Microprocessor Design Made Easy

Using the MC68HC11

Raj Shah
President
Advanced Microcomputer Systems, Inc.
Pompano Beach, Florida

ams
Educational Division

ACKNOWLEDGMENTS

"Microprocessor Design Made easy Using the MC68HC11" was designed and developed through the talents and energies of many hard working and dedicated individuals.

The concept of using EZ-MICRO Tutor Board and EZ-MICRO Manager software along with this workbook in the schools has been well accepted by several community colleges as well as universities.

Library of Congress Catalog-in-Publication Data

Microprocessor Design Made Easy Using the MC68HC11

ISBN 0-9642962-4-1
Third Edition March, 2000
Fourth Edition January, 2002

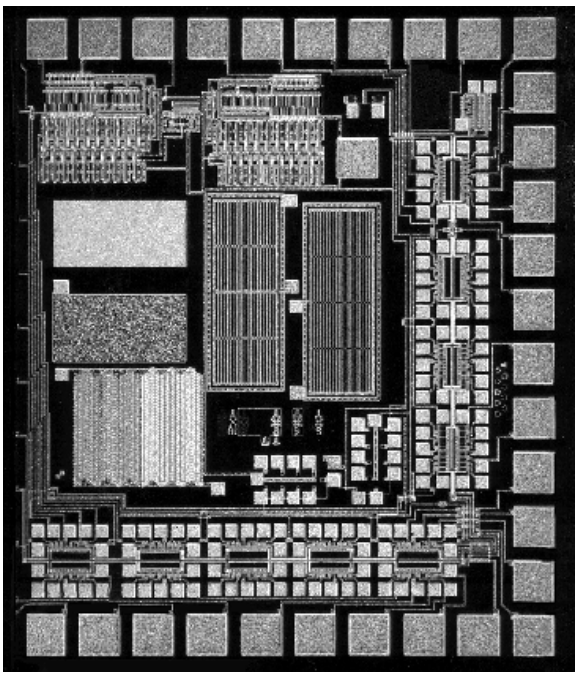
Microprocessor Design Made easy Using the MC68HC11

Copyright © 200, 2001, 2002 by

Advanced Microcomputer Systems Inc
1460 SW 3rd Street, Pompano Beach, FL 33069
Phone: (954) 784-0900 • Fax: (954) 784-0904
E-Mail: info@advancedmsinc.com
Web: www.advancedmsinc.com

All rights reserved.
Printed in the United States of America.

Table of Contents



Lesson One

Introduction to Microprocessors and the Microprocessor System Design Process

When you finish this lesson, you will know:

1. The history and evolution of the microprocessor.
2. The difference between 8-bit, 16-bit, and 32-bit microprocessors.
3. The different types of microprocessors available and their features.
4. The applications of each type of microprocessor.
5. The process of designing a microprocessor-controlled system.

This EZ-COURSEWARE manual describes microprocessors and the microprocessor-system design process. It's written for students studying single-chip microcomputers and microcontrollers for use in consumer products, manufacturing equipment, and laboratory instrumentation. Basically, a microprocessor is a single integrated circuit, often containing millions of transistors, that serves as the "brains" of a larger system, such as a personal computer. The single-chip microprocessor is also an ideal component for controlling mechanical and electrical devices, like a VCR or microwave oven. When this chip controls a specific product it's called a *microcontroller*. This hands-on course focuses on the microcontroller aspect of the industry.

Most of us already use products with microcontrollers embedded in them without even knowing it. Microcontrollers touch almost every facet of daily life and provide sophisticated features to consumer products at low cost. The following is a short list of some common products that use microcontroller computers.

Audio equipment. Most CD players have electronic control buttons, digital displays, and automatic track selection and sequencing — all run by a microcontroller.

Automobiles. Virtually every car in production today uses a microcontroller to control the delivery of fuel and adjust the spark timing for the engine. Microcontrollers can also be found in automatic transmissions, anti-lock brakes, and dashboard gauges.

HVAC controllers. You probably know them better as thermostats, the wall devices that regulate the temperature of our homes and workplace. Unlike the bi-metal devices of old, today's smart heating and air conditioning controllers have a microcontroller for a brain.

Microwave ovens. Many microwave ovens can be programmed for various cooking times and power levels. Some even have sensors that automatically sense the food's temperature and change the program accordingly. A microcontroller controls these and other functions.

Security systems. If it weren't for the microcontroller, which is able to detect motion and verify passwords, personal and building security would still be stuck in the world of locks and deadbolts.

Video equipment. Undoubtedly the biggest consumer of single-chip microcontrollers is the video industry, which includes TVs and VCRs. Think about it. How else could you remotely change channels or program a VCR to record "Star Trek Voyager" weekly without something as smart as a microcontroller?

From Fingers to Transistors

The microcontroller that you will study in this guide is the MC68HC11 (also known as the 68HC11) from Motorola. Before we start, though, a little history is in order so that you know how the microcontroller came into existence.

All microprocessors and microcontrollers are digital devices, in that they use numbers to direct and control their actions — as opposed to analog devices, which use quantities like weights and

measures for their data input. For example, a VCR has a microcontroller for its brain, whereas a toaster usually has an analog bi-metal sensor (its "brain") that senses the amount of moisture in the bread and (hopefully) pops up the toast before it burns.

The first digital computer was "invented" when early humans realized that they could count sheep or bushels of corn using fingers (digits). Since that time, inventors have constantly sought better, faster, and cheaper ways to count and tally.

The earliest known computing instrument is the abacus, a simple adding machine composed of beads strung on parallel wires in a rectangular frame, that's used simply as a memory aid by a person making mental calculations. In contrast, adding machines, electronic calculators, and computers make physical calculations without the need for human reasoning. Blaise Pascal is widely credited with building the first "digital calculating machine" in 1642 as a way to help his father, who was a tax collector. His machine could add up numbers entered by means of dials. In 1671, Gottfried Wilhelm von Leibniz improved on the design by introducing a special "stepped gear" mechanism, which let the adding machine do multiplication, too. However, the prototypes built by Leibniz and Pascal weren't widely used but remained curiosities until more than a century later, when in 1820, Thomas of Colmar (Charles Xavier Thomas) developed the first commercially successful mechanical calculator that could add, subtract, multiply, and divide.

At about the same time (1812), Charles Babbage, a professor of mathematics at Cambridge University, England, advanced the concept of a "difference engine:" an automatic calculating machine that many consider to be the forerunner of today's electronic computer. What Babbage realized was that that many long computations, especially those needed to prepare mathematical tables, consisted of routine operations that were regularly repeated; from this he surmised that it ought to be possible to do these operations automatically. By 1822 he had built a small working model for demonstration. With financial help from the British government, in 1823 Babbage started construction of a full-scale, steam-driven difference engine but lost interest in the project before completing it — partly because the machining techniques of his era weren't precise enough to create so complex a device.

Not for another century (1941) was serious attention again devoted to developing a calculating machine — this time, for generating mathematical tables needed by the World War II armies for determining the trajectory of ballistics. By now, state-of-the-art electronics had advanced to the point where mechanical devices weren't even a consideration. The first electronic calculator was the ENIAC (for Electrical Numerical Integrator And Calculator), which used 18,000 vacuum tubes, occupied 1,800 square feet of floor space, consumed 180 kW of power (enough energy to power a city of 30,000), and ran hot enough to supply all the heat needed to warm the seven-story building it was housed in. The vacuum tubes were soon replaced with transistors (1955), which were supplanted in turn by the integrated circuit in 1964.

The Making of a Microprocessor

But the mindset was still that of counting, just at a faster pace. It wasn't until the advent of the Central Processing Unit (CPU), in 1971, that the concept changed from counting to computing. That's when logic was introduced into the counting process, and ushered in the era of the microprocessor. Simply put, computer logic means that the computer can be "taught" to choose

different courses of action depending on the specific inputs it's been given, especially input from previous calculations. For example, a computer used in a payroll department will "look" at how many income tax deductions a particular employee has chosen, then look at the employee's gross salary, and then access the right look-up table based on both factors to decide how much tax to subtract from that person's paycheck. Similarly, a computerized cash register in a supermarket can "remember" to add sales tax at the end of your bill for the paper towels and laundry soap in your grocery cart, but not for your tax-free onions and chicken wings. This is what separates a calculator from a computer: the ability to choose a course of action dependent on the outcome of a previous calculation. Some call it "machine reasoning." It has since come to be known as *programming*.

At first machine reasoning wasn't all that swift—mostly because the electronics of that time wasn't all that advanced. Where today's Pentium processors sport more than 3 million transistors, the semiconductor makers of twenty years ago were hard pressed to fit a couple thousand transistors on a silicon chip.

The first microprocessor chip was Intel's 4004, a 4-bit processor that contained 2300 transistors and ran at 100 kilohertz (100 kHz). One hertz (Hz) is the time it takes for an electrical signal to swing from positive to negative and back again. For example, the power outlet that you plug your TV and VCR into changes polarity 60 times a second, or 60 Hz. A kilohertz is 1000 Hz. In the computer world, the higher the hertz rating, the faster things get done. Today's microprocessors run at 66 megahertz (66 MHz) and faster.

The 4004 was invented as an engineer's lark that actually turned into a product looking for an application — blue sky stuff. Fortunately, it did find a home — in more than a few Sears hand-held calculators. And for good reason. Unlike an adding machine, the 4004 didn't know how to add or subtract; it was simply a logic chip. Instead, it had something better — it had reasoning plus it had *memory*. Essentially, it had all the answers to any math calculation stored in an area on the microprocessor chip called read-only memory (ROM). Hence, the calculator knew the answer to your math question before you asked. All it had to do was look it up in one of its many tables.

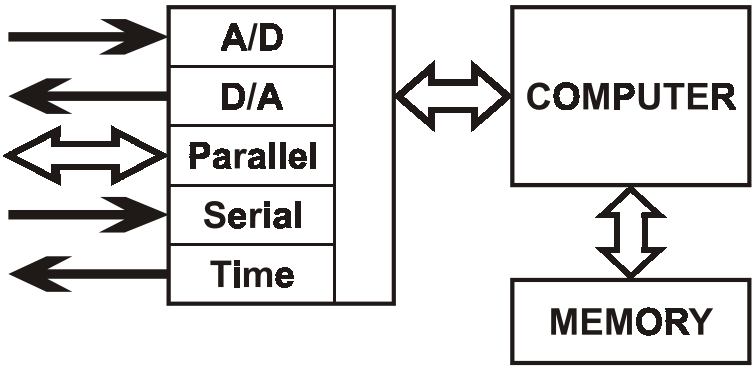
Bits and Bytes

Inspired by the success of the 4004, Intel took another flyer and in 1982 developed the 8008 microprocessor, an 8-bit version of the 4004. Going from 4-bits to 8-bits really gave the microprocessor power, because now it advanced microprocessor status to that of a "real" computer (as in mainframe computing). Eventually, this change would give birth to little laptops with vastly more power and speed than the room-sized computers of forty-odd years ago. To understand how this happened, you need to know about bits and bytes, which calls for another journey back in time.

When people first started counting, they used the obvious — their ten fingers (including thumbs). When our caveman ran out of fingers, he made a mark on the cave wall, or whatever, and started counting on his fingers again. This led to our present 10-base decimal system. Whenever a number is larger than ten, it assumes a new position in the tally. For example, 91 means there are nine sets of ten plus one. A 10-base counting system isn't the only one possible,

though. The Mayan Indians of Central America developed a numerical system based on the number 20. Obviously they included toes as well as fingers.

The computer lacks fingers and toes, but true to its electrical nature it recognizes two values: on and off. If the signal is on, it represents a one (1); if it's off, it represents a zero (0). This method is called *binary* counting. Each digit in the binary system is a power of two, which means the progression is 1, 2, 4, 8, 16, 32, 64...and so forth. In binary, the number 91 is written as 1011011 (64 + 16 + 8 + 2 + 1). A binary digit is called a *bit*. If a binary number is 4-bits in length, it's called a *nibble*; 8-bits is a *byte*. Generally, computer power is expressed in kilobytes. One kilobyte (which is actually 1024 bytes) is written as 1K (note that the "K" is capitalized). String 1,000 K together and you have a megabyte, or 1MB.



Simple Microcontroller

The Personal Computer Revolution

When the 4004's data bus was expanded from 4-bits to 8-bits, the microprocessor rose from calculator to real computer status. A bus is defined as an electrical circuit that transfers data. An 8-bit bus has eight separate wires, each of which carries a binary bit and moves one byte of data at a time; a 16-bit bus has 16 wires. In 1974 the 8008 was upgraded to the 8080 by expanding the address bus to 16-bits wide — and with this change, the personal computer revolution was off and running.

The 8080 could address 64K of memory (the 4004 could manage only 4K) and was an instant success with aspiring computer companies like Atari. Within a year (1976) Intel introduced the 8085, which was basically the 8080 with built-in peripheral logic. (We'll explain peripheral logic in Lesson 2.) By now the technology had advanced to the point where it was possible to place tens of thousands of transistors on a silicon chip, so a lot of space became available for what used to be outboard functions, such as counters and timers. It wasn't long before other silicon foundries jumped on the CPU bandwagon, Zilog with its Z80 (a faster version of the 8080, and undoubtedly the first CPU clone) and MOS Technologies with its 6502. Motorola entered the fray in 1975 with the 8-bit 6800 microprocessor.

By 1978 the personal computer movement was in full stride. The 8080-based Tandy TRS-80 had appeared in 1977, as did the 6502-based Apple II, and the 6502-based Commodore PET made its debut in 1979. By 1980 the market was flooded with dozens of 8080- and Z80-based

machines, most of them from small companies with few resources and even smaller production capabilities. It wasn't until the advent of Intel's 8088 in late 1979 that big industry took notice of this burgeoning market.

Moving Up To 16-bit Technology

Looking to gain a better foothold in a crowded market that was once its sole domain, Intel aggressively shouldered the perils of developing a 16-bit CPU: a risky and costly venture, but one that would knock the socks off the competition. The result was the 8086 — a sturdy workhorse first introduced in 1978 that's still in popular use today, but which received only lukewarm acceptance at the time because of its high price tag. It wasn't the cost of the microprocessor itself that was the problem; it was (at the time) the high cost of the 16-bit support chips needed to make it run. In 1979 Intel found a way to multiplex the 8086's 16-bit data bus down to 8-bits, which drastically reduced the cost of building a 16-bit system using off-the-shelf 8-bit parts. First to jump on the new technology was IBM, which one year later, marketed what is now the industry standard: the IBM Personal Computer.

The 8088 is essentially the same chip as the 8086, in that its internal data path is 16-bits wide. The CPU can process two bytes of information at the same time, thereby increasing throughput and performance. What Intel did with the 8088 was retain the 16-bit CPU core, but reduce the width of the external bus down to 8-bits. Unfortunately, the 8088 was considerably slower than the 8086 because it had to access the 8-bit bus twice before it could process data: once to get the 8-bits near the bottom of the bus and again to input the upper 8-bits. Both the 8086 and 8088 had 20 address lines, which gave them a memory capacity of 1MB — 16 times more than the 64K offered by the 8080, 6800, and 6502.

About the same time, Motorola not-so-quietly introduced its first 16-bit microprocessor: the 68000. Unlike Intel's 8086/8088, the 68000 had 24 address lines that provided access to 16MB of memory — an amount of memory so humongous as to seem mindboggling at the time. Seeing the potential, Apple immediately scooped up the new CPU for use in its new Macintosh computer. Zilog put up a gallant, but unsuccessful, battle to stay in the 16-bit race by introducing the Z8000, an 8086 compatible.

Keeping a tight hold on the reins, Intel immediately began bolstering its line of 16-bit CPUs. First was the 80186, which was an 8086 with several common support functions built right in: clock generator, system controller, interrupt controller, DMA (Direct Memory Access) controller, and timer/counter. However, the 80186 wasn't used as a mainstream CPU for desktop computers, instead finding a short-lived niche in single-board processors for industry.

Next was the 80286, which made its debut in IBM's Personal Computer AT (the AT stands for Advanced Technology) in 1984. The 286, as it's commonly called, paved the way for the advanced Intel-compatible CPUs of today by expanding the number of memory address lines from 20 to 24, resulting in 16MB of memory space — equal to that of Motorola's 68000. But by this time there was a large base of programs written for the 20-bit-wide address space of the 8086 and 8088. To advance the technology without abandoning existing software, the 286 introduced a new mode of operation called *protected mode*. Basically, any program written to the old 8086 standard would run on the new 286 using the standard 1MB of memory provided by the

8086 and 8088, while providing a path to the upper 15MB of memory without conflict. Despite heroic attempts (EMS expanded memory is one), though, DOS-based programs still couldn't break through the 1MB barrier without dragging down the speed of the system to a crawl — or worse, crashing the system. It wasn't until the release of Microsoft's Windows (a powerful, graphics-based overlay on the DOS operating system) that software programs were able to take full advantage of the full 16MB.

By 1985 the field had narrowed to two dominant suppliers of microprocessors: Intel and Motorola. AMD, a prominent silicon foundry, joined this elite circle as a second source for both giants (an infusion of cash and exchange of technology that later led AMD to successfully develop its own line of CPUs). Hitachi, too, took a stab at second-sourcing Motorola products, but it didn't last long (mostly Hitachi acquired the technology for use in its custom products, which included a long list of consumer microcontrollers for use in everything from automobiles to VCRs).

Upping The Ante To 32-bits

In 1985 Intel introduced the first 32-bit microprocessor: the 80386. The 386 provided 32-bit paths on both the data and address buses, which pushed the addressable memory space to a whopping 4 gigabytes (4GB). Like the 286, the 386 used the protected mode for access to memory beyond 1MB. Borrowing from mainframe technology, the 386 was the first to introduce the concept of *instruction pipelining*, also known as *scalar* architecture, which allows the CPU to start working on a new instruction before completing the current one. At long last, the desktop computer could walk and chew gum at the same time! With the invention of Windows a few years later, this new ability evolved into today's multitasking — the computer's capacity to perform several unrelated jobs or run several different programs at the same time.

Over the next few years, Intel released several versions of the 386 microprocessor, most of which differed only in speed. In 1988 Intel introduced the 386SX, which was the 386 equivalent of the 8088. Built around a 32-bit core, the 386SX used multiplexing to reduce the data bus to 16-bits. Like the 8088, it took two passes at the input data before the instruction command could be executed. While the performance of the 386SX was significantly lower than the 386, so was its price. It was, finally, a 32-bit system that John Q. Public could afford. Another version of the 386, the 386SL, came in 1990. Basically, the 386SL is identical to the 386SX, but it also includes power-management circuitry that optimizes the device for use in portable computers — a first for Intel and the industry.

Meanwhile, Motorola was busy grooming its 32-bit microprocessor, the 68020, which came fashionably late to the 32-bit party in the summer of 1985. Like the 386, the 68020 could directly address 4GB of memory. But Motorola upstaged Intel by incorporating a built-in memory *cache* of 256 bytes, a small amount of extremely fast memory that contains the last-used instructions for quick access. A good analogy is your kitchen cupboard. Let's say that you just used the bottle of soy sauce and put it on the counter where you're working within easy reach — this is a cache. Now if the bottle isn't within easy reach, you go the cupboard and start searching for it, which takes more time. This is the same as searching through the motherboard's random-access memory (RAM). If it turns out that you are out of soy sauce, then you have to hop in the car and go to the supermarket. This is the same as going to the hard disk for information. Before the

development of the 68020, only big mainframe computers had cache capacity — desktops didn't.

Shortly after the announcement of the 68020 came the 68030, which had an additional 256-byte cache for data — for a total of 512 bytes (0.5K). While small by today's standards, it was the first time caching was introduced into the world of desktop computing as a speed enhancement. For Motorola to put the cache on the chip itself as a *primary cache*, as opposed to *secondary cache* (which is placed on the motherboard) was unprecedented and a notable landmark. In fact, it took Intel four years to catch up to the new technology.

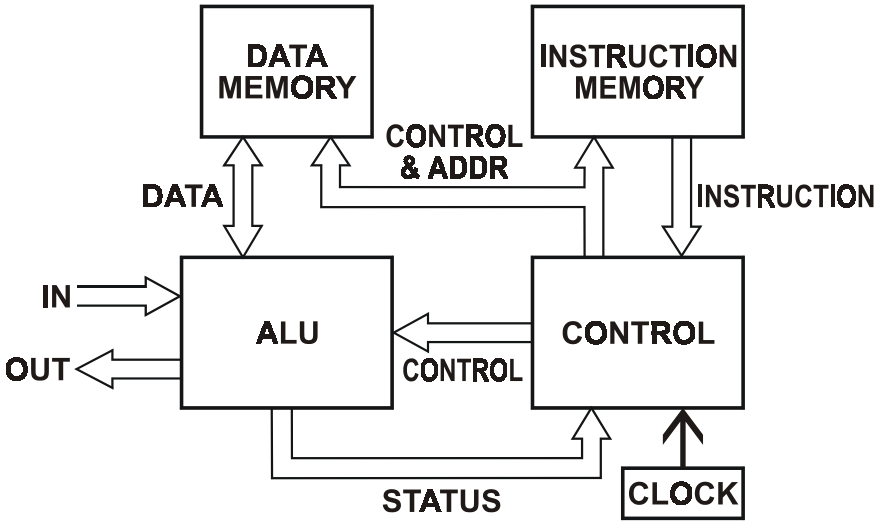
The Intel 486 Family

When Intel finally did catch up, it did so in a big way with the release of the 80486 processor, better known as the 486, in 1989. In addition to adding an 8K primary cache, Intel decided to integrate a floating-point math coprocessor (FPU) to an improved 386 core with scalar architecture, and a full 32-bit data and address bus width.
Because of the high transistor count, which numbered 1.2 million, the infant mortality rate was high. At first, only 30 percent of the chips survived the testing process from start to finish. Most of the failures occurred in the math coprocessor section, which occupied about two-third of the chip's real estate.

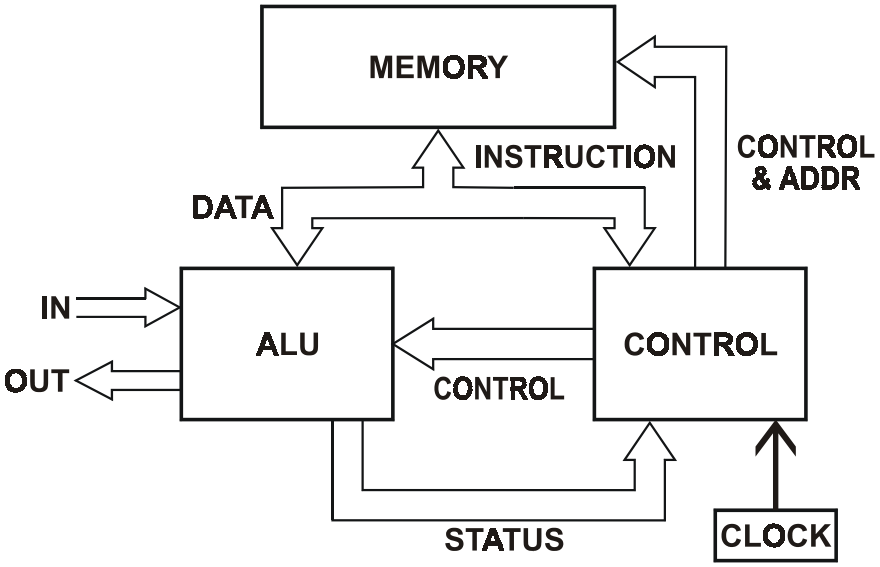
So it took no time at all for Intel to switch to testing its 486 chips for CPU performance first, and follow up with a math coprocessor check. Those chips that passed the first phase but flunked the math were labeled 486SX, and went into lower-priced, conventional desktops without math processors. Those chips that passed their math tests went into a higher-priced model, now called the 486DX. As yields improved, and demand for the lower-priced 486SX increased, the 486SX selection process was winnowed down to testing the CPU only, with a subsequent blowing of the fuse that fed power to the math coprocessor (just in case the math coprocessor was functional, but flawed).

Another important feature of the 486 line was the introduction of 3.3-volt technology — something that Motorola couldn't match until two years later. Up to 1990 microprocessor logic was based on a 5-volt power supply. However, the amount of heat a semiconductor generates is a function of speed multiplied by voltage. By 1990, the speed of computer chips (both microprocessors and external logic chips) hit a heat barrier — if they went any faster, they'd simply burn up. Reducing the voltage reduced the heat build-up and let the chips run faster. By the year 2000, it's expected computer chips will run at 500 MHz using 0.9-volt power sources.

What was Motorola doing all this time? It was busy working on the 68060, a third generation 68000 chip that introduced the concept of *superscalar* pipelining, a technique again borrowed from mainframe technology, which permits multiple instructions to run at the same time. This chip saw the light of day in early 1994. Motorola was also busy developing a line of microcontroller chips, like the 68HC11, which we'll talk about at length starting with Lesson Two.



Harvard Architecture Microprocessor



Princeton Architecture Microprocessor

Hitting the Peak at 64-bits

Which brings us up to date with the events as of 1995: the introduction of the 64-bit microprocessor. Despite delays, Intel was the first to market a 64-bit CPU with the announcement of the Pentium in 1993. Why did Intel call it the Pentium instead of the 586, as one would expect? Because a court ruled that the 586 moniker couldn't be trademarked. (So much for overpaid corporate lawyers.) While the Pentium's data bus is a full 64-bits wide, Intel decided to retain the 32-bit address bus of the 486 and a 4GB memory. All versions of the Pentium have an on-board math coprocessor and 16K of primary cache — 8K for instructions

and 8K for data. The Pentium also sports a superscalar pipeline. While the fastest Pentium sold today runs at 100 MHz, there's a 150-MHz version lurking in the wings, just waiting for its debut.
At about the same time, Motorola conceded that after more than a decade of development, its 68000 architecture had run out of steam. Instead of continuing as a solo act, though, Motorola decided to hook up with IBM and Apple in a three-way consortium to meet the enormous demands of the new 64-bit market. The result was the PowerPC, a 64-bit superscalar CPU that can effectively execute up to three instructions at the same time, compared to the Pentium's two instructions. Like the Pentium, the PowerPC can address up to 4GB of memory, and has a built-in math coprocessor. A pleasant surprise is 32K of cache memory, double that of the Pentium.

The PowerPC is the first desktop microprocessor to implement RISC (Reduced Instruction Set Computing), a type of instruction code first used by IBM in its high-end workstations. With RISC, most instructions execute in only one clock cycle (one clock cycle is equal to one hertz), and instructions can even be completed out of order. By comparison, all Intel processors, including the Pentium, use CISC (Complex Instruction Set Computing) instructions. While CISC instructions are longer and can do in one statement what it takes RISC 100 instructions to accomplish, RISC runs a lot faster. The drawback is that RISC processors require special coding, which means the enormous base of Intel software can't run on the PowerPC without translation, and the time it takes to translate the code slows the PowerPC to a crawl — literally. Performance is about the same as an old 386 system. This potentially-perilous battle between the Pentium and the PowerPC chips is reminiscent of the Betamax-VHS videotape tug-of-war, which left bloodied losers on all sides (especially the consumers who bet on the Beta). Hopefully, Apple and IBM working together will command enough clout that this market won't be ignored, but will inspire ample software written specifically for the PowerPC chip as it is for the Intel chips of the present.

Lesson 1 Questions

1) What is a microprocessor?

2) What is the difference between a microprocessor and a microcontroller?

3) Name three or more household appliances that contain microcontrollers, and explain the microcontroller's function in each product.

4) What company manufactures the MC68HC11 microcontroller?

5) What is the difference between an analog and a digital device?

6) What device replaced vacuum tubes and transistors in computers?

7) What is the most important difference between a calculator and a computer?

8) Can you think of other common examples (besides the ones given in this chapter) of machine reasoning/computer logic?

9) What is a hertz (when it's not a rent-a-car)?

10) Convert the number 50 into binary format.

11) How many bits are in a byte? How many bytes are in a kilobyte?

12) What is a computer bus?

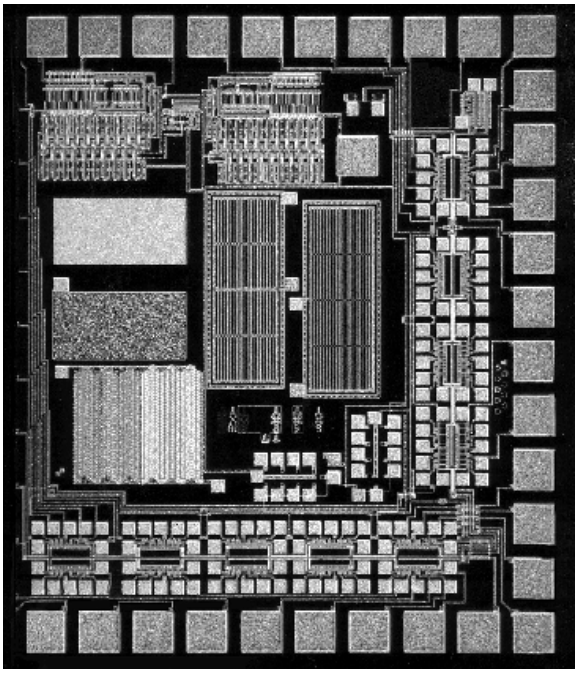
13) What is the difference between 8-bit, 16-bit, and 32-bit microprocessors?

14) How does multiplexing work, and what are the advantages of using multiplexing?

15) What advantages do computers gain from scalar and superscalar architecture?

16) What is cache memory? How does it improve microcontroller performance?

17) Why are computer manufacturers eager to lower the voltage of the power supplies in their microprocessors?



Lesson Two

The MC68HC11 Architecture & its Addressing Modes

1. The internal structure of the MC68HC11.
2. How data is processed inside the MC68HC11.
3. The different memory types, their speed and relative prices.
4. The addressing modes of the MC68HC11.
5. How to move instructions and data to and from the MC68HC11.
6. How to use interrupts.
7. How the built-in analog-to-digital converter works, and how to use it.

Because many colleges and trade schools now require at least one microcomputer course that includes hands-on laboratory work programming and using microcontrollers, EZ-Courseware includes a *microcontroller development system* based on Motorola's very popular — and very powerful — 68HC11 microcontroller chip.

Essentially, a microcontroller development system is a complete printed-circuit assembly that contains all the basic hardware elements of a generic microcontroller circuit. When you use one of these boards, much of your circuit design and testing has been done for you. All you need to add are the sensors and switches specific to your application, and the programming. In fact, many microcontroller-based products are made using an off-the-shelf microcontroller development system built around a standard microprocessor, like the 68HC11.

This lesson contains details of the 68HC11's internal subsystems and functions. While the information in this lesson is much more detailed than would usually be required for normal use of the microcontroller, a user who is familiar with the detailed operation of the part is more likely to find a solution to an unexpected system problem. Often a trick based on software or MCU resources can be used rather than adding expensive external circuitry. This is particularly true of the 68HC11 family of MCUs, which includes many built-in hardware features like timers and I/O port translators.

68HC11 Overview

The 68HC11 is a family of 8-bit microcontrollers fabricated using High-Density Complementary Metal-Oxide Semiconductor (HCMOS) technology. HCMOS provides smaller size and higher speeds than standard CMOS, yet retains CMOS's high noise immunity and low power consumption. In addition, the technology is a fully static design which allows for operations from 3 MHz down to dc that further reduces power usage, making it ideal for use in battery-operated equipment. Several of the 68HC11 MCUs have a low-voltage counterpart, using the 68L11 designation, that can operate over a voltage range of 3.0 to 5.5 volts.

All members of the 68HC11 family are software compatible. They differ only in their hardware features, mostly in the size and type of the built-in memory and the input/output functions. Table 2-1 lists the different versions of the 68HC11 and the unique features of each one.

Part Number	RAM	ROM	EPROM	EEPROM	A/D	I/O	3-Volt Version
68HC11A0	256	0	0	0		22	Yes
68HC11A1	256	0	0	512	Yes	22	Yes
68HC11A7	256	8K	0	0		38	Yes
68HC11A8	256	8K	0	512		38	Yes
68HC11C0	256	0	0	0	Yes	35	No
68HC11D0	192	0	0	0		14	Yes
68HC11D3	192	4K	0	0	No	32	Yes
68HC711D3	256	0	4K	0		32	Yes
68HC11E0	512	0	0	0		22	Yes
68HC11E1	512	0	0	512		22	Yes
68HC11E8	512	12K	0	0		38	Yes
68HC11E9	512	12K	0	512	Yes	38	Yes
68HC711E9	512	0	12K	512		38	No
68HC811E2	256	0	0	2K		38	No
68HC11E20	768	20K	0	512		38	No
68HC11F1	1K	0	0	512	Yes	30	Yes
68HC11G5	512	16K	0	0		66	No
68HC711G5	512	0	16K	0	10-bit	66	No
68HC11G7	512	24K	0	0		66	No
68HC11K0	768	0	0	0		37	Yes
68HC11K1	768	0	0	640		37	Yes
68HC11K3	768	24K	0	0	Yes	62	Yes
68HC11K4	768	24K	0	640		62	Yes
68HC711K4	768	0	24K	640		62	No
68HC11KA0	768	0	0	0		32	Yes
68HC11KA1	768	0	0	640		32	Yes
68HC11KA2	1K	32K	0	640		51	No
68HC711KA2	1K	0	32K	640	Yes	51	No
68HC11KA3	768	24K	0	0		51	Yes
68HC11KA4	768	24K	0	640		51	Yes
68HC711KA4	768	0	24K	640		51	No
68HC11L0	512	0	0	0		30	Yes
68HC11L1	512	0	0	512		30	Yes
68HC11L5	512	16K	0	0	Yes	46	Yes
68HC11L6	512	16K	0	512		46	Yes
68HC711L6	512	0	16K	512		46	Yes
68HC11P2	1K	32K	0	640	Yes	62	No
68HC711P2	1K	0	32K	640		62	No

Table 2-1. MC68HC11 Family Members.

The alphanumeric code for this family of MCUs is shown in Figure 2-1. When a feature applies to all family members, the general coding sequence of MC68HC11 is used, although this is frequently shortened to just 68HC11. When talking about a specific part, the last two characters define the part's features and the amount of built-in memory. For example, A0 tells us that the device has 22 I/O lines with 256 bytes of Random Access Memory (RAM); an A1 designation adds 512 bytes of Electrically Erasable Programmable Read-Only Memory (EEPROM) to the chip.

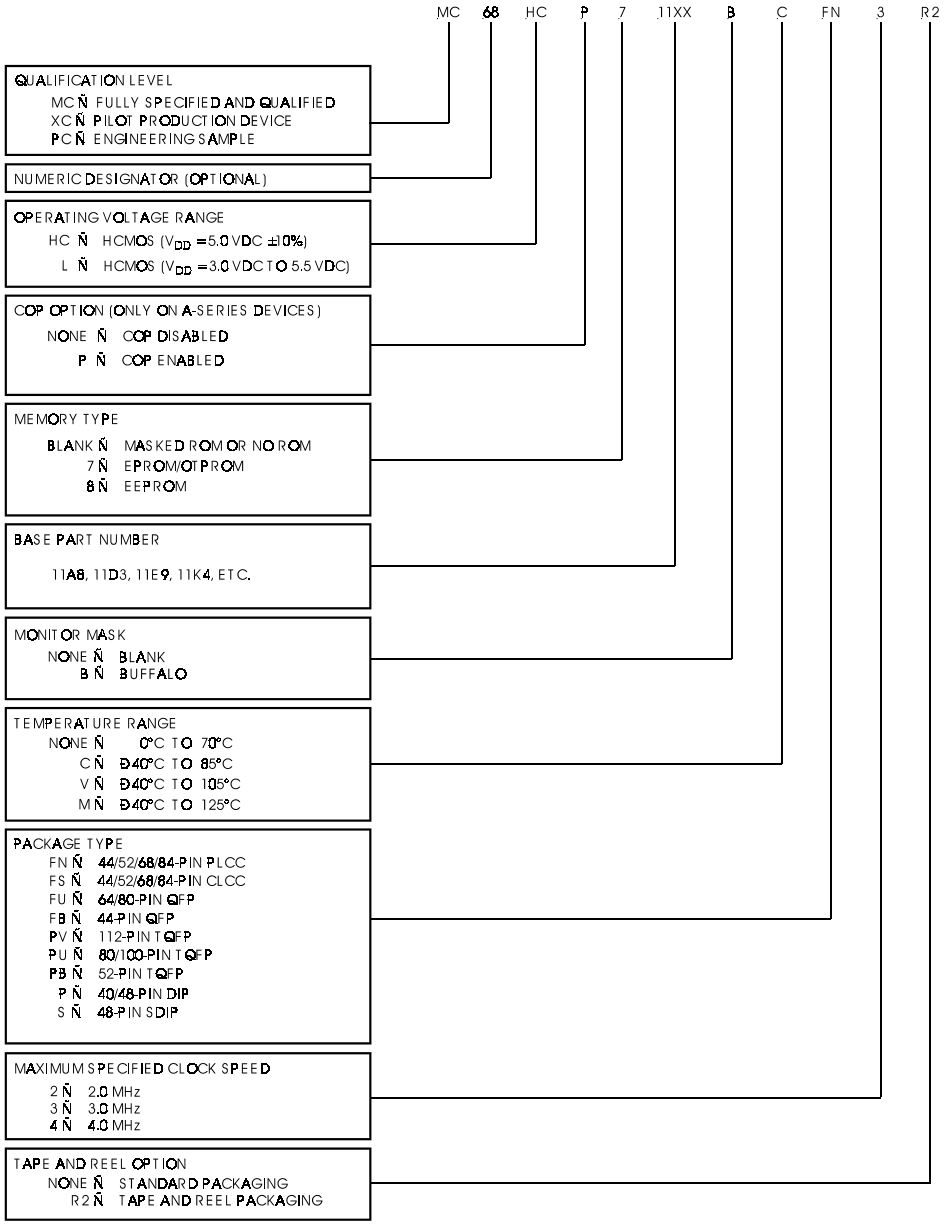


Figure 2-1 MC68HC11 Part Numbering

Inside The 68HC11

The basic microcontroller is the 68HC11DO. Figure 2-2 is a block diagram of the 68HC11A8. This MCU has 8K of Read-Only Memory (ROM), 512 bytes of EEPROM, and 256 bytes of RAM. Also built into the MCU is an eight-channel analog-to-digital (A/D) converter, asynchronous and synchronous serial communications ports, and two general-purpose I/O ports.

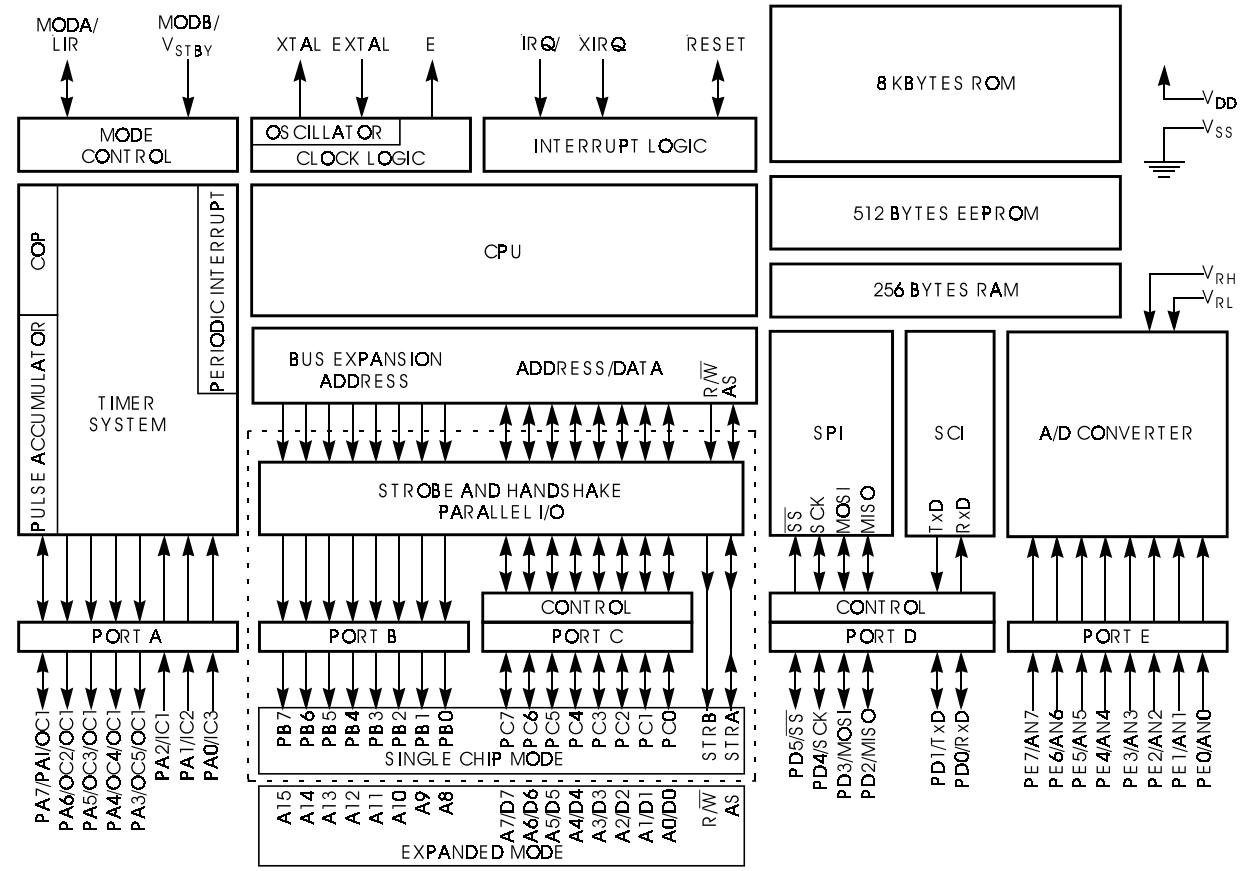


Figure 2-2 68HC11E0 Block Diagram

The main 16-bit, free-running timer has three input-capture lines, five output-compare lines, and a real-time interrupt function. An 8-bit pulse accumulator can count external events or measure external time periods. A watchdog circuit protects against software failures by automatically resetting the MCU in case the clock is lost or runs too slow. And there are 40 I/O pins that link you to the real-time outside world.

Don't let the above terms frighten you. They simply describe the power of the 68HC11 in a nutshell. In the following sections of this lesson, the building blocks of the 68HC11 will reveal themselves as we examine each in detail.

Section One: The Central Processing Unit

At the heart of the 68HC11 is the central processing unit, or CPU, which is responsible for executing all software instructions in their programmed sequence. In addition to running all 6800 and 6801 instructions (the microprocessor used in the first Macintosh computers), the 68HC11 instruction set includes 91 new *opcodes*. An opcode is a mnemonic word, like BSET, that tells the CPU what to do.

Registers

Central to the CPU's operation are internal registers — a special type of imbedded memory that can manipulate operation instructions and temporarily store data. The CPU registers are an integral part of the CPU and are not accessed in the same way as the other registers in the 68HC11. Most of the time these registers are accessible only through opcodes. The 68HC11 CPU contains sevens registers, which are shown in Figure 2-3. The function of each of the CPU registers is discussed in the following sections.

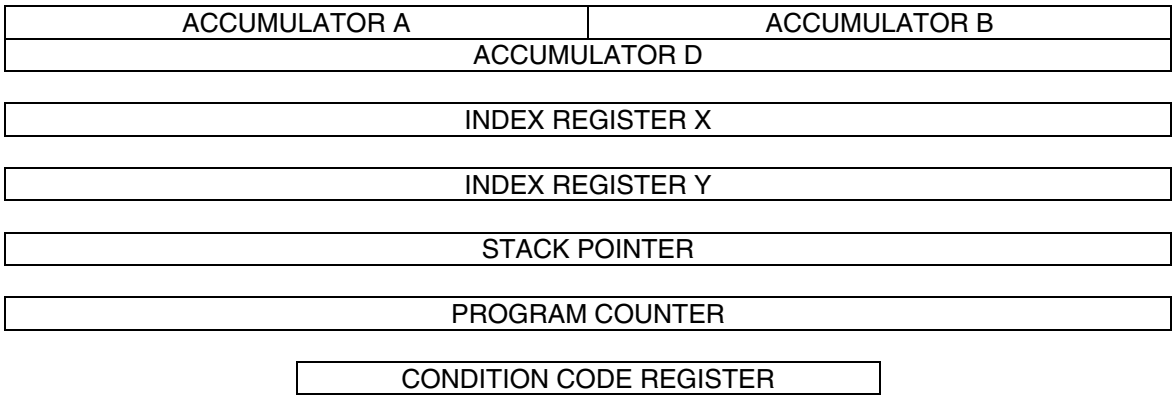


Figure 2-3. MC68HC11A8 CPU registers

Accumulators (A and B)

Any general-purpose register that holds a data number is called an *accumulator*. The 68HC11 CPU has two 8-bit accumulator registers, labeled A and B. The data number in an accumulator may be the result of an instruction — for example, the sum or two numbers — or used as a staging area to hold pending opcodes.

While most operations can use accumulator A or B interchangeably, there are some operations that use only accumulator A or accumulator B. They include:

1. If you want to perform decimal arithmetic, you must use accumulator A.
2. The contents of the condition code register (CCR, explained later) can only be moved to accumulator A, or from accumulator A to the CCR.
3. The ABX and ABY instructions add the contents of accumulator B to one of the two index registers; there is no equivalent instruction for accumulator A.

4. The add, subtract, and compare instructions involving both accumulators A and B (ABA, SBA. and CBA) only operate in one direction; therefore, it is important to plan ahead so the correct operand ends up in the correct accumulator.

Although the 68HC11 is an 8-bit device, there are several instructions that permit 16-bit operations. These instructions use both accumulators A and B as a single 16-bit register known as accumulator D. As you will see in Lesson 3, a 16-bit accumulator saves programming steps when binary values greater than 8-bits are being manipulated. Because accumulator D is a combination of two already-defined registers, it is not counted as a separate register.

Index Registers (X and Y)

For a software program to run, the accumulators have to fetch data bytes from somewhere. Generally, the data comes from one of the built-in memory sources, such as RAM. However, two methods require the use of CPU registers to locate data.

The two 16-bit *index registers*, X and Y, are used for the indexed addressing mode (explained in Section Two below). These registers contain the memory addresses of data to be used for the next operation or operations. The 68HC11 family is unique in that it has two index registers; its predecessors had only one. The second index register is especially useful for moves and in cases where operands from two separate tables are involved in a calculation — like choosing from column A and column B in a Chinese menu. In the earlier 6800 CPUs, the programmer had to store the index to some temporary location so the second index value could be loaded into the index register, which is like having column A and column B on separate menus on separate tables, and you running between the two tables to place an order. Obviously, two index registers are faster than one.

Stack Pointer (SP)

Practical programs frequently need to save numbers generated by an operation for use later in the program. Since the accumulators are too small to store more than a couple bytes and are needed for more immediate operations, the logical solution is to move the data to a temporary location in the RAM main memory area. A *stack* is a means of temporarily storing data and/or return addresses when the CPU is executing a routine. With the 68HC11, the stack may be located anywhere in the 64K address space of the MCU and may be any size up to the amount of memory available in the system.

The 16-bit *stack pointer* register controls the operation of the stack. The number in the stack pointer is a memory address of where the stack begins. Normally, the stack pointer register is initialized by one of the very first instructions in an application program. Each time a new byte is added to the stack, which is called a *push* operation, the stack pointer number is automatically reduced by one, and each time a byte of data is retrieved from the stack, called a *pop*, the stack pointer is automatically incremented by one. Stacks are usually last-in-first-out (LIFO) structures, which means the last element pushed onto the stack is the first one popped from it.

Program Counter (PC)

The *program counter* is a 16-bit register that holds the address of the next instruction to be executed. Despite its name, the program counter is not a counter and it does not count programs; it is an address pointer. This register is linked to the transparent instruction register, which holds the next instruction. The instruction register is said to be transparent because its operation cannot be seen. Software instructions cannot control the instruction register, which is why it doesn't show on 68HC11 register diagram in Figure 2-3. It is deeply embedded in the CPU, and controlled exclusively by the CPU.

Condition Code Register (CCR)

The last CPU register is an 8-bit *conditional code register* that reports CPU status following an operation and sets conditional configurations for the CPU and other hardware. This register contains five status bits, two interrupt masking bits, and a stop disable bit. These bits are defined in Figure 2-4.

Bit	Flag Name	Operation
0	C	Carry from Most-Significant Bit (MSB)
1	V	Two's complement overflow error
2	Z	Zero result
3	N	Negative result
4	I	IRQ interrupt mask
5	H	Half-carry from bit 3
6	X	XIRQ interrupt mask
7	S	Stop disable

Figure 2-4. Condition Code Register bit identifiers

The five status bits, also called status flags, reflect the results of arithmetic or Boolean logic operation. The stop bit is used to enable or disable the stop instruction. Some programmers consider the stop bit dangerous because it stalls the oscillator, which can lead to a system crash if you are not careful.

The interrupt masks are used to disable all or selected CPU and MCU interrupts. Like the instruction register, most of the 68HC11's interrupts are invisible to the user. In the 68HC11, interrupts are signaled by flags. Once during each machine cycle the interrupt flags are checked to see if any are set. A set flag indicates an event has occurred that needs immediate attention. If no flags are set, the CPU takes no action and proceeds to the next instruction. If a flag is set, the CPU may act upon it immediately using built-in firmware or notify the software of the situation via the condition code register. A detailed analysis of 68HC11 interrupts is given in Section Four of this lesson.

Memory

As important as the CPU is to a MCU, it would not be a microcontroller without on-chip memory. Onboard memory is what separates a microprocessor, like Intel's Pentium chip, from a microcontroller: microcontrollers have built-in memory, and microprocessors don't. Computer memory falls into four major categories:

1. **Read/write memory**, which allows the CPU to both write numbers into storage and read them back later.
2. **Read-only memory**, which holds hold numbers in storage that cannot be altered — the numbers are permanent.
3. **Volatile memory**, which loses the information stored in its memory banks when electrical power is removed.
4. **Nonvolatile memory**, which uses a storage technology that does not require electrical power to retain information.

The type of memory that is appropriate for any particular application depends on the application and where you are in the design cycle of the product. The 68HC11 supports four different types of memory:

1. RAM (random access memory)

1. RAM (random access memory)

2. ROM (read-only memory)

3. EPROM (erasable programmable read-only memory)

4. EEPROM (electrically erasable programmable read-only memory)

The size and type of memory included in any particular 68HC11 chip varies. For example, the 68HC11A8 has 256 bytes of RAM, whereas other members of the 68HC11 family have RAM that ranges in size from 192 bytes to 1K. Some support ROM and EEPROM, others don't. Refer to Table 2-1 for details.

RAM

Random access memory is included in all 68HC11 MCUs. Unlike the other 68HC11 memory options, RAM is the only memory that is fast enough for real-time read/write operations. Generally, RAM is volatile memory because it needs periodic refreshing to keep the data alive. While still volatile — in other words, if you loose power, you loose everything in RAM — 68HC11 RAM memory is *static*. Which means if the clock stalls your data isn't lost. As soon as the clock starts up again, the program picks up where it left off. The RAM memory in desktop computers is *dynamic* memory, which looses its data when the clock stops.

As mentioned, the size of the RAM varies according to 68HC11 device type. The more RAM you have, the more items you can juggle at the same time — but the more the MCU will cost. So there is a tradeoff here, and finding the right size RAM for your application may take some serious forethought on how compactly the program code can be written, or how you can trim the end-product's features to meet the target price.

ROM

The primary purpose of the read-only memory is to hold the user's application program, which ideally holds all coding so that the only external input comes from an I/O port device, and not the user. While not as fast as RAM, ROM is cheap, which means you can afford to buy a lot of it. A 68HC11AK2 can support up to 32K of ROM, with the 68HC11A8 sporting 8K of ROM.

Unlike RAM, once a software routine is programmed into a 68HC11 ROM, it is forever carved in stone. There is nothing you can do to change it. Your program codes are placed into the MCU's ROM at the time the MCU is fabricated using *masks* that produce a memory matrix pattern that clones your program on the chip as it comes off the assembly line. While the cost of making the mask is high initially, if thousands of ICs with the same code are manufactured, the cost of the masks is inconsequential. A typical application for a masked ROM is a high-volume product containing a MCU, like those used in microwave ovens, cassette players, and other consumer products.

EPROM

When only a few devices are required for low-volume products or lab development, the cost of a masked ROM is prohibitive. A less expensive and more flexible alternative is to substitute the ROM with an EPROM. Virtually all versions of the 68HC11 have an EPROM equivalent. The cost of the MCU is slightly higher, but you save the expense of making a ROM mask.

EPROM-based MCUs are more flexible because it's easier to make changes in the design. For example, let's say an unexpected bug occurs in your original software model six months after the product has been shipping. If the MCU is ROM-based, this means you have to go through the expense of creating a new ROM mask. With EPROM MCUs, all you have to do is make changes in your software and start programming the new devices with the fixed code. Better yet, it is possible that the defective products already shipped can be fixed in the field by reprogramming the old MCUs, if you choose the right EPROM part.

The EPROM works on the principle of a charged capacitor. Essentially, a capacitor is used to turn a transistor on or off, depending on whether the capacitor is charged or not. If the transistor is turned on, it represent a binary one; turned off, it represents a zero. Once charged, the capacitor will retain its charge for many years. The thousands of capacitors inside the EPROM are properly charged using a PROM programming device. (You will learn how to use one of these programming machines later in this course.)

Discharging the capacitors erases the EPROM's memory. This is done by shining an ultraviolet (UV) light on the top of the chip, which causes the charge to leak off the capacitors. Erasing the EPROM requires a special, intense UV light source of a certain color, and the process generally takes 15 minutes and longer. This is done on purpose to prevent the EPROM from accidentally being erased by exposure to ambient room light or sunlight. Once erased, the EPROM can be reprogrammed. This lets you fix a bug in existing products, or permits you to use the MCU for another application.
For it to work, though, the EPROM has to be erasable, which means the MCU has to come in a ceramic package with a quartz window built into its top. This packaging is more expensive than

plastic packages, which are not erasable. In applications where it is desirable to program the memory during assembly and where erasing is unnecessary, you can save money by buying an EPROM MCU in a plastic package.

Unfortunately, like sunburns, constant exposure to intense UV light takes its toll. Most EPROMs can only be erased and programmed about 100 times before they wear out and fail. Another disadvantage of EPROM is it must be removed from the circuit for erasing and programming. This means the MCU has to be socketed to allow removal. Sockets are costly and less reliable than soldering.

EEPROM

Electrically erasable programmable read-only memory gives you the best of both worlds — easily erasable data with in-circuit programming. EEPROMs are ideal for use in prototype devices and during the development (R&D) stage of a product because the memory can be changed over and over again with little effort.

The storage mechanism in the EEPROM is the same as for the EPROM: an electrically-charged capacitor. The difference between the two is that the EEPROM has an electrical erasing circuit built into it; it does not need a UV window. The MCU controls the erasing hardware through software. Moreover, the erasing procedure takes but a few milliseconds instead of minutes. Better yet, the typical lifetime of an EEPROM is many thousands of cycles.
Despite the ability to erase and write numbers electrically, though, EEPROM memory is not read/write. The write time is much too long for real-time read/write, which needs nanosecond response times. Like the EPROM, it is nonvolatile read-only memory.

Flash Memory

While it is not a 68HC11 option, you've probably heard of flash memory, the newest type of nonvolatile user-programmable memory. The reason for bringing this up is because some members of Motorola's 68HC16 16-bit family of microcontrollers have on-chip flash EPROM. And it is conceivable that flash memory may appear in a future version of a 68HC11 MCU because the technology can pack a lot of memory in a very little space.

Moreover, not all of your designs will be limited solely to the memory on the MCU. There will be plenty of times when you need to add outboard memory. In many circuits, flash EPROM can replace UV-erasable EPROM and EEPROM for nonvolatile storage of programs and data — something to keep in mind, considering the flexibility and low price of flash memory.

"Flash" describes the ability to erase an entire array of memory at once, or "in a flash." Well, almost in a flash. Both RAM and EEPROM are faster. Because the flash EPROM is electrically erasable, it can be erased and programmed in-circuit, eliminating the need to have a socket.

Again, the flash memory's storage technology is based on that of a charged capacitor driving a transistor. Like UV-erasable memory, flash memory must be erased before it can be programmed. On-the-fly overwrite, like RAM and EEPROM supports, is not permitted. With some flash EPROMs, you erase the entire device at once. Others divide the memory into several blocks, each of which is individually erasable.

Unfortunately, the flash EPROM has an erase speed of about 1 second, far too slow for real-time read/write performance. Their life span is about 10,000 cycles, the same as an EEPROM. And they are more expensive than UV-erasable EPROMs, but less expensive than EEPROM. In other words, flash EPROM fills the price/performance gap between cheap UV-erasable EPROM and expensive EEPROM. See the comparison chart in Table 2-2 for more details.

Memory Type	RAM	EPROM	Flash EPROM	EEPROM
Erase Time	0	15 min	1 sec	0
Programming Time	2 us	4 sec	0.5 sec	2.5 sec
Programming Voltage	none	12 V	12 V	none
Program/Erase Cycles	unlimited	<1000	>10,000	>10,000
Erasure Method	electrical	UV light	electrical	electrical
Block Erasable	Yes	No	Yes	Yes
Byte Erasable	Yes	No	No	Yes
Comparative Cost	n/a	4 - 7	8 - 15	15 - 50

Table 2-2. Memory type comparison table

CONFIG Register

The CONFIG register is an unusual control register used to enable or disable ROM, EEPROM, and the EEPROM security feature of the 68HC11. Unlike ordinary control registers, CONFIG is an EEPROM register that retains its contents after all power is removed from the chip (e.g., when shipped from the Motorola factory). However, the contents of this register can be altered to meet the demands of your the end-user customer, making it a powerful tool for custom designs. You will hear more about this register as you proceed through this lesson.

Section Two: Addressing Modes

The CPU is not the only section of the MCU to use registers. There are many more registers spread out among the many MCU functions. All registers outside of the CPU section are associated with input/output operations or the timer unit, and are addressable by the CPU.

Unlike the CPU, though, these registers cannot be accessed via opcodes. Instead, they are located in the 68HC11's memory — be it RAM, ROM, EEPROM, or any combination of the three — along with the software application data. Because these registers reside in the main memory section of the MCU, they are located at protected addresses within the 64K address range of the CPU — reserved memory locations that are off limits to anything but opcodes and instructions. Accidentally writing application data in one of these addresses will crash the system faster than you can say "oops."

In the 68HC11 family of MCUs, memory address locations between hexadecimal 0000 through 003F are reserved for the microcontroller's I/O registers (see Table 2-3 for a copy of the 68HC11 memory map). This method of having I/O registers as easily addressable as data memory locations is called *memory mapped I/O*.

Registers (1 of 2)

(The register block can be remapped to any 4K boundary)

	Bit 7	6	5	4	3	2	1	Bit 0	
$0000	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	PORTA
$0001									Reserved
$0002			CWOM						PIOC
$0003	PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0	PORTC
$0004	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0	PORTB
$0005									Reserved
$0006	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
$0007	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	DDRC
$0008	PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0	PORTD
$0009	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	DDRD
$000A									Reserved
$000B	FOC1	FOC2	FOC3	FOC4	FOC5	0	0	0	CFORC
$000C	OC1M7	OC1M6	OC1M5	OC1M4	OC1M3	0	0	0	OC1M
$000D	OC1D7	OC1D6	OC1D5	OC1D4	OC1D3	0	0	0	OC1D
$000E	Bit 15	14	13	12	11	10	9	Bit 8	TCNT (High)
$000F	Bit 7	6	5	4	3	2	1	Bit 0	TCNT (Low)
$0010	Bit 15	14	13	12	11	10	9	Bit 8	TIC1 (High)
$0011	Bit 7	6	5	4	3	2	1	Bit 0	TIC1 (Low)
$0012	Bit 15	14	13	12	11	10	9	Bit 8	TIC2 (High)
$0013	Bit 7	6	5	4	3	2	1	Bit 0	TIC2 (Low)
$0014	Bit 15	14	13	12	11	10	9	Bit 8	TIC3 (High)
$0015	Bit 7	6	5	4	3	2	1	Bit 0	TIC3 (Low)
$0016	Bit 15	14	13	12	11	10	9	Bit 8	TOC1(High)
$0017	Bit 7	6	5	4	3	2	1	Bit 0	TOC1 (Low)
$0018	Bit 15	14	13	12	11	10	9	Bit 8	TOC2 (High)
$0019	Bit 7	6	5	4	3	2	1	Bit 0	TOC2 (Low)
$001A	Bit 15	14	13	12	11	10	9	Bit 8	TOC3 (High)
$001B	Bit 7	6	5	4	3	2	1	Bit 0	TOC3 (Low)
$001C	Bit 15	14	13	12	11	10	9	Bit 8	TOC4 (High)
$001D	Bit 7	6	5	4	3	2	1	Bit 0	TOC4 (Low)

Table 2-3 68HC11 memory map (1 of 2)

Registers (2 of 2)

	Bit 7	6	5	4	3	2	1	Bit 0	
$001E	Bit 15	14	13	12	11	10	9	Bit 8	TI4O5 (High)
$001F	Bit 7	6	5	4	3	2	1	Bit 0	TI4O5 (Low)
$0020	OM2	OL2	OM3	OL3	OM4	OL4	OM5	OL5	TCTL1
$0021	EDG4B	EDG4A	EDG1B	EDG1A	EDG2B	EDG2A	EDG3B	EDG3A	TCTL2
$0022	OC1I	OC2I	OC3I	OC4I	I4O5I	IC1I	IC2I	IC3I	TMSK1
$0023	OC1F	OC2F	OC3F	OC4F	I4O5F	IC1F	IC2F	IC3F	TFLG1
$0024	TOI	RTII	PAOVI	PAII	0	0	PR1	PR0	TMSK2
$0025	TOF	RTIF	PAOVF	PAIF	0	0	0	0	TFLG2
$0026	DDRA7	PAEN	PAMOD	PEDGE	DDRA3	I4/O5	RTR1	RTR0	PACTL
$0027	Bit 7	6	5	4	3	2	1	Bit 0	PACNT
$0028	SPIE	SPE	DWOM	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
$0029	SPIF	WCOL	0	MODF	0	0	0	0	SPSR
$002A	Bit 7	6	5	4	3	2	1	Bit 0	SPDR
$002B	TCLR	0	SCP1	SCP0	RCKB	SCR2	SCR1	SCR0	BAUD
$002C	R8	T8	0	M	WAKE	0	0	0	SCCR1
$002D	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	SCCR2
$002E	TDRE	TC	RDRF	IDLE	OR	NF	FE	0	SCSR
$002F	R7/T7	R6/T6	R5/T5	R4/T4	R3/T3	R2/T2	R1/T1	R0/T0	SCDR
$0030									Reserved
to									
$0038									Reserved
$0039	0	0	IRQE	DLY	CME	0	CR1	CR0	OPTION
$003A	Bit 7	6	5	4	3	2	1	Bit 0	COPRST
$003B									Reserved
$003C	RBOOT	SMOD	MDA	IRVNE	PSEL3	PSEL2	PSEL1	PSEL0	HPRIO
$003D	RAM3	RAM2	RAM1	RAM0	REG3	REG2	REG1	REG0	INIT
$003E	TILOP	0	OCCR	CBYP	DISR	FCM	FCOP	0	TEST1
$003F	0	0	0	0	0	NOCOP	ROMON	0	CONFIG

Table 2-3 68HC11 memory map (2 of 2)

Addressing Modes

The 68HC11 has six addressing modes which can be used to reference and access memory:

1. **Extended**
2. **Direct**
3. **Immediate**
4. **Inherent**
5. **Indexed**
6. **Relative**

All MCU instructions consist of two parts — an opcode followed by an effective address. The effective address is where the data is located. How many steps the instructions have to go through to address and access the 68HC11's registers impacts significantly on the speed of the application, and the complexity of the programming code.

The following provides an overview of each of the six addressing mode. Complete programming details on these addressing modes will be taken up in later lessons.

Extended

The *extended address* mode is the most robust of the six. As mentioned above, the 68HC11 uses 16-bit addresses. Unlike other addressing modes, the full 16-bit address of the targeted memory area is contained in the extended addressing instruction code, which is why it is often referred to as the *absolute addressing* mode. The minimum extended address mode instruction code consists of three bytes: at least one opcode byte and the two address bytes.

Extended addressing has the advantage that it is less sensitive to address locations or hardware. The instruction works the same at any location in memory. Moreover, the data number can also be anywhere in the memory. The hardware does not care about either, because everything about extended addressing is explicit.

Direct

The *direct address* mode takes advantage of the MCU reserved memory area between hexadecimal memory addresses 0000 and 00FF. (Remember the first reserved register address does not begin until 1000). This area contains 256 bytes of RAM common to all 68HC11 MCUs (except the 68HC11D0 and 68HC11D3, which are limited to 192 bytes). This area is reserved exclusively for temporary data storage.
Because the first address byte of this memory space is always 00 (00FF being the highest memory location), addressing overhead can be reduced from two bytes to one byte. In fact, that is what direct addressing does. In the direct addressing mode, the high-order byte is assumed to be 00, so only the lower byte (00 through FF) has to be included in the instruction. This reduces the instruction from three bytes to two bytes, one for the opcode and one for the least-significant byte (LSB) address, which takes less time to execute. Moreover, it is fast RAM memory, which speeds things up even more.

Immediate

In the *immediate addressing* mode, an address is not needed. The opcode itself seeks out the register specified by the instruction, and immediately inserts the two or more bytes of data following the opcode into the specified register. While an immediate addressing instruction is three or more bytes in length, its operation is very fast because no additional clock ticks are needed to locate the data in memory and then load it into the register. Everything, opcode and data, is built into the immediate addressing instruction.

Inherent

Some opcode instructions do not need either an address or data byte. These opcodes are simply an instruction that causes an operation to take place. A perfect example is the ABA instruction, which tells the CPU to add the contents of the B accumulator to the contents of the A accumulator, then put the sum into the A accumulator. This mode of operation in called *inherent addressing*.

Indexed

The *index addressing* mode uses the X and Y index registers. These registers contain the memory address of data to be used for the next operation or operations. Indexed addressed instructions contain an opcode and an offset byte. The instruction forms the effective address by adding the offset byte to the number in the index register. This method has the advantage that the offset byte is only one byte, whereas the number contained in the index register is two bytes, which means you can point to an absolute address using only two bytes of code instead of the three bytes needed by the extended addressing mode.

Relative

Like indexed addressing, *relative addressing* uses an offset byte to locate data in memory. This time, though, the relative addressing mode is used only for branch instructions. With but a couple exceptions, relative addressed instructions require two bytes of code: an opcode and the relative offset byte. The offset byte can be any value between -128 and +127 bytes. If the branch condition is true, the offset value is added to the contents of the program counter to form the new effective branch address. There will be more on branch programming in Lesson 6.

Section Three: Input/Output Ports

The 68HC11DO has a total of 32 input/output (I/O) pins divided up into four ports, A through D. Each port has eight pins. All these pins are shared between general-purpose I/O usage and at least one other on-chip peripheral function. The four ports are clearly marked in Figure 2-2. Table 2-4 shows each ports primary and alternate function.

Port	Bits	General Purpose Use	Alternate Function
A	0 - 2	Input	Input Capture (IC)
	3 - 6	Output	Output Compare (OC)
	7	Bidirectional	Pulse accumulator
B	0 - 7	Output	Address (in the Extended Mode)
C	0 - 7	Bidirectional	Address/Data (in the Extended Mode)
D	0	Bidirectional	Rx data
	1	Bidirectional	Tx data
	2	Bidirectional	Master in/Slave out
	3	Bidirectional	Master out/Slave in
	4	Bidirectional	Serial clock
	5	Bidirectional	Slave select
	6	Bidirectional	AS
	7	Bidirectional	Read / Write

Table 2-4. Functions of the 68HC11 ports

Like the CPU, the I/O ports are controlled by internal registers. Figure 2-5 shows the registers and control bits connected with parallel input/output operations. These registers and control bits are used to specify the direction of data flow at each bidirectional port pin, as well as to let other on-chip subsystems share use of the I/O pins.

Register Address	Bit Definition							
PORTA $0000	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PIOC $0002	STAF	STAI	CWOM	HNDS	OIN	PLS	EGA	INVB
PORTC $0003	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTB $0004	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTCL *	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DDRC $0007	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTD $0008	0	0	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
DDRD $0009	0	0	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTE *	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PACTL $0026	DDRA7	PAEN	PAMOD	PEDGE	DDRA3	I4/05	RTR1	RTR0
SPCR $0028	SPIE	SPE	DWOM	MSTR	CPOL	CPHA	SPR1	SPR0

Figure 2-5. Parallel I/O registers and control bits.

* Not available on EZ-Micro MCU

Port A

Port A is a general-purpose I/O port that shares its 8 pins with the main timer system and the pulse accumulator system. The PORTA register, at hexadecimal address 0000, controls the digital input and output functions of pins PA0 through PA7 (the actual pin number depends on which package you choose; see Figure 2-6 for an example). Pins PA0, PA1, and PA2 are input only. Pins PA3 through PA6 are output only. Pin PA7 is bidirectional and its direction is determined by the data direction bit (DDRA5) in the PACTL register.

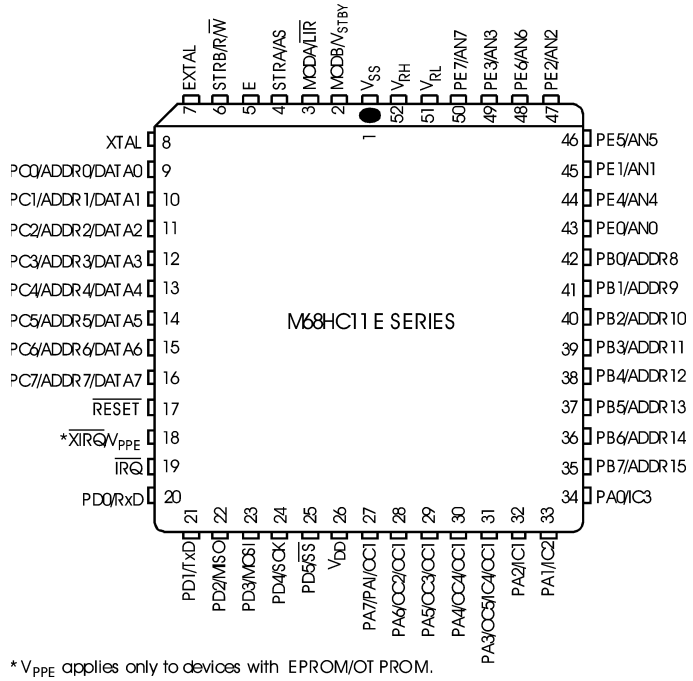


Figure 2-6 Pin Assignments for the MC68HC11E9/711E9 (52-pin LCC)

While the PORTA register may be read at any time. writing to the PORTA register controls the corresponding output pins only if they are *not* controlled by the timer. Details on port A's timing features are given in Section Four of this lesson.

Ports B and C

Port B is a general-purpose, 8-bit output-only port, whereas port C is a general-purpose 8-bit bidirectional port. Ports B and C are used for both data I/O and address I/O, depending on the operating mode of the 68HC11. When the MCU is in the single-chip mode, these 18 pins are used for general-purpose input, output, and handshaking. When the MCU is in the expanded mode, these pins are used for a multiplexed address/data bus.

The mode select pins, MODA and MODB, are used by the 68HC11 to select the operating mode during reset. After reset, the MCU no longer requires input on these pins to keep the device in the selected mode, and they may be used for their alternate function of load instruction register (LIR) indicator and the standby power (V_{STBY}) source to keep the MCU's RAM alive when main power is not present, respectively. These pins are available to you in your EZ-COURSEWARE microcontroller development system.

Single-Chip Mode Operation

In the single-chip mode, ports B and C become parallel I/O ports. Port B is an output-only port that does not have a data direction register. Port C, on the other hand, is a programmable input/output port with a data direction register named DDRC.

There are two handshaking pins associated with these ports when in the single-chip mode, named STRA and STRB. The STRA pin is an edge-detecting input that causes port C data to be latched into a special internal latch register (PORTCL) located at hexadecimal address 0005. The active edge for STRA is software selectable, making it easy to interface this pin with a wide variety of peripheral devices and timing schemes. The STRB pin is an output strobe associated with the handshake I/O functions of ports B and C to indicate valid data is available on the port B output lines.

Expanded Mode Operation

In the expanded modes, the port B and C pins are used for address/data bus to allow the CPU to access up to 64K of memory. To save pins, the address and data signals are multiplexed on eight pins. During the first half of each bus cycle, address *output* signals are present on pins A0 through A7. During the second half of each bus cycle, these same eight pins are used as a *bidirectional* data bus.

Controlling this show is the active-high latch enable AS signal and R/W signal. Address information is allowed into the external latch when AS is high, and the stable address information is latched when AS is low. The E-clock (see Section Four below) is used to synchronize external devices to drive data into the CPU during the second half of the read bus cycle. The R/W signal indicates the direction of the data: high for read, low for write.

Port D

Port D is a 6-pin bidirectional port that can be configured as either general-purpose I/O lines or as an interface to the 68HC11's internal serial communications subsystem. Officially known as the serial peripheral interface (SPI), this serial interface is primarily used to communicate with serial peripheral devices, like a telephone modem or liquid-crystal display (LCD).

Unlike the parallel port communications methods of the B and C ports, which can send or receive an 8-bit byte in a single E-clock tick, only a single bit can be sent at once through the serial port — making serial I/O communications about twelve times slower than parallel I/O communications when you count the control bits needed to keep things in sync.

68HC11 Serial Communications

The many control and data registers for the SCI are in the memory block beginning at hexadecimal address 0000. I won't try detailing all the registers and bits needed for serial communications. That is better left for another lesson. What I will do is outline the serial port and what it can do.

The SCI can operate over a wide range of baud rates. Essentially, the baud rate is equal to the number of bits per second (bps) you can push data through the two wires that make up a serial connection. The baud rates range from 75 bps up to 131, 072 bps. The 68HC11 supports both synchronous and asynchronous serial communications.

For asynchronous communications, pin PD0/RxD is used to receive serial data and pin PD1/TxD is used to transmit serial data. In asynchronous communications, the synchronizing signals are embedded into the data stream, making it the slowest form of I/O communications.

The other four lines of port D can be configured for synchronous serial communications. Basically, the two protocols differ in that synchronous handshaking signals are sent over two separate pins and the data over another two, rather than embedded in the serial data stream as in the asynchronous mode. While synchronous communications is faster than asynchronous, it requires more hardware. Virtually all telephone modems are two-wire asynchronous, whereas many factory sensors are four-wire synchronous for faster MCU response.

Port E

Port E is an 8-bit input port used for general-purpose digital inputs and/or analog voltage signals. This port supports the following inputs:

1. **Use it for all digital inputs**
2. **Use it for all analog inputs**
3. **Use some pins for digital inputs and the remaining pins for analog inputs**

These inputs are designed so that the digital input buffers are disabled at all times except for part of a cycle during an actual digital read of port E. Because of this circuitry, the analog and digital functions do not normally interfere with each other, although turning on the digital buffer during an analog sample may cause small disturbances on the input line that could lead to measurement errors. However, it is easy enough to write software that avoids the two from happening at the same time.
In the analog voltage mode, the selected Port E pins interface with the built-in analog-to-digital (A/D) converter. An A/D converter creates a binary number that is proportional to an unknown input voltage, and is normally used to read signals from analog sensors, like temperature and pressure gauges.

When using the E port for digital input, the digital input data can be found in the PORTE register located at hexadecimal memory location 000A. When using the E port for analog voltage input, the A/D control register (ADCTL), at hexadecimal memory location 0030, selects which pins are to be used for digital logic and which are allocated for analog voltage input. Refer to Section Five in this lesson for details on the A/D converter.

Section Four: Timing Systems

This section describes the timing systems of the 68HC11, which include the main timer, watchdog timer, and interrupts. When writing code, it is important to realize that it is real-world events you are controlling. Hence, you need to touch that outside world every so often to put the MCU in sync with what is happening. This is what the 68HC11 timer systems do.

Figure 2-7 is a block diagram of the main timer system. It will be helpful to refer to this figure as we go through the following discussion. Since the architecture of the main timer is primarily a software-driven system, several application examples are included. Don't be confused if the software examples are sketchy at this point. They will be discussed in detail later in this course.

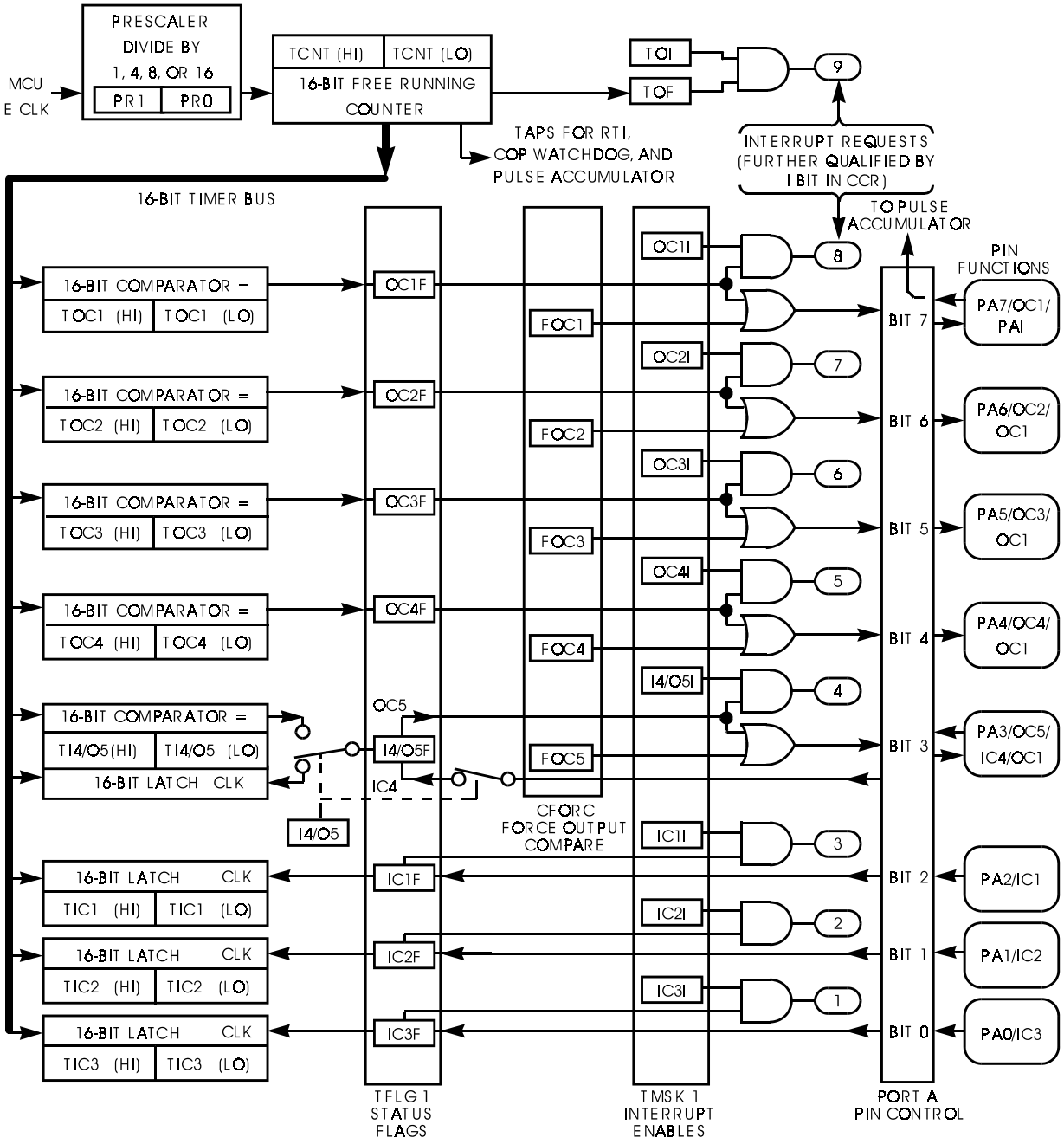


Figure 2-7 Main timer system block diagram

Main Timer

All the MCU's timing signals are derived from its E-clock, which is driven by a crystal-controlled oscillator. Typically, the oscillator runs at 8 MHz, which is reduced to 2 MHz using internal dividers. This produces a fundamental E-clock timing period of 500 nanoseconds (ns). The number of clock cycles required to do an operation is based on this timing. For example, an instruction that takes four clock cycles requires 4 x 500ns, or 2 us to complete. A few of the

68HC11 MCUs run as fast as 3 MHz, using a 12-MHz crystal, which produces a fundamental E-clock timing period of 333 ns.

The E-clock's frequency, and subsequent timing period, can be further reduced using an on-chip *prescaler* (upper left in Figure 2-7). The prescaler can divide the E-clock frequency in increments of 1, 4, 8, and 16. The prescaler is software programmable via two registers. However, the prescaler's rate can only be set once, and this must be done within the first 64 clock cycles after reset. After that, the rate remains fixed until the next reset. In many microcontroller development systems, you do not have access to the bus within the first 64 clock cycles. The prescaler default setting is one.

The prescaler feeds a 16-bit free-running counter. The counter's registers are set to zero on reset. The counter, which is the central element in the main system timer, is then incremented each time it receives a clock signal from the prescaler. Physical time is represented by the number in the counter's 16-bit register. This number is available to several circuits in the MCU, including the input-capture and output-compare functions.

Input Capture

The input-capture (IC) function is typically used to record the time at which an external event occurred by latching the contents of the counter number in one of the three IC registers. This number can be used to measure the period and/or pulse width of an input signal. Or it can be used in conjunction with an output compare function to establish a delay time between an input and output action.

Output Compare

The output-compare (OC) function is also a fundamental element of the timer architecture. Generally, output compare is used to program an action to occur at a specific time. For each of the five output-compare functions, there is a separate 16-bit compare register and a dedicated 16-bit comparator. The value in the compare register is compared to the value of the free-running counter on every bus cycle. When two values match, an output is generated, which sets an output-compare status flag and initiates the automatic actions for that output-compare function. You can use this to do anything from sounding a beeper to cycling a chemical batch process.

Another use of output compare is to generate a specific time delay. For example, to produce a 10-millisecond delay to time programming of an EEPROM byte. When used with the input-capture function, the process can be looped to produce a string of pulses of specific frequency and duty cycle.

Timer Overflow

As previously stated, the free-running timer counter starts at zero and is incremented by one with each prescaler pulse. When the count reaches hexadecimal FFFF, the counter rolls over to zero and starts over again. This is called a *timer overflow*, and when it occurs a timer overflow flag is set. This flag can be used to measure periods that are greater than the range of the timer counter can hold. This is similar to the way we use an hour to represent 60 minutes. It can also be used to generate a hardware interrupt. This flag can be reset by writing to the TOF bit of the free-running

counter's TFLG2 register. If overflow indicators are not important to a particular application, just tell the software to ignore them.

Computer Operating Properly Watchdog

Loosely related to the main timer is the Computer Operating Properly (COP) watchdog timer. The COP watchdog timer is intended to detect software processing errors. When the COP is used, software is responsible for keeping the watchdog timer from timing out. If the watchdog timer times out, it is an indication that the software has stalled and is no longer in charge of the system. When COP times out, it resets the MCU.

The clock input to the COP system is tapped off the free-running counter chain. The time-out period is set by the CR1 and CR2 bits in the configuration options (OPTION) register. While it is a powerful tool, you must use COP carefully. If you are designing a product where it is imperative that the cycle is not interrupted, like a chemical batch process, it is not wise to use COP. The results could be disastrous. In this scenario, use the time-out features of the output-compare function to sound an alarm instead of an automatic COP system reset. The COP system is enabled or disabled using the NOCOP bit in the CONFIG register.

Interrupts

Also linked to the main timer are the 68HC11's real-time interrupts. If you write programs that must respond quickly to events that may occur unpredictably, you should know how to use interrupts.

Most software programs are sequential. That is, they execute a series of instructions, one after the other, in a predefined order. For example, an automatic dishwasher would wash, rinse, and dry in that order. Simple enough. But what if the user wanted to open the door to add a coffee cup to the menagerie of plates, pots, and pans halfway through the cycle? In other words, interrupt the process.

Using interrupts, a microcontroller can quickly detect and respond to events like the following:

1. **A sensor generates a warning, such as fluid level is too high or too low.**
2. **A power failure has occurred.**
3. **A timer times out, announcing a subsequent operation is pending.**
4. **A counter overflows, indicating that a defined number of events have occurred.**
5. **Data arrives at an I/O port for processing.**
6. **A user requests action to be taken by pressing a key.**

Interrupt-driven programs are fast and efficient. They enable a MCU to respond to an event that may occur at any time and return to whatever it was doing before the interrupt occurred — or take another course of action.

In the 68HC11 (and most MCUs), interrupts are signaled by register flags. Once during each machine cycle, the 68HC11 checks all the registers to see if an interrupt flag is set. A flag indicates an event has occurred that requires immediate attention. The 68HC11 has two types of interrupts: external interrupts that stop the program because of an input signal, and those that output a signal from the MCU to an external device to halt its operation.

Interrupt Vectors

External interrupts are received on the IRQ and XIRQ pins. In addition to IRQ and XIRQ, five other pins on the 68HC11 can also be used to generate external interrupt requests: PC0/IC3, PA1/IC2, PC2/IC1, PA7/OC1, and AS/STRA. These pins are associated with the timer and I/O handshakes. Internal interrupts are generated from the serial interface unit (SCI), pulse accumulator, timer unit, a software interrupt instruction, an illegal opcode, COP watchdog, and a clock monitor.

Each interrupt is associated with a vector address. Without getting too complicated at this point, these vectors point to a memory address that initiates a routine to clear the interrupt and go back to the main program where it left off. Each interrupt has a priority, with highest priority given to the most important interrupt request. While priority of interrupts may be of little concern in many applications, I would sure like to know ahead of time when a boiler is getting ready to explode. This scenario definitely takes priority over a burned-out light bulb. Table 2-5 lists the interrupt vectors for the 68HC11A8 from highest to lowest priority.

Interrupt Priorities & Vectors

INTERRUPT TYPE	DEFAULT PRIORITY	HPRIO PSEL3	PROMOTE PSEL2	TO # 1 PSEL1	PRIORITY PSEL0	EZ-MICRO TUTOR PSUEDO VECTOR	INTERRUPT SERVICE ROUTINE STARTING ADD	ROM LOCATIONS HOLDING INTERRUPT SERVICE ROUTINE STARTING ADD
IRQ	1	0	1	1(0)	0(1)	$01EE	$00EE	$FFF2-$FFF1
RTI	2	0	1	1	1	$01EB	$00EB	$FFF0-$FFF1
Input Capture 1	3	1	0	0	0	$01E8	$00E8	$FFEE-$FFEF
Input Capture 2	4	1	0	0	1	$01E5	$00E5	$FFEC-$FFED
Input Capture 3	5	1	0	1	0	$01E2	$00E2	$FFEA-$FFEB
Output Compare 1	6	1	0	1	1	$01DF	$00DF	$FFE8-$FF89
Output Compare 2	7	1	1	0	0	$01DC	$00DC	$FFE6-$FFE7
Output Compare 3	8	1	1	0	1	$01D9	$00D9	$FFE4-$FFE5
Output Compare 4	9	1	1	1	0	$01D6	$00D6	$FFE2-$FFE3
Input Capture 4/Output Compare 5	10	1	1	1	1	$01D3	$00D3	$FFE0-$FFE1
Timer Overflow	11	0	0	0	0	$01D0	$00D0	$FFDE-$FFDF
Pulse Accumulator Overflow	12	0	0	0	1	$01CD	$00CD	$FFDC-$FFDD
Pulse Accumulator Input Edge	13	0	0	1	0	$01CA	$00CA	$FFDA-$FFDB
SPI Serial Transfer Complete	14	0	0	1	1	$01C7	$00C7	$FFD8-$FFD9
SCI Serial System	15	0	1	0	0	$01c4	$00c4	$FFD6-$FFD7
XIRQ	–	–	–	–	–	$01F1	$00F1	$FFF4-$FFF5
SWI	–	–	–	–	–	$01F4	$00F4	$FFF6-$FFF7
Illegal Opcode Trap	–	–	–	–	–	$01F7	$00F7	$FFF8-$FFF9

Table 2-5. MC68HC11 interrupt priorities and vectors

Before you think interrupts are forever carved in stone, think again. You can write your own interrupt service routines. The vector jump table in RAM allows the user to write different routines using the PSEL bits of the HPRIO register, each having its own starting address. The first byte in the vector jump table must always be a JMP (jump) instruction whose opcode is 7E., but more about that later. Just know that you have complete control over the 68HC11's interrupts, except for the reset codes.

Resets

Resets are also tied to the main clock. One of the things about the 68HC11 is that the reset structure in the 68HC11 is quite different from other MCUs. The 68HC11's reset system can generate a reset output if reset-causing conditions are detected by internal hardware or firmware, as evidenced by the optional COP watchdog timer described above.

Once the reset condition is recognized, internal register and control bits are forced to an initial state. The functions and registers involved are:

1. **CPU stack pointer**
2. **Memory Map**
3. **Parallel I/O ports**
4. **Main timer**
5. **Real-time interrupt flags**
6. **Serial communications interface (SCI)**
7. **Serial peripheral interface (SPI)**
8. **All interrupt and EEPROM programming controls**
9. **Built-in A/D converter**

So you have to be careful that a reset does not clear your external operation in the middle of a cycle because of a software glitch. I can't stress enough what the consequences of a 68HC11 reset would do in the middle of a chemical batch process.

Thankfully, most 68HC11 reset interrupts are maskable. That is, you can prevent them from happening by simply setting a bit in a register. Better yet, you can prevent them from happening by paying close attention to interrupts and writing tight code that doesn't bomb. Which is why I recommend EEPROM versions of the 68HC11, or versions without built-in memory for the first go around, as is the case with the 68HC11D0 used in the EZ-Courseware microcontroller development system. MCUs with EEPROM or outboard memory are more expensive, agreeably, but more flexible. Moreover, the code is easily translated into ROM or EPROM after all the bugs are worked out.

In addition to the sources already described, a MCU reset can be forced by pulling the RESET pin low. The outcome is the same an internal reset.

Special Resets

There are two additional reset conditions that place the 68HC11 into special modes of operation. They are *special-test* and *special-bootstrap*. These modes, along with the Single-Chip and Expanded modes discussed in Section Three above, are set via the MODA and MODB bits of the CONFIG register.

Generally, these bits are not available to the end user because of their specialized functions. For example, the test mode overrides several automatic protection mechanism, which leads to risks when working in this mode of operation. However, they are quite helpful when developing new products. Which is why the EZ-Courseware microcontroller development system evaluation board has jumpers for putting the 68HC11 into both of these special operating modes.

Special Test Mode

The special test mode is primarily intended for Motorola internal production testing. However, there are a few cases where the user can use the test mode. For example, one important use of the test mode is to allow programming of the CONFIG register.

Special Bootstrap Mode

When the 68HC11 is reset in the special bootstrap mode, a small on-chip ROM is enabled at hexadecimal addresses BF40 through BFFF. The built-in program in this ROM initializes the SCI serial communications system, checks for a security option, loads a 256-byte program via the SCI port, and jumps to the program loaded at memory address 0000.

The designers of the 68HC11 anticipated that, during product development, there would be times when it would be necessary to jump directly into the EEPROM after reset. The special bootstrap mode does just that. Once booted, the EEPROM software can locate itself at address 0000 and proceed to do what software programs do, like clearing registers of existing data, writing new data into them and doing a clean reboot. Too many users make the mistake of assuming all registers and I/O pins are still in their reset state when a new program loads. This is a way to level the playing field, and branch out to the real working part or the software. Sometimes it can be used for real-world applications, but it is not recommended.

Part Number	RAM	ROM	EPROM	EEPROM	A/D	I/O	3-Volt Version
68HC11A0	256	0	0	0		22	Yes
68HC11A1	256	0	0	512	Yes	22	Yes
68HC11A7	256	8K	0	0		38	Yes
68HC11A8	256	8K	0	512		38	Yes
68HC11C0	256	0	0	0	Yes	35	No
68HC11D0	192	0	0	0		14	Yes
68HC11D3	192	4K	0	0	No	32	Yes
68HC711D3	256	0	4K	0		32	Yes
68HC11E0	512	0	0	0		22	Yes
68HC11E1	512	0	0	512		22	Yes
68HC11E8	512	12K	0	0		38	Yes
68HC11E9	512	12K	0	512	Yes	38	Yes
68HC711E9	512	0	12K	512		38	No
68HC811E2	256	0	0	2K		38	No
68HC11E20	768	20K	0	512		38	No
68HC11F1	1K	0	0	512	Yes	30	Yes
68HC11G5	512	16K	0	0		66	No
68HC711G5	512	0	16K	0	10-bit	66	No
68HC11G7	512	24K	0	0		66	No
68HC11K0	768	0	0	0		37	Yes
68HC11K1	768	0	0	640		37	Yes
68HC11K3	768	24K	0	0	Yes	62	Yes
68HC11K4	768	24K	0	640		62	Yes
68HC711K4	768	0	24K	640		62	No
68HC11KA0	768	0	0	0		32	Yes
68HC11KA1	768	0	0	640		32	Yes
68HC11KA2	1K	32K	0	640		51	No
68HC711KA2	1K	0	32K	640	Yes	51	No
68HC11KA3	768	24K	0	0		51	Yes
68HC11KA4	768	24K	0	640		51	Yes
68HC711KA4	768	0	24K	640		51	No
68HC11L0	512	0	0	0		30	Yes
68HC11L1	512	0	0	512		30	Yes
68HC11L5	512	16K	0	0	Yes	46	Yes
68HC11L6	512	16K	0	512		46	Yes
68HC711L6	512	0	16K	512		46	Yes
68HC11P2	1K	32K	0	640	Yes	62	No
68HC711P2	1K	0	32K	640		62	No

Table 2-1. MC68HC11 Family Members

How The 68HC11 A/D Converter Works

The 68HC11 A/D converter is a *successive-approximation* converter. A successive-approximation converter transforms voltage into 8-bit binary numbers by comparing the unknown input voltage to a known voltage inside the converter. A comparator circuit indicates whether the unknown voltage is higher or lower than the A/D comparison voltage. The A/D comparison voltage is determined by the bits in the successive-approximation register (SAR) register. A block diagram of the 68HC11 A/D converter is shown in Figure 2-8.

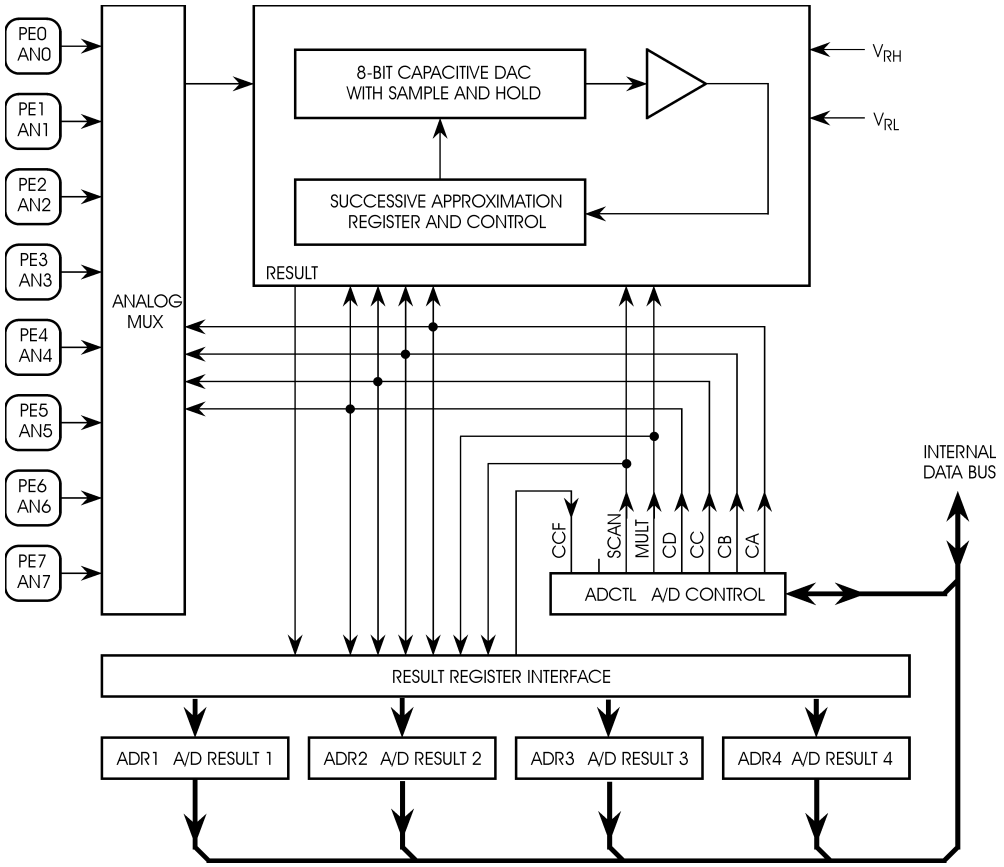


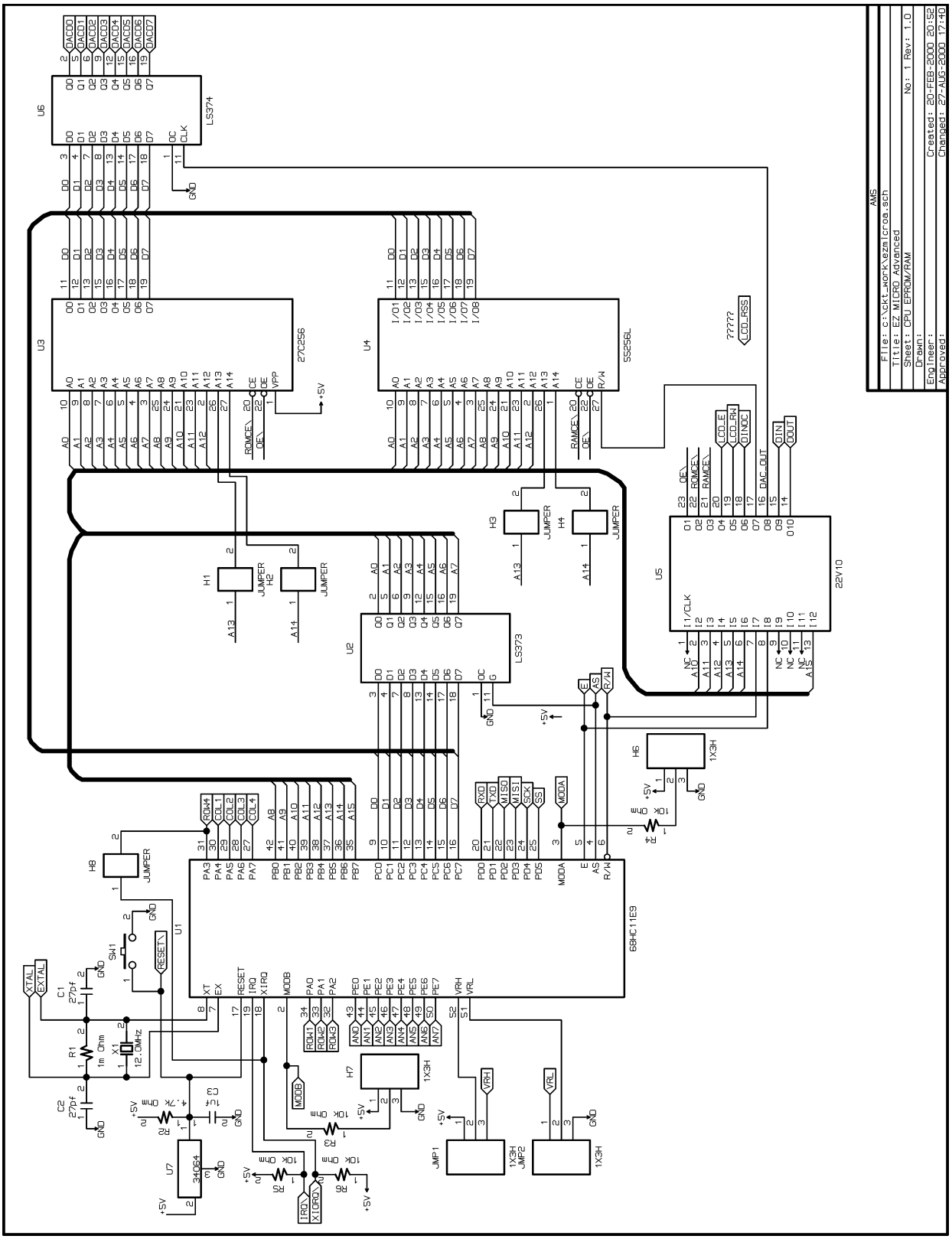
Figure 2-8 A/D converter block diagram

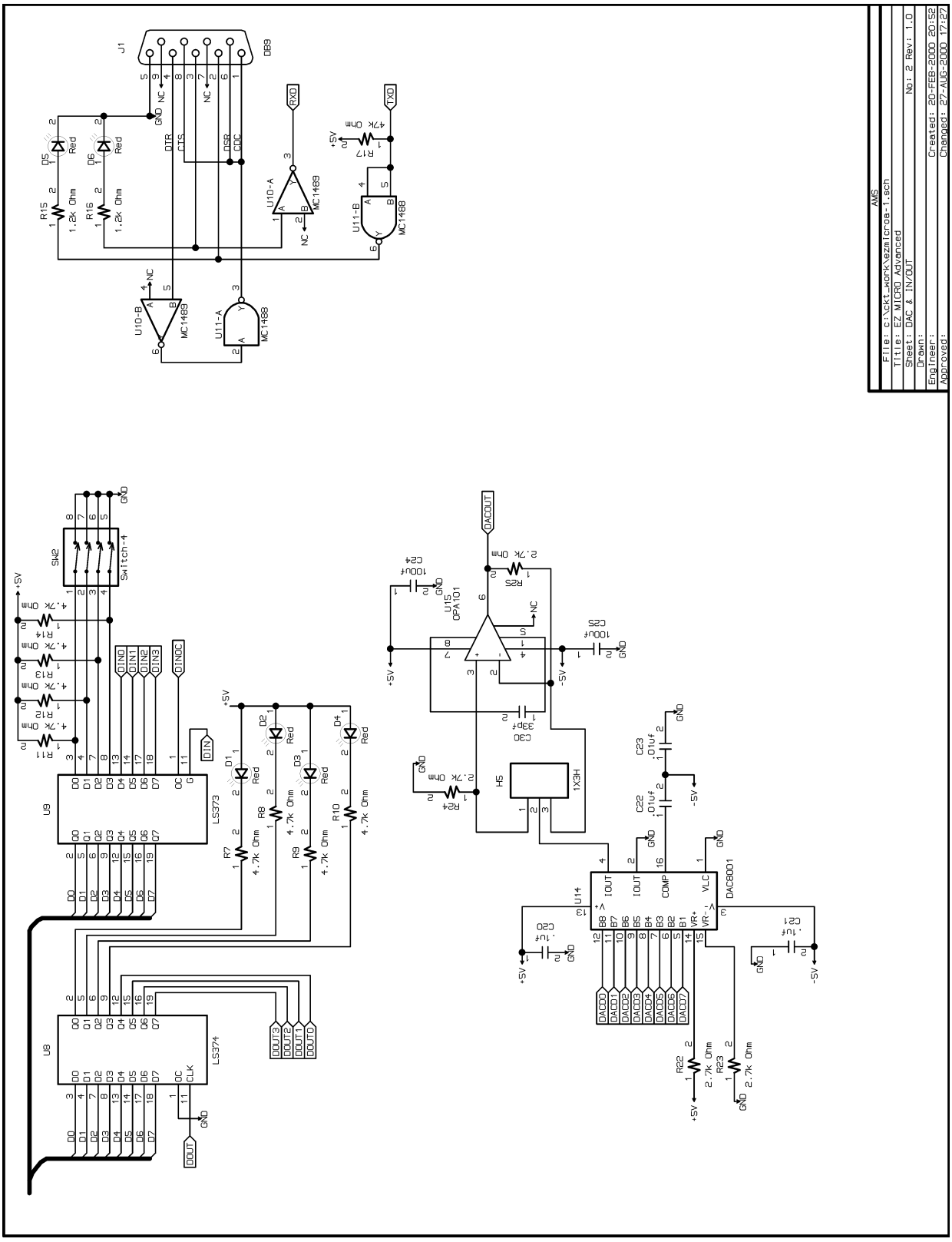
When the software program begins a conversion, the SAR register is loaded with the binary number 1000000 — a value that produces an internal A/D comparison voltage which is halfway between the A/D's high and low reference voltages. The converter then samples the unknown input voltage, which it latches into a sample-and-hold circuit. If the unknown voltage is higher than the A/D voltage, the binary number is incremented to 1100000, which raises the comparison voltage to a higher value, and another comparison is made. If the unknown voltage is lower than the A/D voltage, the binary number is decremented to 0100000, which lowers the comparison voltage, and a comparison is made.

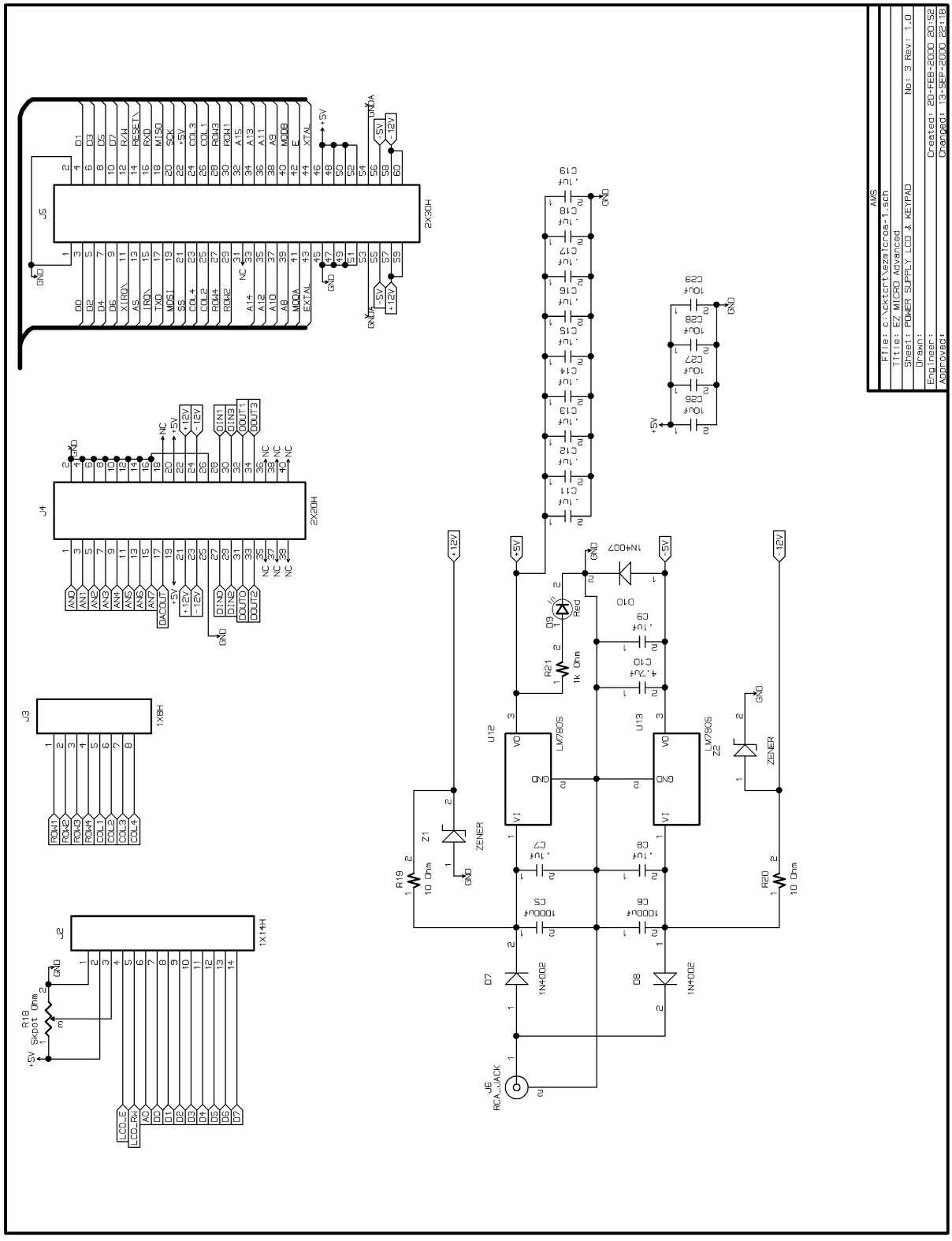
Notice that the comparison process begins with the most-significant bit (MSB) and continues to the least-significant bit (LSB). If the unknown voltage is higher than the comparison voltage, the bit retains its 1 value; if the unknown voltage is lower, the bit is returned to 0. The process is repeated eight times, once for each of the A/D bits, to arrive at a binary value that is equal to the unknown voltage. In a 5-volt reference framework, each LSB binary value is equal to about 20 millivolts; consequently a binary code of 01110111 would equal 2.34 volts.

A/D Accuracy

For the conversion process to be repeatable and accurate, the input voltage must fall within the same range as the A/D's high and low reference voltages. Typically +5 volts and 0 volts are used, which means the external sensor should have a full-range output of +5 volts.


AMS
File: c:\ckt_work\ezmicros.sch
Title: EZ MICRO Advanced
Sheet: CPU EPROM/RAM
No: 1 Rev: 1.0
Drawn:
Engineer:
Approved:
Created: 20-FEB-2000 20:52
Changed: 27-AUG-2000 17:40
U6
LS374
U3
27C256
U4
55256L
U5
22V10
U2
LS373
U1
68HC11E9
H1
H2
H3
H4
H6
H7
H8
JUMPER
1X3H
JMP1
JMP2
SW1
RESET\
XTAL
EXTAL
C1
27pf
C2
27pf
C3
.1uf
R1
1m Ohm
X1
12.0MHz
R2
10K OHM
R3
10K OHM
R4
10K OHM
R5
10K OHM
R6
10K OHM
U7
34064
+5V
GND
LCD_RSS
LCD_E
LCD_RW
DINOC
DIN
DOUT
DAC_OUT
OE\
ROMCE\
RAMCE\
RXD
TXD
MISO
MISI
SCK
SS
MODA
MODB
E
AS
R/W
IRQ\
XIORQ\
VRH
VRL


AMS
File: c:\ckt_work\ezmicro8-1.sch
Title: EZ MICRO Advanced
Sheet: DAC & IN/OUT
No: 2 Rev: 1.0
Drawn:
Engineer:
Created: 20-FEB-2000 20:52
Changed: 27-AUG-2000 17:27
Approved:
J1
DB9
U10-A
U11-B
U10-B
U11-A
MC1489
MC1488
RXD
TXD
R15
R16
1.2k Ohm
D5
D6
Red
DTR
RTS
DSR
CDC
GND
NC
U9
U8
LS373
LS374
SW2
Switch-4
DIN
DOUT
DOUT3
DOUT2
DOUT1
DOUT0
DIN0
DIN1
DIN2
DIN3
DINOC
R7
R8
R9
R10
4.7k Ohm
R21
R22
R23
2.7k Ohm
R24
R25
2.7k Ohm
U15
OPA101
C24
C25
100uf
C30
33pF
H5
1X3H
C23
C22
.01uf
U14
DAC8001
C20
C21
.1uf
DACOUT
DACD0
DACD1
DACD2
DACD3
DACD4
DACD5
DACD6
DACD7
IOUT
COMP
VLC
VR+
VR-
+5V
-5V


AMS
File: c:\cktcr\ezmicroa-1.sch
Title: EZ MICRO Advanced
Sheet: POWER SUPPLY LCD & KEYPAD
No: 3 Rev: 1.0
Drawn:
Engineer:
Approved:
Created: 20-FEB-2000 20:52
Changed: 13-SEP-2000 22:18
J5
2X30H
J4
2X20H
J3
1X8H
J2
1X14H
U12
LM7805
U13
LM7905
D9
LED
Red
D10
1N4007
D7
1N4002
D8
1N4002
J6
RCA_JACK
ZENER
Z1
Z2
R18
5kpot Ohm
R19
10 Ohm
R20
10 Ohm
R21
1k Ohm
+5V
-5V
+12V
-12V
GND
GNDA

Lesson 2 Questions

SECTION ONE

1) What is a microcontroller development system?

2) What does CPU stand for, and what is its purpose in a microcontroller?

3) What is an opcode?

4) Describe what a register does.

5) List the seven CPU registers in the 68HC11 microcontroller.

6) List the four types of memory available in the 68HC11.

7) What does ROM stand for? How is it programmed and when is it used?

8) What type of memory would you use for real-time read/write?

9) What are the two main differences between EPROM and EEPROM?

10) In what type of designs should you use select EPROM over EEPROM?

SECTION TWO

1) Name the six addressing modes of the 68HC11.

2) What is an addressing mode?

3) Describe the structure of a typical 68HC11 instruction.

4) Where are the 68HC11 registers located?

6) Which addressing mode would you use for absolute addressing?

7) What is an offset byte, and what does it do?

8) Do all addressing modes need an address to work? If not, which ones?

9) Do all addressing modes need a data byte to work? If not, which ones?

10) Which of the addressing modes is used for program branching?

SECTION THREE

1) How many I/O ports does the 68HC11 have, and how are they labeled?

2) Which port has both digital and analog inputs?

3) What is the difference between an input line, output line, and a bidirectional line?

4) How many ports have bidirectional lines? Name them.

5) Which port has only six pins, and why is this significant? In other words, what else does this port do that requires less than eight pins?

6) Describe the difference between single-chip mode and extended mode operation.

7) How many data lines are needed for synchronous serial communications? How many for asynchronous?

8) What is a baud?

9) What are the two functions of Port A?

10) Port E has an A/D control register. What is it called and where is it located?

SECTION FOUR

1) What is the E-clock?

2) When would you use the input-capture (IC) and output-compare (OC) the functions?

3) What do you have to do to program the CONFIG register?

4) How does the timer overflow work, and what can you use it for?

5) What is the purpose of the COP watchdog timer?

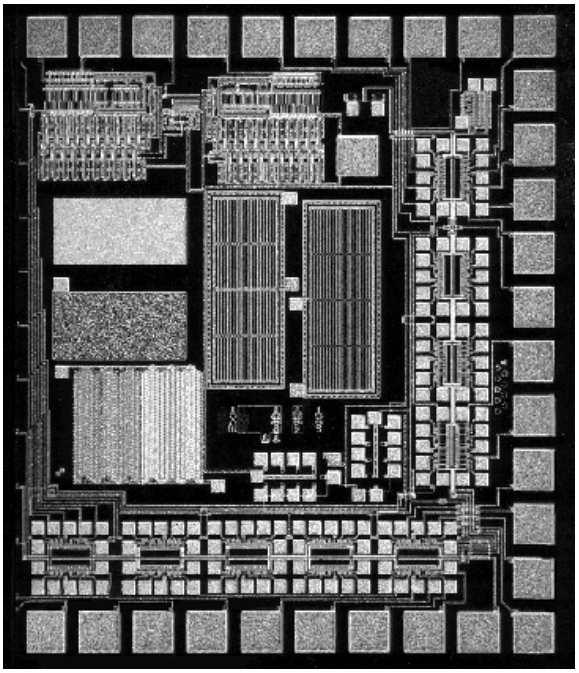
6) How do you keep the COP watchdog timer from timing out?

7) What is an interrupt?

8) Name two situations in which you would use an interrupt.

9) What is an interrupt vector?

10) What effect does a hardware reset have on the stack if the reset occurs while a program is running?



Lesson Three

Programming the MC68HC11

1. The different instruction groups of the MC68HC11
2. Machine language architecture
3. How to use machine language programming
4. Assembly language architecture
5. How to use assembly language programming
6. How to use an assembler and debugger
7. How to download programs on the evaluation board

Now that you understand what the 68HC11 is capable of doing, it is time to show you how to make the MCU work; *e.g.*, how to program the 68HC11. In this lesson, you will learn how to write programming code, convert it into an application program, and load the application program into the EZ-Courseware microcontroller development system.

Instruction Sets

Like all processors, the 68HC11's software operations are done using an *instruction set*, a defined number of opcodes that control the CPU and MCU registers. The 68HC11 comes from a long line of Motorola microprocessors beginning with the MC6800. The 68HC11 can run all the 6800 and 6801 instructions, plus 91 additional instructions.

Instruction Operation

Generally, several machine cycles are needed to complete the operation of an instruction as the CPU proceeds through its various states. A complete instruction operation happens in two distinct phases: the *fetch* phase and the *execute* phase.

During the fetch phase, a copy of the instruction is moved from its memory location into the CPU registers. Usually the control unit will fetch the instructions from the memory in the physical order of the memory registers--one after another. This method reduces the software overhead because the only addressing operation necessary to find the next memory location is to increment the program counter. If the control unit has to fetch the instruction from scattered memory locations, it takes more machine cycles and more time.

During the execute phase, the fetched instruction is run. Once the process is started, the control unit will automatically continue to fetch and execute instructions until something stops it--typically an interrupt or reset. In most MCU applications, programs run in loops, like a dog chasing its tail, so that they continuously control the devices connected to them.

Instruction Groups

An instruction set is divided into functional groups of instructions. That way, it is easier to find your way around when writing a program. For example, if you need an arithmetic function, simply look in that group and find the instruction that is applicable to your program. Basically, there are five major groups, divided accordingly:

1. Arithmetic operations.
2. Logical operations.
3. Load and store operations.
4. Testing and branching.
5. Input and output.

Prebytes

While all 6800 instructions will run on the 68HC11, many of the 68HC11 instructions will not run on the 6800, 6801, 6802, and 68701 processors. That is because these microprocessors do not have a Y index register, like the 68HC11 does. When the Y index register opcodes are included in the instruction set, the number of opcodes exceeds 256--more than can fit into an 8-bit number. Consequently, some 68HC11 instructions have double-byte opcodes.

To make the 68HC11 instruction set backward compatible with previous Motorola 6800 processors, the 68HC11 uses a *prebyte*. The prebyte precedes the normal single opcode byte, and is coded in such a way that if the processor does not recognize it, the system won't crash. Instead, an error flag is set. Prebyte instructions are also used to allow the CPU to perform 16-bit arithmetic operations.

However, the use of a prebyte requires a double-fetch operation, which takes more clock cycles. The result is a performance hit when using the Y index register. Which is why you should use the X index register for all 8-bit opcode operations whenever possible.

68HC11 Instruction Set

Motorola divides the 68HC11 instruction set into 13 functional groups.

The 68HC11 instruction groups are divided as follows:

1. Load, Store, and Transfer
2. Arithmetic Operations
3. Multiply and Divide
4. Logical Operations
5. Data Testing and Bit Manipulation
6. Shifts and Rotate
7. Stack Pointer and Index Register Instructions
8. Condition Code Register (CCR) Instructions
9. Branching
10. Jumps
11. Subroutine Calls and Returns
12. Interrupt Handling
13. Miscellaneous

The following is a short description of each of these groups, and the opcodes that fall within their realm. Many of these groups have subgroups, and some instructions appear in more than one group. For example, the transfer accumulator A to Condition Code Register (TAP) appears in the CCR group and in the load and store group. Examples of the 68HC11 instruction use will follow later in this lesson.

Load, Store, and Transfer

Virtually all 68HC11 activities involve the transfer of data to and from memory, or the I/O ports. Within this group of instructions are four functions. Load moves data into a CPU register. Store moves data from a CPU register to memory or I/O port. Transfer moves data from one CPU register to CPU another. Exchange, which includes Push and Pop, trades data between two locations.

Function	Opcode	IMM	DIR	EXT	INDX	INDY	INH
Clear Memory Byte	CLR			X	X	X	
Clear A	CLRA						X
Clear B	CLRB						X
Load A	LDAA	X	X	X	X	X	
Load B	LDAB	X	X	X	X	X	
Load D	LDD	X	X	X	X	X	
Store A	STAA		X	X	X	X	
Store B	STAB		X	X	X	X	
Store D	STD		X	X	X	X	
Transfer A to B	TAB						X
Transfer A to CCR	TAP						X
Transfer B to A	TBA						X
Transfer CCR to A	TPA						X
Exchange D with X	XGDX						X
Exchange D with Y	XGDY						X
Pull A from Stack	PULA						X
Pull B from Stack	PULB						X
Push A into Stack	PSHA						X
Push B into Stack	PSHB						X

Table 3-1. Load, Store, and Transfer instructions

Arithmetic Operations

This group of instructions includes add, subtract, compare, increment, and decrement. Both 8-bit and 16-bit arithmetic operations are supported, as are 2's-compliment (signed) and binary (unsigned) operations. While test instructions are included, they are seldom used because most arithmetic operations automatically update the condition code register (CCR) bits.

Function	Opcode	IMM	DIR	EXT	INDX	INDY	INH
Add Accumulators	ABA						X
Add B to X	ABX						X
Add B to Y	ABY						X
Add with Carry to A	ADCA	X	X	X	X	X	
Add with Carry to B	ADCB	X	X	X	X	X	
Add Memory to A	ADDA	X	X	X	X	X	
Add Memory to B	ADDB	X	X	X	X	X	
Add Memory to D	ADDD	X	X	X	X	X	
Compare A to B	CBA						X
Compare A to Memory	CMPA	X	X	X	X	X	
Compare B to Memory	CMPB	X	X	X	X	X	
Compare D to Memory	CPD	X	X	X	X	X	
Decimal Adjust A (for BCD)	DAA						X
Decrement Memory Byte	DEC			X	X	X	
Decrement A	DECA						X
Decrement B	DECB						X
Increment Memory Byte	INC			X	X	X	
Increment A	INCA						X
Increment B	INCB						X
2's Complement Memory Byte	NEG			X	X	X	
2's Complement A	NEGA						X
2's Complement B	NEGB						X
Subtract B from A	SBA						X
Subtract with Carry from A	SBCA	X	X	X	X	X	
Subtract with Carry from B	SBCB	X	X	X	X	X	
Subtract Memory from A	SUBA	X	X	X	X	X	
Subtract Memory from B	SUBB	X	X	X	X	X	
Subtract Memory from D	SUBD	X	X	X	X	X	
Test for Zero or Minus	TST			X	X	X	
Test for Zero or Minus A	TSTA						X
Test for Zero or Minus B	TSTB						X

Table 3-2. Arithmetic instructions

Multiply and Divide

While multiply and divide are arithmetic operations, they are grouped separately. A reason for this is because the results of most multiply and divide operations produce a 16-bit result, which takes special handling. The 68HC11 has one multiply and two divide instructions.

Function	Opcode	IMM	DIR	EXT	INDX	INDY	INH
Multiply (A x B > D)	MUL						X
Fractional Divide (D ÷ X > X; r > D)	FDIV						X
Integer Divide (D ÷ X > X; r > D)	IDIV						X

Table 3-3. Multiply and Divide instructions

Logical Operations

This instruction group contains the handful of extremely powerful opcodes that set the microprocessor apart from a simple calculator: logic operations. This group of instructions performs the Boolean logical operations AND, Inclusive OR, exclusive OR, and 1's complement.

Function	Opcode	IMM	DIR	EXT	INDX	INDY	INH
AND A with Memory	ANDA	X	X	X	X	X	
AND B with Memory	ANDB	X	X	X	X	X	
Bit(s) Test A with Memory	BITA	X	X	X	X	X	
Bit(s) Test B with Memory	BITB	X	X	X	X	X	
1's Complement Memory Byte	COM			X	X	X	
1's Complement A	COMA						X
1's Complement B	COMB						X
OR A with Memory (Exclusive)	EORA	X	X	X	X	X	
OR B with Memory (Exclusive)	EORB	X	X	X	X	X	
OR A with Memory (Inclusive)	ORAA	X	X	X	X	X	
OR B with Memory (Inclusive)	ORAB	X	X	X	X	X	

Table 3-4. Logical Operation instructions

Data Testing and Bit Manipulation

This group is used to test and alter the contents of a register or memory on a bit-by-bit or byte basis. Typical usage is to clear or set specific memory bits. Some care is required when modifying some read/write instructions on I/O and control register locations because the physical read location is not always the same as the write location.

Function	Opcode	IMM	DIR	EXT	INDX	INDY	INH
Bit(s) Test A with Memory	BITA	X	X	X	X	X	
Bit(s) Test B with Memory	BITB	X	X	X	X	X	
Clear Bit(s) in Memory	BCLR		X		X	X	
Set Bit(s) in Memory	BSET		X		X	X	
Branch if Bit(s) are Clear	BRCLR		X		X	X	
Branch if Bit(s) are Set	BRSET		X		X	X	

Table 3-5. Data Testing and Bit Manipulation instructions

Shifts and Rotate

The nice thing about binary mathematics is that it is easy to do complex operations, like square roots, using simple shift and rotate operations. The instructions in this group let you do just that. In addition to arithmetic shifts, these instructions also do logical shifts.
The logical-left-shift instructions are shown in parentheses because there is no difference between an arithmetic and a logical left shift. Both opcodes are recognized by an assembler as being one and the same. The different mnemonics simply help the programmer write the code.

Function	Opcode	IMM	DIR	EXT	INDX	INDY	INH
Arithmetic Shift Left Memory	ASL			X	X	X	
Arithmetic Shift Left A	ASLA						X
Arithmetic Shift Left B	ASLB						X
Arithmetic Shift Left D	ASLD						X
Arithmetic Shift Right Memory	ASR			X	X	X	
Arithmetic Shift Right A	ASRA						X
Arithmetic Shift Right B	ASRB						X
(Logical Shift Left Memory)	(LSL)			X	X	X	
(Logical Shift Left A)	(LSLA)						X
(Logical Shift Left B)	(LSLB)						X
(Logical Shift Left D)	(LSLD)						X
Logical Shift Right Memory	LSR			X	X	X	
Logical Shift Right A	LSRA						X
Logical Shift Right B	LSRB						X
Logical Shift Right D	LSRD						X
Rotate Left Memory	ROL			X	X	X	
Rotate Left A	ROLA						X
Rotate Left B	ROLB						X
Rotate Right Memory	ROR			X	X	X	
Rotate Right A	RORA						X
Rotate Right B	RORB						X

Table 3-6. Shift and Rotate instructions.

Stack Pointer and Index Register Instructions

This group of instructions are used to transfer data between the 16-bit index registers and the stack pointer using accumulator D, which has more powerful 16-bit arithmetic capabilities than the index registers.

Function	Opcode	IMM	DIR	EXT	INDX	INDY	INH
Add B to X	ABX						X
Add B to Y	ABY						X
Compare X to Memory	CPX	X	X	X	X	X	
Compare Y to Memory	CPY	X	X	X	X	X	
Decrement Stack Pointer	DES						X
Decrement X	DEX						X
Decrement Y	DEY						X
Increment Stack Pointer	INS						X
Increment X	INX						X
Increment Y	INY						X
Load Stack Pointer	LDS	X	X	X	X	X	
Load X	LDX	X	X	X	X	X	
Load Y	LDY	X	X	X	X	X	
Pull X from Stack	PULX						X
Pull Y from Stack	PULY						X
Push X onto Stack	PSHX						X
Push Y onto Stack	PSHY						X
Store Stack Pointer	STS		X	X	X	X	
Store X	STX		X	X	X	X	
Store Y	STY		X	X	X	X	
Transfer Stack Pointer to X	TSX						X
Transfer Stack Pointer to Y	TSY						X
Transfer X to Stack Pointer	TXS						X
Transfer Y to Stack Pointer	TYS						X
Exchange D with X	XGDX						X
Exchange D with Y	XGDY						X

Table 3-7. Stack Pointer and Index Register instructions

Condition Code Register (CCR) Instructions

As a rule, the 8-bit conditional code register reports CPU status following an operation. However, three bits of this register are user programmable masks that prevent the carry, interrupt, and overflow bits from generating an inIterrupt from any one or all of these functions.

Function	Opcode	IMM	DIR	EXT	INDX	INDY	INH
Clear Carry Bit	CLC						X
Clear Interrupt Mask Bit	CLI						X
Clear Overflow Bit	CLV						X
Set Carry Bit	SEC						X
Set Interrupt Mask Bit	SEI						X
Set Overflow Bit	SEV						X
Transfer A to CCR	TAP						X
Transfer CCR to A	TPA						X

Table 3-8. Condition Code Register (CCR) instructions

Branching

These instructions allow the CPU to make decisions based on the condition code register bits. If you look carefully, you will see that for every branch condition there is a branch for the opposite condition. All branch instructions are 2-byte instructions, with the first byte being the opcode and the second byte being the address offset in 2's complement format.

Function	Opcode	REL	DIR	EXT	INDX	INDY	INH
Branch if Carry Clear	BCC	X					
Branch if Carry Set	BCS	X					
Branch if Equal Zero	BEQ	X					
Branch if Greater Than or Equal	BGE	X					
Branch if Greater Than	BGT	X					
Branch if Higher	BHI	X					
Branch if Higher or Same (same as BCC)	BHS	X					
Branch if Less than or Equal	BLE	X					
Branch if Lower (same as BCS)	BLO	X					
Branch if Lower or Same	BLS	X					
Branch if Less Than	BLT	X					
Branch if Minus	BMI	X					
Branch if Not Equal	BNE	X					
Branch if Plus	BPL	X					
Branch if Bit(s) Clear in Memory Byte	BRCLR		X		X	X	
Branch Never	BRN	X	X		X	X	
Branch if Bit(s) Set in Memory Byte	BRSET						
Branch if Overflow Clear	BVC	X					
Branch if Overflow Set	BVS	X					

Table 3-9. Branch instructions

Jumps

The jump instruction allows control to be passed to any address in the 64K memory map.

Function	Opcode	REL	DIR	EXT	INDX	INDY	INH
Jump	JMP		X	X	X	X	
Branch Always	BRA	X					

Table 3-10. Jump instruction

Subroutine Calls and Returns

A subroutine is a short program within a program that is used over and over again. Rather than having to duplicate the code time after time in the program as needed, the 68HC11 has three subroutine instructions that let the program branch or jump to the routine, run it, then return to where the jump took place.

Function	Opcode	REL	DIR	EXT	INDX	INDY	INH
Branch to Subroutine	BSR	X					
Jump to Subroutine	JSR		X	X	X	X	
Return from Subroutine	RTS						X

Table 3-11. Subroutine Calls and Return Instructions

Interrupt Handling

While most programs run in a loop so that the software continuously services the devices connected to the MCU, there are times when it is better for the MCU to wait for an external interrupt to occur. This allows the MCU to be in a standby mode to conserve power until it is needed, as would be the case when the MCU is used as an intelligent alarm. Three opcodes are associated with this feature.

Function	Opcode	IMM	DIR	EXT	INDX	INDY	INH
Return from Interrupt	RTI						X
Software Interrupt	SWI						X
Wait for Interrupt	WAI						X

Table 3-12. Interrupt Handling instructions

Miscellaneous

This group contains three instructions which do not fit into any other defined group, but are useful in some situations. They are NOP, STOP, and TEST.

NOP does what it sounds like — absolutely nothing. It simply wastes two clock cycles, and is used to add a small delay into the program while it waits for an external peripheral to catch up. For precise delays, NOP can be combined with the BRN branch command to produce a three-cycle NOP.

STOP causes all MCU clocks to freeze, which reduces power consumption to virtually nothing. The TEST instruction is used only during factory testing, and is treated as an illegal opcode in normal operating modes. When used with the Test Mode, though, it can be used to help program the CONFIG register.

Function	Opcode	IMM	DIR	EXT	INDX	INDY	INH
No Operation	NOP						X
Stop All Clocks	STOP						X
Test	TEST						X

Table 3-13. Miscellaneous instructions

Machine Language Programming

Using the opcodes described above, you can write a program that does just about anything you want. This type of programming is called *machine language* programming, because it is done using binary machine code.

An advantage of machine language programs is the programs are very compact and small. Because they are small in size, you don’t need as much RAM to run them, which saves you money. Machine language programs also runs faster than programs that are written using other programming methods, like interpreted and compiled language programming.

Machine Language Architecture

However, machine language programs are not easy to write because there is no room for error. Each line of code must be precise, which means you have to pay close attention to details. An example of machine language code is shown in Figure 3-1.

Machine Code		Comments		
Address	Instruction	Opcode	Mode	Description
C010 C011	DE 30	LDX	DIR	Initialize pointer
C012 C013	96 32	LDAA	DIR	Fetch first value
C014 C015	97 33	STAA	DIR	Store value in A
C016 C017	27 0A	BEQ	REL	Branch to STOP if A equals zero
C018 C019	A6 00	LDAA	IND,X	Fetch next value
C01A C01B	A7 20	STAA	IND,X	Store value in A
C01C	08	INX		Advance pointer to the next entry
C01D C01E C01F	7A 00 33	DEC	EXT	Decrement by one
C020 C021	20 F4	BRA	REL	Branch to A (loop)
C022	3F	STOP		Stop the program

Figure 3-1. This is an example of machine language programming. Only the first two columns, address and instruction, are used in the program. The comments are included in this example only to help you see how the process takes place.

Each operation begins with a memory address. Sometimes the address by itself is enough, which is the case when initializing the reserved memory space for use in the direct addressing mode, but mostly the address points to a memory location.

The next item is the hexadecimal data byte of the opcode instruction, like a fetch data instruction (LDAA). The data byte contains the binary code needed to execute the opcode in the selected addressing mode. For example, the data byte needed to execute the LDA instruction in the direct addressing mode is 96. When working in the immediate addressing mode, the LDA data byte is 86.

Most instructions, like LDA, need more than one clock cycle to complete. For every clock cycle, the address has to be changed (usually in increments of one). Many times a different data byte is used for each cycle. Which is why machine language is so demanding. There is no room for error. One mistake in either an address or opcode byte, and the program crashes.

Assembly Language Programming

While writing a program in machine language should not be a problem for short routines, the task of writing a complex process in machine language is daunting. Fortunately, there is no reason you have to. Why not let a desktop computer write the machine language code for you? Why not, indeed. This is what *assembly language programming* does.

No longer do you have to calculate the memory address or enter the data byte. An *assembler* program does that for you. All you need is an opcode and an operand (when applicable). The assembler then deciphers the opcode, assigns the addresses, sets the proper data bytes, and moves the associated data values into the right registers or memory locations.

Being a programming language rather than a machine code, assembly language programming has the added advantage of easily creating branching and looping *subroutines*. After writing a few machine language programs, you will notice that certain patterns are repeated over and again. The only thing that changes are the addresses. Basically, an assembly language subroutine is a block of code that is a complete program in itself. When embedded in a program, the subroutine can be called into action using a single line of assembly code. The assembler makes sure the addresses are in order and do not conflict.
Another advantage of assembly language programming is that the program is portable between MCUs. Because the command is not MCU specific, a program written for the 68HC11 can be used with other Motorola microprocessors and microcontrollers using a *cross-assembler*.

The Assembly Process

The translation from assembly code into machine language is a three-step process. It starts with a text editor — basically a word processor that is usually customized for listing assembly language commands in a format recognized by the assembler. The file the editor creates is called the *source module,* and its contents are called the *source code*.

The assembler program then reads the source code and translates the 68HC11 mnemonics into memory addresses and data bytes, which are further reduced to binary numbers. These binary numbers are stored in an *object module*; the contents of this module is called the *object code*.

The last step is to create an executable *load module*, such as a disk file, that can be read by a loader program to put the binary numbers into the MCU's memory. The load module, or application program, is generally created using a linker program which adds the loader program to the object code and puts both into a single, executable file. You now have an application program, just like a machine language program, but this time created in easier-to-use assembly language.

At this point, the object code is no longer needed, and can be discarded. The source code, however, should be saved in case you make changes to the program. A new object module has to be created and linked each time you make changes to the source code.

Assembly Code Architecture

While many vendors sell assembler programs for 68HC11 MCUs, they all use the same architecture. Each line of assembly code consists of four fields. They are label, opcode, operand, and comments, in that order (see Figure 3-2).

Figure 3-2. An assembly language instruction consists of four fields: label, opcode, operand, and comment. Of the four, only the opcode is required; the other three are either optional or dependent on the opcode.

Label Field

The first field is an optional label that is used like a bookmark. The main use for a label is to identify the beginning of a subroutine. The label is programmer defined — that is, you decide what the label's name should be. Generally, it is something that is related to the code it points to. For example, if you have a timer routine that you use many times in a program, it would be wise to label the starting line of the subroutine TIMER1 to distinguish it from other timer subroutines you may have. The label must start with an alphabetic character (A - Z) and must not include spaces. If the line does not require a label, a space or tab has to be inserted in its place as a place holder because the assembler has to find something — albeit a transparent character — to sequence its operation.

Opcode Field

After the label is the opcode field. Sometimes referred to the operation field, the opcode field contains the 68HC11 opcode. This is the only line entry that is mandatory; without an opcode, the assembler ignores the program line.

Operand Field

Third in line is the operand. Generally, the operand is an argument needed to qualify the operation of the opcode. This field is also used to hold data that is staged for insertion into a register, or to point to an address. When the opcode indicates the operand is data or an address, there are four ways to identify the type of the number:

No symbol	Decimal number
$	Hexadecimal number
@	Octal number
%	Binary number

Note that if no symbol is used, the number is assumed to be in decimal format. Be careful, though, because some assemblers assume no symbol to be hexadecimal numbers.

Comment Field

The fourth and last field is simply reserved for your comments. Generally, it is used to jog your memory when working through a lengthy program. For example, a comment like "timer1

started" and "temp too high interrupt" helps a lot when reading through a long list of opcodes and subroutines. Like the label field, this field is optional.

Additional Comments

Sometimes it is helpful to insert comments into the source code that are not related to a specific opcode or label. For example, "The following section is a 30-second clock" or "This code executes only when a power failure occurs." To insert references like this, start the line with an asterisk (*). When the assembler comes across a line beginning with an asterisk, it simply skips over it, so there is no performance hit incurred by adding these comments to the source code.

Assembly Language Programming Example

There is no better way to see assembly language in action than through an example. The following short section of code, with added annotation, is lifted from a weight measurement scale used to price produce. Before we begin, let me say that you should not try to run this program. It is taken from context, has been altered to highlight certain assembly language features, and will not execute.

```
* LINES BEGINNING WITH AN ASTERISK ARE IGNORED BY THE ASSEMBLER
* THEY ARE USED TO INSERT COMMENTS INTO THE SOURCE CODE
*
*     THE FOLLOWING CODE IS EXCERPTED FROM A WEIGHT SCALE
*                           EXAMPLE ONLY
*
**                          DATA SECTION
            ORG         $20         ;comments here refer to opcode
operation
DATA1 EQU         $31               ;ASCII value of DATA1
DATA2 EQU         $32               ;ASCII value of DATA2
STORE RMB         3
VALUE FDB         20                ;comments are optional, like above
*
**                                  INITIALIZE PORT D
*
            ORG         $BEF40      ;assembler directive
            LDS         #$00FF      ;initialize stack
            LDX         #$1000      ;initialize X register for indexed
access
            BSET        SPCR,X  $20      ;put port D in wire-or mode
*
*                       INITIALIZE SCI AND RESET BAUD DIVIDER
*
BEGIN LDAA  #$A2        ;divide by 16 (load memory $A2 into A)
            STAA  BAUD,X            ;reset baud divider chain
                                    ;(store A as BAUD)
            LDAA  #$0C        ;rcvr/xmtr enable (load memory $0C into A)
            STAA  SCCR2,X     ;store as SCCR2
            LDAA  DATA2       ;load DATA2 into A
            STAB  CONST,X     ;
*
END         END
```

As you can see, each field has its own column. This makes it easier to read the program, and is easily accomplished using the keyboard <Tab> key. The first column, the label field, is required. If you do not have a label, it must be replaced with a tab or space. This is a must, as is an opcode in the second field. The operand field is dependent on the needs of the opcode. Notice that in two instances the opcode (ORG) is an assembler directive. This will be discussed later. Comments, while optional, should be clear and succinct.

Lesson 3 Questions

1) What is the difference between an instruction set and an instruction group?

2) List the 68HC11 instruction groups.

3) What is a prebyte and when is it used?

4) Why are the 68HC11's arithmetic instructions divided into two groups?

5) List the 68HC11 opcodes associated with interrupt handling.

6) What does the NOP instruction do?

7) Describe the difference between maching language programming and assembly language programming.

8) How many clock cycles are needed to complete an LDA instruction?

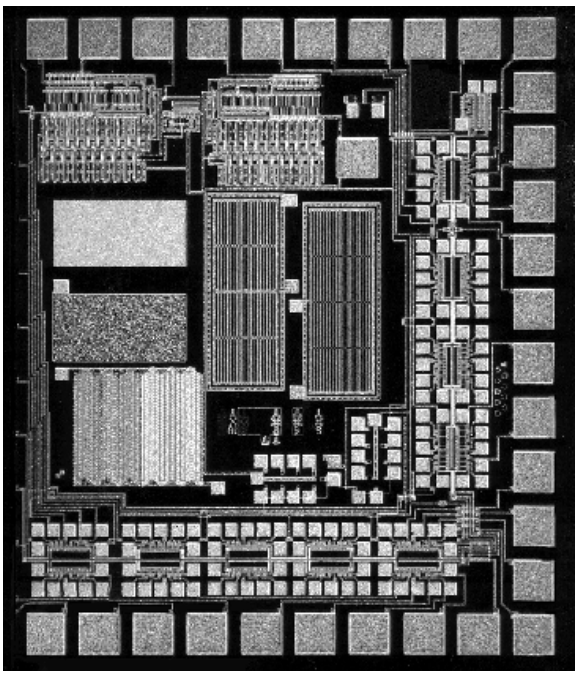
9) What is the significance of having to know the number of machine cycles needed to complete an instruction when working in machine language?

10) What is an assembler? What is a cross-assembler?

11) What is the purpose of the object code?

12) Describe the three steps needed to create an application load module.

13) List the four assembly language code fields, and describe their functions.



Lesson Four

Using the EZ-MICRO Manager software

When you finish this lesson, you will know:

1. How the EZ-MICRO Manager software works
2. Commands of the EZ-MICRO Manager software
3. Features and capabilities of the EZ-MICRO TUTOR™ microprocessor board
4. How to download programs to the EZ-MICRO TUTOR™ microprocessor board

System Requirements

In order to run the EZ-MICRO Manager software, the following equipment is required:

- IBM personal computer (or compatible)
- MS-DOS (version 6.0 or later) for DOS compatible software
  WIN 95/98 Operating System
- Minimum 4 Meg RAM memory
- EGA or VGA graphics adapter card
- RS-232 serial interface port
- A mouse
- A hard disk with atleast 5 MB of Free Space

Software Installation

To install the EZ-MICRO Manager software onto your hard drive, insert the CD that was provided with the EZ-MICRO TUTOR™ microprocessor board.

If your computer is setup to run automatically setup from the CD-ROM, follow the instructions on the screen and installation software will install EZ MICOR software on your hard drive.

If your computer is not setup to run automatic setup from the CD-ROM, using Windows Explorer go to CD-ROM and double click SETUP.EXE file. Then follow the instruction on the screen.

Before starting the EZ-MICRO Manager software, press the master reset button (S1) on the EZ-MICRO TUTOR™ microprocessor board. This will ensure that the software will communicate properly with the hardware.

After the installation is complete, click Start menu on bottom left hand corner of your Window Desktop, goto Programs, select EZ-COURSWARE and select EZ Micro.

How Does the EZ-MICRO Manager Software Work?

The EZ-MICRO Manager software makes it possible to communicate directly to the EZ-MICRO TUTOR™ microprocessor board (Motorola MC68HC11E0FN microprocessor) using any IBM compatible "host" computer that has at least one asynchronous serial port.

All of the commands that are issued to the EZ-MICRO TUTOR™ microprocessor board are selected from the various pull down menus located across the top of the EZ-MICRO Manager presentation screen (Figure 4.1). Therefore, it is not necessary to memorize numerous commands in order to communicate to the CPU using this program.

Several files (including several sample assembly language routines) are included on the EZ-MICRO CD. At AMS, we are always adding new features in the software and more documentation.

Please visit our web site **www.advancedmsinc.com/update/** and download free updates.

The EZ-MICRO Manager Presentation Screen

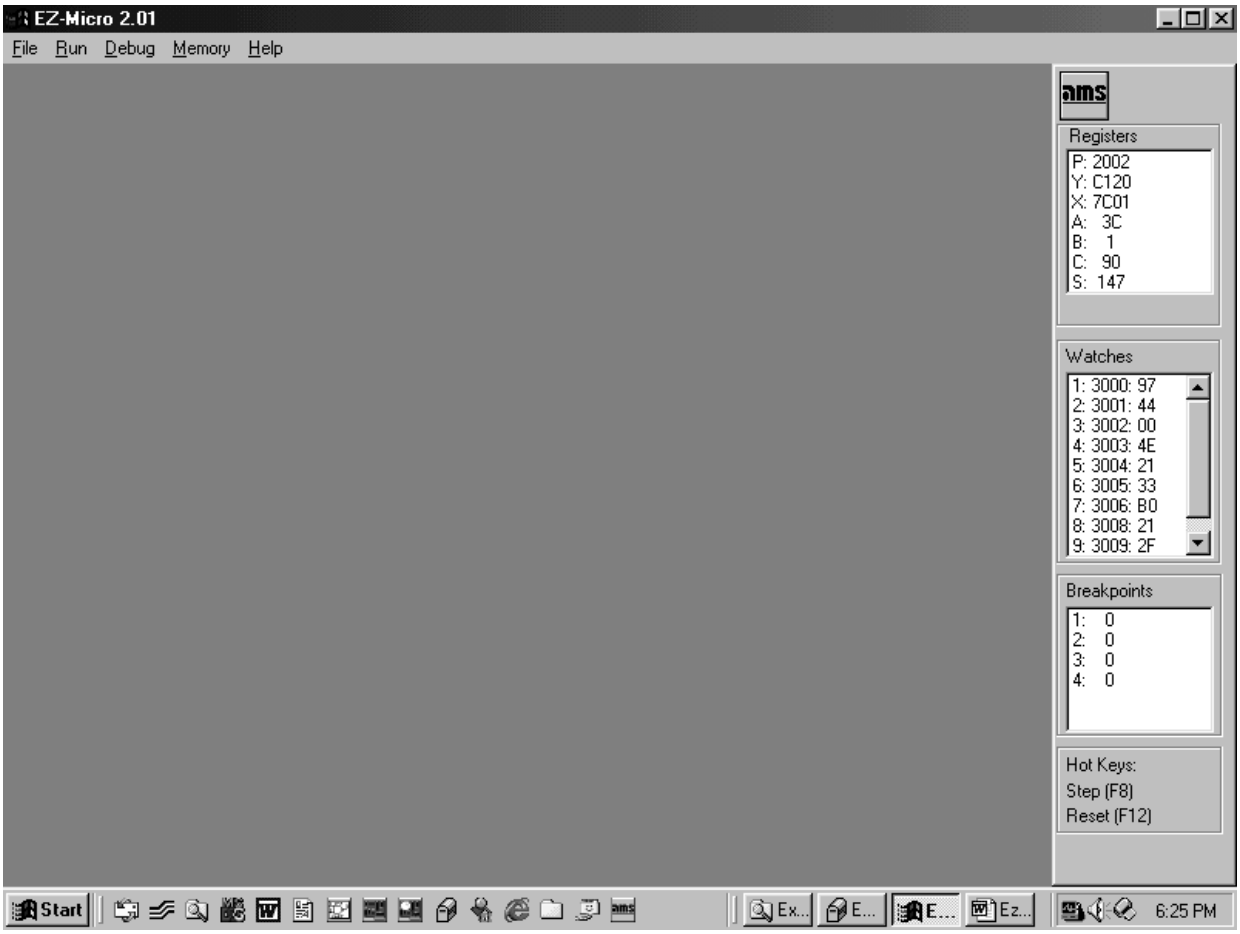


Figure 4.1: EZ-MICRO Manager Presentation Screen.

The EZ-MICRO Manager presentation screen is divided into the following functions.

■ The "**Title Bar**", located at the top of the screen, displays the the title of this software and version number.

■ The **"Action Bar**", located just below the Title Bar, contains the various pull down menus where the commands that are issued to the EZ-MICRO TUTOR™ microprocessor board can be selected.

Notice that in all three cases a menu will be pulled down. To access an adjacent menu, simply press the right or left arrow keys on the keyboard or point at the menu heading with the mouse.

A menu bar is located within the pull down menu and is used to select one of the commands in the menu. The menu bar can be moved up or down by pressing the up or down arrow keys on the keyboard. Once the menu bar is positioned on the desired command, press the ENTER key to select that command. If you are using a mouse, point at the command and click the left mouse button to select the command. Listed to the right of each command in the pull down menus are "hot keys" which allow you to select a command (in the current pulled down menu) by simply pressing its hot key on the keyboard.

■ The "**Registers**" display window is located in the upper right corner of the EZ-MICRO Manager presentation screen. This window is used to display the current 68HC11 microprocessor register contents. These registers are: the Program Counter (P), Y-index (Y), X-index (X), A-accumulator (A), B-accumulator (B), Condition Code (C) and Stack Pointer (S).

Modify Registers

68HC11 microprocessor have several registers namely, program counter (P), Y-index (Y), X-index (X), A-accumulator (A), B-accumulator (B), Condition Code (C) and Stack Pointer (S) register. To modify any of these registes, double click Register value in the register window.

When you double click P, the following dialog box will appear displaying the current Program counter contents. Enter the desired value and Press OK.

Figure 4.14: Modify Registers.

■ The "**Watch Variables**" display window is located below the Registers display window. This window is used to monitor the contents of several memory locations at once while running/debugging the assembled code.

■ The "**Breakpoints**" display window is located below the Watch Variables display window. This window is used to display the current user breakpoints that have been entered into the breakpoint address table.

■ The "**Main Display**" window is the large area located in the left half of the EZ-MICRO Manager presentation screen. This window is used to display the EZ-MICRO TUTOR™ memory contents, disassembled memory listing, trace of program execution and a graphical plot of user memory.

EZ-Micro Manager Commands and Tutorial

The File Menu

The FILE menu contains commands that will access "file type" operations. That is, these commands will read/write to disk memory of the "host" computer or terminal.

Assemble File

The "Assemble File" command in the FILE menu is used to assemble a Motorola 68HC11 assembly language source file into a Motorola S-Record hex file. The assembled S-Record file can be later downloaded to the EZ-MICRO TUTOR™ microprocessor board and executed.

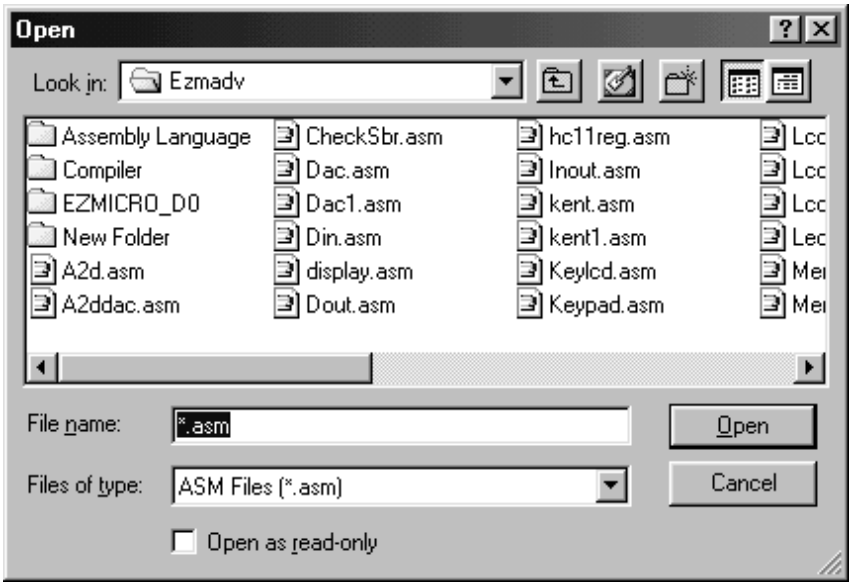


Figure 4.3: Assembly Language Source Files Listing.

Select the Assemble File command from the FILE menu. Select the file "LCD.asm". and press Open. This operation will create two files LCD.S19 and LCD.LST

When viewed by the user, S-Records are essentially character strings made of several fields which identify the record type, record length, memory address, code/data and checksum. Each byte of binary data is encoded as a 2-character hexadecimal number. The first character represents the high order 4 bits, and the second the low order 4 bits of the byte.

The five fields which comprise an S-Record are shown in the following figure:

TYPE	RECORD	LENGTH	ADDRESS	CODE/DATA	CHECKSUM

where the fields are composed as follows:

Field	Printable Characters	Contents
Type	**2**	S-Record type — S0, S1, etc.
Record Length	**2**	The count of the character pairs in the record, excluding the type and record length.
Address	**4,6, or 8**	The 2-, 3-, or 4-byte address at which the data field is to be loaded into memory.
Code/Data	**0-2n**	From 0 to n bytes of executable code, memory loadable data or descriptive information.
Checksum	**2**	The least significant byte of the one's complement of the sum of values represented by the pairs of characters making up the record length, address, and code/data fields.

Eight types of S-Records have been defined to accommodate the needs of encoding, transportation and decoding functions. However, the BUFFALO monitor program that is used by the EZ-MICRO TUTOR™ supports only the S1 and S9 record types. Therefore, all records that are downloaded to the EZ-MICRO TUTOR™ must be S1 type until the S9 record terminates the data transfer.

An S-Record format module may contain S-Records of the following types:

Type	Description
S0	The header record for each block of S-Records. The code/data field may contain any descriptive information identifying the following block of S-Records. The address field is normally zeros.
S1	A record containing code/data and the 2-byte address at which the code/data is to reside.
S2-S8	Not applicable to the EZ-MICRO TUTOR™.
S9	A termination record for a block of S1 records. The address field may optionally contain the 2-byte address of the instruction to which control is to be passed. If not specified, the first entry point specification encountered in the object module input will be used. There is no code/data field.

The following is a sample of a typical S-Record format module:

```
S00600004844521B
S1130000285F245F2212226A000424290008237C2A
S11300100002000800082629001853812341001813
S113002041E900084E42234300182342000824A952
S107003000144ED492
S9030000FC
```

The above module consists of an S0 header record, four S1 code/data records and an S9 termination record.

S0	S-Record type S0, indicating a header record.
06	Hexadecimal 06 (decimal 6), indicating six character pairs (or ASCII bytes) to follow.
00 **00**	Four character 2-byte address (0000h).
48 **44** **52**	ASCII for "H", "D" and "R" (HDR).
1B	Checksum of S0 record.

The first S1 code/data record is defined as follows:

S1	S-Record type S1, indicating a code/data record to be loaded/verified at a two byte address.
13	Hexadecimal 13 (decimal 19), indicating 19 character pairs (or bytes of binary data) to follow.
00 **00**	Four character 2-byte address (0000h). Indicates location where the following data is to be loaded
28,5F **24,5F** **22,12** **22,6A** **00,04** **24,29** **00,08** **23,7C**	The following 16 character pairs (or bytes)is the actual program code/data. **OPCODE INSTRUCTION** ------ -- --------- 28 5F BHCC $0161 24 5F BCC $0163 22 12 BHI $0118 22 6A BHI $0172 00 04 24 BRSET 0,$04,$012F 29 00 BHCS $010D 08 23 7C BRSET 4,$23,$018C The balance of this code is continued in the code/data fields of the remaining S1 records
2A	Checksum of the first S1 record

The third, fourth and fifth lines of the S-Record example are also of type S1.

Finally, the last line is the S9 termination record and is described below.

S9	S-Record type S9, indicating a termination record.
03	Hexadecimal 03 (decimal 3), indicating three character pairs (or bytes of binary data) to follow.
00 **00**	Four character 2-byte address (0000h).
FC	Checksum of the S9 record.

Load S-Record

The "Load S-Record" command in the FILE menu is used to download an assembled Motorola S-Record hex file into user memory located on the EZ-MICRO TUTOR™ microprocessor board.

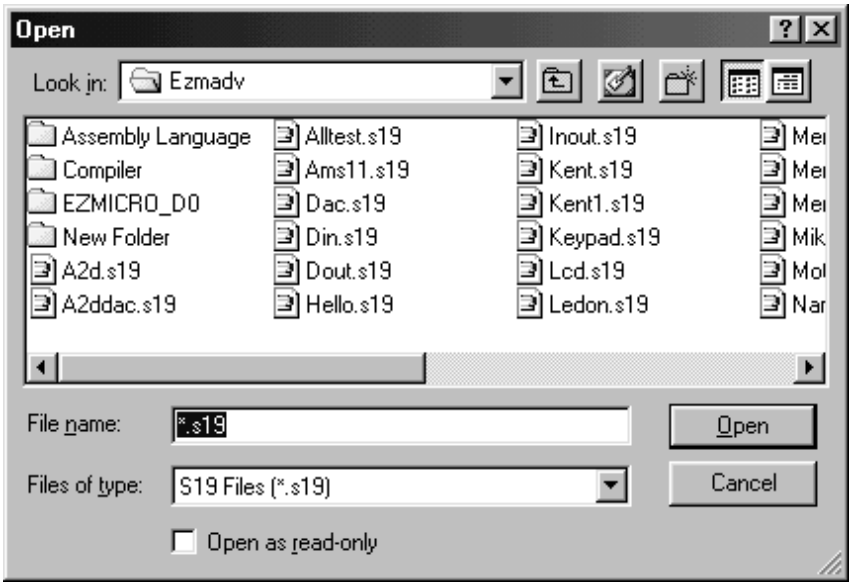


Figure 4.5: Assembled S-Record Hex Files Listing.

Select the S-Record file "LCD.s19" and Press Open. You will see the following dialog box asking to enter Starting Position. This is the address from where the program will start. LCD.ASM does start at 2000. Press OK.

Edit Source

The "Edit Source" command in the FILE menu is used to edit an existing assembly language source file (or any ASCII file).

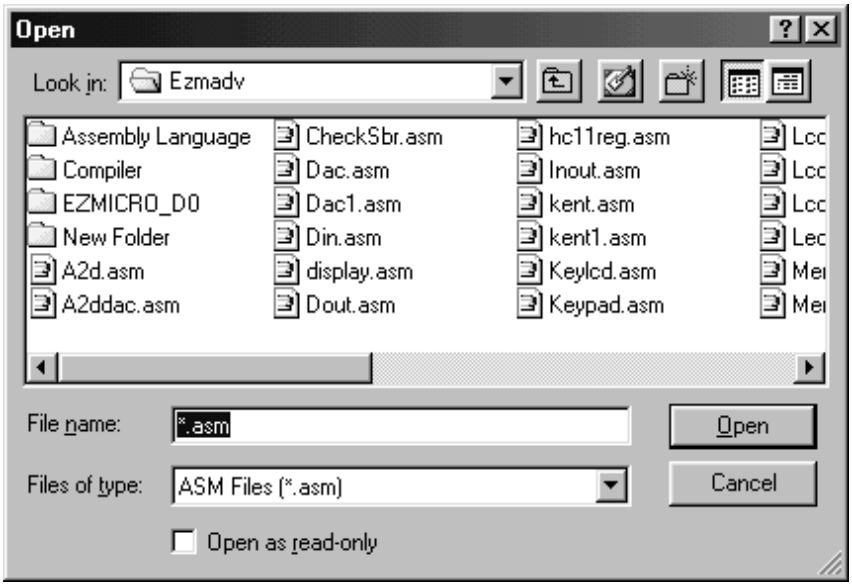


Figure 4.7: Assembly Language Source Files Listing.

Select the file "LCD.asm" and press Open.

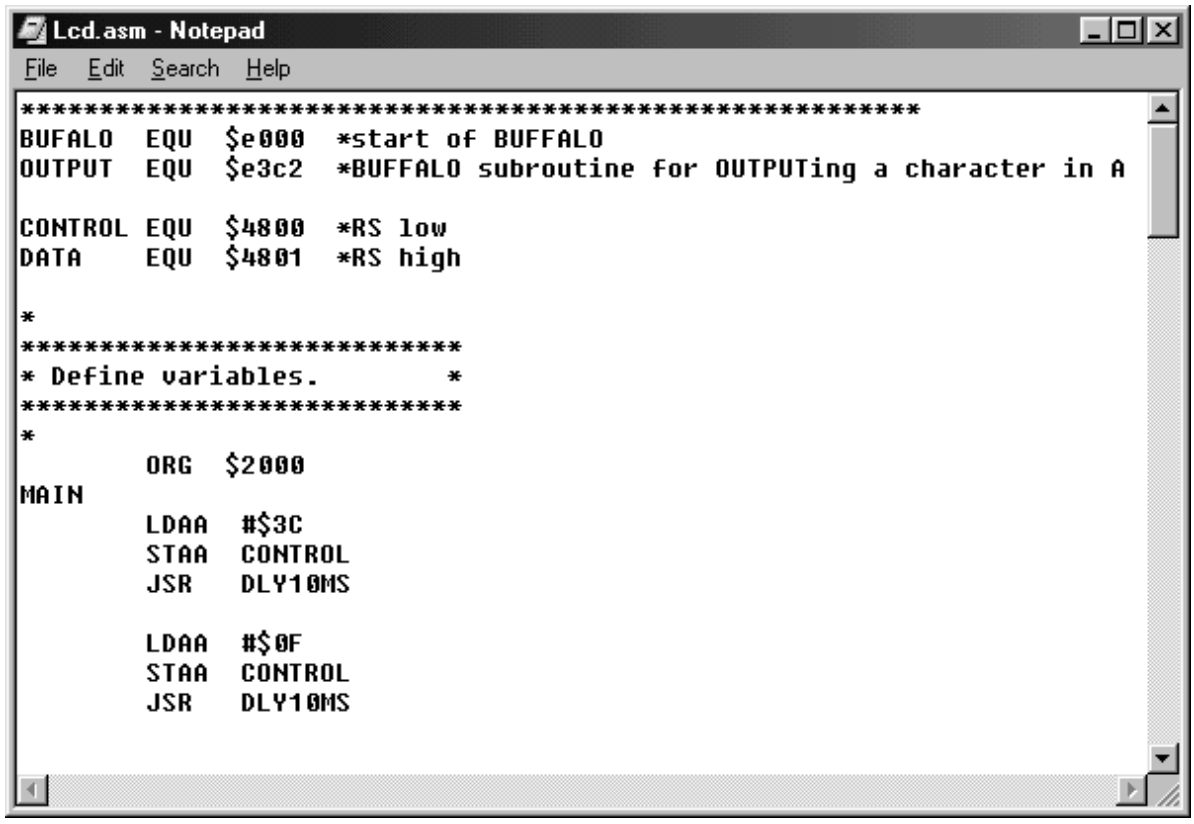


Figure 4.8: Edit Source File Display.

Create Source

The "Create Source" command in the FILE menu is used to create a new assembly language source file.

Configure Port

The "Configure Port" command in the FILE menu is used to configure the COM port (or serial port) of the host computer or terminal for communicating with the EZ-MICRO TUTOR™ microprocessor board.

When the EZ-MICRO TUTOR™ is first powered-on, its serial port will be configured to 9600 baud, no parity, eight bit word length and one stop bit. The options that you select for the host serial port must match these settings (9600,N,8,1) exactly or you will not be able to communicate with the EZ-MICRO TUTOR™. Note: When the EZ-MICRO Manager software is started (in the "normal" mode) for the first time, the Configure Port dialog box will automatically be displayed so that you can select the COM port that you will be using to communicate to the EZ-MICRO TUTOR™ microprocessor board.

When the "Configure Port" command is selected from the SYSTEM menu, a dialog box will appear allowing you to select the port address, baud rate, parity, word length and stop bit parameters. The options that are selected in the Configure Port dialog box will be saved to the EZ-MICRO Manager's configuration file (ezmicro.cfg) when you exit the program normally. That is, by selecting Exit from the FILE menu. Once the serial port options have been selected, the Configure Port dialog box will not be displayed again when the EZ-MICRO Manager software is re-started.

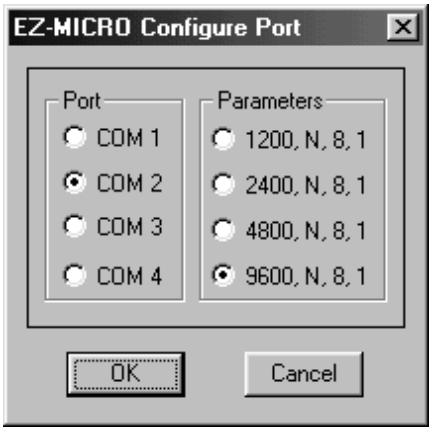


Figure 4.23: Configure Port Options.

It is possible to change the serial port parameters of the EZ-MICRO TUTOR™'s serial interface by configuring the serial port register ($002B). If changes are made to the serial port of the EZ-MICRO TUTOR™, then you will need to select the identical parameters for the host computer's serial port. Note: If a reset occurs on the EZ-MICRO TUTOR™, the serial interface (of the EZ-MICRO TUTOR™) will return to its default value (9600,N,8,1).

Exit

In order to "exit" the EZ-MICRO Manager software normally, select this command. After selecting this command, you will return to the DOS prompt.

Note: If you are evaluating the EZ-MICRO TUTOR™microprocessor board and you are running the EZ-MICRO Manager software in the "normal" mode, then any changes that have been made to the host serial port (ie. baud rate, parity, word length and stop bits) using the Configure Port option will be saved to the configuration file (ezmicro.cfg). These serial port parameters will be used again the next time the program is started.

The Run Menu

The RUN menu contains commands that are used when executing user code on the EZ-MICRO TUTOR™ microprocessor board.

Go

The "Go" command in the RUN menu allows the user to initiate user program execution (free run in real time). The user may optionally specify a starting address where execution is to begin. Otherwise, execution begins at the current program counter (or P-register) address. Program execution continues until a breakpoint is encountered or until the EZ-MICRO TUTOR™ master reset button (S1) is pressed. If a valid breakpoint is not reached by the executed program, then you will need to press the master reset button in order to stop program execution.

Note: The contents of the Registers and Watch Variables windows are only updated after program execution has stopped. Pressing the EZ-MICRO TUTOR™ master reset button will remove the breakpoints that have been entered into the breakpoint address table.

Example: If you are evaluating the EZ-MICRO TUTOR™ microprocessor board then you will want to first assemble the source file "LCD.asm" by selecting the Assemble Source command from the FILE menu. After the source file has been assembled, select Load S-Record from the FILE menu and select the assembled S-Record file "LCD.s19" to be downloaded to the EZ-MICRO TUTOR™ microprocessor board.

Next, select Go from the RUN menu and enter "2000" as the address of the first instruction to be executed. This will begin the execution of the program that we have downloaded to the EZ-MICRO TUTOR™. LCD.ASM program will display a test Pattern 01234567 on first line of LCD and "ABCDEFG" test pattern on the second line.

To stop the program execution, press the master reset button (S1) on the EZ-MICRO TUTOR™ microprocessor board. The EZ-MICRO Manager software will sense that a reset has occurred and will automatically exit from the Go command

To demonstrate the ability of the EZ-MICRO TUTOR™ to stop program execution at a breakpointed address, select the Set Breakpoint command from the DEBUG menu and enter the breakpoint address "2068". This address corresponds to an instruction (in the program that was downloaded to the EZ-MICRO TUTOR™ that .wrties test pattern on first LCD line. Select Go once more from the RUN menu and enter "2000" as the address of the first instruction to be

executed. Notice that the software will display the first line and then MICRO Manager software will sense that a breakpoint has been encountered and program execution will be stopped.

Call

The "Call" command in the RUN menu allows the user to execute a user program subroutine. Execution starts at the current program counter (or P-register) address, unless a starting address is specified. Program execution continues until a breakpoint is encountered or until the EZ-MICRO TUTOR™ master reset button (S1) is pressed. If a valid breakpoint is not reached by the executed program, then you will need to press the master reset button in order to stop program execution.

Note: The contents of the Registers and Watch Variables windows are only updated after program execution has stopped. Pressing the EZ-MICRO TUTOR™ master reset button will remove the breakpoints that have been entered into the breakpoint address table.

Proceed

The "Proceed" command in the RUN menu is used to proceed or continue program execution without having to remove assigned breakpoints. This command is used to bypass assigned breakpoints in a program executed by the GO command. The program will continue to "proceed" to the next assigned breakpoint. If a valid breakpoint is not reached by the executed program, then you will need to press the master reset button in order to stop program execution.
Note: The contents of the Registers and Watch Variables windows are only updated after program execution has stopped. Pressing the EZ-MICRO TUTOR™ master reset button will remove the breakpoints that have been entered into the breakpoint address table.

The EZ-MICRO Manager software will sense when the next valid breakpoint is reached and will automatically exit from the Proceed command

Step (F8 - Hotkey)

The "Step" command in the RUN menu is used to execute each instruction one at a time. Note: Jumper location H8 on the EZ-MICRO TUTOR™microprocessor board must have a jumper strap inserted for this command to function properly. Execution starts at the current program counter (or P-register) address. Each instruction that is executed will be displayed in the Main Display window. To execute the next instruction, simply press the PgDn key on the keyboard (or point at the down arrow button with the mouse and click the left mouse button). The register contents in the Registers display window and the Watch Variables will be updated after each instruction that is executed.

Reset (F12 Hotkey)

The "Reset" command in the RUN menu is used to initiate a hardware reset on the EZ-MICRO TUTOR™ microprocessor board. If you find that the EZ-MICRO Manager software is no longer communicating properly with the EZ-MICRO TUTOR™, then select this command. After selecting this command, press the EZ-MICRO TUTOR™ master reset button (SW1). A message should be displayed stating that a "hardware reset has been detected".

Pressing the master reset button on the EZ-MICRO TUTOR™ will "clear" the current breakpoints that have been set. However, any source code that has been downloaded into user memory will not be affected by this hardware reset.

The Debug Menu

The DEBUG menu contains commands that are used for debugging/executing user code downloaded to the EZ-MICRO TUTOR™ microprocessor board.

Add Variable

The "Add Variable" command in the DEBUG menu is used to add watch variable addresses into the watch variable's address table. A watch variable is simply a memory location on the EZ-MICRO TUTOR™ microprocessor board that you wish to monitor the contents of.

The EZ-MICRO Manager software allows the user to enter up to thirty addresses (or memory locations) into the watch variable's address table. The Save Watches command (in the FILE menu) can be used to save this list of addresses so that they can be quickly re-loaded when the EZ-MICRO Manager software is re-started. This saves the user from having to re-enter the list each time the EZ-MICRO Manager software is run.

Remove Variable

The "Remove Variable" command in the DEBUG menu is used to remove the address of a specific watch variable from the watch variable's address table. When prompted, enter the hex address of the watch variable that you want to remove. Note: This command will be "grayed out" when there are no watch variables currently defined in the watch variable's address table:

Clear All Variables

The first "Clear All" command in the DEBUG menu is used to remove all of the watch variable addresses that have been entered into the watch variable's address table. Note: This command will be "grayed out" when there are no watch variables currently defined in the watch variable's address table.

Set Breakpoint

The "Set Breakpoint" command in the DEBUG menu is used to place an address into the breakpoint address table. When prompted, enter the hexadecimal address of where the breakpoint is to be inserted. A maximum of four breakpoints may be set and will be displayed in the Breakpoints window. If four breakpoints have already been set, the message "Breakpoint table is FULL" will appear when a fifth breakpoint is entered.

Whenever the Go, Call or Proceed commands are selected (from the RUN menu), then "breakpoints" are inserted into the user code at the address specified in the breakpoint address table. During user program execution, the program will stop execution immediately preceding the execution of any instruction's address that is in the breakpoint address table.

Breakpoints are implemented by placing a software interrupt (SWI) at each address specified in the breakpoint address table. The SWI service routine saves and displays the internal machine state, then restores the original opcodes at the breakpoint location before returning control back to the EZ-MICRO Manager program.

Note: SWI opcodes cannot be executed or breakpointed in user code because the BUFFALO monitor program uses the SWI vector. Only RAM locations can be breakpointed. Branch on self instructions cannot be breakpointed.

Remove Breakpoint

The "Remove Breakpoint" command in the DEBUG menu is used to remove one of the breakpoints that have been set in the breakpoint address table. When prompted, enter the address of the breakpoint that is to be removed from this list.

Clear All Breakpoints

The second "Clear All" command in the DEBUG menu is used to remove all of the breakpoints that have been set in the breakpoint address table.

Line Assemble

The "Line Assemble" command in the DEBUG menu is a single line assembler/disassembler that can be used to input code directly into user RAM. Note: This command will be "grayed out" when running the EZ-MICRO Manager software in the demo mode.

When the Line Assemble command is selected, you will be prompted to enter the starting address to begin the "in-line" assembly. Next, you will be prompted to enter a source line. Each source line is converted into the proper machine language code and is stored in memory overwriting previous data on a line-by-line basis at the time of entry.

All valid opcodes are converted to assembly language mnemonics. All invalid opcodes will be displayed as an error message. If an error does occur, then the same address location is re-opened and the source line can be re-entered.

Note: Pressing the ENTER key (without entering anything) will leave the current instruction unchanged and advance to the next valid instruction.

All numerical values are assumed to be hexadecimal. Therefore, no base designators (such as $, %, etc.) are allowed. Operands must be separated by one or more spaces. Any characters after a valid mnemonic are ignored. Addressing modes are designated as follows:

a.) Immediate addressing is designated by preceding the address with a # sign.

b.) Indexed addressing is designated by a comma. The comma must be preceded by a one byte relative offset (even if the offset is 0) and must be followed by an X or Y designation of which index register to use.

c.) Direct and extended addressing is specified by the length of the address operand (1 or 2 digits specifies direct, 3 or 4 digits specifies extended).

d.) Relative offsets for branch instructions are computed by the assembler. Therefore, the valid operand for any branch instruction is the branch-if-true address, not the relative offset.

The Memory Menu

The MEMORY menu contains commands that are used for modifying/displaying user memory located on the EZ-MICRO TUTOR™ microprocessor board.

Memory Dump

The "Memory Dump" command in the MEMORY menu is used to display the hexadecimal contents of user memory (located on the EZ-MICRO TUTOR™ microprocessor board) to the screen.

For example, select the Memory Dump command from the MEMORY menu and you will be prompted to enter the starting memory location to begin the hexadecimal memory dump. Enter the hex address "1000" for this example and press the ENTER key (or click the left mouse button) to accept this hex number. Note: The starting address that is specified may be either RAM or ROM address space (0000h-FFFFh).

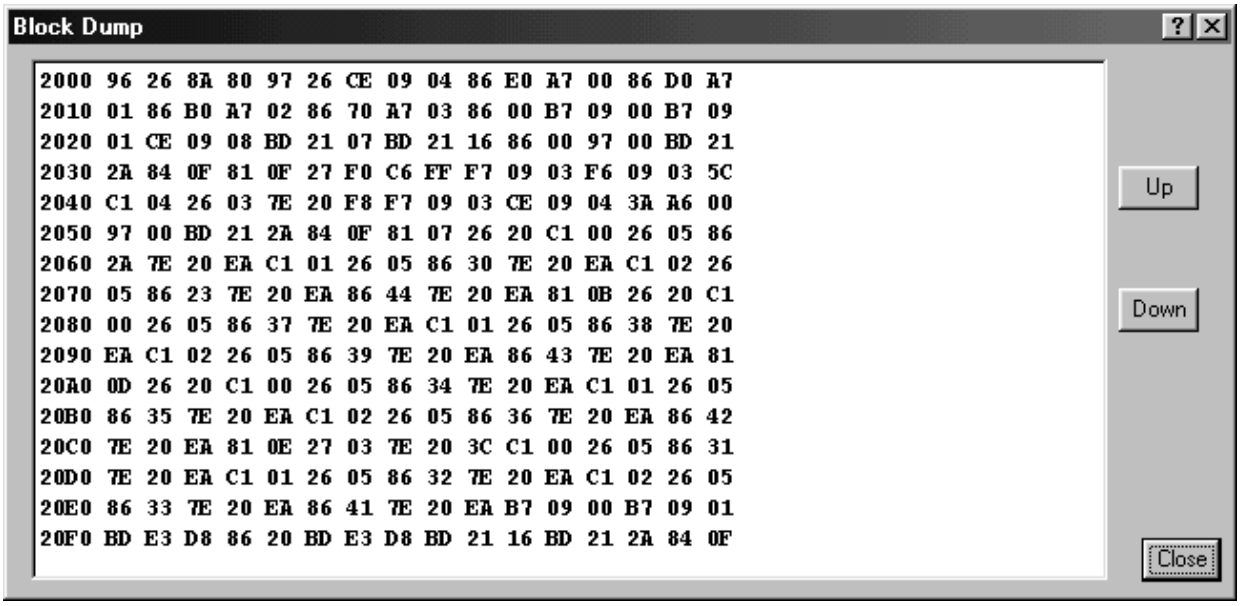


Figure 4.15: Hexadecimal Memory Dump.

Note: The memory addresses will appear as "XX00" only during the demo mode. This is done since there are only 256-bytes of memory (0000h-00FFh) available when running the EZ-MICRO Manager in the demo mode.

Use the PgUp and PgDn keys to view the previous or next page of the hexadecimal memory dump (or point at the up and down arrow buttons with the mouse and click the left mouse button).

Disassemble

The "Disassemble" command in the MEMORY menu is used to display a disassembled listing of the program code to the screen. Note: This command will be "grayed out" when running the EZ-MICRO Manager software in the demo mode.

When the Disassemble command is selected, you will be prompted to enter the address of the first line of code (or instruction) to be disassembled.

Block Fill

The "Block Fill" command in the MEMORY menu allows the user to repeat a specific pattern throughout a determined user memory range. To demonstrate this feature, select the Block Fill command from the MEMORY menu and the following dialog box will be displayed.

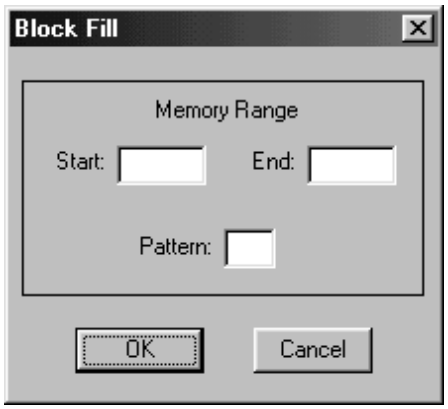


Figure 4.19: Block Fill Memory with Pattern.

Enter "2000" for the starting address location, enter "203F" for the ending address location and enter "AA" for the pattern. Press the OK button to accept the parameters you have just entered.

To view the changes that have been made to the memory locations (2000h-203Fh), select the Memory Dump command from the MEMORY menu and enter "2000" as the starting hex address to begin the memory dump.

Block Move

The "Block Move" command in the MEMORY menu allows the user to copy or move a "block" of memory to a new memory location. To demonstrate this feature, select the Block Move command from the MEMORY menu and the following dialog box will be displayed.

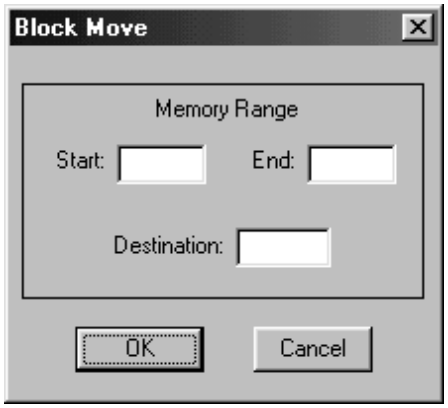


Figure 4.21: Block Move Memory.

Enter "1010" for the starting address location and enter "103F" for the ending address location of the block of memory to move. Enter "1050" for the destination hex address to receive the block (48-bytes) of memory. Press the OK button to accept the parameters you have just entered.

To verify that the block of memory (2000h-203Fh) was copied to 2050h, select the Memory Dump command from the MEMORY menu and enter "2000" as the starting hex address to begin the memory dump.

The HELP Menu

The HELPin HELP menu provide trhe help on the software

About EZ-MICRO

Selecting this command will display the current version of the EZ-MICRO Manager software that is currently running.

Section Two:

EZ-MICRO TUTOR™ Hardware Preparation & Installation

This section provides general information on EZ-MICRO TUTOR™ board, hardware preparation, hardware installation, operating instructions, hardware description and support information.

Features

The EZ-MICRO TUTOR™microprocessor board includes the following features:

- MC68HC11E0 CPU in a 52 pin PLCC package.
- 192 Bytes On-Chip RAM.
- RS-232C compatible I/O interface port.
- Single input power supply to provide all voltage requirements.
- Real-Time interrupt circuit.
- 16-Bit Timer System – 3 Input Capture (IC) Channels / 4 Output Compare (OC) Channels. Additional Channel is software selectable as either Fourth IC or Fifth OC.
- Computer Operating Properly (COP) Watchdog System.
- Synchronous Serial Peripheral Interface (SPI).
- Asynchronous Nonreturn to Zero (NRZ) Serial Communications Interface (SCI).
- 8-Bit Pulse Accumulator.
- 26 Input/Output (I/O) Pins.
- 32 K Bytes Ststic RAM
- 32 K Bytes of EPROM with Monitor Software
- 8 Bit Digital to Analog Converter
- 8 Channels, 8 Bit Analog to Digital converter
- 4X4 Hex Keypad
- 2 Line, 16 Characters Alpha numerical LCD Display.
- 8 Bit Digital Input port, with 4 lines connected to switch
- 8 Bit Digital outout port with 4 lines connected to LEDS

Specifications

The following table lists the specifications of the EZ-MICRO TUTOR™microprocessor board.

CHARACTERISTIC	SPECIFICATION
CPU	MC68HC11E0
Terminal I/O Port	RS-232C compatible
Temperature: Operating Storage	 +25 Degrees C -40 to +85 Degrees C
Relative Humidity	0 to 90% (non-condensing)
Dimensions: Width Length	7 Inch 6 Inch

TABLE 4.1: EZ-MICRO TUTOR BOARD Specifications

General Description

The EZ-MICRO TUTOR™microprocessor board provides an easy way of learning the MC68HC11E0 Motorola microprocessor. The MC68HC11E0 is a high-performance micro controller unit (CPU) that is based on the MC68HC11E9 design. The MC68HC11E0 chip offers high speed, low power consumption and multiplexed buses capable of running up to 3 Mhz.

The MC68HC11E0 is also an excellent economical alternative device for applications that require the HC11 CPU but where fewer peripheral functions and less memory are required. Refer to the Motorola MC68HC11 CPU data sheet (see appendix) for additional device information.

An optional wire-wrap section can be provided with the EZ-MICRO TUTOR™microprocessor board to provide an area for custom interfacing to the CPU. The wire-wrap section can be detached from the EZ-MICRO TUTOR™so that a new wire-wrap section or one of the other EZ-MICRO TUTOR™Lab Board interfacing projects may be connected. This allows for the EZ-MICRO TUTOR™microprocessor board to be used over and over again

Hardware Preparation

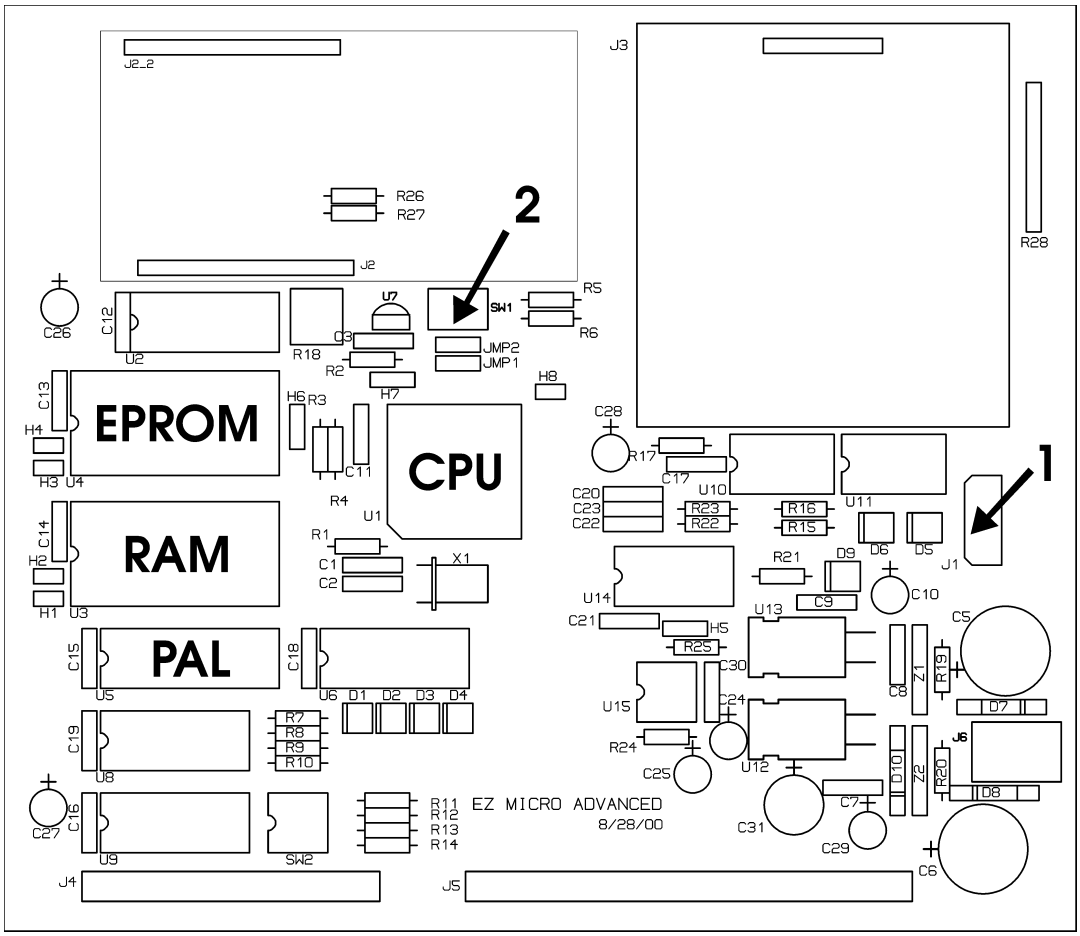
Unpacking Instructions

If the package is damaged upon receipt, request that the carrier's representative be present during the unpacking and inspection of the EZ-MICRO TUTOR™microprocessor board. The following materials should be included in the EZ-MICRO TUTOR™microprocessor “evaluation” package:

- **EZ-MICRO TUTOR™ Microprocessor Board.**
- **9 Volt AC Power supply**.
- **RS-232 Serial Cable (with 9 pin “DB” connectors)**.
- **EZ-MICRO Manager Software CD-ROM**

Hardware Preparation

This portion of the manual describes the inspection and initial jumper selection for the EZ-MICRO TUTOR™ microprocessor board which is shown in the following figure (Fig. 4.27).



1. RS232 Connector: Connect to IBM PC compatible computer serial port

2. Reset Switch

Figure 4.27: EZ-MICRO TUTOR™Microprocessor Board.

The AC power supply adapter (provided) should be connected to the power input connector J6. Proper connection to the AC power supply adapter will be confirmed by the power indicator LED at D9.

The DB-9 (nine pin male) connector J1 has been provided for asynchronous serial communication with an external terminal or host computer. Note: Do not use a "NULL modem" cable when connecting the EZ-MICRO TUTOR™ microprocessor board to the external terminal or host computer. This may damage one or both of the interface ports.

The sixty pin (2X30), double row, right angle connector J5 and forty pin (2 X 20) double row, right angle connector J4 facilitate the interconnection of the EZ-MICRO TUTOR™ microprocessor board to the wire-wrap area or any other external circuits.

The EZ-MICRO TUTOR™microprocessor board has a built-in "power-on" reset logic that will reset the CPU at initial power-up of the device, however, there is also an external user reset button provided at switch location SW1(Labeled 2 in Figure 4-27).

Depending on the user's application, you may need to insert or remove jumper straps in order to configure the operation of the EZ-MICRO TUTOR™microprocessor board. The following sections describe each jumper position and its function. These jumper locations are shown in Figure 4.27.

CPU Mode Select headers (H6 and H7)

The EZ-MICRO TUTOR™microprocessor board can be configured for either the single-chip, expanded-multiplexed, special bootstrap or special test modes of operation via jumper locations H6 and H7. The following table shows the possible modes of operation for EZ-MICRO TUTOR™microprocessor board.

MODA	MODB	
H6	**H7**	**CPU Mode Select**
OUT	OUT	Expanded-Multiplexed
OUT	IN	Special Test
IN	OUT	Single Chip
IN	IN	Special Bootstrap

RS232 Data Interface

Electronic data communications between elements will generally fall into two broad categories: single-ended and differential. RS232 (single-ended) was introduced in 1962, and despite rumors for its early demise, has remained widely used through the industry.

Independent channels are established for two-way (full-duplex) communications. The RS232 signals are represented by voltage levels with respect to a system common (power / logic ground). The "idle" state (in technical term often called MARK) has the signal level negative with respect to common, and the "active" state (in technical term often called SPACE) has the signal level positive with respect to common. RS232 has numerous handshaking lines (primarily used with modems, mouse, scanners etc.), and also specifies a communications protocol.

The RS-232 interface presupposes a common ground between the DTE and DCE. This is a reasonable assumption when a short cable connects the DTE to the DCE, but with longer lines and connections between devices that may be on different electrical busses with different grounds, this may not be true.

RS232 data is bi-polar.... +3 TO +12 volts indicates an "ON or 0-state (SPACE) condition" while A -3 to -12 volts indicates an "OFF" 1-state (MARK) condition.... Modern computer equipment ignores the negative level and accepts a zero voltage level as the "OFF" state. In fact, the "ON" state may be achieved with lesser positive potential. This means circuits powered by 5 VDC are capable of driving RS232 circuits directly, however, the overall range that the RS232 signal may be transmitted/received may be dramatically reduced.

The output signal level usually swings between +12V and -12V. The "dead area" between +3v and -3v is designed to absorb line noise. In the various RS-232-like definitions this dead area

may vary. For instance, the definition for V.10 has a dead area from +0.3v to -0.3v. Many receivers designed for RS-232 are sensitive to differentials of 1v or less.

This can cause problems when using pin powered widgets - line drivers, converters, modems etc. These type of units need enough voltage & current to power them self's up. Typical URART (the RS-232 I/O chip) allows up to 50ma per output pin — so if the device needs 70ma to run we will need to use at least 2 pins for power. Some devices are very efficient and only require one pin (some times the Transmit or DTR pin) to be high — in the "SPACE" state while idle.

The following FIG 1 shows standard 25 pin RS232 connector and pin description. All AMS products uses 9 pin RS232 interface shown in FIG 4.28.

Details of Hardware Design:

The following sections describe each hardware section on the EZ-MICRO TUTOR™ board.

RS-232 Interface

RS-232 (EIA Std) applicable to the 25 pin interconnection of Data Terminal Equipment (DTE) and Data Communications Equipment (DCE) using serial binary data.

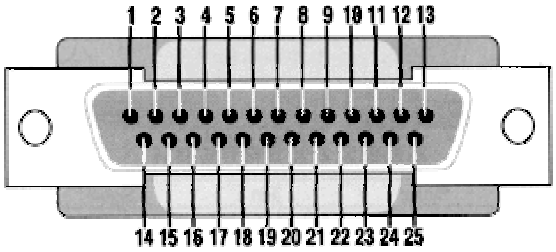


Figure 4.28 25 pin interconnection

Pin No.	Description
1	Protective Ground
2	Transmit Data
3	Receive Data
4	Request To Send
5	Clear To Send
6	Data Set Ready
7	Signal Ground
8	Receive Line Signal Detect/Carrier Detect
9	Reserved
10	Reserved
11	Unassigned
12	Secondary RLSD
13	Secondary CTS

Pin No.	Description
14	Secondary TD
15	Transmitter Signal Element Timing
16	Secondary RD
17	Receiver Signal Timing Element
18	Unassigned
19	Secondary RTS
20	Data Terminal Ready
21	Signal Quality Detector
22	Ring Indicator
23	Data Signal Rate Selector
24	Transmit Signal Element Timing
25	Unassigned

Table 4.5: 25 PIN Connector

Signal Description:

CTS	Clear To Send [DCE --> DTE]
DCD	Data Carrier Detected (Tone from a modem) [DCE --> DTE]
DCE	Data Communications Equipment eg. Modem
DSR	Data Set Ready [DCE --> DTE]
DSRS	Data Signal Rate Selector [DCE --> DTE] (Not commonly used)
DTE	Data Terminal Equipment eg. computer, printer
DTR	Data Terminal Ready [DTE --> DCE]
FG	Frame Ground (screen or chassis)
NC	No Connection
RCk	Receiver (external) Clock input
RI	Ring Indicator (ringing tone detected)
RTS	Ready To Send [DTE --> DCE]
RxD	Received Data [DCE --> DTE]
SG	Signal Ground
SCTS	Secondary Clear To Send [DCE --> DTE]
SDCD	Secondary Data Carrier Detected (Tone from a modem) [DCE --> DTE]
SRTS	Secondary Ready To Send [DTE --> DCE]
SRxD	Secondary Received Data [DCE --> DTE]
STxD	Secondary Transmitted Data [DTE --> DTE]
TxD	Transmitted Data [DTE --> DTE]

Table 4.6 RS232D (9 pin Connector)

SIGNAL	PIN No.
Carrier Detect	1
Receive Data	2
Transmit Data	3
Data Terminal Ready	4
Signal Ground	5
Data Set Ready	6
Request To Send	7
Clear To Send	8
Ring Indicator	9

Table 4.7 RS232D Pin Description

RS-232 Signal Descriptions

The interface transfers data between the computer and the modem via the TD and RD lines. The other signals are essentially used for FLOW CONTROL, in that they either grant or deny

requests for the transfer of information between a DTE and a DCE. Data cannot be transferred unless the appropriate flow control line are first asserted.

The interface can send data either way(DTE to DCE, or, DCE to DTE) independently at the same time. This is called FULL DUPLEX operation. Some systems may utilize software codes so that information may only be transmitted in one direction at a time (HALF DUPLEX), and requires software codes to switch from one direction to another (i.e., from a transmit to receive state).

The following is a list of common RS232 signals.

Request To Send (RTS) This signal line is asserted by the computer (DTE) to inform the modem (DCE) that it wants to transmit data. If the modem decides this is okay, it will assert the CTS line. Typically, once the computer asserts RTS, it will wait for the modem to assert CTS. When CTS is asserted by the modem, the computer will begin to transmit data.

Clear To Send (CTS) Asserted by the modem after receiving a RTS signal, indicating that the computer can now transmit.

Data Terminal Ready (DTR) This signal line is asserted by the computer, and informs the modem that the computer is ready to receive data.

Data Set Ready (DSR) This signal line is asserted by the modem in response to a DTR signal from the computer. The computer will monitor the state of this line after asserting DTR to detect if the modem is turned on.

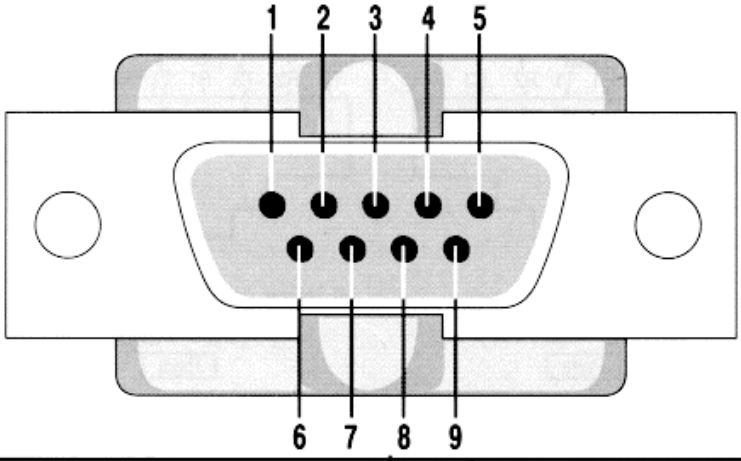
Receive Signal Line Detect (RSLD) This control line is asserted by the modem, informing the computer that it has established a physical connection to another modem. It is sometimes known as **Carrier Detect (CD)**. It would be pointless a computer transmitting information to a modem if this signal line was not asserted. If the physical connection is broken, this signal line will change state.

Transmit Data (TD) is the line where the data is transmitted, a bit at a time.

Receive Data (RD) is the line where data is received, a bit at a time. A lot of signals work in pairs. Some signals are generated by the DTE, and some signals are generated by the DCE. If you were measuring the signals on a computer which was NOT connected to a modem, you could only expect to see those signals that the DTE can generate.
The following table lists some of the signal pairs and the device responsible for generating them.

DTE	DCE
TD	RD
RTS	CTS
DTR	DSR

Table 4.8 Signal Pairs



Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

Figure 4.29 Pin Assignments

Data is transmitted and received on pins 2 and 3 respectively. Data Set Ready (DSR) is an indication from the Data Set (i.e., the modem or DSU/CSU) that it is on. Similarly, DTR indicates to the Data Set that the DTE is on. Data Carrier Detect (DCD) indicates that a good carrier is being received from the remote modem.

Pins 4 RTS (Request To Send - from the transmitting computer) and 5 CTS (Clear To Send - from the Data set) are used to control. In most Asynchronous situations, RTS and CTS are constantly on throughout the communication session. However where the DTE is connected to a multi-point line, RTS is used to turn carrier on the modem on and off. On a multi-point line, it's imperative that only one station is transmitting at a time (because they share the return phone pair). When a station wants to transmit, it raises RTS. The modem turns on carrier, typically waits a few milliseconds for carrier to stabilize, and then raises CTS. The DTE transmits when it sees CTS up. When the station has finished its transmission, it drops RTS and the modem drops CTS and carrier together.

Clock signals (pins 15, 17, & 24) are only used for synchronous communications. The modem or DSU extracts the clock from the data stream and provides a steady clock signal to the DTE. Note that the transmit and receive clock signals do not have to be the same, or even at the same baud rate.

Note: Transmit and receive leads (2 or 3) can be reversed depending on the use of the equipment - DCE Data Communications Equipment or a DCE Data Terminal Equipment.

RS-232 Specifications

SPECIFICATIONS		RS232
Mode of Operation		Single Ended
Total Number of Drivers and Receivers on One Line		1 DRIVER 1 RECVR
Maximum Cable Length		50 FT.
Maximum Data Rate		20kb/s
Maximum Driver Output Voltage		+/-25V
Driver Output Signal Level (Loaded Min.)	**Loaded**	+/-5V to +/-15V
Driver Output Signal Level (Unloaded Max)	**Unloaded**	+/-25V
Driver Load Impedance (Ohms)		3k to 7k
Max. Driver Current in High Z State	**Power On**	N/A
Max. Driver Current in High Z State	**Power Off**	+/-6mA @ +/-2v
Slew Rate (Max.)		30V/uS
Receiver Input Voltage Range		+/-15V
Receiver Input Sensitivity		+/-3V
Receiver Input Resistance (Ohms)		3k to 7k

Table 4.9 RS-232 Specifications

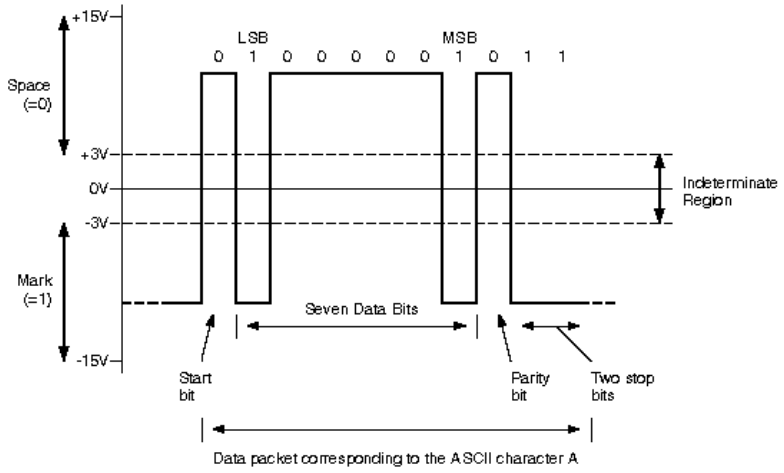


Figure 4.30 One byte of async data

Input Port

The fundamental technique for interfacing I/O devices to a microprocessor is to interconnect the devices to the processor's address, data and control buses. The processor uses the buses to send and receive data from memory and peripheral devices. The input port allows the microprocessor to read in an 8-bit binary number which will be set to a '1' or a '0' by the toggle switches on the EZ-MICOR TUTOR™ board. There are two types of input ports on EZ-MICOR TUTOR™ board. The micorcontroller 68HC11E0 has built-in Input/output ports such as Port A and external input port using 74LS373 (U9). The 74LS373 transparent latch is used to isolate the 8-bit input value from the data bus in order to prevent bus contention. Port A is connected to 4X4 Matrix Keypad and 74LS373 (U9) is connected input swithc SW2. The input port must be memory-mapped into the address space. The 8-bit input value should be enabled onto the data bus only on a CPU read operation from the memory address range designated for the input port. This is explained in deatil in Chapter 9.

Output Port

There are two types of output ports on EZ-MICOR TUTOR™ board. The micorcontroller 68HC11E0 has built-in Input/output ports such as Port A and external output port using 74LS374(U6 and U8). The 74LS374 transparent latch is used to isolate the 8-bit output value from the data bus in order to prevent bus contention. Port A is connected to 4X4 Matrix Keypad and 74LS374 (U6) is connected to Digital to Analog converter U14 and 74LS374 (U8) output latch is connected to LEDs. The output port allows the microprocessor to write out an 8-bit binary number which will be latched by the 74LS374 D-type flip-flops. The output port must be memory-mapped into the address space. The 8-bit output value should be grabbed from the data bus only on a CPU write operation from the memory address range designated for the output port. This is explained in deatil in Chapter 9.

Address decoding and control logic

The 22V10 (U4 on the board) device is a programmable logic device designed to implement custom logic functions similar in complexity to the 74 series of logic chips. It has the advantage of delivering customized functions that might be difficult to implement or take multiple chips using 74 series chips. Programmable array logic (PAL) chips and general array logic (GAL) chips, such as the GAL22V10, provide a convenient means for implementing combinatorial logic functions of moderate complexity. For example, due to its large number of available inputs, the GAL22V10 is useful for address decoding in microprocessor circuits. The GAL22V10 can also be used to implement small registers, counters or simple sequential circuits since it contains 10 flip-flops clocked by a common input signal. For detailed information on the GAL22V10 refer to the data sheets available on the Lattice Semiconductor web page (http://www.lattice.com).

In the EZ-MICRO TUTOR™ system, a GAL22V10 is used to implement the control logic needed to activate the input and output ports and all other peripherals at the appropriate times. The GAL22V10 will decode the address and control signals presented on the buses to determine

when the peripheral should be selected. The Boolean equations are written in PALASM and compiled into a JEDEC file for programming the PAL chip.

At the simplest level, PALASM allows the user to enter Boolean equations that describe an output signal as a sum-of-products function of the input signals. PALASM also includes the higher level "if-then" type statements that can be used in implementing state machines. Complete information on PALASM's state machine language can be found in AMD's PALASM User's Manual. (http://www.amd.com)

A PALASM design file is a simple text file with the extension "pds". The pds file for EZ-MICRO TUTOR™ is included below. PALASM's Boolean operators are:

/ NOT logical operator
*** AND logical operator**
+ OR logical operator
= Output equation operator
; Comment delimiter

The Pin statements allow you to specify the names and attributes of the pins used in your design. Each pin that you use in your design must be declared with pin statement. The pds file for the EZ-MICRO TUTOR™ Gal is shown below. [attach actual file for this PAL]

```
TITLE       68HC11 Eval Board.. U5 8-24-2000
PATTERN     REV 1.5
REVISION        1.5

AUTHOR      AMS Copyright 2000

COMPANY     AMS, INC POMPANO BCH., FL
DATE        07/14/2000

CHIP        U5          PAL22V10

; Address decoder for Micro...
; DIN/DOUT = 0x40XX / 0x41XX
; LCD      = 0x42XX / 0x43XX

CLK A10 A11 A12 A13 A14 RW E NC1 NC2 NC3 GND
A15 DOUT DING DAC_O RAMW DINOC LCD_RW LCD_E RAM ROM OE VCC GLOBAL
EQUATIONS
/OE = E * RW
/RAM = /A15 * /A14 * E
/ROM = A15 * E
/RAMW = E * /RW
DING = /A15 * A14 * /A13 * /A12 * /A11 * /A10 * RW * E
DINOC = /DING
/DOUT = /A15 * A14 * /A13 * /A12 * /A11 * A10 * /RW * E
LCD_RW = /A15 * A14 * /A13 * /A12 * A11 * RW
LCD_E = /A15 * A14 * /A13 * /A12 * A11 * E
/DAC_O = /A15 * A14 * /A13 * A12 * /A11 * /A10 * /RW * E
```

DAC

Most transducers and sensors produce signals that are continuous or analog in nature. For example, sensors that are used to measure force, velocity, acceleration, temperature, volume, weight and proximity typically produce analog voltages or currents. In addition, an electronic system designer may need to produce an analog output signal, such as in a speech synthesis system. Data conversions systems, namely analog-to-digital (A to D) and digital-to-analog (D to A) converters, allow designers to interface digital microprocessors to an analog world.

The DAC0802 is an 8-bit current-output digital-to-analog converter (DAC) from National Semiconductor (http://www.national.com). The detail datasheet is included on the CD-ROM. The DAC converts a binary digital number into an analog signal. Each bit is assigned according to its position (MSB to LSB), a voltage, or a current. For example, the most significant bit (MSB) in an 8 bit number (1000 000) is assigned 128 times the current of the least significant bit (LSB:0000 0001). The sum of all the currents from all the "ON" bits is added and converted to a proportional voltage.

In the EZ-MICRO TUTOR™ system, the operational amplifier U15 converts the unipolar current output from the DAC into a single-ended voltage output. The reference current in the DAC is set by R22. In the EZ-MICRO TUTOR™ system, the reference current is set to 5V / 4.7K = -1.06 mA. The transconductance gain of the op amp is set by R25 and is -(2.7K)(-1.06).

The DAC is configured to operate with a voltage range of 0-5 volts corresponding to the eight-bit values of 0x00-0xFF. Actually, the analog output range of the DAC will be from 0 to 5*(255/256) V.

Vout=(5V/R22)*R25

ADC

The analog-to-digital converter is on-board the HC11E9.The A/D converter is a successive approximation converter; the successive approximation A/D is the most popular type of converter. In this A/D, each digital bit starting from the most significant bit (MSB) is in analog form compared to the analog input voltage. If the analog voltage of this bit is less than the incoming analog voltage, this bit value is retained for the digital output and added to the previous one. Otherwise, it is discarded. The conversion ends when the least significant bits (LSB) have been compared.

The A/D system is an 8-channel, 8-bit, multiplexed input converter. The converter does not require an external sample and hold circuit because the circuit uses an all-capacitive charge redistribution technique to convert analog signals to digital values. A/D converter timing can be synchronized to the system E clock or to an internal resistor capacitor (RC) oscillator.

The A/D converter system consists of four functional blocks: multiplexer, analog converter, digital control, and result storage. We will discuss each block.

Multiplexer – The multiplexer selects one of 16 inputs for conversion. Input selection is controlled by the value of bits CD-CA in the ADCTL register. The eight port E pins are fixed direction analog inputs to the multiplexer, and additional internal analog signal lines are routed

to it. Port E pins can also be used as digital inputs. Digital reads of port E are not recommended during the sample portion of the A/D conversion cycle.

Analog converter – Conversion of an analog input selected by the multiplexer occurs in this block. It contains a digital-to-analog (DAC) array, a comparator and a successive approximation register (SAR). Each conversion is a sequence of eight comparison operation, beginning with the most significant bit (MSB). Each comparison determines the value of a bit in the successive approximation register.

The DAC array performs two functions. It acts as a sample and hold circuit during the entire conversion sequence, and provides comparison voltage to the comparator during each successive comparison. The result of each successive comparison is stored in the SAR. When a conversion sequence is complete, the contents of the SAR are transferred to the appropriate result register. Digital control – All A/D converter operations are controlled by bits in register ADCTL. In addition to selecting the analog input to be converted, ADCTL bits indicate conversion status, and control whether single or continuous conversions are performed. Finally, the ADCTL bits determine whether conversions are performed on single or multiple channels.

Result registers – Four 8-bit registers ADR[4:1] store conversion results. Each of these registers can be accessed by the processor in the CPU. The conversion complete flag (CCF) indicates when valid data is present in the result registers.

A/D converter clocks – The CSEL bit in the OPTION register select whether the A/D converter uses the system E clock or an internal RC oscillator for synchronization. When E-clock frequency is below 750 kHz, charge leakage in the capacitor array can cause errors, and the internal oscillator should be used. When the RC clock is used, additional errors can occur because the comparator is sensitive to the additional system clock noise.

Conversion Sequence – A/D converter operations are performed in sequences of four conversions each. A conversion sequence can repeat continuously or stop after one iteration. The conversion complete flag (CCF) is set after the fourth conversion in a sequence to show the availability of data in the result registers.

An input voltage equal to V_{RL} converts to $00 and an input voltage equal to V_{RH} converts to $FF (full scale), with no overflow indication.

Make sure that your input signal to the ADC is always within the 0-5V range.

LCD

The LCD module is an off-the-shelf display module based on Hitachi's HD44780U Dot Matrix Liquid Crystal Display Controller/Driver. (http://www.hitachi.com) The LCD module displays alphanumeric characters from a character generator ROM and drives a dot-matrix liquid crystal display. Each character is formed within a 5 x 8 dot matrix. The display is 16 characters wide and two lines deep.

The LCD module can interface directly to the microprocessor bus. The enable logic comes from the address decode logic inside the GAL22V10.

Only the instruction register (IR) and the data register (DR) can be controlled by the microprocessor. There are four categories of instruction:

1. **designate LCD functions, such as display format, data length, etc.**
2. **set internal RAM addresses within the LCD module**
3. **perform data transfer with internal RAM in the LCD module**
4. **perform other miscellaneous functions**

Normally, instructions that perform data transfer with internal RAM in the LCD module are used the most. However, auto-increment of LCD RAM addresses after each data write can lighten the program load of the MPU. Since the display instruction can perform concurrently with display data write, the user can minimize system development time with maximum programming efficiency.

Control instructions

```
     Clear display       $01
     Return home         $02
     Entry mode set      $06
     Display on/off
     Cursor shift
     Function set
```

Instruction Description

```
     Clear display
     Clear display writes space code (20h)
```

For the 2-line display, the cursor automatically moves from the first to the second line after the 40th digit of the first line has been written. Thus, since there are only 16 characters in the first line, the address must be again set after the 16th character is completed.

Display character address code

```
          1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
1 line    80 81 82 83 84 85 86 87 C0 C1  C2  C3  C4  C5  C6  C7


          1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
2 line    80 81 82 83 84 85 86 87 88 89  8A  8B  8C  8D  8E  8F
          C0 C1 C2 C3 C4 C5 C6 C7 C8 C9  CA  CB  CC  CD  CE  CF
```

NOTE: Appendix and CD-ROM has more information on LCD programming

Keypad

The EZ-MICRO TUTOR™ hardware supports a 4x4 keypad. The keypad has four rows connected to the lower 4 bits of Port A and four columns connected to the upper 4 bits of Port A. The rows are always outputs; the columns are always inputs. All 16 keys on the keypad are normally open. The four columns are inputs to the processor and are pulled up to +5V through 4.7K resistors on the EZ-MICRO TUTOR™ board. The upper 4 bits of Port A will be read as '1's when no key is pressed.

The microprocessor scans the 4x4 keypad in order to detect key presses by the user. Here's how the algorithm works. The processor drives ROW1 low while leaving the other three row lines high. Next the processor reads the upper nibble of Port A. A low detected on any one of the four lines indicates that a column key has been pressed. This procedure is repeated for each of the four rows.

Note that the EZ-MICRO TUTOR™ monitor uses PA3 to support the Trace function in the debugger. The low level keypad routines also use PA3 as one of the row output lines. For this reason, the user may not Trace or Single Step through the keypad scanning routine.

Installation Instructions

The EZ-MICRO TUTOR™microprocessor board has been designed for table top operation. An optional RS-232C compatible host computer can be used for downloading assembled code to the EZ-MICRO TUTOR™microprocessor board via the I/O port connector J1. Note: Do not use a "NULL modem" cable when connecting the EZ-MICRO TUTOR™microprocessor board to the host computer. This may damage one or both of the serial ports.

Power Supply Connection

The EZ-MICRO TUTOR™ microprocessor board requires +5V, -12V, +12V and ground (GND) for normal operation. The AC power supply unit that is provided is connected to the power supply connector J6 in the EZ-MICRO TUTOR™ microprocessor board. Proper connection of the power supply unit will be confirmed by the power indicator LED (D9).

Terminal I/O Port (RS-232C)

Interconnection of EZ-MICRO TUTOR™microprocessor board with PC compatible computer is accomplished via the supplied RS-232 serial cable. The EZ-MICRO TUTOR™ microprocessor board is wired as data communication equipment (DCE) which will allow for a "straight through" serial cable to be used. The following figure shows the pin assignments of the DB-9 male serial port connector (J1) of the EZ-MICRO TUTOR™ microprocessor board.

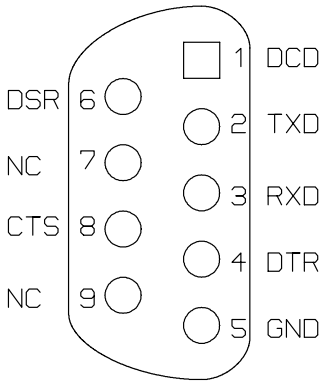


Figure 4.31: DB-9 Pin Configuration

If your application requires a DB-25 male pin configuration, then an optional DB-9 to DB-25 converter may be used and is shown in the following figure.

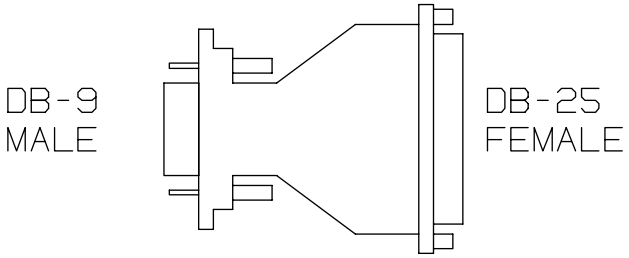


Figure 4.32: DB-25 to DB-9 Converter

Wire-Wrap and EZ-MICRO Labs CPU Interconnection

In order to interface the EZ-MICRO TUTOR™microprocessor board with other project boards, a 60-pin right angle connector (JA1) has been provided. Pin numbers one to forty-four on this connector (JA1) correspond exactly to pins one to forty-four of the MC68HC11E0 CPU. The signal names have been clearly marked next to each pin of connector JA for your convenience (for probing and wire wrapping access). The top and bottom views of this connector is shown in the following figures for your reference.

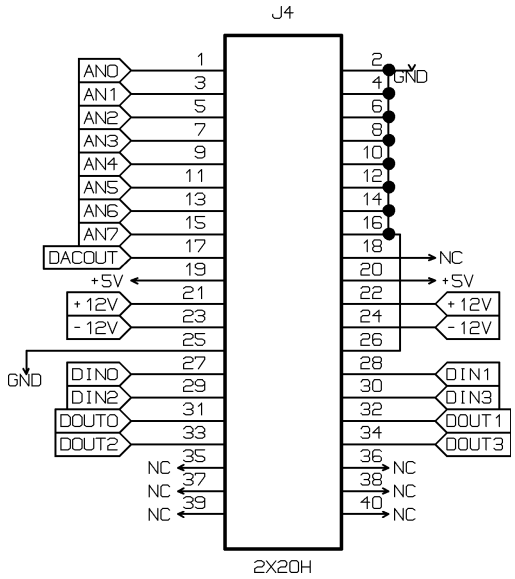


Figure 4.33: EZ-MICRO TUTOR™ Interface Connector (J4) Top View

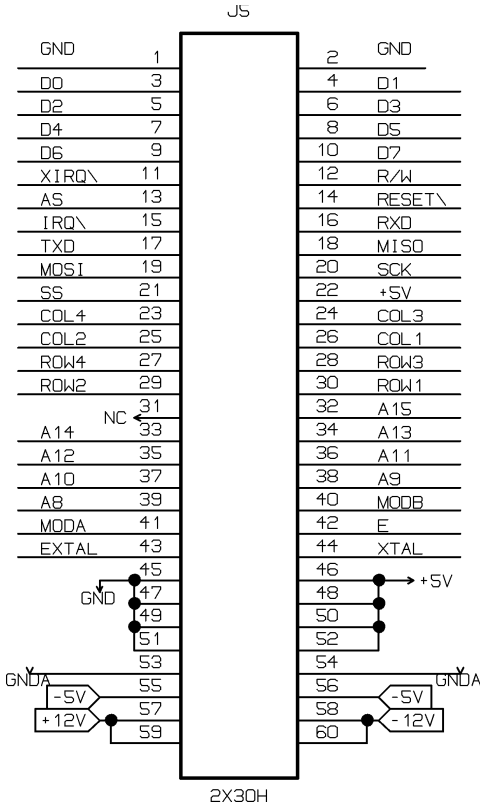


Figure 4.34: EZ-MICRO TUTOR™ Interface Connector (J5) Bottom View

Lesson 4 Questions

SECTION ONE

1) What is the purpose of the EZ-MICRO Manager software ?

2) What should you always do before starting the EZ-MICRO Manager software ?

3) Where is the Action Bar located and what is its purpose ?

4) What type of information does the Status Line display ?

5) How can someone get additional information concerning a particular command in one of the pull down menus ?

6) How does the EZ-MICRO Manager software communicate to the EZ-MICRO TUTOR™ board ?

7) What is the default baud rate of the serial port interface on the EZ-MICRO TUTOR™ board ?

8) How can someone modify the contents of the EZ-MICRO TUTOR™ registers when using the EZ-MICRO Manager software ?

9) What steps must be taken in order to download a program to the EZ-MICRO TUTOR™ board?

10) What type of hex file is produced by the EZ-MICRO Manager assembler ?

11) How does someone begin execution of a program that is downloaded to the EZ-MICRO TUTOR™ board ?

12) Explain two ways execution of a user program can be stopped ?

13) Why would someone want to trace the program execution ?

SECTION TWO

1) What type of CPU is used on the EZ-MICRO TUTOR™ board ?

2) What is the DB-9 connector used for on the EZ-MICRO TUTOR™ board ?

3) What is the purpose of the 60 pin connector (J5) on the EZ-MICRO TUTOR™ board ?

4) What mode of operation is being used on the EZ-MICRO TUTOR™ board and how do you know ?

5) Should you use a NULL modem serial cable to connect the EZ-MICRO TUTOR™ board to the host terminal or computer ?

6) The RS232-C standard specifies a 9 pin male connector for the DTE.

   TRUE or FALSE

7) In RS232, the signal Carrier Detect (CD) indicates the modem is turned on and has received a signal from another modem.

   TRUE or FALSE

8) In RS232, when a DTE asserts the signal Request to Send (RTS), the DCE should respond with Data Terminal Ready (DTR).

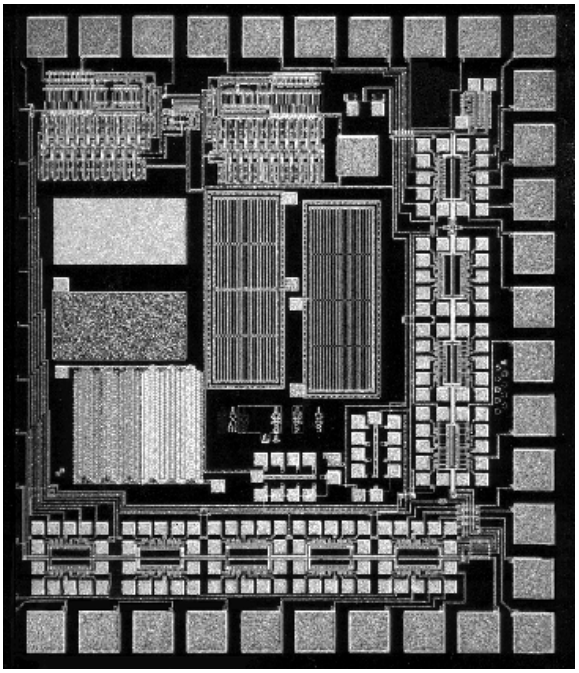
   TRUE or FALSE

9) In RS232, valid voltages are within the range +25 volts to -25 volts.

   TRUE or FALSE

10) A null modem cable allows two DTE devices to connect together using RS232.

    TRUE or FALSE



Lesson Five

Software Interface to EZ-MICRO TUTOR™

The objectives of this chapter are:

1. **To become familiar with the EZ-MICRO TUTOR™ microprocessor board.**
2. **To set up the PC/ DEVELOPMENT system.**
3. **To be introduced to the assembler source code syntax and assembly process.**
4. **Exercise the software development tools.**

The EZ-MICRO TUTOR™ microprocessor board is a powerful embedded microcomputer development system. Combining a personal computer (having at least one serial port) with the EZ-MICRO TUTOR™ allows the user to prototype hardware, develop, execute, and debug software using the Motorola MC68HC11E0 microcomputer chip (which is the heart of the EZ-MICRO TUTOR™).

The main components of the EZ-MICRO TUTOR™ package are:

- **EZ-MICRO TUTOR™ (based on the MC68HC11E0 CPU)**
- **EZ-MICRO Manager Software for the PC**
- **"Microprocessor Design Made Easy" Workbook**
- **Wire Wrap Board (optional)**

This chapter will involve exercising the procedures to:

1. **Connect the EZ-MICRO TUTOR™ Board to the PC.**
2. **Power up EZ-MICRO TUTOR™ Board.**
3. **Get familarized with the EZ-MICRO Manager software.**
4 **Edit and assemble sample code.**
5. **Download, execute and debug the code.**

It is not expected that an in depth understanding of these procedures will be achieved at this time, but that an appreciation of the overall process will be attained.

Procedure

1. Boot up the PC. Run EZ MICRO MANAGER software.

2. The EZ-MICRO Manager program allows you to accomplish several tasks. It allows the serial communication between the EZ-MICRO TUTOR™ microprocessor board and the PC. It also allows you to assemble 68HC11 assembly code, download the assembled code to EZ-MICRO TUTOR™ microprocessor board and execute/debug the code.

3. To provide power to the EZ-MICRO TUTOR™ microprocessor board, connect the 9 volt AC wall transformer that is provided into a 110 Volt AC source and connect the 9 Volt plug into the EZ-MICRO TUTOR™ power jack connector J6.

4. Connect one end of the serial cable that is provided to the COM port (or serial port) of the PC. Connect the other end of the cable to the serial port (9 pin male "DB" connector) of the EZ-MICRO TUTOR™ microprocessor board.

5. The following screen shows opening screen of EZ Micro Manager Software

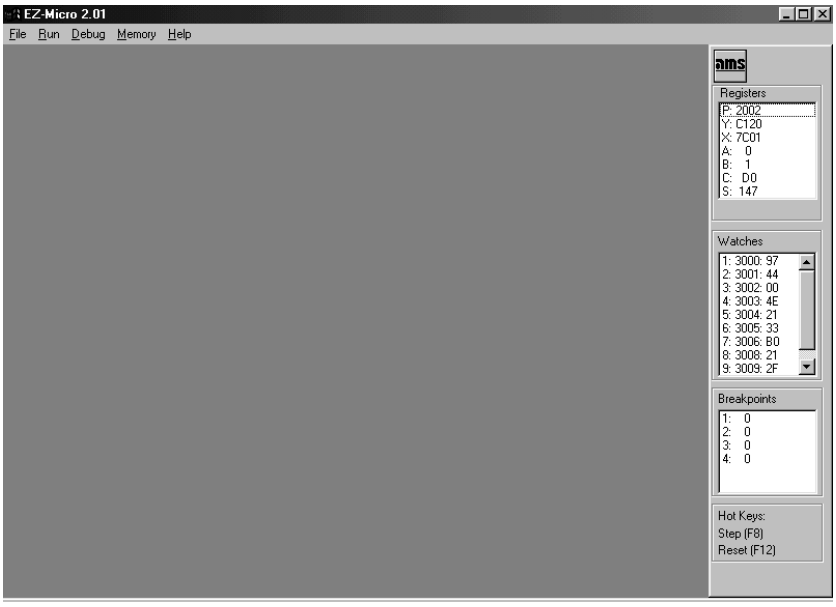


Fig 5-1 Opening Screen of the EZ-MICRO Manager Software

The EZ-MICRO Manager software establishes serial communication with EZ-MICRO TUTOR™ microprocessor board and sends initial data to the CPU. Figure 5-1 shows the opening screen of the EZ-MICRO Manager software. An overview of the available commands supported by the EZ-MICRO Manager program is explained in more detail in Lesson 4.

6. In order to create the assembly source code file "XXXLAB1.ASM" for this example procedure, you must select the Create Source command from the FILE menu and enter the name of the file that you wish to create.

Note: Replace **XXX** by your initials. The EZ-MICRO Manager editor can be used to create source code files that are compatible with the assembler. Enter the following source code:

```
            ORG       $2000
ARRAY       RMB       10
            ORG       $2120
            LDAB      #0          * for i=0 to 9
LOOP1       LDX       #ARRAY      * form array { i}
            ABX
            STAB      0,X   * array {i} = i
            INCB                  * next i
            CMPB      #10         * LAST i?
            BNE       LOOP1       * if not, do again
LOOP2       JMP       LOOP2       * wait here forever
```

After you have finished entering the above assembly language source code, save the file and exit the editor.

7. The next step is to assemble your source code using the EZ-MICRO Manager software. To do this, select Assemble File from the FILE menu and select the name of the assembly language source code file that you just created namely, XXXLAB1.ASM. Two new files will be created by the assembler:

   XXXLAB1.LST — A list file showing the assembled machine code combined with the orginal source code.

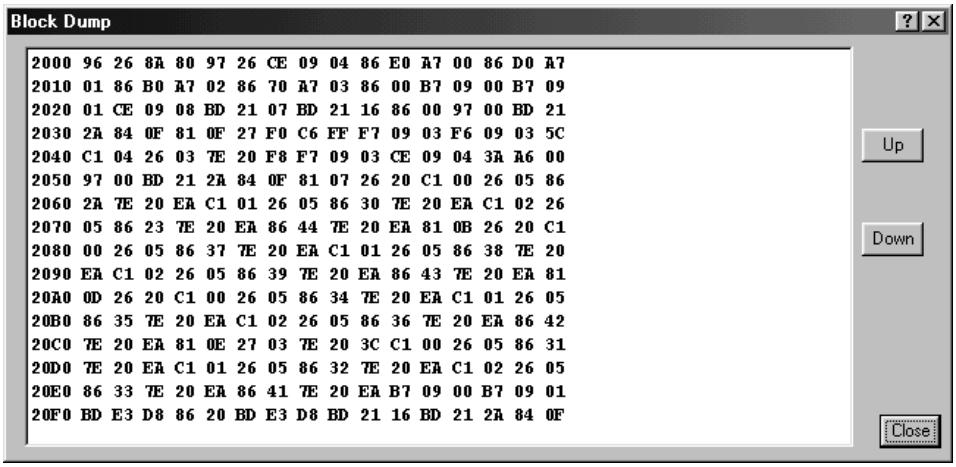
   XXXLAB1. S19 — A Motorola S-Record encoded file that can be downloaded to the EZ-MICRO TUTOR™ and placed into memory for execution.

   To view the assembled program listing, use the EZ-MICRO Manager editor (by selecting the Edit Source command from the FILE menu) to view the file XXXLAB1.LST.
   Look at the listing file to make sure that there are no assembler errors. Any errors encountered by the assembler are probably due to typographical errors. Correct any errors in the XXXLAB1.ASM file and re-assemble until there are no errors.

8. Download the XXXLAB1.S19 file to the EZ-MICRO TUTOR™ microprocessor board. To do this, select the Load S-Record command from the FILE menu and select XXXLAB1.S19.

9. Examine the program in memory by using the Memory Dump command selected from the MEMORY menu. Enter 2000 as the starting memory address to be dispalyed.

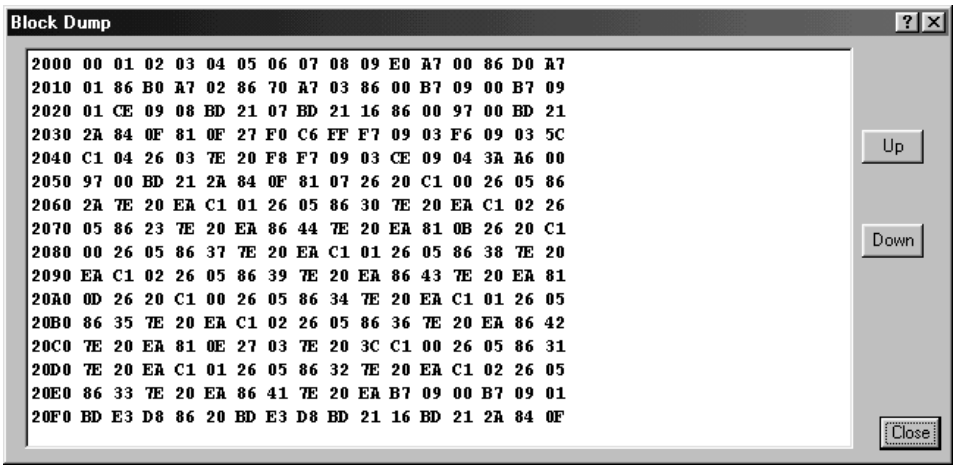
   The following information should be displayed:



   The bytes from $2120 to $212F are the machine code program that has been downloaded to the EZ-MICRO TUTOR™ microprocessor board. The ARRAY memory space is from $2000 to $2009 Note: It is possible that the first two lines may be other values. Do not be concerned at this time. To excute the program, select the GO command in the RUN menu, and enter 2120 as the address to begin program execution.

The EZ-MICRO TUTOR™ will execute the program and then continuously loop on the last instruction "LOOP2 JMP LOOP2" until the RESET button is pressed on the EZ-MICRO TUTOR™ microprocessor board. After pressing the RESET button, examine the memory by using the MEMORY menu and selecting the Memory Dump command. Enter 2100 as the starting memory address.

The display will look as follows :



Note how the bytes in the ARRAY memory space ($2000 to $2009) have been initialized to values 00 through 09.

10. Modify the ARRAY memory locations ($2000 to $2009) so that they contain FF again. To do this, use the Block Fill command in the MEMORY menu. After selecting the Block Fill command, enter 2000 as the *starting* memory address, enter 2009 as the *ending* memory address and enter FF for the *pattern*.

11. The program could be executed again, however aside from the fact that the memory locations change, the program flow is not very clear.The EZ-MICRO Manager offers several commands that can aid in following program flow.These commands work together to support tracing program flow, monitoring register and memory contents, and setting up breakpoints. The details of these commands are contained in Lesson 4. The following exercise will use the Register Modify and Trace commands to step through the program one instruction at a time.

    The Register Modify command can be used to display and modify the contents of the A, B, X, Y, C (condition code), S (stack pointer), and P (program counter) registers. Registers are memory locations that are closely bound to the core microprocessor. Registers have a variety of special functions that regular memory locations do not share.The registers that are used by this program are B, X, C, and P.

    The B register is a general purpose 8 bit wide register that is used to hold the variable that controls the loop count, index into the ARRAY and the data that is stored in the ARRAY.

    The X register is a 16 bit register that is used to address or point to, the array element that will be initialized.

The C register,often called the flag register, has bits that are set or reset depending on various conditions that result from program execution, For this example the Z (zero) flag is used (bit 2 in the C register). The Z flag is set to 1 (true) when the execution of certain instructions results in zero, otherwise it is reset to 0 (false).

Consider the effects of the "CMPB #10" instruction.This instruction compares the contents of the B register to the value 10. If they are equal, the Z flag is set. If not , the Z flag is reset. This flag setting is used in the following instruction "BNE LOOP1" to determine if the loop should be executed again or if it has finished.

The P register contains the address of the next instruction that will be fetched. In order to start tracing the program execution from the entry point at $2120, the P register must be initialized to this value. Modify the contents of the P register as follows: Pull down the DEBUG menu and select Modify Register. A dialog box will appear showing the current register contents.

Change the value of P (or Program Counter) to 2120. Press the OK button to accept this change to the register.

Once the P register is initialized, the Step (F8) command can be used . Select the Step command from the RUN menu.

The following text shows the result of steppiing the program flow through the first loop. The assembly listing on left hand side is what you will see in the window on left labled Program Monitor. The Register values on right hand side are the values you will see in Registers window on upper right hand corner.Note: It is helpful to follow the program execution with the program listing provided at the end of this chapter.

```
LDAB   #$00       P-2122 Y-BFFF X-FDFF A-32 B-00 C-94 S-0141
LDX    #$2100     P-2125 Y-BFFF X-2100 A-32 B-00 C-90 S-0141
ABX               P-2126 Y-BFFF X-2100 A-32 B-00 C-90 S-0141
STAB   $00,X      P-2128 Y-BFFF X-2100 A-32 B-00 C-94 S-0141
INCB              P-2129 Y-BFFF X-2100 A-32 B-01 C-90 S-0141
CMPB   #$0A       P-212B Y-BFFF X-2100 A-32 B-01 C-99 S-0141
BNE    $2122      P-2122 Y-BFFF X-2100 A-32 B-01 C-99 S-0141
LDX    #$2100     P-2125 Y-BFFF X-2100 A-32 B-01 C-91 S-0141
ABX               P-2126 Y-BFFF X-2101 A-32 B-01 C-91 S-0141
STAB   $00,X      P-2128 Y-BFFF X-2101 A-32 B-01 C-91 S-0141
INCB              P-2129 Y-BFFF X-2101 A-32 B-02 C-91 S-0141
CMPB   #$0A       P-212B Y-BFFF X-2101 A-32 B-02 C-99 S-0141
BNE    $2122      P-2122 Y-BFFF X-2101 A-32 B-02 C-99 S-0141
LDX    #$2100     P-2125 Y-BFFF X-2100 A-32 B-02 C-91 S-0141
```

Stepping through the program one instruction at a time is a powerful way to analyze the program flow. However , it can be noted that stepping a code section that loops many times could become very tedious. The EZ-MICRO Manager offers another command called Set

Breakpoint which allows the program to run until it reaches to the instruction where the breakpoint has been established. At that point, the EZ-MICRO Manager monitor program takes over and allows the memory, register and trace commands to be excuted. If there is time left in the lab feel free to experiment with the Set Breakpoint command that is described in Lesson 4 in this courseware workbook.

Laboratory Report

The lab report for this lab will invole writing a short (1 to 4 paragraphs) description for EACH of the major components that is contained in your EZ-MICRO TUTOR™ package .The primary thrust of these summaries is to identify the type of information and capabilities contained in each component. Becoming familiar with the package contents will make it easier to find infomation as it is needed while working with the EZ-MICRO TUTOR™. It is NOT expected that all of the documentation enclosed with the EZ-MICRO TUTOR™ will be memorized, but rather that it should be familiar and quickly accessed to derive necessary information.

Original source listing of XXXLAB1.ASM

```
*****************************************
* Laboratory 1 Source Code              *
* D/ August 4, 2000                     *
* REV 00                                *
* This program is used to demonstrate   *
* the process of editing, assembling    *
* downloading, executing and debugging  *
* code on the EZ-MICRO TUTOR™ Board.    *
*****************************************
*****************************************
* RAM variable storage assignment:      *
* User RAM will start at $2100          *
*                                       *
* Notes:                                *
* The ORG (origin) statement tells the  *
* assembler where to start placing the  *
* following code, in this case, ARRAY   *
* will start at $2100 and continue to   *
* $2109.                                *
*                                       *
* The RMB (reserve memory bytes) statement *
* tells the assembler to reserve 10 bytes  *
* for the ARRAY array space             *
*****************************************
        ORG   $2100
ARRAY   RMB   10
*****************************************
* Pseudo code for this program:         *
*                                       *
* The variable i is a register variable *
*                                       *
* Program: "Initialize ARRAY"           *
* int i, array {0..9}                   *
*                                       *
* begin                                 *
*     for i=0 to 9                      *
*     array{i}=i                        *
* end                                   *
*****************************************
        ORG   $2120
        LDAB  #0
LOOP1   LDX   #ARRAY
        ABX
        STAB  0,X
        INCB
        CMPB  #10
        BNE   LOOP1
LOOP2   JMP   LOOP2
        END
```

Listing produced by the EZ-MICRO Manager assembler (XXXLAB1.LST):

```
0001                    ******************************************
0002                    * Laboratory 1 Source Code                *
0003                    * D/ August 4, 2000                       *
0004                    * REV 00                                  *
0005                    *                                         *
0006                    * This program is used to demonstrate     *
0007                    * the process of editing, assembling      *
0008                    * downloading, executing and debugging    *
0009                    * code on the EZ-MICRO TUTOR™ Board.      *
0010                    ******************************************
0011                    *
0012                    ******************************************
0013                    * RAM variable storage assignment:        *
0014                    * User RAM will start at $2100            *
0015                    *                                         *
0016                    * Notes:                                  *
0017                    * The ORG (origin) statement tells the    *
0018                    * assembler where to start placing the    *
0019                    * following code, in this case, ARRAY     *
0020                    * will start at $2100 and continue to     *
0021                    * $2109.                                  *
0022                    *                                         *
0023                    * The RMB (reserve memory bytes) statement *
0024                    * tells the assembler to reserve 10 bytes  *
0025                    * for the ARRAY array space               *
0026                    ******************************************
0027 2100                       ORG   $2100
0028 2100              ARRAY    RMB   10
0029                    *
0030                    ******************************************
0031                    * Pseudo code for this program:           *
0032                    *                                         *
0033                    * The variable i is a register variable   *
0034                    *                                         *
0035                    * Program: "Initialize ARRAY"             *
0036                    * int i, array {0..9}                     *
0037                    *                                         *
0038                    * begin                                   *
0039                    *     for i=0 to 9                        *
0040                    *     array{i}=i                          *
0041                    * end                                     *
0042                    ******************************************
0043 2120                       ORG   $2120
0044 2120 c6 00                 LDAB  #0
0045 2122 ce 21 00     LOOP1    LDX   #ARRAY
0046 2125 3a                    ABX
0047 2126 e7 00                 STAB  0,X
0048 2128 5c                    INCB
0049 2129 c1 0a                 CMPB  #10
0050 212b 26 f5                 BNE   LOOP1
0051 212d 7e 21 2d     LOOP2    JMP   LOOP2
0052 2130                       END
```

Lesson 5 Questions

1) What tasks does the EZ-MICRO Manager software allow you to accomplish ?

2) How do you start EZ-MICRO Manager software ?

3) How does someone edit a source file ?

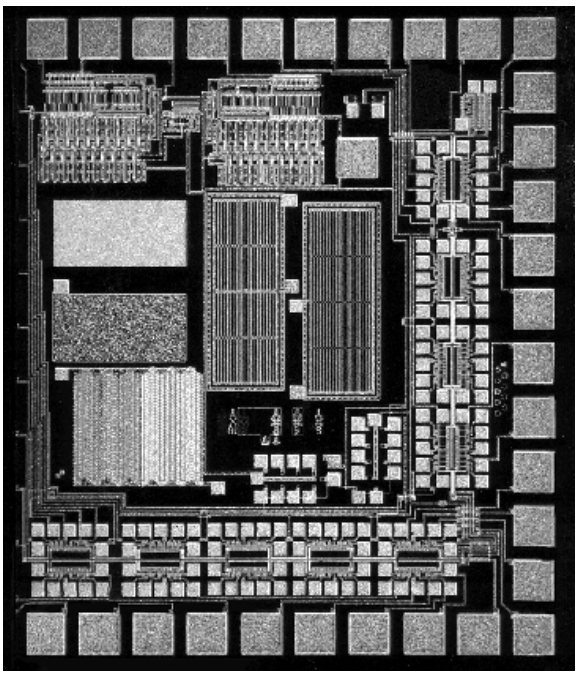
4) Why is it a good idea to generate a list file when assembling the source code ?

5) What is the purpose of a Motorola S-Record file ?

6) How can someone examine the program in memory that has been downloaded to the EZ-MICRO TUTOR™ board ?

7) What is a common method used to terminate a program ?

8) What is the P register used for ?



Lesson Six

Introduction to Programming

You have probably been exposed to computer programming in a high level language. In this Lesson we will provide at starting point for programming at the assembly code level. Computer programs are generally written in symbolic languages, rather than in binary or hex notation. The same is even true of assembly language programming. In Lesson Three we introduced the set of mnemonics that exist for the 'HC11 instruction set. Now we will use this instruction set to build basic programming structures such as tables, lists, loops and subroutines.

The essential difference between all the assembly, or low level, languages and a high level language is the relationship between all the source code instructions and the machine code. An instruction written in an assembly code will be assembled in a one-for-one translation into the machine code. This is not true for a high level language where each statement will probably result in many instructions in machine code.

The main advantage of the compiler that you use to compile your Basic or C is that the source program statements actually look like English and can be read easily. For example, one statement in a high level language can solve a complete problem. It can define a number of alternative results dependent upon any calculable conditions, whereas an instruction in an assembly language could define only one stage in the arithmetic operation or one simple conditional branch.

As a simple example of what is possible with a high level language, a typical statement is shown below in a pseudo-code format:

```
IF (A-B) > 0 THEN C=D
ELSE C=2*A
```

To generate its equivalent in a low level language would be more complex and would require a routine of several program statements. One equivalent routine implemented in 'HC11 assembly language is shown below.

	instruction		explanation of instruction
	CBA		Compare accumulators
	BLE	**ALTB**	Branch if (A-B) =< 0
AGTB	**LDA A**	**MEMD**	Load D from memory into A
	BRA	**NEXT**	Skip to next instruction
ALTB	**ASLA**		multiply by 2
NEXT	**STAA**	**MEMC**	Store result into memory

Through the use of examples, in this Lesson we will provide an introduction to assembly language programming. The techniques described and developed within this Lesson will be used in the Laboratories that accompany each of the remaining Lessons in the EZ-Micro course. The major ideas to be discussed include look-up tables, lists, loops, and subroutines.

Addressing Modes

The most used instruction is probably LDA, which puts a byte into the accumulator. We will use this instruction to explain some addressing modes and to show how to read your Motorola reference materials. The instruction LDA has two types, LDAA and LDAB referring to the A and B accumulators, each being one byte registers. We will mostly use the A register.

LDAA has five assembly language forms(modes).

```
1) IMM      LDAA   #$3B      ;Immediate Mode
2) DIR      LDAA   $49       ;Direct Mode
3) EXT      LDAA   $23B7     ;Extended Mode
4) IND,X    LDAA   $4, X     ;Index X Mode
5) IND,Y    LDAA 0, Y        ;Index Y Mode
```

1) The instruction **LDAA #$3B** means "put the binary equivalent of the hex number 3B into accumulator A". We said binary equivalent because all numbers are reduced to binary when they end up inside any kind or RAM. We could write the instruction as LDAA #59 and get the same result because 59 decimal equals 3B hex, and they both equal 00111011 binary. The assembler will know when it sees the $ sign to read hex and if there is no $ sign it will read decimal. It is often helpful to write the instruction directly in binary, LDAA #%00111011, because this lets you see which bits are high and low. These might be output pins connected to switches.

The key element in this instruction is the # sign. This indicates **immediate mode** and lets the assembler know that you want the **number** 3Bhex put into A and not the number-in-the-address 3Bhex as in the next example.

2) **LDAA $49** means "copy the byte(number) which is in address 49hex of memory into the accumulator A". This is Direct Mode and only memory addresses from O through FFhex (255) are used.

3) The most general form is **LDAA $23B7** which will copy a byte from any address in the 68HC11 memory space (from O thorough FFFFH) into the accumulator. This is extended mode.

4) The instruction **LDAA $4**, X says "read the address in X, add 4 to it to get the address X+4, and then copy the byte in X+4 into the accumulator". Of course we must have an instruction somewhere before in the program which places an address into X (X holds a 2 byte address so this instruction can also work throughout the entire 64K memory area.

For example:

```
            LDX #$1000      ;put number 1000H into X
LDAA $22, X     ;copy byte at address 1022H into A
```

Note that we do not know what is in A now because we do not know what number is in address 1022H.
Lets resolve this problem as follows.

```
LDAB #$23       ;put number 23H into B
STAB $1022      ;copy B into memory at address 1022H
LDX #$1000      ;put number 1000H into X
LDAA $22, X     ;copy 23H from address 1022H into A
```

A now contains the number 23H. -We do not move numbers directly into or out of memory. We always move by way of the accumulator. Accumulator B was used above just to show it is equivalent to A. We could have used only A.

5) **Index Y** is used in the same way as Index X. The assembly language code will be identical. There is a slight difference in how the processor creates the same result. We discuss that now.

First, for practice, look in your references and find the Opcodes for LDAA. In the IMM mode it shows the code as 86 ii. That means the machine language bytes for this instruction are 86H followed by iiH, where ii is the single byte number you placed in the instruction. For example in **LDAA #S3A** the instruction LDAA # is machine code 86H, and ii is replaced by 3AH. That makes a total of two bytes for the instruction (see the Byte and Cycle columns next to the Opcode column). When this instruction ends up in your system memory it will take two consecutive bytes, the first will hold 86H and the second will hold 3AH.

Now look in your reference and find the Opcode for **LDAA $0, Y**. It is 18H+A6H, and the operand ff is $0 in this example. This is a three byte instruction as opposed to LDAA $0, X which is a two byte instruction. Also notice the Index Y LDAA instruction requires five cycles whereas the Index X LDAA requires only four cycles. With a 4MHz crystal each cycle is 1 microsecond. These two-byte Opcodes appear first in the 68HC11. The older Motorola 6800 microcontroller, which uses the same one byte machine codes as the 68HC11, has only one byte instructions.

There are two more addressing modes: Inherent (INH), and Relative (REL).

Inherent is the simplest mode. Inherent mode instructions have nc Operands. For example, DEX is an instruction which reduces the value of the X register by one. If X holds the value 4567H and you give the instruction DEX the new value in X will be 4566H. DEX is a one byte instruction. However the equivalent instruction for register Y, DEY, is a two byte instruction.

Relative addressing refers to the branch instructions. It means you can jump back 128 bytes of code or forward 127 bytes of code relative to your present position. For example, the instruction BRA 100 means jump forward 100(decimal) bytes of code and continue from there. Here is where an assembler does some work for you. The instruction would normally be given as BRA LABEL1 where LABEL1 is a label placed later (but not too much later or you will get an error message, or worse yet not get one) in your code. The assembler will determine how many bytes of code from your present position to LABEL1 and replace LABEL1 in the instruction BRA LABEL1 with the correct number. Otherwise you have to get out your Programming Guide and start counting bytes. By the way, LABEL1 is just a name we made up; it has no special meaning to the 68HC11 or the assembler.

Programming Examples

Now that you understand how to use the reference materials we can go -ahead and give examples without explaining every detail about each instruction. Also when you see the list file created by the assembler you will know how to interpret the hex bytes (Opcodes and Operands) given in the left columns.

Reading and writing the Control Registers

The Control Registers control all the system hardware. Upon reset these registers start at $1000 in memory. We read and change these registers constantly in an embedded system program. In the EZ MICRO Board the Control Registers are relocated starting at 0000H in memory, overlapping external RAM. In a little bit we will see why it is useful to put the number $0000 into the X register and use X Index addressing in the EZ-MICRO BOARD. But for now we use a more direct way of addressing registers.

Using Byte Methods and Extended Addressing

A simple way to turn on the four LED's connected to output port. Since these LED's are connected through 74LS373 Latch and low on output of this latch turns LED on.

```
BUFALO  EQU  $e000  *start of BUFFALO
OUTPUT  EQU  $e3c2  *BUFFALO subroutine for OUTPUTing a character in A
LEDOUT   EQU  $4400
*
****************************
* Define variables.        *
****************************
*
        ORG  $2000


MAIN    LDAA   #$00
        STAA  LEDOUT
        SWI
```

The above program will turn on the 4 led's, but in a rather crude way. It follows that we did not need to send zeros to all of the port pins, four pins would do. In fact by changing all the bits in these registers we may upset some othe part of the system which uses these pins. We need some way to change only those pins of interest while not affecting the others.

Let's try again, this time changing only pins 2, 3, 4, and 5 of port D We use "masks" and logic instructions.

```
;Use AND to set bits to 0's
               LDAA  #%11110000;mask bits 0, 1, 2, and 3 cleared in S
               ANDA  LEDOUT   ; result is in A
               STAA  LEDOUT   ;only needed pins set to zero, led,s or
               END
```

Notice that the instructions ANDA can use extended addressing so the full addresses of LEDOUT can be used in the instructions.

Using Bit Methods with Index Addressing

There are instructions to just set or clear one bit of a byte. These instructions will work on any byte in the system memory. There is just one problem; the instructions will not work in extended mode. There are many instructions that will not work in extended mode, so what we learn here will be very useful. Now we can not just put the absolute address of a control register after an instruction. Instead we will use index mode.

```
;First the program setup.

PORTD    EQU  $0008
DDRD     EQU  $0009

         ORG  $2000
;Setup Register DDRD
         LDX      #$0000
         BSET     DDRD,X %11111111
;
         BCLR     PORTD, X %11110000
         END
```

The BSET and BCLR instructions will meet many of your needs in manipulating the I/O pins of the controller.

Remember the reset location of the Control Registers is starting at 1000H but the EZ-MICRO-11 moves these registers to start at 0000H. DDRD is at 1009H after reset. We then move it to 0009H. Therefore we say DDRD EQU $09 and set X=$0000 so that when we use indexed addressing the address X+DDRD = 0009H.

JUMPs and BRANCHes

JUMPs and BRANCHes change the flow of a program by changing the number in the program counter (PC). This causes the next instruction to be executed at the new address in the PC.

The **JUMP** instruction place the address we want the program to jump to in the PC. The JUMP instruction can be executed in one of two modes: extended or indexed. It can jump anywhere in memory.

```
        JMP Label5          ;jump to execute the instruction at Label5
        .
        .
        .
        Label5 LDX #$1000   ;any instruction codes in memory
```

BRANCH instructions differ from JUMP instructions in that they do not supply a direct address. The address to be jumped to (branched to) is always relative to the current address in the program counter. The addresses must be within 128 bytes from the current PC.

Mostly, branch instructions use the condition codes to decide whether to branch. Condition codes are states of the bits (N,Z,V,C) in the Condition Code Register (CCR). Some branch instruction do not use the condition codes at all. For example, BRA (branch always), BRN (branch never).

```
        ORG    $2000
        BRA    MAIN            ;branch to where MAIN located at 2050H
        ORG    $2050
MAIN    BRA    MAIN            ;program does nothing, loops forever
        END
```

JMP will always work, so why not use JMP instead of BRA? JMP takes up one extra byte of program memory (see your Opcodes table).

BEQ (Branch if equal) tests the state of the Z bit in the CCR and causes a branch if Z is set.

```
;program will compare A-accumulator with ASCII 'A'
;if A-register contains ASCII 'A', program will put a 1 in B-register
;else, a zero will be place in the B-register

        CMPA #'A'           ;compare the content of A-accumulator with
                            ;ASCII A (=41Hex). Z flag is set if equal
        BEQ    ASCII_A      ;branch if Z is set, which is 'A' equals
                            ;A-register
        CLRB                ;clear B-register, inherent mode
        BRA    STOP
ASCII_A                     ;this is a label-only line
        LDAB   #1           ;above label applies to this instruction
STOP    BRA    STOP         ;loop forever, or until interrupt
```

Check CMP (two forms CMPA and CMPB) in your 68HCll Reference Manual. Look where it says **"Operation (ACCX)-(M)"**. That means it will subtract 41H (in this example) from the value in A and inspect the resulting number; A does not change. The only action this instruction takes is to set some CCR bits for use by later instructions. In this example we are interested in the z-bit because that is the only bit which the BEQ instruction looks at.

BRCLR (Branch if bits clear)

Test any bit of any byte. If bit is zero branch to label

```
        TFLG1  EQU    $23      ;MAIN TIMER INTERRUPT FLAG REGISTER 1

               LDX    #$0000

               BRCLR TFLG1,X %00000010 LABEL6
```

```
;The above BRCLR (BRANCH IF BIT(S) CLEAR) will branch to LABEL6 if
;bit 1 of the memory location TFLG1 (memory 6023H) is zero. LABEL6
;will be placed in your program in front of the instruction you want
;to branch to. For example, you may read an input port register and
branch sif the input signal on a particular pin is low.
```

Using Subroutines

When we write a program, we often find that we are repeating the same sequence of instructions in several different parts of the program. These sequences of instructions must be used at different places in the program. Without the subroutines, we would be forced to repeat these sequences of instructions in the source code, and the machine instructions generated by the assembler would be repeated in memory creating a larger program, plus much extra typing. Fortunately, we have the ability to define subroutines and to use the instructions that support them.

```
;MAIN PROGRAM
;Demonstrate subroutine
;The program calls the delay subroutine to delay some machine cycles
;your program lines, and then a jump to sub
        .
        .
        .
        JSR DELAY    ;jump to subroutine
;SUBROUTINE DELAY, subs usually put after the main program

DELAY
        PSHX                ;save content of IX register on stack
        LDX    #FFFF        ;place FFFFh into IX
L1:     DEX                 ;decrement IX ~
        BNE    L1           ;loop back and decrement until IX = 0000
        PULX                ;restore IX register from stack
        RTS                 ;return from subroutine
        END                 ;end of assembly
```

There may have been a value in the X register which we need when we return to the main program. So we push original value X on the stack and save it until the end of the subroutine and then pop (pul) it off the stack just before returning. It is good programming practice to make sure all registers which are used in the sub are restored this way to their original values upon returning from the sub.

Using table lookup

```
;Demonstrate table lookup
;use IX for base address of table
;use B-register for indexing elements in the table. First element
;will have the index of zero.

        LDX #TABLE   ;IX contains address of first element in TABLE
        LDAB   #$3   ;set B-register to get the fourth element ($44)
        ABX          ;Add Accumulator B to Index Register X
```

```
                ;IX now-contains 3003Hex
        LDAA 0,X    ;copy the content at address 3003H into
                    ;A-register

        ......      ;program goes on to somehow use the value in A
        ORG    $3000
TABLE   FCB    $41,$42,$43,$44,$45,$46
;table contains 5 elements indexed by 0,1,2,3,4
```

The assembler equates the address of the first data byte (41H) with t label TABLE. The fourth data byte is 44H and it is 3 data bytes after t first data byte 41H. That is why we put LDAB #$3 to get to the four byte.

Software Interrupt

When you run test programs with the debugger you can insert S instructions (inherent mode) in the program at points of interest. T debugger will stop your program at that point and return to the comput display screen with the current register values. You can also che memory data with the M XXXX command. To continue with the program ju use the C command. This way you can see what is going on with CCR bit the accumulator etc., or your program variables. A little practice wi this technique is worth thousands of words.

Here are some miscellaneous instructions and programming steps you wi find useful.

Register Transfer Instruction

```
        LDAA #$50   ;load A-accumulator with 50Hex
        TAB         ;transfer 50Hex to Accumulator B
```

Clear Instruction

```
        CLRA        ;clear Accumulator A
        CLR $5000   ;clear memory byte at 5000Hex
```

Push, Pop Instruction

```
        PSHX                    ;push index register X onto the STACK
        PSHY
        PSHB
```

Addition Instructions

```
        LDAA   #$50   ;put 50hex into A-accumulator
        STAA   $200   ;store 50Hex at the address of 200Hex
        LDAA   #$6
        ADDA   $200   ;Add 50hex with 6Hex (A now contains 56Hex)
        LDAB   #$2    ;place 2Hex in Accumulator B
        ABA           ;add A and B then put the result in A
                      ;A-register now has 58Hex
```

ABA Instruction

```
        LDAA   #$FF   ;A = FF
        LDAB   #$6    ;B = 6
        ABA           ;A+B-,A , add A and B
                      ;result A = 05H, with carry flag set
                      ;so the result is 105H

;The events above cause a carry flag to be set.
```

Double ADD Instruction

```
        LDD    #$CD8A ;load D-accumulator with CD8Ahex
                      ;(A=CD, B=8A)
        ADDD   #$CllC ;ADD D-accumulator to CllChex
                      ;CD8A+CllC= 8EA6, plus a carry flag set
                      ;A now contains 8E and B contains A6

;total is more than FFFFH so the carry flag was set. total ls 18EA6H
```

NEGATE Instruction

The NEGATE Instruction will perfor,m a 2s complement negation of a number by subtracting it from 00.

```
        LDAA   #%00000101     ;place 5Hex in A-register
        NEGA                  ;A = 11111011
```

Multiplication Instruction

Multiply A x B -> D

```
        LDAA   #$FE
        LDAB   #$FD
        MUL           ;result is FB06hex (A = FB, B = 06)
```

The MUL instruction is unsigned.

AND instruction

```
        LDAA   #%00111001
        ANDA   #%00001111
                              ;after ANDA instruction A =%00001001
```

Look-up Tables

Look-up tables can provide the embedded systems designer the capability to generate or translate data that cannot be calculated due to timing constraints.
Microprocessors are especially well suited to keep track of and access tables and lists.

In Lesson Three, the 2-byte index register was introduced. In the indexed addressing mode, the contents of the index register are used to compute an indexed address. The indexed addressing mode is best illustrated by example. Suppose the index register contains $1000.
In the instruction

```
LDAA    $05,X
```

the X part tells the assembler that this is the indexed addressing mode. Executing the instruction, the 'HC11 adds the offset (5 in this example) to the contents of the index register ($1000 in this example) and the sum is the address of the data which is loaded into accumulator A.

There are instructions that directly affect the index register. Refer to Lesson Three where we saw that the instruction STX is similar to STA; the contents of the index register is stored in two consecutive memory locations. Two other instructions that affect the index register are INX, which will increment the index register and DEX, which will decrement the index register.

In a look-up table two equivalent sets of values are stored side by side so that any value can be retrieved easily. A temperature conversion chart is a familiar example of a look-up table. To look up the equivalent temperature in degrees C for a temperature given in degrees F, we could use the table shown in Figure 6.1 below. The advantage of using this table is that we avoid multiplication by a fraction. The disadvantage of using this table that we cannot lookup the equivalent temperature in degrees C for 20.5 degrees F.

deg F	deg C
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	

Figure 6.1 Lookup Table for Temperature Conversion

There are many uses of tables in computing. For example, a computer can perform the square root operation in a number of ways. One way is to store a table of all the possible values of square roots and look up the answer for a given number. Depending on factors such as length of the calculation and the sized of the table, the programmer will have to choose between performing the required calculation and using a look-up table.

Now, as another example, we will use indexed addressing in the 'HC11 to retrieve the square of a number by retrieving the value from a look-up table. Refer to Figure 6.2, which shows the table is squares stored in data memory beginning at address $1000.

Address	Number	Square
$1000	$00	$00
$1001	$01	$01
$1002	$02	$04
$1003	$03	$09
$1004	$04	$10
$1005	$05	$19

Figure 6.2 Lookup Table in Memory for Computing Squares

Consider the following code segment:

```
LDX     #$1000
LDAA    $2,X
STAA    SQUARE
```

The index register has been loaded to point at the base of the table. The address pointed to by the LDA A instruction is computed by adding the offset ($02) to the contents of the index register ($1000) which $1002. The contents of the location $1002 is $04 which is the squared value that we are after.

Lists

We can think of a list as a one-dimensional table or an array. The text editor in the EZ-Micro Manager software works with a list of ASCII characters. There are many ways to manipulate this list. We may want to move it to another location, search through it for specific items, or delete items.

First, consider a simple example. Suppose we want to create a list of all zeros starting with location $1000 and ending with location $101F. Your software could use the CLR instruction to clear each individual memory location between and including these addresses. However, the resulting program would not make very efficient use of program memory.

Alternatively, we would like to make a loop that uses just one CLR instruction and steps through the list clearing each location sequentially. As we did with tables, we will use the indexed addressing mode.

Loops

Counting loops

In a counting loop we go through the loop a definite number of times. We define a loop counter as a device used to keep track of the number of times through a loop. The loop counter can be a register or a memory location; the only requirement is that we must be able to load, increment, and decrement the loop counter.

Consider an example where we want to go through a loop 5 times. First of all, we can either count up or count down. If we start at 0 and count up, we want the 'HC11 to execute the loop for values of the counter equal to 0, 1, 2, 3, and 4.
Our software must exit the loop after the loop hits 5 but before executing the loop for the sixth time.

If we start at 5 and count down, we want the 'HC11 to execute the loop for values equal to 5, 4, 3, 2, and 1. Our software must exit the loop after the loop counter hits 0 but before executing the loop for the sixth time.

In general, because the 'HC11 can automatically check the Z bit, it is more efficient to count down. In order for our software to detect when the loop counter hits a non-zero number (like 5) we must use the compare instruction. The code segment below shows an implementation of a counter that decrements from 5 to 0. The instructions inside the loop execute exactly 5 times.

```
            LDAA    #5
COUNT       DEC      A
             :
            {instructions inside the loop}
             :
            BNE      COUNT
```

Timing Loops

Counting is a good way to create a programmable delay in your software. The 'HC11E0 in the EZ-Micro Tuotr architecture is running at 1 MHz and yet our application may be required to blink a light once every 5 seconds. Alternatively, our software may need to time a 5 millisecond delay in a communications application.

In any given microprocessor each instruction in each addressing mode takes a precise and deterministic amount of time to execute. This means that we can calculate how long it takes to make a certain number of passes through a loop. By looking at each instruction's information sheet, we can identify the "execution time", a value given in terms of cycles. The cycle time of the 'HC11 on-board the EZ-Micro Tutor Board running from an 8 MHz crystal is 500 ns. The LDAA instruction requires 4 cycles to execute in the immediate addressing mode. This means that it takes 2 usecs for LDAA #5 to execute on the EZ-Micro Tutor system. Take note of Table 6.2 below, where the execution times of a few common looping instructions are provided.

```
LDAA      (imm)   2 cycles
DECA              2 cycles
BNE               4 cycles
NOP               2 cycles
LDX       (imm)   3 cycles
DEX               3 cycles
```

Table 6.2 Instruction Cycle Times

Now let's revisit our code segment that executes a loop five times. This time we will modify the code segment such that there are only two NOP instructions inside the loop. The NOP instruction is typically used to provide additional delay.

```
            LDA     #5
COUNT       DEC     A
            NOP
            NOP
            BNE     COUNT
```

The initialization of the loop only occurs once. The LDA instruction requires 1 usec. The time for 5 passes through the interactive part of the loop can be computed as 5 passes x 5 usec/pass = 25 usec. So the total time required for one execution of the loop is 26 usecs.

The largest loop counter that we can create with a single byte of memory or a single byte register is 256. However if we now use a 16-bit register as a loop counter we can count up to (or down from) 65,535. For a loop that requires 10 cycles per iteration, the total time required for one execution of the loop is now 327,675 usec plus 1 usec to initialize the loop. Notice that as the time required for all the iterations grows large the time required to initialize the loop becomes less and less significant.

The index register is a 16-bit register that is very well-suited for a loop counter.
The instructions LDX, INX and DEX make it easy. For example, consider the following code segment that creates a delay of a little less than 0.3 seconds.

```
            LDX     #65,535             * 3 cycles
COUNT       DEX
                                        * 3 cycles
            NOP                         * 2 cycles
            BNE     COUNT               * 4 cycles
```

Nested timing loops

To produce delays that are greater than 1 second in duration, we just execute the 1 second delay routine more than once. We can control the number of times that the delay routine is implemented by using a counting loop around the delay loop. This programming concept is called nested loops.

Nesting our 0.3 second loop inside a counting loop of 5 will provide us with a delay of about 1.5 seconds. The short segment of assembly code shown below demonstrates one way in which this nested delay can be implemented.

```
                LDA     #5
OUTER_LOOP      DECA
                LDX     #65,535
COUNT           DEX
                BNE     COUNT
                BNE     OUTER_LOOP
```

Subroutines

Depending on how much experience you've had with higher level languages, you may have already been introduced to subroutines and functions. Any routine that can be used over and over again is a good candidate for a subroutine. Take for example the multi-second delay that we just developed in the last section. Suppose we name that code segment DELAY1. Then as part of a much larger program we could just branch to the routine DELAY1.

Of course, there's one problem with this programming technique. After the delay routine finishes execution, what code will the microprocessor execute next? What we would like to do is to return back to the main program and continue executing that code beginning at the instruction immediately following our branch instruction. Sounds like what we need is some kind of a bookmark in our main program to keep track of where we should return to.

As we learned in Lesson Two, practical programs frequently need to save values and addresses generated by an operation for use later in the program. A stack is a means of temporarily storing return addresses or data when the CPU is executing a routine. The stack may be located anywhere in the 64K address space of the 'HC11 and may be any size up to the amount of memory available in the system. The 16-bit register called the stack pointer controls the operation of the stack.

The Program Counter (PC) is a 2-byte register that holds the address of the next location whose contents are to be fetched. Obviously this is an important pointer that we don't want to loose track of.

Now we see that the bookmark that we spoke of earlier is implemented by pushing the program counter onto the stack. This operation frees the program counter for subroutine use. After the microprocessor finishes execution of the subroutine, it pops the old program counter contents from the stack and resumes executing the main program.

When we transfer control from the main program to the subroutine we are calling the subroutine. In the 'HC11 there are two instructions for calling a subroutine — BSR and JSR. Both of these instructions automatically cause the microprocessor to push the current program counter onto the stack before executing the first instruction of the subroutine.

When we transfer control back to the main program from the subroutine we are returning from the subroutine. In the 'HC11 the RTS instruction is used for returning from a subroutine. This instruction automatically causes the microprocessor to pop the program counter from the top of the stack and continue from there.

One additional note — the program counter is a 16-bit register. The stack is 8-bits wide by definition because it resides in memory. For this reason, during a subroutine call, two bytes representing the program counter are pushed onto the stack. First PCL is pushed followed by PCH. Also recall that the stack grows from high memory towards low memory. This means that if the stack pointer contents were $001F and the program was pushed onto the stack then PCL would be in address $001F and PCH would be in $001E. Of course, the program counter is popped off the top of the stack in reverse order. First PCH is popped followed by PCL.

Let's look at one example in detail. Suppose it is necessary to convert a binary number into its ASCII equivalent at a number of different places within a program. This could be accomplished by making multiple copies of the conversion routine and placing them at the required points within the program. The resulting program may be difficult to maintain and requires additional program memory. For this reason, we decide to create a subroutine. The binary number must somehow be transferred to the subroutine for conversion. Similarly the subroutine must somehow transfer the ASCII character back to the main program.

Passing parameters

Subroutines like the DELAY routine do not require that any data be passed into it. By the same reasoning, the DELAY routine does not return any data either. However, usually, we want a subroutine to operate on variable data that is supplied to it by the main program, like the ASCII conversion example. Giving data to a subroutine and getting data from a subroutine is called passing parameters.

One way of passing parameters is through memory locations. Memory locations are defined by the main program and used by the subroutine. One advantage of this scheme is that in general there will be a lot of available memory for passing parameters. One drawback with this scheme is that it is not portable. If I try to call your subroutine with my main program I may not have the memory locations available that you are expecting to be free. Using the ASCII conversion example, this method is demonstrated in the following code segment.

```
            ORG     $2000
MEM_XFER    RMB     1

            ORG     $3000
MAIN        LDX     #$0020
            TXS                     * initialize stack ptr

            LDAB    BYT2CONV        * load ACC B with byte to convert
            STAB    MEM_XFER        * load data memory
            JSR     BN2ASC          * jump to subroutine
            LDAB    MEM_XFER        * retrieve ASCII character
             :                      * continue with main code
             :
```

```
* Subroutine BIN2ASC
* Pulls byte from stack and converts to ASCII character '0' - '9'
* Sets msbit if out of range
BN2ASC    LDAB   MEM_XFER       * load byte from memory into ACCB
          CMPB   #0             * check for out of range
          BMI    B2AERR
          CMPB   #10
          BGE    B2AERR
          BRA    B2A
B2AERR    BCLR
          BSET   #$80
B2A       ORAB   #$30
          STAB   MEM_XFER     * put the ASCII character in memory
          RTS
```

Another way of passing parameters is through an accumulator. This scheme has the advantage of portability. We can save the value that the main program had in the accumulator by pushing that value onto the stack before making the subroutine call. Later the main program can restore the original value to the accumulator by popping that value off the top of the stack. Again using the ASCII conversion example, this method is demonstrated below.

```
          ORG    $2000
MAIN      LDX    #$0020
          TXS                  * initialize stack ptr
          LDAB   BYT2CONV      * load ACC B with byte to convert
          JSR    BN2ASC        * jump to subroutine
           :                   * ASCII character in ACC B
           :                   * continue with main code
           :


* Subroutine BIN2ASC
* Pulls byte from stack and converts to ASCII character '0' - '9'
* Sets msbit if out of range
BN2ASC    CMPB   #0             * check for out of range
          BMI    B2AERR
          CMPB   #10
          BGE    B2AERR
          BRA    B2A
B2AERR    BCLR
          BSET   #$80
B2A       ORAB   #$30
          RTS
```

A third, and perhaps more powerful, method involves storing the parameter on the stack prior to making the subroutine call. This method is commonly used in high level language compilers since it is a flexible scheme that produces portable code. Consider the following code segment of our ASCII conversion example.

```
          ORG    $1000
          ORG    $1000
MAIN      LDX    #$0020
          TXS                    * initialize stack ptr
          LDAB   BYT2CONV        * load ACC B with byte to convert
          PSHB                   * push it onto the stack
          JSR    BN2ASC          * jump to subroutine
          PULB                   * put ASCII character in ACC B
           :                     * continue with main code
           :

* Subroutine BIN2ASC
* Pulls byte from stack and converts to ASCII character '0' - '9'
* Sets msbit if out of range
BN2ASC    TSX                    * copy stack pointer into X
          LDAB   2,X             * point to the byte to convert
                                 * and load in ACCB
          CMPB   #0              * check for out of range
          BMI    B2AERR
          CMPB   #10
          BGE    B2AERR
          BRA    B2A
B2AERR    BCLR
          BSET   #$80
B2A       ORAB   #$30
          STAB   2,X             * put the ASCII char on the stack
          RTS
```

The stack is empty prior to setting up the subroutine call; that is, the stack pointer points to $001F ready to push a byte onto the stack. When the byte to convert is pushed onto the stack, the byte is stored at $001F and the stack pointer is updated to point to $001E, ready for the next push. When the subroutine is called, the return address (contents of the program counter which point to the next instruction to be executed after the JSR) is pushed onto the stack, one byte at a time, and the stack pointer is updated to point at $001C.

At this time, the byte for conversion is stored at (stack pointer + 3). A careful review of the TSX instruction reveals that the address loaded into the X register actually equals the contents of the stack pointer + 1. That is why the constant offset used in the indexed addressing mode is a '2' and not a '3'. Similarly, the TXS instruction loads the contents of the X register - 1 into the stack pointer.

In this example, only a single byte was passed as a parameter. It is not difficult to extend this concept to include a long list of parameters.

A fourth way of passing parameters is by passing a pointer. This may be particularly useful if we have a lot of parameters to pass. Suppose we have a list of numbers for which we want to

compute the sum. Instead of passing the numbers themselves to the subroutine, we can pass a pointer that tells where the numbers are. The code segment below is an example of how a program might pass a set of parameters via a pointer. The list of numbers to be summed is 16 entries long; no check is made for overflowing a value of 255.

```
NUMS       RMB     16                  * save bytes for list

           LDX     #NUMS               * point to start of list
           LDA B   #15                 * pass length-1
           BSR     SUM                 * go compute sum
            :
            :

* Subroutine SUM
* This subroutine computes the sum of a list of numbers
* The starting address of the list is passed in the index
* register. The length-1 is passed in ACCB.
* The subroutine returns the mean value in ACCA

SUM        LDA A   0,X                 * load first num
MORE       INX                         * point at next num
           ADD A   0,X                 * accumulate
           DEC B                       * decrement list length
           BNE     MORE                * is there more?
           RTS                         * no, return sum in ACCA
```

Another typical subroutine that will be used in the EZ-Micro Tutor Laboratories is an initialization routine. Although this routine may be only called once, at the beginning of program execution, it is common to see all initialization take place inside a subroutine. Following this practice will result in a program that is easier to read. Consider the following example where the initialization of the 'HC11 all takes place in a subroutine called INIT. Note that during the initialization phase our software must also enable any interrupts that we are supporting or want to support.

```
START      BSR    INIT         * all initialization performed here
           LDA A  #33
            :
            :

=========
INIT       LDA A   #2          * initialize stack pointer
           STA     there
                               * enable interrupts
           RTS                 * return to main program
```

Interrupt Service Routines

In the previous section we discussed the means by which program execution could be passed from your main program to a special section of code called a subroutine. The interesting thing

about returning from the subroutine was that the microprocessor restores control to the main program at the point where the subroutine was called.

Now we will consider another special section of code called an Interrupt Service Routine, which is a kind of special-purpose subroutine. Whenever an interrupt occurs that needs to be handled by the microprocessor, the microprocessor services the interrupt by transferring control to a program called an interrupt service routine. You will write the ISR especially for the situation that caused the interrupt. After servicing the interrupt, the microprocessor restores control to the main program at the point where the subroutine was called.

Interrupts

We saw that when the microprocessor makes a call to a subroutine, the internal hardware automatically saves the program counter contents. Un subroutine calls only the program counter contents are the microprocessor's responsibility. If we want to save additional registers then it is up to us to push these onto the stack.

When an interrupt occurs, however, we get more help from the microprocessor. When the microprocessor receives an interrupt (except RESET) it pushes the contents of all the registers except for the stack pointer onto the stack. These registers include the program counter (2 bytes), the index register (2 bytes), ACCA (1 byte), ACCB (1 byte), and the status register (1 byte). This means that the stack pointer contents will always be 7 less right after an interrupt. This process is called saving the program context.

The instruction RTI is similar to the RTS instruction in that its purpose is to put things back the way they were before the interrupt. It restores the program context by popping registers from the stack. Control is transferred back to the point in the program where it was interrupt because the program counter contents are restored. The I bit in the status register is automatically restored because we are returning the pre-interrupt status register from the stack.

What operations does our software have to perform in the interrupt service routine?
The answer to this question depends of course on the source of the interrupt. However, we can still make generalizations about what must be accomplished inside your ISR.

The first thing your ISR will have to do is pro-actively control whether or not you want to be interrupted from servicing this interrupt. In the EZ-Micro TUTOR Laboratories, we do not want to support nested interrupts. For this reason, we want to make sure that interrupts are disabled while we are in the interrupt service routine. This is the default case for the 'HC11 microprocessor. Also, we must make sure that when we leave the ISR that the correct interrupts are enabled. Again, the 'HC11 microprocessor takes care of this for us by default.

The second thing your ISR will have to do is to push any data onto the stack that you want to protect. Later, when your ISR is complete and you are ready to return from this interrupt, you can pop these data back off the stack to their original locations.
Next your ISR will perform the "*meat*" of the work. If the source of the interrupt was that serial data was received, then your software will go and read the serial data. We will see many more practical applications of interrupts in the later Lessons and Laboratories.

The final thing that we must do before our ISR passes control back to the main program is clear the source of the interrupt. Under many conditions, the normal program flow will be sufficient to clear the source of the interrupt. For example, vectoring to the ISR and then reading data may be enough to clear the source of a serial port interrupt. However, this may not always be the case. We must make sure that we clear the source of the interrupt. Otherwise, when we return control back to the main program there will be an interrupt pending; it will be the same interrupt that we just service. You can see that the result will be an infinite loop and we will loose control of our program.

Memory Map Conventions

After your software has been written and you are ready to download your code to the EZ-Micro Tuotr hardware platform, exactly where in the microprocessors memory should you load your code? In the EZ-Micro Tutor system architecture, the memory locations from $xxxx through $xxxx are made available to you. Typically, we will originate our programs beginning at location $1000. Any tables or lists of data will be stored in memory beginning at location $0900. Interrupt service routines have their own set of requirements.

Interrupt vectors

Where should our interrupt service routines reside in program memory? The highest locations in the 'HC11 memory are assigned to the interrupt vectors. There are two vectors for each interrupt; one for the low and one for the high-order byte of the address of its service routine. Figure 6.3 shows the assignment of the interrupt vectors for the 'HC11.

$FFD6,D7	SCI
$FFD8,D9	SPI
$FFDA,DB	Pulse Accumulator Edge
$FFDC,DD	Pulse Accumulator Overflow
$FFDE,DF	Timer
$FFE0,E1	Timer Input4/Output5
$FFE2,E3	Timer Output Compare 4
$FFE4,E5	Timer Output Compare 3
$FFE6,E7	Timer Output Compare 2
$FFE8,E9	Timer Output Compare 1
$FFEA,EB	Timer Input Capture 3
$FFEC,ED	Timer Input Capture 2
$FFEE,EF	Timer Input Capture 1
$FFF0,F1	Real-Time interrupt
$FFF2,F3	IRQ* pin
$FFF4,F5	XIRQ* pin
$FFF6,F7	Software Interrupt
$FFF8,F9	Illegal Opcode
$FFFA,FB	COP failure
$FFFC,FD	Clock Monitor Fail
$FFFE,FF	RESET

Figure 6.3 Interrupt Vectors

The location of the interrupt vectors is part of the hardware of the microprocessor, but the contents of the interrupt vectors are determined by the system designer. In the EZ-Micro Tutor architecture, there are "dummy" addresses in each user interrupt vector. This gives you a handle on the interrupt vector so that your interrupt service routine can still be pointed to. You can either place your routine at the dummy address or point to another location in memory by using a jump instruction.

Figure 6.4 below shows you where the dummy addresses are for some interrupts used in the EZ-Micro Tutor Laboratories so your interrupt service routines can be pointed to.

```
SCI      EQU     $01c4
SPI      EQU     $01c7
PAIE     EQU     $01ca
PAO      EQU     $01cd
TOF      EQU     $01d0
TOC5     EQU     $01d3
TOC4     EQU     $01d6
TOC3     EQU     $01d9
TOC2     EQU     $01dc
TOC1     EQU     $01df
TIC3     EQU     $01e2
TIC2     EQU     $01e5
TIC1     EQU     $01e8
RTI      EQU     $01eb
IRQ      EQU     $01ee
XIRQ     EQU     $01f1
SWI      EQU     $01f4
ILLOP    EQU     $01f7
COP      EQU     $01fa
CLM      EQU     $01fd
PVIC1    EQU      $01E8 IC1
PVTOF    EQU      $01D0 TOF
PVOC2    EQU      $01DC OC2
PVOC1    EQU      $01DF OC1
```

Figure 6.4 Interrupt Vectors for EZ-Micro Labs

How to Terminate Your Program

An appropriate topic for the end of this Lesson is a discussion on how your assembly language program should be terminated. Suppose the sole purpose of your program was to add two numbers. Your code would be very straight-forward and might look something like the following.

```
                    ORG     $0900
num1                RMB     1
num2                RMB     1
sum                 RMB     1

                    ORG     $1000
                    LDAA    num1
                    ADD     num2
                    STAA    sum
```

However, you must carefully consider what happens after the microprocessor executes the final STAA instruction. Unless you control what happens next, the program counter will be pointing to the "next instruction". The trouble is that there is no "next instruction". In the Labs for the EZ-Micro Tutor Board you will terminate your program with an infinite loop. Simply add the following lines to the end of your code.

```
TERM_LOOP           NOP
                    NOP
                    BRA     TERM_LOOP
```

Also, consider an example where you are writing software for a microprocessor based system that is interrupt-driven. Most of the time your software is simply waiting for an interrupt to occur. For this application you may want to incorporate a WAI instruction (Wait for Interrupt) into your code.

Questions

Laboratory

The exercises in this Laboratory are designed to tie together all of the programming techniques that were introduced in this Lesson. The simple programs that you write in this Laboratory will become the basis for the more complex programs that you will write in future Laboratories. For this reason, it is important that you understand the concepts discussed on this Lesson.

Indexed addressing will be used extensively in all of the remaining EZ-Micro Tutor Labs. Counting loops and time delay loops will be used in the routines that support the keypad and display.

As our programs become more complicated, we will naturally want to construct our code so that it is more understandable and readable. We will use subroutines throughout. Finally, because we don't want the microprocessor to be constantly polling status flags, our software will be interrupt-driven in the remaining Laboratories.

Laboratory 6_1

Write a program that uses a look-up table to compute the square of a given number. The number will be in the range from 0 to 10. While developing your program, use the following guidelines.

1. Choose $3000 as the origin of your program code segment.
2. Choose $2000 as the origin of your look-up table.
3. Assume that the number to be squared is in Accumulator B. Use LDAB #3 as the default.

Assemble your program and make sure that there were no errors. Download your program to the EZ-Micro Tutor Board. Execute your code and verify proper operation. Using the single-step feature of the EZ-Micro Manager software, step through your code from the beginning. Pay close attention to the X register and observe how indexed addressing works.

Laboratory 6_2

Write a program that creates a delay of approximately 1 second and then increments the contents of Accumulator A and delays again. This delay-then-increment process should continue indefinitely. While developing your program, use the following guidelines.

1. Choose $2000 as the origin of your program code segment.
2. Create a nested loop; use index register X as the inner loop counter and ACCB as the outer.

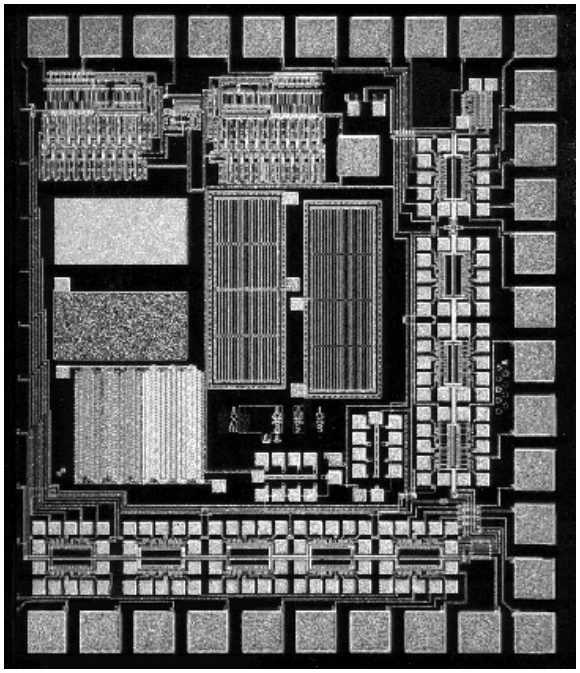
Assemble your program and make sure that there were no errors. Download your program to the EZ-Micro Tutor Board. Execute your code and verify proper operation. As a rough check, halt execution of your program after 10 seconds. You should see a value of $0A in Accumulator A.

Laboratory 6_3

Convert the program from Lab 6_1 into a subroutine. Write a main program that calls the look-up routine four successive times in order to get the square of four different numbers. Your program should terminate at that point. Use the following guidelines as you develop your program.

1. Choose $3000 as the origin of your program code segment.
2. Choose $2000 as the origin of your look-up table.
3. Find the squares of 4, 1, 8, and 3.

Assemble your program and make sure that there were no errors. Download your program to the EZ-Micro Tutor Board. Execute your code and verify proper operation. Using the single-step feature of the EZ-Micro Manager software, step through your code from the beginning. Pay close attention to the stack as program control passes from the main program to the look-up subroutine and back to the main program again.



Lesson Seven

Introduction to C Programming

History of C

"C was originally developed by Dennis Ritchie of the Bell Laboratories in 1972. The language was named "C" because it was the successor to a language named "B" (*no lasting fame, however*). C was developed as a high level language that could be used to rewrite the UNIX operating system. Today, most of the UNIX operating system and most of the programs that run under UNIX are written in C."

— C Programming Language
Earl L. Adams 1992

Benefits of C

C has an extensive library for mathematical computations, character analysis, input and output functions, hadware structure, and graphics. While some functions are used more that others, they are all offered and are able to be used by the best programmers. Why write a code to find square roots or alter strings, when there are "header" files that can be used to call these special functions. C is also a relatively easy language to learn, so that you can also write some programs for your everyday life. Yes, there are more advance programs, but it is easy enough to learn to write simple programs.

C Compiler

Basicly, a compiler takes as input a file containing "Source Code", or a set of (semi) human readable commands, and translates it into machine code. The translation is the output of the compiler. As far as running the code, that is done by the OS, at least the invokeing of the machine code. Technicaly, the computer (hardware and all) "runs" the code.

Keywords

There are 32 words defined as keywords in C. These have predefined uses and cannot be used for any other purpose in a C program. They are used by the compiler as an aid to compiling the program. They are always written in lower case. A complete list follows;

```
auto        double    int         struct
break       else      long        switch
case        enum      register    typedef
char        extern    return      union
const       float     short       unsigned
continue    for       signed      void
default     goto      sizeof      volatile
do          if        static      while
```

In addition to this list of keywords, your compiler may define a few more. If it does, they will be listed in the documentation that came with your compiler. Each of the above keywords will be defined, illustrated, and used in this tutorial.

Data Types and Constants

There are four basic data types. They are Integer, Floating Point, Double and a Character.

Integer

There are two type of Intergers namely, Whole numbers (which can be positive or negative) and Unsigned integers. In both of these types there can be long or short integers.

int is the keyword used to define integers in C Programming. The following C statement is being used to declare integer variable called weight and the value of this variable is 16.

```
int weight;
weight = 16;
```

Floating Point

The floating point numbers can be postive or negative and they are fractional numbers such as 3.14 or -2.5.

float is the keyword used to define Floating Point numbers. The following C statement is used to define float variable pi and the value of this variable is 3.14

```
float pi;
pi = 3.14;
```

Double

Double numbers also can be positive or negative and they are exponential numbers such as 3.14E5.double is the keyword used to define Double numbers. The following C statement is used to define pi with more accuracy than 2 digit decimal number defined in above example.

```
double pi;
pi = 314E+5;
```

Character

As the name implies these are the ASCII charactres such as X ,Y,Z etc. A *char* is the keyword being used to define Character. The following C statement is used to define key which as value of X

```
char key;
key = 'X';
```

Must use Single quotes to define Single character.

Sample program illustrating each data type

```
#include < stdio.h >

main()
{
     int  weight;
     float  pi;
     char  key;
     double  pi;

     weight = 16;         /* assign integer value */
     pi = 3.14;           /* assign float value */
     key = 'X';           /* assign character value */
     pi = 3.14E5;         /* assign a double value */

     printf("value of weight = %d\n", sum );
     printf("value of pi = %f\n", money );
     printf("value of key = %c\n", letter );
     printf("value of pi = %e\n", pi );
}


Sample program output
value of sum = 16
value of money = 3.140000
value of letter = X
value of pi = 3.14000e+05
```

Operators:

There are two types of Operators, Arithmetic operator and the Relational Operators.

Arithmetic Operators

The symbols of the arithmetic operators are:

Operation	Operator
Multiply	*
Divide	/
Addition	+
Subtraction	-
Increment	++
Decrement	--
Modulus	%

The Relational Operators

These type of operators are used to copmare two different variables.

Operation	Operator
equal to	===
not equal	!=
less than	<
less than or equal to	<=
greater than	>
greater than or equal to	>=

Arrays

Arrays are used to define multiple variables of the same data type. For example the values of stock prices for the different Dow 30 companies. If you do not use arrays and define as Floating point values of different stock prices the C statement will look like this

```
float stock1 = 100.5;
float stock2 = 5.75;
float stock3 = 39.5;
.....................
.....................    and so on
```

If you have to keep track of stock prices all 7,500+ publicly traded companies the program can become cumbersome.

Instead, if you define this variable as Array, it will be as follows,

```
float stock[30];
stock[0] = 100.5;
stock[1] = 5.75;
stock[2] = 39.5;
.................
................. and so on
```

In the above example, stock prices of each company is represented by stock[0], stock[1], and so on. Notice that indexing starts at 0 (zero) and not 1. Arrays, like other variables in C, must be declared before they can be used.

Individual value in Array is called an element and the square brackets [ ] are being used as part of the syntex.

In the above example of stock prices we can refer one of the element of the array as stock[i]. The value of I can be from 0 to 30.

```
stock[2] = IBM;  Assignment of the values to a Array element

IBM = x[2];    Assignment of a array element to a variable
called IBM
```

Structures

A structure is a group of data types. These data type can be same data type or it can be different. In the above example of stock prices, we can define one structure nyse (New York Stock Exachnge) that consists of different companies, for example DOW 30, FORTUNE 500 and DOTCOM 50. The structur will be defined as follows,

```
struct  nyse {
     float  dow30
     float  fortune500
     float  dotcom50
}
```

Print Formatting :

The following characters, placed after the \ character in a printf() statement, have the following effect.

Modifier	Effect on Printing
\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
****	backslash
\"	double quote
\'	single quote
\<enter>	line continuation
\nnn	nnn = octal character value
\0xnn	nn = hexadecimal value (some compilers only)

Pointers

Pointer are very widley used in C langaage. It is little bit difficult to understand at first, but once you understand the concept it will make programming very easy and very structure. In previous example when we define variable either interger, float or character, the compiler assigns memory address for each variable. For example when we defined pi as float variable, the compiler assigns a specific memory location. For simplicity let us assume this location is 1000.The value of pi (3.14) is stored in this memory location.

```
1000 3.14
```

Now let us use pointer to define the variable pi.

```
float pi = 3.14  *float_pointer
```

Here * before float defines the variable to be of type pointer.

```
float_pointer=&pi
```

The above statement assigns the memory address of pi to float_pointer. The memory address of this pointer will be something different than 1000. Again for simplicity let us assume that it is 2000. So the loation 2000 does not hold the value of pi but it holds the memory address where the value of pi is stored. In our example it will be 1000.

Memory address	Value stored
2000	1000
1000	3.14

Initially it does not explains what are the advanteges of using the pointers. But once you understand all the uses of pointers in C, especially using pointers defining arrays and strings.

There are three ways to use pointer in C.

1 Handle variable parameters passed to functions
2 Create dynamic data structures at run time
3 Another way to access information stored in arrays.

As mentioned before to go into more detail on this subject is beyond the scope of this book. We have given several references at the end of book and also we have included some freeware tutorial on the CD. Please refer any of these references for more detail explanation.

Functions

One of the main advantage of C langauage is that it offers the concept of modular and structured way of programming. Function is used to carry out different tasks. If you are familiar with assembly language programming where it was called subroutine.

All C programs has atleast one function and that is main and it is called as follows.

```
void main( )
```

There are two types of function. One type of function where no data is being passed between caller and functions itself. The other one where data is being passed.

The syntax is to include void before the name of the function if there are no data or parameters are being passed and opening and closing brackets ( ) after the name of the function. In the case where the data or parameters are bening passed you use following syntax.

```
int add ( int A, int B )

{

     return A+B;

}
```

This is the most simple function but it explains the concept. A and B are the parameters being passed to function and function return the answer.

Another example of the function is where you are printing something on the termianl. For example,

```
printf("This is EZ MICRO Tutorial in C Programmin\n")
```

Loops

You can implement Loops in C programming three different ways.

1. **for loop**
2. **while**
3. **Do…. While**

The syntax of implementing for loop is as follows,

```
     /* this program how to use for statement */
     #include <stdio.h>
     main()   /* Program introduces the for statement, prints
the value of DOW 30 stock prices */
     {
          float  stock;
          for( i = 0; i <= 30; i = i + 1 )
               printf("%d ", stock(i) );
          printf("\n");
     }
     Sample Program Output


     100.5
     5.75
     39.5
     …….
     …… and so on
```

While

The syntax for while statement is as follows,

```
     while (x < y)
     {
         x++;
         printf("%d\n",x);
     }
```

The syntax of implementing for do-while is as follows,

```
     do
     {
         x++;
         printf("%d\n",x);
     }
     while ( x < y )
```

Making Decisions

C programming language provides two ways to make decisions. One where you can use if statemant and the second choice is where you can use switch and case statement.

If Statements

The if statement allows two different opeartions to occur depending upon the circumstances. Fo example

```
if ( temp < 30)
        printf ("Too Cold for me\n");
```

switch() case:

If there are more than two circumstances switch case must be used. This makes easier to follow the program.

```
#include <stdio.h>
main()
{
     int operation, A, B, C;
     printf("enter any two numbers ");
     scanf("%d %d", &A, &B );
     printf("enter operation to be performed\n");
     printf("1=add\n");
     printf("2=subtract\n");
     printf("3=multiply\n");
     printf("4=divide\n");
     scanf("%d", &opearation );
     switch( opearation ) {
          case 1: C = A + B; break;
          case 2: C = A - B; break;
          case 3: C = A * B; break;
          case 4: C = A /B; break;
          default: printf("Invalid selection\n");
     }
     printf("%d Result %d is %d\n", C );
```

The break at the end of each case statement indicates it is the end of that particular operation and go to the statement after right brace.You must remember that you can not use expression or ranges for the case, it must be either interger or character constants.

The following operations are explained with one statement or their calling syntax. Again for detail info pl. refer to any of the sources listed at the end of book or some on the CD.

KEYBOARD INPUT

```
scanf("%d", &number);
```

ACCEPTING or outputting SINGLE CHARACTERS FROM THE KEYBOARD

```
ch = getchar();
putchar(ch);
```

FILE INPUT/OUTPUT

```
in_file = fopen( "myfile.dat", "r" );
```

DATA CONVERSION

The following functions convert between data types.

`atof()`	converts an ascii character array to a float
`atoi()`	converts an ascii character array to an integer
`itoa()`	converts an integer to a character array

UNION

This is a special data type which looks similar to a structure, but is very different. The declaration is,

```
union  mixed {
     char  letter;
     float radian;
     int   number;
};
union mixed all;
```

DECLARING VARIABLES TO BE REGISTER BASED

```
register int index;
```

DECLARING VARIABLES TO BE EXTERNAL

```
extern  int  move_number;
```

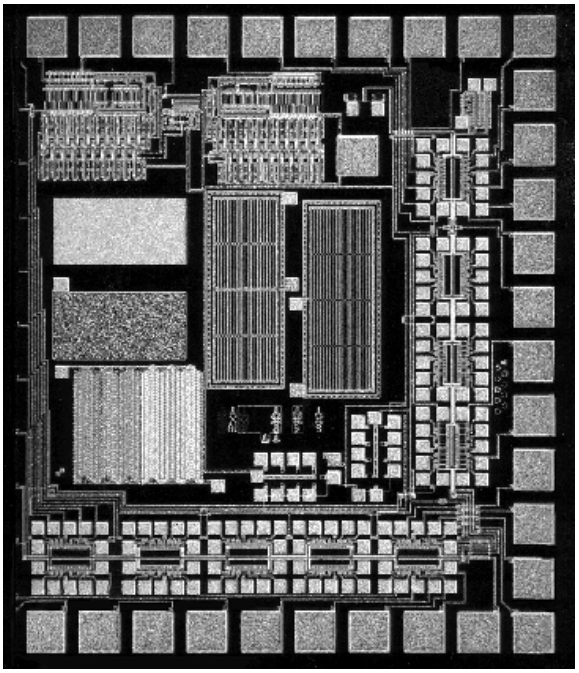
POINTERS TO FUNCTIONS

```
int  (*func_pointer)();
```

SYSTEM CALLS

Calls may be made to the Operating System to execute standard OPsys calls, eg,

```
#include <process.h>
main()  /* SYS.C    */
{
     char *command = "dir";
     system( "cls" );
     system( command );
}
```



Lesson Eight

HC11 Technical Information

Instruction set

The instruction set table lists all the valid op codes and the details of the corresponding instruction. It is a principal aid to machine language and assembly language programmers. The instruction set table lists the instructions in alphabetic order by name. Unfortunately, this means that you must know the name of an instruction before you cab find it in the table. However, the names of most instructions strongly imply the operation performed, so a little creative guessing will help you find the desired instruction quickly.

```
register a =  8 bit
register b =  8 bit
register d = 16 bit (a = MSB, b = LSB)
register x = 16 bit
register y = 16 bit
```

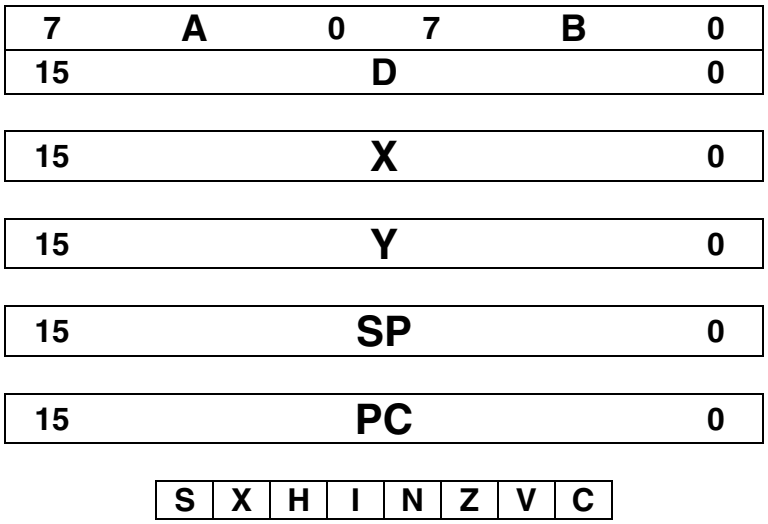
```
cc notes:
-  not changed
x  updated according to data
0  set to 0
1  set to 1
3  c|=(msn>9)
4  Most significant bit of b*
5  Set when interrupt occurs* If previously set,
   a Non-Maskable interrupt is required to exit the wait state*
```

	addressing modes							flags				
instruct*	immed	direct	ind,x	ind,y	extend	inherent	explanation	h	i	n	z	v	c
aba	-- - -	-- - -	-- - -	-- - -	-- - -	1b 1 1	a+=b	x	-	x	x	x	x
abx	-- - -	-- - -	-- - -	-- - -	-- - -	3a 1 1	x+=(uc)b	-	-	-	-	-	-
aby	-- - -	-- - -	-- - -	-- - -	-- - -	183a 1 1	y+=(uc)b	-	-	-	-	-	-
adca	89 2 2	99 3 2	a9 4 2	18a9 4 2	b9 4 3	-- - -	a+=m+c	x	-	x	x	x	x
adcb	c9 2 2	d9 3 2	e9 4 2	18e9 4 2	f9 4 3	-- - -	b+=m+c	x	-	x	x	x	x
adda	8b 2 2	9b 3 2	ab 4 2	18ab 4 2	bb 4 3	-- - -	a+=m	x	-	x	x	x	x
addb	cb 2 2	db 3 2	eb 4 2	18eb 4 2	fb 4 3	-- - -	b+=m	x	-	x	x	x	x
addd	c3 3 3	d3 4 2	e3 5 2	18e3 5 2	f3 5 3	-- - -	d+=m	-	-	x	x	x	x
anda	84 2 2	94 3 2	a4 4 2	18a4 4 2	b4 4 3	-- - -	a&=m	-	-	x	x	0	-
andb	c4 2 2	d4 3 2	e4 4 2	18e4 4 2	f4 4 3	-- - -	b&=m	-	-	x	x	0	-
asl	-- - -	-- - -	68 6 2	1868 6 2	78 6 3	-- - -	m<<=1	-	-	x	x	x	x
asla	-- - -	-- - -	-- - -	-- - -	-- - -	48 1 1	a<<=1	-	-	x	x	x	x
aslb	-- - -	-- - -	-- - -	-- - -	-- - -	58 1 1	b<<=1	-	-	x	x	x	x
asld	-- - -	-- - -	-- - -	-- - -	-- - -	05 1 1	d<<=1	-	-	x	x	x	x
asr	-- - -	-- - -	67 6 2	1867 6 2	77 6 3	-- - -	(i)m>>=1	-	-	x	x	x	x
asra	-- - -	-- - -	-- - -	-- - -	-- - -	47 1 1	(i)a>>=1	-	-	x	x	x	x
asrb	-- - -	-- - -	-- - -	-- - -	-- - -	57 1 1	(i)b>>=1	-	-	x	x	x	x
bcc	24 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(cc)	-	-	-	-	-	-
bclr	-- - -	15 6 3	1d 7 3	181d 7 3	-- - -	-- - -	m&=im	-	-	x	x	0	-
bcs	25 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(cs)	-	-	-	-	-	-
beq	27 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(eq)	-	-	-	-	-	-
bge	2c 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(ge)	-	-	-	-	-	-
bgt	2e 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(gt)	-	-	-	-	-	-
bhi	22 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(hi)	-	-	-	-	-	-

bhs	24	3	2	--	-	-	--	-	-	--	-	-	--	-	-	bra(hs)	-	-	-	-	-	-
bita	85	2	2	95	3	2	a5	4	2	18a5	4	2	b5	4	3	--	-	-	a&m	-	-	x	x	0	-
bitb	c5	2	2	d5	3	2	e5	4	2	18e5	4	2	f5	4	3	--	-	-	b&m	-	-	x	x	0	-
ble	2f	3	2	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	bra(le)	-	-	-	-	-	-
blo	25	3	2	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	bra(lo)	-	-	-	-	-	-
bls	23	3	2	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	bra(ls)	-	-	-	-	-	-
blt	2d	3	2	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	bra(lt)	-	-	-	-	-	-
bmi	2b	3	2	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	bra(mi)	-	-	-	-	-	-
bne	26	3	2	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	bra(ne)	-	-	-	-	-	-
bpl	2a	3	2	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	bra(pl)	-	-	-	-	-	-
bra	20	3	2	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	bra	-	-	-	-	-	-
brclr	--	-	-	13	6	4	1f	7	4	181f	7	4	--	-	-	--	-	-	bra(!(m&im))	-	-	x	x	0	-
brn	21	3	2	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	bra(0)	-	-	-	-	-	-
brset	--	-	-	12	6	4	1e	7	3	181e	7	3	--	-	-	--	-	-	bra(!(/m&im))	-	-	x	x	0	-
bset	--	-	-	14	6	3	1c	7	3	181c	7	3	--	-	-	--	-	-	m|=im	-	-	x	x	0	-
bsr	8d	5	2	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	bsr	-	-	-	-	-	-
bvc	28	3	2	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	bra(vc)	-	-	-	-	-	-
bvs	29	3	2	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	bra(vs)	-	-	-	-	-	-
cba	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	11	1	1	a-b	-	-	x	x	x	x
clc	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	0c	1	1	c=0	-	-	-	-	-	0
cli	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	0e	1	1	i=0	-	0	-	-	-	-
clr	--	-	-	--	-	-	6f	5	2	186f	5	2	7f	5	3	--	-	-	m=0	-	-	x	x	0	0
clra	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	4f	1	1	a=0	-	-	x	x	0	0
clrb	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	5f	1	1	b=0	-	-	x	x	0	0
clv	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	0a	1	1	v=0	-	-	-	-	0	-
cmpa	81	2	2	91	3	2	a1	4	2	18a1	4	2	b1	4	3	--	-	-	a-m	-	-	x	x	x	x
cmpb	c1	2	2	d1	3	2	e1	4	2	18e1	4	2	f1	4	3	--	-	-	b-m	-	-	x	x	x	x
com	--	-	-	--	-	-	63	6	2	1863	6	2	73	6	3	--	-	-	m=~m	-	-	x	x	0	1
coma	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	43	1	1	a=~a	-	-	x	x	0	1
comb	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	53	1	1	b=~b	-	-	x	x	0	1
cpd	1a83	5	4	1a93	6	3	1ab3	7	4	cdb3	7	4	1aa3	7	3	--	-	-	d-m	-	-	x	x	x	x
cpx	8c	3	3	9c	4	2	ac	5	2	18ac	5	2	bc	5	3	--	-	-	x-m	-	-	x	x	x	x
cpy	188c	3	3	189c	4	2	18ac	5	2	1aac	5	2	18bc	5	3	--	-	-	x-m	-	-	x	x	x	x
daa	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	19	2	1	a=da(a)	-	-	x	x	x	3
dec	--	-	-	--	-	-	6a	6	2	6a	6	2	7a	6	3	--	-	-	m-=1	-	-	x	x	x	-
deca	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	4a	1	1	a-=1	-	-	x	x	x	-
decb	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	5a	1	1	b-=1	-	-	x	x	x	-
des	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	34	1	1	s-=1	-	-	-	-	-	-
dex	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	09	1	1	x-=1	-	-	-	x	-	-
eim	--	-	-	75	6	3	65	7	3	1865	7	3	--	-	-	--	-	-	m^=im	-	-	x	x	0	-
eora	88	2	2	98	3	2	a8	4	2	18a8	4	2	b8	4	3	--	-	-	a^=m	-	-	x	x	0	-
eorb	c8	2	2	d8	3	2	e8	4	2	18e8	4	2	f8	4	3	--	-	-	b^=m	-	-	x	x	0	-
inc	--	-	-	--	-	-	6c	6	2	186c	6	2	7c	6	3	--	-	-	m+=1	-	-	x	x	x	-
inca	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	4c	1	1	a+=1	-	-	x	x	x	-
incb	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	5c	1	1	b+=1	-	-	x	x	x	-
ins	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	31	1	1	s+=1	-	-	-	-	-	-
inx	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	08	1	1	x+=1	-	-	-	x	-	-
jmp	--	-	-	--	-	-	6e	3	2	186e	3	2	7e	3	3	--	-	-	jmp	-	-	-	-	-	-
jsr	--	-	-	9d	5	2	ad	5	2	18ad	5	2	bd	6	3	--	-	-	jsr	-	-	-	-	-	-
ldaa	86	2	2	96	3	2	a6	4	2	18a6	4	2	b6	4	3	--	-	-	a=m	-	-	x	x	0	-
ldab	c6	2	2	d6	3	2	e6	4	2	18e6	4	2	f6	4	3	--	-	-	b=m	-	-	x	x	0	-
ldd	cc	3	3	dc	4	2	ec	5	2	18ec	5	2	fc	5	3	--	-	-	d=m	-	-	x	x	0	-
lds	8e	3	3	9e	4	2	ae	5	2	18ae	5	2	be	5	3	--	-	-	s=m	-	-	x	x	0	-
ldx	ce	3	3	de	4	2	ee	5	2	18ee	5	2	fe	5	3	--	-	-	x=m	-	-	x	x	0	-
lsr	--	-	-	--	-	-	64	6	2	1864	6	2	74	6	3	--	-	-	(u)m>>=1	-	-	x	x	x	x
lsra	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	44	1	1	(u)a>>=1	-	-	x	x	x	x
lsrb	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	54	1	1	(u)b>>=1	-	-	x	x	x	x
lsrd	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	04	1	1	(u)d>>=1	-	-	x	x	x	x
mul	--	-	-	--	-	-	--	-	-	--	-	-	--	-	-	3d	7	1	d=a*b	-	-	-	-	-	4

neg	-- - -	-- - -	60 6 2	1860 6 2	70 6 3	-- - -	m=-m	- - x x x x
nega	-- - -	-- - -	-- - -	-- - -	-- - -	40 1 1	a=-a	- - x x x x
negb	-- - -	-- - -	-- - -	-- - -	-- - -	50 1 1	b=-b	- - x x x x
nop	-- - -	-- - -	-- - -	-- - -	-- - -	01 1 1	nop	- - - - - -
oim	-- - -	72 6 3	62 7 3	1862 7 3	-- - -	-- - -	m|=im	- - x x 0 -
oraa	8a 2 2	9a 3 2	aa 4 2	18aa 4 2	ba 4 3	-- - -	a|=m	- - x x 0 -
orab	ca 2 2	da 3 2	ea 4 2	18ea 4 2	fa 4 3	-- - -	b|=m	- - x x 0 -
psha	-- - -	-- - -	-- - -	-- - -	-- - -	36 4 1	*s--=a	- - - - - -
pshb	-- - -	-- - -	-- - -	-- - -	-- - -	37 4 1	*s--=b	- - - - - -
pshx	-- - -	-- - -	-- - -	-- - -	-- - -	3c 5 1	*s--=x	- - - - - -
pula	-- - -	-- - -	-- - -	-- - -	-- - -	32 3 1	a=*++s	- - - - - -
pulb	-- - -	-- - -	-- - -	-- - -	-- - -	33 3 1	b=*++s	- - - - - -
pulx	-- - -	-- - -	-- - -	-- - -	-- - -	38 4 1	x=*++s	- - - - - -
rol	-- - -	-- - -	69 6 2	1869 6 2	79 6 3	-- - -	m=rol(m)	- - x x x x
rola	-- - -	-- - -	-- - -	-- - -	-- - -	49 1 1	a=rol(a)	- - x x x x
rolb	-- - -	-- - -	-- - -	-- - -	-- - -	59 1 1	b=rol(b)	- - x x x x
ror	-- - -	-- - -	66 6 2	1866 6 2	76 6 3	-- - -	m=ror(m)	- - x x x x
rora	-- - -	-- - -	-- - -	-- - -	-- - -	46 1 1	a=ror(a)	- - x x x x
rorb	-- - -	-- - -	-- - -	-- - -	-- - -	56 1 1	b=ror(b)	- - x x x x
rti	-- - -	-- - -	-- - -	-- - -	-- - -	3b a 1	rti	x x x x x x
rts	-- - -	-- - -	-- - -	-- - -	-- - -	39 5 1	rts	- - - - - -
sba	-- - -	-- - -	-- - -	-- - -	-- - -	10 1 1	a-=b	- - x x x x
sbca	82 2 2	92 3 2	a2 4 2	18a2 4 2	b2 4 3	-- - -	a-=m+c	- - x x x x
sbcb	c2 2 2	d2 3 2	e2 4 2	18e2 4 2	f2 4 3	-- - -	b-=m+c	- - x x x x
sec	-- - -	-- - -	-- - -	-- - -	-- - -	0d 1 1	c=1	- - - - - 1
sei	-- - -	-- - -	-- - -	-- - -	-- - -	0f 1 1	i=1	- 1 - - - -
sev	-- - -	-- - -	-- - -	-- - -	-- - -	0b 1 1	v=1	- - - - 1 -
slp	-- - -	-- - -	-- - -	-- - -	-- - -	1a 4 1	sleep	- - - - - -
staa	-- - -	97 3 2	a7 4 2	18a7 4 2	b7 4 3	-- - -	m=a	- - x x 0 -
stab	-- - -	d7 3 2	e7 4 2	18e7 4 2	f7 4 3	-- - -	m=b	- - x x 0 -
std	-- - -	dd 4 2	ed 5 2	18ed 5 2	fd 5 3	-- - -	m=d	- - x x 0 -
sts	-- - -	9f 4 2	af 5 2	18af 5 2	bf 5 3	-- - -	m=s	- - x x 0 -
stx	-- - -	df 4 2	ef 5 2	18ef 5 2	ff 5 3	-- - -	m=x	- - x x 0 -
suba	80 2 2	90 3 2	a0 4 2	18a0 4 2	b0 4 3	-- - -	a-=m	- - x x x x
subb	c0 2 2	d0 3 2	e0 4 2	18e0 4 2	f0 4 3	-- - -	b-=m	- - x x x x
subd	83 3 3	93 4 2	a3 5 2	18a3 5 2	b3 5 3	-- - -	d-=m	- - x x x x
swi	-- - -	-- - -	-- - -	-- - -	-- - -	3f c 1	swi	- 1 - - - -
tab	-- - -	-- - -	-- - -	-- - -	-- - -	16 1 1	b=a	- - x x 0 -
tap	-- - -	-- - -	-- - -	-- - -	-- - -	06 1 1	ccr=a	x x x x x x
tba	-- - -	-- - -	-- - -	-- - -	-- - -	17 1 1	a=b	- - x x 0 -
tim	-- - -	7b 4 3	6b 5 3	186b 5 3	-- - -	-- - -	m-im	- - x x 0 -
tpa	-- - -	-- - -	-- - -	-- - -	-- - -	07 1 1	a=ccr	- - - - - -
tst	-- - -	-- - -	6d 4 2	186d 4 2	7d 4 3	-- - -	m-0	- - x x 0 0
tsta	-- - -	-- - -	-- - -	-- - -	-- - -	4d 1 1	a-0	- - x x 0 0
tstb	-- - -	-- - -	-- - -	-- - -	-- - -	5d 1 1	b-0	- - x x 0 0
tsx	-- - -	-- - -	-- - -	-- - -	-- - -	30 1 1	x=s+1	- - - - - -
txs	-- - -	-- - -	-- - -	-- - -	-- - -	35 1 1	s=x-1	- - - - - -
wai	-- - -	-- - -	-- - -	-- - -	-- - -	3e 9 1	wait	- 5 - - - -
xgdx	-- - -	-- - -	-- - -	-- - -	-- - -	18 2 1	exg(d,x)	- - - - - -

CPU Registers



The MC68HC11 has two 8-bit accumulators (A and B), one 16-bit accumulator (D) formed by A and B; where A is the MSB and B is the LSB. The microcontroller also has two 16-bit index registers, namely X and Y. A 16-bit stackpointer (SP) and a 16 bit programcounter (PC). The condition-code-register, or flag-register, has 8 flags:

```
Stop disable (S)
X-interrupt (X)
Half carry from bit-3 (H)
I-interrupt (I)
Negative (N)
Zero (Z)
Two's complement overflow error (V)
Carry from most significant bit (C)
```

I/O Registers

The following quick-reference table shows all control registers and bits. The addresses are those resulting from a hardware reset. The registers are generally grouped according to the device they are associated with. Some registers are input/output registers while other perform only control functions. Reference appropriate text material before using these registers.

```
w = write
r = read

1000 porta
1001 reserved
1002 pioc
      w 1******* staf: strobe a flag, set at active edge of stra
pin/inactive
      w *1****** stai: hardware interrupt request when STAF=1
      w **1***** cwom: port c open-drain
      w ***1**** hnds: handshake/simple strobe mode
      w ****1*** oin:  output/input handshake select
      w *****1** pls:  STRB pulse/level active
      w ******1* ega:  rising/falling edge select for STRA
      w *******1 invb: STRB active high/low
1003 portc
1004 portb
1005 portcl
1006 reserved
1007 ddrc: data direction register for port c (1=output)
1008 portd (b0**b5)
     rw 1******* mode0 or boot: strb
     rw 1******* mode1 or test: r/w
     rw *1****** mode0 or boot: stra
     rw *1****** mode1 or test: as
     rw **1***** pd5/ss*
     rw ***1**** pd4/sck
     rw ****1*** pd3/mosi
     rw *****1** pd2/miso
     rw ******1* pd1/txd
     rw *******1 pd0/rxd
1009 ddrd: data direction register for port d (1=output)
(b0**b5)
100A porte: port e data register
100B cforc: timer compare force register
      w 1******* foc1
      w *1****** foc2
      w **1***** foc3
      w ***1**** foc4
      w ****1*** foc5
      w 11111111 force compare(s)
      w *****xxx reserved
100C oc1m: set bits to enable oc1 to control corresponding
pin(s) of port a
      w 1******* oc1m7
      w *1****** oc1m6
      w **1***** oc1m5
      w ***1**** oc1m4
```

```
      w ****1*** oc1m3
      w *****xxx reserved
      w 11111111 force compare(s)
100D oc1d: if oc1mx is set, data in oc1dx is output to port a
bit-x
 on successful oc1 compares
      w 1******* oc1d7
      w *1****** oc1d6
      w **1***** oc1d5
      w ***1**** oc1d4
      w ****1*** oc1d3
      w *****xxx reserved
      w 11111111 force compare(s)
100E tcnt: timer counter register
1010 tic1: timer input capture register
1012 tic2: timer input capture register
1014 tic3: timer input capture register
1016 toc1: timer output compare register
1018 toc2: timer output compare register
101A toc3: timer output compare register
101C toc4: timer output compare register
101E toc5: timer output compare register
1020 tctl1: timer control register 1
      w 1******* om2
      w *1****** ol2
      w **1***** om3
      w ***1**** ol3
      w ****1*** om4
      w *****1** ol4
      w ******1* om5
      w *******1 ol5
     for all pairs:
      w 00****** timer disconnected from output pin logic
      w 01****** ocx output line: toggle
      w 10****** ocx output line: 0
      w 11****** ocx output line: 1
1021 tctl2: timer control register 2
      w xx****** reserved
      w **1***** edg1b
      w ***1**** edg1a
      w ****1*** edg2b
      w *****1** edg2a
      w ******1* edg3b
      w *******1 edg3a
     for all pairs:
      w **00**** capture: disabled
      w **01**** capture: on rising edge only
      w **10**** capture: on falling edge only
```

```
      w **11**** capture: on any edge
1022 tmsk1: main timer interrupt mask reg 1
      w 1******* oc1l: output compare 1 interrupt enable
      w *1****** oc2l: output compare 2 interrupt enable
      w **1***** oc3l: output compare 3 interrupt enable
      w ***1**** oc4l: output compare 4 interrupt enable
      w ****1*** oc5l: output compare 5 interrupt enable
      w *****1** ic1l: input compare 1 interrupt enable
      w ******1* ic2l: input compare 2 interrupt enable
      w *******1 ic3l: input compare 3 interrupt enable
1023 tflg1: main timer interrupt flag reg 1
      w 1******* oc1f: clear output compare flag 1
      w *1****** oc2f: clear output compare flag 2
      w **1***** oc3f: clear output compare flag 3
      w ***1**** oc4f: clear output compare flag 4
      w ****1*** oc5f: clear output compare flag 5
      w *****1** ic1f: clear input capture flag 1
      w ******1* ic2f: clear input capture flag 2
      w *******1 ic3f: clear input capture flag 3
1024 tmsk2: misc timer interrupt mask reg 2
      w 1******* toi:   timer overflow interrupt enable
      w *1****** rtii:  interrupt enable
      w **1***** paovi: pulse accumulator overflow interrupt
enable
      w ***1**** paii:  pulse accumulator input interrupt
enable/disable
      w ****xx** reserved
      w ******00 pr1,pr0: timer prescale factor  1
      w ******01 pr1,pr0: timer prescale factor  4
      w ******10 pr1,pr0: timer prescale factor  8
      w ******11 pr1,pr0: timer prescale factor 16
1025 tflg2: misc timer interrupt flag reg 2
      w 1******* tof:   clear timer overflow flag
      w *1****** rtif:  clear real time (periodic) interrupt
flag
      w **1***** paovf: clear pulse accumulator overflow flag
      w ***1**** paif:  clear paif pulse accumulator input edge
flag
      w ****xxxx reserved
1026 pactl: pulse accumulator control register
      w 1******* ddra7: data direction for port a bit7 is
output/input
      w *1****** paen:  enable/disable pulse accumulator system
enable
      w **1***** pamod: pulse acc mode: gated time
accumulation/event counter
      w ***1**** pedge: pulse acc edge ctrl: rising/falling
edges
```

```
      w ****xx** reserved
      w ******00 rtr1,rtr0: interrupt rate divide by 2^13
      w ******01 rtr1,rtr0: interrupt rate divide by 2^14
      w ******10 rtr1,rtr0: interrupt rate divide by 2^15
      w ******11 rtr1,rtr0: interrupt rate divide by 2^16
1027 pacnt: pulse accumulator count register
1028 spcr: spi control register
      w 1******* spie: spi interrupt enable
      w *1****** spe:  spi system enable
      w **1***** dwom: port d: open-drain/normal
      w ***1**** mstr: master/slave mode
      w ****1*** cpol: clock polarity
      w *****1** cpha: clock phase
      w ******00 spr1,spr0: spi e-clock divided by  2
      w ******01 spr1,spr0: spi e-clock divided by  4
      w ******10 spr1,spr0: spi e-clock divided by  8
      w ******11 spr1,spr0: spi e-clock divided by 16
1029 spsr: spi status register
     r  1******* spif: spi interrupt request
     r  *1****** wcol: write collision status flag
     r  ***1**** modf: spi mode error interrupt status flag
     r  **x*xxxx reserved
102A spdr: spi data register
102B baud: sci baud rate control register
      w 1******* tclr: clear baud counter chain (test only)
      w *x****** reserved
      w **00**** scp1,scp0: serial prescaler select: divide e-
clock by  1
      w **01**** scp1,scp0: serial prescaler select: divide e-
clock by  2
      w **10**** scp1,scp0: serial prescaler select: divide e-
clock by  4
      w **11**** scp1,scp0: serial prescaler select: divide e-
clock by 13
      w ****1*** rckb: sci baud rate clock test (test only)
      w *****000 scr2,scr1,scr0: prescaler output divide by   1
      w *****001 scr2,scr1,scr0: prescaler output divide by   2
      w *****010 scr2,scr1,scr0: prescaler output divide by   4
      w *****011 scr2,scr1,scr0: prescaler output divide by   8
      w *****100 scr2,scr1,scr0: prescaler output divide by  16
      w *****101 scr2,scr1,scr0: prescaler output divide by  32
      w *****110 scr2,scr1,scr0: prescaler output divide by  64
      w *****111 scr2,scr1,scr0: prescaler output divide by 128
102C sccr1: sci control register 1
      w 1******* r8:   receive bit 8
      w *1****** t8:   transmit bit 8
      w ***1**** m:    ninth data bit
```

```
      w ****1*** wake: wake up by address mark/idle line (msb/no
low)
      w **x**xxx reserved
102D sccr2: sci control register 2
      w 1******* tie:  transmit interrupt enable
      w *1****** tcie: transmit complete interrupt enable
      w **1***** rie:  receiver interrupt enable
      w ***1**** ilie: enable/disable idle line interrupts
      w ****1*** te:   transmitter enable (toggle to queue idle
character)
      w *****1** re:   receiver enable on/off
      w ******1* rwu:  receiver asleep/normal
      w *******1 sbk:  send break
102E scsr: sci status register
     r  1******* tdre: transmit data register empty flag
     r  *1****** tc:   transmit complete flag
     r  **1***** rdrf: receive data register full flag
     r  ***1**** idle: idle line detected flag
     r  ****1*** or:   over-run error flag
     r  *****1** nf:   noise error flag
     r  ******1* fe:   framing error flag
     r  *******x reserved
102F scdr: sci data register
1030 adctl: a/d control/status register
     r  1******* ccf:  conversions complete flag (sets after
fourth conversion)
     rw *x****** reserved
      w **1***** scan: convert continuously/4 conversions and
stop
      w ***1**** mult: convert four channel group/single channel
     rw ****0000 cd,cc,cb,ca: ad0 port e bit0
     rw ****0001 cd,cc,cb,ca: ad1 port e bit1
     rw ****0010 cd,cc,cb,ca: ad2 port e bit2
     rw ****0011 cd,cc,cb,ca: ad3 port e bit3
     rw ****0100 cd,cc,cb,ca: ad4 port e bit4
     rw ****0101 cd,cc,cb,ca: ad5 port e bit5
     rw ****0110 cd,cc,cb,ca: ad6 port e bit6
     rw ****0111 cd,cc,cb,ca: ad7 port e bit7
     rw ****10xx cd,cc,cb,ca: reserved
     rw ****1100 cd,cc,cb,ca: Vref hi
     rw ****1101 cd,cc,cb,ca: Vref low
     rw ****1110 cd,cc,cb,ca: Vref hi/2
     rw ****1111 cd,cc,cb,ca: test/reserved
1031 adr1
1032 adr2
1033 adr3
1034 adr4
1035 reserved
```

```
1036 reserved
1037 reserved
1038 reserved
1039 option: system configuration options
      w 1******* adpu: a->d system powered up/down
      w *1****** csel: a->d and ee use an internal-r-c/system-e
clock
      w **1***** irqe: irq configured for falling edges/low
level
      w ***1**** dly:  enable/disable oscillator start-up delay
(from stop)
      w ****1*** cme:  slow or stopped clocks cause
reset/disabled
      w *****x** reserved
      w ******00 cr1,cr0: e/2^15 divided by  1
      w ******01 cr1,cr0: e/2^15 divided by  4
      w ******10 cr1,cr0: e/2^15 divided by 16
      w ******11 cr1,cr0: e/2^15 divided by 64
103A coprst: arm/reset cop timer circuitry, write $55 and $aa to
reset cop watchdog timer
103B pprog: eeprom programming register
      w 1******* odd:   program  odd rows in half of eeprom
(test only)
      w *1****** even:  program even rows in half of eeprom
(test only)
      w **x***** reserved
      w ***1**** byte:  erase only one byte/row or all of eeprom
      w ****1*** row:   erase only one 16 byte row/all 512 bytes
of eeprom
      w *****1** erase: erase/normal read of program mode
      w ******1* eelat: eeprom busses configured for program or
erase/read
      w *******1 eepgm: program or erase power switched on/off
                        to program eeprom:
                         * set eelat
                         * write data to desired address
                         * set eepgm for the required
programming time
                        to erase eeprom:
                         * select row = 1/0
                         * select byte = 1/0
                         * set erase and eelat = 1
                         * write to an eeprom address to be
erased
                         * set eepgm for the required erase time
period
103C hprio: highest priority interrupt and misc*
      w 1******* rboot: boot rom enabled/not in map (normal)
```

```
      w *00***** smod,mda: single chip mode
      w *01***** smod,mda: expanded multiplexed mode
      w *10***** smod,mda: special bootstrap
      w *11***** smod,mda: special test
      w ***1**** irv:      data from internal reads visible/not
                           on external bus
                           reset to in test/boot mode: 1, in
normal modes: 0
     r  ****0000 psel3,2,1,0: timer overflow
     r  ****0001 psel3,2,1,0: pulse accum* overflow
     r  ****0010 psel3,2,1,0: puls accum* input edge
     r  ****0011 psel3,2,1,0: spi serial xfer complete
     r  ****0100 psel3,2,1,0: sci serial system
     r  ****0101 psel3,2,1,0: reserved (default to irq)
     r  ****0110 psel3,2,1,0: irq (ext pin or parallel i/o
     r  ****0111 psel3,2,1,0: real time interrupt
     r  ****1000 psel3,2,1,0: timer input capture 1
     r  ****1001 psel3,2,1,0: timer input capture 2
     r  ****1010 psel3,2,1,0: timer input capture 3
     r  ****1011 psel3,2,1,0: timer output compare 1
     r  ****1100 psel3,2,1,0: timer output compare 2
     r  ****1101 psel3,2,1,0: timer output compare 3
     r  ****1110 psel3,2,1,0: timer output compare 4
     r  ****1111 psel3,2,1,0: timer output compare 5
103D init: ram and i/o mapping register
      w xxxx**** ram3,2,1,0: ram      block position
(x000**x0ff)
      w ****xxxx reg3,2,1,0: register block position
(x000**x03f)
        00000001 at reset
103E test1: factory test register
      w 1******* tilop: test illegal opcode
      w *x****** reserved
      w **1***** occr:  output condition code register status to
timer port
      w ***1**** cbyp:  timer divider chain bypass
      w ****1*** disr:  disable resets from cop and clock
monitor
      w *****1** fcm:   force clock monitor failure
      w ******1* fcop:  force cop watchdog failure
      w *******1 tcon:  test configuration
103F config: configuration control register
      w ****1*** nosec: dis/enable security mode   (security
only if mask option)
      w *****1** nocop: dis/enable cop system
      w ******1* romon: en/disable    rom at $e000**$ffff
      w *******1 eeon:  en/disable eeprom at $b600**$b7ff
```

note: The bits of this register are implemented with eeprom cells programming and erasure follow normal eeprom procedures. The erased state of this location is $0f. A new value programmed into this register is not readable until after a subsequent reset sequence.

Interrupt vectors

In this list you can find the many interrupt vectors of the HC11. The interrupts are ranked from a low to high priority.

```
    cc register mask
        /   local mask
       /   /
FFC0 ? ????? reserved
FFC2 ? ????? reserved
FFC4 ? ????? reserved
FFC6 ? ????? reserved
FFC8 ? ????? reserved
FFCA ? ????? reserved
FFCC ? ????? reserved
FFCE ? ????? reserved

         cc register mask
        /   local mask
       /   /
FFD0 ? ????? reserved
FFD2 ? ????? reserved
FFD4 ? ????? reserved
FFD6 i div*  sci serial system
             masks:
             rie  for SCI receive data register full
             rie  for SCI receiver overrun
             ilie for SCI idle line detect
             tie  for SCI transmit data register empty
             tcie for SCI transmit complete
FFD8 i psie  SPI serial transfer complete
FFDA i paii  pulse accumulator input edge
FFDC i paovi pulse accumulator overflow
FFDE i toi   timer overflow
         cc register mask
        /   local mask
       /   /
FFE0 i oc5i  timer output compare 5
FFE2 i oc4i  timer output compare 4
ffe4 i oc3i  timer output compare 3
FFE6 i oc2i  timer output compare 2
```

```
FFE8 i oc1i  timer output compare 1
FFEA i ic3i  timer input capture 3
FFEC i ic2i  timer input capture 2
FFEE i ic1i  timer input capture 1

        cc register mask
       /   local mask
      /   /
FFF0 i rtii  real time interrupt
FFF2 i div*  irq (external pin or parallel i/o)
             masks:
             none for external pin
             stai for parallel i/o handshake
FFF4 x none  xirq pin (pseudo non-maskable interrupt)
FFF6 - none  swi
FFF8 - none  illegal opcode trap
FFFA - nocop cop timeout
FFFC - cme   cop clock monitor timeout
FFFE - none  reset
```

Instruction set

The instruction set table lists all the valid op codes and the details of the corresponding instruction. It is a principal aid to machine language and assembly language programmers. The instruction set table lists the instructions in alphabetic order by name. Unfortunately, this means that you must know the name of an instruction before you cab find it in the table. However, the names of most instructions strongly imply the operation performed, so a little creative guessing will help you find the desired instruction quickly.

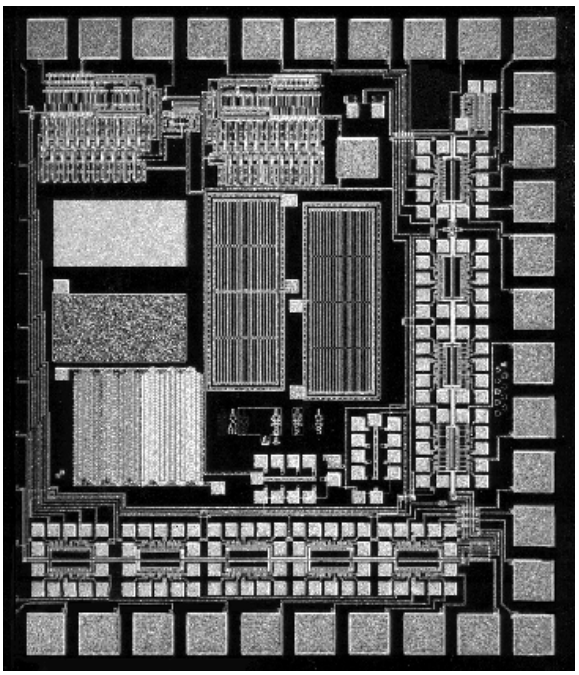
```
register a =  8 bit
register b =  8 bit
register d = 16 bit (a = MSB, b = LSB)
register x = 16 bit
register y = 16 bit
```

```
cc notes:
-  not changed
x  updated according to data
0  set to 0
1  set to 1
3  c|=(msn>9)
4  Most significant bit of b.
5  Set when interrupt occurs. If previously set,
   a Non-Maskable interrupt is required to exit the wait state.
```

	addressing modes							flags
instruct.	immed	direct	ind,x	ind,y	extend	inherent	explanation	h i n z v c
aba	-- - -	-- - -	-- - -	-- - -	-- - -	1b 1 1	a+=b	x - x x x x
abx	-- - -	-- - -	-- - -	-- - -	-- - -	3a 1 1	x+=(uc)b	- - - - - -

aby	-- - -	-- - -	-- - -	-- - -	-- - -	183a 1 1	y+=(uc)b	- - - - - - -
adca	89 2 2	99 3 2	a9 4 2	18a9 4 2	b9 4 3	-- - -	a+=m+c	x - x x x x
adcb	c9 2 2	d9 3 2	e9 4 2	18e9 4 2	f9 4 3	-- - -	b+=m+c	x - x x x x
adda	8b 2 2	9b 3 2	ab 4 2	18ab 4 2	bb 4 3	-- - -	a+=m	x - x x x x
addb	cb 2 2	db 3 2	eb 4 2	18eb 4 2	fb 4 3	-- - -	b+=m	x - x x x x
addd	c3 3 3	d3 4 2	e3 5 2	18e3 5 2	f3 5 3	-- - -	d+=m	- - x x x x
anda	84 2 2	94 3 2	a4 4 2	18a4 4 2	b4 4 3	-- - -	a&=m	- - x x 0 -
andb	c4 2 2	d4 3 2	e4 4 2	18e4 4 2	f4 4 3	-- - -	b&=m	- - x x 0 -
asl	-- - -	-- - -	68 6 2	1868 6 2	78 6 3	-- - -	m<<=1	- - x x x x
asla	-- - -	-- - -	-- - -	-- - -	-- - -	48 1 1	a<<=1	- - x x x x
aslb	-- - -	-- - -	-- - -	-- - -	-- - -	58 1 1	b<<=1	- - x x x x
asld	-- - -	-- - -	-- - -	-- - -	-- - -	05 1 1	d<<=1	- - x x x x
asr	-- - -	-- - -	67 6 2	1867 6 2	77 6 3	-- - -	(i)m>>=1	- - x x x x
asra	-- - -	-- - -	-- - -	-- - -	-- - -	47 1 1	(i)a>>=1	- - x x x x
asrb	-- - -	-- - -	-- - -	-- - -	-- - -	57 1 1	(i)b>>=1	- - x x x x
bcc	24 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(cc)	- - - - - -
bclr	-- - -	15 6 3	1d 7 3	181d 7 3	-- - -	-- - -	m&=im	- - x x 0 -
bcs	25 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(cs)	- - - - - -
beq	27 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(eq)	- - - - - -
bge	2c 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(ge)	- - - - - -
bgt	2e 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(gt)	- - - - - -
bhi	22 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(hi)	- - - - - -
bhs	24 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(hs)	- - - - - -
bita	85 2 2	95 3 2	a5 4 2	18a5 4 2	b5 4 3	-- - -	a&m	- - x x 0 -
bitb	c5 2 2	d5 3 2	e5 4 2	18e5 4 2	f5 4 3	-- - -	b&m	- - x x 0 -
ble	2f 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(le)	- - - - - -
blo	25 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(lo)	- - - - - -
bls	23 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(ls)	- - - - - -
blt	2d 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(lt)	- - - - - -
bmi	2b 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(mi)	- - - - - -
bne	26 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(ne)	- - - - - -
bpl	2a 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(pl)	- - - - - -
bra	20 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra	- - - - - -
brclr	-- - -	13 6 4	1f 7 4	181f 7 4	-- - -	-- - -	bra(!(m&im))	- - x x 0 -
brn	21 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(0)	- - - - - -
brset	-- - -	12 6 4	1e 7 3	181e 7 3	-- - -	-- - -	bra(!(/m&im))	- - x x 0 -
bset	-- - -	14 6 3	1c 7 3	181c 7 3	-- - -	-- - -	m|=im	- - x x 0 -
bsr	8d 5 2	-- - -	-- - -	-- - -	-- - -	-- - -	bsr	- - - - - -
bvc	28 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(vc)	- - - - - -
bvs	29 3 2	-- - -	-- - -	-- - -	-- - -	-- - -	bra(vs)	- - - - - -
cba	-- - -	-- - -	-- - -	-- - -	-- - -	11 1 1	a-b	- - x x x x
clc	-- - -	-- - -	-- - -	-- - -	-- - -	0c 1 1	c=0	- - - - - 0
cli	-- - -	-- - -	-- - -	-- - -	-- - -	0e 1 1	i=0	- 0 - - - -
clr	-- - -	-- - -	6f 5 2	186f 5 2	7f 5 3	-- - -	m=0	- - x x 0 0
clra	-- - -	-- - -	-- - -	-- - -	-- - -	4f 1 1	a=0	- - x x 0 0
clrb	-- - -	-- - -	-- - -	-- - -	-- - -	5f 1 1	b=0	- - x x 0 0
clv	-- - -	-- - -	-- - -	-- - -	-- - -	0a 1 1	v=0	- - - - 0 -
cmpa	81 2 2	91 3 2	a1 4 2	18a1 4 2	b1 4 3	-- - -	a-m	- - x x x x
cmpb	c1 2 2	d1 3 2	e1 4 2	18e1 4 2	f1 4 3	-- - -	b-m	- - x x x x
com	-- - -	-- - -	63 6 2	1863 6 2	73 6 3	-- - -	m=~m	- - x x 0 1
coma	-- - -	-- - -	-- - -	-- - -	-- - -	43 1 1	a=~a	- - x x 0 1
comb	-- - -	-- - -	-- - -	-- - -	-- - -	53 1 1	b=~b	- - x x 0 1
cpd	1a83 5 4	1a93 6 3	1ab3 7 4	cdb3 7 4	1aa3 7 3	-- - -	d-m	- - x x x x
cpx	8c 3 3	9c 4 2	ac 5 2	18ac 5 2	bc 5 3	-- - -	x-m	- - x x x x
cpy	188c 3 3	189c 4 2	18ac 5 2	1aac 5 2	18bc 5 3	-- - -	x-m	- - x x x x
daa	-- - -	-- - -	-- - -	-- - -	-- - -	19 2 1	a=da(a)	- - x x x 3
dec	-- - -	-- - -	6a 6 2	6a 6 2	7a 6 3	-- - -	m-=1	- - x x x -
deca	-- - -	-- - -	-- - -	-- - -	-- - -	4a 1 1	a-=1	- - x x x -
decb	-- - -	-- - -	-- - -	-- - -	-- - -	5a 1 1	b-=1	- - x x x -
des	-- - -	-- - -	-- - -	-- - -	-- - -	34 1 1	s-=1	- - - - - -
dex	-- - -	-- - -	-- - -	-- - -	-- - -	09 1 1	x-=1	- - - x - -
eim	-- - -	75 6 3	65 7 3	1865 7 3	-- - -	-- - -	m^=im	- - x x 0 -
eora	88 2 2	98 3 2	a8 4 2	18a8 4 2	b8 4 3	-- - -	a^=m	- - x x 0 -
eorb	c8 2 2	d8 3 2	e8 4 2	18e8 4 2	f8 4 3	-- - -	b^=m	- - x x 0 -
inc	-- - -	-- - -	6c 6 2	186c 6 2	7c 6 3	-- - -	m+=1	- - x x x -
inca	-- - -	-- - -	-- - -	-- - -	-- - -	4c 1 1	a+=1	- - x x x -
incb	-- - -	-- - -	-- - -	-- - -	-- - -	5c 1 1	b+=1	- - x x x -
ins	-- - -	-- - -	-- - -	-- - -	-- - -	31 1 1	s+=1	- - - - - -
inx	-- - -	-- - -	-- - -	-- - -	-- - -	08 1 1	x+=1	- - - x - -
jmp	-- - -	-- - -	6e 3 2	186e 3 2	7e 3 3	-- - -	jmp	- - - - - -
jsr	-- - -	9d 5 2	ad 5 2	18ad 5 2	bd 6 3	-- - -	jsr	- - - - - -
ldaa	86 2 2	96 3 2	a6 4 2	18a6 4 2	b6 4 3	-- - -	a=m	- - x x 0 -

ldab	c6 2 2	d6 3 2	e6 4 2	18e6 4 2	f6 4 3	-- - -	b=m	- - x x 0 -
ldd	cc 3 3	dc 4 2	ec 5 2	18ec 5 2	fc 5 3	-- - -	d=m	- - x x 0 -
lds	8e 3 3	9e 4 2	ae 5 2	18ae 5 2	be 5 3	-- - -	s=m	- - x x 0 -
ldx	ce 3 3	de 4 2	ee 5 2	18ee 5 2	fe 5 3	-- - -	x=m	- - x x 0 -
lsr	-- - -	-- - -	64 6 2	1864 6 2	74 6 3	-- - -	(u)m>>=1	- - x x x x
lsra	-- - -	-- - -	-- - -	-- - -	-- - -	44 1 1	(u)a>>=1	- - x x x x
lsrb	-- - -	-- - -	-- - -	-- - -	-- - -	54 1 1	(u)b>>=1	- - x x x x
lsrd	-- - -	-- - -	-- - -	-- - -	-- - -	04 1 1	(u)d>>=1	- - x x x x
mul	-- - -	-- - -	-- - -	-- - -	-- - -	3d 7 1	d=a*b	- - - - - 4
neg	-- - -	-- - -	60 6 2	1860 6 2	70 6 3	-- - -	m=-m	- - x x x x
nega	-- - -	-- - -	-- - -	-- - -	-- - -	40 1 1	a=-a	- - x x x x
negb	-- - -	-- - -	-- - -	-- - -	-- - -	50 1 1	b=-b	- - x x x x
nop	-- - -	-- - -	-- - -	-- - -	-- - -	01 1 1	nop	- - - - - -
oim	-- - -	72 6 3	62 7 3	1862 7 3	-- - -	-- - -	m|=im	- - x x 0 -
oraa	8a 2 2	9a 3 2	aa 4 2	18aa 4 2	ba 4 3	-- - -	a|=m	- - x x 0 -
orab	ca 2 2	da 3 2	ea 4 2	18ea 4 2	fa 4 3	-- - -	b|=m	- - x x 0 -
psha	-- - -	-- - -	-- - -	-- - -	-- - -	36 4 1	*s--=a	- - - - - -
pshb	-- - -	-- - -	-- - -	-- - -	-- - -	37 4 1	*s--=b	- - - - - -
pshx	-- - -	-- - -	-- - -	-- - -	-- - -	3c 5 1	*s--=x	- - - - - -
pula	-- - -	-- - -	-- - -	-- - -	-- - -	32 3 1	a=*++s	- - - - - -
pulb	-- - -	-- - -	-- - -	-- - -	-- - -	33 3 1	b=*++s	- - - - - -
pulx	-- - -	-- - -	-- - -	-- - -	-- - -	38 4 1	x=*++s	- - - - - -
rol	-- - -	-- - -	69 6 2	1869 6 2	79 6 3	-- - -	m=rol(m)	- - x x x x
rola	-- - -	-- - -	-- - -	-- - -	-- - -	49 1 1	a=rol(a)	- - x x x x
rolb	-- - -	-- - -	-- - -	-- - -	-- - -	59 1 1	b=rol(b)	- - x x x x
ror	-- - -	-- - -	66 6 2	1866 6 2	76 6 3	-- - -	m=ror(m)	- - x x x x
rora	-- - -	-- - -	-- - -	-- - -	-- - -	46 1 1	a=ror(a)	- - x x x x
rorb	-- - -	-- - -	-- - -	-- - -	-- - -	56 1 1	b=ror(b)	- - x x x x
rti	-- - -	-- - -	-- - -	-- - -	-- - -	3b a 1	rti	x x x x x x
rts	-- - -	-- - -	-- - -	-- - -	-- - -	39 5 1	rts	- - - - - -
sba	-- - -	-- - -	-- - -	-- - -	-- - -	10 1 1	a-=b	- - x x x x
sbca	82 2 2	92 3 2	a2 4 2	18a2 4 2	b2 4 3	-- - -	a-=m+c	- - x x x x
sbcb	c2 2 2	d2 3 2	e2 4 2	18e2 4 2	f2 4 3	-- - -	b-=m+c	- - x x x x
sec	-- - -	-- - -	-- - -	-- - -	-- - -	0d 1 1	c=1	- - - - - 1
sei	-- - -	-- - -	-- - -	-- - -	-- - -	0f 1 1	i=1	- 1 - - - -
sev	-- - -	-- - -	-- - -	-- - -	-- - -	0b 1 1	v=1	- - - - 1 -
slp	-- - -	-- - -	-- - -	-- - -	-- - -	1a 4 1	sleep	- - - - - -
staa	-- - -	97 3 2	a7 4 2	18a7 4 2	b7 4 3	-- - -	m=a	- - x x 0 -
stab	-- - -	d7 3 2	e7 4 2	18e7 4 2	f7 4 3	-- - -	m=b	- - x x 0 -
std	-- - -	dd 4 2	ed 5 2	18ed 5 2	fd 5 3	-- - -	m=d	- - x x 0 -
sts	-- - -	9f 4 2	af 5 2	18af 5 2	bf 5 3	-- - -	m=s	- - x x 0 -
stx	-- - -	df 4 2	ef 5 2	18ef 5 2	ff 5 3	-- - -	m=x	- - x x 0 -
suba	80 2 2	90 3 2	a0 4 2	18a0 4 2	b0 4 3	-- - -	a-=m	- - x x x x
subb	c0 2 2	d0 3 2	e0 4 2	18e0 4 2	f0 4 3	-- - -	b-=m	- - x x x x
subd	83 3 3	93 4 2	a3 5 2	18a3 5 2	b3 5 3	-- - -	d-=m	- - x x x x
swi	-- - -	-- - -	-- - -	-- - -	-- - -	3f c 1	swi	- 1 - - - -
tab	-- - -	-- - -	-- - -	-- - -	-- - -	16 1 1	b=a	- - x x 0 -
tap	-- - -	-- - -	-- - -	-- - -	-- - -	06 1 1	ccr=a	x x x x x x
tba	-- - -	-- - -	-- - -	-- - -	-- - -	17 1 1	a=b	- - x x 0 -
tim	-- - -	7b 4 3	6b 5 3	186b 5 3	-- - -	-- - -	m-im	- - x x 0 -
tpa	-- - -	-- - -	-- - -	-- - -	-- - -	07 1 1	a=ccr	- - - - - -
tst	-- - -	-- - -	6d 4 2	186d 4 2	7d 4 3	-- - -	m-0	- - x x 0 0
tsta	-- - -	-- - -	-- - -	-- - -	-- - -	4d 1 1	a-0	- - x x 0 0
tstb	-- - -	-- - -	-- - -	-- - -	-- - -	5d 1 1	b-0	- - x x 0 0
tsx	-- - -	-- - -	-- - -	-- - -	-- - -	30 1 1	x=s+1	- - - - - -
txs	-- - -	-- - -	-- - -	-- - -	-- - -	35 1 1	s=x-1	- - - - - -
wai	-- - -	-- - -	-- - -	-- - -	-- - -	3e 9 1	wait	- 5 - - - -
xgdx	-- - -	-- - -	-- - -	-- - -	-- - -	18 2 1	exg(d,x)	- - - - - -



Lesson Nine

Interfacing Input/Output and Timer Devices

This Lesson describes parallel I/O operations and the main timer system in the 'HC11. These subsystems were introduced to the student in Lesson Two where we discussed the 'HC11E0 architecture. In that Lesson, we learned that there are two general-purpose I/O ports built into the MCU. Also, the main 16-bittimer has three input capture lines, five output-compare lines, and a real-time interrupt function. Overall, there are 32 I/O pins that link your software to the real-time outside world.

Small package size and low pin count are important for low-cost applications. However, each I/O function will potentially add more pins to the package. Let's make a wish-list of I/O functions that we may want to have on our microcontroller or microprocessor and keep a running tally of pin count.

I/O wish list	pins
16 general-purpose I/O ports	16
3 input capture ports based on timer	3
5 output control ports based on timer	5
8 expanded data bus	8
16 expanded address bus	16
1 R/W control line	1
1 asynchronous serial port	2
1 synchronous serial port	4
total	**55**

If we add power and ground, the crystal oscillator, and reset, then it is easy to see that we can quickly get into packages that contain more than 60 pins or more. These packages are expensive to manufacture and the large footprint takes up expensive real estate on your PC board. We need some means of minimizing pin count.

One of the ways that microcontrollers can maximize the functionality of the I/O pins is to implement shared functions for each of the ports. This is the case for all of the 32 input/output pins on the 'HC08D0. Port A is shared with the timer and input/output capture functions. Port B and port C are shared with the expanded mode of operation by multiplexing the external address and data bus. Port D is shared with two serial ports and two control lines needed for expanded mode.

In the following sections of this Lesson and later in Lesson Eight, each of these I/O subsystems will be discussed in detail. By discussing each of the four ports in a separate section in this Lesson, we will cover general-purpose I/O, the timer subsystem, the single-chip mode of operation, and the expanded mode of operation in this Lesson. The two serial ports will be discussed in Lesson Ten.

Port A

The I/O for the timer subsystem is multiplexed with general-purpose I/O on port A. Pins PA0, PA1, and PA2 can be used for general-purpose input or as input edge-detect pins. Port A reads are completely independent of timer input-capture functions. Pins PA3 - PA6 can be used for general purpose output or as output-compare pins. When one of these pins is being used for an

output-compare function it cannot be written directly as if it were a general-purpose output. The functionality of these pins is controlled by the PAEN bit in the PACTL register. Pin PA7 is bidirectional. Its direction is determined by the data direction bit (DDRA5) in the PACTL register.

The two registers associated with port A are shown below in Figure 1 and Figure 2. In the PACTL register we are concerned with bit 7 (DDRA7) and bit 6 (PAEN). We will discuss the functionality of the remaining five bits later in this section. The PORTA register is the data register for port A.

7	6	5	4	3	2	1	0	
DDRA7	PAEN	PAMOD	PEDGE	0	0	RTR1	RTR0	PACTL
								$0026

Figure 9.1 Port A control register

7	6	5	4	3	2	1	0		
D7	D6	D5	D4	D3	D2	D1	D0		PORTA
									$0000

Figure 9.2 Port A data register

The three tables below summarize the programming options for controlling the functionality of port A. Table 9.1 describes the three input pins, Table 9.2 describes the four output pins, and Table 9.3 describes the one input-output pin.

PAEN	DDRA7	Function
0	X	general-purpose inputs
1	X	input capture pins

Table 9.1 PA0/PA1/PA2 Functionality

PAEN	DDRA7	Function
0	X	general-purpose outputs
1	X	output-compare pins

Table 9.2 PA3/PA4/PA5/PA6 Functionality

PAEN	DDRA7	Function
0	0	general-purpose input
0	1	general-purpose output
1	0	pulse accumulator input
1	1	output compare pin

Table 9.3 PA7 Functionality

Having discussed the programmable modes of operation for port A, we must now go into more detail about the main timer. The main timer is at the heart of the pulse accumulator subsystem and the output compare subsystem.

Main Timer

The main timer system is based on a free-running 16-bit timer with a four-stage programmable prescaler. Three independent input-capture functions can be used to automatically latch the time when a selected transition is detected at the corresponding timer input pin. Five output-compare functions are included to automatically generate output signals for timing software delays. Also a programmable periodic interrupt is available off the main timing chain. The user can select one of four rates for the interrupt.

The timer subsytem is controlled by a set of registers which contain the appropriate control bits. Each of these registers will be identified and described in the following paragraphs.

Free running counter

The central element in the main timer system is a 16-bit free-running counter. The timer counter register (TCNT) is meant to be read using a double-byte read instruction such as load D or load X. This assures that the two bytes read from the TCNT register belong with each other.

7	6	5	4	3	2	1	0	TCNT
D15	D14	D13	D12	D11	D10	D9	D8	$000E
D7	D6	D5	D4	D3	D2	D1	D0	$000F

Figure 9.3 Timer counter register

A programmable prescaler allows the user to select one of four clocking rates to drive the 16-bit main timer counter. The fastest rate causes the main timer to clock at the E-clock rate, which results in a timer resolution of 500ns and a timer range of 32.77 ms between overflows (for E=2MHz). The slowest rate cause the main timer to clock at an E divided by 16 rate, which results in a timer resolution of 8 usec and a timer range of 524.3 ms between overflows (for E=2 MHz). You can see that this choice allows the programmer to make the classic trade-off between range and resolution.

The following register and paragraphs explain the prescaler select bits. There are two bits to consider, PR1 and PR0, in the TMSK2 register. Note that the other bits in this register are not related to the timer prescaler.

7	6	5	4	3	2	1	0	TMSK2
TOI	RTII	PAOVI	PAII	0	0	PR1	PR0	$0024

Figure 9.4 Timer Mask 2 Register

These two bits select the prescale rate for the main 16-bit free-running timer. Table 9.4 summarizes the programming options for controlling the prescaler. Note that in normal modes, this prescale factor cannot be changed "on the fly". The prescaler can only be changed once in the first 64 bus cycles after reset.

PR1	PR0	Prescale Factor	Resolution/Range
0	0	1	500ns/32.77ms
0	1	4	2us/131.1ms
1	0	8	4us/262.1ms
1	1	16	8us/524.3ms

Table 9.4 Resolution and Range of prescaler at E=2MHz

Real-time interrupt (RTI)

The real-time interrupt (RTI) function can be used to generate hardware interrupts at a fixed periodic rate. In order to accommodate a variety of applications, four different rates are available for the RTI signal. These rates are a function of the MCU oscillator frequency and the value of two software-accessible control bits (RTR1 and RTR0).

The clock source for the RTI function is a free running clock that cannot be stopped or interrupted. For this reason, the time between successive RTI time-outs will be a constant; the time will not depend on any software latencies.

Warning. The most common problem that the user encounters with the RTI system is that they forget to clear the RTIF (the interrupt flag) after it is recognized. Note that if the flag is not cleared by a specific software write to the TFLG2 register, it will already be pending the next time it is checked. This sequence results in an infinite loop.

The following registers and paragraphs explain the RTI control bits and the registers in which they can be found. There are four bits to consider. There are two bits available to program the RTI rate, one bit to enable the interrupt, and one bit to provide a flag indicating that the RTI has occurred.

7	6	5	4	3	2	1	0	TMSK2
TOI	RTII	PAOVI	PAII	0	0	PRI	PRQ	$0024

7	6	5	4	3	2	1	0	TFLG2
TOF	RTIF	PAOVF	PAIF	0	0	0	0	$0025

7	6	5	4	3	2	1	0	PACTL
DDRA7	PAEN	PAMOD	PEDGE	0	0	RTR1	RTR0	$0026

Figure 9.5 Real-time Interrupt Registers

First, consider the two bits that determine the rate at which interrupts will be requested by the RTI system. These bits are RTR1 and RTR0 from the PACTL register. Table 9.5 summarizes the programming options for controlling the prescaler.

RTR1	RTR0	(E/8192) divided by	RTI Rate
0	0	1	4.10 ms
0	1	2	8.19 ms
1	0	4	16.38ms
1	1	8	32.77ms

Table 9.5 Programming RTI rate at E=2MHz

Next, consider real-time interrupt enable bit, RTII, from the TMSK2 register. This bit allows the user to configure the RTI system for polled or interrupt-driven operation but does not affect the setting or clearing of the RTIF flag. When RTII is zero, interrupts are inhibited, and the RTI system is operating in a polled mode.

Finally, consider the real-time interrupt flag, RTIF, from the TFLG2 register. This status bit is automatically set to one at the end of every RTI period. The user can clear this bit by writing to the TFLG2 register with a '1' in the corresponding flag bit location (bit 6 in this case). Remember the warning that was given above regarding clearing the RTIF.

Input capture

Each input-capture function includes a 16-bit latch, input edge-detection logic, and interrupt generation logic. Each 16-bit latch captures the current value of the free-running counter when a selected edge is detected at the corresponding timer input pin. The software can select the edge polarity that will be recognized by programming appropriate control bits. The user can configure each of the three input-capture pins independently of the other two. The EDGxB and EDGxA pair of bits determine which edge(s) the input capture functions will be sensitive to. The three bit pairs are packed into the TCTL2 register as shown in Figure 9.6.

7	6	5	4	3	2	1	0	TCTL2
0	0	EDG1B	EDG1A	EDG2B	EDG2A	EDG3B	EDG3A	$0021

Figure 9.6 Timer Control Register 2

These bit pairs are encoded as shown in Table 9.6 for controlling the input capture configuration.

EDGxB	EDGxA	Configuration
0	0	capture disabled
0	1	capture on rising edges
1	0	capture on falling edges
1	1	capture on either edge

Table 9.6 Input Capture Configuration

The input-capture latches can be read by software as pairs of 8-bit registers. As long as double-byte read instructions such as load D (LDD) is used to read input-capture values, the programmer is assured that the two bytes belong with each other. The three pairs of 8-bit registers are shown in Figure 9.7.

7	6	5	4	3	2	1	0	T1C1
D15	D14	D13	D12	D11	D10	D9	D8	$0010
7	6	5	4	3	2	1	0	
D7	D6	D5	D4	D3	D2	D1	D0	$0011
7	6	5	4	3	2	1	0	T1C2
D15	D14	D13	D12	D11	D10	D9	D8	$0012
7	6	5	4	3	2	1	0	
D7	D6	D5	D4	D3	D2	D1	D0	$0013
7	6	5	4	3	2	1	0	T1C3
D15	D14	D13	D12	D11	D10	D9	D8	$0014
7	6	5	4	3	2	1	0	
D7	D6	D5	D4	D3	D2	D1	D0	$0015

Figure 9.7 Input Capture Registers

Output compares

Each of the five output-compare functions has a 16-bit compare register and a dedicated 16-bit comparator. When a match is detected between the free-running timer value and the 16-bit compare register, a status flag is set and an interrupt can be generated.

Output pins for the five output-compare functions can be used as general-purpose output pins or as timer outputs that are directly controlled by the timer system.

The 16-bit output-compare register for each output-compare function can be read or written by software as a pair of 8-bit registers. There are five pairs of 8-bit registers as shown in Figure 9.8.

7	6	5	4	3	2	1	0	TOC1
D15	D14	D13	D12	D11	D10	D9	D8	$0016
7	6	5	4	3	2	1	0	
D7	D6	D5	D4	D3	D2	D1	D0	$0017
7	6	5	4	3	2	1	0	TOC2
D15	D14	D13	D12	D11	D10	D9	D8	$0018
7	6	5	4	3	2	1	0	
D7	D6	D5	D4	D3	D2	D1	D0	$0019
7	6	5	4	3	2	1	0	TOC3
D15	D14	D13	D12	D11	D10	D9	D8	$001A
7	6	5	4	3	2	1	0	
D7	D6	D5	D4	D3	D2	D1	D0	$001B
7	6	5	4	3	2	1	0	TOC4
D15	D14	D13	D12	D11	D10	D9	D8	$001C
7	6	5	4	3	2	1	0	
D7	D6	D5	D4	D3	D2	D1	D0	$001D
7	6	5	4	3	2	1	0	TOC5
D15	D14	D13	D12	D11	D10	D9	D8	$001E
7	6	5	4	3	2	1	0	
D7	D6	D5	D4	D3	D2	D1	D0	$001F

Figure 9.8 Output Compare Registers

The software has further control over the operation of the five outputs. The following two registers contain the output-compare status flags and the local interrupt enable bits for the output-compare functions. There are five interrupt enable bits and five output control flags that are found in two registers as shown in Figure 9.9 and Figure 9.10 below.

7	6	5	4	3	2	1	0	TMSK1
OC1I	OC2I	OC3I	OC4I	OC5I	IC1I	IC2I	IC3I	$0022

Figure 9.9 Output-Compare Interrupt Enables

7	6	5	4	3	2	1	0	TFLG1
OC1F	OC2F	OC3F	OC4F	OC5F	IC1F	IC2F	IC3F	$0023

Figure 9.10 Output-Compare Flags

The OCxF status bit is automatically set to one each time the corresponding output-compare register matches the free-running timer. This status bit is by writing to the FLG1 register with a '1' in the corresponding data bit position. The OCxI control bit allows the user to configure each output-compare function for polled or interrupt-driven operation. When the OCxI bit is '0', the

corresponding output-compare interrupt is inhibited, and the output compare is operating in a polled mode.

If the OCxI control bit has been programmed to a '1' and a hardware interrupt request is generated, then the programmer must clear the OCxF bit by writing to the TFLG1 register before leaving the interrupt service routine.

Pulse Accumulator

The pulse accumulator is based on an 8-bit counter and can be configured to operate as a simple event counter or for gated time accumulation. Control bits allow the user to configure and control the pulse accumulator sub-system. Two maskable interrupts are associated with the system, each having its own controls and interrupt vector.

The pulse accumulator system is based on an 8-bit up-counter that can be read or written at any time. The pulse accumulator enable (PAEN) control bit enables/disables this 8-bit counter. The pulse accumulator mode (PAMOD) bit controls the mode of operation. In the event counting mode, the 8-bit counter is incremented by one at each active edge of the PAI pin. In the gated time accumulation mode, the 8-bit counter is clocked by the E clock divided by 64. This clock is gated by the PAI pin.

The input buffer on the PAI pin is always connected from the pin to the pulse accumulator and port A read logic, but the output buffer is enabled or disabled by the data direction control bit (DDRA7) in the pulse accumulator control (PACTL) register.

The following register and paragraphs describe the pulse accumulator related bits in the PACTL register.

7	6	5	4	3	2	1	0	PCNTL
DDRA7	PAEN	PAMOD	PEDGE	0	0	RTR1	RTR0	$0026

Figure 9.11 Pulse Accumulator Control Register

The 8-bit pulse accumulator counter, PACNT, is not affected by reset and can be read or written at any time. When the count in the PACNT register overflows from 255 to 0 an interrupt will be generated.

7	6	5	4	3	2	1	0	PACNT
D7	D6	D5	D4	D3	D2	D1	D0	$0027

Figure 9.12 Pulse Accumulator Count Register

Pulse accumulator mode

We can program the microcontroller for pulse accumulator mode and then select whether rising edges or falling edges on the PAI pin will be recognized. Furthermore, if we preload the PACNT register with the twos complement of our desired count, N, the counter will overflow after 255-N counts. This is an easy way to count pieces on an assembly line or cycles of an incoming signal. It is even possible to count more than 256 events.

Gated time accumulation mode

This mode changes the pulse accumulator into a timer. The 8-bit PACNT register is incremented every 64th E-clock cycle as long as the PAI pin is active.

A common use of the gated time accumulation mode is to measure the duration of single pulses. We can set the counter to zero before the pulse starts, and the resulting pulse time is directly read when the pulse is finished.

It is easy to measure times that are in the range of the 8-bit timer. However, it is also possible to measure times longer than the range of the 8-bit counter.

This completes our discussion of port A. We have identified the control and status registers that are associated with each of the main timer subsystems. We should now be able to configure the eight port A bits for general-purpose input/output. Alternatively we should be able to program the port A bits for gated time accumulation, if required.

Port B

The eight port B pins are fixed-direction output pins. However the functionality of these pins in shared between general-purpose outputs and high-order address outputs. There is no data direction register or special control register associated with port B. In single-chip mode, port B can be used for eight general-purpose output ports. In expanded mode, the MCU core takes control of port B and utilizes the eight bits for the higher order address outputs.

7	6	5	4	3	2	1	0	PORTB
D7	D6	D5	D4	D3	D2	D1	D0	$0004

Figure 9.13 Port B Data Register

Port C

Port C is also a complex port on the 'HC11E0 because it can act as general-purpose bidirectional I/O or as a time-multiplexed address/data bus port. It will be easier to explain port C operation if we break down the functionality into single-chip and expanded mode operation.

Single-Chip mode

In this mode, port C is a general-purpose, 8-bit, bidirectional I/O port. The selection of whether each pin is an input or an output is independently controlled by a corresponding bit in the data direction register. For port C this is DDRC.

7	6	5	4	3	2	1	0	PORTC
D7	D6	D5	D4	D3	D2	D1	D0	$0003

Figure 9.14 Port C Data Register

7	6	5	4	3	2	1	0	DDRC
DDRC7	DDRC6	DDRC5	DDRC4	DDRC3	DDRC2	DDRC1	DDRC0	$0007

Figure 9.15 Port C Data Direction Register

It is possible to program the outputs of port C for wired-OR operation. The CWOM (think wired-OR mode) control bit in the PIOC register (see below) allows the programmer to disable the P-channel driver of the complementary pair output buffer. The N-channel driver is not affected by CWOM. For this reason, if the user programs the CWOM bit to a logic '1' then all of the port C pins become open-drain outputs. Note that the CWOM bit effects all eight bits of port C. Port C can only be configured for wired-OR operation when the 'HC11 is in the single-chip mode of operation.

7	6	5	4	3	2	1	0	PIOC
0	0	CWOM	0	0	0	0	0	$0003

Figure 9.16 Parallel I/O Control Register

Expanded Mode operation

In this mode, port C handles both the 8-bit data bus and the lower 8 bits of the address bus. These two sets of 8 bits are time-multiplexed on port C and are active on opposite edges of the E-clock, so no compromise of throughput is necessary.

We will return to this topic of single-chip mode and expanded-mode later in this Lesson. At that time we will review the configuration of the EZ-Micro Tutor™ system architecture and determine several schemes that will allows to expand the hardware in order to support future Laboratories. But for now we must conclude our discussions of the four I/O ports.

Port D

Port D is the fourth parallel I/O port on the 'HC11. Port D supports 6 general-purpose, bidirectional I/O pins that can be independently configured either as inputs or outputs. This port shares its functionality between being general-purpose I/O and supporting two separate serial interface subsystems. The two serial interface subsystems are called the serial communications interface (SCI) and the serial peripheral interface (SPI).

When port D is configured for general-purpose I/O, the selection of whether each of the pins is an input or an output can be independently controlled by programming the corresponding bit in the data direction register. A '1' programmed into a particular data direction register makes the corresponding pin an output. Alternatively, a '0' programmed into a particular data direction register makes the corresponding pin an input. All I/O pins are inputs at power-on reset.

7	6	5	4	3	2	1	0	PORTD
0	0	D5	D4	D3	D2	D1	D0	$0008

Figure 9.17 Port C Data Direction Register

7	6	5	4	3	2	1	0	DDRD
0	0	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0	$0009

Figure 9.18 Port D Data Direction Register

When the SCI receiver is enabled, the PD0/RxD pin becomes an input dedicated to the RxD function. Likewise, when the SCI transmitter is enabled, the PD1/TxD pin becomes an input dedicated to the TxD function. We will have more to say about the SCI subsystem in Lesson Eight.

The SPI subsytem is a 4-wire system. Three of the pins are dedicated to the SPI functions when the SPI system is enabled. They are PD2/MISO, PD3/MOSI, and PD4/SCK. The fourth pin, PD5/SS* can be used as a general-purpose output by setting the corresponding DDRD5 bit as long as the SPI is configured for the master mode of operation. We will have more to say about the SPI subsystem in Lesson Eight.

The remaining two pins (bit 6 and bit 7) in port D are PD6/AS and PD7/RW*. Both of these pins are control lines used in the expanded operating mode.

We have now completed our review of the four I/O ports on the 'HC11D0. During this review, we discovered that each of the I/O ports shares its pins with more than one function. We also saw that the function of the port B pins and the port C pins depends on the operating mode of the 'HC11. The next step in our discussion on how to interface parallel I/O to the 'HC11E0 is understanding these operating modes.

MCU Operating Modes

The mode of operation for the 'HC11 upon power-up is determined by the states present on the two mode control pins, MODA and MODB. For our discussions here in the EZ-Micro Lessons, we will consider two "normal" modes of operation. The normal modes of operation are both selected by having a logic '1' on the MODB pin during reset. This leaves the MODA pin as the selector between the two modes that we are interested in:

MODA = 0 normal single chip mode
MODA = 1 normal expanded mode

In the next paragraphs we will discuss in more detail what is meant by single chip and expanded mode.

Single Chip Mode

Because the single chip mode does not require and external address and data bus functions, all of the I/O that are multiplexed with address and data functions become available for general-purpose parallel I/O. In this mode, all software needed to control the MCU is contained in the internal memories.

Expanded Mode

This mode of operation allows external memory and peripheral devices to be accessed by an external address and data bus. Now it is interesting to note that this is a time-multiplexed address/data bus. In order to save I/O pins, the low-order eight bits of the address are multiplexed with the data on the port C pins. In this manner only 18 pins are needed to support a 16-bit address bus, an 8-bit data bus and two control lines.

How are the data and lower address lines multiplexed? An external transparent latch such as the 74LS373 separates the two sets of 8 lines. This transparent latch is clocked by the address strobe signal (AS). In expanded mode, the R/W* line functions as the read/write bus control signal.

EZ-Micro Tutor™ Board

Refer to page 1 of the EZ-Micro PC board schematic; this schematic is back in Section 2 of Lesson 4, Figure 4.26. On this page the 68HC11E0 is designated as U1. The EZ-Micro Tutor™ Board is shipped to you with jumpers H3 and H4 not inserted. When the 'HC11 is powered on the 2-bit state that is present on the MODA and MODB pins is '11'; both inputs are high. The 'HC11 is therefore configured in the normal expanded mode of operation.

The high eight bits of the address bus are output on port B. The low eight bits of the address bus are multiplexed with the 8-bit data bus on port C. These multiplexed lines are separated by the transparent latch designated as U2. Note that the latch is directly clocked by the address strobe signal (AS).

In the EZ-Micro architecture, the 'HC11E0 is configured in normal expanded mode in order to take advantage of the available off-chip ROM and RAM. For this reason, port C is not available to us for general-purpose I/O.

However, we are free to add memory-mapped peripherals to the address space of the 'HC11E0 provided that we do not create contention with the ROM and RAM that is present on the EZ-Micro main board. The 16-bit address bus can access up to 64K of external address space. The factory default memory map for the EZ-Micro Tutor™ Board is shown in Figure 9.19 below.

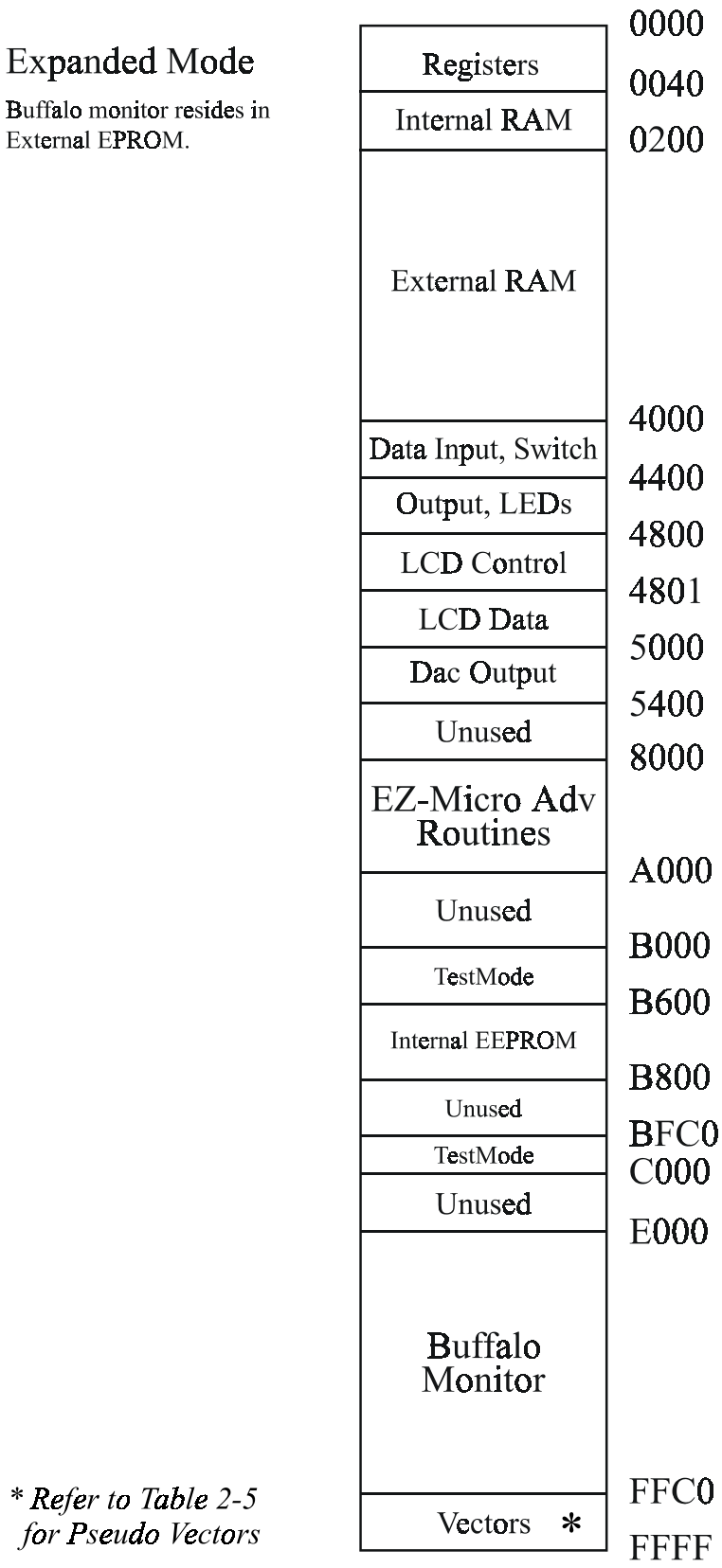


Figure 9.19 EZ-Micro Memory Map

Laboratory 9.1

Keypad

This Laboratory will interface device to the EZ-Micro Tutor™ Board. The input device will be a 4X4 keypad and the output device will be PC Monitor.

The keypad (K1) is comprised of 4 columns by 4 rows. Each contact is a single-pole/single-throw switch. The four rows are driven as outputs and are normally high; the four columns are inputs. Note that each of the row and column lines is pulled up through a resistor to +5V. The algorithm for making the determination of which key was pressed can be described as follows:

```
1. drive row 1 low
2. read column 1; if low then key is '1'
3. read column 2; if low then key is '2'
4. read column 3; if low then key is '3'
5  read column 4; if low then key is 'A'
6. return row 1 high; drive row 2 low
6. read column 1; if low then key is '4'
7.        ... and so on ...
```

This algorithm is called "scanning" the keypad and kepad connections are called 4X4 matrix.

The general-purpose I/O lines from port A are used to scan the keypad. The four input pins (PA0, PA1, PA2, PA3) are connected to the rows. The four output pins (PA4, PA5, PA6, P7) are connected to the columns. Our software can scan the keypad by writing and reading to and from port A.

Procedure

This Laboratory will involve writing a program which will input any key from the keypad and displaying on PC monitor.

Use the following guidelines as you develop your program:

1) Store your program variables in external RAM beginning at address 2000h.

2) Load your program to begin execution from external RAM beginning at address 2000h. Use the "ORG $2000" assembler directive to accomplish this task.

Testing the Keypad

To test Keypad circuit, first supply power to your EZ-Micro Tutor™ Board. Run the EZ-Micro Manager software and assemble the KEYPAD.ASM file that is on the program disk. After assembling the KEYPAD.ASM file, download the KEYPAD.S19 file to your EZ-Micro Tutor™ Board. Run the program KEYPAD by selecting the "GO" command from the RUN pull-down menu. Enter 2000 as the hex address of the first instruction to be executed.

As you press any key on the keyapd, PC monitor display will display the key that was pressed. Pree ESC to get out of this program

If the test described above did not work properly, step through the following procedure:

1) Make sure power is applied to the EZ-Micro Tutor™ Board. The power indicator LED (D3) should be on.

2) Make sure all connections between the Keypad/Display Lab Board and the EZ-Micro Tutor™ Board are correct and making a good connection.

3) Make sure that the KEYPAD.ASM file is properly assembled and the KEYPAD.S19 file has been properly created.

4) Make sure that the KEYPAD.S19 file is downloaded properly to The EZ-Micro Tutor™ Board. Download again if necessary.

Laboratory 9.2

LCD Display

The EZ Micro Tutor comes with 2 lines 16 characters LCD display. The LCD display is connected as I/O device to Micorcontroller. The I/O decoding is done through 22V10 GAL(U5). And address line A0.

There are three control lines that control the LCD display. These control lines are LCD_E, LCD_RW, and LCD_RSS. The first two control lines LCD_E and LCD_RW are controlled through U5 (GAL) and LCD_RSS is connected to Address line A0.

The IO addresses LCD display occupies are $4800(HEX) and $4801(HEX). The address $4800 is the control.

The following assembly program LCD.ASM is used to test LCD display. It writes number 0 on 1st line and character A on 2nd line.

LCD.ASM

```
*************************************************************************
* This Program (LCD.ASM) tests LCD Display                              *
* on EZ-MICRO TUTOR Board From Advanced Micorcomputer Systems Inc.      *
* The updated software can be downloaded from our website               *
* www.advancedmsinc.com                                                 *
* copyright 2000 by Advanced Micorcomputer Systems Inc.                 *
*************************************************************************
```

```
* Equates
CONTROL EQU  $4800            *RS low
DATA    EQU  $4C00            *RS high

* External references
BUFALO  EQU  $e000       *start of BUFFALO monitor
OUTPUT  EQU  $e3c2       *BUFFALO subroutine for OUTPUTing a character in A

        ORG  $2000
MAIN
        LDAA  #$3C            *Function set: data length=8, lines=2,
dots=5x10
        STAA  CONTROL
        JSR   DLY10MS

        LDAA  #$0F            *Display ON, cursor ON, blink On
        STAA  CONTROL
        JSR   DLY10MS

        LDAA  #$01            *Clear display
        STAA  CONTROL
        JSR   DLY10MS

        LDAA  #$06            *Entry mode set: increment, no display shift
        STAA  CONTROL
        JSR   DLY10MS

 * Let's begin at first character in first line, address = $80
        LDAA  #$80            *Set DDRAM address
        STAA  CONTROL
        JSR   DLY10MS

        LDAA  #'0'            *Write '0' to first character in first line
        STAA  DATA
        JSR   DLY10MS

* Now let's skip to first character in second line, address = $C0
        LDAA  #$C0            *Set DDRAM address
        STAA  CONTROL
        JSR   DLY10MS

        LDAA  #'A'            *Write 'a' to first character in second line
        STAA  DATA
        JSR   DLY10MS

* Delay subroutine
DLY10MS EQU  *             delay 10ms at E = 2MHz
        PSHX
        LDX  #$0D06
DLYLP   DEX
        BNE  DLYLP
        PULX
        RTS
```

Laboratory 9.3

Digital Input and Digital output

The EZ-Mico Tutor board has one external Digital input Port 74LS373 (U9). The four input lines are connected to Dip switch SW2. These individual switches simulate sensor switch or any other kind of ON-OFF switch.The other four input lines are connected to J4 connector.
The IO address of this Input port is $4000 Hex. The board has one external Digital output port 74LS374 U9. The four input lines are connected to four LEDS D1through D4.The other four output lines are connected to J4 connector. The IO address of the LED output port is $4400.
The following assembly program reads input port (DIP switch) and turn on resepective LED.

DIN.ASM

```
*************************************************************************
* This Program (DIN.ASM) tests Input port on EZ-MICRO TUTOR Board       *
* From Advanced Micorcomputer Systems Inc.                              *
* The updated software can be downloaded from our website               *
*                                                                       *
* www.advancedmsinc.com                                                 *
*                                                                       *
* copyright 2000 by Advanced Micorcomputer Systems Inc.                 *
*                                                                       *
*************************************************************************

* Digital input test

BUFALO  EQU  $e000  *start of BUFFALO
OUTPUT  EQU  $e3c2  *BUFFALO subroutine for OUTPUTing a character in A
IN      EQU  $4000  *memory mapped address of digital input latch
LEDS    EQU  $4400  *memory mapped address of leds

        ORG  $2000

MAIN

        LDAA  IN                *read switches
        NOP
        NOP
        STAA  LEDS              *write to leds
        NOP
        JMP  MAIN               *run until user presses reset
```

Laboratory 9.4

Measuring period of input signal using TCAP and concept of interrupts

This Laboratory is designed to provide experience with two major concepts: measuring the period of an input signal using the input capture time, and interrupts.

The input capture timer supports accurate timing of the period and/or duty cycle of input signals. As we learned in this Lesson, the 'HC11E0 can be configured for up to 3 input capture channels at a time. Since each channel functions independently, it is only necessary to study and understand the operation of one channel in order to understand them all.

All of the timer related functions of the 'HC11E0 microcontroller (input capture, output compare, pulse accumulator, and watchdog timer) are based on a 16-bit free-running counter. This counter simply increments once for each E clock (2 MHz on the EZ-Micro Tutor™ Board with the 8 MHz system clock) from $0000 to $FFFF.

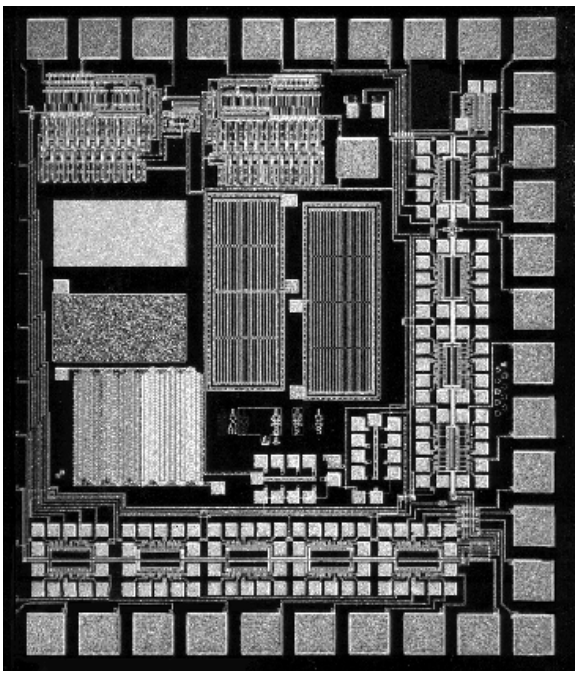
The input capture function is composed of a 16-bit register that can be loaded with the current counter value when an input signal active edge occurs. We learned in this Lesson that the input capture function is controlled by configuration registers that can be programmed to specify such parameters as the polarity of the active edge, status flags, and interrupt processing.

It is to our benefit to free the processor from the burden of polling flags to identify when an input capture has occurred. If a polling scheme were used, the processor would have little time to perform any other tasks that may be necessary for the system to function. Furthermore, the polling scheme may introduce unwanted latency because we may not "see" that the flag has been set immediately. Combining the timer input capture process with an interrupt control scheme will free the processor to perform other tasks during the time between the active edges of the input signal that triggers the input capture.

Interrupts are conceptually quite simple. Every time the phone rings, an interrupt occurs. Whatever is being done is temporarily suspended, the phone is answered and the call processed, then the suspended task is picked up from where it was left off. In fact this is quite analogous to processing an interrupt with the 'HC11E0 microcontroller. In order to set up an interrupt process, the following steps must be performed.

1) Create the Interrupt Service Routine (ISR) software that will execute when the interrupt is processed. Typically this involves reading or writing to the I/O system that generates the interrupt signal. As we were warned in the Lesson, the interrupt flag must be cleared during the processing of the ISR. This is to prevent the occurrence of false interrupts due to a single stimulus. The process of clearing the interrupt flag is dependent on the I/O system that is generating the interrupt. The ISR will normally terminate with a Return From Interrupt (RTI) instruction to return back to the interrupted process.

2) Install the address that points to the ISR in the vector location that is associated with the system that will generate the interrupt signal. Each source of interrupt has a unique pair of addresses that are used to direct the interrupt process to the entry point for the ISR.



Lesson Ten

Serial Input/Output

Transferring data between systems can be accomplished in a number of different ways. So far, the input/output operations that we discussed in Lesson Seven have dealt with parallel I/O, meaning the simultaneous transmission of a group of bits (a byte in our case). In this chapter we'll introduce serial I/O, meaning the transmission of bits one at a time.

A serial input port is a single wire over which serial data is input to the microcontroller from a peripheral device. A serial output port is a wire over which serial data is output from the microcontroller to a peripheral device. The major advantage of serial I/O is that it uses fewer wires than parallel I/O. Remember from Lesson Seven that increased pin count can have a direct impact on package cost as well as PC board cost. The major draw back of serial I/O is that is slower than parallel I/O. To send one ASCII character serially, we need to transmit a byte of data 1 bit at a time instead of all at once.

The most apparent application of serial I/O is in transmitting coded characters, such as ASCII. For this reason we will be discussing mostly serial transmission of ASCII characters in this Lesson.

One way we can transmit serial ASCII characters is asynchronously, meaning that indefinite periods of time may lapse between transmission of characters. There is no shared serial clock between the two devices. Each character must be unique and recognizable by itself.

Alternatively, we can transmit serial ASCII characters synchronously by taking the same amount of time to transmit each character with no time lapses in between characters. Synchronous transmissions require that a common external clock exist between the two devices that are communicating. Typically one of the devices functions as the master, providing the serial clock as an additional input to the slave device.

In this Lesson we will consider both asynchronous serial I/O and synchronous serial I/O. The HC11E0 supports both of these serial transmission schemes by multiplexing these functions on port D. When the serial communications interface (SCI) is enabled, the PD0/RxD pin becomes an input and the PD1/TxD pin becomes an output. These two wires comprise the asynchronous serial port. When the serial peripheral interface (SPI) system is enabled, the PD2/MISO, PD3/MOSI, PD4/SCK, and PD5/SS* pins become dedicated to SPI functions. These four wires comprise the synchronous serial port.

Serial Communications Interface (SCI)

Asynchronous serial data

A serial ASCII character may be represented as a series of high and low voltage levels that can be translated into 1's and 0's. Figure 10.1 shows the ASCII character A ($41) depicted graphically as highs and lows. The bits are transmitted with the LSB first so time increases from right to left in the figure.

7	6	5	4	3	2	1	0	
A	0	1	0	0	0	0	0	1

The transitions from '0' to '1' and from '1' to '0' seem easy to detect but how can the asynchronous receiver detect that there are five sequential zeros present in the middle of the data stream?

Bits transmitted in an asynchronous serial stream have to be defined in terms of bit times. We can choose from among a number of standard bit times for a particular application. The bit time depends directly on the number of bits per second, called the baud rate; the bit time is the reciprocal of the baud rate. Some common available baud rates and their corresponding bit times are shown in Table 10.1.

110	9.09 ms
150	6.67 ms
300	3.33 ms
600	1.67 ms
1200	0.83 ms
2400	0.417 ms
4800	0.208 ms
9600	0.104 ms
12000	0.083 ms
19200	0.052 ms

Table 10.1 Common Baud Rates and Bit Times

In order to define the beginning and end of an ASCII character, the data bits are framed with one start bit at the beginning of the character and one or more stop bits at the end of the character. When no characters are being transmitted, the data line is high, called a mark. The start bit is a '0', called a space. The stop bit or bits are a '1' and the line stays high if no further characters are transmitted.

The number of characters transmitted per second is called the character rate. To calculate the character rate from the baud rate, we need to know how many bits are in a character. A character with 1 stop bit contains 10 bits and a character with 2 stop bits contains 11 bits. You may have seen options for character formats on the setup screen of your modem application software for example.

Inside the 'HC11E0 there are registers for initializing and configuring the asynchronous serial communications interface (SCI). The fundamental clock running in the SCI module is the 16x receiver baud rate clock. This fast clock allows the SCI receiver hardware to oversample the incoming bitstream by a factor of 16. This 16x clock is derived from the crystal frequency and the baud rate, in turn, is derived from the 16x clock.

SCI registers

The user can program the SCI configuration by five registers: BAUD, SCCR1, SCCR2, SCSR, and SCDR. Each of these registers is discussed in detail in the following sections.

The baud rate control register (BAUD) is used to select the baud rate. Normally this register is written once during initialization to set the SCI baud rate. The baud rate is derived from the E clock and both the receiver and transmitter use the same baud rate.

7	6	5	4	3	2	1	0	BAUD
TCLR	0	SCP1	SCP0	RCKB	SCR2	SCR1	SCR0	$002B

Figure 10.1 BAUD Register

Two bits in the BAUD register select a prescale factor for the baud generator: SCP1 and SCP0. Table 10.2 shows the highest baud rates that result for the four prescaler values at the EZ-Micro crystal frequency of 4 MHz.

SCP1	SCP0	Division Factor	XTAL=4MHz	comment
0	0	1	62.50K Baud	not practical
0	1	3	20.83K Baud	not practical
1	0	4	15.62K Baud	not practical
1	1	13	4800 Baud	very common

Table 10.2 Baud Rate Prescaler Selects

In addition to programming the SCPx bits, the three SCRx bits select a binary submultiple of the highest baud rate. The result of these two subdividers working in series is the 16x receiver baud rate clock. Table 10.3 shows the SCI baud rates that result for various settings of the SCRx bits and the 4800 baud rate from Table 10.2.

SCR2	SCR1	SCR0	Division Factor	XTAL=4MHz	comment
0	0	0	1	4800 Baud	common
0	0	1	2	2400 Baud	very common
0	1	0	4	1200 Baud	very common
0	1	1	8	600 Baud	not practical
1	0	0	16	300 Baud	slow but common
1	0	1	32	150 Baud	not practical
1	1	0	64	75 Baud	not practical
1	1	1	128	37 Baud	not practical

Table 10.3 Baud Rate Selects

The serial transmission can be configured for either 8-bit or 9-bit data characters. The 9-bit data format is most commonly used for an extra stop bit. The SCCR1 register contains the control bits related to programming the data format.

7	6	5	4	3	2	1	0	SCCR1
R8	T8	0	M	WAKE	0	0	0	$002C

Figure 10.2 SCI Control Register 1

The M bit controls the character length for both the transmitter and receiver at the same time. Programming a '0' into this bit selects one start bit, eight data bits, and one stop bit. Programming a '1' into this bit selects one start bit, nine data bits, and one stop bit. In the Laboratories associated with the EZ-MICRO TUTOR™ Boards, we will always be operating the SCI in the eight data bit mode, which is the default coming out of reset. For this reason, we will not offer a detailed discussion regarding the 9-bit data format. For more information about this mode, the interested reader should refer to the M68HC11 Reference Manual.

The main control register for the SCI subsystem is SCCR2; this register is shown in Figure 10.3 below. All eight bits in this register are important control bits and we need to understand the operation of each one.

7	6	5	4	3	2	1	0	SCCR2
TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	$002D

Figure 10.3 SCI Control Register 2

The reset state of the SBK bit (Send Break) is '0', which enables normal transmitter operation. Programming this bit to '1' enables the transmitter to send synchronous break characters. A break character holds the transmit line at logic zero.

Similarly, the reset state of the RWU (Rx Wake Up) bit is '0', which enables normal receiver operation. Programming this bit to '1' causes the receiver to go to sleep until the start of the next message. Your software can set the RWU bit after deciding that a particular SCI message is of no interest. The receiver wake up logic automatically recognizes when the message of no interest is complete and clears the RWU bit to wake up the receiver.

The RE bit is the receiver enable bit. When a '1' is programmed into this bit the SCI receiver is enabled; when a '0' is programmed into this bit the SCI receiver is disabled.

The TE bit is the transmitter enable bit. When a '1' is programmed into this bit the SCI transmitter is enabled; when a '0' is programmed into this bit the SCI receiver is disabled. Note that the transmitter cannot be turned off in the middle of a character. The transmitter keeps control of the TxD pin until any character in progress is finished.

ILIE represents the Idle-Line Interrupt enable bit. Programming this bit to a '1' allows the hardware to generate an SCI interrupt request when IDLE is set to one. Programming this bit to a

'0' disables IDLE interrupts. Software will have to poll the IDLE flag. We will discuss the IDLE flag in the next section when we consider the SCI status register.

The RIE bit controls Receive Interrupt Enables. When this bit is a '0' rx interrupts are disabled. When this bit is a '1' an SCI interrupt is requested when either RDRF or OR is set to one. These status bits are two of the flags contained in the SCI status register.

TCIE is the Transmit Complete Interrupt Enable bit. Similar to the bits above, programming this bit to a '0' will disable the TC interrupt. Programming this bit to a '1' will enable the TC interrupt and the hardware will request an SCI interrupt whenever TC is set to one. When we discuss the SCI status register you will see the TC flag.

The TIE bit is the transmit interrupt enable bit. When a '1' is programmed into this bit, TDRE interrupts are enabled; when a '0' is programmed into this bit, TDRE interrupts are disabled. The TDRE is the normal indication from the SCI hardware that a new character may now be written to the SCDR.

Having discussed the main control register, we will now discuss the status register. The seven bits associated with the SCI subsystem are located in the SCSR, which is shown in Figure 10.4 below.

7	6	5	4	3	2	1	0	SCCR2
TDRE	TC	RDRF	IDLE	OR	NF	FE	0	$002E

Figure 10.4 SCI Status Register

The function of these seven bits can be grouped into two categories. Some of these bits generate hardware interrupt requests; others indicate errors in the reception of a character. All of the status bits are automatically set by the corresponding conditions having been met in the SCI logic. It is important to note that once set, these bits remain set until software completes a clearing sequence. For example, to clear the TDRE flag, software must read the SCSR while the TDRE bit is set, and then write to the TDR. This process does not involve any additional instructions because these are exactly the normal steps in response to the TDRE anyway.

The Framing Error bit (FE) will be set by the SCI subsystem hardware when a framing error was detected for the character in the SCDR. The definition of a framing error is when the stop bit is not detected in the last bit time.

NF is the Noise Flag bit. This bit will be set to a '1' when the data recovery logic inside the SCI hardware detected noise during reception of the character in the SCDR. Noise is indicated during the data bit times if the three samples taken near the middle of the bit time do not unanimously agree. Like many systems, we will ignore the NF bit and rely on the fact that the data recovery logic has already made a good first-order attempt to correct the problem.

When the Idle_line Detect bit (IDLE) is set to a '1', the TxD line has become idle. The idle condition is defined as at least a full character time of logic one on the Rxd line. For our case, a character time is 10 bit times (M=0).

The Receive Data Register Full bit (RDRF) will be set to '1' as the normal indication that a character has been received by the SCI. Specifically, a character has been received and has transferred from the receive shift register to the parallel SCDR where the software can read it. Note that the NF and FE status bits will provide additional information about the newly received character in the SCDR. Furthermore, the OR flag would indicate that another character was serially received and was ready to be transferred from the shift register to the parallel SCDR, but the previously received character was not read yet.

TC is the Transmit Complete bit. When this bit is set to a '1' the transmitter has completed sending and has reached an idle state. The TDRE bit indicates that the last character has transferred from the SCDR to the transmit shift register. The TC bit indicates that the last bit has been shifted out.

The Transmit Data Register Empty bit (TDRE) will be set to '1' as an indication that a new character may now be written to the SCDR. In normal transmit operation, this bit will be checked by your software to see if the SCDR can accept new data from you. The SCI is double-buffered. This means that the TDR holds the second character in line while the transmit serial shift register sends out the next character.

The SCI Data Register (SCDR) shown in Figure 10.5 below is actually two separate registers. When the SCDR is read, the read-only RDR is accessed; when the SCDR is written, the write-only TDR is accessed. This register location ($002F) may be referred to by any of the mnemonics SCDR, TDR or RDR.

SCDR

7	6	5	4	3	2	1	0	$002F
R7	R6	R5	R4	R3	R2	R1	R0	RDR (read)
T7	T6	T5	T4	T3	T2	T1	T0	TDR (write)

Figure 10.5 SCI Data Register

We stated in the section above that the SCI receiver is double buffered. This means that the receiver can be processing up to two characters at any given time. One of the characters is stored in the readable parallel receive data buffer (RDR) and the other is in the process of shifting into the receiver serial shift register. Without double-buffering the transmitting device on the other end would be required to insert delays between transmitted characters to avoid a receiver overrun. Note that an overrun can still occur in a double-buffered SCI receiver if a serial character is received and is ready to transfer into the parallel RDR before the previously received character is read out.

Summary of Normal Operation.

In the EZ-MICRO TUTOR™ Laboratories, how will we be configuring the SCI? In general, we will be programming the EZ-MICRO TUTOR™ SCI port for "normal" operation. Many of the special operating modes and corresponding status flags will not be used. Because of this fact, we can provide a summary of normal operation in this section.

Baud Rate Register

As we saw in Table 10.2 and Table 10.3, our choices for practical baud rates are rather limited. With the 4 MHz crystal that is on the EZ-MICRO TUTOR™ Board, we can generate 2400 baud, 1200 baud or 300 baud. For 2400 baud operation, we should program SCP1 = '1', SCP0 = '1', SCR2 = '0', SCR1 = '0', and SCR0 = '1'. Load the BAUD register ($002B) with $31.

SCI Control Register 1

The M bit in the SCCR1 determines the length of SCI characters for both the transmitter and receiver. The most common configuration is one start bit, eight data bits, and one stop bit, which is selected by M = '0'. We will be using this configuration with the EZ-Micro Board. A value of $00 should be in the SCCR1 register; this is the default value at reset.

Control Register 2 - Receiver control.

The first thing we have to do is program a '1' into the RE bit; this will enable the SCI receiver. We won't enable the receiver wake-up feature so we will program a '0' into the RWU bit. When a character is received and transferred into the parallel RDR, we want to get a RDRF interrupt. For this reason, we will program a '1' into the RIE bit in the SCCR2 register. Lastly, we will disable IDLE interrupts by programming a '0' into the ILIE bit.

Control Register 2 - Transmitter control

The first thing we have to do is program a '1' into the TE bit; this will enable the SCI receiver. We won't enable the transmitter to send synchronous break characters so we will program a '0' into the SBK bit. When the character previously written to the SCDR has been transferred to the transmit shift register, we don't want to get a TDRE interrupt; we will have the software poll for this bit. For this reason, we will program a '1' into the TIE bit. We will disable TC interrupts by programming a '0' into to the TCIE bit.

Reviewing the previous two paragraphs, we see that programming a value of $2C into the SCCR2 will configure the SCI for what we will call our normal mode of operation on the EZ-MICRO TUTOR™ Board.

Status Register

Out of the seven status bits associated with the SCI in the SCSR2 register, we will only be concerned with two of these bits in normal operation. The TDRE bit will be set by the SCI hardware to indicate that a new character may now be written to the SCDR. When the TDRE bit is set, an interrupt will not be generated; out software will poll this bit. The RDRF bit will be set to indicate that a character has been received and has transferred from the receive shift register to the parallel SCDR where software can read it. When the RDRF is set, an interrupt request will be generated by the hardware. We will have to write an interrupt service routine to handle this serial interrupt.

Synchronous Serial Peripheral Interface (SPI)

The serial peripheral interface (SPI) is the second of two independent serial communications subsystems included on the 'HC11E0. During an SPI transfer, data is simultaneously shifted out serially and shifted in serially. A dedicated serial clock line synchronizes shifting and sampling of the information on the two serial data lines. There will typically be one device that acts as master on the SPI bus. All other devices will act as slaves. A slave select line allows individual selection of a slave SPI device; slave devices that are not selected do not interfere with SPI bus activities. The master device will drive the slave select lines and the serial clock line.

SPI Transfer

As we will see later, there are two control bits that can be programmed to select one of four SPI transfer formats. Figure 10.6 is a timing diagram of one particular case. (CPHA = '1' and CPOL = '0').

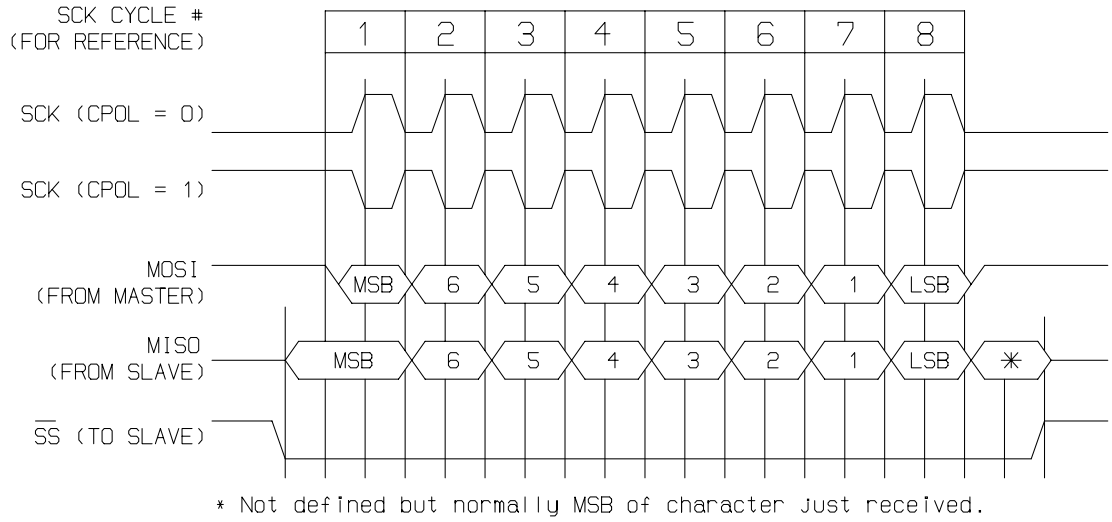


Figure 10.6. Typical SPI Transfer Format

SPI Registers

Inside the 'HC11E0 there are registers for initializing and configuring the serial peripheral interface (SPI). The user can program the SPI configuration by three registers: SPCR, SPSR and SPDR. Each of these registers is discussed in detail in the following sections. Also because the port D data direction control register influence SPI activities, it will be discussed briefly. A detailed discussion about the port D pins can be found in Lesson Seven.

The main control register for the SPI system is the SPI Control Register (SPCR). Your software can read or write this register at any time. This register is shown in Figure 10.7 below.

7	6	5	4	3	2	1	0	SPCR
SPIE	SPE	DWOM	MSTR	CPOL	CPHA	SPR1	SPR0	$0028

Figure 10.7 SPI Control Register

The two bits, SPR0 and SPR1, control the bit rate for transfers when the SPI is operating as a master. Note that when the SPI is operating as a slave, the serial clock is an input form the master. In this mode, SPR0 and SPR1 have no control. Figure 10.8 shows the bit rates that can be programmed by SPR0 and SPR1. The SPI clock rate is derived from a programmable divider off E clock.

SPR1	SPR0	Division Factor	XTAL=4MHz
0	0	2	500K bits/second
0	1	4	250K bits/second
1	0	16	62.5K bits/second
1	1	32	31.25K bits/second

Figure 10.8 SPI Bit Rate Selects

The CPHA bit controls the selection of one of two, fundamentally different transfer formats.

The CPOL bit controls clock polarity. When we program a '1' into this bit, active low clocks are selected and SCK will idle in the low state. When we program a '0' into this bit, active high clocks are selected and SCK will idle in the high state.

MSTR is the Master/Slave Mode Select bit. Programming a '1' into this bit will configure the SPI as master; programming a '0' into this bit will configure the SPI as slave.

DWOM is not a SPI control bit.

Programming the SPE bit to a '1' will enable the SPI system. Programming the SPE bit to a '0' will disable the SPI system.

Finally, the SPIE bit is the SPI Interrupt Enable control bit. If this bit contains a '1' then an SPI interrupt will be requested at the end of a SPI transfer. If this bit contains a '0' then and interrupt will not be requested. The software can poll the SPIF flag in order to detect the end of a SPI transfer.

The read-only SPI Status Register (SPSR) contains status flags indicating the completion of a SPI transfer and the occurrence of certain SPI errors. These flags are set automatically by the SPI subsystem hardware. Refer to Figure 10.9 for the location of status bits within the register.

7	6	5	4	3	2	1	0	SPSR
SPIF	WCOL	0	MODF	0	0	0	0	$0029

Figure 10.9 SPI Status Register

The error flags, WCOL and MODF, are automatically set if the hardware detects an error condition during the SPI transmission. Write collision errors and mode faults are beyond the scope of this Lesson. For more information about these faults, the interested reader should refer to the M68HC11 Reference Manual.

The SPI Transfer Complete Flag (SPIF) is automatically set to a '1' at the end of a SPI transfer. Generally speaking, SPIF is set at the end of the eighth SCK cycle. Depending on the particular mode of operation and whether the device is acting as master or slave, will determine the exact time that SPIF will be set.

The port D Data Direction Register (DDRD) was discussed in Lesson Seven. The description of the DDRD in this section is only intended to cover material related to the SPI system. Refer to Figure 10.10 below as each bit is described in detail.

7	6	5	4	3	2	1	0	DDRD
0	0	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0	$0009
SS*	SCK	MOSI	MISO	TxD	RxD			

Figure 10.10 Port D Data Direction Register

When the SPI system is enabled as a slave (SPE=1; MSTR=0), the PD5/SS* pin is the slave select input. Conversely when the SPI system is enabled as a master (SPE=1; MSTR=1), the function of the PD5/SS* pin now depends on the value in DDRD5. When a '0' is programmed in DDRD5, the SS* pin is used as an input to detect mode-fault errors. As we indicated earlier, mode-fault errors are beyond the scope of this Lesson. When a '1' is programmed in DDRD5, the PD5/SS* pin now acts a general-purpose output that is not affected by the SPI system. Many times in typical applications, the PD5/SS* pin is connected to the SS* input pin of a slave device.

DDRD4 is the Data Direction Control for bit 4 of port D. It shares functionality with SCK. When the SPI system is enabled as a slave, the PD4/SCK pin acts as the SPI serial clock input. When the SPI system is enabled as a master, the DDRD4 bit must be set to '1' to enable the SCK output.

The Data Direction Control bit DDRD3 must be set to '1' to enable the master serial data output when the SPI system is enabled as a master. When the SPI system is enabled as a slave, the PD3/MOSI pin acts as the slave serial data input, regardless of the state of DDRD3.

The final bit in the port D Data Direction Control Register is DDRD2. This bit must be set to '1' to enable the slave serial data output when the SPI system is enabled as a slave. When the SPI system is enabled as a master, the PD2/MISO pin acts as the master serial data input, regardless of the state of DDRD2.

Summary of Normal Operation

In the EZ-MICRO TUTOR™ Laboratories, how will we be configuring the SPI? In general, we will be programming the EZ-MICRO TUTOR™ SPI port for "normal" operation. Many of the special operating modes and corresponding status flags will not be used. Because of this fact, we can provide a summary of normal operation in this section. Note that the SPI master and the SPI slave will be configured differently.

SPCR

Configuring one device as slave master will require that set the MSTR bit in the SPI control register. In addition to setting this bit, we will select CPOL = '0' and CPHA = '1'. The selection of the SPI bit rate is arbitrary as long as the SPI master and SPI slave are configured to operate at the same rate. For the EZ-Micro Laboratories in this Lesson, we will select a divide by 4 configuration. Furthermore, we will enable SPI interrupts by setting the SPIE bit. Based on this discussion, we should program a value of $D5 into the control register on the master and a value of $C5 into the control register on the slave side.

SPSR

When our software reads the SPI status register, we will only be concerned with the SPIF bit, a flag indicating the completion of an SPI transfer.

DDRD

The SPI master will be driving SCK and SS* as outputs. For this reason, we will program $38 into DDRD (address $0009) on the master device. Conversely, the SPI slave will be receiving SCK and SS* as inputs. For this reason, we will program $04 into DDRD on the slave device.

SPDR

We can read and write to the SPI data register at address $002A. This register is double-buffered in and single-buffered out.

Questions

Laboratory 10.1

Procedure

This Laboratory involves interfacing two EZ-MICRO TUTOR™ Main Boards using the built-in SCI ports. The hardware interconnections between the two EZ-MICRO TUTOR™ Main Boards are detailed below in Figure 10.11. You will be implementing a three-wire bidirectional interface scheme. The required signals are TxD, RxD and Ground. Note that the TxD signal from one system connects to the RxD signal from the other.

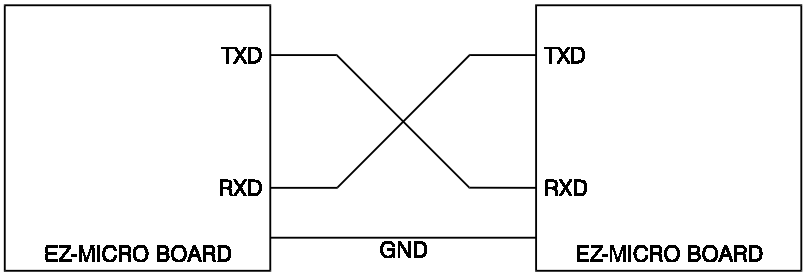


Figure 10.11 SCI Interface

No handshaking of any kind is implemented for simplicity purposes. Additional lines for handshake support to address overrun errors may be typically implemented in some applications. For example, in an RS-232 interface between your computer and your external modem, two handshake signals are defined — RTS* (Request to Send) and CTS* (Clear to Send). These lines are used to determine when characters can be transferred without overrun problems. Other schemes may use additional signal.

In this experiment, the receiver will be capable of accepting characters without an overrun error. This is a realistic assumption because our software will be dedicated to servicing the serial port.

Similar to Laboratory 9_1, you will be using the Keypad/Display Lab Board. System 'A' will send a character ('%') to system 'B' which will interpret this character as a "request for data" (RFD) code. When system 'B' receives this code, it will transmit the number "123" to system 'A' which will then display that number on the LED displays using the display driver routine from Laboratory 9_2.

A good scheme for testing the operation of the system is to use your PC to emulate each of the systems for the other one before connecting them together for final system integration. Thus, the PC could emulate the master module and send the request character (type the '%' character) and could display the number "123" to test the slave module. The PC could emulate the slave module and monitor the output of the master module to look for the request character (%), then respond with the typed number "123". This scheme allows testing of one module at a time to ease the integration process.

Because the sequence of operations is critical to the successful interfacing of the two EZ-Micro Boards, the pseudo-code for system 'A' and system 'B' processors is provided at the end of this Laboratory.

Laboratory Report

The laboratory report for this experiment will involve planning a handshaking scheme for your serial interface. This handshaking scheme can involve software/hardware or software only schemes. Your handshaking scheme does not need to be implemented for this report.

The report should consist of an introduction, description of the proposed handshaking scheme and documentation of the proposed changes to the existing system. The documentation could consist of schematic diagrams of the interface connections, detailed pseudo code or assembly language listings of the handshake routines, along with timing diagrams to indicate the sequence of the handshake process.

Laboratory 10.2

Procedure

This Laboratory involves interfacing two EZ-Micro Main Boards using the built-in SPI ports. The hardware interconnections between the two EZ-Micro Main Boards are detailed below in Figure 10.12. You will be implementing a four-wire bidirectional interface scheme. The required signals are MISO, MOSI, SS* and Ground. In the figure below, system 'A' is the SPI master and system 'B' is the SPI slave. Note that MOSI signal from one system connects to the MISO signal from the other. Furthermore, *note* that the SPI master is using PD5/SS* as a general-purpose output, whereas the SPI slave is using PD5/SS* and the slave select input.

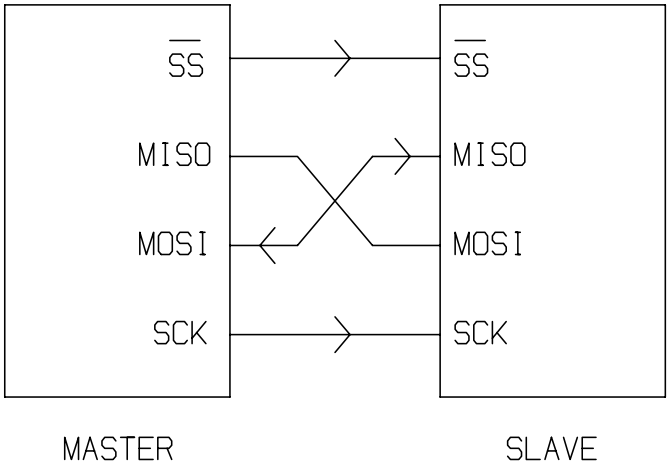


Figure 10.12 SPI Interface

Again, similar to Laboratory 9_1, you will be using the Keypad/Display Lab Board. The master will send a character ('%') to the slave which will interpret this character as a "request for data" (RFD) code. When the slave receives this code, it will write the number "1" to the SPI data

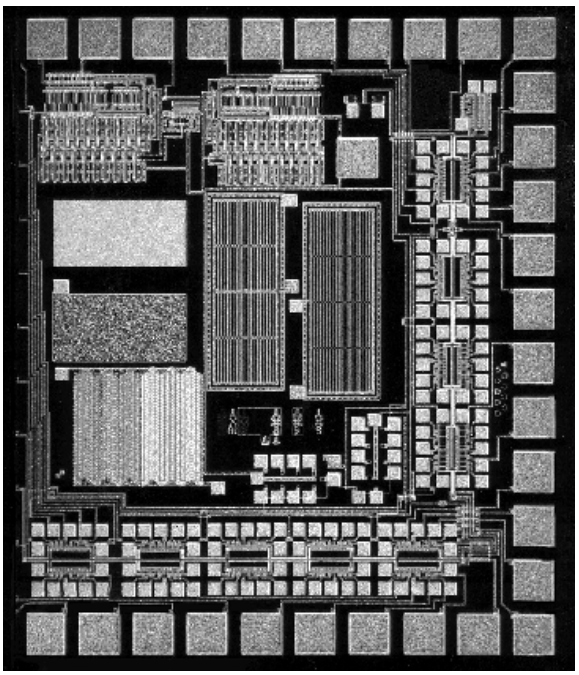
register and wait. Note that the slave has no means of telling the host that the data is ready. After a delay, the master will send another character ('%') to the slave. The slave data will be automatically returned to the master, which will then display that number on the LED displays using the display driver routine from Laboratory 9_2.

Because the sequence of operations is critical to the successful interfacing of the two EZ-Micro Boards, the pseudo-code for the master and slave processors is provided at the end of this Laboratory.

Laboratory Report

This laboratory report will involve planning a handshaking scheme for the SPI interface. As it currently stands, our interface is rather one-sided. The master can assert SS* at any time and transfer data to the slave. At that time, the slave has the opportunity to transfer data to the master. However, our interface does not provide the slave device with a means of telling the master that it has data ready to send. Your new handshaking scheme can involve both hardware and software. This scheme does not have to be implemented for this report. You only have to describe your idea.

The report should consist of an introduction, description of the proposed scheme, and documentation of the proposed changes to the existing system. The documentation could consist of schematic diagrams of the interface connections, detailed pseudo code or assembly language listings of the handshake routine, and timing diagrams to indicate the sequence of the handshake process.



Lesson Eleven

Analog to Digital and Digital to Analog Conversion

Despite the explosion of digital electronics, we still live in an analog world. For this reason, we must be familiar with the process of converting analog signals to digital words and the process of converting digital words to analog signals.

As engineers, we must make tradeoffs between cost and performance. How fast should we sample the input signal? How many bits of resolution do we need? What types of errors can our system tolerate? These are the types of questions that the material presented in this Lesson will help to answer.

D/A Conversion

Many systems accept a digital word as an input signal and translate it or convert it to an analog voltage or current. These systems are called digital-to-analog converters. The resolution of the D/A converter defines the smallest change of output voltage as a fraction of full scale. For the case of an 8-bit D/A converter (like the one on the EZ-Micro A/D-D/A Lab Board), the output can increase in steps of 19.53 mV up to a maximum of 256 steps. Each step is called a quantum.

The conversion time allows us to calculate the speed of conversion of a D/A converter and thereby obtain the maximum conversion frequency or the maximum number of conversions that can be carried out per second. The conversion time is the time required for the output to reach the desired value within the specified error. It is dependent on the components used and the worst case is when all the bits change simultaneously from 0 to 1.

The last important characteristic of a D/A converter is its accuracy. It is defined as the difference between the output signal obtained and the calculated theoretical value. The accuracy is usually expressed as a percentage of full-scale or as a fraction of the quantum and rarely in absolute units of mV or uV. The accuracy quoted by manufacturers is usually +/- 1/2 quantum. All bits are involved in the definition of the accuracy but their effect varies according to their weight. To understand this, consider that the MSB controls a corresponding voltage equal to one half the full scale voltage.

Types of DACs

The D/A conversion process is quite straightforward and is easier to understand than the A/D conversion. In fact, many of the A/D conversion techniques incorporate a DAC as an essential step. The examples which follow will generally make use of a 4-bit digital word. This is only a matter of convenience; the concepts can readily be extended to 8-bits and beyond.

Weighted Resistor

A weighted resistor network is one method of achieving the digital-to-analog conversion. Such a circuit is given in Figure 11.1, where VA is the analog voltage output and RL is selected to be much larger than 8R. The weighting of the resistor ladder is that of binary coded decimal (8, 4, 2, 1), with the most significant bit (MSB) resistance being one-eighth that of the LSB.

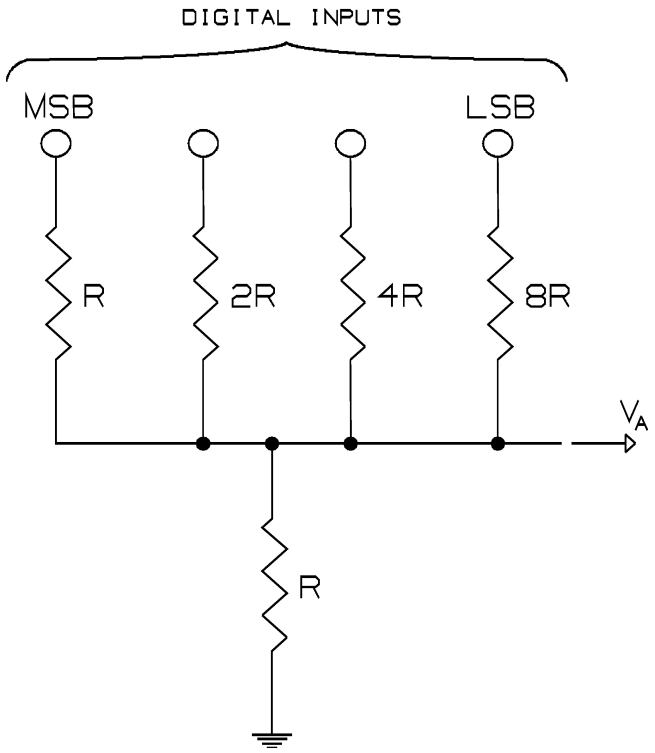


Figure 11.1 Weighted Resistor Network

Each of the digital inputs will have a voltage corresponding to a '1' or a '0' in whatever logic line is being converted. The output voltage, VA, can be determined for the weighted resistor ladder and the general expression can be reduced to

$$VA = (8Vm + 4Vnm + 2Vnl + VL)/15$$

where Vm is the voltage condition of the MSB, Vnm is that of the next most significant bit, Vnl is that of the next least significant bit, and Vl is that of the LSB. Compute the values for various binary inputs with V1 = 3.0 V or 0.0 V. Does the output behave as you expected?

The accuracy and stability of the DAC in Figure 11.1 depend primarily on the absolute accuracy of the resistors how the resistors track with each other over temperature. The weighted resistor ladder suffers in implementation from the fact that a wide range of resistor values is needed. For example, for a 10-bit DAC the largest resistance is 51.2 Mohm if the smallest is 10 kohm. The ladder-type converter described in the following avoids these difficulties.

Binary Ladder-type

The binary ladder network largely overcomes the problems of the weighted resistor network. The binary ladder D/A circuit is shown in Figure 11.2. Only two resistor values are needed. We observe from the figure that at any of the ladder nodes the resistance is 2R looking to the left or the right or toward the switch.

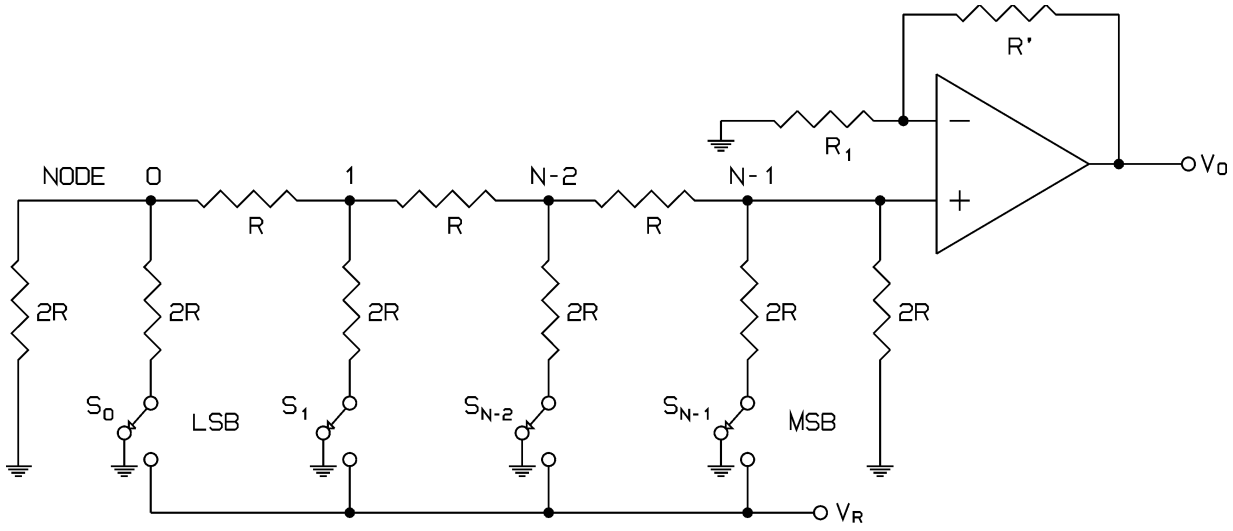


Figure 11.2 Ladder-type D/A Converter

For example, to the left of node 0 there is 2R to ground; to the left of node 1 there is a parallel combination of two 2R resistors to ground in series with R, for a total resistance of 2R, and so forth. Therefore, if any switch, take for example N-2, is connected to VR, the resistance seen by VR is 2R + R -= 3R and the voltage at node N-2 is (VR/3R)R = VR/3.

Multiplying D/A Converter

The output voltage from a D/A converter is the product of a digital word and an analog reference voltage. So, in a sense, every DAC is a multiplying DAC. For our purposes, in a multiplying DAC the reference voltage or the digital input code can change the output voltage polarity. The reference voltage can be either positive or negative.

Input Codes

Frequently used input codes for D/A converters are sign-magnitude, offset-binary and two's-complement bipolar codes. These codes are graphically described in Figure 11.3 for an 8-bit example. Sign magnitude is especially useful for outputs that are in the vicinity of zero. The most significant bit represents the sign, '0' means positive. The remaining seven bits represent the magnitude in the range from 0 to 127. Offset binary is the easiest to implement in the converter hardware. However, there is a major bit transition in going from just below zero up to zero. Note that one level below zero is represented as 01111111 and zero is represented as 00000000; seven bits have to flip. Two's complement is more computer compatible. This code consists of a binary number for the positive magnitudes and the two's complement of each positive code for the negative magnitudes.

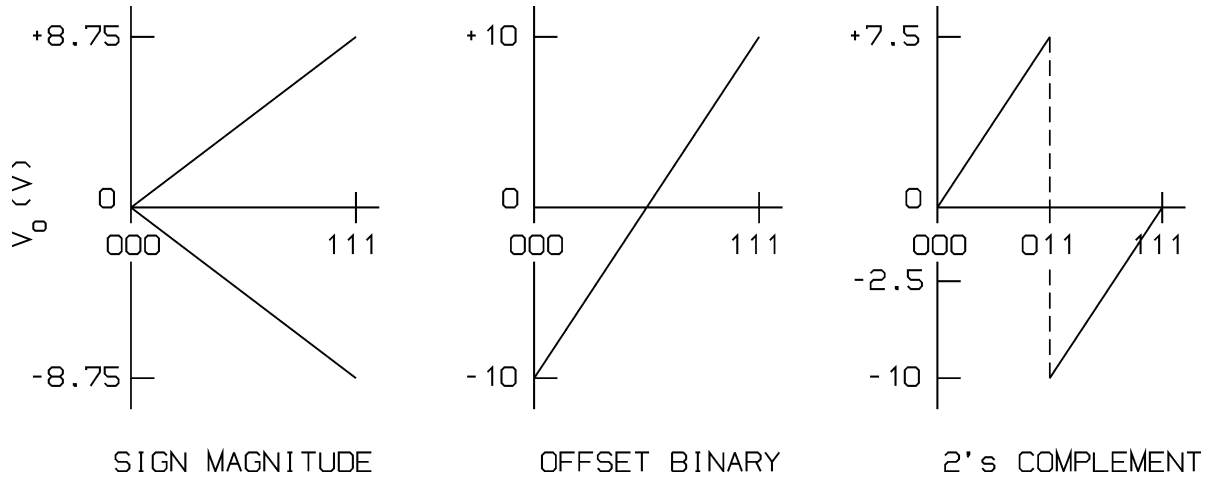


Figure 11.3 DAC Input Codes (8-bit example)

In your software that you develop in the Labs, use offset-binary input code when you write to the DAC08032 on the EZ-Micro A/D-D/A Lab Board. With the reference voltage set to 6.0 V, the relationship between digital input code out observed output voltage is given by:

$$V_{out} = 6.0\,V \frac{(\text{digital code} - 128)}{128}$$

Note that offset binary is readily converted to two's-complement by simply complementing the MSB.

A/D Conversion

Analog to digital conversion is the process of transforming a continuous signal into a discrete time representation. This process has been describe as the process of converting real world signals to a form that used in the electrical world of digital systems. Extensive theoretical and mathematical studies have been performed analyzed the A/D process as well as the D/A process. The introduction provided in this section will try to identify the key concepts involved with analog-to-digital conversion.

Consider the continuous time analog function f(t) shown in Figure 11.4. The amplitude of this signal varies between 0 and 1 over the period shown from t0 to tp. To determine the amplitude of this signal at time ts, it could be approximated by the representation "0.5" which would be reasonably close. On the other hand, it may be more accurate to say "0.53" or even "0.5268345500284384". In fact, the absolute value of f(t) at time ts might go on forever! Deciding how many places of accurateness are necessary to represent f(t) at ts is the process of discrete representation. Analog-to-digital conversion is the process of determining a number that closely approximates an input signal at a particular point in time.

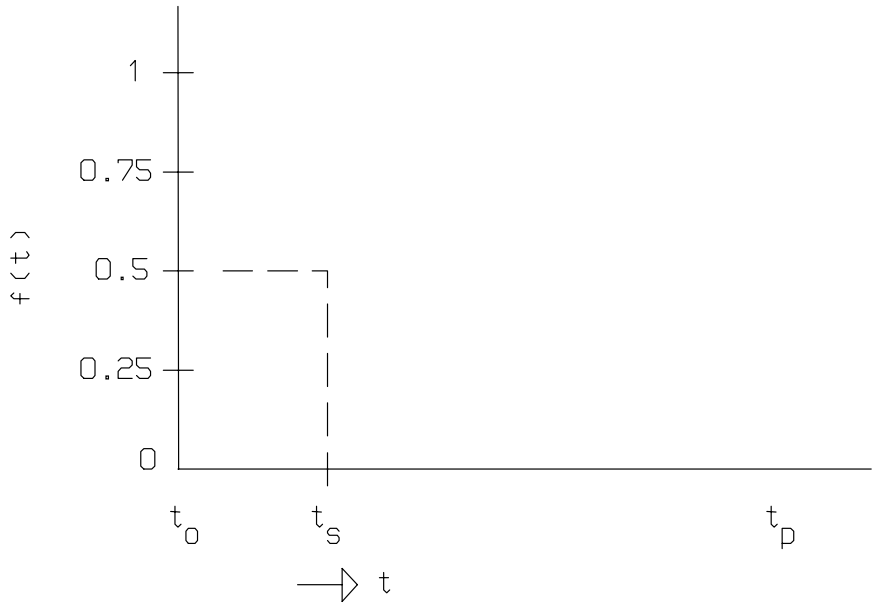


Figure 11.4 Continuous Time Signal

All A/D converters implement the operations of sampling, quantizing, and encoding in order to relate analog input to digital values. One conversion is performed during each sample period T. Quantization of a sampled analog waveform involves the assignment of a finite number of amplitude levels that increases to some full-scale (FS) value such as 5 volts. The quantizing interval q shown in Figure 11.5(a) represents the least significant bit (LSB) resolution. Its value is determined by both the full-scale amplitude value and by the total number of level provided by an n-bit converter, which is 2^n. If we assume that the A/D converter is based on uniform quantization, the quantization error e is a sawtooth function range in value up to +\- 1/2 LSB as shown in Figure 11.5(b).

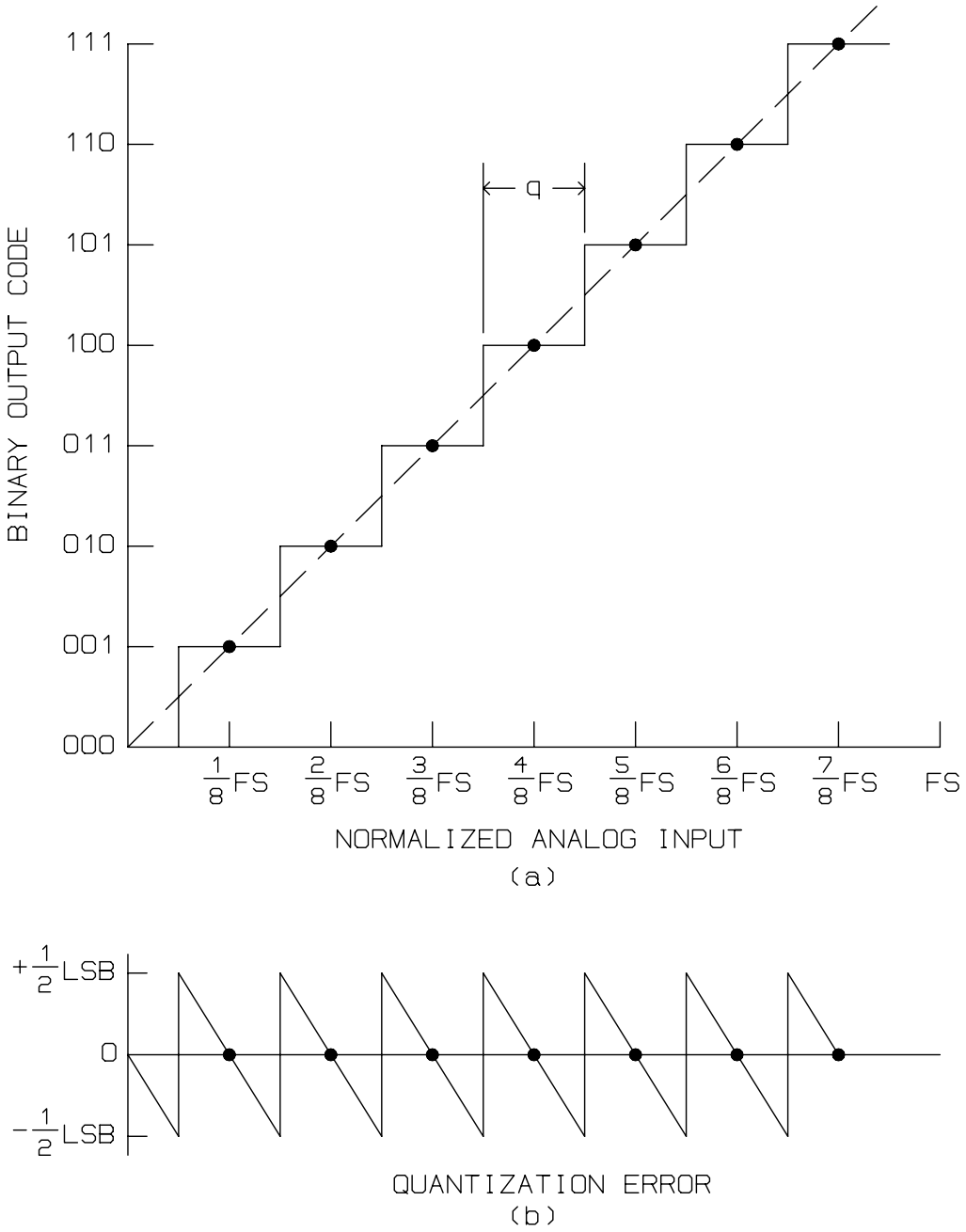


Figure 11.5 Three Bit A/D Converter

An 8 bit converter is used on the EZ-Micro A/D-D/A Lab Board. Such a converter will represent a signal as one of 2^8 or 256 steps. If the Vrh to Vrl reference voltage is 5 volts, each discrete step will represent about 19.53 mV. The quantization error will be in the range of +/- 1/2 LSB, or about +/- 10 mV. In comparison, 12-bit converter would have a step size of about 1.22 mV.

Aliasing

If the signal is not sampled at a sufficient rate, the signal energy that is contained in the higher frequencies will be "aliased" to a lower frequency. This aliasing distortion will be investigated in one of the Laboratory experiments at the end of this Lesson.

Aliasing is illustrated in the time domain by Figure 11.6, which demonstrates the unique mapping of an analog input sample to its sampled-data representation, as shown by the dots. However, the sampled-data representation does not have a unique reverse mapping, as demonstrated by the dashed line aliased waveform.

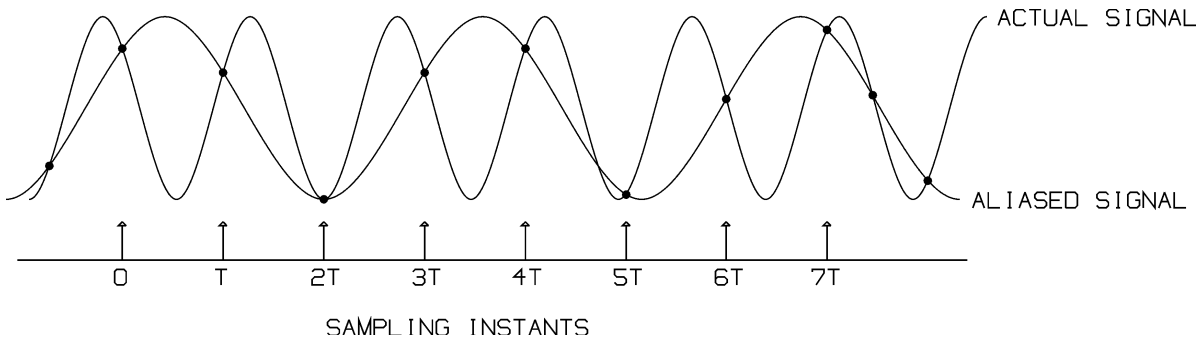


Figure 11.6 Signal Aliasing

Antialiasing filters

In most cases, it is desirable to minimize the sampling rate of a discrete-time system for processing analog systems. This is because the amount of digital signal processing as well as the memory requirements are proportional to the number of samples to be processed. In most applications a prefilter, called an anti-aliasing filter, is used. Consider an example from speech processing. Speech signals may have significant frequency content throughout the range from 4 Hz to 20 kHz. However, intelligible speech can be processed even if we only consider the low frequency band up to about 3.5 kHz. For this reason, speech processing systems may include a lowpass anti-aliasing filter with a cutoff frequency of about 3.5 kHz on the front end.

On the EZ-Micro A/D-D/A Lab Board there is a lowpass anti-aliasing filter on the front end. Refer to the schematic diagram of the A/D-D/A Lab Board. The anti-aliasing filter is a 2nd order lowpass Butterworth filter that is comprised of the op amp designated as U12-C and the associated passive components surrounding it. Note that an analog input applied to the test point designated as T3 will go through the front-end filter but the same input applied instead to the test point designated as T4 will not go through the filter.

Types of A/D Converters

There are many ways in which an analog signal may be "digitized" or sliced into N equivalent segments. Almost all of them can be categorized into one of several basic approaches. Thus, this section will emphasize the only the major categories. Refer to Table 11.1 as we discuss two of the major techniques used in practical A/D converters, one of which is employed on the A/D converter in the EZ-Micro system.

Method and Speed T	Technique	12-bit Rate	8-bit Rate
Integrating,	Dual Slope	100 Hz	1 kHz
1 sec to 1 ms	Charge balancing	50 Hz	500 Hz
	Voltage to freq	2.5 Hz	40 Hz
Voltage comparison,	Successive approx	100 kHz	1 Mhz
1 ms to 10 ns	Tracking	250 kHz	4 kHz
	Simultaneous		25 MHz

Table 11.1 A/D Converter Methods

All A/D converters employ one or more comparators. The integrator type A/D converter is used extensively because it represents an excellent compromise of conversion rate, accuracy, and cost. The essential block diagram of the dual slope integrator system is given in Figure 11.7. At the start, the control logic closes switch AS1 and opens switch AS2. Thus Vin is applied to the integrator. At the same time, the clock is enabled and begins to add pulses to the counter. These processes continue until the counter has been filled. the next clock pulse reloads the counter to zero and transmits an overflow pulse to the control logic. This causes switch AS1 to be opened and AS2 to be closed, thus applying -Vref, which is a known voltage, to the integrator. Whatever value of voltage was stored on the integrator, Vin/RC, will be "removed" linearly with time by -Vref/RC. The comparator, which serves as a zero crossing detector, signals when the removal of voltage from the capacitor is complete. This disables the clock. The key is that the number of counts in the counter is proportional to the unknown voltage. Get it?

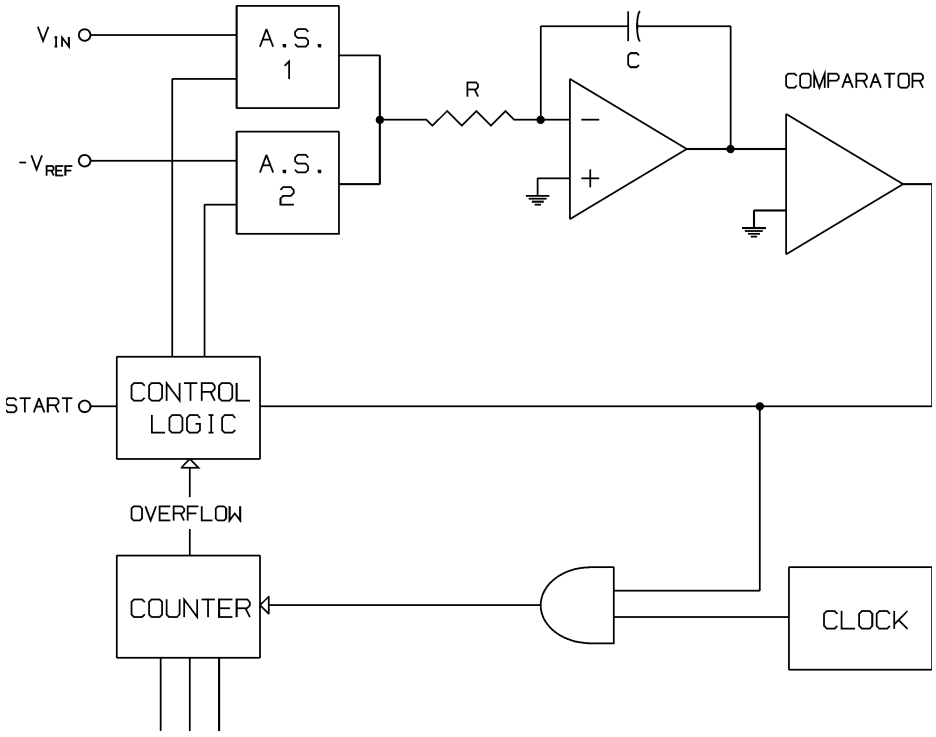


Figure 11.7 Dual Slope ADC

The successive-approximation type of A/D converter offers an enhanced rate of conversion with high accuracy. The functional block diagram of a successive approximation A/D converter, as shown in Figure 11.8, is deceptively simple. The important feature is the special control logic. The functionality of this logic is illustrated by the waveform in the lower portion of the figure. The control logic sequentially determines whether the input is (a) outside the range of the D/A converter, (b) greater than the D/A output for a '1' in the MSB, ... and so forth until (c) greater than the D/A output for a '1' in LSB. There is a significant amount of digital logic required in the implementation of the successive-approximation type of A/D converter. Note that N fixed time periods are needed to deliver an N-bit output providing a constant conversion time. Conversion accuracy depends upon the stability of the reference voltage and the error budget includes a term representing the +/- ½ LSB quantization error as discussed in the previous section.

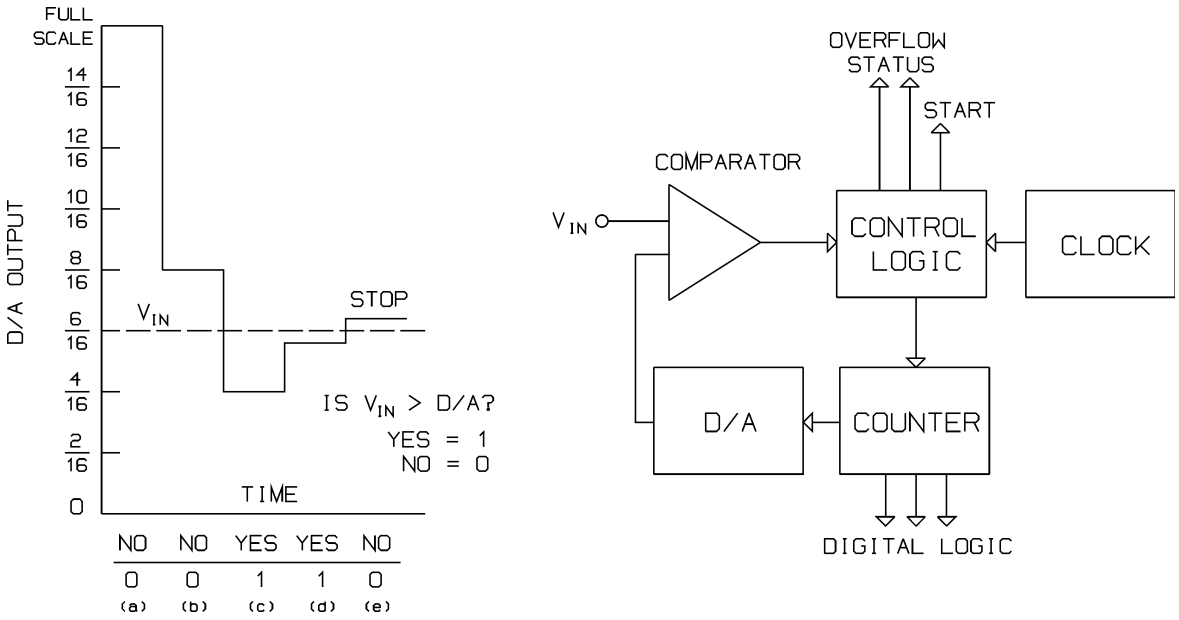


Figure 11.8 Successive approximation ADC

Parallel A/D Converters

The parallel A/D converter is the fastest of all types of converters discussed here since the conversion is made in parallel. The functional diagram of a 3-bit parallel converter is given in Figure 11.9. If an input voltage between 0V and 7V is to be converted, then the reference voltage at comparator A should be 6.5 V. Likewise, the reference voltage value for B should be 5.5 volts, and so on with 0.5 V reference for G. The resistors are selected to achieve this series of references. The comparator outputs must be encoded to produce the appropriate digital word. The digital logic gates necessary to achieve these expressions become the encoder portion of the converter. The conversion rate is limited only by the switching times of the gates and comparators. The accuracy is relatively poor.

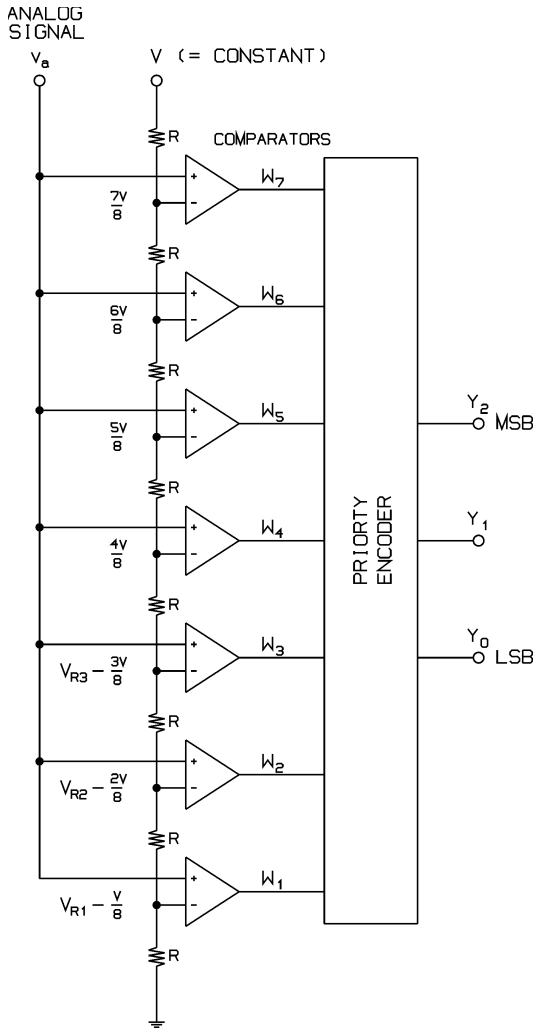


Figure 11.9 Parallel A/D Converter

The voltage-comparison tracking converter is especially useful for parallel conversion systems that employ one A/D converter per channel. In operation its conversion time is variable, and it tracks the input signal at a rate of 2^(-n) per count of the n-bit counter. For example, an 8-bit converter with an integral 1 MHz clock will track an input signal up to a 2^(-8) x 1 MHz or 4 kHz conversion rate, with 256 counts per conversion.

The ADC that is used on the EZ-Micro A-D/D-A Lab Board is the ADC0804 from National Semiconductor. This converter is an 8-bit successive approximation analog-to-digital converter. Its conversion time is specified as 100 usec with the on-chip clock generator running at 640 kHz, which is how the clock is configured on the Lab Board. Analog voltages in the range from 0V to 5v will be converted to 8-bit binary representations in the range from $00 to $FF.

The ADC0804 supports a parallel interface that has been designed by the manufacturer to be easy to interface with the host microprocessor. The access time for the parallel read operation is specified to be 135 ns. Other ADCs may interface with the microprocessor via a serial interface. For example, some converters support the SPI interface.

A/D in 68HC11E0 Microcontroller

The analog-to-digital (A/D) system, a successive approximation converter, uses an all-capacitive charge redistribution technique to convert analog signals to digital values.

Overview

The A/D system is an 8-channel, 8-bit, multiplexed-input converter. The converter does not require external sample and hold circuits because of the type of charge redistribution technique used. A/D converter timing can be synchronized to the system E clock or to an internal resistor capacitor (RC) oscillator. The A/D converter system consists of four functional blocks: multiplexer, analog converter, digital control, and result storage. Refer to Fig 11.10.

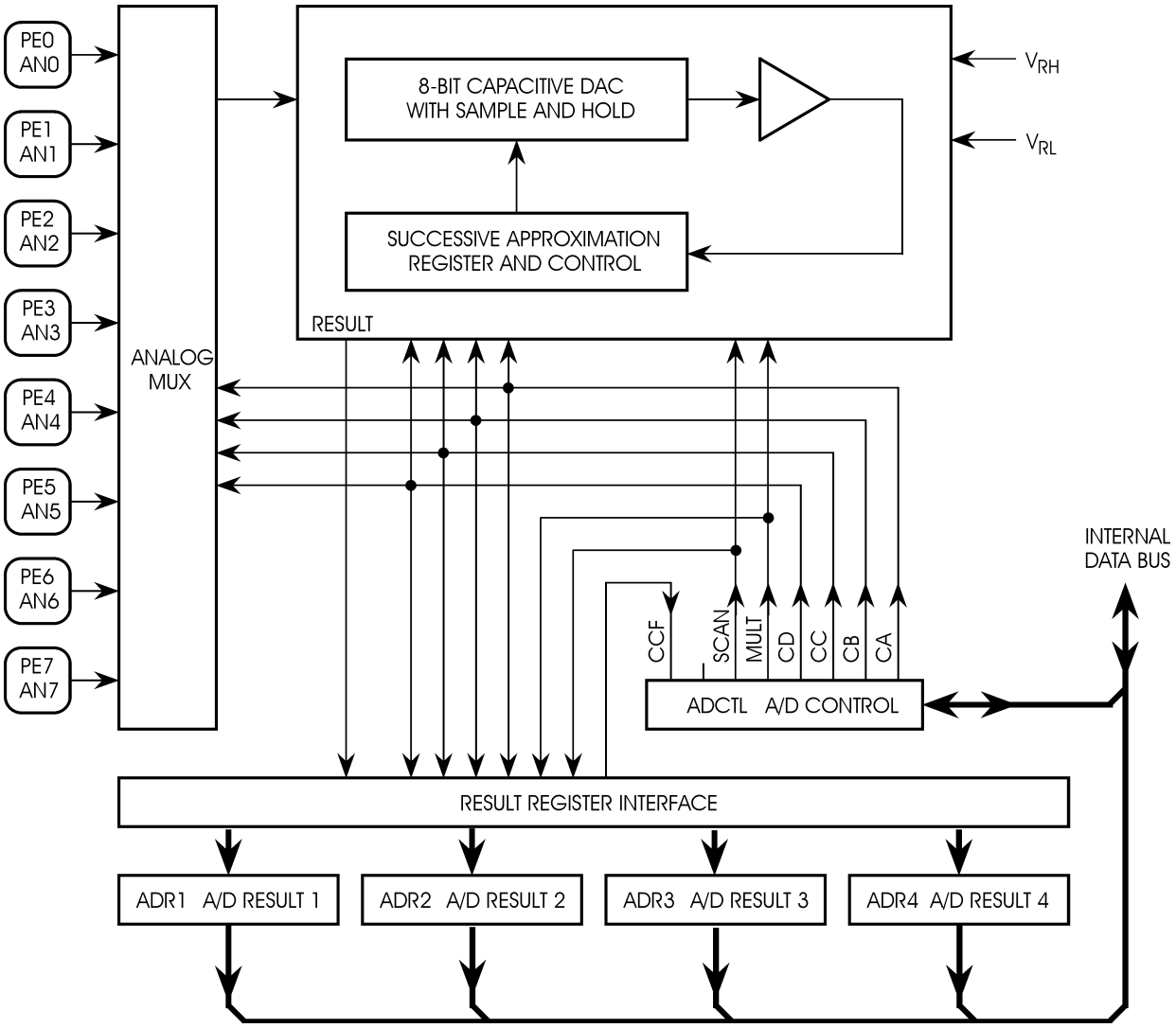


Figure 11.10 A/D Converter Block

Multiplexer

The multiplexer selects one of 16 inputs for conversion. Input selection is controlled by the value of bits CD:CA in the ADCTL register. The eight port E pins are fixed-direction analog inputs to the multiplexer, and additional internal analog signal lines are routed to it. Port E pins also can be used as digital inputs. Digital reads of port E pins are not recommended during the sample portion of an A/D conversion cycle, when the gate signal to the N-channel input gate is on. Because no P-channel devices are directly connected to either input pins or reference voltage pins, voltages above V_{DD} do not cause a latchup problem, although current should be limited according to maximum ratings.

Analog Converter

Conversion of an analog input selected by the multiplexer occurs in this block. It contains a digital-to-analog capacitor (DAC) array, a comparator, and a successive approximation register (SAR). Each conversion is a sequence of eight comparison operations, beginning with the most significant bit (MSB). Each comparison determines the value of a bit in the successive approximation register. The DAC array performs two functions. It acts as a sample and hold circuit during the entire conversion sequence and provides comparison voltage to the comparator during each successive comparison. The result of each successive comparison is stored in the SAR. When a conversion sequence is complete, the contents of the SAR are transferred to the appropriate result register. A charge pump provides switching voltage to the gates of analog switches in the multiplexer. Charge pump output must stabilize between 7 and 8 volts within up to 100 µs before the converter can be used. The charge pump is enabled by the ADPU bit in the OPTION register.

Digital Control

All A/D converter operations are controlled by bits in register ADCTL. In addition to selecting the analog input to be converted, ADCTL bits indicate conversion status and control whether single or continuous conversions are performed. Finally, the ADCTL bits determine whether conversions are performed on single or multiple channels.

Result Registers

Four 8-bit registers ADR[4:1] store conversion results. Each of these registers can be accessed by the processor in the CPU. The conversion complete flag (CCF) indicates when valid data is present in the result registers. The result registers are written during a portion of the system clock cycle when reads do not occur, so there is no conflict.

A/D Converter Clocks

The CSEL bit in the OPTION register selects whether the A/D converter uses the system E clock or an internal RC oscillator for synchronization. When E-clock frequency is below 750 kHz, charge leakage in the capacitor array can cause errors, and the internal oscillator should be used. When the RC clock is used, additional errors can occur because the comparator is sensitive to the additional system clock noise.

Conversion Sequence

A/D converter operations are performed in sequences of four conversions each. A conversion sequence can repeat continuously or stop after one iteration. The conversion complete flag (CCF) is set after the fourth conversion in a sequence to show the availability of data in the result registers. Figure 11.11 shows the timing of a typical sequence. Synchronization is referenced to the system E clock.

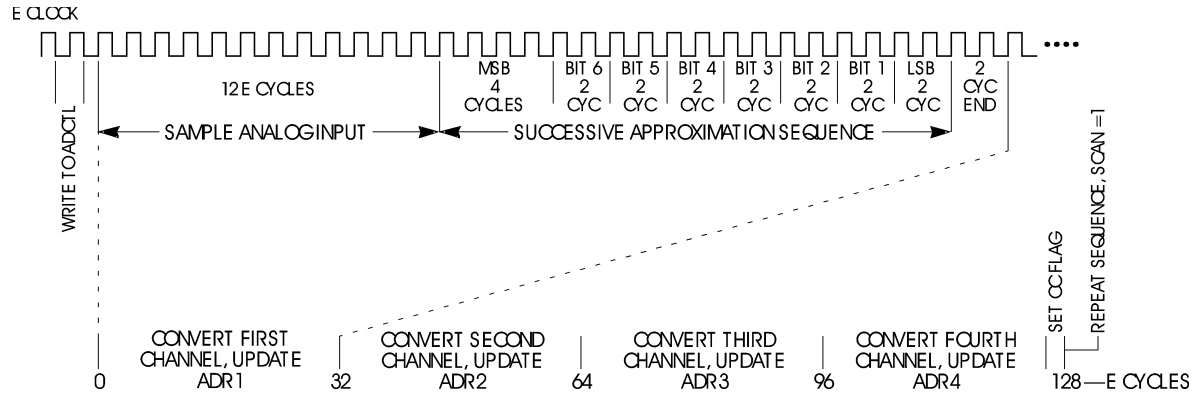


Figure 11.11 A/D Conversion Sequence

A/D Converter Power-Up and Clock Select

Bit 7 of the OPTION register controls A/D converter power-up. Clearing ADPU removes power from and disables the A/D converter system. Setting ADPU enables the A/D converter system. Stabilization of the analog bias voltages requires a delay of as much as 100 µs after turning on the A/D converter. When the A/D converter system is operating with the MCU E clock, all switching and comparator operations are inherently synchronized to the main MCU clocks. This allows the comparator output to be sampled at relatively quiet times during MCU clock cycles. Since the internal RC oscillator is asynchronous to the MCU clock, there is more error attributable to internal system clock noise. A/D converter accuracy is reduced slightly while the internal RC oscillator is being used (CSEL = 1).

Address: $1039

	Bit 7	6	5	4	3	2	1	Bit 0
Read: / **Write:**	**ADPU**	**CSEL**	**IRQE(1)**	**DLY(1)**	**CME**		**CR1(1)**	**CR0(1)**
Reset:	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**

1. Can be written only once in first 64 cycles out of reset in normal modes or at any time in special modes

= Unimplemented

Figure 11.12 System Configuration Options Register (OPTION)

ADPU —A/D Power-Up Bit

0 = A/D powered down

1 = A/D powered up

CSEL — Clock Select Bit

0 = A/D and EEPROM use system E clock.

1 = A/D and EEPROM use internal RC clock.

IRQE — Configure IRQ for Edge-Sensitive Only Operation

DLY — Enable Oscillator Startup Delay Bit

0 = The oscillator startup delay coming out of stop is bypassed and the MCU resumes processing within about four bus cycles.

1 = A delay of approximately 4000 E-clock cycles is imposed as the MCU is started up from the stop power-saving mode. This delay allows the crystal oscillator to stabilize.

CME — Clock Monitor Enable Bit

Bit 2 — Not implemented

Always reads 0

CR[1:0] — COP Timer Rate Select Bits

Conversion Process

The A/D conversion sequence begins one E-clock cycle after a write to the A/D control/status register, ADCTL. The bits in ADCTL select the channel and the mode of conversion. An input voltage equal to V_{RL} converts to $00 and an input voltage equal to V_{RH} converts to $FF (full scale), with no overflow indication. For ratiometric conversions of this type, the source of each analog input should use V_{RH} as the supply voltage and be referenced to V_{RL}.

Channel Assignments

The multiplexer allows the A/D converter to select one of 16 analog signals. Eight of these channels correspond to port E input lines to the MCU, four of the channels are internal reference points or test functions, and four channels are reserved. Refer to Table 11.2

Channel Number	Channel Signal	Result in ADRx if MULT = 1
1	AN0	ADR1
2	AN1	ADR2
3	AN2	ADR3
4	AN3	ADR4
5	AN4	ADR1
6	AN5	ADR2
7	AN6	ADR3
8	AN7	ADR4
9 - 12	Reserved	—
13	V_{RH} (1)	ADR1
14	V_{RL} (1)	ADR2
15	$(V_{RH})/2$ (1)	ADR3
16	Reserved (1)	ADR4

1. Used for factory testing

Table 11.2 Converter Channel Assignments

Single-Channel Operation

The two types of single-channel operation are:

1. When SCAN = 0, the single selected channel is converted four consecutive times. The first result is stored in A/D result register 1 (ADR1), and the fourth result is stored in ADR4. After the fourth conversion is complete, all conversion activity is halted until a new conversion command is written to the ADCTL register.

2. When SCAN = 1, conversions continue to be performed on the selected channel with the fifth conversion being stored in register ADR1 (overwriting the first conversion result), the sixth conversion overwriting ADR2, and so on.

Multiple-Channel Operation

The two types of multiple-channel operation are:

1 When SCAN = 0, a selected group of four channels is converted one time each. The first result is stored in A/D result register 1 (ADR1), and the fourth result is stored in ADR4. After the fourth conversion is complete, all conversion activity is halted until a new conversion command is written to the ADCTL register.

2. When SCAN = 1, conversions continue to be performed on the selected group of channels with the fifth conversion being stored in register ADR1 (replacing the earlier

conversion result for the first channel in the group), the sixth conversion overwriting ADR2, and so on.

Operation in Stop and Wait Modes

If a conversion sequence is in progress when either the stop or wait mode is entered, the conversion of the current channel is suspended. When the MCU resumes normal operation, that channel is resampled and the conversion sequence is resumed. As the MCU exits wait mode, the A/D circuits are stable and valid results can be obtained on the first conversion. However, in stop mode, all analog bias currents are disabled and it is necessary to allow a stabilization period when leaving stop mode. If stop mode is exited with a delay (DLY = 1), there is enough time for these circuits to stabilize before the first conversion. If stop mode is exited with no delay (DLY bit in OPTION register = 0), allow 10 ms for the A/D circuitry to stabilize to avoid invalid results.

A/D Control/Status Register

All bits in this register can be read or written, except bit 7, which is a read-only status indicator, and bit 6, which always reads as 0. Write to ADCTL to initiate a conversion. To quit a conversion in progress, write to this register and a new conversion sequence begins immediately.

Address: $1030

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	CCF		SCAN	MULT	CD	CC	CB	CA
Write:								
Reset:	0	0	Indeterminate after reset					

1. Can be written only once in first 64 cycles out of reset in normal modes or at any time in special modes

= Unimplemented

Figure 11.13. A/D Control/Status Register (ADCTL)

CCF — Conversion Complete Flag

A read-only status indicator, this bit is set when all four A/D result registers contain valid conversion results. Each time the ADCTL register is overwritten, this bit is automatically cleared to 0 and a conversion sequence is started. In the continuous mode, CCF is set at the end of the first conversion sequence.

Bit 6 — Unimplemented

Always reads 0

SCAN — Continuous Scan Control Bit

When this control bit is clear, the four requested conversions are performed once to fill the four result registers. When this control bit is set, conversions are performed continuously with the result registers updated as data becomes available.

MULT — Multiple Channel/Single Channel Control Bit

When this bit is clear, the A/D converter system is configured to perform four consecutive conversions on the single channel specified by the four channel select bits CD:CA (bits [3:0] of the ADCTL register). When this bit is set, the A/D system is configured to perform a conversion on each of four channels where each result register corresponds to one channel.

Address: $1030

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	**CCF**		**SCAN**	**MULT**	**CD**	**CC**	**CB**	**CA**
Write:								
Reset:	**0**	**0**	**Indeterminate after reset**					

= Unimplemented

Figure 11.13. A/D Control/Status Register (ADCTL)

CCF — Conversion Complete Flag

A read-only status indicator, this bit is set when all four A/D result registers contain valid conversion results. Each time the ADCTL register is overwritten, this bit is automatically cleared to 0 and a conversion sequence is started. In the continuous mode, CCF is set at the end of the first conversion sequence.

Bit 6 — Unimplemented

Always reads 0

SCAN — Continuous Scan Control Bit

When this control bit is clear, the four requested conversions are performed once to fill the four result registers. When this control bit is set, conversions are performed continuously with the result registers updated as data becomes available.

MULT — Multiple Channel/Single Channel Control Bit

When this bit is clear, the A/D converter system is configured to perform four consecutive conversions on the single channel specified by the four channel select bits CD:CA (bits [3:0] of the ADCTL register). When this bit is set, the A/D system is configured to perform a conversion on each of four channels where each result register corresponds to one channel.

NOTE: *When the multiple-channel continuous scan mode is used, extra care is needed in the design of circuitry driving the A/D inputs. The charge on the capacitive DAC array before the sample time is related to the voltage on*

the previously converted channel. A charge share situation exists between the internal DAC capacitance and the external circuit capacitance. Although the amount of charge involved is small, the rate at which it is repeated is every 64 µs for an E clock of 2 MHz. The RC charging rate of the external circuit must be balanced against this charge sharing effect to avoid errors in accuracy. Refer to M68HC11 Reference Manual, Motorola document order number M68HC11RM/AD, for further information.

CD:CA — Channel Selects D:A Bits

Refer to Table 11.3. When a multiple channel mode is selected (MULT = 1), the two least significant channel select bits (CB and CA) have no meaning and the CD and CC bits specify which group of four channels is to be converted.

Channel Select Control Bits CD:CC:CB:CA	Channel Signal	Result in ADRx if MULT = 1
0000	AN0	ADR1
0001	AN1	ADR2
0010	AN2	ADR3
0011	AN3	ADR4
0100	AN4	ADR1
0101	AN5	ADR2
0110	AN6	ADR3
0111	AN7	ADR4
10XX	Reserved	—
1100	V_{RH} (1)	ADR1
1101	V_{RL} (1)	ADR2
1110	$(V_{RH})/2$ (1)	ADR3
1111	Reserved (1)	ADR4

1. Used for factory testing

Table 2. A/D Converter Channel Selection

A/D Converter Result Registers

These read-only registers hold an 8-bit conversion result. Writes to these registers have no effect. Data in the A/D converter result registers is valid when the CCF flag in the ADCTL register is set, indicating a conversion sequence is complete. If conversion results are needed sooner, refer to Figure 11.11, which shows the A/D conversion sequence diagram.

Register name: **Analog-to-Digital Converter Result Register 1** **Address: $1031**

	Bit 7	6	5	4	3	2	1	Bit 0
Read:								
Write:								

Reset: **Indeterminate after reset**

Register name: **Analog-to-Digital Converter Result Register 1** **Address: $1032**

	Bit 7	6	5	4	3	2	1	Bit 0
Read:								
Write:								

Reset: **Indeterminate after reset**

Register name: **Analog-to-Digital Converter Result Register 1** **Address: $1033**

	Bit 7	6	5	4	3	2	1	Bit 0
Read:								
Write:								

Reset: **Indeterminate after reset**

Register name: **Analog-to-Digital Converter Result Register 1** **Address: $1034**

	Bit 7	6	5	4	3	2	1	Bit 0
Read:								
Write:								

Reset: **Indeterminate after reset**

= Unimplemented

Figure 11.14. Analog-to-Digital Converter Result Registers (ADR1–ADR4)

Laboratory 11.1

Digital to Analog output

The Micro Tutor Board has Digital to Analog chip and Current to Voltage coveter Op-AMP circuit. DAC is connected to Output Latch 74LS373 and it has IO address of $5000 Hex. The following Assembly language Program outputs Hex number from 00 to FF and it starts all over again. This will give Sawtooth output at Pin 6 of U15 or pin 17 of J4. Switch jumper H5 from 1-2 to 2-3. The sawtooth output will be in reverse direction.

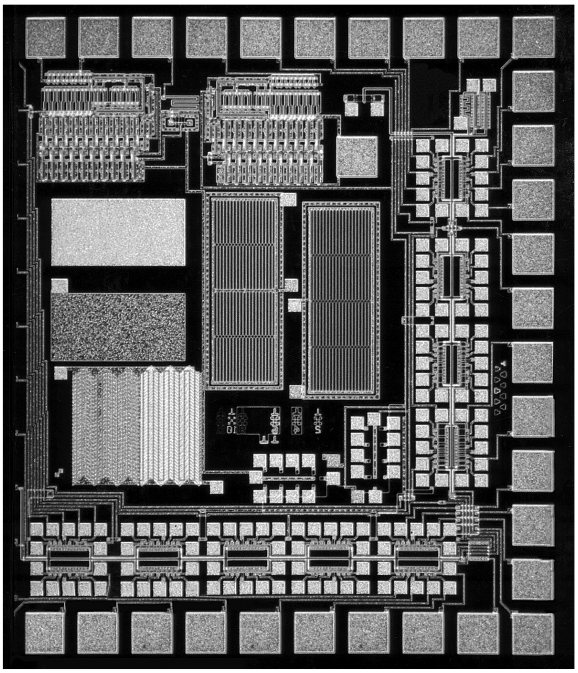
DAC.ASM

```
*************************************************************************
* This Program (DAC.ASM) tests Digital to Analog circuit                *
* on EZ-MICRO TUTOR Board From Advanced Micorcomputer Systems Inc.      *
* The updated software can be downloaded from our website               *
*                                                                       *
* www.advancedmsinc.com                                                 *
*                                                                       *
* copyright 2000 by Advanced Micorcomputer Systems Inc.                 *
*                                                                       *
*************************************************************************
* DAC test
* output a ramp function

BUFALO  EQU  $e000      *start of BUFFALO
DAC   EQU  $5000        *memory mapped location of DAC latch

        ORG  $2000
MAIN
        LDAA  #$00
LOOP    INCA                    *increment ramp value
        STAA  DAC
        JMP   LOOP
* Delay subroutine
DLY10MS EQU  *             delay 10ms at E = 2MHz
        PSHX
        LDX  #$0D06
DLYLP   DEX
        BNE  DLYLP
        PULX

        RTS
```

A2D.ASM

```
*************************************************************************
* This Program (A2D.ASM) tests Analog to Digital circuit                *
* on EZ-MICRO TUTOR Board From Advanced Micorcomputer Systems Inc.      *
* The updated software can be downloaded from our website               *
* www.advancedmsinc.com                                                 *
* copyright 2000 by Advanced Micorcomputer Systems Inc.                 *
*                                                                       *
*************************************************************************
* Analog to Digital Converter Test

* Equates

* Registers
ADCTL    EQU  $30
ADR1     EQU  $31
ADR2     EQU  $32
ADR3     EQU  $33
ADR4     EQU  $34

* RAM

* External References
BUFALO   EQU  $E000
OUTPUT   EQU  $E3D8
*
****************************
* Define variables.        *
****************************
      ORG  $2000
MAIN
      JSR  INIT_AD         *initialize A/D converter hardware
      JSR  INIT_LCD        *initialize LCD module

TOP   LDAA #5              *select channel 5
      STAA ADCTL           *this starts the conversion
      BRCLR ADCTL $80 *    *branch to self until CCF=1

      LDAA  #$02
      STAA  CONTROL        *LCD return home
      JSR   DLY10MS

      LDX #ADR1            *point X register at ADR 1 register
      LDAA 0,X             *do the read
      JSR  CONVERT         *convert to BCD and return in 2 (ascii) nibbles
ACCA and ACCB
      STAA  DATA           *save value in HC11 ram
      JSR   DLY10MS
      JSR  OUTPUT   *report upper nibble (ascii) to EZ Micro Advanced manager

      TBA                  *now process lower nibble
      STAA  DATA
      JSR   DLY10MS
      JSR  OUTPUT   *report lower nibble (ascii) to EZ-Micro Advanced manager

      LDAA #$0D    *send carriage return (CR) to EZ-Micro Advanced manager
```

```
      JSR  OUTPUT
      JMP  TOP

* Initialize A/D converter hardware
INIT_AD
        LDAA #5              *select channel 5
        STAA ADCTL
        RTS

* Initialize LCD module
INIT_LCD
        LDAA  #$3C     *Function set: data length=8, lines=2, dots=5x10
        STAA  CONTROL
        JSR   DLY10MS

        LDAA  #$0F            *Display ON, cursor ON, blink On
        STAA  CONTROL
        JSR   DLY10MS

        LDAA  #$01            *Clear display
        STAA  CONTROL
        JSR   DLY10MS

        LDAA  #$06            *Entry mode set: increment, no display shift
        STAA  CONTROL
        JSR   DLY10MS

      * Let's begin at first character in first line, address = $80
        LDAA  #$80            *Set DDRAM address
        STAA  CONTROL
        JSR   DLY10MS
        RTS

* Delay subroutine
DLY10MS EQU  *             delay 10ms at E = 2MHz
        PSHX
        LDX  #$0D06
DLYLP   DEX
        BNE  DLYLP
        PULX
        RTS
* Convert binary to ascii subroutine
* Binary value to convert is passed in ACCA
* BCD value returned in ACCA:ACCB
* example:   ACCA=$14
*            ACCA=$31 ACCB=$34
CONVERT PSHA
        ANDA  #$0F
        ADDA  #$30
        TAB
        PULA
        LSRA
        LSRA
        LSRA
        LSRA
        ADDA  #$30
        RTS
```



Lesson Twelve

Assembler Manual

Introduction

This is the user's reference manual for the IBM-PC hosted Motorola 8 bit cross assemblers. It details the features and capabilities of the cross assemblers, assembler syntax and directives, options, and listings. It is intended as a detailed reference and an introduction for those unfamiliar with Motorola assembler syntax and usage.

Assemblers are programs that process assembly language source program statements and translate them into executable machine language object files. A programmer writes his source program using any text editor or word processor that can produce an ASCII text output. With some word processors this is known as "non document" mode. Non document mode produces a file without the non-printable embedded control characters that are used in document formating. (Caution: assembling a file that has been formatted with embedded control characters may produce assembler errors. The solution is to convert the source file to ASCII text.) Once the source code is written, the source file is assembled by processing the file via the assembler. Cross assemblers (such as the Motorola Assemblers) allow source programs written and edited on one computer (the host) to generate executable code for another computer (the target). The executable object file can then be downloaded and run on the target system. In this case the host is an IBM-PC or compatible and the target system is based on a Motorola 68HC11 microprocessor.

Assembly Language

The symbolic language used to code source programs to be processed by the Assembler is called assembly language. The language is a collection of mnemonic symbols representing: operations (i.e., machine instruction mnemonics or directives to the assembler), symbolic names, operators, and special symbols. The assembly language provides mnemonic operation codes for all machine instructions in the instruction set. The instructions are defined and explained in the Programming Reference Manuals for the specific devices, available from Motorola. The assembly language also contains mnemonic directives which specify auxiliary actions to be performed by the Assembler. These directives are not always translated into machine language.

Operating Environment

These assemblers will run on any IBM-PC, XT, AT, PS-2, or true compatible. The assemblers may be run off of a floppy disk drive or they may be copied onto a hard drive for execution. DOS 2.0 or later is required.

Assembler Processing

The Macro Assembler is a two-pass assembler. During the first pass, the source program is read to develop the symbol table. During the second pass, the object file is created (assembled) with reference to the table developed in pass one. It is during the second pass that the source program listing is also produced. Each source statement is processed completely before the next source statement is read. As each statement is processed, the Assembler examines the label, operation code, and operand fields. The operation code table is scanned for a match with a known opcode.

During the processing of a standard operation code mnemonic, the standard machine code is inserted into the object file. If an Assembler directive is being processed, the proper action is taken.

Any errors that are detected by the Assembler are displayed before the actual line containing the error is printed. If no source listing is being produced, error messages are still displayed to indicate that the assembly process did not proceed normally.

Coding Assembly Language Programs

Programs written in assembly language consist of a sequence of source statements. Each source statement consists of a sequence of ASCII characters ending with a carriage return. Appendix A contains a list of the supported character set.

Source Statement Format

Each source statement may include up to four fields: a label (or "*" for a comment line), an operation (instruction mneumonic or assembler directive), an operand, and a comment.

Label Field

The label field occurs as the first field of a source statement. The label field can take one of the following forms:

1. An asterisk (*) as the first character in the label field indicates that the rest of the source statement is a comment. Comments are ignored by the Assembler, and are printed on the source listing only for the programmer's information.

2. A whitespace character (blank or tab) as the first character indicates that the label field is empty. The line has no label and is not a comment. 3. A symbol character as the first character indicates that the line has a label. Symbol characters are the upper or lower case letters a- z, digits 0-9, and the special characters, period (.), dollar sign ($), and underscore (_). Symbols consist of one to 15 characters, the first of which must be alphabetic or the special characters period (.) or underscore (_). All characters are significant and upper and lower case letters are distinct. A symbol may occur only once in the label field. If a symbol does occur more than once in a label field, then each reference to that symbol will be flagged with an error.

With the exception of some directives, a label is assigned the value of the program counter of the first byte of the instruction or data being assembled. The value assigned to the label is absolute. Labels may optionally be ended with a colon (:). If the colon is used it is not part of the label but merely acts to set the label off from the rest of the source line. Thus the following code fragments are equivalent:

```
here:   deca
        bne here

here    deca
        bne here
```

A label may appear on a line by itself. The assembler interprets this as set the value of the label equal to the current value of the program counter. The symbol table has room for at least 2000 symbols of length 8 characters or less. Additional characters up to 15 are permissible at the expense of decreasing the maximum number of symbols possible in the table.

Operation Field

The operation field occurs after the label field, and must be preceded by at least one whitespace character. The operation field must contain a legal opcode mneumonic or an assembler directive. Upper case characters in this field are converted to lower case before being checked as a legal mneumonic. Thus 'nop', 'NOP', and 'NoP' are recognized as the same mneumonic. Entries in the operation field may be one of two types:

Opcode. These correspond directly to the machine instructions. The operation code includes any register name associated with the instruction. These register names must not be separated from the opcode with any whitespace characters. Thus 'clra' means clear accumulator A, but 'clr a' means clear memory location identified by the label 'a'.

Directive. These are special operation codes known to the Assembler which control the assembly process rather than being translated into machine instructions.

Operand Field

The operand field's interpretation is dependent on the contents of the operation field. The operand field, if required, must follow the operation field, and must be preceded by at least one whitespace character. The operand field may contain a symbol, an expression, or a combination of symbols and expressions separated by commas. The operand field of machine instructions is used to specify the addressing mode of the instruction, as well as the operand of the instruction.

The following tables summarize the operand field formats for the various processor families. (NOTE: in these tables parenthesis "()" signify optional elements and angle brackets "<>" denote an expression is inserted. These syntax elements are present only for clarification of the format and are not inserted as part of the actual source program. All other characters are significant and must be used when required.)

M68HC11 Operand Syntax

For the M68HC11, the following operand formats exist:

Operand Format	M68HC11 Addressing Mode
no operand	accumulator and inherent
<expression>	direct, extended, or relative
#<expression>	immediate
<expression>,X	indexed with X register
<expression>,Y	indexed with Y register
<expression> <expression>	bit set or clear
<expression> <expression> <expression>	bit test and branch

The bit manipulation instruction operands are separated by spaces in this case since the HC11 allows bit manipulation instructions on indexed addresses. Thus a ',X' or ',Y' may be added to the final two formats above to form the indexed effective address calculation.

Details of the M68HC11 addressing modes may be found in Appendix B.The operand fields of assembler directives are described in Chapter 4.

Expressions

An expression is a combination of symbols, constants, algebraic operators, and parentheses. The expression is used to specify a value which is to be used as an operand.

Expressions may consist of symbols, constants, or the character '*' (denoting the current value of the program counter) joined together by one of the operators: + - * / % & | ^ .

Operators

The operators are the same as in c:

+	add
-	subtract
*	multiply
/	divide
%	remainder after division
&	bitwise and
|	bitwise or
^	bitwise exclusive or

Expressions are evaluated left to right and there is no provision for parenthesized expressions. Arithmetic is carried out in signed two- complement integer precision (that's 16 bits on the IBM PC).

Symbols

Each symbol is associated with a 16-bit integer value which is used in place of the symbol during the expression evaluation. The asterisk (*) used in an expression as a symbol represents the current value of the location counter (the first byte of a multi-byte instruction).

Constants

Constants represent quantities of data that do not vary in value during the execution of a program. Constants may be presented to the assembler in one of five formats: decimal, hexadecimal, binary, or octal, or ASCII. The programmer indicates the number format to the assembler with the following prefixes:

$	HEX	or trailing H
%	BINARY	or trailing B
@	OCTAL	or trailing Q
'	ASCII	

Unprefixed constants are interpreted as decimal. The assembler converts all constants to binary machine code and are displayed in the assembly listing as hex.

A decimal constant consists of a string of numeric digits. The value of a decimal constant must fall in the range 0-65535, inclusive. The following example shows both valid and invalid decimal constants:

VALID	INVALID	REASON INVALID
12	123456	more than 5 digits
12345	12.3	invalid character

A hexadecimal constant consists of a maximum of four characters from the set of digits (0-9) and the upper case alphabetic letters (A-F), and is preceded by a dollar sign ($). Hexadecimal constants must be in the range $0000 to $FFFF. The following example shows both valid and invalid hexadecimal constants:

VALID	INVALID	REASON INVALID
$12	ABCD	no preceding "$"
$ABCD	$G2A	invalid character
$001F	$2F018	too many digits

A binary constant consists of a maximum of 16 ones or zeros preceded by a percent sign (%). The following example shows both valid and invalid binary constants:

VALID	INVALID	REASON INVALID
%00101	1010101	missing percent
%1	%10011000101010111	too many digits
%10100	%210101	invalid digit

An octal constant consists of a maximum of six numeric digits, excluding the digits 8 and 9, preceded by a commercial at-sign (@). Octal constants must be in the ranges @0 to @177777. The following example shows both valid and invalid octal constan ts:

VALID	INVALID	REASON INVALID
@17634	@2317234	too many digits
@377	@277272	out of range
@177600	@23914	invalid character

A single ASCII character can be used as a constant in expressions. ASCII constants are preceded by a single quote ('). Any character, including the single quote, can be used as a character constant. The following example shows both valid and inval id character constants:

VALID	INVALID	REASON INVALID
'*	'VALID	too long

For the invalid case above the assembler will not indicate an error.Rather it will assemble the first character and ignore the remainder.

Comment Field

The last field of an Assembler source statement is the comment field. This field is optional and is only printed on the source listing for documentation purposes. The comment field is separated from the operand field (or from the operation field if no operand is required) by at least one whitespace character. The comment field can contain any printable ASCII characters.

Comments may also begin with a semicolon. When a semicolon is used the comment may begin in any column.

Assembler Output

The Assembler output includes an optional listing of the source program and an object file which is in the Motorola S Record format. Details of the S Record format may be found in Appendix E.The Assembler will normally suppress the printing of the source listing. This condition, as well as others, can be overridden via options supplied on the command line that invoked the Assembler.

Each line of the listing contains a reference line number, the address and bytes assembled, and the original source input line. If an input line causes more than 6 bytes to be output (e.g. a long FCC directive), additional bytes (up to 64) are listed on succeeding lines with no address preceding them.

The assembly listing may optionally contain a symbol table or a cross reference table of all symbols appearing in the program. These are always printed at the end of the assembly listing if either the symbol table or cross reference table options (Paragraph 4.8) are in effect. The symbol table contains the name of each symbol, along with its defined value. The cross reference table additionally contains the assembler-maintained source line number of every reference to every symbol. The format of the cross reference table is shown in Appendix D.

Error Messages

Error diagnostic messages are placed in the listing file just before the line containing the error. The format of the error line is:

```
Line_number:        Description of error
  or
Line_number:        Warning ---- Description of error
```

Errors in pass one cause cancellation of pass two. Warning do not cause cancellation of pass two but are indications of a possible problem. Error messages are meant to be self-explanatory. If more than one file is being assembled, the file name precedes the error:

```
File_name,Line_number: Description of error
```

Some errors are classed as fatal and cause an immediate termination of the assembly. Generally this happens when a temporary file cannot be created or is lost during assembly.

Assembler Directives

The Assembler directives are instructions to the Assembler, rather than instructions to be directly translated into object code. This chapter describes the directives that are recognized by the

Assemblers. Detailed descriptions of each directive are arranged alphabetically. The notations used in this chapter are: ( ) Parentheses denote an optional element.

XYZ The names of the directives are printed in capital letters.

< > The element names are printed in lower case and contained in angle brackets. All elements outside of the angle brackets '<>' must be specified as-is. For example, the syntactical element (<number>,) requires the comma to be specified if the optional element <number> is selected. The following elements are used in the subsequent descriptions:

<comment>	A statement's comment field
<label>	A statement label
<expression>	An Assembler expression
<expr>	An Assembler expression
<number>	A numeric constant
<string>	A string of ASCII characters
<delimiter>	A string delimiter
<option>	An Assembler option
<symbol>	An Assembler symbol
<sym>	An Assembler symbol
<sect>	A relocatable program section

In the following descriptions of the various directives, the syntax, or format, of the directive is given first. This will be followed with the directive's description.

BSZ - Block Storage of Zeros

(<label>) BSZ <expression> (<comment>)

The BSZ directive causes the Assembler to allocate a block of bytes. Each byte is assigned the initial value of zero. The number of bytes allocated is given by the expression in the operand field. If the expression contains symbols that are either undefined or forward referenced (i.e. the definition occurs later on in the file), or if the expression has a value of zero, an error will be generated.

EQU - Equate Symbol to a Value

<label> EQU <expression> (<comment>)

The EQU directive assigns the value of the expression in the operand field to the label. The EQU directive assigns a value other than the program counter to the label. The label cannot be redefined anywhere else in the program. The expression cannot contain any forward references or undefined symbols. Equates with forward references are flagged with Phasing Errors.

FCB - Form Constant Byte or DB - Define Byte

(<label>) FCB <expr>(,<expr>,...,<expr>) (<comment>)

The FCB directive may have one or more operands separated by commas. The value of each operand is truncated to eight bits, and is stored in a single byte of the object program. Multiple operands are stored in successive bytes. The operand may be a numeric constant, a character constant, a symbol, or an expression. If multiple operands are present, one or more of them can be null (two adjacent commas), in which case a single byte of zero will be assigned for that

operand. An error will occur if the upper eight bits of the evaluated operands' values are not all ones or all zeros.

FCC - Form Constant Character String

(<label>) FCC <delimiter><string><delimiter> (<comment>)

The FCC directive is used to store ASCII strings into consecutive bytes of memory. The byte storage begins at the current program counter. The label is assigned to the first byte in the string. Any of the printable ASCII characters can be contained in the string. The string is specified between two identical delimiters which can be any printable ASCII character. The first non-blank character after the FCC directive is used as the delimiter.
Example:

```
LABEL1  FCC , ABC,
```

assembles ASCII ABC at location LABEL1

FDB - Form Double Byte Constant or DW - Define Word

(<label>) FDB <expr>(,<expr>,...,<expr>) (<comment>)

The FDB directive may have one or more operands separated by commas. The 16-bit value corresponding to each operand is stored into two consecutive bytes of the object program. The storage begins at the current program counter. The label is assigned to the first 16-bit value. Multiple operands are stored in successive bytes. The operand may be a numeric constant, a character constant, a symbol, or an expression. If multiple operands are present, one or more of them can be null (two adjacent commas), in which case two bytes of zeros will be assigned for that operand.

FILL - Fill Memory

(<label>) FILL <expression>,<expression>

The FILL directive causes the assembler to initialize an area of memory with a constant value. The first expression signifies the one byte value to be placed in the memory and the second expression indicates the total number of successive bytes to be initialized. The first expression must evaluate to the range 0-255. Expressions cannot contain forward references or undefined symbols.

ORG - Set Program Counter to Origin

ORG <expression> (<comment>)

The ORG directive changes the program counter to the value specified by the expression in the operand field. Subsequent statements are assembled into memory locations starting with the new program counter value. If no ORG directive is encountered in a source program, the program counter is initialized to zero. Expressions cannot contain forward references or undefined symbols.

PAGE - Top of Page

PAGE

The PAGE directive causes the Assembler to advance the paper to the top of the next page. If no source listing is being produced, the PAGE directive will have no effect. The directive is not printed on the source listing.

RMB - Reserve Memory Bytes or DS - DEFINE(reserve) STORAGE

(<label>) RMB <expression> (<comment>)

The RMB directive causes the location counter to be advanced by the value of the expression in the operand field. This directive reserves a block of memory the length of which in bytes is equal to the value of the expression. The block of memory reserved is not initialized to any given value. The expression cannot contain any forward references or undefined symbols. This directive is commonly used to reserve a scratchpad or table area for later use.

ZMB - ZERO MEMORY BYTES (same as BSZ)

(<label>) ZMB <expression> (<comment>)

The ZMB directive causes the Assembler to allocate a block of bytes. Each byte is assigned the initial value of zero. The number of bytes allocated is given by the expression in the operand field. If the expression contains symbols that are either undefined or forward references, or if the expression has a value of zero, an error will be generated.

APPENDIX A

Character Set

The character set recognized by the Assemblers is a subset of ASCII. The ASCII code is shown in the following figure. The following characters are recognized by the Assembler:

1. The upper case letters A through Z and lower case letters a through z.

2. The digits 0 through 9.

3. Five arithmetic operators: +, -, *, / and % (remainder after division).

4. Three logical operators: &, |, and ^.

5. The special symbol characters: underscore (_), period (.),and dollar sign ($). Only the underscore and period may be used as the first character of a symbol.

6. The characters used as prefixes for constants and addressing modes:

#	Immediate addressing
$	Hexadecimal constant
&	Decimal constant
@	Octal constant
%	Binary constant
'	ASCII character constant

7. The characters used as suffixes for constants and addressing modes:

,X	Indexed addressing

8. Three separator characters: space, carriage return, and comma.

9. The character "*" to indicate comments. Comments may contain any printable characters from the ASCII set.

10. The special symbol backslash "\" to indicate line continuation. When the assembler encounters the line continuation character it fetches the next line and adds it to the end of the first line. This continues until a line is seen which doesn't end with a backslash or until the system maximum buffer size has been collected (typically greater or equal to 256).

ASCII CHARACTER CODES

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SPC	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

APPENDIX B

Addressing Modes

M68HC11 Addressing Modes.

Prebyte

The number of combinations of instructions and addressing modes for the 68HC11 is larger than that possible to be encoded in an 8-bit word (256 combinations). To expand the opcode map, certain opcodes ($18, $1A, and $CD) cause the processor to fetch the next address to find the actual instruction. These opcodes are known as prebytes and are inserted automatically by the assembler for those instructions that require it.l In general the instructions contained in the alternate maps are those involving the Y register or addressing modes that involve the Y index register. Thus the programmer make the tradeoff between the convenience of using the second index register and the additional time and code space used by the prebyte.

Inherent or Accumulator Addressing

The M68HC11 includes some instructions which require no operands. These instructions are self-contained, and employ the inherent addressing or the accumulator addressing mode.

Immediate Addressing

Immediate addressing refers to the use of one or more bytes of information that immediately follow the operation code in memory. Immediate addressing is indicated by preceding the operand field with the pound sign or number sign character (#). The expression following the # will be assigned one byte of storage.

Relative Addressing

Relative addressing is used by branch instructions. Branches can only be executed within the range -126 to +129 bytes relative to the first byte of the branch instruction. For this mode, the programmer specifies the branch address expression and places it in the operand field. The actual branch offset is calculated by the assembler and put into the second byte of the branch instruction. The offset is the two's complement of the difference between the location of the byte immediately following the branch instruction and the location of the destination of the branch. Branches out of bounds are flagged as errors by the assembler.

Indexed Addressing

Indexed addressing is relative one of the index registers X or Y. The address is calculated at the time of instruction execution by adding a one-byte displacement to the current contents of the X register. The displacement immediately follows the operation code in memory. If the displacement is zero, zero resides in the byte following the opcode. Since no sign extension is performed on a one-byte displacement, the offset cannot be negative. Indexed addressing is indicated by the characters ",X" following the expression in the operand field. The special case of ",X", without a preceding expression, is treated as "0,X".

Direct and Extended Addressing

Direct and extended addressing utilize one (direct) or two (extended) bytes to contain the address of the operand. Direct addressing is limited to the first 256 bytes of memory. Direct and extended addressing are indicated by only having an expression in the operand field. Direct addressing will be used by the Assembler whenever possible.

BIT(S) SET or CLEAR

The addressing mode used for this type of instruction is direct, although the format of the operand field is different from the direct addressing mode described above. The operand takes the form <expression 1> <expression 2> where the two expressions are separated by a blank.

<expression 1> signifies the operand address and may be either a direct or an indexed address. When the address mode is indexed, <expression 1> is followed by ',R' where R is either X or Y. This allows bit manipulation instructions to operate across the complete 64K address map. <expression 2> is the mask byte. The bit(s) to be set or cleared are indicated by ones in the corresponding location(s) in the mask byte. The mask byte must be an expression in the range 0-255 and is encoded by the programmer.

BIT TEST and BRANCH

This combines two addressing modes: direct or indexed and relative. The format of the operand is: <expression 1> <expression 2> <expression 3> where the expressions are separated by blanks. <expression 1> identifies the operand an may indicate either a direct or indexed address. Indexed addresses are signified with ',R' following the expression where R is either X or Y. <expression 2> is the mask byte. The bit(s) to be set or cleared are indicated by ones in the corresponding location(s) in the mask byte. The mask byte must be an expression in the range 0-255 and is encoded by the programmer. <expression 3> is used to generate a relative address, as described above in "relative addressing".

APPENDIX C

Directive Summary

A complete description of all directives appears in Chapter 3.

ASSEMBLY CONTROL

ORG Origin program counter

SYMBOL DEFINITION

EQU Assign permanent value

DATA DEFINITION/STORAGE ALLOCATION

BSZ Block storage of zero; single bytes

FCB Form constant byte

FCC Form constant character string

FDB Form constant double byte

FILL Initialize a block of memory to a constant

RMB Reserve memory; single bytes

ZMB Zero Memory Bytes; same and BSZ

APPENDIX D

Assembler Listing Format

The Assembler listing has the following format:

```
LINE#   ADDR OBJECT CODE BYTES          [ # CYCLES] SOURCE LINE
```

The LINE# is a 4 digit decimal number printed as a reference. This reference number is used in the cross reference. The ADDR is the hex value of the address for the first byte of the object code for this instruction.

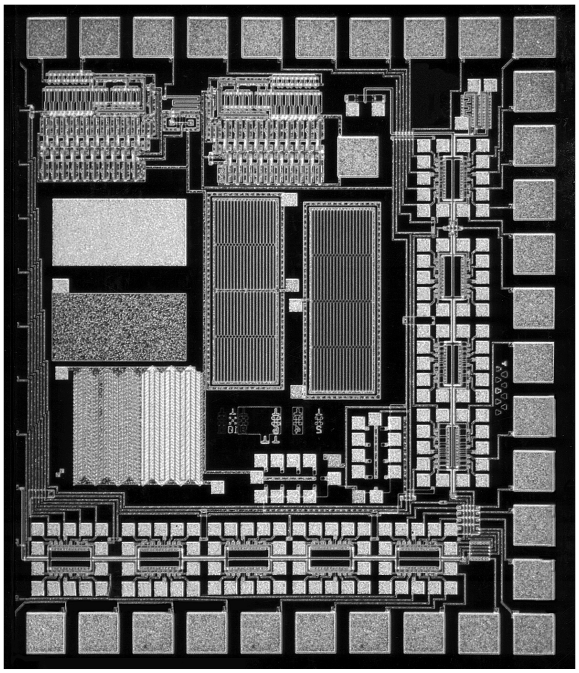
The OBJECT CODE BYTES are the assembled object code of the source line in hex. If an source line causes more than 6 bytes to be output (e.g. a long FCC directive), additional bytes (up to 64) are listed on succeeding lines with no address preceding them.
The # CYCLES will only appear in the listing if the "c" option is in effect. It is enclosed in brackets which helps distinguish it from the source listing. The SOURCE LINE is reprinted exactly from the source program, including labels. The symbol table has the following format:

```
SYMBOLADDR
```

The symbol is taken directly from the label field in the source program. The ADDR is the hexadecimal address of the location referenced by the symbol.
The cross reference listing has the following format:

```
SYMBOLADDR   *LOC1 LOC2 LOC3 ...
```

The SYMBOL and ADDR are the same as above. The * indicates the start of the line reference numbers. The LOCs are the decimal line numbers of the assembler listing where the label occurs.



Lesson Thirteen

Monitor Commands

Monitor Firmware on EZ MICRO-11 Tutor Boards

The monitor program in the EPROM assists in the development of embedded software for, in this case, the 68HC11. The program resides in the target ROM and assists in dowloading, executing and debugging software. Once past the development and debug of a product, the monitor software would be removed and the code modified to run automatically without the monitor.

Among the functions the Buffalo Monitor performs are:

Loading programs
Executing programs
Viewing memory
Changing memory
Setting breakpoints
Single-Stepping through code.

To start using the Buffalo monitor directly (versus using higher-level tools that come with our development board) we can send and receive data through the RS-232 port. The Buffalo Monitor monitors this line and processes characters read from it. Follow the following steps to setup the RS-232 port and access the Buffalo Monitor software.

1. Launch HyperTerminal from Windows (hypertrm.exe). This would typically be under the Communications menu. If you don't have it installed, you may need to install it from your Windows disk(s).
2. It will ask for a name for the new connection. Call it a name you'll remember as being a direct RS-232 connection, such as "RS232Con". Note that the type of connection you'll be setting up is not specific to the Buffalo monitor. It's simply an RS-232 connection.
3. In the "Connect To" window, choose "Direct to Comm 1" or whatever communication port you're using.
4. For the communication options, choose 9600 baud, 8 data bits, no parity, 1 stop bit and Hardware flow control.
5. As for advanced options, this can be hardware dependent. On my computer, I use the FIFO buffers option, have the transmit **buffer** set to it's highest setting (16) and the receive buffer not quite as high (14). This seems to work well.
6. Within the HyperTerminal window, hit <Enter>, and you should see a ">" prompt. Hit <Enter> again and you should see a listing of Buffalo Monitor commands.
7. Choose "File -> Save As", and save your configuration so it's available next time.

Buffalo Monitor Commands

By hitting <Enter> at the Buffalo Monitor prompt, you see a list of commands. Case is not important and the commands shown are:

Command	Options	Description
ASM	[]	Line assembler/disassembler
	/	Do same address
	^	Do previous address
	<Ctrl> J	Do next address
	<Return>	Do next opcode
	<Ctrl> A	Quit
BF	<addr1> <addr2> [<data>]	Block fill
BR	[-] [<addr>]	Setup breakpoint table
BULK		Erase the EEPROM
BULKALL		Erase the EEPROM and CONFIG
CALL	[<addr>]	Call user subroutine
EEMOD	[<addr>] [<addr>]	Modify EEPROM range
GO	[<addr>]	Execute user code
LOAD, VERIFY	[T] <host download command>	Load or verify S-records
MD	[<addr1> <addr2>]	Memory dump
MM	[<addr>]	Memory modify
	/	Open same address
	<Ctrl> H or ^	Open previous address
	<Ctrl> J or <Space>	Open next address
	<Return>	Quit
	<addr>O	Compute offset to <addr>
MOVE	<s1> <s2> [<d>]	Block move
OFFSET	[-] <arg>	Offset for download
PROCEED		Proceed/continue execution
RM	[P, Y, X, A, B, C or S]	Register modify
STOPAT	<addr>	Trace until specified address
T	[<n>]	Trace n instructions
TM		Transparent mode
	<Ctrl> A	Exit
	<Ctrl> B	Send break
<Ctrl> H		Backspace
<Ctrl> W		Wait for any key
<Ctrl> X or <Delete>		Abort/cancel command
<Enter>		Repeat last command

Downloading S-Records using Hyperterminal and the Buffalo Monitor

Downloading S-records uses the "Load" command. The procedure is as follows:

1. At the Buffalo monitor prompt, type "Load T" to enter terminal download moad. The microboard will "wait" for the S-record to come in the RS-232 port.

2. From the pulldown menus on HyperTerminal, choose Transfer->Send Text File. Choose the file to download. Note that the default "*.TXT" list of files will not include S19 files, so change this to "*.*" (all files).

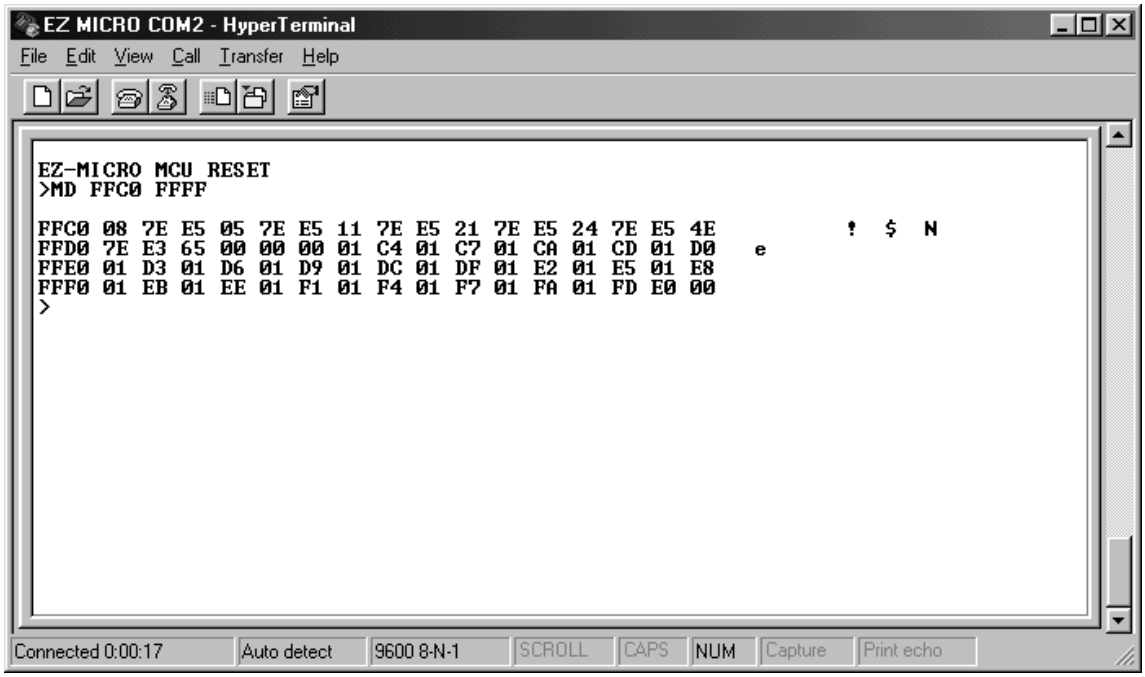
3. Choose "Open" to send the file. The Buffalo monitor should response with a "done". Note: do not hit <Enter> after the "done". An <Enter> is a command to the monitor to repeat the last command. We don't want to download a 2nd time.Apparently, the download can hang occassionaly after the download is complete. If this happens (you should see the "done" in a few seconds) press the Reset button on the board. This will no clear the RAM you just downloaded the code to, so you don't need to download again.

4. To execute the code, type "G 2000" to start execution at address 2000, presuming you're using the CRT.S file I supplied which places the code at 2000.The Buffalo Monitor and Interrupts

The 68HC11 is the same microcontroller regardless of whether it has a Buffalo Monitor or not. Therefore, for the Buffalo Monitor to work it must work within the design of this microcontroller. For example, when a power-up reset occurs, the microcontroller will always vector to the address defined for the Power-Up Reset interrupt. The Buffalo Monitor cannot change this; it's how the 68HC11 is designed. For the Buffalo Monitor to take control of the 68HC11, it must either put code in this address location, or change this interrupt's vector address.

The first thing we need to figure out is how the Buffalo monitor uses the 68HC11 registers and interrupt vectors to control the microcontroller. The Buffalo Montior is memory-mapped to addresses E000 through FFC0 (all addresses in this write-up are in Hex). The Interrupt Vector Table is mapped from FFC0 to FFFF, although FFC0 through FFD5 is reserved. By doing a memory dump of this table:

> md FFC0 FFFF

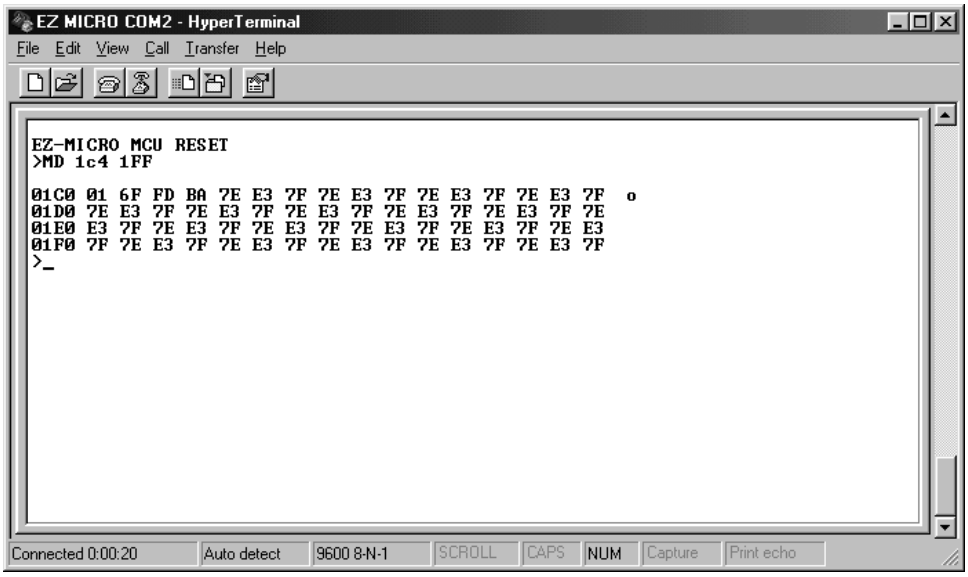


From this, we see the following Interrupt Vectors are setup:

Interrupt	Location	Memory Value
SCI Serial System	FFD6, FFD7	01C4
SPI Serial Trans Comp	FFD8, FFD9	01C7
Pulse Accum Input Edge	FFDA, FFDB	01CA
Pulse Accum Overflow	FFDC, FFDD	01CD
Timer Overflow	FFDE, FFDF	01D0
Timer IC 4 / OC 5	FFE0, FFE1	01D3
Timer OC 4	FFE2, FFE3	01D6
Timer OC 3	FFE4, FFE5	01D9
Timer OC 2	FFE6, FFE7	01DC
Timer OC 1	FFE8, FFE9	01DF
Timer IC 3	FFEA, FFEB	01E2
Timer IC 2	FFEC, FFED	01E5
Timer IC 1	FFEE, FFEF	01E8
Real Time Interrupt	FFF0, FFF1	01EB

Parallel I/O Handshake/IRQ	FFF2, FFF3	01EE
XIRQ Pin	FFF4, FFF5	01F1
Software Interrupt	FFF6, FFF7	01F4
Illegal Opcode Trap	FFF8, FFF9	01F7
COP Failure	FFFA, FFFB	01FA
COP Clock Monitor Fail	FFFC, FFFD	01FD
Power-Up / Reset	FFFE, FFFF	E000

All of the addresses are in the range from 0x01C4 to 0x01FD. Using the same memory dump command to see this section of memory (plus a couple bytes beyond 0x00FD) using:
> md 1c4 1ff



Looking at this output, we see that the vector table addresses jump to an instruction 7E (the microcontroller always executes the byte at the jump location so it must be an instruction). 7E is a jump instruction, so 7E E3 71 (the bytes at the address 1C0) is a "Jump E371" instruction. The result of this is to perform a jump to E371, not 01C4, the address in the vector table.

The reason this is done is because the Buffalo Monitor must control some of the interrupt addresses (such as power-up) so it sets the addresses it needs to the appropriate code in the Buffalo Monitor, and sends the rest to a section in RAM where we can edit the addresses to be what we want. Because the Buffalo Montior is in ROM, we cannot change the desired Interrupt Vector addresses directly.

If you try to modify the Interrupt Vector Table addresses, you won't be able to because the device that contains the Buffalo Monitor and the Vector Table is in ROM. To set-up an ISR, you need to change the address at the location the Vector Table specifies, accounting for the first jump instruction. So, the memory from 00C4 to 00FF (00FD + 2 bytes for the jump address) is used for the "RAM-based Vector table" and cannot be used in our code.

For example, the Real-Time Interrupt (RTI) jumps to 00EB, which then jumps to E371. You'll notice that all these interrupts except for the power-up reset jump to E371. This is in the Buffalo Monitor memory and is undoubtedly a "do nothing" routine that simply returns. If you want to

write your own interrupt handler for the RTI, you must put the address of your handler in place of E371 in bytes 00EC and 00ED.

It's clear that the Buffalo Monitor makes some assumptions about how it can use the memory. To make sure we don't "step on its toes" when running our programs we need to be aware of what memory we can use, and what the Buffalo Monitor uses.

Buffalo Monitor Functions

The Buffalo Monitor contains functions that we can use to input and output data. The table below lists the available functions and what address to call to execute them. For more detail, see the Buffalo Monitor manual.

Function	Address	Description
UPCASE	FFA0	Convert character to uppercase
WCHEK	FFA3	Test character for whitespace
DCHEK	FFA6	Check character for delimiter
INIT	FFA9	Initialize I/O device
INPUT	FFAC	Read I/O device
OUTPUT	FFAF	Write I/O device
OUTLHLF	FFB2	Convert left nibble to ASCII and output
OUTRHLF	FFB5	Convert right nibble to ASCII and output
OUTA	FFB8	Output ASCII character
OUTlBYT	FFBB	Convert binary byte to 2 ASCII characters and output
OUT1BSP	FFBE	Convert binary byte to 2 ASCII characters and output followed by space
OUT2BSP	FFCl	Convert 2 consecutive binary bytes to 4 ASCII characters and output followed by space
OUTCRLF	FFC4	Output ASCII carriage return followed by line feed
OUTSTRG	FFC7	Output ASCII string until end of transmission ($04)
OUTSTRGO	FFCA	Same as OUTSTRG except leading carriage return and line fees is skipped
INCHAR	FFCD	Input ASCII character and echo back

By calling the specified function, we can use these to input and output data.

Debugging with the Buffalo Monitor

Several of the commands in the Buffalo Monitor can be used to help debug software. However, care must be taken to not use resources in our software that the Buffalo Monitor uses when implementing these commands. The Buffalo Monitor manual gives a detailed analysis of all commands and what resources they use. Here, I'll walk through an example debugging effort using some of these commands.

It's common for software to "break" in some manner that is hard to track down. One example is when the software "hangs". When this happens, there's no feedback to give hints as to what the software is doing.

One option is to sprinkle "printf's" (in our case, PutStrRs232) to give us some history, but this changes the code and can cause the problem to go away or manifest itself differently. This is precisely what happened when I was trying to debug such an error. Adding PutStrRs232's caused the program to run, but without them, it just hung. Also, printf's can also be a slow process.

Instead, we can use the Buffalo Monitor to track program execution. A few key commands will be needed. We need to be able to set a breakpoint, single step through code and observe variables. We need to know the address of such things, so your compile option must have generated a listing of some type (LST file) that gives this information. While debugging, you'll be looking at this listing repeatedly to figure out where to set the next breakpoint, where you're at in the execution, etc.

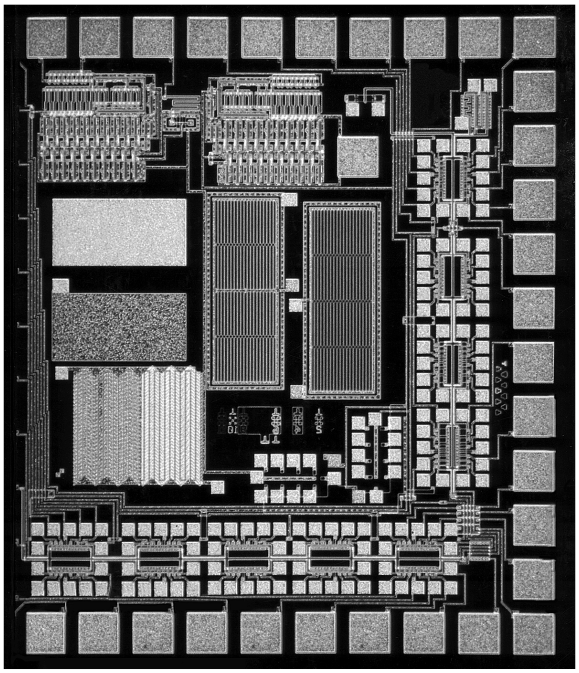
We launch the program using the "Go" command, with an address. This sets the program counter to the address we specify and starts execution. To keep the program from continuing indefinitely, we first must setup a breakpoint at a desired location. Find the desired location by looking at the output LST file and finding the code that matches the C code where you want to break. Since you don't know where the software is hanging, you may need to do this several times, putting breakpoints in various placed, until you start to locate where the software is hanging. Unfortunately, we don't have a way to intermix the C and assembly code, so this can be cumbersome.

A breakpoint is set by typing "B <address>" at the Buffalo prompt. This sets a breakpoint at the specified address. You can set up to 4 breakpoints. Once you've set 4 of them, you need to clear some. All breakpoints can be cleared by typing "B -" at the prompt. Breakpoints use the Software Interrupt (SWI instruction). Therefore, to use breakpoints our software cannot use any SWIs.

Once we've "breakpointed" at the desired location, we can single step and watch what's happening. This is done using the trace command "T". The trace command uses XIRQ and OC5 interrupts, so these must also be available. Note that the Buffalo monitor uses OC5 to generate an XIRQ interrupt, so the output of OC5, Port A pin 3, must be tied to the XIRQ pin. This is already done on the EVBU board.

By executing "T" repeatedly, you can single step through the code and see where the code is going. However, the Main Timer is still running even when execution is stopped. Since the 68HC11 inhibits interrupts when one occurs, higher-priority interrupts shouldn't cause a problem, but when you single step the interrupts are enabled for a short time.
Care must be taken when debugging software in this way.

Reference Motorola Manual
Web site of Prof R J Weber University of California



Lesson Fourteen

6811 Commands by Subject

Loads, Stores, and Transfers

		Addressing Mode [Cycles to complete]						
Function	**Mnemonic**	**REL**	**IMM**	**DIR**	**EXT**	**INDX**	**INDY**	**INH**
Clear Memory Byte	CLR	-	-	-	6	6	7	-
Clear Accumulator A	CLRA	-	-	-	-	-	-	2
Clear Accumulator B	CLRB	-	-	-	-	-	-	2
Load Accumulator A	LDAA	-	2	3	4	4	5	-
Load Accumulator B	LDAB	-	2	3	4	4	5	-
Load Double Accumulator D	LDD	-	3	4	5	5	6	-
Pull A from Stack	PULA	-	-	-	-	-	-	4
Pull B from Stack	PULB	-	-	-	-	-	-	4
Push A onto Stack	PSHA	-	-	-	-	-	-	3
Push B onto Stack	PSHB	-	-	-	-	-	-	3
Store Accumulator A	STAA	-	-	3	4	4	5	-
Store Accumulator B	STAB	-	-	3	4	4	5	-
Store Double Accumulator D	STD	-	-	4	5	5	6	-
Store Double Accumulator D	STD	-	-	4	5	5	6	-
Transfer A to B	TAB	-	-	-	-	-	-	2
Transfer A to CCR	TAP	-	-	-	-	-	-	2
Transfer B to A	TBA	-	-	-	-	-	-	2
Transfer CCR to A	TPA	-	-	-	-	-	-	2
Exchange D with X	XGDX	-	-	-	-	-	-	3
Exchange D with Y	XGDY	-	-	-	-	-	-	4

Arithmetic Operations

		Addressing Mode [Cycles to complete]						
Function	**Mnemonic**	**REL**	**IMM**	**DIR**	**EXT**	**INDX**	**INDY**	**INH**
Add Accumulators	ABA	-	-	-	-	-	-	2
Add Accumulator B to X	ABX	-	-	-	-	-	-	3
Add Accumulator B to Y	ABY	-	-	-	-	-	-	4
Add with Carry to A	ADCA	-	2	3	4	4	5	-
Add with Carry to B	ADCB	-	2	3	4	4	5	-
Add Memory to A	ADDA	-	2	3	4	4	5	-
Add Memory to B	ADDB	-	2	3	4	4	5	-
Add Memory to D (16 Bit)	ADDD	-	4	5	6	6	7	-
Compare A to B	CBA	-	-	-	-	-	-	2
Compare A to Memory	CMPA	-	2	3	4	4	5	-
Compare B to Memory	CMPB	-	2	3	4	4	5	-
Compare D to Memory (16 Bit)	CPD	-	5	6	7	7	7	-
Decimal Adjust A (for BCD)	DAA	-	-	-	-	-	-	2
Decrement Memory Byte	DEC	-	-	-	6	6	7	-
Decrement Accumulator A	DECA	-	-	-	-	-	-	2
Decrement Accumulator B	DECB	-	-	-	-	-	-	2
Increment Memory Byte	INC	-	-	-	6	6	7	-
Increment Accumulator A	INCA	-	-	-	-	-	-	2
Increment Accumulator B	INCB	-	-	-	-	-	-	2
Twos Complement Memory Byte	NEG	-	-	-	6	6	7	-
Twos Complement Accumulator A	NEGA	-	-	-	-	-	-	2
Twos Complement Accumulator B	NEGB	-	-	-	-	-	-	2
Subtract with Carry from A	SBCA	-	2	3	4	4	5	-
Subtract with Carry from B	SBCB	-	2	3	4	4	5	-
Subtract Memory from A	SUBA	-	2	3	4	4	5	-
Subtract Memory from B	SUBB	-	2	3	4	4	5	-
Subtract Memory from D (16 Bit)	SUBD	-	4	5	6	6	7	-
Test for Zero or Minus	TST	-	-	-	6	6	7	-
Test for Zero or Minus A	TSTA	-	-	-	-	-	-	2
Test for Zero or Minus B	TSTB	-	-	-	-	-	-	2
Multiply (D = A x B)	MUL	-	-	-	-	-	-	10
Fractional Divide [X = (D/X) ; D = remainder]	FDIV	-	-	-	-	-	-	41
Integer Divide [X = (D/X) ; D = remainder]	IDIV	-	-	-	-	-	-	41

Logical Operations

		Addressing Mode [Cycles to complete]						
Function	**Mnemonic**	**REL**	**IMM**	**DIR**	**EXT**	**INDX**	**INDY**	**INH**
AND A with Memory	ANDA	-	2	3	4	4	5	-
AND B with Memory	ANDB	-	2	3	4	4	5	-
Bits(s) Test A with Memory	BITA	-	2	3	4	4	5	-
Bits(s) Test B with Memory	BITB	-	2	3	4	4	5	-
Ones Complement Memory Byte	COM	-	-	-	6	6	7	-
Ones Complement A	COMA	-	-	-	-	-	-	2
Ones Complement B	COMB	-	-	-	-	-	-	2
OR A with Memory (Exclusive)	EORA	-	2	3	4	4	5	-
OR B with Memory (Exclusive)	EORB	-	2	3	4	4	5	-
OR A with Memory (Inclusive)	ORAA	-	2	3	4	4	5	-
OR B with Memory (Inclusive)	ORAB	-	2	3	4	4	5	-

Data Testing and Bit Manipulation

		Addressing Mode [Cycles to complete]						
Function	**Mnemonic**	**REL**	**IMM**	**DIR**	**EXT**	**INDX**	**INDY**	**INH**
Bit(s) Test A with Memory	BITA	-	2	3	4	4	5	-
Bit(s) Test B with Memory	BITB	-	2	3	4	4	5	-
Clear Bit(s) in Memory	BCLR	-	-	6	-	7	8	-
Set Bit(s) in Memory	BSET	-	-	6	-	7	8	-
Branch if Bit(s) Clear	BRCLR	-	-	6	-	7	8	-
Branch if Bit(s) Set	BRSET	-	-	6	-	7	8	-

Shifts and Rotates

		Addressing Mode [Cycles to complete]						
Function	**Mnemonic**	**REL**	**IMM**	**DIR**	**EXT**	**INDX**	**INDY**	**INH**
Arithmetic Shift Left Memory	ASL	-	-	-	6	6	7	-
Arithmetic Shift Left A	ASLA	-	-	-	-	-	-	2
Arithmetic Shift Left B	ASLB	-	-	-	-	-	-	2
Arithmetic Shift Left Double	ASLD	-	-	-	-	-	-	3
Arithmetic Shift Right Memory	ASR	-	-	-	6	6	7	-
Arithmetic Shift Right A	ASRA	-	-	-	-	-	-	2
Arithmetic Shift Right B	ASRB	-	-	-	-	-	-	2
(Logical Shift Left Memory)	(LSL)	-	-	-	6	6	7	-
(Logical Shift Left A)	(LSLA)	-	-	-	-	-	-	2
(Logical Shift Left B)	(LSLB)	-	-	-	-	-	-	2
(Logical Shift Left Double)	(LSLD)	-	-	-	-	-	-	3
Logical Shift Right Memory	LSL	-	-	-	6	6	7	-
Logical Shift Right A	LSRA	-	-	-	-	-	-	2
Logical Shift Right B	LSRB	-	-	-	-	-	-	2
Logical Shift Right Double	LSLD	-	-	-	-	-	-	3
Rotate Left Memory	ROL	-	-	-	6	6	7	-
Rotate Left A	ROLA	-	-	-	-	-	-	2
Rotate Left B	ROLB	-	-	-	-	-	-	2
Rotate Right Memory	ROR	-	-	-	6	6	7	-
Rotate Right A	RORA	-	-	-	-	-	-	2
Rotate Right B	RORB	-	-	-	-	-	-	2

As mentioned in the M68HC11 Reference Manual, the logical-left-shift instructions are shown in parentheses because there is not difference between an arithmetic and a logical left shift. The mnemonics are recognized by the assembler as equivalent.

Stack and Index Register Instructions

		Addressing Mode [Cycles to complete]						
Function	**Mnemonic**	**REL**	**IMM**	**DIR**	**EXT**	**INDX**	**INDY**	**INH**
Add Accumulator B to X	ABX	-	-	-	-	-	-	3
Add Accumulator B to Y	ABY	-	-	-	-	-	-	4
Compare X to Memory (16 Bit)	CPX	-	4	5	6	6	7	-
Compare Y to Memory (16 Bit)	CPY	-	4	5	6	6	7	-
Decrement Stack Pointer	DES	-	-	-	-	-	-	3
Decrement Index Register X	DEX	-	-	-	-	-	-	3
Decrement Index Register Y	DEY	-	-	-	-	-	-	4
Increment Stack Pointer	INS	-	-	-	-	-	-	3
Increment Index Register X	INX	-	-	-	-	-	-	3
Increment Index Register Y	INY	-	-	-	-	-	-	4
Load Index Register X	LDX	-	3	4	5	5	6	-
Load Index Register Y	LDY	-	4	5	6	6	6	-
Load Stack Pointer	LDS	-	3	4	5	5	6	-
Pull X from Stack	PULX	-	-	-	-	-	-	5
Pull Y from Stack	PULY	-	-	-	-	-	-	6
Push X onto Stack	PSHX	-	-	-	-	-	-	4
Push Y onto Stack	PSHY	-	-	-	-	-	-	5
Store Index Register X	STX	-	-	4	5	5	6	-
Store Index Register Y	STY	-	-	5	6	6	6	-
Store Stack Pointer	STS	-	-	4	5	5	6	-
Transfer SP to X	TSX	-	-	-	-	-	-	3
Transfer SP to Y	TSY	-	-	-	-	-	-	4
Transfer X to SP	TXS	-	-	-	-	-	-	3
Transfer Y to SP	TYS	-	-	-	-	-	-	4
Exchange D with X	XGDX	-	-	-	-	-	-	3
Exchange D with Y	XGDY	-	-	-	-	-	-	4

Condition Code Register Instructions

		Addressing Mode [Cycles to complete]						
Function	**Mnemonic**	**REL**	**IMM**	**DIR**	**EXT**	**INDX**	**INDY**	**INH**
Clear Carry Bit	CLC	-	-	-	-	-	-	2
Clear Interrupt Mask Bit	CLI	-	-	-	-	-	-	2
Clear Overflow Bit	CLV	-	-	-	-	-	-	2
Set Carry Bit	SEC	-	-	-	-	-	-	2
Set Interrupt Mask Bit	SEI	-	-	-	-	-	-	2
Set Overflow Bit	SEV	-	-	-	-	-	-	2
Transfer A to CCR	TAP	-	-	-	-	-	-	2
Transfer CCR to A	TPA	-	-	-	-	-	-	2

Branches, Jumps, and Subroutines

		Addressing Mode [Cycles to complete]						
Function	**Mnemonic**	**REL**	**IMM**	**DIR**	**EXT**	**INDX**	**INDY**	**INH**
Branch if Carry Clear [C = 0 ?]	BCC	3	-	-	-	-	-	-
Branch if Carry Set [C = 1 ?]	BCS	3	-	-	-	-	-	-
Branch if Equal Zero [Z = 1 ?]	BEQ	3	-	-	-	-	-	-
Branch if Great Than or Equal [Signed >=]	BGE	3	-	-	-	-	-	-
Branch if Great Than [Signed >]	BGT	3	-	-	-	-	-	-
Branch if Higher [Unsigned >]	BHI	3	-	-	-	-	-	-
Branch if Higher or Same (same as BCC) [Unsigned >=]	BHS	3	-	-	-	-	-	-
Branch if Less Than or Equal [Signed <=]	BLE	3	-	-	-	-	-	-
Branch if Lower (same as BCS) [Unsigned <]	BLO	3	-	-	-	-	-	-
Branch if Lower or Same [Unsigned <=]	BLS	3	-	-	-	-	-	-
Branch if Less Than [Signed <]	BLT	3	-	-	-	-	-	-
Branch if Minus [N = 1 ?]	BMI	3	-	-	-	-	-	-
Branch if Not Equal [Z = 0 ?]	BNE	3	-	-	-	-	-	-
Branch if Plus [N = 0 ?]	BPL	3	-	-	-	-	-	-
Branch if Bit(s) Clear in Memory Byte [Bit Manipulation]	BRCLR	-	-	6	-	7	8	-
Branch Never [3-cycle NOP]	BRN	3	-	-	-	-	-	-
Branch if Bit(s) Set in Memory Byte [Bit Manipulation]	BRSET	-	-	6	-	7	8	-
Branch if Overflow Clear	BVC	3	-	-	-	-	-	-
Branch if Overflow Set	BVS	3	-	-	-	-	-	-
Jump	JMP	-	-	-	3	3	4	-
Branch to Subroutine	BSR	6	-	-	-	-	-	-
Jump to Subroutine	JSR	-	-	5	6	6	7	-
Return from Subroutine	RTS	-	-	-	-	-	-	5

Interrupt Handling

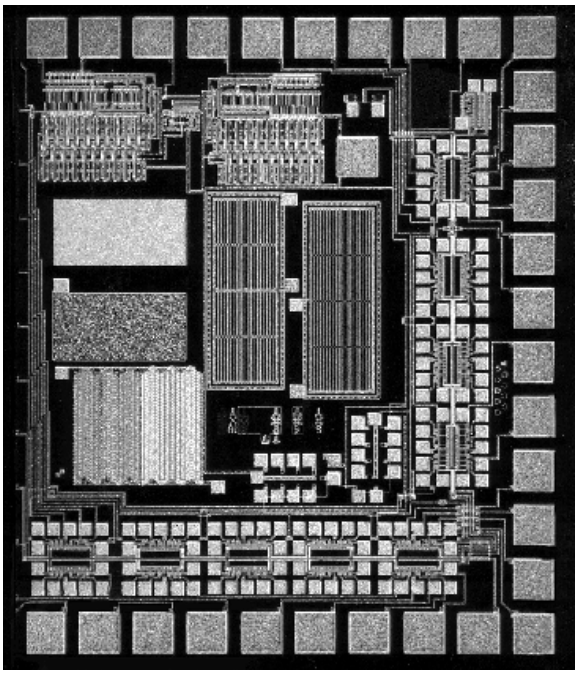
		Addressing Mode [Cycles to complete]						
Function	**Mnemonic**	**REL**	**IMM**	**DIR**	**EXT**	**INDX**	**INDY**	**INH**
Return from Interrupt	RTI	-	-	-	-	-	-	12
Software Interrupt	SWI	-	-	-	-	-	-	14
Wait for Interrupt	WAI	-	-	-	-	-	-	**

As mentioned in the M68HC11 E Series Technical Data Document, 12 cycles are used beginning with the opcode fetch. A wait state is entered which remains in effect for an integer number of MPU E-Clock cycles (n) until an interrupt is recognized. Finally, two additional cycles are used to fetch the appropriate interrupt vector (14 + n total).

Miscellaneous Instructions

		Addressing Mode [Cycles to complete]						
Function	**Mnemonic**	**REL**	**IMM**	**DIR**	**EXT**	**INDX**	**INDY**	**INH**
No Operation (2-cycle delay)	NOP	-	-	-	-	-	-	2
Stop Clocks	STOP	-	-	-	-	-	-	2
Test	TEST	-	-	-	-	-	-	**

**Infinity or until reset occurs.



Appendix

A) References

B) Specfications For Liquid Crystal Display Module

C) LCD Driver

D) MC68HC711E9 Technical Summary

A) References

Using the MC8HC11 microcontroller: a guide to interfacing and programming the M68HC11 microcontroller
Skroder, John C.

ISBN 0-13-120676-1
1997 by Prentice-Hall, Inc.
Simon & Schuster / A Viacom Company
Upper Saddle River, New Jersey 07458

Data acquisition and process control with the M68HC11 microntroller
Frederick F. Driscoll, Robert F. Coughlin, Robert S. Villanucci.

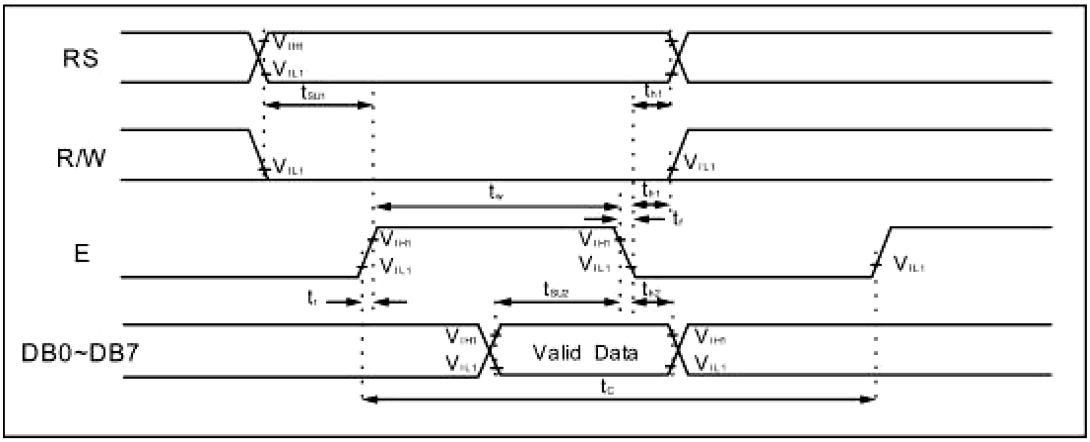
ISBN 0-02-330555-X
1997 by Macmillan Publishing Co, a division of Macmillan, Inc.
Macmillan Publishing Co.
866 Third Avenue, New York, NY 10022

M68HC11 Reference Manual
© Motorola, Inc.

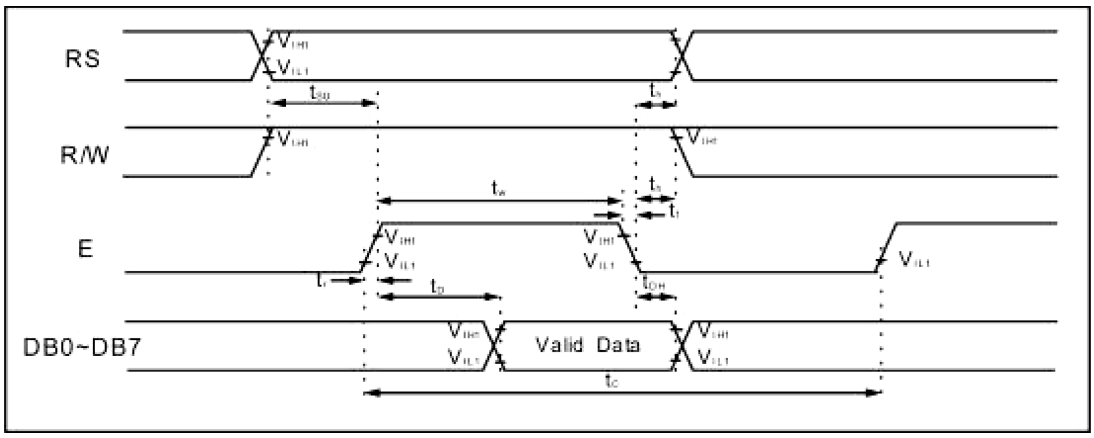
B) Specfications For Liquid Crystal Display Module
Model No: DV-16252-S2RB

Mode	Characteristic	Symbol	Min.	Typ.	Max.	Unit
Write Mode (Refer to Fig-6)	E Cycle Time	tc	500	-	-	ns
	E Rise / Fall Time	t_R,t_F	-	-	20	
	E Pulse Width (High, Low)	tw	230	-	-	
	R/W and RS Setup Time	tsu1	40	-	-	
	R/W and RS Hold Time	t_{H1}	10	-	-	
	Data Setup Time	tsu2	80	-	-	
	Data Hold Time	t_{H2}	10	-	-	
Read Mode (Refer to Fig-7)	E Cycle Time	tc	500	-	-	ns
	E Rise / Fall Time	t_R,t_F	-	-	20	
	E Pulse Width (High, Low)	tw	230	-	-	
	R/W and RS Setup Time	tsu	40	-	-	
	R/W and RS Hold Time	t_H	10	-	-	
	Data Output Delay Time	t_D	-	-	120	
	Data Hold Time	t_{DH}	5	-	-	

Timing Characteristics



Write Mode Timing Diagram



Read Mode Timing Diagram

Appendix C

INSTRUCTION DESCRIPTION

Outline

To overcome the speed difference between the internal clock of KS0066U and the MPU clock, KS0066U performs internal operations by storing control informations to IR or DR. The internal operation is determined according to the signal from MPU, composed of read/write and data bus (Refer to Table 7).
Instructions can be divided largely into four groups:

1) KS0066U function set instructions (set display methods, set data length, etc.)
2) address set instructions to internal RAM
3) data transfer instructions with internal RAM
4) others

The address of the internal RAM is automatically increased or decreased by 1.

Note: During internal operation, Busy Flag (DB7) is read "High".
Busy Flag check must be preceded by the next instruction.
When an MPU program with checking the Busy Flag (DB7) is made, it must be necessary 1/2 fosc for executing the next instruction by the falling edge of the 'E' signal after the Busy Flag (DB7) goes to "Low".

Contents

1) Clear Display

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	0	0	1

Clear all the display data by writing "20H" (space code) to all DDRAM address,
and set DDRAM address to "00H" into AC (address counter).
Return cursor to the original status, namely, bring the cursor to the left edge on the first line of the display.
Make the entry mode increment (I/D = "High").

2) Return Home

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	0	1	-

* "-": dont care

Return Home is cursor return home instruction.
Set DDRAM address to "00H" into the address counter.
Return cursor to its original site and return display to its original status, if shifted.
Contents of DDRAM does not change.

3) Entry Mode Set

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	1	I/D	SH

Set the moving direction of cursor and display.

I/D: Increment / decrement of DDRAM address (cursor or blink)
When I/D = "High", cursor/blink moves to right and DDRAM address is increased by 1.
When I/D = "Low", cursor/blink moves to left and DDRAM address is decreased by 1.
* CGRAM operates the same way as DDRAM, when reading from or writing to CGRAM.

SH: Shift of entire display
When DDRAM read (CGRAM read/write) operation or SH = "Low", shifting of entire display is not performed.
If SH = "High" and DDRAM write operation, shift of entire display is performed according to I/D value
(I/D = "High": shift left, I/D = "Low": shift right).

4) Display ON/OFF Control

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	1	D	C	B

Control display/cursor/blink ON/OFF 1 bit register.

D: Display ON/OFF control bit
When D = "High", entire display is turned on.
When D = "Low", display is turned off, but display data remains in DDRAM.

C: Cursor ON/OFF control bit
When C = "High", cursor is turned on.
When C = "Low", cursor is disappeared in current display, but I/D register preserves its data.

B: Cursor Blink ON/OFF control bit
When B = "High", cursor blink is on, which performs alternately between all the "High" data and display characters at the cursor position.
When B = "Low", blink is off.

5) Cursor or Display Shift

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	1	S/C	R/L	-	-

Shifting of right/left cursor position or display without writing or reading of display data.
This instruction is used to correct or search display data.(Refer to Table 6)
During 2-line mode display, cursor moves to the 2nd line after the 40th digit of the 1st line.
Note that display shift is performed simultaneously in all the lines.
When displayed data is shifted repeatedly, each line is shifted individually.
When display shift is performed, the contents of the address counter are not changed.

Table 6. Shift Patterns According to S/C and R/L Bits

S/C	R/L	Operation
0	0	Shift cursor to the left, AC is decreased by 1
0	1	Shift cursor to the right, AC is increased by 1
1	0	Shift all the display to the left, cursor moves according to the display
1	1	Shift all the display to the right, cursor moves according to the display

6) Function Set

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	DL	N	F	-	-

DL: Interface data length control bit
When DL = "High", it means 8-bit bus mode with MPU.
When DL = "Low", it means 4-bit bus mode with MPU. Hence, DL is a signal to select 8-bit or 4-bit bus mode.
When 4-bit bus mode, it needs to transfer 4-bit data twice.

N: Display line number control bit
When N = "Low", 1-line display mode is set.
When N = "High", 2-line display mode is set.

F: Display font type control bit
When F = "Low", 5 ×8 dots format display mode is set.
When F = "High", 5 ×11 dots format display mode.

7) Set CGRAM Address

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0

Set CGRAM address to AC.
This instruction makes CGRAM data available from MPU.

8) Set DDRAM Address

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0

Set DDRAM address to AC.
This instruction makes DDRAM data available from MPU.
When 1-line display mode (N = Low), DDRAM address is from "00H" to "4FH".
In 2-line display mode (N = High), DDRAM address in the 1st line is from "00H" to "27H", and DDRAM address in the 2nd line is from "40H" to "67H".

9) Read Busy Flag & Address

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0

This instruction shows whether KS0066U is in internal operation or not.
If the resultant BF is "High", internal operation is in progress and should wait until BF is to be Low, which by then the next instruction can be performed. In this instruction you can also read the value of the address counter.

10) Write data to RAM

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	0	D7	D6	D5	D4	D3	D2	D1	D0

Write binary 8-bit data to DDRAM/CGRAM.
The selection of RAM from DDRAM, and CGRAM, is set by the previous address set instruction (DRAM address set, CGRAM address set).
RAM set instruction can also determine the AC direction to RAM.
After write operation, the address is automatically increased/decreased by 1, according to the entry mode.

11) Read data from RAM

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	1	D7	D6	D5	D4	D3	D2	D1	D0

Read binary 8-bit data from DDRAM/CGRAM.
The selection of RAM is set by the previous address set instruction. If the address set instruction of RAM is not performed before this instruction, the data that has been read first is invalid, as the direction of AC is not Yet determined. If RAM data is read several times without RAM address instructions set before read operation, the correct RAM data can be obtained from the second. But the first data would be incorrect, as there is no time margin to transfer RAM data.
In case of DDRAM read operation, cursor shift instruction plays the same role as DDRAM address set instruction, it also transfers RAM data to output data register.
After read operation, address counter is automatically increased/decreased by 1 according to the entry mode.
After CGRAM read operation, display shift may not be executed correctly.

NOTE: In case of RAM write operation, AC is increased/decreased by 1 as in read operation.
At this time, AC indicates the next address position, but only the previous data can be read by the read instruction.

Table 7. Instruction Table

Instruction	Instruction Code										Description	Execution time (fosc= 270 kHz)
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear Display	0	0	0	0	0	0	0	0	0	1	Write "20H" to DDRAM and set DDRAM address to "00H" from AC	1.53 ms
Return Home	0	0	0	0	0	0	0	0	1	-	Set DDRAM address to "00H" from AC and return cursor to its original position if shifted. The contents of DDRAM are not changed.	1.53 ms
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	SH	Assign cursor moving direction and enable the shift of entire display.	39 µs
Display ON/ OFF Control	0	0	0	0	0	0	1	D	C	B	Set display(D), cursor(C), and blinking of cursor(B) on/off control bit.	39 µs
Cursor or Display Shift	0	0	0	0	0	1	S/C	R/L	-	-	Set cursor moving and display shift control bit, and the direction, without changing of DDRAM data.	39 µs
Function Set	0	0	0	0	1	DL	N	F	-	-	Set interface data length (DL: 8-bit/4-bit), numbers of display line (N: 2-line/1-line) and, display font type (F:5×11dots/5×8 dots)	39 µs
Set CGRAM Address	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0	Set CGRAM address in address counter.	39 µs
Set DDRAM Address	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Set DDRAM address in address counter.	39 µs
Read Busy Flag and Address	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Whether during internal operation or not can be known by reading BF. The contents of address counter can also be read.	0 µs
Write Data to RAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write data into internal RAM (DDRAM/CGRAM).	43 µs
Read Data from RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read data from internal RAM (DDRAM/CGRAM).	43 µs

* "-": dont care

NOTE: When an MPU program with checking the Busy Flag(DB7) is made, it must be necessary 1/2Fosc is necessary for executing the next instruction by the falling edge of the 'E' signal after the Busy Flag (DB7) goes to "Low".

INTERFACE WITH MPU

1) Interface with 8-bit MPU

When interfacing data length are 8-bit, transfer is performed at a time through 8 ports, from DB0 to DB7. Example of timing sequence is shown below.

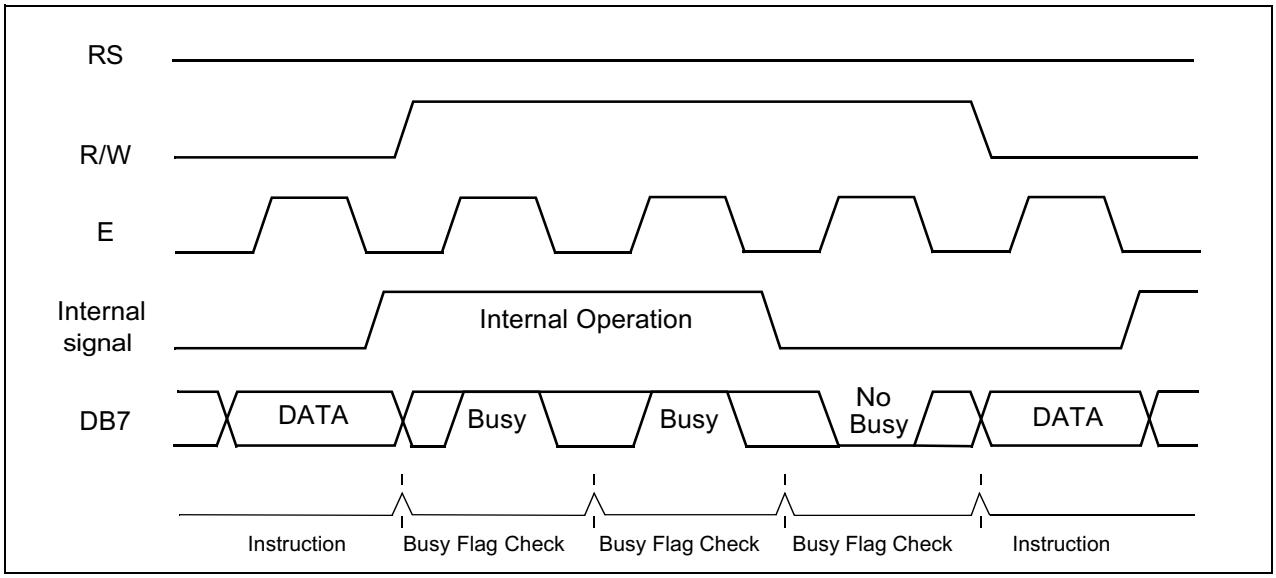


Figure 4 . Example of 8-bit Bus Mode Timing Diagram

2) Interface with 4-bit MPU

When interfacing data length are 4-bit, only 4 ports, from DB4 to DB7, are used as data bus. At First, the higher 4-bit (in case of 8-bit bus mode, the contents of DB4 - DB7), and then the lower 4-bit (in case of 8-bit bus mode, the contents of DB0 - DB3) are transferred. So transfer is performed twice Busy Flag outputs "High" after the second transfer is ended.

Example of timing sequence is shown below.

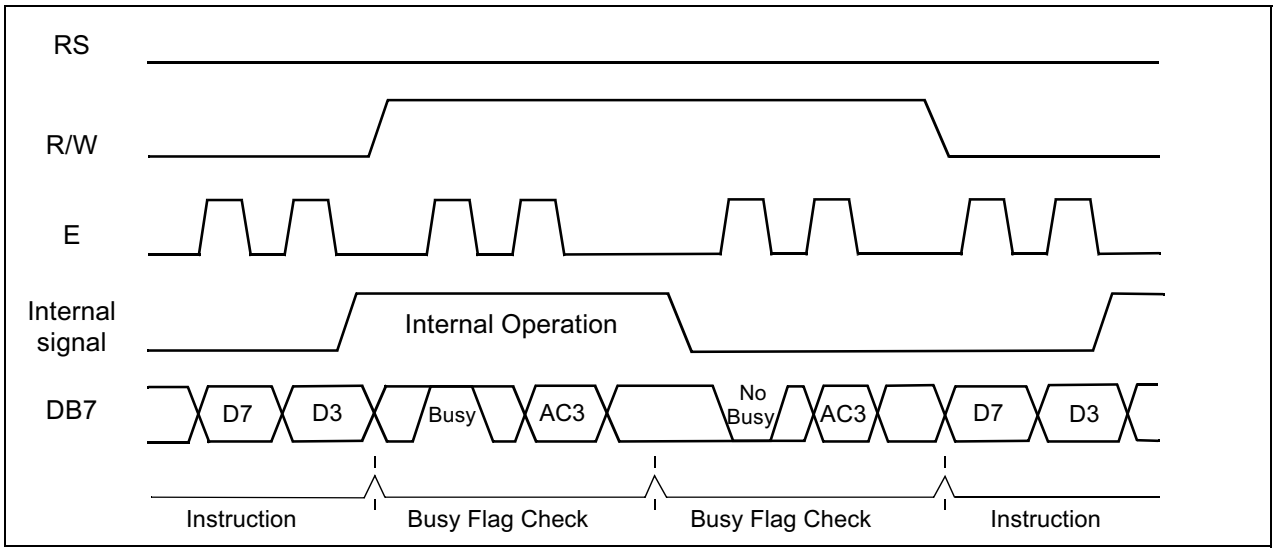
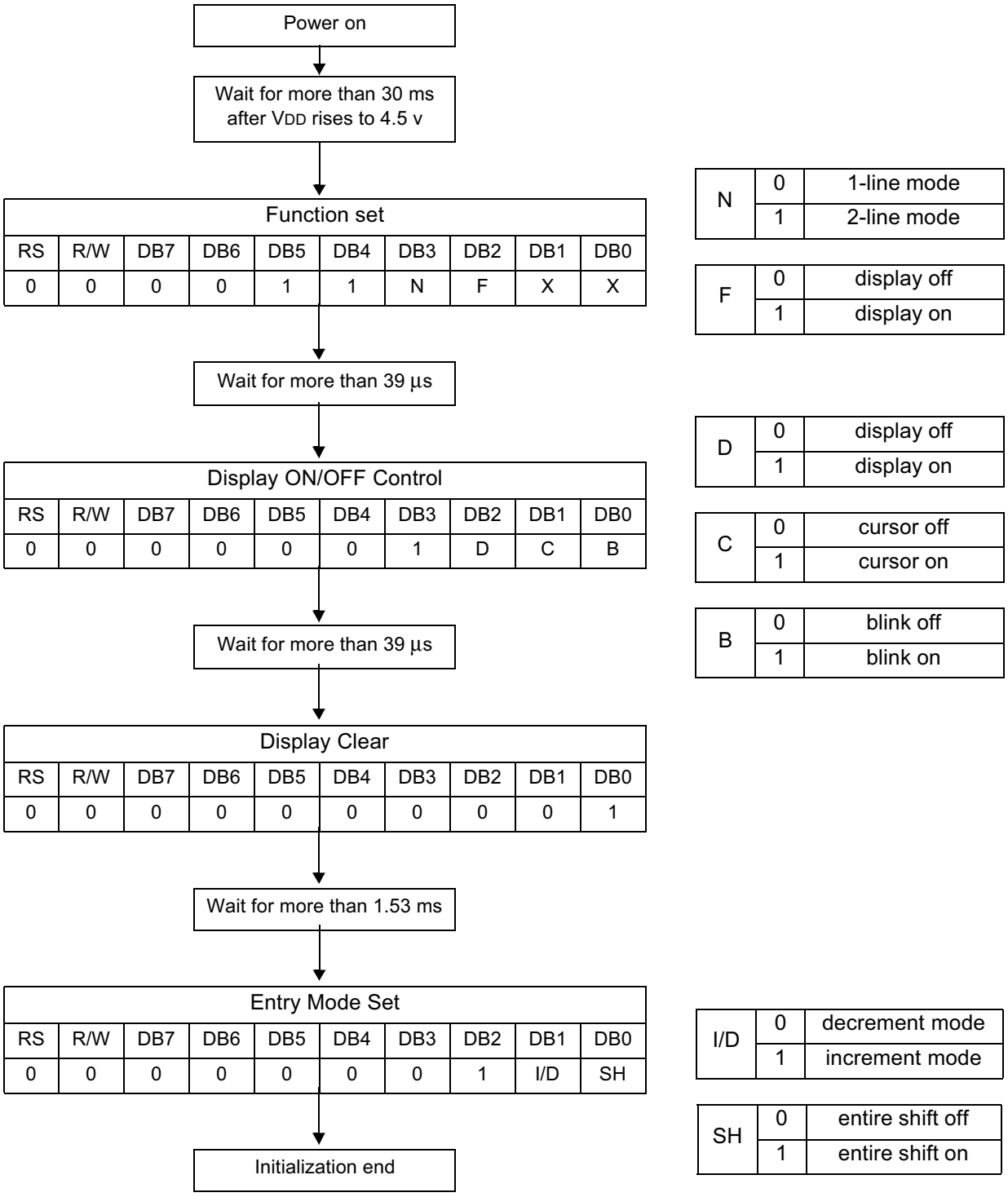


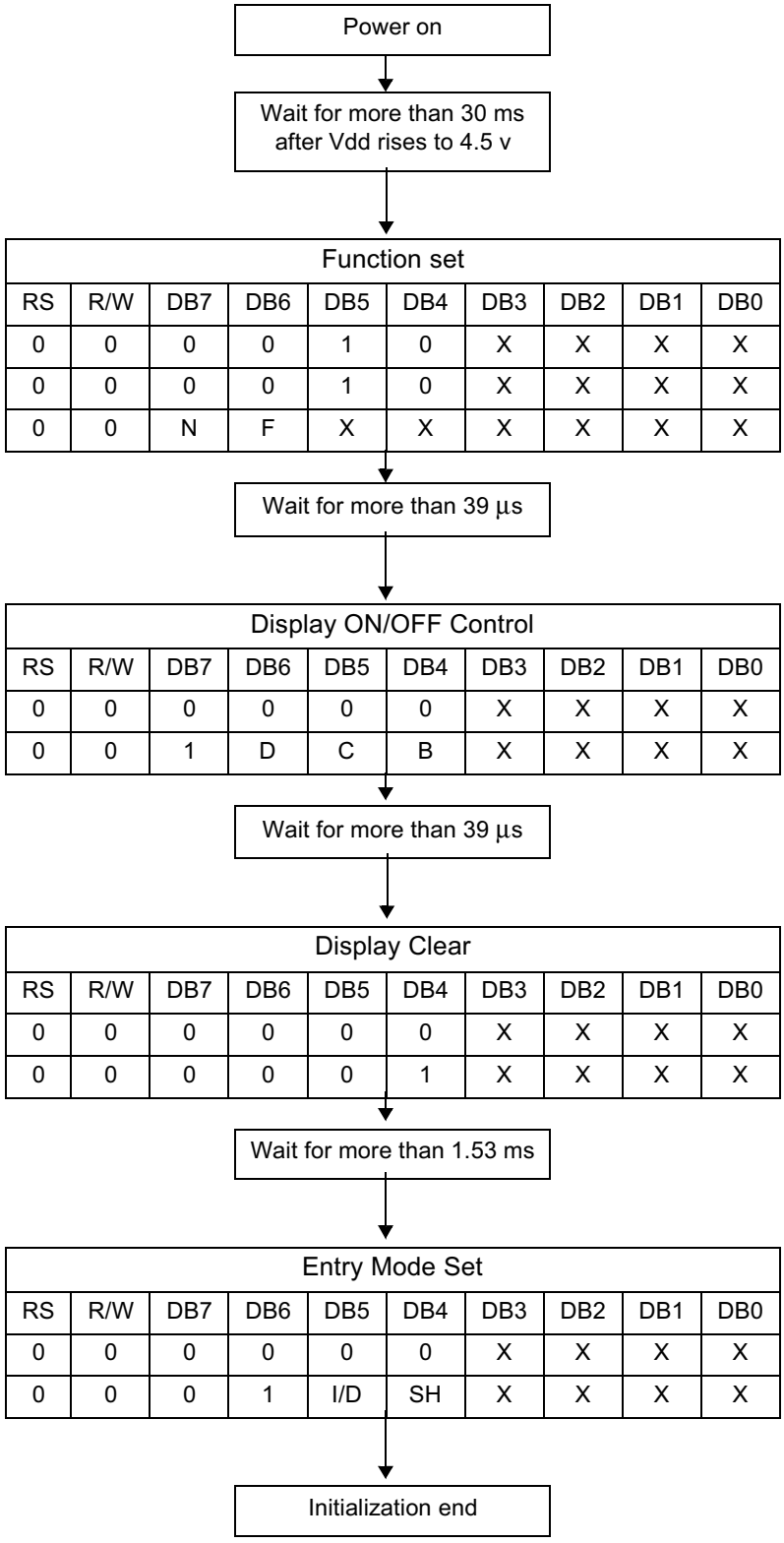
Figure 5 . Example of 4-bit Bus Mode Timing Diagram

INITIALIZING BY INSTRUCTION

1) 8-bit interface mode (Condition: fosc = 270KHZ)



2) 4-bit interface mode (Condition: fosc = 270KHZ)



N	0	1-line mode
	1	2-line mode

F	0	display off
	1	display on

D	0	display off
	1	display on

C	0	cursor off
	1	cursor on

B	0	blink off
	1	blink on

I/D	0	decrement mode
	1	increment mode

SH	0	entire shift off
	1	entire shift on