the front end) is to examine the source code written by a human programmer and make sure it obeys the rules of the high-level language; then, if all is well, it will convert the steps of that code into a number of generic "tokens." The second function (the back end) is to take that set of generic tokens and convert it into a stream of machine code. Obviously, the back end must produce a machine code stream that is specific to the instruction set of the target machine: the machine on which the executable version of the program is to be run.[9]

The KISS Principle Reasserts Itself

You've doubtless heard of the "Keep It Simple Stupid" (KISS) principle - a way of saying that a back-to-basics approach to things can often be a revelation. Well, the computer industry had its own KISS moment back in the mid-1980s.

A study came out of Stanford University in California from a team headed by Professor John L. Hennessy. This study was the result of work that had taken several years to complete. The team had analyzed–in exhaustive detail–the machine code streams produced by a wide a range of compiler products. The results pointed to a somewhat shocking conclusion: one that changed the whole field of processor design. The study found that 90 % of compiler-generated software used only about 10 % of the available instructions on any given processor type. So, it seemed that all the effort that processor designers had put into designing ever more ambitious instruction sets was wasted; most of the software running on these computers actually wasn't using their more sophisticated features!

When Hennessy's team sought the reasons for this underuse of instruction sets, they found that the main underlying cause wasn't a technical one at all–it was a commercial one.

Team members realized that the market had evolved in such a way that most compilers were being created by independent companies, not by the computer manufacturers themselves. These compiler companies were achieving economies of scale by creating their products in such a way that they could be used on many ranges

[9]There are quite a few other intermediate steps performed by a compiler; these are only the major ones.

of computers. Thus, a compiler vendor might have a FORTRAN compiler which worked on IBM, ICL, DEC, Intel, and Unisys machines. That compiler would have a common front-end section, and a manufacturer-specific back-end section.

Given the cost and complexity of developing compilers for all these platforms, the back-end sections tended to use only the simpler instructions of the computers concerned, and not the more complex, unique, ones. It simply wasn't worth the compiler vendor's time and effort to optimize the back-end part of the compiler per computer architecture. This then was the main reason most of the software analyzed by the Stanford study used only 10 % of the available instruction sets. Additionally, during the later 1980s RAM memory sizes in computers grew much larger; in 1988 even a desktop machine would have 8 or perhaps 16 megabytes of RAM installed. That meant that the need to keep program sizes to the absolute minimum was easing, further reducing the need to use complex, machine-specific, instructions.

Professor Hennessy's Stanford team reflected that, through the 1980s, improvements in speed and device density in the underlying silicon technology had been used to enable more complex processor architectures and larger and ever more complex instructions sets. However, the team's detailed analyses of numerous software programs conclusively showed that, in fact, very few programs made use of these advanced features. They characterized the state-of-the-art machines of the mid-1980s as CISC (Complex Instruction Set–pronounced "SISK") computers.

They posed a new question: given the great increases in capacity and speed of semiconductors, if processors had stayed very simple, with small, elegant instruction sets, how much faster would they be running now? They imagined a stripped-down processor along these lines, and they called it a RISC computer.

They showed that if you designed a machine with a uniform instruction set, in which each instruction had the same format, and in which there were only very simple conditional branch instructions,[10] then you could dedicate more chip space to features that would enhance execution speed, such as a subsystem to prefetch instructions from memory into the processor "pipeline," meaning that the processor was continuously busy, rather than spending an appreciable percentage of its time waiting for its next instruction to be fetched. In an ideal RISC design, the processor completes one machine instruction for each and every clock cycle–something CISC processors could never do. In other words, the goal of a RISC processor design is that if its clock speed is, for example, 20 MHz, then it will be able to execute 20 million instructions per second.

RISC Goes Primetime

The work done by the Hennessy team was so influential that, within only a few years, it changed the course of computer technology. The R2000 from MIPS Computer (released in late 1985) was the first commercially available microprocessor to implement the RISC principles. It took a couple of years, but eventually, when the R2000 was implemented in Unix systems from DEC and Silicon Graphics (SGI), among others, its performance left equivalent CISC-based machines for dead.

The R2000 was swiftly followed by the R3000 and successive generations of RISC processors from MIPS and many others. The RISC processors outperformed their CISC predecessors for almost all mainstream applications.

Since that time, all new mainstream processor designs have used most of the ideas embodied in the RISC philosophy. By 1990, CISC designs were either starting to fade away or–as with the Intel range used in personal computers–being updated to include as much RISC-ness as possible, while still retaining historical compatibility. In other cases, companies brought out new RISC architectures to replace eclipsed CISC architectures; for example, DEC's Alpha RISC architecture replaced its older VAX range of CISC processors and Apple and Sun Microsystems traveled a similar route in changing their base hardware platforms. The blazing performance of MIPS Computer products meant that they showed up in a new class of products: game consoles. Crack open an old PlayStation or a Nintendo 64 and you'll find a MIPS chip in there doing the graphics chores.

---

[10] Such an instruction might be "branch if zero"—meaning if the result of the last instruction was zero then do "this," or if it was not zero then do "that."

Wraps Off AVR

In 1996 the semiconductor company, Atmel, released a new product called AVR (by the way, Atmel says that the initials AVR don't stand for anything in particular). The AVR is a microcontroller chip designed, from the ground up, around the RISC principles whose history and provenance we discussed in the previous section.

This innovative product used, for the first time on a microcontroller, flash memory, meaning that it could easily be reprogrammed with new software while in situ[11] on its application board. AVR also included innovations around the amount of I/O capability it had on-chip–it featured more than was usual at the time.

It was around this time that it started to become essential to differentiate between microprocessors (a processor on a chip) and microcontrollers (a potentially complete computer, with processor, memory, and I/O subsystem on a chip).

After the first 8-bit AVR microcontroller was released by Atmel in 1996, there was a steady stream of new AVRs to follow, each faster and more capable than the last. This eventually included a family of 32-bit AVR processors for use in very demanding applications such as engine management systems.

The AVR family has several primary characteristics:

- It is a common family of processors with code compatibility across the range because the processors all use the same RISC processor core.
- The range of code-compatible chips allows the designer to find the right trade-off between features and cost. All AVR chips have the microprocessor core, but each chip in the range features a different set of peripheral ports and devices, with differing amounts of flash and RAM memory. This range of products allows the designer to select the chip which offers exactly the right amount of capability, and price point, for the job in hand. This is very important when designing commercial products: for several reasons, the number and cost of components are often a key decider of the success or failure of a consumer product.
- AVR espouses RISC design principles and makes very good use of each clock cycle, allowing it to outperform older architectures running at the same clock speed. Of course, since not all microcontroller uses are time-critical, this is less of a consideration in some applications than others, and speed of execution is not the only thing to be considered in designing a microcontroller application. Nevertheless, where speed is an issue–or likely to become an issue–AVR is a very good choice.

[11] The first microprocessors had no on-chip program memory; they needed external ROM chips to hold their programs. The second generation of microcontrollers offered updatable program memory, but it was implemented using EEPROM (Electrically Erasable Programmable Read Only Memory), which required that the chip be removed from circuit and put into an infrared light box which caused the light-sensitive cells on the memory portion of the chip to be reset. Once erased, the memory could be reprogrammed. Flash memory—used on third-generation microcontrollers onward—behaves very much like a RAM and is thus far easier to use and reuse: with careful design the flash memory can be updated in place (i.e., without physically removing the microcontroller from the application circuit).

Summary

The application of the RISC philosophy to computers in general enabled a big step up in computer performance, and when used in the AVR family, it made it possible to offer a highly performing processor core that could be common throughout the extensive AVR product range.

The advent of the microcontroller truly revolutionized the consumer electronics field, and many others (such as the automotive field). The availability of microcontrollers also facilitated the creation of whole new industries and classes of devices–such as GPS (global positioning system) receivers and MP3 players.

So, now that you know why you should be interested in microcontrollers, and why AVRs are such a good entryway into this fascinating subject—it's time to start getting our hands dirty with some practical work!

Coming Up Next

Building our AVR test bed and development system. Putting together the basic tools and equipment we need to get going.