THE 8051/8052 MICROCONTROLLER

Architecture, Assembly Language, and Hardware Interfacing

Craig Steiner

Universal Publishers
Boca Raton, Florida
USA • 2005

The 8051/8052 Microcontroller

Architecture, Assembly Language, and Hardware Interfacing

ISBN: 1-58112-459-7
Author: Craig Steiner
Cover Photo and Design: Erika Oliden González

Universal Publishers
Boca Raton, FL USA
www.universal-publishers.com

For Erika

Preface

Despite its relatively old age, the MCS-51 (8052) line of microcontrollers remains one of the most popular in use today. Many derivative microcontrollers have since been produced that are based on—and are compatible with—the 8052. Thus the ability to program an 8052 is an important skill for anyone that plans to develop microcontroller-based products.

This book was written to help the student, hobbyist, and professional understand the 8052 architecture and learn to write assembly language software as well as the general concepts involved in constructing an 8052-based device at the hardware level. Even if you'll be developing your program in 'C', a working knowledge of the underlying architecture as well as the microcontroller's assembly language is extremely important in writing efficient code that will separate a typical 'C' programmer from a far more versatile embedded 8052 software developer.

This book is intended to be both instructional as well as an easy-to-use reference. The various chapters of the book will explain the 8052 architecture step by step so that someone just beginning to use the 8052 may get a firm grasp of the architecture. The appendices are a useful reference that will assist both the novice programmer as well as the experienced developer long after the architecture has been mastered.

No specific knowledge of the 8052 is required but the book does assume the following:

1. A general and conceptual knowledge of programming.
2. An understanding of decimal, hexadecimal, and binary number systems.
3. A general knowledge of electronics and integrated circuits, though no prior knowledge of the 8052 IC is assumed.

This book will first briefly explain what a microcontroller is, then focus on the details of the 8052 microcontroller, describe its architecture and explain its assembly language. We'll then provide a design for a working single-board computer (SBC) and discuss each section of the design at the hardware level and cover the software and hardware development tools that make it possible to create and test the firmware developed by the user. The book will close by implementing software that can access each part of the SBC's hardware and describe a "monitor program" which can be used to control it.

There is some overlap, however, in that the architecture cannot be fully explained without some use of assembly language while the assembly language cannot be fully explained without first understanding the architecture. For this reason there will be sporadic references to assembly language instructions in chapters prior to assembly language being formally explained. In these cases, the references to assembly language instructions will be brief, well-commented, and, hopefully, understandable in the context of the architecture that is being explained. If an assembly language instruction used in the first few chapters isn't clear—or if you're anxious to read more about the instruction—feel free to jump ahead to the chapter on assembly language and then come back to where you were. Don't feel that you must read the entire book in a strictly sequential fashion.

I hope you find this book useful and educational. I would cordially invite you to visit my website,

http://www.8052.com, where you will find additional resources, example source code, and a message forum with a user community of over 12,000 people that exchange ideas, comments, questions, and answers on the 8052 microcontroller.

Additionally, I would invite you to submit any comments, suggestions, or corrections regarding the material in this book to me personally so that future editions may be further improved. A list of errata and corrections for the book will be found at **http://www.8052.com/book**.

I would like to extend my thanks to Atmel and Dallas Semiconductor which both provided me with sample microcontrollers that were extremely useful in the design of the SBC. And, of course, I would like to thank everyone who assisted or encouraged me during the writing of this book. While not en exhaustive list, I would like to specifically thank the following people for their assistance: Gerard Steiner, Steve Taylor, Agustin Dominguez, Erik Malund, Michael Karas, and my wife Erika Oliden. Whether it was proof-reading the manuscript, providing feedback on circuit diagrams, or simply tolerating the long hours I put in writing this book, their help and support was invaluable.

Craig Steiner
craigsteiner@8052.com
http://www.8052.com

Table of Contents

8052 ARCHITECTURE

CHAPTER 1: INTRODUCTION TO 8052 MICROCONTROLLERS

Many developers new to microcontrollers, including the 8052, come from a PC/Windows or Macintosh environment. While most programming concepts will transfer over to the 8052 environment with no problem, there are some issues that may need clarification as you enter the microcontroller world. Before delving into the details of microcontrollers and, specifically, the 8052, we'll address some common stumbling blocks.

1.1 What is a Microcontroller?

A **microcontroller** (often abbreviated **MCU**) is a single integrated circuit that executes a user program, normally for the purpose of *controlling* some device—hence the name micro*controller*.

Microcontrollers are normally found in devices such as microwave ovens, automobiles, keyboards, CD players, cell phones, VCRs, security systems, time & attendance clocks, electronic toys, etc. These are devices that require some amount of computing power but don't require so much as to justify the use of a more complex and expensive 486 or Pentium system which generally requires a large amount of supporting circuitry and memory. A microwave oven just doesn't need that much processing power.

Microcontroller-based systems are generally physically smaller, more reliable, and cheaper than full-blown PC-based systems. They are ideal for the types of applications described above where unit cost and size are very important considerations. In such applications it is almost always desirable to produce designs that utilize the smallest number of integrated circuits, occupy the smallest amount of physical space, require the least amount of energy, and cost as little as possible.

1.1.1 Microcontroller Program Storage

The program for a microcontroller is normally stored on a memory IC—called an EPROM—or in the microcontroller IC itself.

An **EPROM** (Electrically Programmable Read Only Memory) is a special type of IC that does nothing more than store program code or other data that is not lost when power is removed. Traditionally, software for a microcontroller is assembled or compiled on a PC and is subsequently programmed (or "burned") into an EPROM which is then physically inserted into the circuitry of the hardware. The microcontroller accesses the program stored in the EPROM and executes it. This allows the program to be made available to the microcontroller without the need for a hard drive, floppy drive, nor any of the other circuitry and software necessary to manage such devices.

In recent years a growing number of microcontrollers have offered the capability of having programs loaded internally into the microcontroller IC itself. Rather than having a circuit that includes both a microcontroller and an external EPROM, it is now possible to have a just a microcontroller which stores the program code internally.

1.1.2 “Loading” a Microcontroller Program

The manner in which software is transferred from the PC to the hardware depends on whether the design uses an external EPROM or is using a more modern microcontroller that allows the program to be loaded into the microcontroller IC itself.

Programming an EPROM requires special hardware called a **device programmer** (see section 16.1). An EPROM programmer is a device that connects to the PC via the serial, parallel, or USB port. The EPROM is placed into a connector ("socket") and special software transfers the program from the PC to the device programmer which in turn “burns” the program onto the chip. Once the program is burned into the chip, the EPROM is removed from the device programmer and inserted into the physical hardware in which it is to run. EPROMs can subsequently be erased and reprogrammed by exposing them to ultraviolet light for a brief period of time. Devices called **EPROM erasers** exist which illuminate EPROMs with ultraviolet light for the purpose of erasing them.

Programming a microcontroller that stores the program within the microcontroller itself generally requires a serial or parallel port be available for downloading updates to the program. Most of these devices can be programmed by traditional device programmers, as described in the previous paragraph, but if a circuit is being designed from scratch it is usually a good idea to plan for the possibility of programming the microcontroller without removing the IC from the circuit itself—this is especially true of surface-mount parts that may be difficult to remove from the circuit. The datasheet for the microcontroller being used should provide the information necessary to design the circuit for this "in-circuit programming" capability. Datasheets for microcontrollers are available from the websites of each of the companies that produce them.

1.2 What is an 8051 or 8052?

The **8052** is an 8-bit microcontroller originally developed by Intel in the late 1970s. It includes an instruction set of 255 operation codes (**opcodes**), 32 input/output lines, three user-controllable timers, an integrated and automatic serial port, and 256 bytes of on-chip RAM. The **8051** is similar but has only two timers and 128 bytes of on-chip RAM.

The 8052 was designed such that control of the MCU and input/output between the MCU and external devices is accomplished primarily using **Special Function Registers** (**SFR**s, see chapter 3). Each SFR has an address between 128 and 255. Additional functions can be added to new *derivative* MCUs by adding additional SFRs while remaining compatible with the original 8052. This allows the developer to use the same software development tools with any device that is “8052-compatible.”

Over time, other semiconductor firms adopted the "8052 core" for their microcontrollers, using the same instruction set and underlying SFRs. This allowed the 8052 architecture to become an industry-wide standard. Now, more than 20 years later, dozens of semiconductor companies produce hundreds of microcontrollers that are based on the original 8052 core. The additional features that each semiconductor-firm offers in their MCUs are accessed by utilizing new SFRs in addition to the standard 8052 SFRs that are found in all 8052-compatible MCUs.

While most microcontrollers based on the 8052 core will include the standard SFRs from the 8052 core, some derivatives may only implement a subset of them. They may also change the function of some bits. This is generally not the case, but it is something to look out for when using derivative chips.

In this book, the term "**8052**" will refer to any MCU that is compatible with the original 8052. As a minimum it will support the 8052 instruction set, support the 8052's 26 SFRs, provide three user timers, and have at least 256 bytes of internal RAM.

The term "**8051**" will refer to any 8052-compatible MCU that doesn't meet the specifications in the previous paragraph, rather mirroring Intel's 8051 microcontroller which was a more limited version of the 8052. As a minimum, an 8051 must support the 21 SFRs supported by the original 8051 and support the standard 8052 instruction set. Generally an 8051 will have two user timers and 128 bytes of internal RAM, although some devices have as little as 64 bytes.

1.2.1 Derivative Chips

The term "**derivative chip**" refers to any 8051 or 8052-compatible microcontroller. There are currently hundreds of derivatives produced by dozens of semiconductor firms.

A derivative will generally—but not always—be able to execute a standard 8052 program without modification. A derivative chip must be based on the 8052 instruction set and support the appropriate SFRs (at least 21 SFRs for an 8051 or 26 for an 8052). Some derivatives operate at different speeds than a standard 8052 so some adjustment to program timing may be necessary when moving to a given derivative from a standard device.

Software development tools designed for the 8052 can always be used to develop software for any derivative chip as long as the programmer explicitly defines any new SFRs that are supported by the derivative chip that they are using.

1.3 Using Windows or Linux

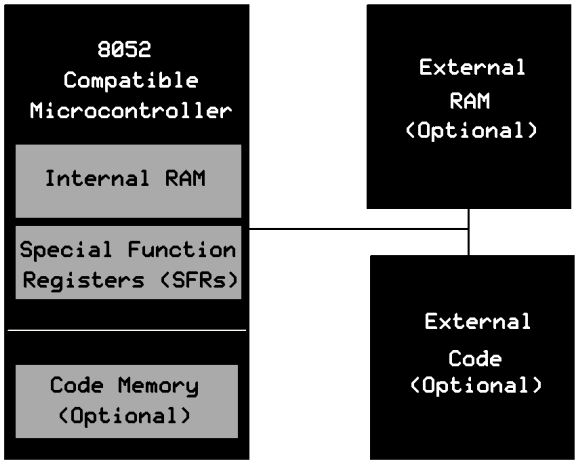
Some people ask whether or not a microcontroller is Windows-compatible or if they can load Linux on their microcontroller. No, a microcontroller cannot run Windows nor can it run Linux. Nor is a microcontroller Windows-compatible, per se. Your microwave oven doesn't run Linux and your automobile is not Windows-compatible.

It is important to remember what microcontrollers are used for. A microcontroller is not a personal computer. It is an integrated circuit that will run short, specialized programs to control hardware devices such as those mentioned earlier. A microwave oven and an automobile simply don't need complex operating systems to perform their designated functions.

That said, Windows or Linux can be used to develop programs *for* microcontrollers. Many Windows products exist that allow 8052 programs to be written in the Windows environment even though the software will ultimately be run by a microcontroller. Once the software is executed by a microcontroller it doesn't matter whether it was originally developed under Windows, Linux, or any other operating system.

CHAPTER 2: TYPES OF MEMORY

The 8052 has three very general types of memory. To program the 8052 effectively it is necessary to have a basic understanding of these memory types. They are: On-Chip Memory, External Code Memory, and External RAM.



Code Memory is code (or program) memory that is used to store the actual program. This often resides off-chip in the form of an EPROM. Many modern derivative chips allow program storage on the MCU itself (on-chip code memory) and some modern derivative chips do not even support the concept of having code memory located off-chip.

External RAM is RAM memory that resides off-chip. This is typically in the form of standard static RAM.

On-Chip Memory refers to any memory (code, RAM, or other) that physically exists in the MCU itself. On-chip memory can be of several types which will be discussed in section 2.3.

2.1 Code Memory

Code memory is the memory that holds the actual 8052 program that is to be run. This memory is conventionally limited to a maximum of 64K and comes in many shapes and sizes: Code memory may be found on-chip, either burned into the microcontroller as ROM or EPROM, or loaded into flash program memory in newer derivatives. Code may also be stored completely off-chip in an external ROM or, more commonly, an external EPROM. Various combinations of these memory types may also be used—that is to say, it is often possible to have 4K of code memory on-chip and 64k of code memory off-chip in an EPROM, depending on the derivative chip being used.

When the program is stored on-chip, the 64K maximum is often reduced to 1k, 2k, 4k, 8k, 16k, or 32k. This varies depending on the derivative chip in question. Each derivative offers specific capabilities and one of the distinguishing factors is the amount of on-chip code memory the part offers.

Code memory of a “classic” 8052 system is usually off-chip EPROM. This is especially true in low-cost development systems and in systems developed by students in which learning how to interface with off-chip memory is part of the exercise.

Since code memory is restricted to 64K, 8052 programs are limited to 64K. Some compilers offer ways to get around this limit when used with specially wired hardware and a technique known as “memory banking.” However, without such special compilers and hardware configurations, programs are limited to 64K.

Some manufacturers such as Dallas Semiconductor, Philips, and Analog Devices have special 8052 derivatives that can address several megabytes of memory.

2.1.1 Memory Architecture

In any given system, the memory architecture can normally be described as either "Harvard" or "Von Neumann". In the simplest terms, this describes whether code and data are contained in two separate memory areas or share a common memory space.

Von Neumann architecture is a system in which code (compiled program, instructions, and constants) and data (variables, registers, etc.) are stored in the same memory space. This is the architecture used in PCs where programs are loaded on-demand from hard drives and subsequently executed from memory. This architecture is very flexible but has the potential disadvantage that a poorly written program can actually destroy itself—or destroy other programs—by erroneously writing data to an area of memory that holds program code. In Windows, this type of problem commonly results in the infamous "General Protection Fault" error message.

Harvard architecture is a system in which code and data are stored in separate memory space. This architecture has the advantage that a program cannot accidentally overwrite itself. This architecture is commonly used in embedded systems where the program is seldom modified nor is it loaded and unloaded.

8052 systems generally are designed with a Harvard architecture. Thus there may be 64k of data memory available (often in the form of an external RAM IC) and 64k of code memory (often in the form of an external EPROM). An 8052 program running under this architecture cannot modify itself intentionally or accidentally.

The 8052 can be used in a Von Neumann architecture by connecting the MCU to external RAM in a certain way—this will be discussed in section 14.2.5. In practice, this is seldom done.

2.2 External RAM

Although 8052s contain a small amount of on-chip internal RAM (see section 2.3.1), external RAM is also supported.

As the name suggests, external RAM is any random access memory that is found off-chip. Since the memory is off-chip the assembly language instructions to access it are slower and less flexible. For example, to increment an internal RAM location by 1 requires only 1 instruction and 1 instruction cycle. To

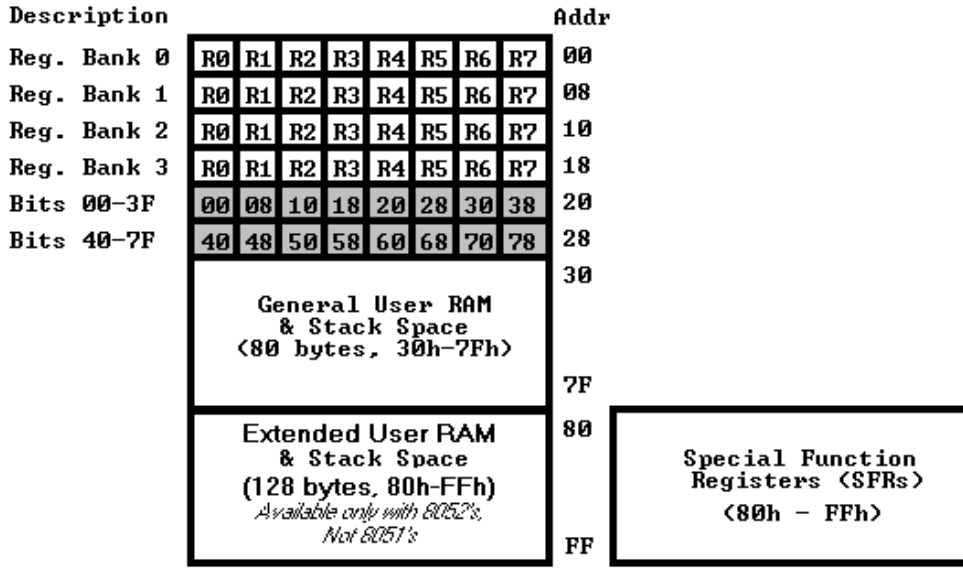
increment a value stored in external RAM requires 4 instructions and 7 instruction cycles. In this case, external memory is seven times slower and requires four times as much program memory to manipulate.

What external RAM loses in speed and flexibility it gains in quantity. While internal RAM is normally limited to 256 bytes (128 with 8051s), the 8052 supports external RAM up to 64K.

Generally, the 8052 may only address 64k of RAM. To expand RAM beyond this limit requires programming and hardware tricks.

2.3 On-Chip Memory

As mentioned at the beginning of this chapter, the 8052 includes a certain amount of on-chip memory. Aside from on-chip code memory, on-chip memory is really one of two types: **Internal RAM** and **Special Function Register** (SFR) memory. The layout of the 8052's internal memory is presented in the following memory map.



2.3.1 Internal RAM

As the figure above shows, the 8052 has a bank of 256 bytes of internal RAM. This internal RAM is found on-chip within the 8052 so it is the fastest RAM available. It is also the most flexible in terms of reading, writing, and modifying its contents. Internal RAM is volatile so when the 8052 is powered-up, this memory is undefined.

The 256 bytes of internal RAM are subdivided as shown in the memory map above. The first eight bytes (00h - 07h) are "register bank 0". By manipulating the Program Status Word (PSW) SFR, a program may choose to use register banks 0, 1, 2, or 3. These alternative register banks are located in internal RAM in

addresses 08h through 1Fh. "Register banks" are discussed in section 2.3.1.2. For now it is sufficient to know that they are located in—and are part of—internal RAM.

Bit memory also is located in and is a part of internal RAM. Bit memory is covered in section 2.3.1.3. For now just keep in mind that bit memory actually resides in internal RAM from addresses 20h through 2Fh.

Also note that all of internal RAM is general-use, byte-wide memory, regardless of whether it is used by register banks or bit memory. This means that internal RAM allocated to register banks 1, 2, and 3 and bit memory may be used for other purposes if the program will not be using those register banks and/or bit variables.

The remaining 208 bytes of internal RAM, from addresses 30h through FFh, may be used by user variables that need to be accessed frequently or at high-speed. This area is also utilized by the microcontroller as a storage area for the operating stack. This significantly limits the 8052's stack size since the area reserved for the stack is only 208 bytes—and usually much less since the 208 bytes of internal RAM must be shared between the stack and user variables.

While the 8052 has 256 bytes of internal RAM, the 8051 only has 128 bytes. This further restricts the amount of RAM available on-chip for user variables and the operating stack.

2.3.1.1 Stack

The stack is a "last in, first out" (LIFO) storage area that exists in internal RAM. It is used by the 8052 to store values that the user program manually pushes onto the stack as well as to store the return addresses for calls and interrupt service routines.

The stack is defined and controlled by an SFR called the stack pointer, or SP. SP, as a standard 8-bit SFR, holds a value between 0 and 255 that represents the internal RAM address of the *end* of the current stack. If a value is removed from the stack, it will be taken from the internal RAM address pointed to by SP and SP will subsequently be decremented by 1. If a value is pushed onto the stack, SP will first be incremented and the value will be inserted in internal RAM at the address now pointed to by SP.

SP is initialized to 07h when an 8052 is first powered-up. This means the first value to be pushed onto the stack will be placed at internal RAM address 08h (07h + 1), the second will be placed at 09h, etc.

By default, the 8052 initializes the stack pointer (SP) to 07h. This means that the stack will start at address 08h and expand upwards. If a program will be using the alternate register banks (banks 1, 2 or 3) it must initialize the stack pointer to an address above the highest register bank that it will be using, otherwise the stack will overwrite the alternate register banks. Similarly, if the program will be using bit variables it is usually a good idea to initialize the stack pointer to some value greater than 2Fh to guarantee that the bit variables are protected from the stack.

2.3.1.2 Register Banks

The 8052 includes eight "R" registers which are used in many of its instructions. These "R" registers are

numbered from 0 through 7 (R0, R1, R2, R3, R4, R5, R6, and R7) and are generally used to assist in manipulating values and moving data from one memory location to another. For example, to add the value of R4 to the accumulator, the following assembly language instruction would be used:

```
ADD A,R4
```

Thus if the accumulator (A) contained the value 6 and R4 contained the value 3, the accumulator would contain the value 9 after this instruction was executed. However, as the memory map shows, the "R" register R4 is really part of internal RAM. Specifically, R4 is address 04h of internal RAM. Thus the above instruction is functionally equivalent to the following operation:

```
ADD A,04h
```

This instruction adds the value found in internal RAM address 04h to the value of the accumulator, leaving the result in the accumulator. Since R4 is really internal RAM address 04h, the above instruction produces the same result as the previous ADD instruction.

But as the memory map also shows, the 8052 has four distinct register banks. When the 8052 is first initialized, register bank 0 (addresses 00h through 07h) is used by default. However, the user program may instruct the 8052 to use one of the alternate register banks; i.e., register banks 1, 2, or 3. In that case, R4 will no longer be the same as internal RAM address 04h. For example, if the program instructs the 8052 to use register bank 1, register R4 will now be synonymous with internal RAM address 0Ch. If the program selects register bank 2, R4 will be synonymous with 14h, and if it selects register bank 3 it will be synonymous with address 1Ch.

The register bank is selected by setting or clearing the bits RS0 and RS1 in the Program Status Word (PSW) Special Function Register.

```
MOV PSW,#00h        ;Sets register bank 0
MOV PSW,#08h        ;Sets register bank 1
MOV PSW,#10h        ;Sets register bank 2
MOV PSW,#18h        ;Sets register bank 3
```

The above instructions will make more sense after the topic of Special Function Registers is covered in section 2.3.2 and in chapter 3.

The concept of register banks adds a great level of flexibility to the 8052, especially when dealing with interrupts (more about interrupts in chapter 10). But always remember that the register banks reside in the first 32 bytes of internal RAM.

If only the first register bank (i.e. bank 0) is being used, internal RAM locations 08h through 1Fh may be used for anything the programmer desires. If the program uses register banks 1, 2, or 3, be very careful about using addresses below 20h as the program may end up overwriting the value of "R" registers in other register banks.

2.3.1.3 Bit Memory

The 8052, being a communications and control-oriented microcontroller that often has to deal with “on”