The following material is excerpted from:

The Microcontroller Idea Book

Circuits, Programs, & Applications

featuring the 8052-BASIC Microcontroller

by Jan Axelson

copyright 1994, 1997 by Jan Axelson

ISBN 0-9650819-0-7
Published by Lakeview Research
Distribution by International Thomson Publishing (ITP) in arrangement with
Peer-to-Peer Communications.

For more information, contact:

Lakeview Research
2209 Winnebago St.
Madison, WI 53704
USA

Phone: 608-241-5824
Fax: 608-241-5848
Email: jaxelson@lvr.com
World Wide Web: http://www.lvr.com

1

Microcontroller Basics

This chapter introduces you to the world of microcontrollers, including definitions, some history, and a summary of what's involved in designing and building a microcontroller project.

What's a Microcontroller?

A microcontroller is a computer-on-a-chip, or, if you prefer, a single-chip computer. *Micro* suggests that the device is small, and *controller* tells you that the device might be used to control objects, processes, or events. Another term to describe a microcontroller is *embedded controller,* because the microcontroller and its support circuits are often built into, or embedded in, the devices they control.

You can find microcontrollers in all kinds of things these days. Any device that measures, stores, controls, calculates, or displays information is a candidate for putting a microcontroller inside. The largest single use for microcontrollers is in automobiles—just about every car manufactured today includes at least one microcontroller for engine control, and often more to control additional systems in the car. In desktop computers, you can find microcontrollers inside keyboards, modems, printers, and other peripherals. In test equipment, microcontrollers make it easy to add features such as the ability to store measurements, to create and store user routines, and to display messages and waveforms. Consumer products that use microcontrollers include cameras, video recorders, compact-disk players, and ovens. And these are just a few examples.

A microcontroller is similar to the microprocessor inside a personal computer. Examples of microprocessors include Intel’s 8086, Motorola’s 68000, and Zilog’s Z80. Both microprocessors and microcontrollers contain a central processing unit, or CPU. The CPU executes instructions that perform the basic logic, math, and data-moving functions of a computer. To make a complete computer, a microprocessor requires memory for storing data and programs, and input/output (I/O) interfaces for connecting external devices like keyboards and displays.

In contrast, a microcontroller is a single-chip computer because it contains memory and I/O interfaces in addition to the CPU. Because the amount of memory and interfaces that can fit on a single chip is limited, microcontrollers tend to be used in smaller systems that require little more than the microcontroller and a few support components. Examples of popular microcontrollers are Intel’s 8052 (including the 8052-BASIC, which is the focus of this book), Motorola’s 68HC11, and Zilog’s Z8.

A Little History

To understand how microcontrollers fit into the always-expanding world of computers, we need to look back to the roots of microcomputing.

In its January 1975 issue, Popular Electronics magazine featured an article describing the Altair 8800 computer, which was the first microcomputer that hobbyists could build and program themselves. The basic Altair included no keyboard, video display, disk drives, or other elements we now think of as essential elements of a personal computer. Its 8080 microprocessor was programmed by flipping toggle switches on the front panel. Standard RAM was 256 bytes and a kit version cost $397 ($498 assembled). A breakthrough in the Altair’s usability occurred when a small company called Microsoft offered a version of the BASIC programming language for it.

Of course, the computer world has changed a lot since the introduction of the Altair. Microsoft has become an enormous software publisher, and a typical personal computer now includes a keyboard, video display, disk drives, and Megabytes of RAM. What’s more, there’s no longer any need to build a personal computer from scratch, since mass production has drastically lowered the price of assembled systems. At most, building a personal computer now involves only installing assembled boards and other major components in an enclosure.

A personal computer like Apple’s Macintosh or IBM’s PC is a general-purpose machine, since you can use it for many applications—word processing, spreadsheets, computer-aided design, and more—just by loading the appropriate software from disk into memory. Interfaces to personal computers are for the most part standard ones like those to video displays, keyboards, and printers.

But along with cheap, powerful, and versatile personal computers has developed a new interest in small, customized computers for specific uses. Each of these small computers is dedicated to one task, or a set of closely related tasks. Adding computer power to a device can enable it to do more, or do it faster, better, or more cheaply. For example, automobile engine controllers have helped to reduce harmful exhaust emissions. And microcontrollers inside computer modems have made it easy to add features and abilities beyond the basic computer-to-phone-line interface.

In addition to their use in mass-produced products like these, it's also become feasible to design computer power into one-of-a-kind projects, such as an environmental controller for a scientific study or an intelligent test fixture that ensures that a product meets its specifications before it's shipped to a customer.

At the core of many of these specialized computers is a microcontroller. The computer's program is typically stored permanently in semiconductor memory such as ROM or EPROM. The interfaces between the microcontroller and the outside world vary with the application, and may include a small display, a keypad or switches, sensors, relays, motors, and so on.

These small, special-purpose computers are sometimes called single-board computers, or SBCs. The term can be misleading, however, since the computer doesn't have to be on a single circuit board, and many types of computer systems, such as laptop and notebook computers, are now manufactured on a single board.

New Tools

To design and build a computer-controlled device, you need skills in both circuit design and software programming. The good news is that a couple of recent advances have simplified the tasks involved.

One is the introduction of microcontrollers themselves, since they contain all of the elements of a computer on a single chip. Using a microcontroller can reduce the number of components and thus the amount of design work and wiring required for a project. The 8052-BASIC microcontroller even includes its own programming language, called BASIC-52.

The other development is personal computers themselves. A desktop computer can help tremendously by serving as a *host system* for writing and testing programs. As you are developing a project, you can use a serial link to connect the host system to a *target system*, which contains the microcontroller circuits you are testing. You can then use the personal computer's keyboard, video display, disk drives, and other resources for writing and testing programs and transferring files between the two systems.

Project Steps

Putting together a microcontroller project involves several steps:

1. Define the task
2. Design and build the circuits
3. Write the control program
4. Test and debug

Sometimes the steps won't follow exactly in this order. You may begin writing your program before you build the circuits, or you may build and test some of the circuits before you start programming. But however you go about it, each of the above steps is part of the process. To see what's involved in each step, let's look at each in more detail.

Defining the Task

Every project begins with an idea, or a problem that needs a solution. For example, How can I monitor light intensity at different locations and times of day to find the best location for a solar collector? Or how can I automate the process of drilling printed-circuit boards? Or how can I create a computer-controlled, animated display for a store window?

Once you know what you want to accomplish, you need to determine whether or not your idea is one that requires a computer at all. In general, a computer is the way to go when the circuits must make complex decisions or deal with complex data. For example, a simple AND gate can easily decide whether or not two inputs are both valid logic highs, and will change its output accordingly. But it would require many small-scale chips to build a circuit that stores a series of values representing sensor outputs and the times they occurred, and displays the information in an easily understandable form.

This type of application is where microcontrollers come in handy. Inside, microcontrollers are little more than a carefully designed array of logic gates and memory cells, but modern fabrication processes allow thousands of these to fit on a single chip. Since the basic functions of a microcontroller—performing arithmetic, logic, data-moving, and program branching functions—are common ones that are useful in many applications, it's practical to design and market a chip that performs these functions. The user accesses the abilities of the microcontroller by writing a program that performs the desired functions.

On the other end of the scale, how do you know if an idea is suitable for a microcontroller, or whether you should use a full desktop computer? If your design requires users to enter or view complex commands, data, or graphical information, or if you need large amounts of data or program storage, then a system with keyboard, full-screen display, and disk drives makes sense. For simpler designs, a microcontroller with perhaps a keypad, small display, and solid-state memory (no disk drives) can often do the job, with less expense and smaller size.

In fact, recently the two extremes have been meeting. Some 32-bit microcontrollers are as capable as desktop systems, and notebook-size computers are available with solid-state, diskless storage. Also, expansion cards, other hardware, and software are now available for those who want to use desktop computers for monitoring and control tasks. So there's something for everyone.

The 8052-BASIC chip described in this book is perfect for many simpler applications, especially control and monitoring tasks. Because the chip is easy to use, it's a good way to learn about microcontrollers and computers in general. Although you can't do the most complex projects with it, you can do a lot, at low cost and without a lot of hassle.

Designing and Building

When you're ready to design and build the circuits for a project, there are several ways to proceed. You can design your circuits from scratch, using manufacturers' data books as guides; you can follow a tested design (a kit or project presented in a magazine for example); or you can buy an assembled single-board computer, adding only the interfaces and programming your application requires. This book presents designs that you can build yourself, but you can also use a kit or assembled board as a base if you wish.

Choosing a chip. Does it matter which microcontroller chip you use? All microcontrollers contain a CPU, and chances are that you can use any of several devices for a specific project.

Within each device family, you'll usually find a selection of family members, each with different combinations of options. For example, the 8052-BASIC is a member of the 8051 family of microcontrollers, which includes chips with program memory in ROM or EPROM, and with varying amounts of RAM and other features. You select the version that best suits your system's requirements.

Microcontrollers are also characterized by how many bits of data they process at once, with a higher number of bits generally indicating a faster or more powerful chip. Eight-bit chips are popular for simpler designs, but 4-bit, 16-bit, and 32-bit architectures are also available. The 8052-BASIC is an 8-bit chip.

Power consumption is another consideration, especially for battery-powered systems. Chips manufactured with CMOS processes usually have lower power consumption than those manufactured with NMOS processes. Many CMOS devices have special standby or "sleep" modes that limit current consumption to as low as a few microamperes when the circuits are inactive. Using these modes, a data logger can reduce its power consumption between samples, and power up only when it's time to take data.

The 8052-BASIC chip is available in both NMOS and CMOS versions. The original 8052-BASIC was an NMOS chip, offered directly from Intel. (Intel's term for its NMOS process is HMOS.) Although Intel never offered a CMOS version directly, Micromint became a source by ordering a batch of CMOS 8052's with the BASIC-52 programming language in ROM. The CMOS version, the 80C52-BASIC, has maximum power consumption of 30 milliamperes, compared to 175 milliamperes for the NMOS 8052-BASIC.

All microcontrollers have a defined instruction set, which consists of the binary words that cause the CPU to carry out specific operations. For example, the instruction *0010 0110* tells an 8052 to add the values in two locations. The binary instructions are also known as operation codes, or opcodes for short. The opcodes perform basic functions like adding, subtracting, logic operations, moving and copying data, and controlling program branching.

Control circuits often require reading or changing single bits of input or output, rather than reading and writing a byte at a time. For example, a microcontroller might use the eight bits of an output port to switch power to eight sockets. If each socket must operate independently of the others, a way is needed to change each bit without affecting the others. Many microcontrollers include bit-manipulation (also called Boolean) opcodes that easily allow programs to set, clear, compare, copy, or perform other logic operations on single bits of data, rather than a byte at a time.

Options for storing programs. Another consideration in circuit design is how to store programs. Instead of using disk storage, most microcontroller circuits store their programs on-chip. For one-of-kind projects or small-volume production, EPROM has long been the most popular method of program storage. Besides EPROMs, other options include EEPROM, ROM, nonvolatile (NV), or battery-backed, RAM, and Flash EPROM. The program memory may be in the microcontroller chip, or a separate component.

To save a program in **EPROM**, you must set the EPROM's data and address pins to the appropriate logic levels for each address and apply special programming voltages and control signals to store the data at the selected address. The programming process is sometimes called *burning* the EPROM. You erase the contents by exposing the chip's quartz window, and the circuits beneath it, to ultraviolet energy.

Some microcontrollers contain a one-time-programmable, or field-programmable, EPROM. This type has no window, so you can't erase its contents, but because it's cheaper than a windowed IC, it's a good choice when a program is finished and the device is ready for quantity production.

Several techniques are available for programming EPROMs and other memory chips. With a manual programmer, you flip switches to toggle each bit and program the EPROM byte by byte. This is acceptable for short programs, but quickly becomes tedious with a program of any length. Computer control simplifies the job greatly. With an EPROM programmer that connects to a personal computer, you can write a program at your keyboard, save it to disk if you wish, and store the program in EPROM in a few easy steps. Data sheets for EPROMs rarely specify the number of erase and reprogramming cycles a device is guaranteed for, but a typical EPROM should endure 100 erase/program cycles, and usually many more.

EEPROMs are much like EPROMs except that they are electrically erasable—no ultraviolet source is required. Limitations of EEPROMs include slow speed, high cost, and a limited number of times that they can be reprogrammed (typically 10,000 to 100,000).

ROMs are cost-effective when you need thousands of copies of a single program. ROMs must be factory-programmed and once programmed, can't be changed.

NVRAM typically includes a lithium cell, control circuits, and RAM encapsulated in a single IC package. When power is removed from the circuit, the lithium cell takes over and preserves the information in RAM, for 10 years or more. You can reprogram an NVRAM n infinite number of times, with the only limitation being battery life.

Flash EPROM is electrically erasable, like EEPROM, but most Flash devices erase all at once, or in a few large blocks, rather than byte-by-byte like EEPROM. Some Flash EPROMs require special programming voltages. As with EPROMs, the number of erase/program cycles is limited.

The 8052-BASIC uses two types of program memory. An 8-kilobyte, or 8K, on-chip ROM stores the BASIC-52 interpreter. For storing the BASIC-52 programs that you write, the BASIC-52 language has programming commands that enable you to save programs in external EPROM, EEPROM, or NVRAM.

Other memory. Most systems also require a way to store data for temporary use. Usually, this is RAM, whose contents you can change as often as you wish. Unlike EPROM, ROM, EEPROM, and NVRAM, the contents of the RAM disappear when you remove power the chip (unless it has battery back-up).

Most microcontrollers include some RAM, typically a few hundred bytes. The 8052-BASIC has 256 bytes of internal RAM. A complete 8052-BASIC system requires at least 1024 bytes of external RAM as well.

I/O options. Finally, input/output (I/O) requires design decisions. Most systems require interfaces to things like sensors, keypads, switches, relays, and displays. Most microcontrollers have ports for interfacing to the world outside the chip. The 8052-BASIC uses many of its ports for accessing external memory and performing other special functions, but some port bits are available for user applications, and you can easily increase the available I/O by adding support chips.

Writing the Control Program

When it's time to write the program that controls your project, the options include using machine code, assembly language, or a higher-level language. Which programming language you use depends on things like desired execution speed, program length, and convenience, as well as what's available in your price range.

Machine code. The most fundamental program form is machine code, the binary instructions that cause the CPU to perform the operations you desire.

Assembly language. One step removed from machine code is assembly language, where abbreviations called mnemonics (memory aids) substitute for the machine codes. The mnemonics are easier to remember than the machine codes they stand for. For example, in the 8052's assembly language, the mnemonic *CLR C* means clear the carry bit, and is easier to remember than its binary code (*11000011*).

Since machine code is ultimately the only language that a CPU understands, you need some way of translating assembly-language programs into machine code. For very short programs, you can *hand assemble*, or translate the mnemonics yourself by looking up the machine codes for each abbreviation. Another option is to use an *assembler*, which is software that runs on a desktop computer and translates the mnemonics into machine code. Most assemblers provide other features, such as formatting the program code and creating a listing that shows both the machine-code and assembly-language versions of a program side -by-side.

Higher-level languages. A disadvantage to assembly language is that each device family has its own set of mnemonics, so you have to learn a new vocabulary for each family you work with. To get around this problem, higher-level languages like C, Pascal, Fortran, Forth, and BASIC follow a standard syntax so that programs are more portable from one device to another. The idea is that with minor changes, you can use a language like BASIC to write programs for many different devices. In reality, each language tends to develop many different dialects, depending on the chip and the preferences of the language's vendor, so porting a program to a different device isn't always effortless. But there are many similarities among the dialects of a single language, so, as with spoken language, a new dialect is easier to learn than a whole new language.

Higher-level languages also simplify programming by allowing you to do in one or a few lines what would require many lines of assembly code to accomplish.

Interpreters and **compilers** are two forms of higher-level languages. An interpreter translates a program into machine code each time the program runs, while a compiler translates only once, creating a new, executable file that the computer runs directly, without re-translating.

As a rule, interpreters are very convenient for shorter programs where execution speed isn't critical. With an interpreted language, you can run your program code immediately after you write it, without a separate compile or assembly step. A compiler is a good choice when a program is long or has to execute quickly. A single language like BASIC may be available in both interpreted and compiled versions.

Each device family requires its own interpreter or compiler to translate the higher-level code into the machine code for that device. In other words, you can't use QuickBASIC for IBM PCs to program an 8052 microcontroller—you need a compiler that generates program code for the 8052.

Compared to an equivalent program written in assembly language, a compiled program usually is larger and slower, so assembly language is the way to go if a program must be as fast or as small as possible. A higher-level language also may not offer all of the abilities of assembly code, though you can get around this by calling subroutines in assembly language when necessary.

BASIC-52 is an interpreted language, but BASIC compilers for the 8052 are also available. In fact, you can have the best of both worlds by testing your programs with the BASIC-52 interpreter, and compiling the finished product for faster execution and other benefits of the compiled version.

Testing and Debugging

After you've written a program, or a section of one, it's time to test it and as necessary, find and correct mistakes to get it working properly. The process of ferreting out and correcting mistakes is called debugging. Easy debugging and troubleshooting can make a big difference in how long it takes to get a system up and running. As with programming, you have several options here as well.

Testing in EPROM. One way is to burn your program into EPROM, install the EPROM in your system, run the program, and observe the results. If problems occur (as they usually will) you modify the program, erase and reburn the EPROM, and try again, repeating as many times as necessary until the system is operating properly.

Development systems. Another option is to use a development system. A typical development system consists of a monitor program, which is a program stored in EPROM or other memory in the microcontroller system, and a serial link to a personal computer. Using the abilities of the monitor program, you can load your program from a personal computer into RAM (instead of the more permanent EPROM) on the microcontroller system, then run the program, modify it, and retry as often as necessary until the program is working properly.

Most development systems also allow single-stepping, setting breakpoints, and viewing and changing the data in memory. In single-stepping, you run the program one step at time, pausing after each step, so you can more easily monitor what the circuits and program are doing at each step. A breakpoint is a program location where the program stops executing and waits for a command to continue. You can set breakpoints at critical spots in your program. At any breakpoint, you can view or change the contents of memory or perform other tests.

Simulators. Another development tool is a simulator, which is software that runs on a desktop computer and uses the video display to demonstrate what would happen if a specific microprocessor or microcontroller were to run a particular program. You can look “inside” the simulated chip, observe the contents of internal memory, and single-step or set breakpoints to stop program execution at a desired program location or condition. In this way, you can get a program working properly before you commit it to EPROM. One drawback to simulators is that they can’t mimic all features of the chip of interest, especially interrupt-response and timing characteristics.

Emulators. An in-circuit emulator (ICE) is hardware that replaces the microprocessor in question by plugging into the microprocessor’s socket on the device you want to test. Like a simulator, an emulator lets you control program execution and monitor what happens at each program step. Microprocessor emulators typically are expensive. A ROM emulator is a lower-cost option that simulates an EPROM (using RAM, for example) for program storage, and usually provides the abilities of a development system as well.

The 8052-BASIC’s development system. The 8052-BASIC system and a personal computer form a complete development system for writing, testing, and storing programs. The personal computer’s keyboard and screen make it easy to write and run programs and view the results.

BASIC-52 has many built-in debugging features that make it easy to test programs. You can run a program immediately after writing it, without having to assemble, compile, or program an EPROM. You can use a `STOP` statement and `CONT` (continue) command to set breakpoints and resume executing your program. You can use `PRINT` statements to display variables as the program runs. And, if you wish, you can use your personal computer for writing programs off-line and uploading and downloading them to the 8052-BASIC system.