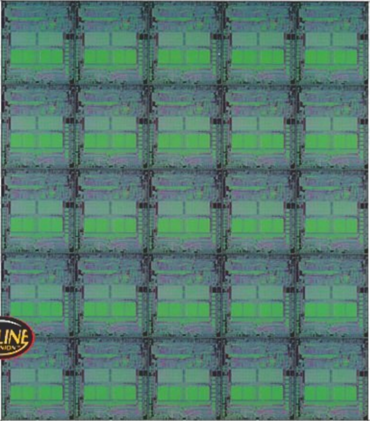
DELMAR
THOMSON LEARNING

Technician's Guide to the 68HC11 Microcontroller



ONLINE COMPANION

Black

chapter 1

Introduction to Computer Hardware

Objectives

After completing this chapter, you should be able to:

- Describe the fundamental elements of every computer system: processor, memory, and input/output
- Compare elements of the HC11 block diagram to the fundamentals of every computer system
- Describe the use of busses to connect computer elements
- Explain the three major functional units of a processor
- Illustrate the typical registers inside the processor
- List the HC11 processor registers
- Discuss the HC11 processor modes
- Compare and contrast various memory types
- Describe the on-chip memory of the HC11
- Specify input/output functions present on most computers
- Use some basic BUFFALO commands to control the EVBU

Outline

Introduction

Computer systems have been developed for a variety of functions and purposes. General-application desktop machines are the most common. They run a variety of software applications, such as word processing, financial management and data processing. They have all but replaced the typewriter as a necessary business tool. Computers are also present in automobiles, appliances, airplanes and all types of controllers and electromechanical devices.

Despite the differences among these computer systems, they all share fundamental components and design. The purpose of this chapter is to provide an understanding of the fundamental components of a computer system. A conceptual presentation regarding the elements of every computer system is made with sufficient detail to establish a foundation for these concepts. The concepts will then be extended to the HC11 hardware.

1.1 Elements of Every Computer

All computers are made up of a group of three fundamental elements: a central processor, memory, and input/output devices. Figure 1.1 shows a block diagram of a computer that includes these three elements. The examination of any kind of desktop computer, workstation or computer control system will reveal at least this minimum structure. Many computing devices will have multiple processors, multiple memory types, and numerous input/output devices. In many cases, the input/output devices contain all three elements as a unit. Video cards for personal computers, for example, always contain a video processor and memory in addition to their inherent input/output capability.

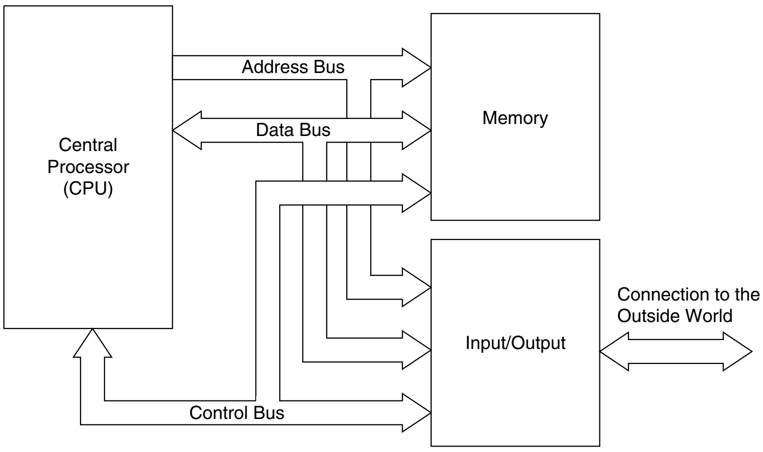


Figure 1.1 Fundamental Block Diagram of a Computer

HC11 Hardware Block Diagram

The HC11 is a computer system on a single chip that contains the three functional blocks of a computer system. Its internal central processor is a member of the 6800 family of processors; it has on-board memory and sophisticated on-chip input/output capabilities.

The HC11 block diagram is shown in Figure 1.2. This block diagram is specific to the M68HC11E9 version of the HC11. This version of the HC11 is used on the development board (EVBU), as well as in the examples used throughout this text. Three types of memory are included on-chip: RAM, ROM and EEPROM. The HC11 also supports expanded off-chip memory. It contains five on-chip input/output functions, an analog-to-digital converter, and a sophisticated timing system that supports numerous event-driven functions. The address, data and control busses are not shown in this diagram; the processor is connected to the memory and input/output functions inside the HC11 chip in the manner illustrated in Figure 1.1.

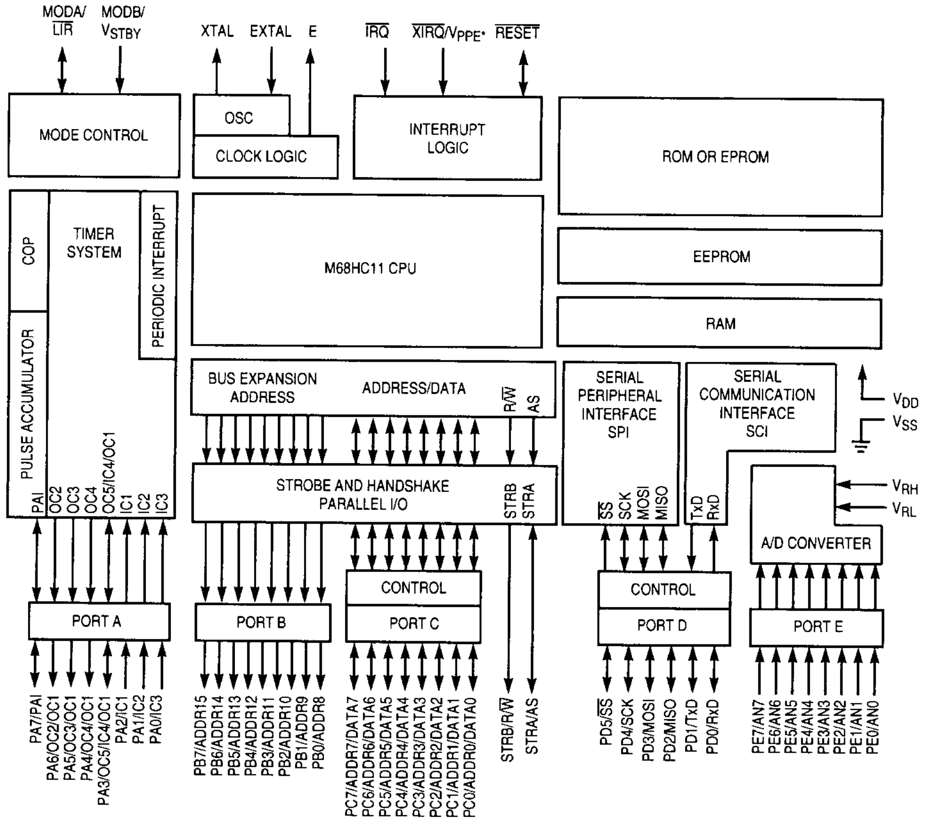

MODA/ LIR
MODB/ VSTBY
XTAL
EXTAL
E
IRQ
XIRQ/VPPE*
RESET
MODE CONTROL
OSC
CLOCK LOGIC
INTERRUPT LOGIC
ROM OR EPROM
COP
TIMER SYSTEM
PERIODIC INTERRUPT
M68HC11 CPU
EEPROM
RAM
PULSE ACCUMULATOR
PAI OC2 OC3 OC4 OC5/IC4/OC1 IC1 IC2 IC3
BUS EXPANSION ADDRESS
ADDRESS/DATA
R/W
AS
STROBE AND HANDSHAKE PARALLEL I/O
STRB STRA
SERIAL PERIPHERAL INTERFACE SPI
SS SCK MOSI MISO
SERIAL COMMUNICATION INTERFACE SCI
TxD RxD
VDD
VSS
VRH
VRL
A/D CONVERTER
CONTROL
PORT A
PORT B
PORT C
PORT D
PORT E
PA7/PAI PA6/OC2/OC1 PA5/OC3/OC1 PA4/OC4/OC1 PA3/OC5/IC4/OC1 PA2/IC1 PA1/IC2 PA0/IC3
PB7/ADDR15 PB6/ADDR14 PB5/ADDR13 PB4/ADDR12 PB3/ADDR11 PB2/ADDR10 PB1/ADDR9 PB0/ADDR8
PC7/ADDR7/DATA7 PC6/ADDR6/DATA6 PC5/ADDR5/DATA5 PC4/ADDR4/DATA4 PC3/ADDR3/DATA3 PC2/ADDR2/DATA2 PC1/ADDR1/DATA1 PC0/ADDR0/DATA0
STRB/R/W STRA/AS
PD5/SS PD4/SCK PD3/MOSI PD2/MISO PD1/TxD PD0/RxD
PE7/AN7 PE6/AN6 PE5/AN5 PE4/AN4 PE3/AN3 PE2/AN2 PE1/AN1 PE0/AN0

Figure 1.2 M68HC11E9 Block Diagram

NOTE: Throughout this text, references to the HC11 presume the M68HC11E9 version of the chip; thus, the shorter, more general name "HC11" will be used, unless otherwise specified.

Central Processor

The **processor** is the device at the center of the machine. It has the responsibility to execute instructions, manipulate data and perform arithmetic functions. It controls and manages the activities of the entire machine. The human brain is the ultimate processor. It can receive and process instructions, process data (like visual images and sounds) and perform arithmetic calculations. However, the human brain is much, much more than a processor because it has the ability to think and to reason. Computer processors cannot think in the same sense.

The term **Central Processing Unit (CPU)** is used to refer to the main processor in a system. The CPU often works in conjunction with a set of processors to complete a whole system. Modern computers contain additional processors, other than the CPU. They contain video processors, input/output processors, memory controllers, interrupt controllers and math co-processors, to name a few. Since these other processors are subordinate to the central processor, they are often called **sub-processors**. Many peripheral devices, such as harddrives, printers and video projection systems, have dedicated processors embedded into their control circuitry.

Memory

Memory is a term that refers to any component that stores data and programs used by the processor. Memory can have many forms. There are semiconductor memories, magnetic memories, and optical memories. Semiconductor memories include read only memory (ROM) and read/write memory (RAM). Magnetic memories include floppy disk drives, hard disk drives and tape systems. Optical memories include CD-ROM, DVD and optical disks. A thorough presentation of memory relevant to the HC11 will be provided in chapter 8. Section 1.3 will address concepts of memory that are applicable to all computer systems and a necessary foundation for this study of the HC11.

Input/Output

Input/Output is a term that refers to any subsystem that has the responsibility of receiving data for the processor (input) or sending data out from the processor (output). Input/Output is typically abbreviated as **I/O** and does not necessarily imply that a particular device has both input and output capability. Typical input devices are keyboards, mice or scanners. Typical output devices are printers and monitors. Typical devices that perform both input and output functions are modems or tape drives. Collectively, all these devices are referred to as I/O devices. The HC11 I/O capability is discussed in detail in chapters 10 through 13.

Busses

The processor communicates with the memory and I/O via busses. On a computer, a **bus** is a set of two or more conductors that are grouped together to form a parallel information path to and/or from the processor. The bus size, given in bits, is a measure of the number of conductors that can be active simultaneously on the bus. There are three major busses on computer systems: the data bus, the address bus and the control bus.

The **data bus** is responsible for the transfer of data between the processor and memory or the processor and I/O. It is a **bidirectional** bus, because data can travel to or from the processor and other devices. Typically, a data bus transfers data in byte widths. Thus, a data bus is an 8-bit, 16-bit or 32-bit bus (1 byte, 2 bytes or 4 bytes wide). The number of bits, or data bus width, also directly correlates with the default processing capacity of the processor. Typically, processors and computers are referred to by the size of their data busses. An HC11 is considered an 8-bit processor because it has an 8-bit data bus and has the default processing capability of one byte. Pentium-based personal computers are 32-bit or 64-bit machines. Thus, they can process four or eight bytes simultaneously.

The **address bus** is responsible for the transfer of addresses from the processor to memory or to I/O. The address is used to identify specific memory locations or I/O devices. It is a **unidirectional** or one-way bus, because processors are the only devices that can create an address for memory or I/O. Typically, an address bus transfers addresses over the bus in double byte widths. Thus, an address bus is usually a 16-bit, 32-bit, or 64-bit bus (2 bytes, 4 bytes or 8 bytes wide). The HC11 uses a 16-bit address bus.

The **control bus** is responsible for the control signals necessary to interface the various devices within a computer system. This bus is not typically structured as a fixed number of conductors, as are the data and address busses. It is more likely to be a collection of all other signals necessary for proper operation of the system.

Figure 1.3 illustrates how busses might be implemented between an HC11 and some expanded off-chip memory device. This example uses a 16-bit address bus, an 8-bit data bus and two control signals that would be part of a larger system control bus.

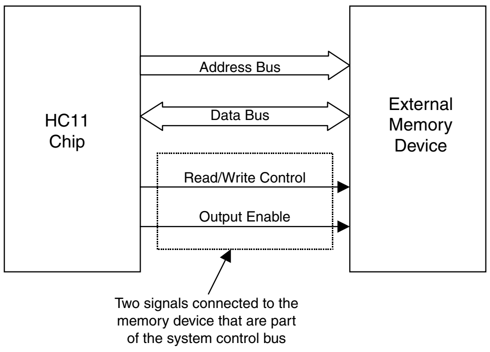

HC11
Chip
Address Bus
Data Bus
Read/Write Control
Output Enable
External
Memory
Device
Two signals connected to the memory device that are part of the system control bus

Figure 1.3 Computer Bus Example

Self-Test Questions 1.1

1. What are the three main hardware components of a computer?
2. What is the function of each of the three hardware components of a computer?
3. What connects the three hardware components and allows the data, addresses and control signals to move between these components?

1.2 Elements of Processors

Processors have the job of processing the data within a computer and controlling the overall operation of the system. The basic block diagram of a processor includes an arithmetic logic unit, processor registers and a control unit, as shown in Figure 1.4.

Arithmetic Logic Unit

The **arithmetic logic unit** (**ALU**) is responsible for mathematical and logical operations. Most processors support addition and subtraction, logical AND, OR and NOT operations and data shifting. All of these operations are performed in an ALU. The ALU receives data from the processor registers and from external memory via the external data bus. It does not store data, but it returns the result of the operations to the registers or to memory. The ALU is interfaced to a block of processor registers and

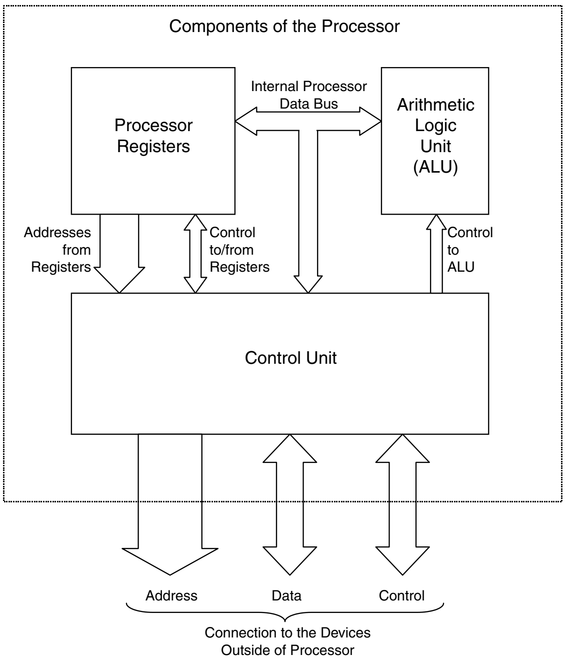

Components of the Processor
Processor Registers
Internal Processor Data Bus
Arithmetic Logic Unit (ALU)
Addresses from Registers
Control to/from Registers
Control to ALU
Control Unit
Address
Data
Control
Connection to the Devices Outside of Processor

Figure 1.4 Processor Block Diagram

to the control block, as shown in Figure 1.4. The HC11 has an ALU that can perform operations on 8-bit and 16-bit data.

Processor Registers

The **processor registers** are a set of registers needed to perform the instruction execution. The registers are used to temporarily store data and memory addresses, as well as to contain status and control information. These registers are accessible to the user via instructions. A basic register block configuration is shown in Figure 1.5. Each processor will have a unique design that may not include all the registers described here. Often a processor will contain multiple copies of these registers for versatility and enhanced functionality.

Accumulators are special registers directly linked to the ALU to assist with the arithmetic operations. The results of arithmetic operations are stored in an accumulator. The size of the accumulators is related to the size of the data bus. If the processor is an 8-bit processor, then it will have 8-bit accumulators. If it is a 16-bit processor, then it will have 16-bit accumulators. Moreover, processors often have the ability to process data of various sizes. For example, an 8-bit processor may also be able to process 16-bit data. If the processor has a 32-bit configuration, it can also process data in 16-bit or even 8-bit words.

The **program counter** is a processor register that keeps track of the address of the next location in memory that will be accessed. Every processor must have a register that performs the function of a program counter. The program counter is the same size as

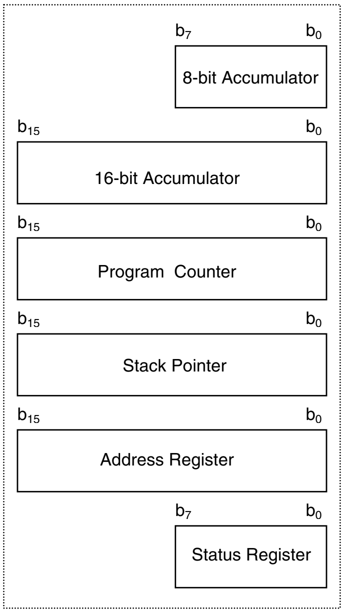


Figure 1.5 Basic Processor Registers

the address bus. If the processor has a 16-bit address bus, the program counter is a 16-bit register. Instructions require one or more bytes of memory for their completion; therefore, the program counter must be capable of incrementing through memory as the instructions are executed. Without the program counter, the processor would not be able to determine which instruction to execute. This register is sometimes called the **instruction pointer**.

The processor has a need to temporarily store and retrieve data from memory during the processing of instructions. This temporary memory area is called the stack. The processor must have a register that contains an address pointer that indicates the next available memory location on the stack. This register is called the **stack pointer**. The stack pointer is the same size as the address bus. If the processor has a 16-bit address bus, the stack pointer is a 16-bit register.

Most general-purpose processors contain registers that are used specifically by the instructions to address memory. These registers have various names, but generally they are called **address registers**. The use of address registers allows for simpler instructions, because they do not need to be concerned about the address. The address required to complete the instruction is already provided in the address register.

Every processor must have a register dedicated to reporting status. The **status register** contains bits that indicate certain results of the last operation. The most common status bits are sign flags, carry/borrow flags, zero flags and overflow flags. The sign flags indicate the sign of the last data processed. The carry/borrow flags indicate that an arithmetic operation produced a result larger than could fit into the register used or that it had to borrow to complete the operation. The zero flag indicates that result of the last operation was zero, and the overflow flag indicates that a sign overflow occurred (the sign of the last operation is wrong).

Control Unit

The **control unit** within the processor is responsible for reading the instruction from memory; it then ensures that the instructions are executed. The control function is driven by the instructions that are decoded by the control unit. The decoded instruction causes a series of steps to be followed during the execution. Figure 1.6 shows the major components of the control unit within a processor.

The **memory address register (MAR)** is a special address register that is linked to the program counter. The job of the MAR is to contain the address of the current data word that is being addressed. The **memory data register (MDR)** is another special register that resides between the data bus and the various processor registers. It provides buffering and control of the movement of data into and out of the processor. The MAR and MDR are not accessible to the user, but are used internally by the processor for address and data bus interface and to assist in the execution of instructions.

The **instruction register (IR)** is a special register that always contains the opcode for the current instruction. This instruction opcode is read from memory during a fetch

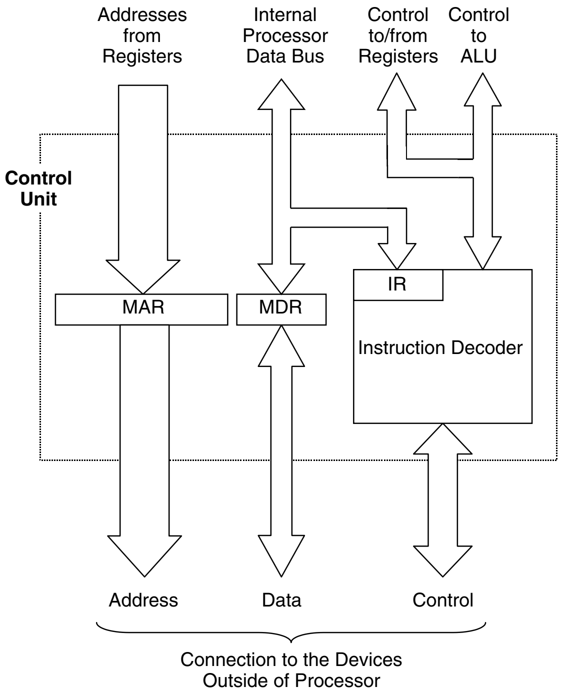

Addresses from Registers
Internal Processor Data Bus
Control to/from Registers
Control to ALU
Control Unit
MAR
MDR
IR
Instruction Decoder
Address
Data
Control
Connection to the Devices Outside of Processor

Figure 1.6 Components of the Processor Control Block

cycle and deposited into this register. The opcode remains in this register until it is overwritten by the opcode for the next instruction.

Opcodes, operands and instructions are explained in Chapter 2.

NOTE: The instruction register should be called the opcode register, because it contains only the opcode of the instruction. Operands and operand addresses are not processed by the instruction register.

The **instruction decoder** is the main section of the control block. It has the job of decoding the instruction in the instruction register and controlling the execution of the instruction. It causes each step of the instruction execution to take place. If an address needs to be read from memory, the instruction decoder will cause that to take place. If data needs to be added, it will tell the ALU to perform an addition operation, and so on.

The process of fetching and executing an instruction is covered in chapter 2.

HC11 Processor

The HC11 is a member of the Motorola 6800 family of processors. It is compatible with the M6800 and M6801 processors. In addition to executing all M6800 and M6801 instructions, the HC11 supports many additional instructions. The HC11 supports 16-bit by 16-bit divide instructions, twelve instructions that support bit-manipulation, and various instructions that support a second address register (Y), none of which was available on the original 6800.

The HC11 MCU is an 8-bit processor, which supports a 16-bit address bus. It contains an 8-bit ALU that can be configured to perform some 16-bit operations.

HC11 Processor Registers

The HC11 has a complete set of processor registers. This set of registers is called the **programmer's model,** as shown in Figure 1.7. It has two 8-bit accumulators that can be configured as a 16-bit accumulator for some operations. It has two16-bit address registers, a 16-bit stack pointer and a 16-bit program counter. Finally, it contains an 8-bit status register that contains three control bits and five status flags. None of these registers is addressable as memory; they can only be accessed via instructions. These registers are described in the following sections.

Accumulator A. An 8-bit register that is used as the primary data-processing register. All arithmetic and logical instructions can operate on data in this register. Many special-purpose instructions can only operate on data in this register. This register is referred to as the "A" register or as "AccA."

Accumulator B. An 8-bit register that is identical to Accumulator A in function. Some instructions have versions that access Accumulator A only and do not support Accumulator B. There are also two instructions that utilize Accumulator B and do not support Accumulator A. This register is referred to as the "B" register or as "AccB."

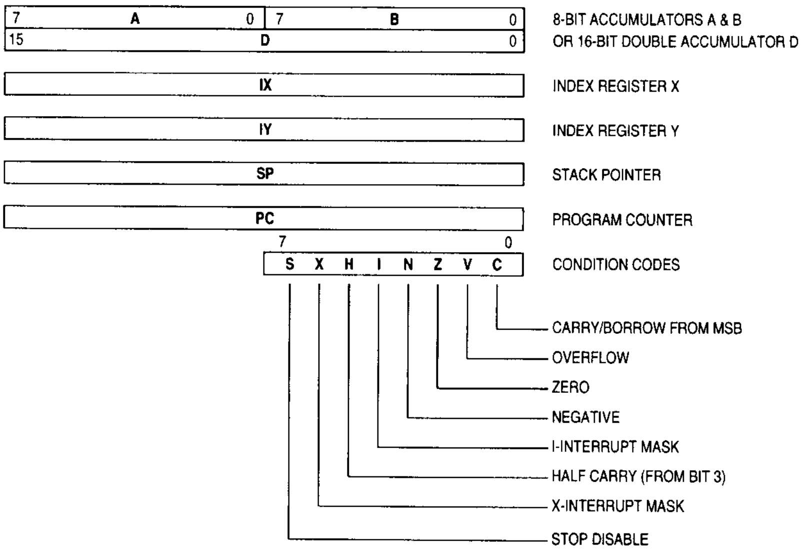


Figure 1.7 HC11 Programmer's Model (*adapted with permission from Motorola*)

Accumulator D (A:B). A 16-bit register that is the two 8-bit accumulators joined together (the colon between A and B indicates the joining of the two registers). It is not a separate hardware register. Although the HC11 is technically an 8-bit processor, several instructions are provided that allow 16-bit operations. This joining of A and B is referred to as the "D" register or as "AccD."

Index Register X. A 16-bit address register that is used by the indexed addressing mode instructions. It can also be used as a general-purpose, 16-bit data register. When used with the indexed addressing mode instructions, it contains a 16-bit base address. When used as a general-purpose data register, 16-bit data of any type can be stored and processed in this register. This register is referred to as the "IX" or simply the "X" register.

The role of the memory addressing registers is explained in chapter 2 during the presentation on memory addressing modes. The use of these registers to address memory is emphasized in chapter 5.

Index Register Y. A 16-bit address register that is identical in function to index register X. This register is referred to as the "IY" or simply the "Y" register.

Stack Pointer. A 16-bit register that always contains the address of the next available stack memory location. It is referred to as the "S" or the "SP" register.

The use of the stack pointer and the function of the memory stack are explained in chapter 6.

Program Counter. A 16-bit address register that always contains the address of the next location in memory that will be addressed. It is referred to as the "PC."

Condition Code Register. An 8-bit status and control register. Five of the eight bits (H, N, Z, V and C) are status flags, which indicate the results of the last processor operation. The remaining three bits (S, X and I) are control bits for advanced processor functions. This register is referred to as the "C" register, the "CCR" register and the "Status" register.

The function of the status flags is presented in chapter 3 (Instruction Set). The function of the control bits is presented in chapter 10 (Interrupts).

HC11 Processor Modes

The HC11 supports four hardware modes: Single Chip, Expanded, Special Test and Bootstrap. Each hardware mode configures the HC11 to perform a special class of hardware functions.

Single chip mode is the normal operating mode. As the name implies, the single chip mode requires that all software needed to control the processor is contained in the on-chip memories. External address and data busses are not available. **Expanded mode** is an alternative normal operating mode. It allows for off-chip memory connection. In this mode, the PORTB and Port C pins are converted to a multiplexed address and data bus. In addition, the strobe A and strobe B control lines become the address strobe and read/write control lines.

Special Test mode is a special mode intended primarily for testing during the chip production process. The **bootstrap mode** is another special hardware mode, designed

MODB	MODA	Mode
1	0	Single Chip
1	1	Expanded
0	0	Bootstrap
0	1	Test

Figure 1.8 Hardware Mode Select (*courtesy of Motorola*)

to allow loading of permanent programs and other production related programming tasks.

NOTE: This text focuses only on the normal modes and primarily on operation in the single chip mode. Some examples and explanation of the use of the expanded mode are provided in chapter 8 on memory. The special modes will not be discussed, because of the advanced nature of their function.

The hardware mode is selected by the logic levels on two mode control pins (MODA and MODB) when the processor is in the reset state. Figure 1.8 shows the logic levels required to select each of the four modes. The MODB bits selects between the normal modes (Single Chip and Expanded) and the special modes (Bootstrap and Test).

Self-Test Questions 1.2

1. What are the three main components of a central processor?
2. What is the function of each of the three components of a processor?
3. What registers make up the M68HC11 programmer's model?

1.3 Introduction to Memory

Memory is a term that refers to any component that stores data and programs for the processor. The programs (software), as well as the program data, are stored in the memory during execution and processing. Memory consists of storage locations, where each location can contain one byte of information. A unique address is assigned to each storage location in memory so that it can be individually accessed. The HC11 uses memory that contains a byte of data in each storage location.

A **memory address** is an *n*-bit binary number that the processor uses to select a specific memory location. If the processor wishes to communicate with memory location 7, it produces the 16-bit binary address for this location (%0000 0000 0000 0111) and places this address on the address bus. The number of address bits in a system determines the number of unique addresses that can be created by the processor, as shown in

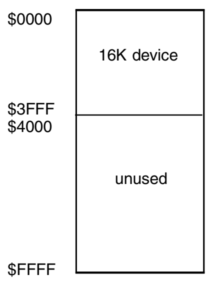


Figure 1.9 Example of Memory Map

Equation 1.1. The HC11 uses a 16-bit address. Therefore, it can address 2^{16} or 64K unique memory locations.

$$\text{\# of unique addresses} = 2^n \qquad \text{Equation 1.1}$$

where n is the number of address bits in the system.

Although an address can be generated from each combination of binary bits in the address, there may not be physical memory at all locations in the address range. A **memory map** designates the memory addresses that are connected to physical memory locations and indicates which locations are unused. For example, a system may have a 16-bit address bus, which allows it to address 64K memory locations. However, the system may have only 16K of physical memory connected. In this case, the memory map indicates the address range within the 64K range that will be used for this 16K device. Figure 1.9 illustrates how this memory map would look.

The **data** is the information stored at each memory location. Since all data is just a group of binary bits, it all appears about the same to the processor. It is impossible by simple inspection to know if a value in memory is a number, a special code or a piece of a larger address. The real value of the data is how it is used. The meaning of the data comes from the order in which it appears in the memory, as well as how it is used by the processor.

The process of storing data in memory is called a **write operation**. When a specific memory location is addressed, the data can be written to the data bus by the processor, as shown in Figure 1.10. The processor generates the address $0005 and sends it to the memory device on the address bus. Then the processor places the data $CF on the data bus. The processor tells the memory what type of operation to perform via the control signals. Writing to memory is a **destructive** process, which means that any data previously in the memory location is overwritten by the new data and the old data is lost. After a write operation, the data will remain in the memory location until it is overwritten by another write operation.

Data can also move from the memory location to the processor via a **read operation,** as shown in Figure 1.11. A read operation causes a copy of the data to be sent to the processor from the memory. The processor initiates a read operation by placing the address ($0007) on the address bus. It then signals the memory to

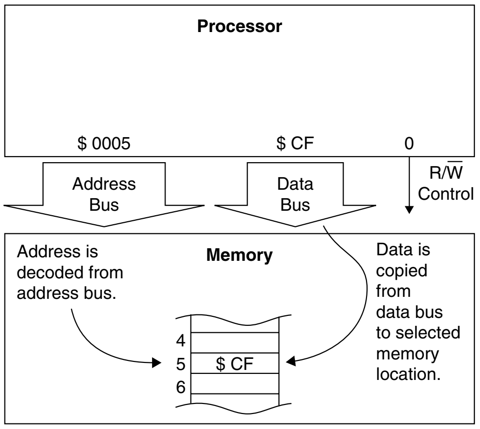

Processor
$ 0005
$ CF
0
Address Bus
Data Bus
R/$\overline{W}$ Control
Memory
Address is decoded from address bus.
Data is copied from data bus to selected memory location.
4
5
6
$ CF

Figure 1.10 Processor Write Operation

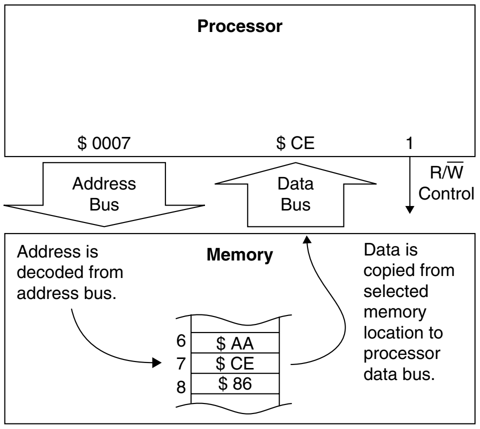

Processor
$ 0007
$ CE
1
Address Bus
Data Bus
R/$\overline{W}$ Control
Memory
Address is decoded from address bus.
Data is copied from selected memory location to processor data bus.
6
7
8
$ AA
$ CE
$ 86

Figure 1.11 Processor Read Operation

place the data ($CE) from the addressed location on the data bus. The processor waits a specific length of time and then reads the data from the data bus. The signal between the processor and the memory that controls the direction of data movement is usually called read/write. Reading from memory is a **nondestructive** process, which means that data in the memory location is not affected.

Self-Test Questions 1.3

1. What is an address?
2. How many unique addresses can be generated from a 12-bit address bus? A 20-bit address bus?
3. In which direction does the data move during a read operation?

1.4 Memory Types

The physical memory locations can be represented by various types of memory devices. Two different types of memory made from semiconductor materials will be described: volatile and nonvolatile. **Volatile** memory is a memory that retains the data only when the power is applied. If the power is removed, the data stored in the memory locations will be lost. As the name implies, nonvolatile memory is not volatile. A **nonvolatile** memory retains the data with or without power being applied to the device.

RAM

RAM stands for Random Access Memory; however, it is better described as read/write memory. Most memory on modern computers is **random access,** which means that it can be randomly addressed (the data is accessed in any order the processor chooses). Sequentially accessed memory is used only for special processing and timing applications and will not be described in this text.

RAM is **volatile**, retaining data only when the power is applied. If the power going to RAM is turned off, all data contained in it is lost. When power is reapplied, the content of RAM is undefined, which means the data can be any value. RAM is used for temporary storage of programs while they are being executed, as well as for temporary storage of data while it is being processed. RAM is relatively fast memory when compared to the whole family of various memory devices. RAM is also relatively less expensive, uses less power and is smaller than most memory devices.

ROM

ROM stands for Read Only Memory. ROM is used for permanent storage of programs and data. It is a **nonvolatile** memory; therefore, it always retains its data, whether power is applied or not. Standard ROM must be loaded with the permanent programs and data during the manufacturing process. Other types of ROM have been developed that allow programs and data to be written to them after the manufacturing process.

Programmable Read Only Memory or **PROM** is designed to be programmed in the field, after the manufacturing process. This provides the user the ability to purchase one chip and use it for various purposes dependent upon the field application. PROM can be programmed one time. If a program or data contained in a PROM needs to be modified, the PROM must be discarded and a new one must be programmed.

Erasable Programmable Read Only Memory, **EPROM**, is a type of PROM that can be erased by applying an intense ultraviolet light to the memory circuitry. Once it has been erased, it can be reprogrammed using a special programming fixture. Chips that contain EPROM have a small window on the chip package that allows the light to reach the memory circuitry. Once an EPROM is programmed, this window is covered to stop light from entering the device and inadvertently erasing the memory locations.

Electrically Erasable Programmable Read Only Memory, **EEPROM**, can be erased with electrical signals and then reprogrammed. EEPROM is used in various devices that

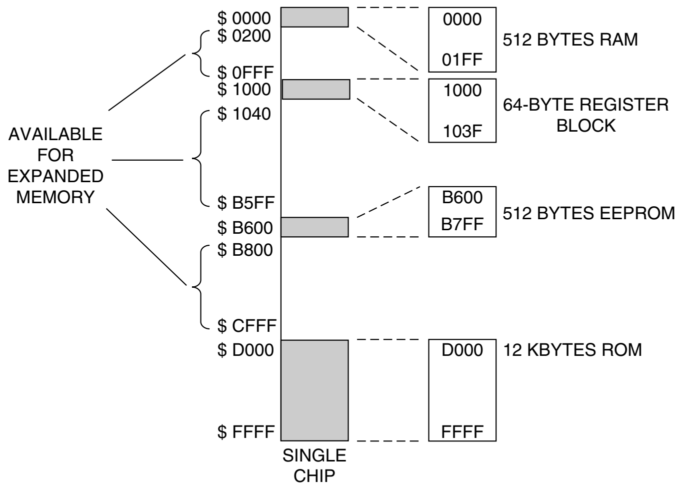


Figure 1.12 Memory Map on the HC11E9

require permanent memory yet need the convenience of electrical erasure. Single locations within an EEPROM can be erased and reprogrammed.

HC11 Memory

The HC11E9 contains three types of on-chip memory, RAM, EEPROM and ROM, as shown in Figure 1.12. There are 512 bytes of RAM located in the memory map, from $0000 through $01FF. In addition to the RAM, the HC11 contains 512 bytes of EEPROM and 12K of ROM. The EEPROM is located from $B600 through $B7FF in the memory map, and the ROM is located from $D000 through $FFFF. In addition to this memory, a 64-byte register block is located at $1000 through $103F within the memory map.

Refer to chapter 8 for a further presentation regarding HC11 memory.

Self-Test Questions 1.4

1. What two types of memory devices make up the memory system of a computer?
2. What type of ROM can be erased by electrical signals?
3. How much RAM is contained on the HC11E9?

1.5 Input/Output

The third element of every computer system is the I/O. All computers must input data from at least one source. The data is processed internally and then output to an external device. Not all I/O capabilities are related to specific devices. For example, most desktop computer systems have general I/O ports. A **port** is an I/O connection that allows for the movement of data between the computer and an I/O device. There are two major types of ports on computers: serial ports and parallel ports. A **serial port** is

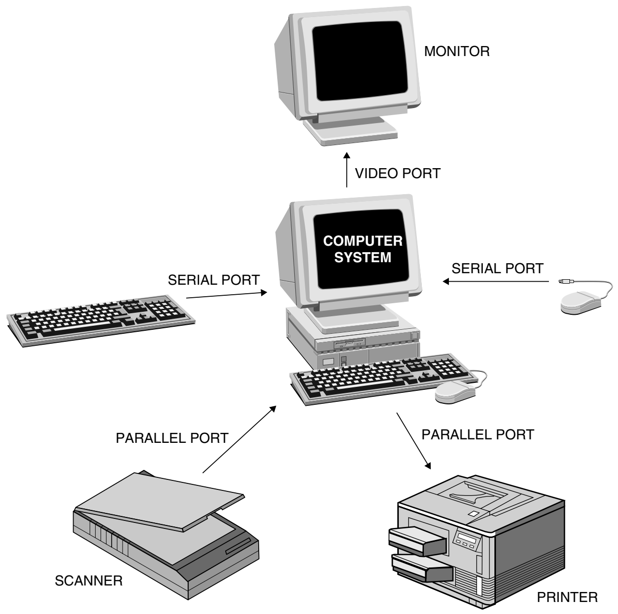


Figure 1.13 I/O Devices Connected to a Computer System

designed to allow two-way transfer of data as a serial data stream. **Serial** means one bit is transferred at a time over a single wire or line. Modems use serial connections to interface to the computer. High-speed serial ports are becoming the standard for communications. A **parallel port** is designed to transfer data in parallel. **Parallel** means multiple bits are transferred simultaneously over multiple wires (typically, 8). Historically, most scanners and printers use parallel ports to interface with the computer.

A special type of I/O requires an **analog** interface. Many data sources in the real world are analog. For the computer to access this data, it must be converted into a digital form, using an analog to digital converter. When the computer needs to send data to an analog device, a digital to analog converter is necessary. Examples of analog data are temperature, wind speed, and humidity.

Figure 1.13 illustrates various I/O devices that may be connected to a computer system.

I/O on the HC11

The HC11 contains several on-chip ports that provide a variety of input/output functions. The I/O capabilities are organized as shown in Figure 1.14.

Port A can be used for digital I/O as well as for multiple timer functions. The digital I/O capabilities include three dedicated output pins, three dedicated input pins and two programmable I/O pins. A 16-bit free running counter is included to support the input-capture and output-compare timer functions. In addition, an 8-bit pulse accumulator subsystem and a periodic interrupt generator are provided as part of the Port A timing system.

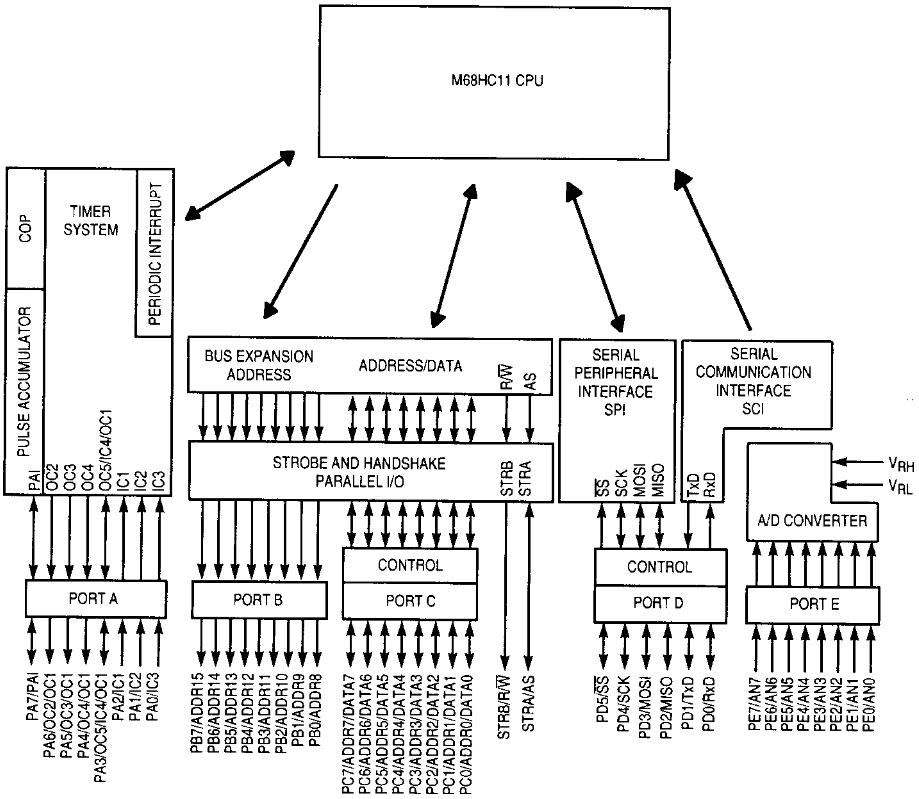


Figure 1.14 HC11 I/O Ports (*adapted with permission from Motorola*)

See chapter 12 for explanation of the Port A timing system.

Port B is an 8-bit general-purpose output port. Port C is an 8-bit general-purpose port that can be configured for either input or output operation.

The primary I/O functions of Port B and Port C are presented in chapter 9.

Port D is a general-purpose digital port and also supports serial communication functions. The digital I/O function uses the six I/O pins of Port D as programmable I/O pins. When used for serial communications, two independent serial interfaces can be configured. They are called the serial communications interface (SCI) and the serial peripheral interface (SPI).

The serial communications capabilities built into Port D are discussed in chapter 13.

Port E is an 8-bit general-purpose digital input port that can be configured as an 8-channel analog-to-digital converter with an internal resolution of 8-bits.

The analog input capabilities of the HC11 are presented in chapter 11.

Self-Test Questions 1.5

1. What is a port?
2. What is the difference between a serial and a parallel port?
3. Why are analog interfaces needed?
4. How many I/O ports are contained on the HC11? What are they?

1.6 EVBU / BUFFALO

A full explanation of the function of the EVBU is presented in the Universal Evaluation Board User's Manual. Refer to section 4.6 of the EVBU User's Manual for a description of the BUFFALO monitor commands.

The **EVBU** is the Motorola M68HC11 Universal Evaluation Board. It is a development tool for HC11 microcontroller-based designs. It provides a variety of features with which to experiment with the functions of the HC11 and to test designs based on the HC11. It also works well as an educational tool. In addition to the HC11 chip, the EVBU contains the following hardware features:

- M68HC68 Real-time clock chip. The real-time clock chip provides accurate timing information in seconds, minutes, hours, and so on.
- Standard serial communications port. This port allows connection to an external computer.
- Breadboard area. A small solder pad development area is provided on the EVBU layout.

The EVBU has a monitor program that serves as an operating environment. The **monitor program** is a piece of software that provides a controlled environment in which the HC11 can operate. It also provides a **user interface (UI)**. The UI allows a user of the EVBU to enter some simple commands to access memory, run programs and monitor the function of the HC11. The monitor program is called **BUFFALO** (Bit User Fast Friendly Aid to Logical Operations). BUFFALO works in conjunction with the EVBU hardware to provide the following features:

- One-line assembler. The user can enter programs directly into memory.
- Program download. Programs that have been created externally can be loaded directly into memory.
- Troubleshooting capabilities. BUFFALO commands allow users to execute programs one step at a time, to stop programs at a particular instruction and to examine the contents of the registers and memory.

BUFFALO is burned into the HC11 on-chip ROM during the manufacturing process. It occupies 8K of the 12K of available ROM. In addition, some of the RAM is used by BUFFALO as system temporary memory. Therefore, only part of the RAM is available for user programs and data. However, the EEPROM is unused by the EVBU and is available for user programs. The ROM on the EVBU contains the system control program.

NOTE: BUFFALO is located in the on-chip ROM from $E000 through $FFFF. It uses some of the on-chip RAM as temporary memory. User programs must not use RAM between $0030 and $00FF, to avoid conflicts with the system temporary space.

A photograph of the EVBU is shown in Figure 1.15. The block diagram of the EVBU is shown in Figure 1.16.

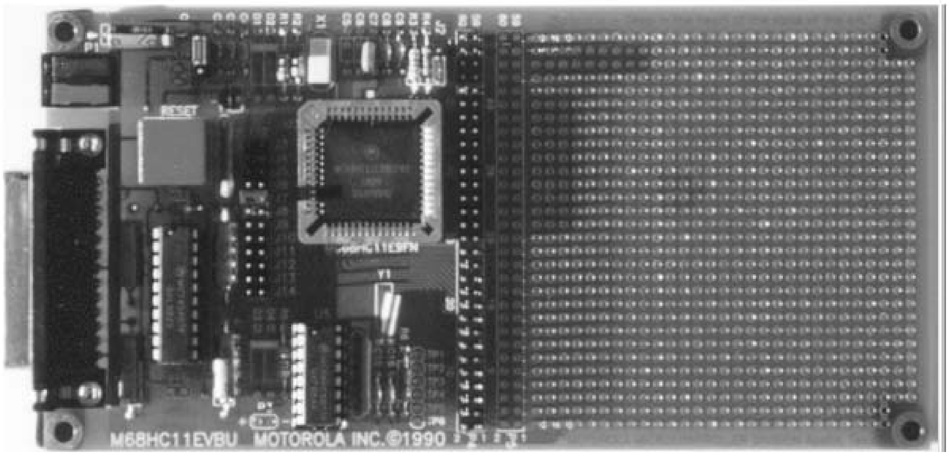


Figure 1.15 EVBU Photo

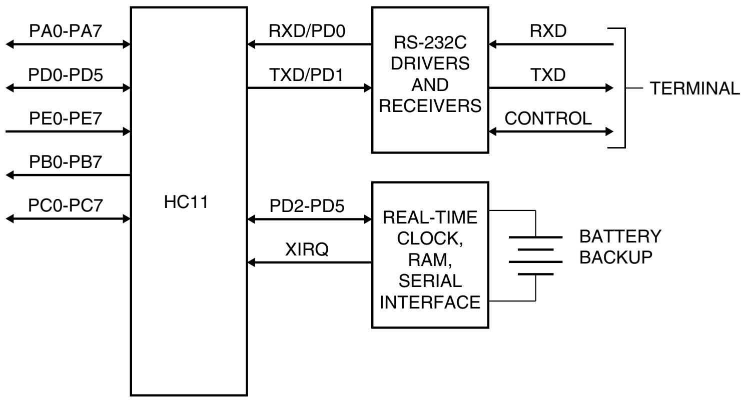


Figure 1.16 EVBU Block Diagram

Basic EVBU Function

The BUFFALO monitor program controls the HC11 within the environment created by the EVBU hardware. The user can type commands at the BUFFALO prompt that tell BUFFALO to perform a variety of memory operations and control functions. For example, BUFFALO provides a command that allows the user to display a block of memory to the screen. This command is called Memory Display and is abbreviated by using the letters MD followed by the address range that will be dumped to the screen. The operation of this instruction is shown in Figure 1.17. The user types in the MD command at the BUFFALO prompt, followed by the start and end addresses of the range to be displayed (in this case $E640–$E68F). Note that hex is assumed for the addresses that are supplied with the BUFFALO command. Thus, the user should not use the preceding "$" character to indicate hex. Then BUFFALO displays values stored in memory for this range. Each row of the output contains the start address or the address of the first byte displayed, followed by the hex values for the data stored in 16 sequential memory locations. At the end of the line, the ASCII equivalent characters are displayed for each value shown on the line (i.e., $42 = "B", $55 = "U", $46 = "F," etc.). When the display is complete, a new BUFFALO prompt is displayed, prompting the user for another BUFFALO command.

The concepts of ASCII communications and use of ASCII characters are presented in Chapter 13.

Rather than display all the commands supported by BUFFALO and the EVBU at this point in the text, a brief summary of several useful commands is provided in Figure 1.18. The four commands listed in this table are used to display and change the data in the memory and processor registers in the HC11. They are very useful from the beginning of the study of the HC11 and will be used in the problems at the end of this chapter.

The following examples are provided to show the operation of these instructions.

A complete list of BUFFALO commands is provided in section 3 of the EVBU User's Manual.

```
> MD E640 E68F<cr>

E640 42 55 46 46 41 4C 4F 20 33 2E 34 20 28 65 78 74 BUFFALO 3.4 (ext
E650 29 20 2D 20 42 69 74 20 55 73 65 72 20 46 61 73 ) - Bit User Fas
E660 74 20 46 72 69 65 6e 64 6c 79 20 41 69 64 20 74 t Friendly Aid t
E670 6f 20 4C 6f 67 69 63 61 6C 20 4F 70 65 72 61 74 o Logical Operat
E680 69 6F 6E 04 57 68 61 74 3F 04 54 6F 6F 20 4C 6F ion What? Too Lo

>
```

Figure 1.17 Operation of the MD Command in BUFFALO

Command	Usage	Description
MD	MD<Start><End>	Memory Display: Displays successive memory locations in rows of 16 bytes starting with the start address <Start> and ending with the row that contains the ending address <End>.
MM	MM<Address>	Memory Modify: Display a Single byte of memory specified by <Address> to the screen and allow the user to modify this data.
BF	BF<Start><End><Data>	Block Fill: Fills successive memory locations with the data <Data> starting with the start address <Start> and ending with the row that contains the ending address <End>.
RM	RM	Register Modify: Display the contents of the programmer's model registers and allow the user to modify their contents.

Figure 1.18 Summary of BUFFALO Commands

Example 1.1

Problem: Use the BUFFALO monitor commands to load a range of memory ($0000–$000F) with $20 and then display the range of memory on the monitor.

Solution: This problem requires the use of two BUFFALO commands: BF and MD. First, BF will be used to fill the range with the value $20, and then the MD command will be used to display this on the monitor. The part that is typed by the user is bolded.

```
> BF 0000 000F 20 <cr>

> MD 0000 000F<cr>

 0000 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20

>
```

Example 1.2

Problem: Use the BUFFALO monitor commands to modify the data stored in memory location $0003 so that it contains $FE and then to display the range of memory ($0000–$000F) on the monitor.

Solution: This problem requires the use of two BUFFALO commands: MM and MD. First MM will be used to modify the value at $0003 to $FE, and then the MD command will be used to display a range on the monitor. The part that is typed by the user is bolded.

```
> MM 0003<cr>

0003 20 FE<cr>

> MD 0000 000F<cr>

0000 20 20 20 FE 20 20 20 20 20 20 20 20 20 20 20 20

>
```

Example 1.3

Problem: Use the BUFFALO monitor commands to display the data stored in the processor registers. Modify the contents of the X register to be $1000 and AccA to be $3B. Redisplay the contents of these registers to verify that the changes took place.

Solution: This problem requires the use of one BUFFALO command: RM. First RM will be used to display the register contents and modify the values in the X register and AccA. Then the RM command will be used again to display the contents and verify that the changes took place. The part that is typed by the user is bolded.

```
> RM<cr>
P-0100 Y-FFFF X-FFFF A-FF B-FF C-D1 S-0041
P-0108<SPACE BAR>
Y-FFFF<SPACE BAR>
X-FFFF 1000<SPACE BAR>
A-FF 3B<cr>
> RM<cr>
P-0100 Y-FFFF X-1000 A-3B B-FF C-D1 S-0041
>
```

Self-Test Questions 1.6

1. What is the EVBU?
2. What is the name of the monitor program on the EVBU?
3. Which BUFFALO monitor command is used to fill a range of memory locations with a particular value?

Summary

All computer systems consist of three major components: processor, memory and I/O. The processor is the controlling element of the system. It manages the movement of data to and from the memory and I/O components. The processor is also made up of three major components: ALU, internal registers and control block. The ALU is responsible for the arithmetic and logical functions performed by the processor. The registers provide temporary storage of data and addresses for efficient processing, and the control block manages the execution of the instructions. Memory contains the programs and data used by the processor. Two major types of memory are represented in computer systems, RAM and ROM. Finally, all computer systems must have some connection to the external world. These connections are accomplished by the I/O capabilities. A keyboard is an example of a standard input device, and a monitor is an example of a standard output device.

Chapter Questions

Section 1.1

1. What are the three major components of a computer system? Why are all of the components necessary in an operational computer system?
2. How does the HC11 meet the functional components of a computer system (processor, memory and I/O)?
3. How are address, data and control signals connected to the various components within the computer system?

Section 1.2

4. What is a processor? What function does a processor perform?
5. What are the sections of a processor?
6. What is the purpose of the ALU?
7. List the common register types in a microprocessor. What purpose does each of the registers have, generally?
8. Does the Control Unit contain any registers? If so, how are they different from the registers in the processor?
9. What kind of processor is embedded in the HC11 microcontroller?
10. How many bits of data can be processed at a time by the HC11 processor?
11. What is in the HC11 programming model?
12. Does the HC11 support different hardware modes? What are they?

Section 1.3

13. If a system has a 20-bit address bus, how many unique addresses can be addressed?
14. What is a memory map?
15. What is the size of the HC11 address bus? What is the maximum number of memory locations that can be addressed by the HC11?
16. If a system has a 16-bit address bus and must address two 32 Kbyte memory chips, draw a picture of how the memory map would be configured to uniquely address each location within the memory devices.
17. How does the processor communicate to memory whether it wishes to perform a read or a write operation?
18. Why is reading memory nondestructive to the data in the addressed memory location?
19. Why is writing to memory destructive to the data that was originally in the addressed memory location?

Section 1.4

20. Explain the differences among ROM, EPROM and EEPROM.
21. Why must every computer system have some nonvolatile memory?
22. What types and quantities of memory are on the M68HC11E9?
23. Is the entire address space used by the on-chip memory on the HC11? If not, how much address space is available?

Section 1.5

24. What types of ports are typically available on computer systems?

25. How many ports are available on the HC11?
26. What types of I/O functions are supported by the ports on the HC11?

Section 1.6

27. What is the EVBU? What is its purpose?
28. What is BUFFALO? What is its purpose?
29. How many parameters are used by the BF command?

Answers to Self-Test Questions

Section 1.1

1. Processor, memory and I/O
2. The processor is at the center of the computer. The processor manipulates the data, performs arithmetic functions and manages the processes. The memory is used to store data and programs. The I/O is used to move data to/from the computer and external devices.
3. Busses. The address bus transfers address information from the processor to the memory and I/O devices. The data bus is a bidirectional bus that transfers data to/from the processor, memory and I/O. The control bus is a collection of control signals necessary to control the processes of the computer.

Section 1.2

1. Arithmetic logic unit (ALU), the register block and the control block.
2. The ALU has the job of performing the actual arithmetic data processing. It can do arithmetic and logical operations. The register block contains a set of registers that are used by the processor to store data temporarily, accumulate data, manipulate addresses, and so on. The control block decodes instructions and generates all of the timing and control signals necessary to execute the instructions.
3. The HC11 programming model contains two 8-bit accumulators (A and B) that can be configured as a 16-bit accumulator (D); two address registers (X and Y); a stack pointer (SP), which contains the address of the next available stack location; the program counter (PC), which contains the address of the next instruction to be executed; and the condition code register (CCR), which contains three control bits and five status flags.

Section 1.3

1. An *n*-bit binary number that the processor uses to select a specific memory location.
2. 12-bit bus $\rightarrow n = 12$, therefore $2^n = 2^{12} = 4K$ addresses can be generated.
   20-bit bus $\rightarrow n = 20$, therefore $2^n = 2^{20} = 1M$ addresses can be generated.
3. The data moves from the memory to the processor during a read operation.

Section 1.4

1. Volatile and nonvolatile, or RAM and ROM.
2. The EEPROM is electrically erasable.
3. There are 512 bytes of RAM on the HC11E9.

Section 1.5

1. A port is an I/O connection that allows data to move between the computer and a variety of peripheral devices.
2. A serial port transfers data one bit at a time on a single wire. A parallel port transfers data multiple bits at a time via multiple wires.
3. Analog interfaces are required because most things in the real world are analog. Examples of analog data are temperature, wind speed, humidity and time.
4. There are five I/O ports on the HC11: PORTA, PORTB, PORTC, PORTD and PORTE.

Section 1.6

1. The EVBU is the HC11 Universal Evaluation Board.
2. The monitor program on the EVBU is called BUFFALO (Bit User Fast Friendly Aid to Logical Operations).
3. The BF command allows a range of memory locations to be filled with the data value given in the command (i.e., BF 0006 001D 40 will fill \$0006–\$001D with the value \$40).

c h a p t e r 2

Introduction to Computer Software

Objectives

After completing this chapter, you should be able to:

- Describe the difference between source code and machine code
- Define opcode, operand and address of an operand
- Explain the purpose of memory addressing modes
- Show how memory addressing modes work on the HC11
- Explain how an instruction is fetched from memory and executed by the processor
- Use flowcharts to show the flow/function of a program

Outline

Introduction

The programs that run on computers are called software. The software consists of ordered sets of instructions that tell the hardware what it should do. The computer hardware will not function without the software controlling each function. The instructions control the processes that are executed, the data that is processed and the order in which these operations are completed. This chapter will introduce some concepts behind the software that controls the hardware.

2.1 Programming the Computer

A **program** is a detailed list of steps that must be followed to complete a task. Many jobs in life follow the same steps each time they are accomplished. Someone who gets up in the morning to go to work often follows a specific set of steps to prepare for work. Cooking a meal from a recipe requires following a set of instructions in a specific order. Most tasks become so natural that the fact that specific steps are followed is forgotten.

Programs tell computers what to do. When the tasks become more complex, more instructions are required to accomplish the task. Unlike humans and animals, computers do nothing except what they are told by the programs. In fact, they do only what they are told. In this sense, they are 100% obedient. They never complain and never have to be fed. They definitely can't think. The strength of computers comes from the fact that they perform many very simple instructions millions of times a second to accomplish much greater tasks.

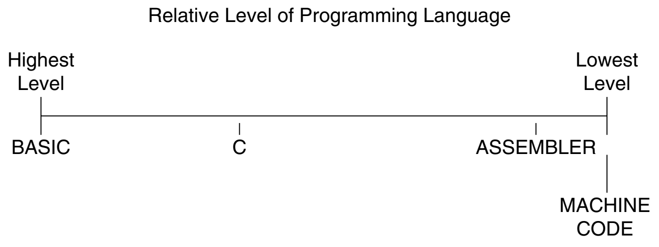
Some computers are designed to execute only one program. For example, the computer embedded in the transmission controller on a minivan executes one program. The job of this computer is to control the shifting of the automatic transmission. It has one program or set of instructions. Each time the computer is activated when the car is turned on, it begins to execute the set of instructions that controls the automatic transmission. It knows of nothing else, cares about nothing at all and never complains about its job. This type of computer is a **dedicated** computer.

Many computers are more general in nature. They are designed to allow a variety of programs to be executed. In some cases, they even allow multiple programs to be executed at the same time. Desktop personal computers, for example, are designed to run word processors, e-mail programs, spreadsheets and essentially any program that someone can conceive and write. This type of computer is a **general-purpose** computer.

Source Code

Source code is a program written in a programming language. **Programming languages** are made up of English-like words that communicate the instructions to the computer. Hundreds of programming languages have been developed for this purpose. Programming languages are designed to help humans with the process of programming computers. Programs written in computer languages must be converted into machine code and loaded into the computer memory. Figure 2.1 shows the relative spectrum of computer languages and provides some samples of how the code might read.

High-level languages are designed to be more removed from machine code and more like a spoken language. They have syntax similar to sentence structure, making them even easier to read. In a high-level language, single keywords can cause many machine-level instructions to be executed. Examples of high-level languages are BASIC,



ENGLISH	BASIC	C		ASSEMBLER		MACHINE	
if X is equal	if (X == Y) then	if (X == Y)		LDAA	X	96	00
to Y, then set	Z = X	Z = X;		CMPA	Y	91	01
Z equal to X,	else	else		BEQ	PAST	27	02
otherwise set	Z = Y	Z = Y;		LDAA	Y	96	01
Z equal to Y	end if		PAST	STAA	Z	97	02

Figure 2.1 Various Levels of Computer Code

FORTRAN and COBOL. High-level source code must be processed by a program called a **compiler** or **interpreter.** It translates the high-level language to the machine-level code that can be loaded into the processor on the target computer.

Assembly is the lowest level of all computer languages. **Assembly language** programming is a low-level programming language because it is most like the actual machine code. Assembly language source code is made up of a set of abbreviations, which are specific to a processor or processor family. The abbreviations are called mnemonics (pronounced ne-mon-icks). A **mnemonic** correlates to a single machine code instruction. The mnemonics are designed to be easy to read and easy to remember. They are a significant improvement over the binary machine codes. The assembly language source code is processed by a program called an **assembler,** which translates the mnemonics into the machine-level code that can be loaded into the computer.

In addition to low-level and high-level languages, some languages are in between. They are in between because they have characteristics of both types. These mid-level languages have features similar to those of the low-level languages, yet maintain the strengths of the high-level languages. Some examples of languages that fit this description are "C", "C++" and FORTH.

High-level and mid-level languages have an advantage. The same source code file written in a high- or mid-level language can be compiled to run on almost any hardware. This type of source code is called **portable**. Because assembly-level languages are so close to the actual machine code, portability is limited to processors within the same family. In most cases it is not portable at all.

On the other hand, low-level languages also have an advantage. Since the low-level source code is so similar to the actual machine code, assembly-level programming tends to result in machine code that is smaller and consequently runs faster than the same programs written in a higher-level language.

Machine Code

There is only one language that the computer hardware understands, machine language. The machine language is made up of machine code. The **machine code** is multibit binary codes that tell the computer the specific tasks it is to perform. Machine code is the lowest level of all computer languages. In general, it is very difficult for humans to communicate with computers solely using machine language. The groups of 1's and 0's tend to look alike, and the probability of error is very high. Other computer languages have been developed to aid humans when they program computers.

NOTE: Machine code is commonly referred to as object code, and a machine code program is referred to as an object code program. Because the term "machine code" implies low-level codes that the machine uses, it will be used exclusively in this text. To avoid confusion, the term "object code" will not be used in this text.

The machine code consists of operational codes (opcodes), operands (data) and addresses of the operands. The assemblers and compilers produce the machine code from the lines of code written in the programming language.

An **opcode** is a multibit code that identifies an instruction. Each opcode contains specific information about the instruction to be executed, as well as how to execute it. Many instructions can be implemented in a variety of ways on a computer. Each method requires a unique opcode; thus, a single instruction will be implemented with one or more opcodes.

An **operand** is data that is operated upon by the instruction. For example, an addition instruction adds operands, and a load instruction reads an operand from an address in memory and loads it into a processor register. The **operand field** is a group of up to three bytes following the opcode. The operand field contains bytes necessary to complete the instruction. It can contain 8-bit and 16-bit data, 8-bit addresses and 16-bit addresses. In some cases, the operand field contains nothing at all, because all necessary data is already in the processor registers.

NOTE: The term "operand" is not consistently used throughout computer literature and is often used to refer to any bytes that occupy the operand field. This text will use the term operand in its literal sense, as the data that is operated upon by the instruction. This usage is consistent with the Motorola HC11 data manuals. Addresses and offsets will not be referred to as operands in the text, although they all occupy bytes in the operand field and are commonly referred to this way.

An **address** is a pointer into memory. Each memory location has a unique address by which it is identified. Most small computers have a byte-oriented memory; therefore, each address points to the location of a byte of data in memory. Addresses can be expressed in an absolute sense. An absolute address is a complete address, or an address that consists of all the address bits. Addresses can also be expressed as an offset. An **offset** is a value that is added to a base address to reference another memory location.

Offsets are commonly used to allow addressing of memory without specific reference to the entire address. Programs that use offsets typically result in less actual machine code because the offsets are smaller than complete addresses. Since these programs occupy fewer bytes of memory, often they will execute faster.

HC11 Machine Code

HC11 machine code is made up of opcodes, operands and addresses. The HC11 uses an 8-bit opcode and supports various ways of accessing operands in memory. Figure 2.2 illustrates three examples of the relationship of the opcode to the bytes in the operand field. Further examples of the relationship of opcodes, the operand field, operands and mnemonic instructions will be provided in sections 2.3 and 2.4 of this chapter.

Since the HC11 uses an 8-bit opcode, 256 unique opcode values can be generated (2^8 = 256). This is the maximum number of unique binary values that can be derived from an 8-bit binary code. Thus, the HC11 should be limited to 256 instruction opcodes, yet

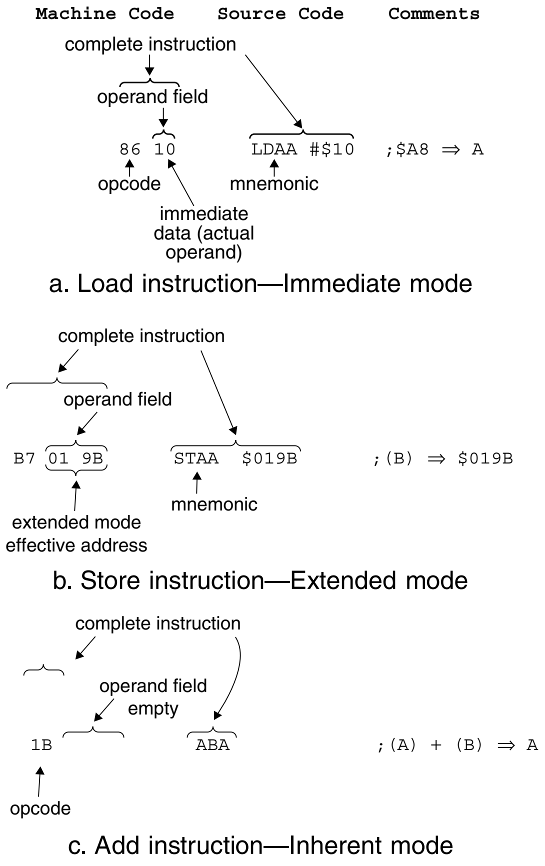


Figure 2.2 Opcode and Operand Field Examples

the HC11 has 308. The designers of the HC11 built in a mechanism that allows for each of the 308 instruction opcodes to be properly handled. Some of these instructions require an additional opcode byte, which is called a **prebyte**. The prebyte directs the instruction decoder to perform two fetch cycles. Three of the opcodes are actually opcode prebytes: $18, $1A and $CD. Thus, every instruction that uses a prebyte also has an additional opcode. Appendix C provides a complete listing of all opcodes and prebytes used on the HC11.

Self-Test Questions 2.1

1. What is a computer program?
2. What is the difference between source code and machine code?
3. What is the operand and how is it used?
4. What is an opcode prebyte?

2.2 Memory Addressing Modes

An **addressing mode** is a method of accessing operands. The addressing mode defines how the instruction will be coded and how it will operate to access and process data. Many instructions can operate in different modes, yet some operate only in a single mode. The addressing mode also defines how the effective address of an operand will be generated. The **effective address** is the address of the operand in memory. Each addressing mode accesses the effective address in a different manner. For example, the effective address for an immediate mode instruction is the address of the data following the opcode. Furthermore, an indexed mode instruction must calculate the effective address from the contents of the index register and the 8-bit offset provided in the instruction.

The application of the addressing modes on the HC11 is presented in section 2.3.

Methods of Addressing Memory

Every memory location in memory has a unique address. When data is needed from memory to complete an instruction, the processor generates the effective address and accesses the data stored at that location. There are many methods used by most computers to produce the effective addresses of data in memory. Five methods will be described in the following sections.

Immediate

Immediate addressing is used when the operand immediately follows the instruction opcode in memory. Thus, the memory access is limited to those locations immediately following the instruction opcode.

A number may be needed in a processor to complete a set of instructions. If this number is always the same value, it could be easily loaded into the register using the

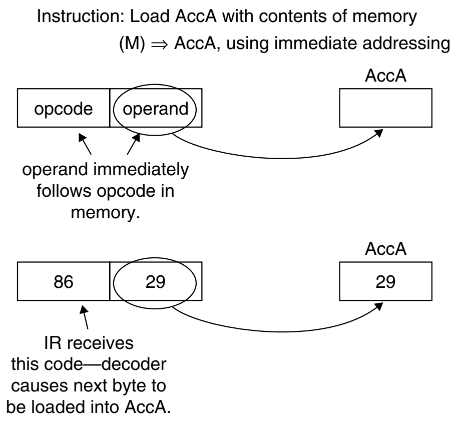

Instruction: Load AccA with contents of memory
(M) ⇒ AccA, using immediate addressing
AccA
opcode
operand
operand immediately follows opcode in memory.
AccA
86
29
29
IR receives this code—decoder causes next byte to be loaded into AccA.

Figure 2.3 Loading AccA Using Immediate Addressing

immediate mode. The number actually resides in the memory location immediately following the instruction opcode, as shown in Figure 2.3.

Absolute

Absolute addressing is used to access the operand directly from memory. The address of the operand follows the instruction opcode in memory. This mode is implemented on different processors in several forms.

Assume for a minute that an operand is in memory location $0012. This address would follow the instruction opcode in memory. When the instruction is executed, the address is retrieved from the memory and then used as the address of the operand. Additional machine cycles must be executed to then retrieve the actual operand that is processed, as shown in Figure 2.4.

Implied

In **implied addressing**, no address is necessary because the location of the data is implied by the instruction. For example, if data already exists in two separate accumulators in the processor, this data can be directly added without accessing memory. The instruction that would be used to add the contents of these two accumulators together would be implemented in the implied mode, since the data is implied as shown in Figure 2.5. Because the data is already provided in the two accumulators, the data does not need to be retrieved from memory before the addition operation is completed. The instruction is inherently complete, by itself.

Indexed

Indexed addressing uses an address in a register called an index register combined with an address offset to determine the location of the operand in memory. The

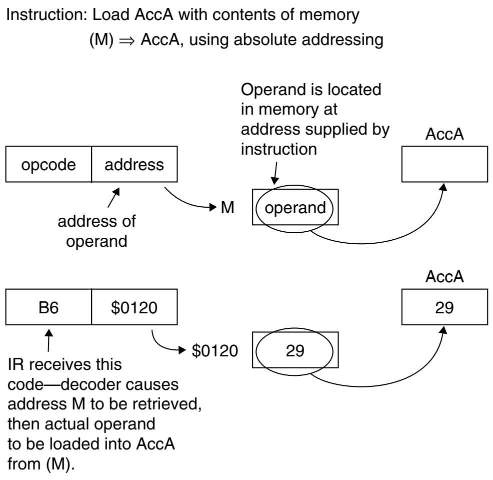

Instruction: Load AccA with contents of memory
(M) ⇒ AccA, using absolute addressing
Operand is located in memory at address supplied by instruction
AccA
opcode
address
M
operand
address of operand
AccA
B6
$0120
29
$0120
29
IR receives this code—decoder causes address M to be retrieved, then actual operand to be loaded into AccA from (M).

Figure 2.4 Loading AccA Using Absolute Addressing

instruction indicates which register to use, and the address offset follows the instruction opcode in memory. The address of the operand is calculated by adding the address offset to the address from the index register or by indexing the base address contained in the register. This addressing mode allows for simple access to data stored in a table. A detailed presentation of the HC11 implementation of indexed addressing mode is provided in chapter 5.

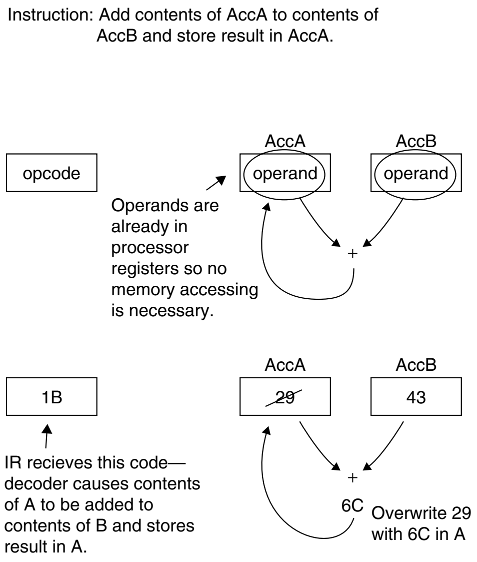

Instruction: Add contents of AccA to contents of AccB and store result in AccA.
opcode
AccA
operand
AccB
operand
Operands are already in processor registers so no memory accessing is necessary.
+
1B
AccA
~~29~~
AccB
43
+
6C
Overwrite 29 with 6C in A
IR recieves this code—decoder causes contents of A to be added to contents of B and stores result in A.

Figure 2.5 Adding AccA to AccB Using Inherent Addressing

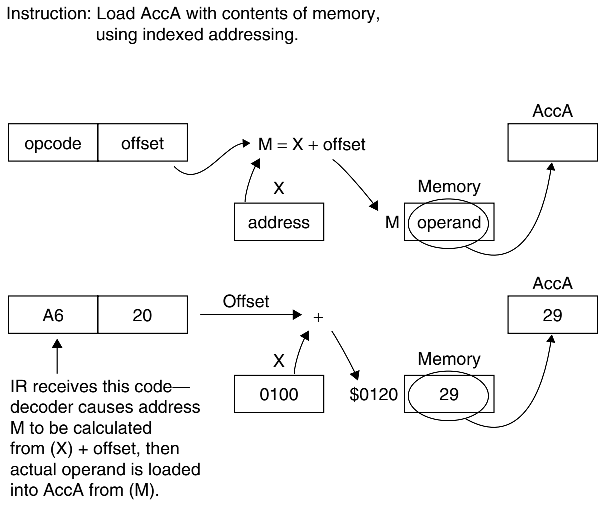

Instruction: Load AccA with contents of memory, using indexed addressing.
opcode
offset
M = X + offset
X
address
Memory
M
operand
AccA
A6
20
Offset
+
X
0100
Memory
$0120
29
AccA
29
IR receives this code—decoder causes address M to be calculated from (X) + offset, then actual operand is loaded into AccA from (M).

Figure 2.6 Loading AccA Using Indexed Addressing

Assume for a minute that an operand is in memory location $1012 and that the address $1000 is contained in a processor index register. In this case an address offset of $12 would follow the instruction opcode in memory. When the instruction is executed, the address offset is retrieved from the memory and then added to the address in the index register to calculate the address of the operand ($1012). This calculated address would then be used to access the operand in memory, as shown in Figure 2.6.

Relative

Relative addressing is used to change the flow of the program so that the instructions do not have to be laid out in a sequential manner. The address of the next instruction is described relative to the current position in memory. Instructions that use the relative mode express the location of the next instruction with an address displacement that is added to the current memory location. Thus, the next location is relative to the current position and not an address, which is absolutely specified. This addressing mode is used only to express the location of the next instruction to execute, rather than the location of an operand in memory. The displacement is a signed 2's complement value, which occupies a single byte. Since the displacement is a signed value, the flow of the program can change to instructions before or after the current instruction in memory.

Assume for a minute that a relative mode instruction is located at $01A7 and that the next instruction is located at $01A9. When the relative mode instruction is executed, the address offset is retrieved from the memory and used to calculate the address of the next instruction to execute. The reference point is always the address of the next instruction in memory. Therefore, as shown in Figure 2.7, the address $01A9 is the reference point, and the new address is calculated relative to this position.

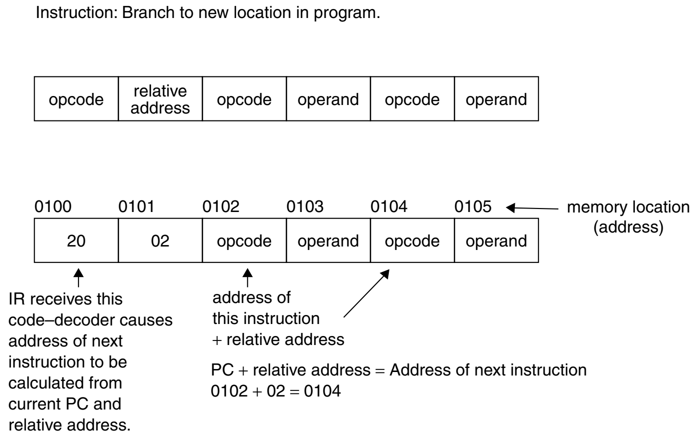

Instruction: Branch to new location in program.
opcode
relative address
opcode
operand
opcode
operand
0100
0101
0102
0103
0104
0105
memory location (address)
20
02
opcode
operand
opcode
operand
IR receives this code–decoder causes address of next instruction to be calculated from current PC and relative address.
address of this instruction + relative address
PC + relative address = Address of next instruction
0102 + 02 = 0104

Figure 2.7 Branching Using Relative Addressing

Self-Test Questions 2.2

1. What are the five addressing modes used on computers?
2. What is the difference between absolute and relative addressing?
3. Which addressing mode is used to change the flow of a program?

2.3 HC11 Addressing Modes

The HC11 utilizes six addressing modes: inherent, immediate, extended, direct, indexed and relative. The indexed addressing mode is implemented using either index register and is often referred to as two separate modes: indexed X and indexed Y. Each mode has an abbreviation that is used throughout the Motorola documentation, as summarized in Figure 2.8. These abbreviations are also used throughout this text.

8-bit versus 16-bit Memory Access

On the HC11, most data is stored as bytes. Each byte can be accessed in memory via a single read or write operation. However, 16-bit data and addresses are needed in most programs. Since data is accessed in memory via single-byte read/write operations, the 16-bit data and addresses are divided into a high byte and a low byte. The **high byte** is the upper 8 bits of the 16-bit word, and the **low byte** is the lower 8 bits.

Motorola uses a "hi-byte first" convention for all 16-bit storage. "**Hi-byte first**" means the high byte of a 16-bit data word or address will be stored in the first address, followed in memory by the low byte. Instructions that use extended addressing mode or 16-bit operands have the need to access information stored in memory in this format. Other manufacturers use a "low-byte first" convention.

Mode Name	Abbreviation	Operand Field	Description
Inherent	INH	Empty	Memory access is not required. Operand is implied by the instruction. Operand field is empty.
Immediate	IMM	ii	Operand is contained in the memory location(s) following opcode (ii), which is the operand field.
Extended	EXT	hhll	Operand contained in memory location pointed to by \$hhll, where hh is the high-order byte and ll is the low-order byte of the 16-bit absolute address. Both hh and ll are located in the operand field.
Direct	DIR	dd	Operand contained in memory location pointed to by \$00dd, where dd is the direct mode address located in the operand field.
Indexed	INDX INDY	ff	Operand contained in the memory location addressed by the contents of the index register + ff, where ff is the 8-bit unsigned index mode offset located in the operand field.
Relative	REL	rr	If branch test passes, program execution will proceed at PC+\$rr, where \$rr is the signed 8-bit relative mode displacement located in the operand field. If the branch test fails, it will execute the next instruction.

Figure 2.8 Summary of HC11 Addressing Modes

Motorola uses the capital letter "M" to designate a memory address. When the two memory locations are required to access two bytes of data, M:M+1 is used to designate the addresses of two consecutive locations. In this case, M is the address of the high byte and M+1 is the address of the low byte. M+1 indicates that the low byte is stored in the memory address immediately following (one more than M) the high byte.

Transfer Notation

When the contents of a register are being read as part of the operation, the location being read is shown in parentheses. For example, the contents of AccA are shown as (A). The contents of memory location \$0120 are shown as (\$0120). The "contents of" notation only refers to reading data from a location. When data is being written to a location, the name of the location is used without the parentheses. For example, "store the contents of AccA into memory location \$B600" is written as (A) → \$B600; "load AccA with the contents of location \$0002" would be written as (\$0002) → A.

NOTE: Instructions will be commented using transfer notation, where the parentheses () indicate "contents of" and → indicates the direction of the data movement.

Immediate (IMM)

The immediate mode is used when the operand is a constant. The actual operand occupies the operand field of the machine code instruction. Thus, the effective address

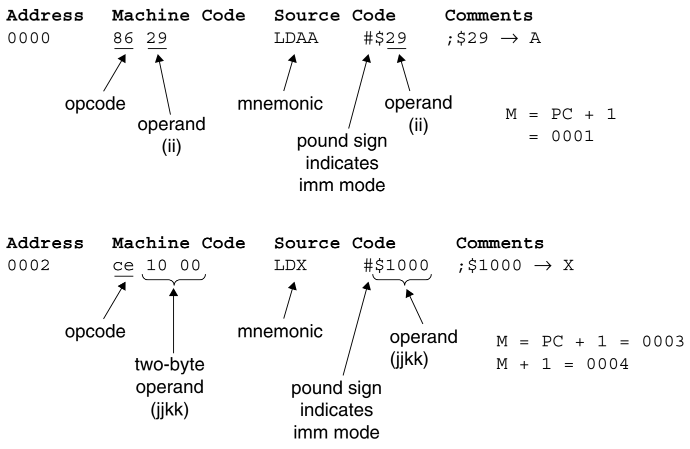


Figure 2.9 Examples of HC11 Immediate Mode Instructions

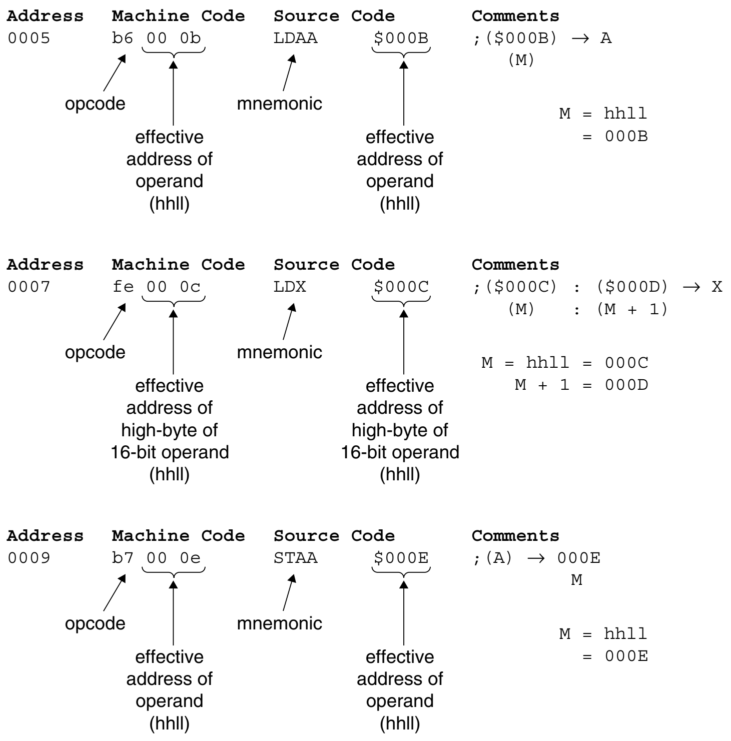
of the operand is the memory address immediately following the opcode. When the immediate operand is two bytes for 16-bit operations, these bytes occupy the two memory locations immediately following the opcode in memory. Examples of HC11 immediate mode instructions are shown in Figure 2.9.

Extended (EXT)

The HC11 extended addressing mode is the name Motorola gave to this absolute addressing mode. The extended mode is used when the operand is anywhere within the 64K memory map, $0000–$FFFF. The high-order and low-order bytes of the effective address occupy the operand field of the machine code instruction. The high-order byte of the effective address is always the first of the two bytes followed by the low-order. This high-order, low-order convention is strictly followed by functions within the HC11. To complete the execution of an extended mode instruction, both bytes must be retrieved from the operand field and joined to form the effective address of the operand. This effective address is used to access the actual operand. Examples of HC11 extended mode instructions are shown in Figure 2.10.

Direct (DIR)

The HC11 direct mode is a form of absolute addressing. It uses an 8-bit absolute address. This 8-bit address becomes the low-order byte of the effective address, and the high-order byte of the effective address is $00. The direct mode is used on the HC11 only when the operand is located within the first 256 locations of the memory map, $0000–$00FF. Thus, the direct mode is sometimes called zero-page addressing. The low-order byte of the effective address occupies the operand field of the machine code instruction. To complete the execution of a direct mode instruction, the low-order byte of the effective address must be retrieved from the operand field and the effective address must be calculated, then the actual operand can be accessed using this effective



```
Address   Machine Code   Source Code       Comments
0005      b6 00 0b       LDAA    $000B     ;($000B) → A
                                              (M)

                                                 M = hhll
                                                   = 000B

Address   Machine Code   Source Code       Comments
0007      fe 00 0c       LDX     $000C     ;($000C) : ($000D) → X
                                              (M)    : (M + 1)

                                            M = hhll = 000C
                                               M + 1 = 000D

Address   Machine Code   Source Code       Comments
0009      b7 00 0e       STAA    $000E     ;(A) → 000E
                                                    M

                                                 M = hhll
                                                   = 000E
```

opcode
mnemonic
effective address of operand (hhll)
effective address of operand (hhll)
opcode
mnemonic
effective address of high-byte of 16-bit operand (hhll)
effective address of high-byte of 16-bit operand (hhll)
opcode
mnemonic
effective address of operand (hhll)
effective address of operand (hhll)

Figure 2.10 Examples of HC11 Extended Mode Instructions

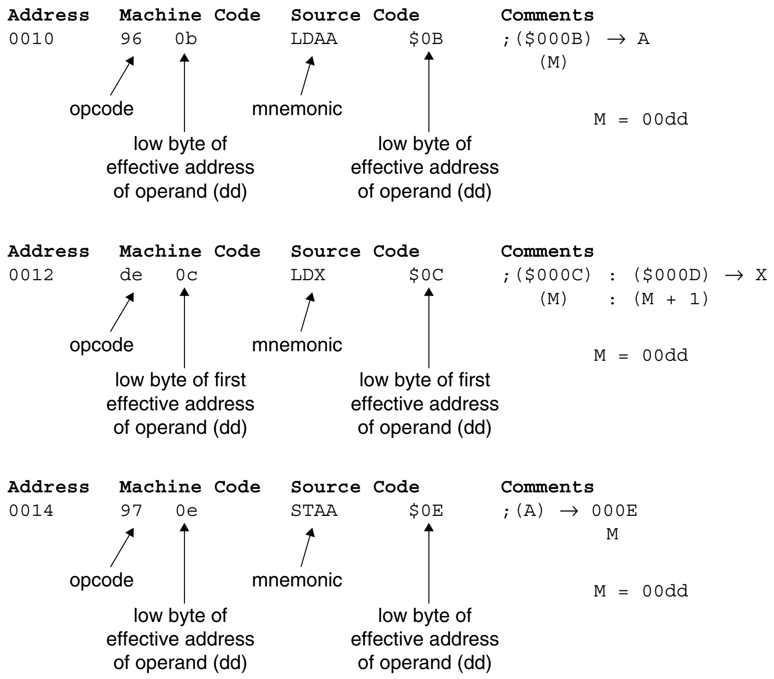
address. Functionally the direct mode is identical to the extended mode, but it executes faster since only one byte of address must be accessed from the operand field to calculate the effective address of the operand. Examples of HC11 direct mode instructions are shown in Figure 2.11.

Inherent (INH)

The HC11 inherent mode is used when the operand is inherent or implied. The operands are already contained in processor registers prior to execution of an inherent mode instruction. The operand field of an inherent mode instruction is empty. Examples of HC11 inherent mode instructions are shown in Figure 2.12.

Indexed (INDX, INDY)

The HC11 indexed mode is used when the operand may be at varying addresses within the 64K memory map, \$0000–\$FFFF. It allows indexing from a reference base address contained in one of the 16-bit index registers. An 8-bit offset is contained in the operand field of the indexed mode instruction. The effective address is the sum of the address contained in the designated index register and this indexed mode offset.

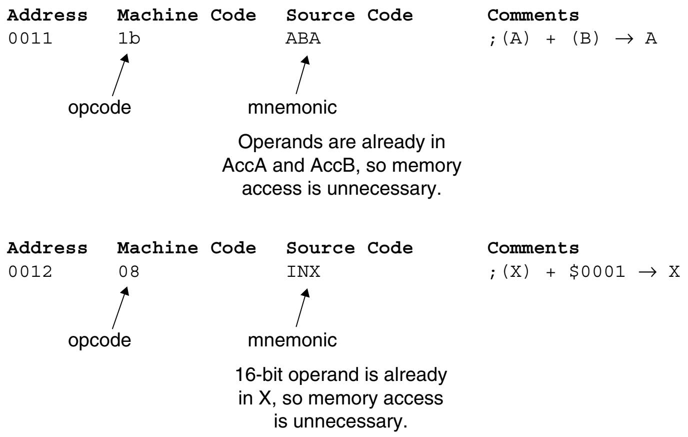


```
Address   Machine Code   Source Code       Comments
0010      96   0b        LDAA      $0B     ;($000B) → A
                                              (M)
                                                    M = 00dd

Address   Machine Code   Source Code       Comments
0012      de   0c        LDX       $0C     ;($000C) : ($000D) → X
                                              (M)    : (M + 1)
                                                    M = 00dd

Address   Machine Code   Source Code       Comments
0014      97   0e        STAA      $0E     ;(A) → 000E
                                                  M
                                                    M = 00dd
```

opcode
low byte of effective address of operand (dd)
mnemonic
low byte of effective address of operand (dd)
opcode
low byte of first effective address of operand (dd)
mnemonic
low byte of first effective address of operand (dd)
opcode
low byte of effective address of operand (dd)
mnemonic
low byte of effective address of operand (dd)

Figure 2.11 Examples of HC11 Direct Mode Instructions

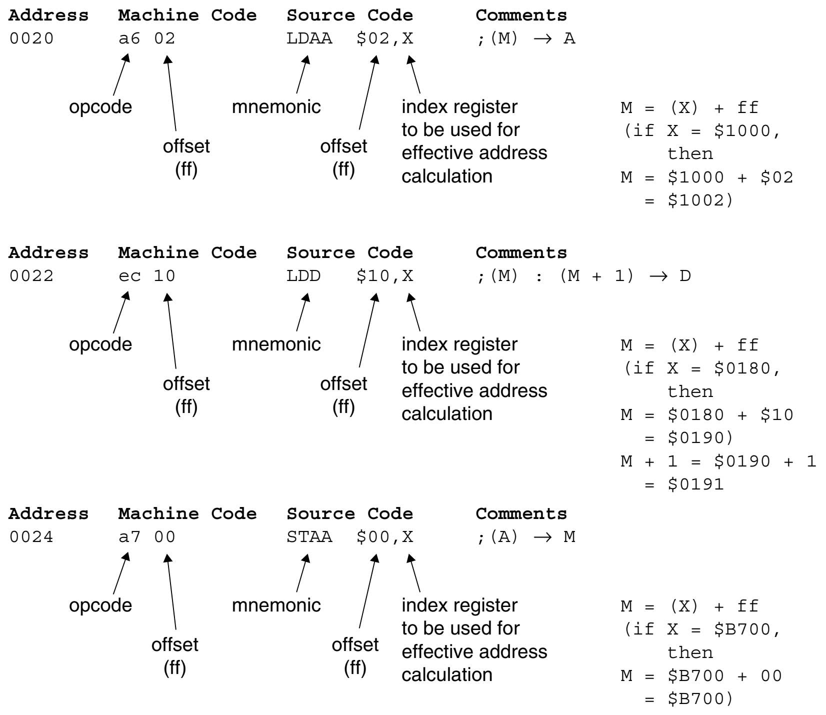
To complete the execution of an indexed mode instruction, the offset must be retrieved from the operand field. The effective address must be calculated using the contents of the index register plus the index offset; then the actual operand is accessed using this effective address. The contents of the index register are not changed when this address is calculated. The calculated address is kept in an internal temporary register. Indexed



```
Address   Machine Code   Source Code       Comments
0011      1b             ABA               ;(A) + (B) → A

Address   Machine Code   Source Code       Comments
0012      08             INX               ;(X) + $0001 → X
```

opcode
mnemonic
Operands are already in AccA and AccB, so memory access is unnecessary.
opcode
mnemonic
16-bit operand is already in X, so memory access is unnecessary.

Figure 2.12 Examples of HC11 Inherent Mode Instructions



Address Machine Code Source Code Comments
0020 a6 02 LDAA $02,X ;(M) → A
opcode
offset (ff)
mnemonic
offset (ff)
index register to be used for effective address calculation
M = (X) + ff (if X = $1000, then M = $1000 + $02 = $1002)
Address Machine Code Source Code Comments
0022 ec 10 LDD $10,X ;(M) : (M + 1) → D
opcode
offset (ff)
mnemonic
offset (ff)
index register to be used for effective address calculation
M = (X) + ff (if X = $0180, then M = $0180 + $10 = $0190)
M + 1 = $0190 + 1 = $0191
Address Machine Code Source Code Comments
0024 a7 00 STAA $00,X ;(A) → M
opcode
offset (ff)
mnemonic
offset (ff)
index register to be used for effective address calculation
M = (X) + ff (if X = $B700, then M = $B700 + 00 = $B700)

Figure 2.13 Examples of HC11 Indexed Mode Instructions

mode instructions are easily found in the code to their unique format. The instruction is followed by the offset (ff), a comma and a reference to either of the index registers (X or Y). Examples of HC11 indexed mode instructions are shown in Figure 2.13.

Relative (REL)

On the HC11, the relative mode is used only for branch instructions. Relative mode instructions do not process data, but control the flow of the program. If the branch test passes, the 8-bit relative address is sign extended to 16 bits and added to the contents of the program counter to form the effective branch address. If the branch test fails, nothing is added to the contents of the program counter. Execution thus proceeds with the next instruction in sequence. No examples will be shown at this point in the text.

Self-Test Questions 2.3

1. What are the six addressing modes used on the HC11? How do they relate to the five addressing modes present on most computers?
2. What is the difference between HC11 extended and direct addressing modes?
3. What is the difference between INDX and INDY?

2.4 Processing Instructions

When the processor comes out of reset, it initializes the program counter, then immediately begins to execute instructions. The first step is to read an instruction opcode from memory. The opcode is an 8-bit code that tells the processor which instruction to execute and which addressing mode to use. Once the opcode has been fetched from memory, it is decoded by the instruction decoder. After it is decoded, the processor is ready to complete the steps necessary to execute the instruction. The decoding process selects the specific set of steps that will be followed, which internal registers to use and which addressing mode to use. The processor does not "think" about what it has to do; it just methodically executes the instructions, based on the built-in sequence for the specific instruction.

NOTE: Before the execution of an instruction is considered complete, the program counter has been updated to point to the address of the next instruction and the CCR has been updated to reflect the status of the operation.

Machine Cycles

Each instruction is executed by a series of machine cycles (two or more). A **machine cycle** is a short set of steps that are performed during a single clock cycle. The HC11 performs a four-step process during each machine cycle. This four-step process is generated by a two-phase non-overlapping clock, as shown in Figure 2.14. Phase 1 (PH1) controls steps one and two of the cycle. They have to do with the use of the program counter. Phase 2 (PH2) controls the last two steps. They have to do with memory access and other processing functions necessary to complete the instruction. Each step of the cycle is driven by one of the edges of PH1 and PH2. These edges control the sequence of events during each cycle.

NOTE: Nothing happens in a microprocessor unless there is an edge present from a clock.

Fetch Cycles

The job of the **fetch cycle** is to read the instruction opcode from the memory. The instruction opcode is always located in memory at the address that is contained in the

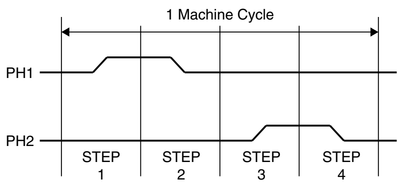


Figure 2.14 HC11 Four-Step Machine Cycle

program counter. A fetch cycle is the first machine cycle for each instruction. A new fetch cycle is started automatically following the completion of the previous instruction.

The fetch cycle starts by moving the contents of the program counter to the memory address register. Then the program counter is incremented so that it is always looking at the next location in the program. The address in the memory address register is driven out onto the address bus. This is the address of the instruction opcode in memory. The memory is enabled during the third step. The memory location responds to the enable by placing the opcode onto the data bus. The fourth step of the fetch cycle loads the opcode from the data bus into the Instruction Register, where it is immediately decoded.

NOTE: Some HC11 instructions utilize an opcode prebyte; thus, two fetch cycles are required. Instruction execution cannot begin until both the prebyte and the actual opcode have been fetched from memory. The prebyte indicates to the processor that another opcode is necessary before the execution can begin.

Figure 2.15 illustrates the events of the fetch cycle for the immediate mode instruction LDAA #$29. The opcode for this instruction is $86. It is the job of the fetch cycle to read this opcode from memory and load it into the instruction register for decoding. The first step consists of copying the contents of the program counter into the memory address register ($0000 → MAR). Then the program counter is incremented ($0001 → PC). The third step enables the memory location, which causes the memory to drive the opcode onto the data bus ($86 → Data Bus). Finally, the opcode is loaded into the instruction register ($86 → IR) where it is decoded. This completes the fetch cycle.

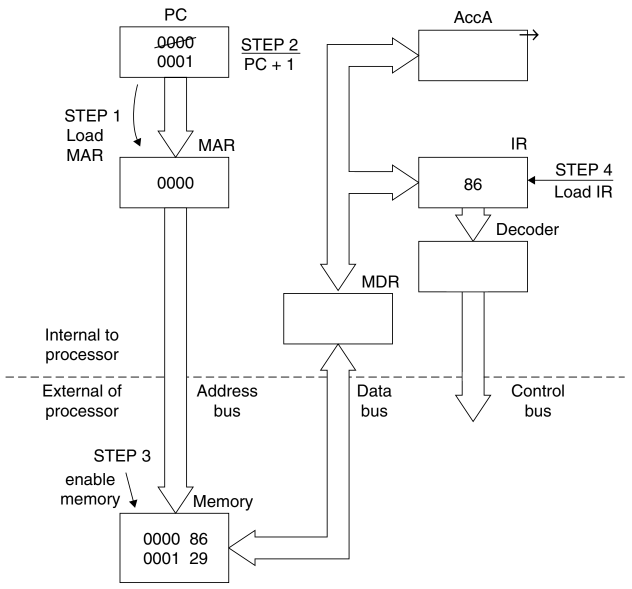


Figure 2.15 Fetch Cycle for LDAA #$29

Execute Cycles

After the opcode is decoded, the processor methodically executes the instruction. The execution requires one or more machine cycles, which are called execute cycles. Each **execute cycle** is a built-in sequence of events that completes the operation of the instruction. Instruction execution is unique for each instruction. The process that is followed is dependent not only upon the instruction type but also upon the addressing mode used. The first job is to locate the operands. Inherent mode instructions do not require this step because the operands are already available in the processor registers, if any are required. Immediate mode instructions provide operands in the memory location immediately following the opcode for the instruction. Direct, extended, indexed and relative mode instructions must gather address information from the operand field before the operands can be accessed and processed.

Figure 2.16 illustrates the steps that must be followed to execute the LDAA #$29 instruction that was fetched in Figure 2.15. The first step is to copy to contents of the program counter to the memory address register ((PC) → MAR). This places the address $0001 in the memory address register. Since this is an immediate mode instruction, this address represents the effective address of the operand. The next step is to increment the program counter ((PC) + 1 → PC) so that it is properly indicating the next memory location in sequence. The next step is to enable memory location $0001. It responds by putting the operand on the data bus. To complete the execution of the instruction, the operand ($29) is loaded from the data bus into AccA and the status flags in the CCR are updated. The LDAA #$29 instruction requires only one execute cycle to complete the load operation.

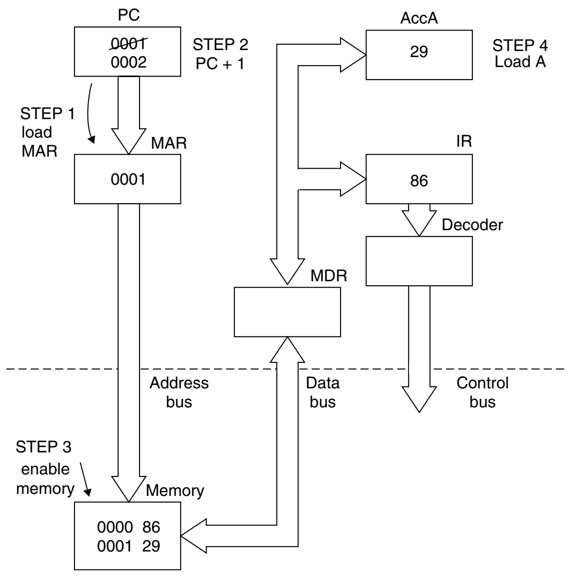


Figure 2.16 Execute Cycle for LDAA #$29

NOTE: Each instruction requires at least two machine cycles to complete, one fetch cycle and one execute cycle. Some require as many as 41, 1 fetch and 40 executes. Assume that memory location $019B contains $36 before the start of this program.

Example 2.1

Problem: Using the following four-instruction program. Describe what the processor does to complete the necessary fetch and execute cycles.

```
Address    Machine Code   Source Code    Comments
 0100       86 a8          LDAA  #$A8     ;$A8  →  A
 0102       f6 01 9b       LDAB  $019B    ;($019B)  →  B
 0105       1b             ABA            ;(A) + (B)  →  A
 0106       97 20          STAA  $20      ;(A)  →  $0020
```

Solution:

Operation	Description	Functional Result	
Fetch LDAA opcode	Contents of PC are copied to MAR.	(PC) → MAR	$0100 → MAR
(1 machine cycle)	PC is incremented.	(PC) + 1 → PC	$0101 → PC
	Opcode is read from memory.	($0100) → data bus	$86 → data bus
	Opcode is loaded into IR.	Opcode → IR	$86 → IR
Execute LDAA	Contents of PC are copied to MAR.	(PC) → MAR	$0101 → MAR
(1 machine cycle)	PC is incremented.	(PC) + 1 → PC	$0102 → PC
	Operand is read from memory.	($0101) → data bus	$A8 → data bus
	Operand is loaded into AccA.	Operand → A	$A8 → A
Fetch LDAB opcode	Contents of PC are copied to MAR.	(PC) → MAR	$0102 → MAR
(1 machine cycle)	PC is incremented.	(PC) + 1 → PC	$0103 → PC
	Opcode is read from memory.	($0102) → data bus	$F6 → data bus
	Opcode is loaded into IR.	Opcode → IR	$F6 → IR
Execute LDAB	Contents of PC are copied to MAR.	(PC) → MAR	$0103 → MAR
(3 machine cycles)	PC is incremented.	(PC) + 1 → PC	$0104 → PC
	Hi-byte of effective address (hh) is read from memory.	($0103) → data bus	$01 → data bus
	hh is loaded into Temp16H.	hh → Temp16H	$01 → Temp16H
	Contents of PC are copied to MAR.	(PC) → MAR	$0104 → MAR
	PC is incremented.	(PC) + 1 → PC	$0105 → PC
	Lo-byte of effective address (ll) is read from memory.	($0104) → data bus	$9B → data bus
	ll is loaded into Temp16L.	ll → Temp16L	$9B → Temp16L
	Contents of Temp16 are copied to MAR.	(Temp16) → MAR	$019B → MAR
	Operand is read from memory.	($019B) → data bus	$36 → data bus
	Operand is loaded into AccB.	Operand → B	$36 → B
Fetch ABA opcode	Contents of PC are copied to MAR.	(PC) → MAR	$0105 → MAR
(1 machine cycle)	PC is incremented.	(PC) + 1 → PC	$0106 → PC
	Opcode is read from memory.	($0106) → data bus	$1B → data bus
	Opcode is loaded into IR.	Opcode → IR	$1B → IR

Operation	Description	Functional Result	
Execute ABA (1 machine cycle)	Operands are read from registers and added.	(A) + (B) → data bus	\$A8 + \$36 → data bus
	Result is loaded into AccA.	Result → A	\$DE → A
Fetch STAA opcode (1 machine cycle)	Contents of PC are copied to MAR.	(PC) → MAR	\$0106 → MAR
	PC is incremented.	(PC) + 1 → PC	\$0107 → PC
	Opcode is read from memory.	(\$0106) → data bus	\$97 → data bus
	Opcode is loaded into IR.	Opcode → IR	\$97 → IR
Execute STAA (2 machine cycles)	Contents of PC are copied to MAR.	(PC) → MAR	\$0107 → MAR
	PC is incremented.	(PC) + 1 → PC	\$0108 → PC
	Lo-byte of effective address (dd) is read from memory.	(\$0107) → data bus	\$20 → data bus
	dd is loaded into Temp16L.	dd → Temp16L	\$20 → Temp16L
	Temp16H is cleared.	\$00 → Temp16H	
	Contents of Temp16 are copied to MAR.	(Temp16) → MAR	\$0020 → MAR
	Operand is retrieved from AccA.	(A) → data bus	\$DE → data bus
	Operand is loaded into memory.	Operand → M	\$DE → \$0020

Self-Test Questions 2.4

1. What is the function of a fetch cycle?
2. What takes place during the execute cycles of an instruction?
3. How does Motorola store 16-bit data in memory?

2.5 Program Flow

Programs can be written to perform a variety of tasks. The flow of a program is determined by the instructions used to implement the program. Often software defects are caused by a flaw in the flow of a program. A **defect** is an undesirable behavior of the software. Software defects are often referred to as "**bugs.**" The best way to write a program is to first document what the program is going to do. A good way of doing this is by drawing a flowchart.

A **flowchart** is a means of organizing the flow of a program. It is used to show the function and flow of a program. It is a graphical representation of processes, input and output of data, decisions and loops. Each function performed is represented by a symbol in the chart. Figure 2.17 shows some basic flowcharting symbols. Each symbol represents specific program functions.

The rounded rectangle is used for program termination. It is used at the beginning and end of each program. The rectangle is used as a process box. A description of the process is included in the box. The parallelogram (leaning rectangle) is used for input and output to and from the system. The diamond is used for a decision. The decision is shown as an expression inside the diamond. The rectangle with stripes is used to show a subroutine (a process accomplished by a routine outside the main body of the program). The circle is used as an on-page connector, and the home-plate (pentagon)

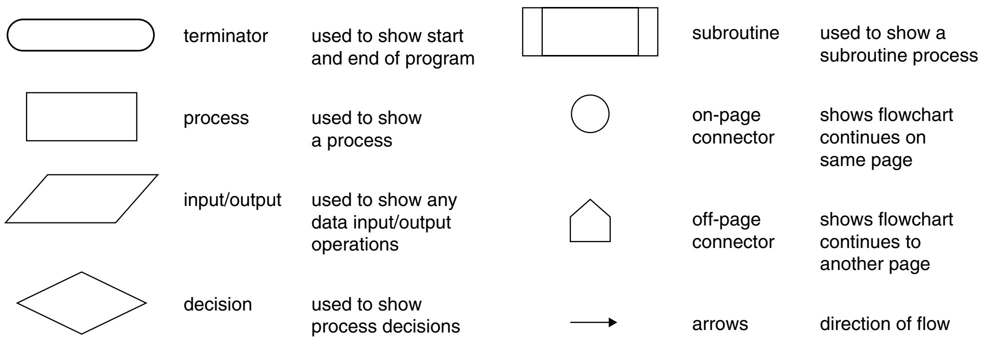

terminator
used to show start and end of program
process
used to show a process
input/output
used to show any data input/output operations
decision
used to show process decisions
subroutine
used to show a subroutine process
on-page connector
shows flowchart continues on same page
off-page connector
shows flowchart continues to another page
arrows
direction of flow

Figure 2.17 Basic Flowcharting Symbols

indicates the flowchart continues onto another page. The arrows indicate the direction of the program flow.

Flowcharts can be used to show the flow of any operation in any environment, yet the presentation in this text will be limited to software functions.

Example 2.2

Problem: Draw a flowchart for answering the phone after it rings three times.

Solution: The flowchart is shown in Figure 2.18. The terminator symbol is used to start the program. The first thing that needs to be done is to count the ring. If there have been less than three rings, keep counting; otherwise, answer the phone. Say "hello." If it is a salesman, hang up; otherwise have a conversation, say "goodbye" and then hang up.

Example 2.3

Problem: Draw a flowchart for a 10 ms time delay.

Solution: The flowchart is shown in Figure 2.19. The terminator symbol is used to start and end the flowchart. First a counter has to be loaded with a value equivalent with the 10 ms delay. Then the counter is decremented. If it is equal to zero, the delay is completed; otherwise, go back and decrement the counter again and continue this process until it equals zero.

Self-Test Questions 2.5

1. What is the purpose of a flowchart?
2. What symbol is used to show a decision?
3. What symbol is used to show a subroutine process?

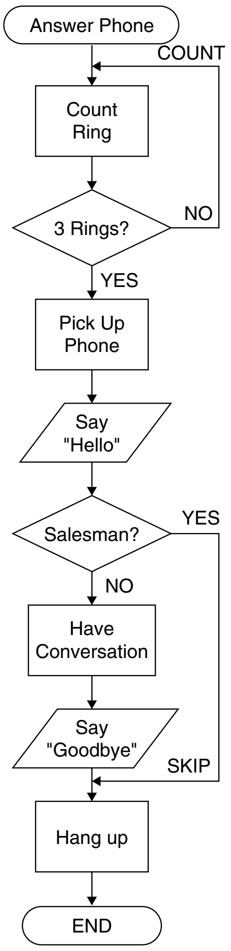


Figure 2.18 Flowchart for Answering the Phone after Three Rings

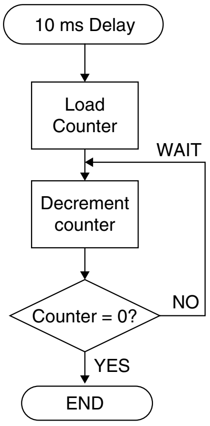


Figure 2.19 Flowchart of a 10 ms Time Delay

Summary

This chapter contains a discussion of the role of software on a computer. Software programs are ordered sets of instructions that control the hardware. Software must be written by programmers as source code. It is written in a variety of programming languages like assembly, FORTRAN, BASIC and "C." The source code uses English-like keywords and abbreviations called mnemonics to form a structured set of operations for the hardware to perform. The source code is converted into object or machine code by a program called an assembler or a compiler. The machine code contains only the opcodes, operands and addresses needed by the hardware to complete each instruction.

The processor must access memory on a regular basis to read and write the data that is processed by the instructions. Each processor supports several methods it can use to access the memory called memory addressing modes. Some of the addressing modes are called immediate mode, direct mode, indexed mode and relative mode. Some modes allow memory to be accessed with an absolute address and others use an address that is relative to the current location in the program or relative to the address stored in a register. The inherent mode is used for all instructions for which the operand is implied.

The processor runs a program sequentially. It first reads an instruction opcode from memory and decodes it and then performs the operation designated by the instruction opcode. When the processor reads the instruction opcode from memory, it is performing a *fetch* cycle. The data is processed during *execute* cycles. Every instruction involves at least one fetch cycle and at least one execute cycle. Some instructions require many execute cycles to complete the operation.

Chapter Questions

Section 2.1

1. What is the difference between a computer's hardware and its software?
2. Is assembly-level programming a high-level or a low-level programming language? Why?
3. What is the difference between source code and machine code?
4. Define the term *machine code.*
5. What is an opcode?
6. What is an operand?
7. What can be contained in the operand field?
8. What is a prebyte? What does it tell the processor to do?

Section 2.2

9. What is an effective address and how is it used?
10. When using the absolute addressing mode, where is the operand?
11. What is the difference between immediate and inherent addressing modes?
12. Which addressing mode is used when the data is implied?
13. What makes indexed addressing unique?
14. Relative addressing mode does not process data in the same sense as the other addressing modes. What does it do?

Section 2.3

15. How many addressing modes are provided on the HC11?
16. How does the HC11 implement the generalized absolute addressing method?
17. Which memory locations can be addressed by the HC11 direct mode?
18. Is there ever a situation where there are two effective addresses for a single instruction?
19. When using transfer notation, what symbols are used to show "contents of"?

Section 2.4

20. How many steps are performed during each machine cycle of the processor?

21. What happens to the PC during the second step of the fetch cycle?
22. After the processor fetches an opcode from memory, which processor register holds the opcode while it is being decoded?
23. Which register holds the address being currently used by the address bus?
24. How many fetch cycles are required by each instruction on the HC11?
25. How many execute cycles are required by each instruction on the HC11?

Section 2.5

26. What is a flowchart?
27. What symbol is used to indicate that a process is accomplished in a subroutine?

Chapter Problems

1. Identify the addressing mode used by each of the following instructions.
   a. LDAA $1002
   b. LDY #$B600
   c. ABA
   d. STAA $04,X
   e. BRA $FC
   f. ADDA $32
   g. BPL $02
   h. ADCA #$32
2. Calculate the effective address(es) used by each of the following instructions.
   a. LDAB $0C
   b. LDD $B600
   c. ABA
   d. ADDA $32
   e. LDAA $1032
3. Draw a flowchart that shows the process of checking the oil in a car.
4. Draw a flowchart of the process of taking a shower.
5. Draw a flowchart of the process of cooking burgers on the BBQ.

Answers to Self-Test Questions

Section 2.1

1. An ordered set of instructions to perform a task.
2. Source code contains the instructions written by the programmer. The machine code is the actual operational code and operands (in binary or hex) that execute on the machine.
3. The operand is the actual data that is processed by the instruction.
4. An opcode prebyte tells the instruction decoder that another opcode byte must be fetched from memory before the whole opcode can be interpreted.

Section 2.2

1. Implied/Inherent, Immediate, Absolute, Indexed and Relative.
2. When using absolute addressing mode, the address following the mnemonic is

the effective address of the operand. When using relative addressing mode, the value following the mnemonic is the displacement. The displacement is the relative distance to the next instruction.

3. Relative addressing mode is always used to change the flow of the program.

Section 2.3

1. The HC11 has an inherent mode (INH), an immediate mode (IMM), two absolute modes (EXT and DIR), two indexed modes (INDX and INDY) and a relative mode (REL).
2. Extended mode on the HC11 uses 16-bit addresses as the effective address in the instruction. The Direct mode on the HC11 uses an 8-bit address, which is used as the lower byte of the 16-bit address where the upper byte is zero.
3. INDX and INDY are both indexed addressing modes. INDX uses the X index register, and INDY uses the Y index register.

Section 2.4

1. The fetch cycle retrieves the instruction opcode from memory and places it into the instruction register.
2. The actual operation of the instruction is completed during the execute cycles of the instruction.
3. Motorola uses a high-byte first format. The high byte of a 2-byte data block is written to the first memory location and the low byte is written to the next location.

Section 2.5

1. A flowchart is a diagram that shows the function and flow of a program.
2. The diamond symbol is used to show a decision in the flowchart.
3. The subroutine symbol is a rectangle with an extra line on each end.

c h a p t e r 3

HC11 Programming

Objectives

After completing this chapter, you should be able to:

- Define the five Condition Code Register Status Flags
- Write simple programs that move data via load, store and transfer instructions
- Perform simple arithmetic and logic operations using a variety of add, subtract and logic instructions
- Illustrate how serial shifting of data is possible in software
- Perform multiplication and division operations
- Write short programs using INH, IMM, DIR and EXT addressing modes

Outline

Introduction

The **instructions** are the commands that control what the processor does. Without the instructions, the processor is nonfunctional. The instruction set on the HC11 consists of 308 unique opcodes derived from 145 mnemonic key words. Many of the mnemonic instructions operate in more than one addressing mode. Thus, 145 unique mnemonic forms can generate 308 opcodes. For example, the LDAA instruction can be implemented using the IMM, DIR, EXT, INDX and INDY addressing modes. Thus, five opcodes are required for the LDAA instruction so that each of these addressing modes can be uniquely expressed to the processor during opcode fetch. Each of these five implementations accomplishes the same task of loading a byte of data from memory to AccA; they just accomplish it in different ways.

The instruction set has been broken down by the author into several categories of similar function. Each group will be discussed as a set. Explanations pertaining to the entire group will be provided at the beginning of each major section. Instructions will then be described in small groups. A functional explanation of each instruction will be provided. In addition, each instruction presentation will include an explanation of the addressing modes in which the instruction operates, as well as the effect the instruction has on the CCR.

Although each addressing mode is explained, the examples and problems in this chapter are limited to the application of the INH, IMM, DIR and EXT modes. The application of instructions that use the REL, INDX and INDY modes are reserved for chapters 4 and 5.

The lower-case "x" is used in conjunction with the instruction mnemonics to indicate a wildcard. In each case, the x means that the mnemonic has several variations that support different registers within the processor register block. The register designation is substituted in place of the x to form a specific mnemonic that performs a function on a single register. For example, LDAx can have the form LDAA or LDAB, indicating load accumulator A or load accumulator B. Another example is the STx instructions that can have the forms STD, STS, STX, and STY. These instructions are used to store the contents of the double accumulator, the stack pointer register, the X index register or the Y index register into memory respectively. Figure 3.1 summarizes the registers and their corresponding single-letter designations.

NOTE: The P designation is sometimes confused for the program counter when it is used in the instruction mnemonic. Motorola uses the P to designate the CCR in just two instructions: TPA and TAP; the function of these instructions is explained later in this chapter.

The program counter register is not included in this table because it is not directly referenced by any instructions (i.e., there is not an instruction that allows the PC to be directly loaded or stored). In addition, some instruction mnemonics include references to bits in the CCR (i.e., SEC—set the carry flag in the CCR).

This chapter directly correlates to section 6.3 and Appendix A of the HC11 Reference Manual and Table 3-2 of the Technical Data Manual. Much of this chapter relies on the information on the instruction set that is contained in these sections.

Designation	Register	Register Size (bits)
A	Accumulator A	8
B	Accumulator B	8
P	Condition Code Register (CCR)	8
D	Double Accumulator D (A:B)	16
S	Stack Pointer Register	16
X	Index Register X	16
Y	Index Register Y	16

Figure 3.1 Register Designations Used by HC11 Instruction Mnemonics

3.1 Condition Code Register Status Flags

In chapter 1, the condition code register is introduced as one of the registers in the programmer's model. The instruction set descriptions, covered in this chapter and throughout the remainder of the text, provide further information on how the specific instructions affect these bits. This section provides additional insight into the function and use of these bits.

Figure 3.2 illustrates the layout of the HC11 Condition Code Register (CCR). Three bits are control bits (S, X and I). Since these bits have to do with processor interrupts and special modes, the function of these bits is reserved for chapter 10, when interrupts are discussed. This section will focus on the definition of the remaining five bits: H, N, Z, V and C. They indicate the status (or condition) of the processor at the end of the instruction and are referred to as the **status flags**.

Addition and subtraction instructions affect all of the status flags. Therefore, they serve as good illustrations of what conditions cause the various flags to be set. Figure 3.3 shows the layout of an 8-bit addition and an 8-bit subtraction. In the addition example, the B word (B7–B0) is added to the A word (A7–A0) to form the 8-bit result R (R7–R0). In the subtraction example, the B word (B7–B0) is subtracted from the A word (A7–A0) to form the 8-bit result R (R7–R0).

The **H bit** is the half carry flag. This flag is set when there is a carry from the lower nibble to the upper nibble during addition operations. If there is no carry from the lower nibble, this bit is cleared. Addition instructions are the only instructions that affect this flag. It is used only by the DAA instruction during the decimal adjustment after the addition of two BCD values. It is undefined after reset.

CCR	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
	S	X	H	I	N	Z	V	C
RESET	1	1	U	1	U	U	U	U

Figure 3.2 Condition Code Register Bits

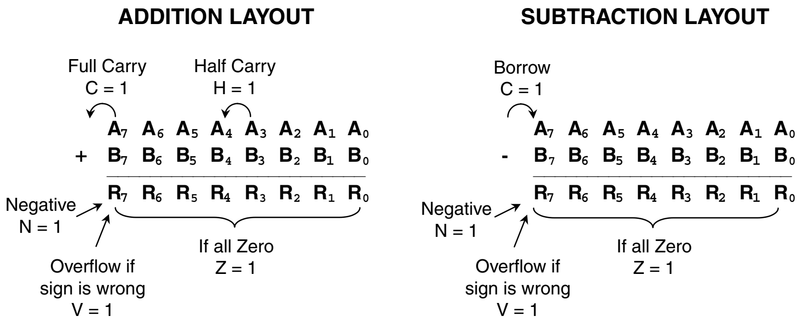


Figure 3.3 Meaning of the Condition Codes for Addition and Subtraction

Further explanation on BCD arithmetic and decimal adjustment is found later in this chapter.

The **N bit** is the negative flag. This bit indicates a negative condition. It is equal to the sign of the result and is equal to the sign bit (MSB) of the result. It is set if the data is negative and cleared if the data is positive. It is undefined after reset.

The **Z bit** is the zero flag. This bit indicates a zero condition. It is set if the result of the operation is zero. If the result is not zero, then this bit is cleared. It is undefined after reset.

The **V bit** is the overflow flag. This bit indicates a sign overflow has occurred. It is set when the operation caused a 2's complement overflow. When this bit is set, it indicates that the sign (N flag) of the result of the last operation is wrong. When it is cleared, the sign (N flag) of the result of the last operation is correct. It is undefined after reset.

The **C bit** is the full carry/borrow flag. When this bit is set, it indicates that the operation produced a full carry or that a full borrow was required to complete the operation. It is undefined after reset.

NOTE: Many instructions use these bits to indicate status. In some cases, the meaning of these bits varies from the definitions provided for addition and subtraction. These exceptions will be discussed as they arise.

Self-Test Questions 3.1

1. How many bits are in the CCR? How many are used for status?
2. What happens to the Z flag if the result of the last operation is equal to zero?
3. If the V flag is set, what is true regarding the sign of the last result?

3.2 Data Movement

Data movement instructions are concerned with the copying or moving of data. During the execution of a program, the opcodes, operands and addresses are located in memory. Before a program starts, the processor knows nothing about what it is going

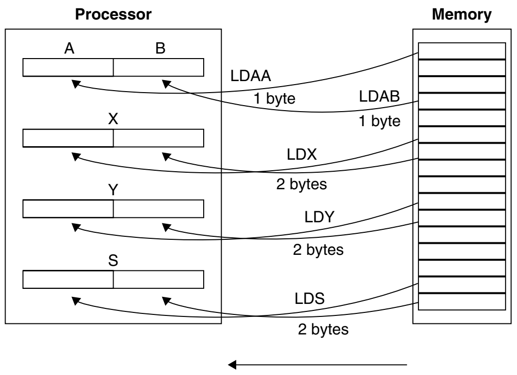


Figure 3.4 Direction of Data Movement During a Load Instruction

to do. The program must load the addresses and data into the processor required to complete the execution of the program. The movement of data can be accomplished three different ways:

1. from the processor registers to memory
2. from memory to the processor registers
3. from one processor register to another processor register.

NOTE: The HC11 does not support instructions that allow direct memory to memory transfers using a single instruction. An example of how to accomplish memory to memory transfers is provided in chapter 5.

Load Instructions

Load instructions are responsible for the movement of data from memory to the processor registers, as shown in Figure 3.4. Load instructions perform an input operation by reading one or more bytes of data into a processor register from memory. Each of the load instructions that copies two bytes of data follows the "high byte first" convention.

The family of load instructions is summarized in Figure 3.5. LDAA and LDAB load one byte into one of the 8-bit accumulators. LDD, LDS, LDX and LDY load two bytes into the double accumulator, stack pointer, index register X or index register Y respectively.

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
LDAA LDAB	Load Acc with contents of memory	(M) ⇒ A	X	X	X	X	X	-	-	-	-	-	-	↕	↕	0	-
LDD	Load D with contents of memory	(M) ⇒ A, (M+1) ⇒ B	X	X	X	X	X	-	-	-	-	-	-	↕	↕	0	-
LDS LDX LDY	Load 16-bit Register with contents of memory	(M):(M+1) ⇒ S (M):(M+1) ⇒ X (M):(M+1) ⇒ Y	X	X	X	X	X	-	-	-	-	-	-	↕	↕	0	-

Figure 3.5 Family of Load Instructions

Each load instruction operates in IMM, DIR, EXT, INDX and INDY addressing modes. All load instructions affect only three status flags in the CCR: N and Z reflect the actual condition of the data that was loaded. The V bit is cleared, indicating that the sign (N flag) is correct. Load instructions are complementary to store instructions.

Example 3.1

Problem: Describe what each of the following instructions accomplishes and then indicate the addressing mode used, effective address(es) of the operand, the operand and resulting status in the CCR, given the following:

```
0010  81  22  33  44  55  66  77  88     CCR  $D0
0120  2A  00  A6  00  00  FF  00  00
   Address    Machine Code   Source Code
a. 01D9       86 16          LDAA  #$16
b. 0112       D6 10          LDAB  $10
c. B6C3       FE 01 23       LDX   $0123
```

Solution:

a. $16 → A. Mode = IMM (the data is preceded by the # sign). Effective address = $01DA. It is the address immediately following the opcode in memory. Operand = $16. Data stored at the effective address. N = 0, Z = 0, V = 0 → CCR = $D0.

b. ($0010) → B. Mode = DIR (single byte in the operand field without # sign). Effective address = $0010. Effective address in direct mode is equal to $00dd. Operand = $81. Data stored at the effective address N = 1, Z = 0, V = 0 → CCR = $D8.

c. ($0123):($0124) → X. Mode = EXT (two bytes in the operand field without # sign). Effective addresses = $0123 and $0124. The first effective address in extended mode is equal to $hhll (M) and second is M+1. Operand = $0000. Data stored at the effective addresses. N = 0, Z = 1, V = 0 → CCR = $D4.

Example 3.2

Problem: Each load instruction occupies a fixed number of bytes in memory and requires a fixed number of machine cycles to execute. For each of the following instructions, use the Motorola documentation to determine the number of bytes it will occupy in memory, the name of each byte using the Motorola terminology, determine the opcode and how many machine cycles are required to complete the instruction.

(Hint: Refer to Appendix A of the Reference Manual and Table 3.2 of the Technical Data Manual.)

a. LDAB #$16

b. LDX $10

c. LDY $0123

d. LDAA $B600

Solution:

a. Bytes = 2. LDAB IMM will occupy two bytes, the first byte is the opcode and the second is the operand (ii). Opcode = $C6. Cycles = 2, 1 fetch and 1 execute.

b. Bytes = 2. LDX DIR will occupy two bytes, the first byte is the opcode and the second is the low-byte of the effective address (dd). Opcode = $DE. Cycles = 4, 1 fetch and 3 executes.

c. Bytes = 4. LDY EXT will occupy four bytes, the first byte is the prebyte, the second is the opcode and the last two are the first effective address of the operand (hhll). Opcode = $18 FE ($18 is a prebyte and $FE is the actual opcode). Cycles = 6, 2 fetches and 4 executes.

d. Bytes = 3. LDAA EXT will occupy three bytes, the first byte is the opcode and the last two are the effective address of the operand (hhll). Opcode = $B6. Cycles = 4, 1 fetch and 3 executes.

Store Instructions

Store instructions are responsible for the movement of data from the processor registers to memory as shown in Figure 3.6. Store instructions perform an output operation by writing one or more bytes of data into memory from a processor register. Each of the store instructions that stores two bytes of data follows the "high byte first" convention.

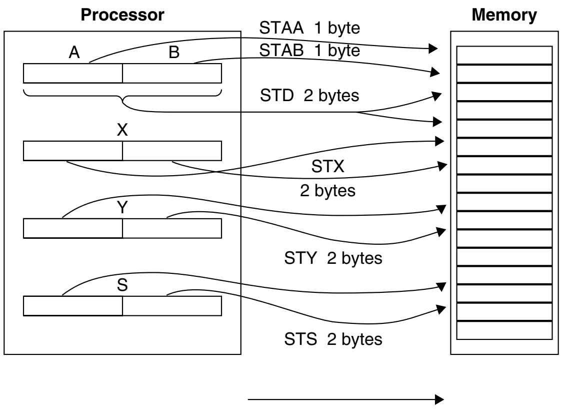


Figure 3.6 Direction of Data Movement During a Store Instruction

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
STAA STAB	Store Acc to contents of memory	(A) ⇒ M (B) ⇒ M	-	X	X	X	X	-	-	-	-	-	-	↕	↕	0	-
STD	Store D to contents of memory	(A) ⇒ M, (B) ⇒ M+1	-	X	X	X	X	-	-	-	-	-	-	↕	↕	0	-
STS STX STY	Store 16-bit Register to contents of memory	(S) ⇒ M:M+1 (X) ⇒ M:M+1 (Y) ⇒ M:M+1	-	X	X	X	X	-	-	-	-	-	-	↕	↕	0	-

Figure 3.7 Family of Store Instructions

The family of store instructions is summarized in Figure 3.7. STAA and STAB store a byte from one of the 8-bit accumulators into memory. STD, STS, STX and STY store two bytes from the double accumulator, stack pointer, index register X or index register Y into memory respectively. Each of the store instructions operates in the DIR, EXT, INDX and INDY addressing modes. All store instructions affect only three status flags in the CCR: N and Z reflect the actual condition of the data that was loaded. The V bit is cleared, indicating that the sign (N flag) is correct. Store instructions are complementary to load instructions.

Example 3.3

Problem: Describe what each of the following instructions accomplishes and then indicate the addressing mode used, effective address(es) of the operand, the operand and resulting status in the CCR, given the following:

```
      A    $80      X    $B600
      B    $FF      Y    $1000
      CCR  $E1      S    $0041
   Address  Machine Code  Source Code
a. 0002     B7 00 16      STAA  $0016
b. 018F     D7 10         STAB  $10
c. E2B3     DD 08         STS   $08
```

Solution:

a. Mode = EXT. The two bytes in the operand field without # sign. Effective address = $0016. The first effective address in extended mode is equal to $hhll. Operand = $80. Data from AccA. N = 1, Z = 0, V = 0. Thus CCR = $E9.

b. Mode = DIR. The single byte in the operand field without # sign. Effective address = $0010. Effective address in direct mode is equal to $00dd. Operand = $FF. Data from AccB. N = 1, Z = 0, V = 0. Thus CCR = $E9.

c. Mode = DIR. The single byte in the operand field without # sign. Effective addresses = $0008 and $0009. First effective address in direct mode is equal to $00dd (M), second is M+1. Operand = $0041. Data from SP. N = 0, Z = 0, V = 0. Thus CCR = $E1.

Example 3.4

Problem: Each store instruction occupies a fixed number of bytes in memory and requires a fixed number of machine cycles to execute. For each of the following instructions, use the Motorola documentation to determine the number of bytes it will occupy in memory, the name of each byte using the Motorola terminology, determine the opcode and how many machine cycles are required to complete the instruction.

(Hint: Refer to Appendix A of the Reference Manual and Table 3.2 of the Technical Data Manual.)

```
a. STAB $16
b. STX  $10
c. STY  $0123
d. STAA $B600
```

Solution:

a. Bytes = 2. STAB DIR will occupy two bytes; the first byte is the opcode and the second is the low byte of the effective address (dd). Opcode = $D7. Cycles = 3, 1 fetch and 2 executes.

b. Bytes = 2. STX DIR will occupy two bytes; the first byte is the opcode and the second is the low byte of the first effective address (dd). Opcode = $DF. Cycles = 4, 1 fetch and 3 executes.

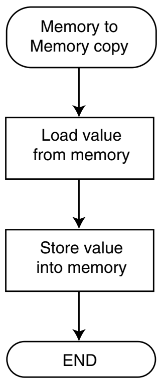
c. Bytes = 4. STY EXT will occupy four bytes; the first byte is the prebyte, the second is the opcode and the last two are the first effective address of the operand (hhll). Opcode = $18 FF FE ($18 is a prebyte and $FF is the actual opcode). Cycles = 6, 2 fetches and 4 executes.

d. Bytes = 3. STAA EXT will occupy three bytes; the first byte is the opcode and the last two are the effective address of the operand (hhll). Opcode = $B7. Cycles = 4, 1 fetch and 3 executes.

Example 3.5

Problem: Draw flowchart and write the program that will copy a single byte from $0100 to $0180. Assemble the code, starting at location $0000.

Solution: In order to copy a byte from one memory location to another, the byte must be loaded into a processor register and then stored to the new location. This



Example 3.5 Flowchart

program reads the byte stored at location $0100 and loads it into AccA. Then it takes the value from AccA and writes it out to memory location $0180.

```
Address   Machine Code   Source Code    Comments
0000      B6 01 00       LDAA  $0100    ;($0100) → A
0003      B7 01 80       STAA  $0180    ;(A) → $0180
```

Clear Instructions

Clear instructions are responsible for clearing the contents of a processor register or a memory location. The clear instructions responsible for clearing a processor register are very similar to load instructions. Functionally, $00 is loaded into the register. The clear instruction responsible for clearing a memory location is very similar to a store instruction. Functionally, $00 is stored in the memory location. Clear instructions provide a distinct advantage over load and store instructions. Each clear instruction occupies fewer bytes of memory than the equivalent combination of load and store instructions.

The clear instructions are summarized in Figure 3.8. CLRA and CLRB work like load instructions in that $00 is loaded into AccA or AccB. The mnemonic form, CLR, works like a store instruction in that $00 is stored into a memory location. The CLRA and CLRB operate in the inherent mode, and CLR operates in the EXT, INDX and INDY

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
CLRA CLRB	Clear Acc	$00 ⇒ A $00 ⇒ B	-	-	-	-	-	X	-	-	-	-	-	0	1	0	0
CLR	Clear contents of memory	$00 ⇒ M	-	-	X	X	X	-	-	-	-	-	-	0	1	0	0

Figure 3.8 Clear Instructions

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
TAB TBA	Transfer Acc to the other Acc	(A) ⇒ B (B) ⇒ A	-	-	-	-	-	X	-	-	-	-	-	↕	↕	0	-
TAP	Transfer A to CCR	(A) ⇒ CCR	-	-	-	-	-	X	-	↕	↓	↕	↕	↕	↕	↕	↕
TPA	Transfer CCR to A	(CCR) ⇒ A	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
TSX TSY	Transfer Stack Pointer to Index Reg	(S) + 1 ⇒ X (S) + 1 ⇒ Y	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
TXS TYS	Transfer Index Reg to Stack Pointer	(X) - 1 ⇒ S (Y) - 1 ⇒ S	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-

Figure 3.9 Transfer Instructions

addressing modes. No support is provided for the DIR addressing mode. Four status flags in the CCR are affected by clear instructions: N = 0 since zero is a positive number, Z = 1 because the result of the last operation was zero, V = 0 since the sign of the number is correct (positive) and C = 0 because there was no carry.

Transfer Instructions

The transfer instructions are responsible for the movement of data from one processor register to another. They copy the data from the source register and to the destination register. The operation of the transfer instructions is summarized in Figure 3.9.

TAB copies data from AccA to AccB, and TBA copies data from AccB to AccA. These instructions operate in the inherent addressing mode. These instructions affect three status flags in the CCR: N and Z reflect the actual condition of the data after the transfer operation. The V bit is cleared, indicating that the sign (N flag) is correct.

TAP copies data from AccA to the CCR. It causes all eight bits of the CCR to be updated with the data from AccA. However, the X bit can only be cleared by this instruction. Attempts to set the X bit are ignored. TPA copies data from the CCR to A. TPA has no affect on the CCR. The TPA and TAP instructions operate in the inherent addressing mode.

Further explanation of the X bit is provided in chapter 10.

The TSX and TSY instructions take the contents of the stack pointer register, add one to it and load this value into one of the two index registers. The TXS and TYS instructions take the contents of the designated index register, subtract one and load this value into the stack pointer register. These instructions operate in the inherent addressing mode and have no affect on the CCR.

Exchange Instructions

Exchange instructions are responsible for swapping the contents of the double accumulator with the contents of one of the index registers. In essence, they act like a double transfer instruction because data is moved to and from each register. The two exchange instructions are summarized in Figure 3.10. XGDX swaps the

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
XGDX XGDY	Exchange D with Index Reg	D ↔ X D ↔ Y	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-

Note: D ↔ X implies that (D) → X while the original (X) → D.

Figure 3.10 Exchange Instructions

contents of the D and X registers. XGDY swaps the contents of the D and Y registers. These instructions operate in the inherent addressing mode and have no affect on the CCR.

Self-Test Questions 3.2

1. Do load instructions perform read or write operations?
2. Do store instructions perform read or write operations?
3. Which instruction acts like a store instruction but only stores $00?
4. Why can all transfer instructions operate in the inherent addressing mode?
5. How many data bits are copied from each register during an exchange instruction?

3.3 Addition

Addition instructions are a subset of the many arithmetic instructions that use the internal arithmetic logic unit. Each instruction performs a mathematical addition operation on the contents of an accumulator or a memory location. The result of each of these operations is written back into the accumulator or memory location designated by the instruction.

Add Instructions

Add instructions are responsible for adding data to the contents of one of the accumulators. Both 8-bit and 16-bit add instructions are supported. In each case, the result of the addition operation is kept in the accumulator. The add instructions are summarized in Figure 3.11.

ADDA and ADDB are used for 8-bit addition. An 8-bit operand from memory is added to the contents of AccA or AccB. ADDD is used to accomplish 16-bit addition operations. A 16-bit operand is added to the contents of AccD, the double accumulator. These instructions operate in the IMM, DIR, EXT, INDX and INDY addressing modes. The ADDA and ADDB instructions affect all five status flags in the CCR: H, N, Z, V and C reflect the actual condition of the data after the add operation. The ADDD instruction affects four status flags in the CCR: N, Z, V and C reflect the actual condition of the data after the add operation. H is not defined for 16-bit operations.

ABA, ABX and ABY allow the contents of the B accumulator to be added to the A, X or Y registers respectively. These instructions operate in the inherent addressing mode.

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
ADDA ADDB	Add contents of memory to Acc	(A) + (M) ⇒ A (B) + (M) ⇒ B	X	X	X	X	X	-	-	-	-	↕	-	↕	↕	↕	↕
ADDD	Add contents of memory to D	(D)+(M):(M+1) ⇒ D	X	X	X	X	X	-	-	-	-	-	-	↕	↕	↕	↕
ABA	Add B to A	(A) + (B) ⇒ A	-	-	-	-	-	X	-	-	-	↕	-	↕	↕	↕	↕
ABX ABY	Add B to Index Reg	(X) + $00:(B) ⇒ X (Y) + $00:(B) ⇒ Y	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-

Figure 3.11 Addition Instructions

The ABA instruction affects all five status flags in the CCR: H, N, Z, V and C reflect the actual condition of the data after the add operation. The ABX and ABY instructions have no effect on the contents of the CCR.

Example 3.6

Problem: Describe what each of the following instructions accomplishes and then indicate the addressing mode used, effective address(s) of the operand, the result and resulting status in the CCR, given the following:

```
     0028  81 22 33 4A 55 6B C7 88
     0160  FF 00 D6 00 10 1F 10 0F

     A   $80 X $B600
     B   $2F Y $0100
     CCR $D3 S $0041
   Address  Machine Code  Source Code
a.   B700      8B 65         ADDA  #$65
b.   000E      FB 01 62      ADDB  $0162
c.   012C      D3 29         ADDD  $29
d.   F4B3      3A            ABX
```

Solution:

a. Mode = IMM. The byte in the operand field has # sign. Effective address = $B701. It is the address immediately following the opcode in memory. $80 + $65 = $E5 → A. H = 0, N = 1, Z = 0, V = 0, C = 0 → CCR = $D8.

b. Mode = EXT. The two bytes in the operand field without # sign. Effective address = $0162. Effective address in extended mode is equal to $hhll. (B) + ($0162) = $2F + $D6 = $05 → B. H = 1, N = 0, Z = 0, V = 0, C = 1 → CCR = $E1.

c. Mode = DIR. The single byte in the operand field without # sign. Effective addresses = $0029 and $002A. First effective address in direct mode is equal to $00dd (M) and second is M+1. (D) + ($0029):($002A) = $802F + $2233 = $A262 → D. N = 1, Z = 0, V = 0, C = 0 → CCR = $D8.

d. Mode = INH. The operand field is empty. Effective addresses = N/A. Memory is not accessed. (X) + $00:(B) = $B600 + $002F = $B62F → X. Does not affect status flags, thus CCR = $D3.

Example 3.7

Problem: Each add instruction occupies a fixed number of bytes in memory and requires a fixed number of machine cycles to execute. For each of the following instructions, use the Motorola documentation to determine the number of bytes it will occupy in memory, name the bytes using the Motorola terminology, and determine the opcode and how many machine cycles are required to complete the instruction.

```
a. ADDB #$16
b. ABA
c. ABY
d. ADDA $B6
```

Solution:

a. Bytes = 2. ADDB IMM will occupy two bytes; the first byte is the opcode and the second is the immediate data (ii). Opcode = $CB. Cycles = 2, 1 fetch and 1 execute.

b. Bytes = 1. ABA INH will occupy one byte, the opcode. Opcode = $1B. Cycles = 2, 1 fetch and 1 execute.

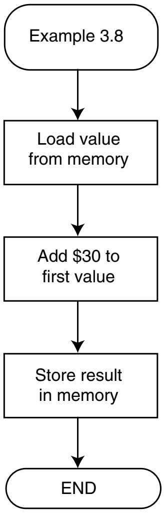
c. Bytes = 2. ABY INH will occupy two bytes; the first byte is the prebyte and the second is the opcode. Opcode = $18 3A. Cycles = 4, 2 fetches and 2 executes.

d. Bytes = 2. ADDA DIR will occupy two bytes; the first byte is the opcode and the second byte is the low byte of the effective address of the operand (dd). Opcode = $9B. Cycles = 2, 1 fetch and 1 execute.

Example 3.8

Problem: Draw flowchart and write the program that will load a byte from memory $0000, add $30 to it, then store the result in locations $0121. Assemble the code, starting at location $0100, and use AccB for the addition. Assume the value stored at $0000 is $D0.

Solution: The program starts by loading AccB, using direct mode with the value stored at $0000. Then, using immediate mode, it adds $30 to the first value and



Example 3.8 Flowchart

loads the result back into AccB. Finally, using extended mode, it stores the result in location $0121.

```
Address   Machine Code   Source Code       Comments
0000      D6 00          LDAB   54M$00     ;($0000) → B
0002      CB 30          ADDB   #F54M$30   ;(B) + $30 → B
0004      F7 01 21       STAB   $0121      ;(B) → $0121
```

BCD Addition

The HC11 supports addition of binary coded decimal data. It provides the DAA instruction to perform decimal adjustments of the output of the hex adder. DAA operates only on the data in AccA; therefore, only AccA can be used for BCD addition. The DAA instruction only performs the proper BCD adjustments following the ADDA, ADCA or ABA addition operations because these instructions affect the H flag in the CCR and operate on data in AccA. The two operands (or values being added) must be valid BCD values before the addition otherwise the DAA instruction will not perform the proper adjustments. Because the ALU will add these values as if they are Hex values, the DAA is necessary to adjust the result back to a valid BCD state. Figure 3.12 summarizes what will happen when the DAA instruction is executed. The DAA instruction affects four status flags in the CCR: N and Z reflect the actual condition of the data after the add operation. Sign-overflow is undefined for BCD, yet the V flag will reflect the standard overflow condition as if the data were in hex. The C flag will reflect the state according to the data shown in Figure 3.12.

State of C Bit Before DAA (Column 1)	Upper Half-Byte of Acca (Bits 7–4) (Column 2)	Initial Half-Carry H Bit from CCR (Column 3)	Lower Half-Byte of Acca (Bits 3–0) (Column 4)	Number Added to Acca by DAA (Column 5)	State of C Bit After DAA (Column 6)
0	0-9	0	0-9	00	0
0	0-8	0	A-F	06	0
0	0-9	1	0-3	06	0
0	A-F	0	0-9	60	1
0	9-F	0	A-F	66	1
0	A-F	1	0-3	66	1
1	0-2	0	0-9	60	1
1	0-2	0	A-F	66	1
1	0-3	1	0-3	66	1

NOTE

Columns (1) through (4) of the table represent all possible cases that can result from any of the operations ABA, ADDx, or ADCx, with initial carry either set or clear, applied to binary-coded operands. The table shows hexadecimal values.

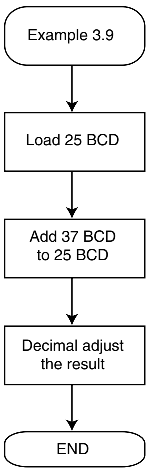
Figure 3.12 BCD Adjustment after Addition (*adapted with permission from Motorola)*

Example 3.9

Problem: Draw a flowchart and write the program that will load 25 BCD, add 37 BCD to it, then decimal adjust the result. Assemble the code, starting at location $0100.

Solution: The program starts by using immediate mode to load AccA with 25 BCD. Then, using immediate mode, it adds 37 BCD to the first value and loads the result back into AccA. Since the ALU performs a hex addition, 25 and 37 are viewed as hex values ($25 and $37), resulting in the sum $5C. Finally, using DAA, the program adjusts the result back to a valid BCD state by adding the adjustment value $06, which is determined from the table in Figure 3.12. The second line of the table applies because C = 0 (C was cleared by the add instruction), the upper nibble = $5, H = 0 (H was cleared by the add instruction) and the lower nibble = $C. The result is the proper BCD value of 62.

```
Address   Machine Code   Source Code        Comments
0000      86 25          LDAA   #F54M$25    ;$25 → A
0002      8B 37          ADDA   #F54M$37    ;$25 + $37 = $5C → A
0004      19             DAA                ;$5C + $06 = $62 → A
```



Example 3.9 Flowchart

Increment Instructions

Increment instructions are a special class of addition instructions. They add one to the operand, then overwrite the previous operand with this result. The increment instructions are summarized in Figure 3.13.

The INC instruction operates on data stored in memory. It operates in the EXT, INDX and INDY addressing modes. No support is provided for the DIR addressing mode. The INCx and INx forms operate on data stored in one of the processor registers and therefore use the inherent mode. The A, B, S, X and Y registers are incremented by the INCA, INCB, INS, INX and INY instructions respectively. The INC, INCA and INCB instructions affect three status flags in the CCR: N, Z and V reflect the actual condition of the data after the increment operation. The INX and INY instructions affect only one flag: Z is updated to reflect the actual condition of the data after the increment. The INS instruction has no effect on the CCR.

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
INCA INCB	Increment Acc	(A) + 1 ⇒ A (B) + 1 ⇒ B	-	-	-	-	-	X	-	-	-	-	-	↕	↕	↕	-
INS	Increment the stack pointer	(S) + 1 ⇒ S	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
INX INY	Increment Index Register	(X) + 1 ⇒ X (Y) + 1 ⇒ Y	-	-	-	-	-	X	-	-	-	-	-	-	↕	-	-
INC	Increment contents of memory	(M) + 1 ⇒ M	-	-	X	X	X	-	-	-	-	-	-	↕	↕	↕	-

Figure 3.13 Increment Instructions

No sample code is provided at this point for increment and decrement instructions. However, several examples in chapters 4 and 5 use increment instructions.

Self-Test Questions 3.3

1. Which flag bit is only affected by addition instructions?
2. What is the purpose of the DAA instruction?
3. Can the function of the increment instruction be accomplished with addition and subtraction instructions?

3.4 Subtraction

Subtract instructions are a subset of the many arithmetic instructions that use the internal arithmetic logic unit. Each instruction performs a mathematical subtraction operation on the contents of an accumulator or a memory location. The result of each of these operations is written back into the accumulator or memory location designated by the instruction.

Subtract Instructions

Subtract instructions are responsible for subtracting data from the contents of one of the accumulators. Both 8-bit and 16-bit subtract instructions are supported. In each case, the result of the subtraction operation is kept in the accumulator. The subtract instructions are summarized in Figure 3.14.

SUBA and SUBB are used for 8-bit subtraction. An 8-bit operand from memory is subtracted from the contents of AccA or AccB. SUBD is used to perform 16-bit subtraction operations. A 16-bit operand is subtracted from the contents of AccD, the double accumulator. These instructions operate in the IMM, DIR, EXT, INDX and INDY addressing modes. The SUBA, SUBB and SUBD instructions affect four status flags in the CCR: N, Z, V and C reflect the actual condition of the data after the subtract operation. H is not defined for subtraction.

SBA allows the contents of AccB to be subtracted from AccA. This instruction operates in the inherent addressing mode. The SBA instruction affects four status flags in the CCR: N, Z, V and C reflect the actual condition of the data after the subtract operation. H is not defined for subtraction.

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
SUBA SUBB	Subtract contents of memory from Acc	(A) - (M) ⇒ A (B) - (M) ⇒ B	X	X	X	X	X	-	-	-	-	-	-	↕	↕	↕	↕
SUBD	Subtract contents of memory from D	(D)-(M):(M+1) ⇒ D	X	X	X	X	X	-	-	-	-	-	-	↕	↕	↕	↕
SBA	Subtract B from A	(A) – (B) ⇒ A	-	-	-	-	-	X	-	-	-	-	-	↕	↕	↕	↕

Figure 3.14 Subtraction Instructions

Example 3.10

Problem: Describe what each of the following instructions accomplishes and then indicate the addressing mode used, effective address(es) of the operand, the result and resulting status in the CCR, given the following:

```
        0028   81 22 33 4A 55 6B C7 88
        0160   FF 00 D6 00 10 1F 10 0F
        A    $80       X    $B600
        B    $2F       Y    $0100
        CCR  $D3       S    $0041
     Address   Machine Code   Source Code
a.   B700      80 65          SUBA  #$65
b.   000E      F0 01 62       SUBB  $0162
c.   012C      93 29          SUBD  $29
```

Solution:

a. Mode = IMM. The byte in the operand field has # sign. Effective address = $B701. It is the address immediately following the opcode in memory. $80 – $65 = $1B → A. N = 0, Z = 0, V = 1, C = 0, thus CCR = $D2.

b. Mode = EXT. The two bytes in the operand field without # sign. Effective address = $0162. Effective address in extended mode is equal to $hhll. (B) – ($0162) = $2F – $D6 = $59 → B. N = 0, Z = 0, V = 0, C = 1 → CCR = $D1.

c. Mode = DIR. The single byte in the operand field without # sign. Effective addresses = $0029 and $002A. First effective address in direct mode is equal to $00dd (M) and second is M+1. (D) – ($0029):($002A) =$802F – $2233 = $5DFC → D. N = 0, Z = 0, V = 1, C = 0 → CCR = $D8.

Example 3.11

Problem: Each subtract instruction occupies a fixed number of bytes in memory and requires a fixed number of machine cycles to execute. For each of the following instructions, use the Motorola documentation to determine the number of bytes it will occupy in memory, name the bytes using the Motorola terminology and determine the opcode and how many machine cycles are required to complete the instruction.

```
a. SUBB #$16
b. SBA
c. SUBA $B6
```

Solution:

a. Bytes = 2. SUBB IMM will occupy two bytes; the first byte is the opcode and the second is the immediate data (ii). Opcode = $C0. Cycles = 2, 1 fetch and 1 execute.

b. Bytes = 1. SBA INH will occupy one byte, the opcode. Opcode = $10. Cycles = 2, 1 fetch and 1 execute.

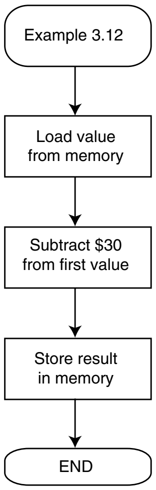
c. Bytes = 2. SUBA DIR will occupy two bytes; the first byte is the opcode and the second byte is the low byte of the effective address of the operand (dd). Opcode = $90. Cycles = 2, 1 fetch and 1 execute.

Example 3.12

Problem: Draw a flowchart and write the program that will load a byte from memory $0000, subtract $30 from it, then store the result in locations $0121. Assemble the code, starting at location $0100 and use AccB for the addition. Assume the value stored at $0000 is $21.

Solution: The program starts by loading AccB using direct mode with the value stored at $0000. Then, using immediate mode, it subtracts $30 from the first value and loads the result back into AccB. Finally, using extended mode, it stores the result in location $0121.

```
Address  Machine Code  Source Code    Comments
0000     D6 00         LDAB  $00      ;($0000) → B
0002     C0 30         SUBB  $30      ;(B) - $30 → B
0004     F7 01 21      STAB  $0121    ;(B) → $0121
```



Example 3.12 Flowchart

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
NEGA NEGB	2's Complement of Acc	$00 – (A) ⇒ A $00 – (B) ⇒ B	-	-	-	-	-	X	-	-	-	-	-	↕	↕	↕	↕
NEG	2's Complement of contents of memory	$00-(M) ⇒ M	-	-	X	X	X	-	-	-	-	-	-	↕	↕	↕	↕

Figure 3.15 Negate Instructions

Negate Instructions

The negate instructions change the sign of a value in AccA, in AccB or in a memory location. It accomplishes this sign change by using the 2's complement negation operation on the 8-bit operand. Specifically, these instructions subtract the operand from zero, then overwrite the previous operand with the result, as shown in Figure 3.15.

NEGA and NEGB use the contents of AccA or AccB as the operand. These two mnemonic forms operate in the inherent addressing mode. NEG uses the contents of a memory location as the operand; however, it is available only in the EXT, INDX and INDY addressing modes. No support is provided for the DIR addressing mode. Like the subtract instructions, the NEGA, NEGB and NEG instructions affect four status flags in the CCR: N, Z, V and C reflect the actual condition of the data after the negate operation.

Because of the similarity the negate instructions have to the subtract instructions, no additional sample code is provided.

Decrement Instructions

Decrement instructions are a special class of subtraction instructions. They subtract one from the operand, then overwrite the previous operand with this result. The decrement instructions are summarized in Figure 3.16.

The DEC instruction operates on data stored in memory; however, it operates only in the EXT, INDX and INDY addressing modes. No support is provided for the DIR addressing mode. The DECx and DEx forms operate on data stored in one of the processor registers and therefore use the inherent mode. The A, B, S, X and Y registers

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
DECA DECB	Decrement Acc	(A) - 1 ⇒ A (B) - 1 ⇒ B	-	-	-	-	-	X	-	-	-	-	-	↕	↕	↕	-
DES	Decrement Stack Pointer	(S) - 1 ⇒ S	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
DEX DEY	Decrement Index Reg	(X) - 1 ⇒ X (Y) - 1 ⇒ Y	-	-	-	-	-	X	-	-	-	-	-	-	↕	-	-
DEC	Decrement contents of memory	(M) - 1 ⇒ M	-	-	X	X	X	-	-	-	-	-	-	↕	↕	↕	-

Figure 3.16 Decrement Instructions

are decremented by the DECA, DECB, DES, DEX and DEY instructions respectively. The DEC, DECA and DECB instructions affect three status flags in the CCR: N, Z and V reflect the actual condition of the data after the decrement operation. The DEX and DEY instructions affect only one flag: Z is updated to reflect the actual condition of the data after the decrement. The DES instruction has no effect on the CCR.

No sample code is provided at this point for increment and decrement instructions. However, several examples in chapters 4 and 5 use decrement instructions.

Self-Test Questions 3.4

1. Which flag bit is not affected by subtract instructions?
2. Why are the negate instructions grouped with subtract instructions?
3. Can the function of the decrement instructions be accomplished with subtraction instructions?

3.5 Logic

Logic instructions, a subset of the instruction set, require the use of the internal arithmetic logic unit. Each instruction performs a logical operation on the contents of an accumulator or on the operand stored in memory. The result of each of these operations is written back into the accumulator or the memory location designated by the instruction. The logical AND, OR, XOR and NOT operations are supported by the HC11. Each of these instructions performs the logical operation on a group of eight bits. In essence, eight 2-bit pairs are created and operated on independently to produce the result. The logic instructions are summarized in Figure 3.17.

The ANDx instructions perform a logical AND operation on an 8-bit operand and the data contained in one of the 8-bit accumulators, as shown in Figure 3.18. ANDA and ANDB overwrite the contents of the A or B accumulator with the result of the AND operation. ORAx and EORx are functionally identical to the ANDx instructions, except they perform logical OR and XOR operations respectively. The ANDx, ORAx, and EORx instructions are available in the IMM, DIR, EXT, INDX and INDY addressing

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
ANDA ANDB	AND Acc with contents of memory	(A) • (M) ⇒ A (B) • (M) ⇒ B	X	X	X	X	X	-	-	-	-	-	-	↕	↕	0	-
ORAA ORAB	OR Acc with contents of memory	(A) + (M) ⇒ A (B) + (M) ⇒ B	X	X	X	X	X	-	-	-	-	-	-	↕	↕	0	-
EORA EORB	XOR Acc with contents of memory	(A) ⊕ (M) ⇒ A (B) ⊕ (M) ⇒ B	X	X	X	X	X	-	-	-	-	-	-	↕	↕	0	-
COMA COMB	NOT Acc 1's complement of Acc	!(A) ⇒ A !(B) ⇒ B	X	X	X	X	X	-	-	-	-	-	-	↕	↕	0	1
COM	1's Complement of contents of memory	!(M) ⇒ M	-	-	X	X	X	-	-	-	-	-	-	↕	↕	0	1

Figure 3.17 Logic Instructions

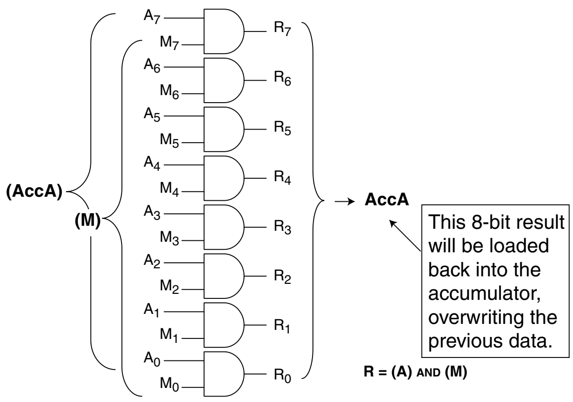

(AccA)
(M)
AccA
This 8-bit result will be loaded back into the accumulator, overwriting the previous data.
R = (A) AND (M)

Figure 3.18 Logical AND Operation on the HC11 (ANDx). The 8-bit contents of AccA is ANDed as bit pairs with the 8-bit contents of a memory location. Eight 2-input AND gates are used for the operation. The result is loaded back into AccA. OR (ORAx) and XOR (EORx) perform in the same way, except OR gates and XOR gates are used, respectively.

modes. They affect three status flags in the CCR: N and Z reflect the actual condition of the data after the logic operation, and V = 0 indicates that the sign of the result is correct.

The COMx instructions perform a logical NOT operation on an 8-bit operand. COMA and COMB overwrite the contents of AccA or AccB with the result of the NOT operation. The COM form overwrites the contents of a memory location with the result of the NOT operation. This operation is also referred to as the 1's complement, thus the use of the mnemonic "COM." The COMA and COMB instructions are available in the INH addressing mode only. The COM instruction is available in the EXT, INDX

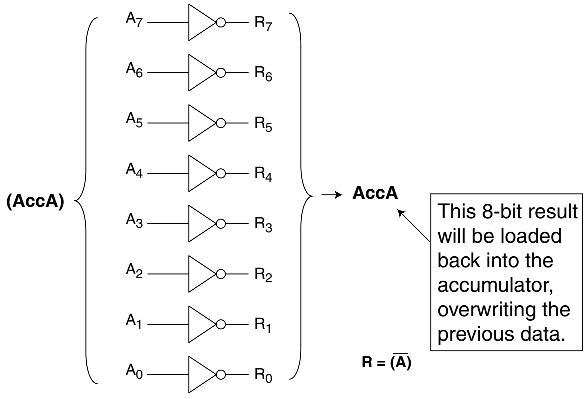

(AccA)
AccA
This 8-bit result will be loaded back into the accumulator, overwriting the previous data.
R = $(\overline{A})$

Figure 3.19 Logical NOT Operation on the HCII (COMx). The 8-bit contents of AccA are inverted. Eight NOT gates are used for the operation. The result is loaded back into AccA. COM performs in the same way, except the contents of a memory location are the input and the result overwrites the data in the memory location.

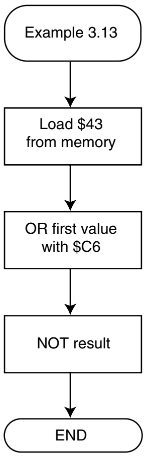
and INDY addressing modes, as shown in Figure 3.19. No support is provided for the DIR addressing mode. These instructions affect four status flags in the CCR: N and Z reflect the actual condition of the data after the logic operation, V = 0 indicates that the sign of the result is correct and C is set.

Example 3.13

Problem: Draw a flowchart and write the program that will perform a logical NOR operation on $43 and $C6. Load AccA with the first value and NOR it with another value. Use immediate mode as much as possible.

Solution: The program starts by loading $43 into AccA using immediate mode. Then, using immediate mode, it ORs $43 with $C6. Finally, the result is NOTed to complete the NOR operation.

```
Address   Machine Code   Source Code       Comments
0000      86 43          LDAA   #F54M$43   ;$43 → A
0002      8A C6          ORAA   #F54M$C6   ;$43 OR $C6 = $C7 → A
0004      43             COMA              ;!(A) = $38 →D A
```



Example 3.13 Flowchart

Self-Test Questions 3.5

1. What logical functions are supported by the HC11?
2. Which of the logical functions can operate on data in the 8-bit accumulators? On data in memory?

3.6 Shifting and Rotating

Several types of data shifting and rotating instructions are supported on the HC11. **Shifting** of data is based on the principles of serial shift registers. Data is moved one bit to the right (toward the LSB) or one bit to the left (toward the MSB). Logical shifting, as well as arithmetic shifting, is supported. **Logical shifting** causes a "0" to be shifted into one end of the data word, pushing all of the data bits over one position in the data word, as shown in Figure 3.20.

Arithmetic shifting is similar to logical shifting with one exception. When shifting right (toward the LSB), the sign bit is maintained. Instead of shifting a "0" into the MSB of the data word, a copy of the sign bit (MSB) is shifted and the original sign bit remains in the MSB position of the data word, as shown in Figure 3.20c.

Data rotating is a special form of data shifting. When **rotating** data, the output of the shifted data is fed back to the input of the shift register and is shifted back into the data word. Both right and left rotate are supported by the HC11, as shown in Figure 3.20.

All shift and rotate instructions on the HC11 utilize the C flag of the CCR to catch the output data bit as it is shifted out of the data word. Whatever bit is being shifted out is moved into the C bit. This permits a simple bit-level test to immediately follow the operation.

Logical Shift Instructions

The logical shift left (LSL) instructions are responsible for shifting a data word one bit to the left. The MSB is shifted into the C flag of the CCR, and the LSB is filled with zero. In Figure 3.21a, there is an 8-bit data word $A7 that will be shifted left. Before the shift is performed, the C flag in the CCR contains zero. When the shift left operation

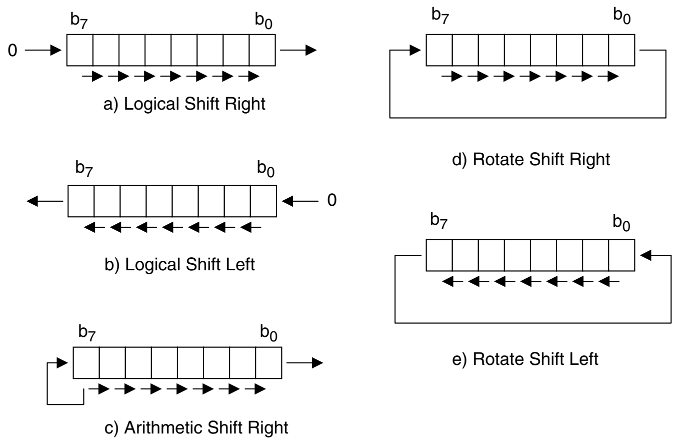


Figure 3.20 Data Shifting and Rotating

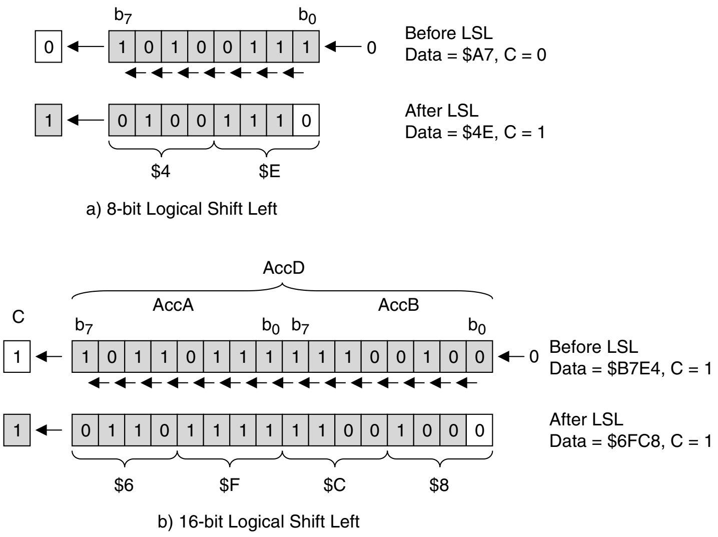


Figure 3.21 Logical Shift Left Examples

is performed, a "0" is shifted into the LSB from the right. The original 8-bit word is moved one bit position to the left, and MSB is moved into the C flag of the CCR. Thus the resulting data word is $4E and C flag = 1. Figure 3.21b shows how the same process applies to a 16-bit data word stored in AccD.

The logical shift right (LSR) instructions are responsible for shifting a data word one bit to the right. The LSB is shifted into the C flag of the CCR and the MSB is filled with zero. In Figure 3.22a, there is an 8-bit data word $A7 that will be shifted right.

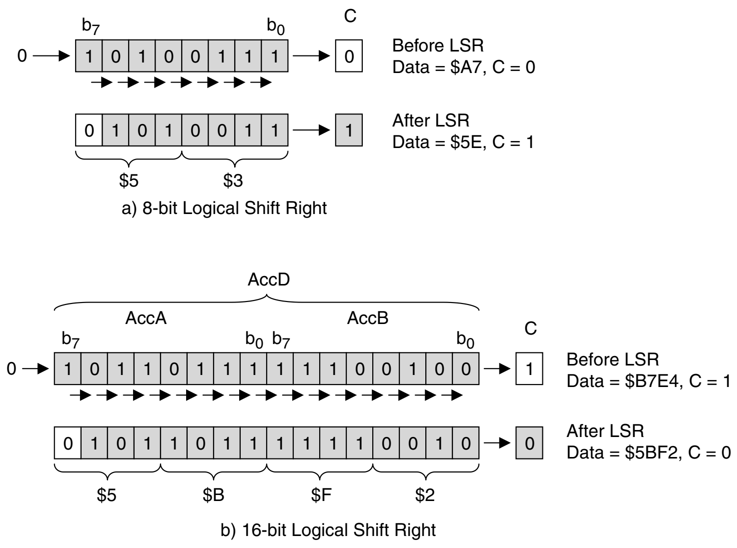


Figure 3.22 Logical Shift Right Examples

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
LSLA LSLB LSLD	Logical shift left Acc	*See Figure 3.21*	-	-	-	-	-	X	-	-	-	-	-	↕	↕	↕	↕
LSL	Logical shift left contents of memory	*See Figure 3.21*	-	-	X	X	X	-	-	-	-	-	-	↕	↕	↕	↕
LSRA LSRB LSRD	Logical shift right Acc	*See Figure 3.22*	-	-	-	-	-	X	-	-	-	-	-	0	↕	↕	↕
LSR	Logical shift right contents of memory	*See Figure 3.22*	-	-	X	X	X	-	-	-	-	-	-	0	↕	↕	↕

Figure 3.23 Logical Shift Instructions

Before the shift is performed, the C flag in the CCR contains zero. When the shift right operation is performed, a "0" is shifted into the MSB from the left. The original 8-bit word is moved one bit position to the right and LSB is moved into the C flag of the CCR. Thus the resulting data word is $53 and C = 1. Figure 3.22b shows how the same process applies to a 16-bit data word stored in AccD.

There are four versions of the logical shift left instruction: LSL, LSLA, LSLB, and LSLD, as shown in Figure 3.23. The LSL instruction operates directly on an 8-bit data word stored at a memory location. It operates in the EXT, INDX and INDY addressing modes. No support is provided for the DIR addressing mode. The LSLA and LSLB instructions operate on the 8-bit data contained in AccA and AccB respectively. The LSLD instruction operates on the 16-bit data word contained in AccD. LSLA, LSLB and LSLD operate in the inherent addressing mode. The logical shift left instructions affect four status flags in the CCR: N, Z, V and C reflect the actual condition of the data after the operation. The V bit indicates sign error for these instructions. V =1 if the sign of the data changed during the shift operation.

There are four versions of the logical shift right instruction: LSR, LSRA, LSRB, and LSRD, as shown in Figure 3.23. There implementation is identical to the logical shift left instructions already described. The logical shift right instructions affect four flag bits in the CCR: N is cleared and Z, V and C reflect the actual condition of the data after the operation. The V bit indicates sign error for these instructions. V =1 if the sign of the data changed during the shift operation.

Example 3.14

Problem: Describe what each of the following instructions accomplishes and then indicate the addressing mode used, the result and resulting status in the CCR, given the following:

```
0000   81 22 E4 4A 55 6B C7 90      A     $C7
0198   FE DC BA 00 10 1F 00 FF      B     $5E
                                    CCR   $D4
```

```
    Address   Machine Code   Source Code
a.  B700      48             LSLA
b.  002E      74 01 9A       LSR   $019A
c.  012C      04             LSRD
```

Solution:

a. Mode = INH. Nothing in the operand field. Shift left \$C7, make LSB 0, stuff MSB into C flag. Result \$8E, C = 1. N = 1, Z = 0, V = 0, C = 1 → CCR = \$D9.

b. Mode = EXT. The two bytes in the operand field without # sign. Shift right \$BA, make MSB 0, stuff LSB into C flag. Result \$5D, C = 0. N = 0, Z = 0, V = 1, C = 0 → CCR = \$D2.

c. Mode = INH. Nothing in the operand field. Shift right \$C75E, make MSB 0, stuff LSB into C flag. Result \$63AF, C = 0. N = 0, Z = 0, V = 1, C = 0 → CCR = \$D2.

Arithmetic Shift Instructions

The arithmetic shift right instructions are similar to the logical shift right instructions. They shift a data word one bit to the right. The LSB is shifted into the C flag of the CCR; however, the MSB is retained. By holding the MSB, the sign of the data is retained allowing special arithmetic operations on 2's complement data (signed data). In Figure 3.24, there is an 8-bit data word \$A7 that will be shifted right. Before the shift is performed, the C flag in the CCR contains zero. When the shift right operation is performed, the MSB from the data is shifted back into the MSB position. The original 8-bit word is moved one bit position to the right and LSB is moved into the C flag of the CCR. Thus the resulting data word is \$D3 and C flag = 1.

There are three versions of this instruction: ASR, ASRA and ASRB, as shown in Figure 3.25. The ASR operates directly on an 8-bit data word stored at a memory location. It operates in the EXT, INDX and INDY addressing modes. No support is provided for the DIR addressing mode. The ASRA and ASRB instructions operate on the 8-bit data contained in the A and B accumulators respectively. There is no 16-bit version of the arithmetic shift right instruction. ASRA and ASRB operate in the inherent addressing mode. The arithmetic shift right instructions affect four status flags in the CCR: N, Z, V and C reflect

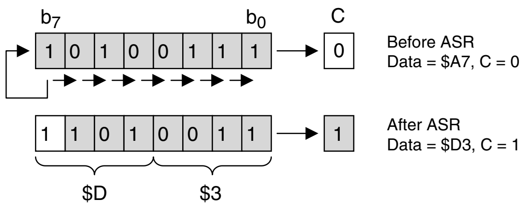


Figure 3.24 Arithmetic Shift Right Example

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
ASRA ASRB	Arithmetic shift right Acc	*See Figure 3.24*	-	-	-	-	-	X	-	-	-	-	-	↕	↕	↕	↕
ASR	Arithmetic shift right contents of memory	*See Figure 3.24*	-	-	X	X	X	-	-	-	-	-	-	↕	↕	↕	↕

Figure 3.25 Arithmetic Shift Instructions

the actual condition of the data after the operation. The V bit still indicates sign error for these instructions, even though a sign error cannot occur since the sign bit is retained.

NOTE: The logical shift left and arithmetic shift left instructions are identical in function and share the same opcodes. The different mnemonic forms are provided for each type only to aid in the readability of source code (i.e., when logical operations are being performed, logical mnemonic forms are used). Therefore, no discussion or examples of the operation of arithmetic shift left instructions are provided.

Example 3.15

Problem: Describe what each of the following instructions accomplishes and then indicate the addressing mode used, the result and resulting status in the CCR, given the following:

```
    0100   24 78 BA 41 93 1F BE FF     B     $41
                                       CCR   $D4

    Address   Machine Code   Source Code
a.  0100      57             ASRB
b.  01E3      77 01 06       ASR    $0106
```

Solution:

a. Mode = INH. Nothing in the operand field. Shift right $41, retain the MSB, stuff LSB into C flag. Result $20, C = 1. N = 0, Z = 0, V = 0, C = 1, thus CCR = $D1.

b. Mode = EXT. The two bytes in the operand field without # sign. Shift right $BE, retain the MSB, stuff LSB into C flag. Result $DF, C = 0. N = 1, Z = 0, V = 0, C = 0, thus CCR = $D8.

Rotate Instructions

The rotate left (ROL) instructions are responsible for rotating a data word one bit to the left. The MSB is shifted into the C flag of the CCR, and the LSB is filled with the

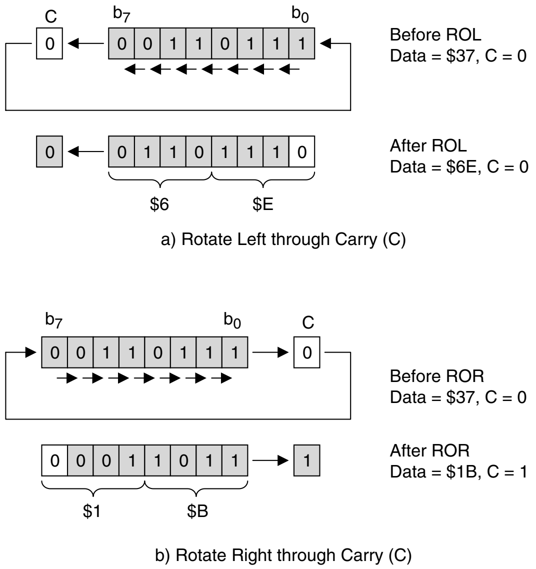


Figure 3.26 Rotate Examples

prior contents of the C flag. In Figure 3.26a, there is an 8-bit data word $37 that will be rotated left. Before the rotate is performed, the C flag in the CCR contains zero. When the rotate left operation is performed, the MSB is shifted into the C flag and the prior value in C is rotated into the LSB from the right. The original 8-bit word is moved one bit position to the left. Thus the resulting data word is $6E and C flag = 0.

The rotate right (ROR) instructions are responsible for rotating a data word one bit to the right. The LSB is shifted into the C flag of the CCR, and the MSB is filled with the prior contents of the C flag. In Figure 3.26b, there is an 8-bit data word $37 that will be rotated right. Before the rotate is performed, the C flag in the CCR contains zero. When the rotate left operation is performed, the LSB is shifted into the C flag and the prior value in C is rotated into the MSB from the left. The original 8-bit word is moved one bit position to the right. Thus the resulting data word is $1B and C flag = 1.

There are three versions of the rotate left instruction: ROL, ROLA and ROLB, as shown in Figure 3.27. The ROL operates directly on an 8-bit data word stored at a memory location. It operates in the EXT, INDX and INDY addressing modes. No support is provided for the DIR addressing mode. The ROLA and ROLB instructions operate on the 8-bit data contained in AccA and AccB respectively. They operate in the inherent addressing mode. The rotate left instructions affect four status flags in the CCR: N, Z, V and C reflect the actual condition of the data after the operation. The V bit indicates sign error for these instructions. V = 1 if the sign of the data changed during the rotate operation.

There are three versions of the rotate right instruction: ROR, RORA and RORB, as shown in Figure 3.27. The ROR operates directly on an 8-bit data word stored at a

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
ROLA ROLB	Rotate left Acc	*See Figure 3.26*	-	-	-	-	-	X	-	-	-	-	-	↕	↕	↕	↕
ROL	Rotate left contents of memory	*See Figure 3.26*	-	-	X	X	X	-	-	-	-	-	-	↕	↕	↕	↕
RORA RORB	Rotate right Acc	*See Figure 3.26*	-	-	-	-	-	X	-	-	-	-	-	↕	↕	↕	↕
ROR	Rotate right contents of memory	*See Figure 3.26*	-	-	X	X	X	-	-	-	-	-	-	↕	↕	↕	↕

Figure 3.27 Rotate Instructions

memory location. It operates in the EXT, INDX and INDY addressing modes. No support is provided for the DIR addressing mode. The RORA and RORB instructions operate on the 8-bit data contained in AccA and AccB respectively. They operate in the inherent addressing mode. The rotate right instructions affect four status flags in the CCR: N, Z, V and C reflect the actual condition of the data after the operation. The V bit indicates sign error for these instructions. V = 1 if the sign of the data changed during the rotate operation.

Example 3.16

Problem: Describe what each of the following instructions accomplishes and then indicate the addressing mode used, the result and resulting status in the CCR, given the following:

```
    0100  24 78 BA 41 93 1F BE FF      B    $45
                                       CCR  $D7

    Address  Machine Code  Source Code
a.  0100     56            RORB
b.  01E3     79 01 06      ROL    $0106
```

Solution:

a. Mode = INH. Nothing in the operand field. Rotate right $45, MSB = C, stuff LSB into C flag. Result $A2, C = 1. N = 1, Z = 0, V = 1, C = 1, → CCR = $DB.

b**.** Mode = EXT. The two bytes in the operand field without # sign. Rotate left $BE, LSB = C, stuff MSB into C flag. Result $7D, C = 1. N = 0, Z = 0, V = 1, C = 1, → CCR = $D3.

Self-Test Questions 3.6

1. Which status flag bit is used by all shift and rotate instructions?
2. What is the difference between arithmetic shift and logical shift?
3. What type of rotate instruction is used when the data is shifting from the MSB to the LSB of the data word?

3.7 Multiplication and Division

The HC11 supports 8-bit by 8-bit multiplication as well as 16-bit by 16-bit division. Multiplication is accomplished by the MUL instruction. Division is accomplished by two divide instructions, IDIV and FDIV. These instructions are summarized in Figure 3.28.

Multiply Instruction

MUL calculates a 16-bit unsigned product from two 8-bit unsigned words. It assumes that the 8-bit words are in each of the 8-bit accumulators, AccA and AccB. The 16-bit result is stored in the 16-bit accumulator, AccD, as shown in Figure 3.29. MUL instructions interpret all values as unsigned numbers. MUL operates in the inherent addressing mode and affects only the C flag of the CCR. C = 1 only when the MSB of lower byte of the result is set. The carry flag is used for multi-precision multiplication.

See Appendix A for an explanation of multi-precision multiplication.

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
MUL	8-bit Multiply	(A) * (B) ⇒ D	-	-	-	-	-	X	-	-	-	-	-	-	-	-	↕
IDIV	16-bit integer divide	(D) / (X) ⇒ X Remainder ⇒ D	-	-	-	-	-	X	-	-	-	-	-	-	↕	0	↕
FDIV	16-bit fractional divide	(D) / (X) ⇒ X Remainder ⇒ D	-	-	-	-	-	X	-	-	-	-	-	-	↕	↕	↕

Figure 3.28 Multiply and Divide Instructions

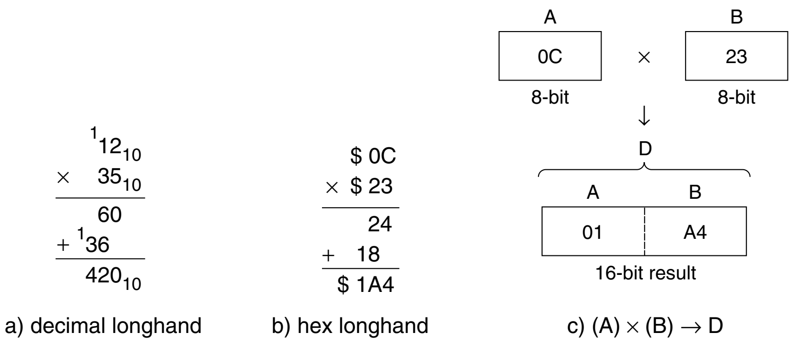


Figure 3.29 HCII Multiplication

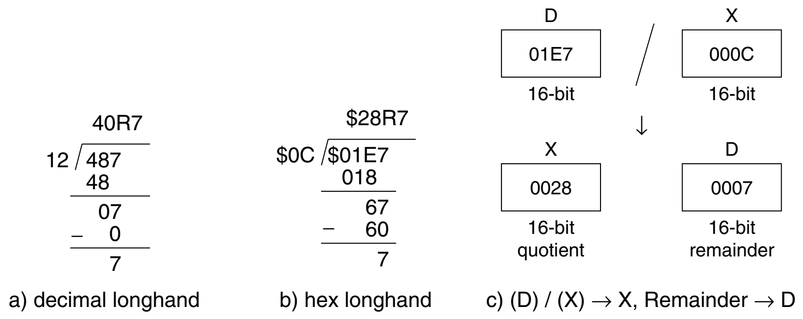


Figure 3.30 HCII Integer Divide

Integer Division Instruction

IDIV calculates a 16-bit unsigned quotient from two 16-bit unsigned words. The 16-bit numerator is contained in the D accumulator, and the 16-bit denominator is contained in the X register. The result (or the quotient) is placed in the X register and the remainder is placed in the D accumulator, as shown in Figure 3.30. A divide by zero condition causes the quotient to be set to $FFFF and the remainder is undefined. IDIV operates in the inherent addressing mode. It affects three status flags in the CCR: V= 0 and Z and C reflect the actual condition of the data after the operation. C = 1 only if the denominator was $0000.

Fractional Division Instruction

FDIV calculates a 16-bit unsigned fraction of two 16-bit unsigned words. The 16-bit numerator is contained in the D accumulator and is expected to be smaller than the 16-bit denominator contained in the X register. The result (or quotient) is placed in the X register and the remainder is placed in the D accumulator. The result is interpreted as a binary weighted fraction. An overflow occurs when the numerator is greater than the denominator. An overflow condition causes the quotient to be set to $FFFF and the remainder is undefined. FDIV operates in the inherent addressing mode. It affects three status flags in the CCR: V = 1 only if the numerator was greater than the denominator, Z reflects the actual condition of the data after the operation and C = 1 only if the denominator was $0000.

Figure 3.31 contains an example that illustrates the concept of **binary-weighted fractions**. The decimal fraction is 3/16, shown in decimal form as 0.1875. This fraction can be represented in 16-bit hex as $0003/$0010. The purpose of executing the FDIV instruction is to calculate a binary-weighted equivalent of 0.1875. If $0003 is loaded into AccD and $0010 into the X register, the FDIV instruction would result in $3000. This result represents the binary weighting shown in Figure 3.32. From this illustration, it is clear that 3/16 is equal to 1/8 plus 1/16. Thus, the binary-weighted fraction $3000 is equal to 3/16.

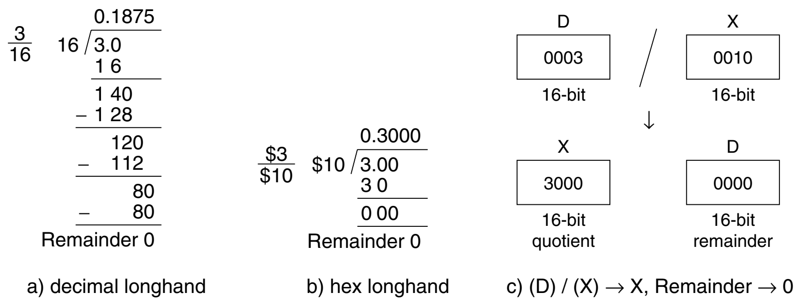


Figure 3.31 HCII Fractional Divide

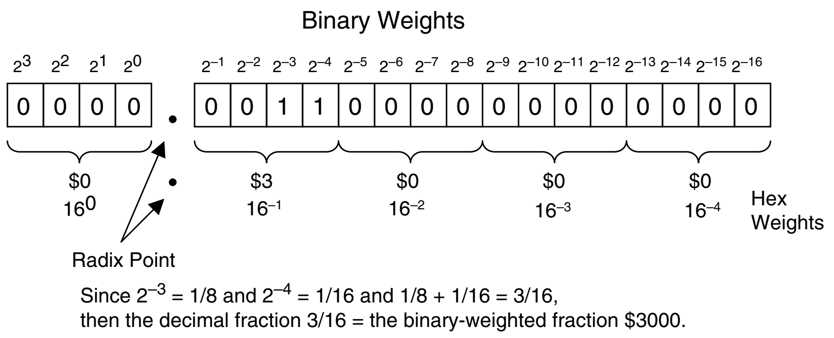


Figure 3.32 Binary-Weighted Fractional Equivalents

NOTE: Most calculators cannot perform calculations on fractional hex and binary numbers. The result of the FDIV instruction can be calculated by multiplying the numerator by $10000, then dividing by the denominator.

Example 3.17

Problem: Describe the result of each of the following. Show the result (and remainder, if any) in hex.

```
     Address   Machine Code   Source Code
a.   0100      86 2D          LDAA   #45
     0102      C6 61          LDAB   #97
     0104      3D             MUL
b.   016B      CC 05 98       LDD    #1432
     016E      C6 00 C5       LDX    #197
     0171      02             IDIV
```

Solution:

a. This program multiplies two decimal numbers, 45 and 97. The numbers are converted into hex and loaded into AccA and AccB respectively. The MUL instruction multiplies the two 8-bit numbers to determine the 16-bit product that is loaded into AccD.

$$45 \times 97 = 4{,}365 \rightarrow \$2D \times \$61 = \$110D.\ \$110D = 4{,}365.$$

b. This program divides two decimal numbers, 1,432 by 197. The numbers are converted into hex and loaded into AccD and the X register respectively. The IDIV instruction divides the two 16-bit numbers to determine the 16-bit quotient that is loaded into the X register; the remainder is loaded into AccD.

$$1{,}432/197 = 7\ \mathrm{R}\ 53\ (7.2690) \rightarrow \$0598/\$00C5 = \$0007\ \mathrm{R}\ \$0035\ (\$7.44).$$

Self-Test Questions 3.7

1. What is the largest number that can be directly multiplied on the HC11?
2. Where is the result of the MUL instruction stored?
3. Which register contains the remainder of a division?
4. What happens if an attempt is made to divide by zero?
5. Which division instruction would be used to find the binary-weighted equivalent fraction of any number less than zero?

3.8 Status Flag Manipulation

Sometimes the state of a status flag needs to be set or cleared prior to some other operation to assure proper operation of another instruction. The HC11 provides four instructions that can directly manipulate status flags in the CCR. These instructions are summarized in Figure 3.33. CLC and SEC clear or set the carry flag (C) in the CCR, respectively. CLV and SEV clear or set the twos complement overflow flag (V) in the CCR, respectively. These four instructions operate in the inherent addressing mode and have no other effect on the CCR.

For example, all of the shift and rotate instructions use the C flag of the CCR. Thus, the C flag needs to be initialized to the desired state (1 or 0) before performing these operations. Many instructions, like the load instructions, do not affect the C flag. If the load instruction is used prior to a rotate and it is desirable to have the C flag cleared, simply execute the CLC instruction.

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
CLC	Clear carry flag	$0 \Rightarrow C$	-	-	-	-	-	X	-	-	-	-	-	-	-	-	0
SEC	Clear carry flag	$1 \Rightarrow C$	-	-	-	-	-	X	-	-	-	-	-	-	-	-	1
CLV	Clear overflow flag	$0 \Rightarrow V$	-	-	-	-	-	X	-	-	-	-	-	-	-	0	-
SEV	Clear overflow flag	$1 \Rightarrow V$	-	-	-	-	-	X	-	-	-	-	-	-	-	1	-

Figure 3.33 Status Flag Manipulation Instructions

Self-Test Questions 3.8

1. Can status flags in the CCR be directly modified by instructions? How?
2. What instruction is used to set the V flag?

Summary

Instructions tell the processor which functions should be performed. The order in which the instructions are presented to the processor determines the order in which the functions are performed. Some instructions are responsible for the movement of data. The HC11 instruction set allows data to be moved using three different methods: from memory to the processor registers (load instructions), from the processor registers to memory (store instructions) and from one processor register to another processor register (transfer and exchange instructions).

Other instructions are responsible for mathematical calculations. The basic arithmetic operations of addition, subtraction, multiplication and division are supported. In addition, instructions to increment, decrement or change the sign of a value are supported. The logical operations AND, OR, XOR and NOT are also provided.

The operation of single instructions and the combination of a few instructions in short program segments are the focus of this chapter. The application and examples are limited to the inherent, immediate, direct and extended addressing modes to reinforce their use.

Chapter Questions

Section 3.1

1. What is the H flag used for?
2. What is the N flag used for?

Section 3.2

3. List the three ways that the movement of data can be accomplished.
4. What type of instructions are used to accomplish the data movement methods listed for question #3?
5. What is the address of the data that gets loaded by the following instruction: LDAA $4B?
6. How many bytes of data are loaded by the following instruction: LDX $00?
7. What is the mnemonic used for the instruction that stores the contents of AccA?
8. What is the source of the data that gets stored by the following instruction: STAB $019F?
9. Why is the CLRA instruction similar to the LDAA instruction?
10. What happens to AccA when a TAB instruction is executed?
11. How many instructions would be necessary to duplicate the function of a single exchange instruction?

Section 3.3

12. What condition causes the C flag to be set during an add instruction?
13. What condition causes the H flag to be set during an add instruction?
14. What actually takes place when the following instruction is executed: ADDA $0100?
15. Can the DAA instruction perform a decimal adjust after a subtract instruction?
16. How can the function of the INCA instruction be duplicated using an ADDA instruction?

Section 3.4

17. What condition causes the C flag to be set during a subtract instruction?
18. What condition causes the H flag to be set during a subtract instruction?
19. What addressing modes support the negate instructions?
20. What is the advantage of a decrement instruction over the equivalent subtract instruction?

Section 3.5

21. What four logical operations are supported by the HC11?
22. What series of instructions could be used to create a logical NAND function?

Section 3.6

23. What is the significant difference between a logical and an arithmetic shift right?
24. Which bit in the CCR is used by all of the shift and rotate instructions?
25. How many times would the LSRA instruction have to be executed to shift the upper nibble to the lower nibble position?
26. Why are there different mnemonic forms for logical and arithmetic shift left if they share the same opcodes?
27. Which flags in the CCR are affected by the ROLA instruction?

Section 3.7

28. Where are the results of a multiply instruction located at the end of the instruction?
29. Where is the result of an integer divide instruction located at the end of the instruction?
30. What happens if the X register contains zero and the IDIV instruction is executed?
31. If the numerator is less than the denominator, which divide instruction should be used?

Section 3.8

32. What addressing mode is used by the CLC instruction?

Chapter Problems

1. Using the diagram in Figure 3.3, derive a logic function for the Z flag. Validate the function with the data from Figure 3.34.
2. Using the data in Figure 3.34, express the results of the following instructions.

```
Registers  C =  $C4        Memory  00F8   00 11 22 33 44 55 66 77
           D =  $B600              0100   F3 56 E3 DB A1 A0 09 00
           X =  $00A0              0108   22 44 52 88 63 77 74 33
           Y =  $0110              0110   FF 00 FF 11 FF 22 FF 83
           S =  $0107              0118   F1 3B BB B6 D4 AD CE 00
```

Figure 3.34 Default Data Set #1

Which addressing mode is used for each instruction? NOTE: Each instruction is independent of the others. The results are not cumulative.

a. `LDAA  #$F9`
b. `STX   $0108`
c. `TBA`
d. `CLRB`
e. `XGDX`
f. `LDD   $F9`
g. `TSX`
h. `STS   $0193`
i. `CLR   $0029`
j. `TAP`
k. `LDAB  $FB`

3. Using the data in Figure 3.35 draw a flowchart and write a program that will transfer the contents of AccA to AccB and then exchange the contents of the X and D registers.
4. Using the data from Figure 3.35, express the results of the following instructions. Which addressing mode is used for each instruction? NOTE: Each instruction is independent of the others. The results are not cumulative.

a. `ADDA  #$49`
b. `ABX`
c. `INCB`
d. `DEC   $0100`
e. `ADDD  #$0100`
f. `SUBB  $13`
g. `DES`
h. `ADDB  $010B`
i. `SBA`

```
Registers  C =  $D5        Memory  0000   52 88 63 33 44 55 66 77
           D =  $0082              0008   F3 56 E3 DB A1 A0 09 00
           X =  $0100              0010   22 44 52 88 63 77 74 33
           Y =  $0000              0100   FF 00 88 63 77 74 FF 83
           S =  $0041              0108   44 52 88 63 52 88 63 00
                                   0110   33 44 55 B6 A1 A0 09 00
```

Figure 3.35 Default Data Set #2

5. Draw a flowchart and write a program that will add $3C and $42 and store the sum at location $0020.
6. Draw a flowchart and write a program that will add the BCD values $26 and $39, perform the proper decimal adjust as needed and store the sum at location $0180.
7. Draw a flowchart and write a program that will subtract the contents of memory locations $0000 and $0001 and store the result in location $0002.
8. Using the data from Figure 3.35, express the results of the following instructions. Which addressing mode is used for each instruction? NOTE: Each instruction is independent of the others. The results are not cumulative.

   a. `EORB   $0E`
   b. `COMA`
   c. `COM    $0103`
   d. `ORAB   $0012`
   e. `ANDB   $06`
9. Draw a flowchart of a program that multiplies the two numbers located in memory locations $0134 and $0135 and then divides the result by $000A.
10. Write a program that will divide $04A3 by $001B. Show the result and remainder and where they are stored.
11. Write instructions that will do the following:

    a. Set the C flag in the CCR.
    b. Clear the V flag in the CCR
    c. Set the Z flag in the CCR.
12. Using the data from Figure 3.35, comment each line of code using transfer notation and express the results of the programs. NOTE: The results of each instruction within each program are cumulative; however, each program is independent of the others.

    a.
    ```
    LDAA   $03
    LDAB  #$59
    ABA
    DAA
    ```
    b.
    ```
    ADDA    $0100
    SUBA   $0A,Y
    STAA   $0102
    ```
    c.
    ```
    DECB
    COMA
    MUL
    ```

Answers to Self-Test Questions

Section 3.1

1. There are a total of 8-bits in the CCR. Five bits are used for status.
2. The Z flag is set (Z=1) indicating the zero condition.

3. When the V flag is set (V=1), a sign overflow has occurred which means that the sign of the previous result is wrong.

Section 3.2

1. Load instructions are bringing data into the processor from memory, so they are read instructions.
2. Store instructions are copying data from the processor into memory, so they are write instructions.
3. The CLR instruction stores zeros in memory.
4. Because the operands are already in a register that is accessed by the transfer instructions.
5. 16-bits are copied from two different registers.

Section 3.3

1. The H flag is only set by addition instructions.
2. DAA will adjust a hex result from the addition of two BCD numbers back to a valid BCD.
3. Yes. Increment adds one to the register.

Section 3.4

1. The H flag is only set by addition instructions.
2. The negate instructions perform a subtract from zero operation to complete the negation.
3. Yes. Decrement subtracts one from the register.

Section 3.5

1. The HC11 supports AND, OR, XOR and NOT.
2. ANDx, ORAx, EORx and COMx operate exclusively on data in AccA or AccB. Only COM will operate on data in memory.

Section 3.6

1. The C flag is used by all shift and rotate instructions.
2. The arithmetic shift right instructions retain the sign bit, but the logical shift right instructions shift a zero into the MSB. There is no functional difference between arithmetic and logical left instructions.
3. MSB to LSB movement would require a rotate right instruction.

Section 3.7

1. The largest multiplication operation directly supported on the HC11 is unsigned 8-bit × 8-bit, therefore the largest value is 255.
2. The result of MUL is stored in the 16-bit AccD.
3. The X register contains the remainder of a division operation.
4. The result is set to $FFFF in AccD, the remainder is undefined and C flag is set.
5. FDIV calculates binary weighted fractions of values less than zero.

Section 3.8

1. Yes. The CLC, SEC, CLV and SEV instructions modify the C and V bits.
2. SEV will set the V Flag.

c h a p t e r 4

Branching and Loops

Objectives

After completing this chapter, you should be able to:

- Use branch and jump instructions in short programs
- Calculate the destination address when using branch instructions
- Calculate the relative address when using branch instructions
- Evaluate the conditional branch instructions to determine whether the branch test passes or fails
- Use compare instructions to update the status flags in the condition code register
- Use branch instructions to implement the IF-THEN-ELSE programming structure
- Implement finite loops using the WHILE and UNTIL programming structures
- Make a simple time delay using a counter in a finite program loop

Outline

Introduction

Chapter 3 focused on instructions that have various jobs of manipulating the data in the registers and in memory. This chapter will focus on a different class of instructions that have the job of controlling or altering the flow of a program. The function of the jump instruction is discussed first, followed by a presentation on the function of the branch instructions.

Sometimes the flow of the program needs to be altered unconditionally. For example, every time a program reaches a certain point it may need to go to back to the beginning and start over. Two methods will be presented to accomplish the unconditional change in the flow of the program, one using the jump instruction and the other using the branch always instruction.

Most of the time the need to alter the flow of a program is conditional. For example, an instruction might say to go back to the beginning of the program and start over only if some condition is true. Conditional instructions perform tests that determine if the alternative paths will be followed. Three different programming structures will be presented with practical applications that all utilize the conditional methods of altering the flow of a program.

This chapter directly correlates to sections 6.2.6 and 6.3.4 of the HC11 Reference Manual and Section 3.4.6 of the Technical Data Manual.

4.1 Jumping

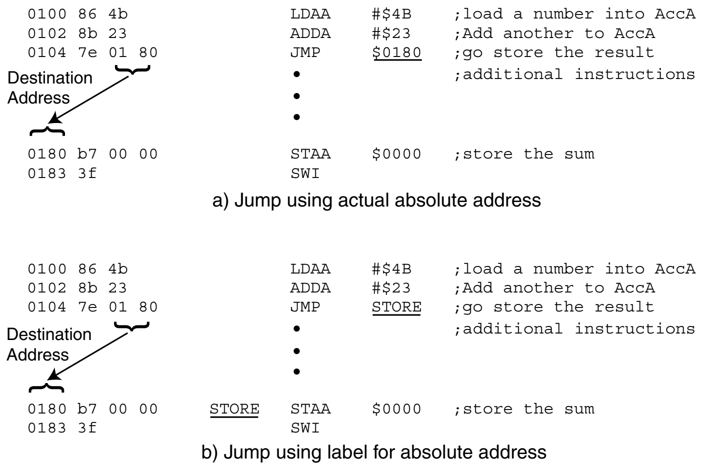
Jump instructions cause a change in the program flow to any location within the addressable range of the processor. The **jump** instruction directly loads the program counter (PC) with a 16-bit address. This address is the location of the next instruction to be executed, commonly referred to as the **destination address**. This jump instruction is unconditional; thus it will alter the flow of the program each and every time it is executed. Jump instructions are used in a program any time an absolute change of program flow is required.

The jump instruction has only one mnemonic form on the HC11, JMP. The JMP instruction is defined for EXT, INDX and INDY addressing modes, as shown in Figure 4.1. The addressing mode is used to designate the destination address. The JMP instruction does not affect the bits in the CCR.

Jump instructions used in the source code are usually followed by a label. The **label** is a name that the programmer makes up. Typically, it represents the destination address

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
JMP	Change flow to absolute address	hhll → PC	-	-	X	X	X	-	-	-	-	-	-	-	-	-	-

Figure 4.1 Jump Instructions



```
0100 86 4b                LDAA    #$4B    ;load a number into AccA
0102 8b 23                ADDA    #$23    ;Add another to AccA
0104 7e 01 80             JMP     $0180   ;go store the result
                          •               ;additional instructions
                          •
                          •

0180 b7 00 00             STAA    $0000   ;store the sum
0183 3f                   SWI
```

a) Jump using actual absolute address

```
0100 86 4b                LDAA    #$4B    ;load a number into AccA
0102 8b 23                ADDA    #$23    ;Add another to AccA
0104 7e 01 80             JMP     STORE   ;go store the result
                          •               ;additional instructions
                          •
                          •

0180 b7 00 00     STORE   STAA    $0000   ;store the sum
0183 3f                   SWI
```

b) Jump using label for absolute address

Figure 4.2 Jump Instructions

of the jump or branch instruction. The same label is used to the left of the instruction that will be executed following the jump instruction. Consider the program example shown in Figure 4.2. This program loads a number into AccA, then adds another number to the value already in AccA. Next it jumps to another part of memory and stores the result of the addition. Figure 4.2a shows the jump instruction in the extended mode and the destination address is \$0180. Figure 4.2b illustrates how the same result can be accomplished by using the label “STORE.” The label “STORE” represents the destination address and is properly replaced with the actual destination address in memory.

Self-Test Questions 4.1

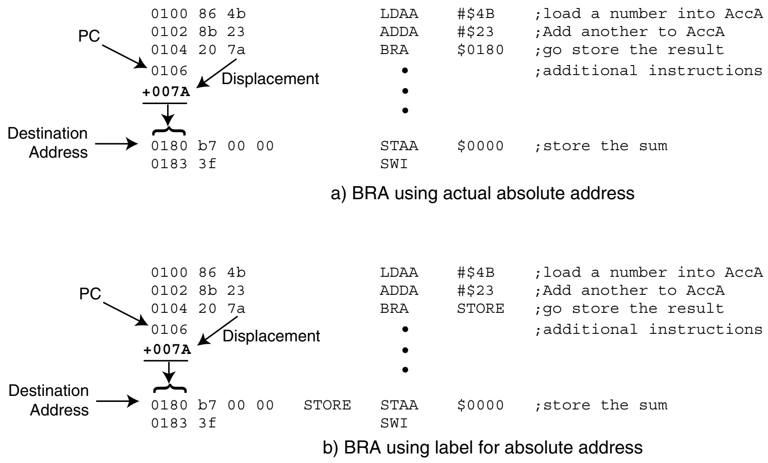
1. What is a label? How are labels used with jump instructions?
2. What kind of address is used by all jump instructions?

4.2 Branching and Relative Address Mode

Branch instructions differ from jump instructions in that they do not supply the absolute address of the next instruction. When a **branch** instruction is executed, the destination address must be calculated relative to the current program counter. Thus, this set of instructions uses the relative addressing mode.

Calculating the Destination Address

In the relative mode, a relative address follows the opcode in memory. It is added to the current program counter to determine the destination address. The **relative address** is a signed 8-bit number that indicates the number of bytes to branch forward



```
0100 86 4b                LDAA    #$4B    ;load a number into AccA
0102 8b 23                ADDA    #$23    ;Add another to AccA
0104 20 7a                BRA     $0180   ;go store the result
0106                       •              ;additional instructions
                           •
                           •
0180 b7 00 00             STAA    $0000   ;store the sum
0183 3f                   SWI
```

a) BRA using actual absolute address

```
0100 86 4b                LDAA    #$4B    ;load a number into AccA
0102 8b 23                ADDA    #$23    ;Add another to AccA
0104 20 7a                BRA     STORE   ;go store the result
0106                       •              ;additional instructions
                           •
                           •
0180 b7 00 00     STORE   STAA    $0000   ;store the sum
0183 3f                   SWI
```

b) BRA using label for absolute address

Figure 4.3 BRA Instructions

or backward in memory to the destination address. It is a displacement from the current position in memory. Since it is a signed 8-bit number, it must be sign extended to 16 bits before being added to the contents of the 16-bit program counter. The result of this operation produces the destination address, as shown in Equation 4.1.

$$DA = PC + rr \qquad \text{Equation 4.1}$$

DA = Destination Address

PC = Address in the Program Counter

rr = Sign-Extended Relative Address

Figure 4.3 contains the same example used in Figure 4.1, except the jump instructions have been replaced with branch instructions. Instead of the absolute 16-bit address following a jump instruction, a relative address is provided following the branch instruction. This relative address must be added to the current program counter to calculate the destination address. In Figure 4.3, the branch instruction is assembled as two bytes in memory. The opcode for a BRA instruction is $20; it is located at location $0104. The operand field is located at location $0105 and is occupied by the relative address $7A. Figure 4.4 summarizes the sequence of events that occur to process this instruction.

Since the program counter contains $0180 at the end of this instruction, the instruction located at $0180 (STAA $0000) will be executed. The flow of the program was altered by this branch instruction, because the next sequential instruction was located at $0106. Each time a branch instruction is executed the processor must calculate the destination address in this manner. Additional examples of how this is done are shown in Example 4.1.

Operation	Description	Functional Result	
Fetch BRA opcode (1 machine cycle)	Contents of PC are copied to MAR.	(PC) → MAR	\$0104 → MAR
	PC is incremented.	(PC) + 1 → PC	\$0105 → PC
	Opcode is read from memory.	(\$0105) → data bus	\$20 → data bus
	Opcode is loaded into IR.	Opcode → IR	\$20 → IR
Execute BRA (2 machine cycles)	Contents of PC are copied to MAR.	PC → MAR	\$0105 → MAR
	PC is incremented.	PC + 1 → PC	\$0106 → PC
	Displacement is read from memory.	rr → MDR	\$7A → MDR
	Calculate destination address.	PC + rr → PC	\$0106 + \$007A = \$0180 → PC

Figure 4.4 Process of Executing the BRA Instruction

Example 4.1

Problem: Calculate the destination address (DA) for each of the following branch instructions.

	Address	Machine Code	Source Code	
a.	0109	20 6C	BRA	PAST
b.	01E2	20 D3	BRA	LOOP
c.	B723	20 9B	BRA	AGAIN

Solution: In each case, the contents of the PC must be determined. Then the sign-extended relative address must be added to the PC to calculate the destination address (Equation 4.1). The PC is the address of the next sequential instruction. Since each of these instructions occupies two bytes of memory, the address of the next instruction (PC) will be two greater than the address of the BRA instruction.

a. PC = \$0109 + 2 → \$010B
rr = \$6C, sign extend → $\underline{\text{+\$006C}}$
DA = PC + rr → \$0177

b. PC = \$01E2 + 2 → \$01E4
rr = \$D3, sign extend → $\underline{\text{+\$FFD3}}$
DA = PC + rr → \$01B7

c. PC = \$B723 + 2 → \$B725
rr = \$9B, sign extend → $\underline{\text{+\$FF9B}}$
DA = PC + rr → \$B6C0

Calculating the Relative Address

All relative mode instructions calculate the destination address by adding the relative address to the current program counter. In Figure 4.3 it is shown that the source code includes the actual destination address (i.e., BRA \$0180), yet the machine code uses the displacement. How does the assembler calculate the relative address from the destination address? The answer is found by subtracting PC from both sides of Equation 4.1, as shown in Equation 4.2.

$$\begin{aligned} DA &= PC + rr && \text{Subtract PC from both sides} \\ DA - PC &= PC - PC + rr && \text{Rearrange} \\ rr &= DA - PC \end{aligned}$$

Equation 4.2

Because the relative mode offset is a signed 8-bit value, the distance that a program can branch forward or backward is limited. The largest positive decimal number that can be represented in eight bits is 127, and the largest decimal negative number is –128. Thus, forward displacement is limited to 127 bytes, and backward displacement is limited to 128 bytes. Each time a branch instruction is assembled, the assembler must calculate the relative address. Examples of how this is done are shown in Example 4.2.

Example 4.2

Problem: Calculate the relative address (rr) for each of the following branch instructions.

```
 Address   Machine Code   Source Code
a.    0109      20 ??            BRA  PAST
      010B      8B 30     OVER   ADDA #$30
      010D      97 10     PAST   STAA $10
b.    018E      C6 0A     BACK   LDAB 0,X
      0190      CB 30            ADDB #$30
      0192      08               INX
      0193      20 ??            BRA  BACK
c.    0173      20 ??            BRA  MAIN
                                 . . .
      01E2      4A        MAIN   DECA
d.    E204      86 45     OUTA   LDAA #$45
                                 . . .
      E27A      20 ??            BRA  OUTA
```

Solution: In each case, the contents of the PC must be determined. Then the relative address can be calculated by subtracting the PC from the DA (Equation 4.2). The PC is the address of the next sequential instruction. Since each of these instructions occupies two bytes of memory, the address of the next instruction (PC) will be two greater than the address of the BRA instruction.

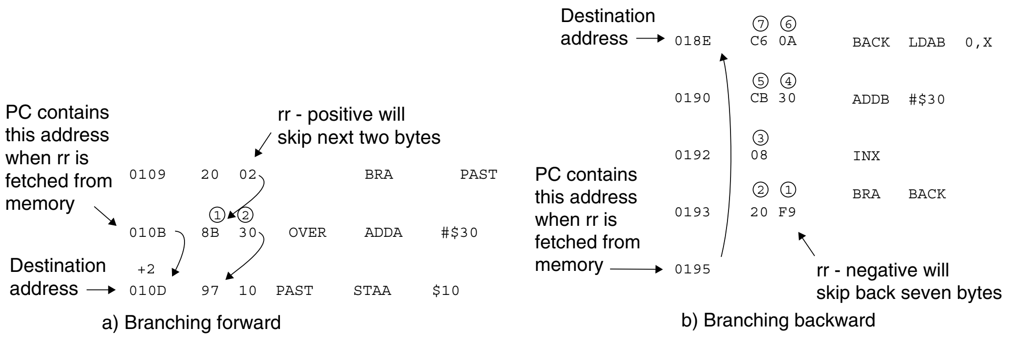
```
a.   DA = 010D        →   $010D
     PC = $0109 + 2  →  +$010B
     rr = DA – PC     →   $0002     rr  =  $02
b.   DA = 018E        →   $018E
     PC = $0193 + 2  →  +$0195
     rr = DA – PC     →   $FFF9     rr  =  $F9
c.   DA = 01E2        →   $01E2
     PC = $0173 + 2  →  +$0175
     rr = DA – PC     →   $006D     rr  =  $6D
d.   DA = E204        →   $E204
     PC = $E25A + 2  →  +$E25C
     rr = DA – PC     →   $FFA8     rr  =  $A8
```

Nature of the Relative Address

Further explanation is in order regarding the nature of the relative address. The relative address was defined earlier as the number of bytes to branch forward or backward in memory to the destination address. What does it mean to branch forward or backward? Figure 4.5 provides some insight into this process.

NOTE: When branching forward, the relative address is positive. It is the number of bytes of memory that will be skipped forward, starting with the first byte following the relative address in memory and going up to, but not including, the byte stored in the destination address.

In Example 4.2a, the relative address is calculated to be $02. This means that the program will skip forward past two bytes of memory. The two bytes of the ADDA #$30 instruction, $8B and $30, are skipped, as shown in Figure 4.5a. The instruction located two bytes forward in memory is executed.



```
0109   20 02          BRA    PAST
010B   8B 30   OVER   ADDA   #$30
010D   97 10   PAST   STAA   $10
```

```
018E   C6 0A   BACK   LDAB   0,X
0190   CB 30          ADDB   #$30
0192   08             INX
0193   20 F9          BRA    BACK
0195
```

PC contains this address when rr is fetched from memory
rr - positive will skip next two bytes
① ②
+2
Destination address
a) Branching forward
Destination address
⑦ ⑥
⑤ ④
③
② ①
PC contains this address when rr is fetched from memory
rr - negative will skip back seven bytes
b) Branching backward

Figure 4.5 The Nature of the Relative Address

NOTE: When branching backward, the relative address is negative. It is the number of bytes of memory that will be skipped backward, starting with the relative address and going back to, and including, the byte stored in the destination address.

In Example 4.2b, the relative address was calculated to be \$F9 (-7_{10}). This means that the program will skip backward seven bytes of memory. The two bytes of the BRA instruction, the one byte of the INX instruction, the two bytes of the ADDA instruction and the two bytes of LDAB instruction are skipped, as shown in Figure 4.5b. The instruction located seven bytes backward in memory is executed.

Self-Test Questions 4.2

1. What kind of address is used by all branch instructions?
2. If the address of a BRA instruction is \$0023 and the Offset = \$62, what is the destination address?
3. If the address of a BRA instruction is \$B61D and the Offset = \$9A, what is the destination address?
4. Calculate the relative address (rr) for each of the following branch instructions.

```
0193     20 ??          BRA   END
                        . . .
01DD     E6 00    END   LDAB   0,X
```

5. Calculate the relative address (rr) for each of the following branch instructions.

```
E25A     97 25    MSG1  STAA $25
                        . . .
E273     20 ??          BRA   MSG1
```

4.3 Branch Instructions

There are 21 branch instructions defined for the HC11. Three of these instructions, BRCLR, BRSET and BSR, have special applications. Their function will not be presented in this section. BRCLR and BRSET are described in chapter 9, and BSR is described in chapter 6. The function of the remaining 18 branch instructions will be the focus of this section.

Each branch instruction performs a test to determine if the branch will be taken. As each test has a complementary form, the instructions are paired. For example, branch if equal to zero (BEQ) is complementary to branch if not equal to zero (BNE). BEQ will cause a branch to occur if the Z flag of the CCR is set, which indicates that the result of a prior instruction was zero. BNE will cause a branch to occur if the Z flag of the CCR is cleared, which indicates that the result of a prior instruction was not zero.

The branch instructions are categorized into functional groups for presentation in this text. These functional groups are unconditional, conditional-simple, conditional-

Mnemonic	Description	Boolean Conditional Test	Complement Form	Functional Group
BRA	Branch always	Always passes	BRN	Unconditional
BRN	Branch never	Never passes	BRA	Unconditional
BCC	Branch if carry cleared (see BHS)	? C = 0	BCS	Conditional-simple
BCS	Branch if carry set (see BLO)	? C = 1	BCC	Conditional-simple
BEQ	Branch if equal to zero	? Z = 1	BNE	Conditional-simple
BMI	Branch if minus (negative)	? N = 1	BPL	Conditional-simple
BNE	Branch if not equal to zero	? Z = 0	BEQ	Conditional-simple
BPL	Branch if plus (positive)	? N = 0	BMI	Conditional-simple
BVC	Branch if overflow cleared	? V = 0	BVS	Conditional-simple
BVS	Branch if overflow is set	? V = 1	BVC	Conditional-simple
BHI	Branch if higher	? C + Z = 0	BLS	Conditional-unsigned
BHS	Branch if higher or same (see BCC)	? C = 0	BLO	Conditional-unsigned
BLO	Branch if lower (see BCS)	? C = 1	BHS	Conditional-unsigned
BLS	Branch if lower or same	? C + Z = 1	BHI	Conditional-unsigned
BGE	Branch if greater than or equal	? N ⊕ V = 0	BLT	Conditional-signed
BGT	Branch if greater than	? Z+(N⊕V) = 0	BLE	Conditional-signed
BLE	Branch if less than or equal	? Z+(N⊕V) = 1	BGT	Conditional-signed
BLT	Branch if less than	? N ⊕ V = 1	BGE	Conditional-signed

Figure 4.6 Summary of Branch Instructions

unsigned and conditional-signed. Each functional group is described in subsequent sections. Figure 4.6 summarizes the 18 branch instructions. All branch instructions use the relative addressing mode and have no affect on the CCR.

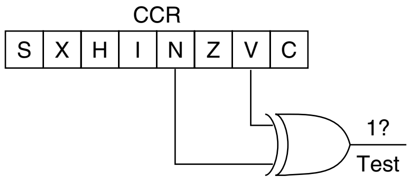
Unconditional Branching

There are two unconditional branch instructions: BRA and BRN. The BRA instruction always passes the branch test and will always branch, as the mnemonic implies (BRA = Branch Always). The BRN instruction never passes the branch test and will never branch, as the mnemonic implies (BRN = Branch Never). The BRN instruction is very useful during troubleshooting to replace another branch instruction as well as in timing loops to cause a time delay.

Conditional Branching

The branch tests are **simple** if they consist of testing the state of a single bit in the CCR. The branch tests are "unsigned" because they perform a branch test from an unsigned perspective. The **unsigned perspective** ignores the sign bit and views all eight (or sixteen) data bits as magnitude. Each of these tests is concerned about the relationship of two unsigned data words. The branch tests are "signed" because they perform a branch test from a signed perspective. The **signed perspective** views the MSB as a sign bit and the remaining bits as a 2's complement magnitude. Each of these tests is

N	V	XOR Test	
0	0	0	fail
0	1	1	pass
1	0	1	pass
1	1	0	fail



$N \oplus V = 1$ only when N and V are opposites, as shown in truth table.

Figure 4.7 BLT Branch Test (? N ⊕ V = 1)

concerned about the relationship of two signed data words. Although BEQ and BNE are conditional-simple, they are used for equality tests in all groups because equal is equal regardless of the sign of the data.

Each of the conditional branch instructions requires that a branch test be performed before the address of the next instruction is calculated. The branch test consists of evaluating a Boolean expression to determine if the condition has been met. For example, if the program is concerned about overflow conditions, it might perform a BVS instruction. The BVS instruction tests to see if the V flag in the CCR is set. The test is shown as ?V = 1, which is read as "does V = 1?" If the V flag is set, the BVS will cause the displacement to be added to the value in the program counter to calculate the address of the next instruction to execute.

NOTE: The branch test is a Boolean expression that must be evaluated before the branch can take place. If the expression evaluates to a true state, the branch test passes and the branch takes place. If the expression evaluates to a false state, the branch test fails and the program falls through to the next instruction.

Some of the conditional-unsigned branches and all of the conditional-signed branches evaluate a Boolean expression for the branch test. Rather than testing for the state of a single bit, each of these branch tests evaluates the status of two or three bits. For example, the branch if less than (BLT) instruction evaluates the Boolean expression, "? N ⊕ V = 1", which is read "Is (N XOR V) equal to 1?" The HC11 performs this test with logic gates, as shown in Figure 4.7. The condition is true only when N and V are opposites.

To emphasize the concept of evaluating the Boolean expression, one more example will presented. Consider the branch if greater than (BGT) instruction. It evaluates the Boolean expression, "? Z + (N ⊕ V) = 0", which is read "Is (Z OR (N XOR V)) equal to 0?" The HC11 performs this test with logic gates, as shown in Figure 4.8. The condition is true only when Z is zero and N and V are equal.

Example 4.3

Problem: Calculate the destination address for each of the following conditional branch instructions if the CCR register contains $DA before the branch is executed.

	Address	Machine Code	Source Code
a.	01D9	2B 0C	BMI PAST
b.	0112	25 53	BCS LOOP
c.	B6C3	23 8B	BLS AGAIN
d.	E0F4	2E F2	BGT AGAIN

Solution: In each case, the processor must determine if the branch test passes or fails. In addition, the contents of the PC must be determined. If the test fails, DA = PC. If the test passes, then the sign-extended relative address must be added to the PC to calculate the destination address (Equation 4.1). Since each of these instructions occupies two bytes of memory, the address of the next instruction (PC) will be two greater than the address of the branch instruction.

CCR = \$DA, thus N = 1, Z = 0, V = 1, C = 0.

a. BMI test checks if N = 1. Since N = 1, the test passes and DA = PC + rr.
PC = \$01D9 + 2 → `$01DB`
rr = \$0C, sign extend → `$000C`
DA = PC + rr → `$01E7`

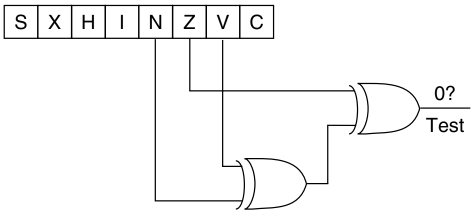
b. BCS test checks if C = 1. Since C = 0, the test fails and DA = PC.
DA = PC = \$0112 + 2 → `$0114`

c. BLS test checks if Z or (N XOR V) = 0. Since Z = 0 and (N XOR V) = 0, then the test passes and DA = PC + rr.
DA = PC = \$B6C3 + 2 → `$B6C5`

d. BGT test checks if N = 1. Since N = 1, the test passes and DA = PC + rr.
PC = \$E0F4 + 2 → `$E0F6`
rr = \$F2, sign extend → `$FFF2`
DA = PC + rr → `$E0E8`

Many instructions update the status flags with the information necessary to complete the branch test. For example, when an LDAA instruction is executed, the N and Z flags

Z	N	V	N⊕V	Z + (N⊕V)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	0
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	1



Z + (N⊕V) = 0 only when both Z and (N⊕V) are equal to zero, as shown in truth table.

Figure 4.8 BGT Branch Test (?Z + (N ⊕ V) = 0)

are updated with the state of the data that was loaded. The C flag is also cleared by an LDAA instruction. A conditional branch instruction can immediately follow and perform branch tests relevant to the status recorded during the LDAA instruction.

Example 4.4

Problem: Calculate the destination address for the following conditional branch instruction.

```
Address   Machine Code   Source Code
01D3      C6 84          LDAB #$84
01D5      2B 03          BMI  PAST
```

Solution: The processor must determine if the branch test passes or fails. In addition, the contents of the PC must be determined. If the test fails, DA = PC. If the test passes, then the sign-extended relative address must be added to the PC to calculate the destination address (Equation 4.1). Since the branch instruction occupies two bytes of memory, the address of the next instruction (PC) will be two greater than the address of the branch instruction.

Since the data loaded by the LDAB is negative, then the N flag in the CCR will be set. Because BMI test checks if N = 1, then the test passes and DA = PC + rr.

```
PC = $01D5 + 2          →   $01D7
rr = $03, sign extend   →  +$0003
DA = PC + rr            →   $01DA
```

Self-Test Questions 4.3

1. Calculate the destination address for the following conditional branch instruction if the CCR register contains $D3 before the branch is executed.

```
0181    2A 2C        BPL  T12
```

2. Calculate the destination address for the following conditional branch instruction if the CCR register contains $DC before the branch is executed.

```
01EE    26 05        BNE  LAST
```

3. Calculate the destination address for the following conditional branch instruction.

```
0177    C6 32        LDAB #$32
0179    CB DE        ADDB #$CE
017B    27 A3        BEQ  TXNOW
```

4.4 Preparation for a Valid Branch Test

In some cases, the branch performs a test that requires the comparison of two bytes of data. For example, BGT is testing to see if one signed number is greater than another signed number. If the first of the two numbers is the greater, the branch test will pass; otherwise, it will fail and the program will fall through. Since all conditional branch instructions test for some condition in the flags of the CCR, these flags must be updated prior to the execution of the BGT in order for a valid comparison to take place. Technically, the BGT instruction does not compare two numbers, but only tests to see if a greater-than condition existed after a comparison was done.

In order to allow the flags to be properly updated prior to a branch instruction that performs a comparative branch test, the HC11 provides a group of compare instructions.

Compare Instructions

Compare instructions are a special set of instructions that only affect status flags in the CCR. No data is changed by any of these instructions. Their job is to compare two numbers and manipulate the flags in the CCR to reflect the relationship of the two numbers. Compare instructions are only executed in preparation for a conditional branch test. Thus, they are used exclusively in conjunction with conditional branch instructions. The comparison of two data words is accomplished by subtracting the two words. Simple logic shows why a subtraction is necessary to complete the compare.

For example, let A be the first word and B be the second word. If A > B, then the result of A – B must be a positive number. If A = B, then the result of A – B must be zero; and if A < B, then the result of A – B must be a negative number.

NOTE: A valid comparison of two data words can take place only if the second word is subtracted from the first word.

Remember that the compare instruction is not concerned with the value of the result. It performs the subtraction operation, updates the appropriate flags in the condition code register and then discards the result of the subtraction. A subtract instruction will cause the same status in the CCR; however, the result of the subtract operation will be saved and the data will be changed. Compare instructions are summarized in Figure 4.9.

The CBA instruction is responsible for comparing the data in AccA to the data in AccB. It operates in the inherent addressing mode. CMPA and CMPB compare the data from an 8-bit accumulator to a byte of data in memory. CPD, CPX and CPY compare the data from a 16-bit register to two bytes of data in memory. Each of these instructions operates in the IMM, DIR, EXT, INDX or INDY address modes. The compare

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
CMPA CMPB	Compare Acc to contents of memory	(A) - (M) (B) - (M)	X	X	X	X	X	-	-	-	-	-	-	↕	↕	↕	↕
CPD CPX CPY	Compare 16-bit register to contents of memory	(D) - (M):(M+1) (X) - (M):(M+1) (Y) - (M):(M+1)	X	X	X	X	X	-	-	-	-	-	-	↕	↕	↕	↕
CBA	Compare A to B	(A) - (B)	-	-	-	-	-	X	-	-	-	-	-	↕	↕	↕	↕

Figure 4.9 Compare Instructions

instructions affect four status flags in the CCR: N, Z, V and C reflect the actual condition of the data after the compare operation.

Example 4.5

Problem: Compare $43 to $56 and update the condition code flags.

```
Address   Machine Code   Source Code
  01D3       86 43          LDAA #$43
  01D5       81 56          CMPA #$56
```

Solution: The compare instruction will subtract $56 from $43, update the condition codes and then discard the result of the subtraction.

```
  $43
 -$56
  $ED
```

$ED is a negative number, thus N is set (N = 1). It is non-zero, thus Z is cleared (Z = 0), there was no sign overflow (the sign of the result is correct), so V is cleared (V = 0) and a borrow was necessary to complete the subtraction, so C is set (C = 1).

Compare to Zero Instructions

Often there is a need to compare some value to zero. Rather than use a regular compare instruction, the HC11 provides special instructions that compare only to zero. They work identically to the compare instructions except that they never require that the second value be provided because it is assumed to be zero. These instructions are called test instructions and come in three mnemonic forms: TST, TSTA and TSTB. They are summarized in Figure 4.10.

TSTA and TSTB compare the contents of one of the 8-bit accumulators to the number zero. The TSTx instructions accomplish this by subtracting zero from the value being tested. They operate in the inherent addressing mode. TST compares a byte of data in memory to zero. It operates in EXT, INDX or INDY addressing modes. No support is

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
TSTA TSTB	Test contents of Acc for zero or minus	(A) - $00 (B) - $00	-	-	-	-	-	X	-	-	-	-	-	↕	↕	0	0
TST	Test contents of memory for zero or minus	(M) - $00	-	-	X	X	X	-	-	-	-	-	-	↕	↕	0	0

Figure 4.10 Compare to Zero Instructions

provided for the DIR addressing mode. The compare instructions affect four status flags in the CCR: N and Z reflect the actual condition of the data after the compare operation. V = 0 and C = 0 because a sign error and borrow condition cannot occur when zero is subtracted from another number.

Example 4.6

Problem: Load a value in AccB and then do a compare to zero.

```
Address   Machine Code   Source Code
 0102       86 01          LDAA #$01
 0104       4D             TSTA
```

Solution: The compare instruction will subtract $00 from $43, update the condition codes and then discard the result of the subtraction.

```
 $01
-$00
 $01
```

$43 is a positive number, thus N is cleared (N = 0). It is non-zero, thus Z is cleared (Z = 0); the V and the C flags are always cleared (V = 0, C = 0).

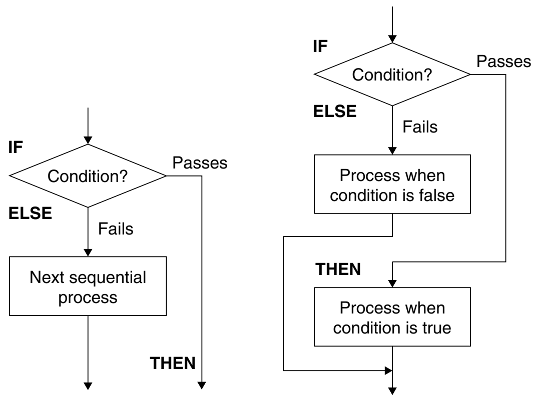
Self-Test Questions 4.4

1. Compare instructions are typically used just prior to a branch instruction. What function do compare instructions perform?
2. What is the difference between the CMPA and TSTA instructions?

4.5 Conditional Flow Using IF-THEN-ELSE

Conditional transfer of control instructions is dependent on the result of a branch test. If the branch test passes, the flow of the program is altered. If the branch test fails, the flow falls through to the next instruction. The **IF-THEN-ELSE** structure uses conditional branch instructions to determine IF a condition is true. IF the condition is true the branch test passes, THEN the program will branch, ELSE the branch test fails and the program falls through to the next instruction.

Using a flowchart, Figure 4.11 illustrates how the IF-THEN-ELSE structure works. The IF is shown with the diamond or decision symbol on the flow chart. The IF is implemented with a conditional branch instruction. IF the test performed by the conditional branch passes, the THEN path will be followed. IF the test performed by the conditional branch fails, then the program falls through to the next instruction in sequence, which is the ELSE process block.



a) Simple IF-THEN b) Complete IF-THEN-ELSE

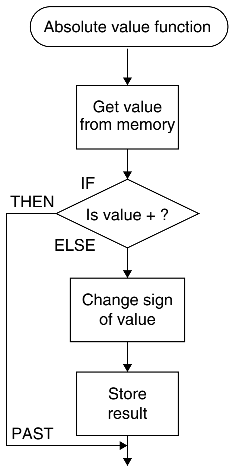
Figure 4.11 IF-THEN-ELSE Flowchart

In some cases, there are specific operations for either condition, which must be performed. Figure 4.11b illustrates the flow of the IF-THEN-ELSE for this scenario. When this scenario occurs, an additional unconditional branch instruction must be included to avoid processing the THEN block after the ELSE block is processed.

Example 4.7

Problem: Write a segment of code using the IF-THEN-ELSE structure to determine the absolute value of an 8-bit signed value. Assume the value is stored at location $0000. Overwrite the value with the absolute value.

Solution:



```
LDAA    $00

BPL     PAST

NEGA

STAA     $00
```

Example 4.7 Flowchart

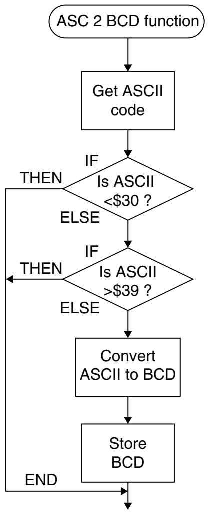
```
        * Absolute value function using IF-THEN-ELSE

0120 96 00        LDAA    $00     ;Get the value from memory
0122 2a 03        BPL     PAST    ;IF positive, THEN do nothing
0124 40           NEGA            ;ELSE change the sign
0125 97 00        STAA    $00     ;Overwrite the original value
0127         PAST . . .           ;Continue other processes
```

The flowchart of Example 4.7 illustrates how this IF-THEN-ELSE would be structured. The LDAA instruction updates the N flag in the CCR; after it successfully reads the value from memory location $0000 and loads AccA, the N flag properly indicates the sign of the data. The BPL instruction checks if N = 0 before performing the branch. If N = 0, the number is already positive so no further action is required to determine the absolute value. The program branches down in the code to PAST, where other processes continue. Notice that the relative address is $03 in memory location $0123. This indicates the number of bytes the program must branch forward to reach PAST (three bytes). There are three bytes between the BPL instruction and the label PAST: $40, $97 and $00. If the original value is negative, the N flag will be set. If N = 1, the branch test fails and the flow falls through to the NEGA instruction, which changes the sign of the 2's complement data. Finally, the absolute value is stored back in memory.

Example 4.8

Problem: Write a segment of code using the IF-THEN-ELSE structure to convert an ASCII code of a hex digit to the actual hex value. Assume the ASCII code is stored in memory at $0010 and that the resulting HEX value will be stored at $0011.

Solution:



```
LDAA    $10

CMPA    #$30
BLO     END

CMPA    #$39
BH1     END

SUBA    #$30

STAA    $11
```

Example 4.8 Flowchart

```
        * Code to convert an ASCII code to BCD digit
        * ASCII $30 → BCD $0, $31 → BCD $1, etc.

0000 96 10          LDAA    $10       ;Get the ASCII code

0002 81 30          CMPA    #$30      ;check code for BCD $0
0004 25 08          BLO     END       ;IF < ASCII $30, not BCD

0006 81 39          CMPA    #$39      ;check code for BCD $9
0008 22 04          BHI     END       ;IF > ASCII $30, not BCD

000a 80 30          SUBA    #$30      ;Convert ASCII to BCD
000c 97 11          STAA    $11       ;Store BCD value

000e        END     . . .             ;continue other processes
```

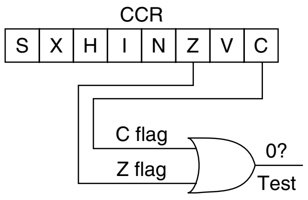
Example 4.8 uses two IF-THEN-ELSE structures to determine if the ASCII code represents a valid BCD digit. It first tests to see if the code is lower (below $30) than the valid range and then if it is higher than the valid range, as shown in the flowchart.

The first CMPA instruction updates the flags in the CCR, in preparation for the branch-if-lower test. The BLO instruction checks if C = 1 before performing the branch. If C = 1, the ASCII code was lower than $30. Since the BCD digits 0 through 9 are represented by the ASCII codes $30 through $39, this is the lowest valid ASCII code that can be converted by this function. The program branches down in the code to END, where other processes continue. Notice that the relative address for the BLO is $08. This indicates the number of bytes the program must branch forward to reach END. If C = 0, the ASCII code was not lower than $30, the branch test fails and the flow falls through to the next CMPA instruction.

The second CMPA instruction updates the flags in the CCR, in preparation for the branch-if-higher test. The BHI instruction evaluates the Boolean expression C + Z = 0 as the branch test before performing the branch. Functionally, this is a two-input OR gate with the C flag connected to one input and the Z flag connected to the other input, as shown in Figure 4.12. The only way to get a zero out of the OR gate is to have two zeros going in. If the expression equals 0, the ASCII code was higher than $39. The program branches down in the code to END, where other processes continue. Notice that the relative address for the BHI is $04. This indicates the number of bytes the program must branch forward to reach END. If the expression evaluates to 1, the ASCII code was not higher than $39, the branch test fails and the flow falls through to the SUBA instruction.

Now that it has been established that the ASCII code is a valid code for a BCD digit (i.e., $30 through $39), the conversion can be completed. The conversion is

C	Z	OR Test	
0	0	0	pass
0	1	1	fail
1	0	1	fail
1	1	1	fail



C + Z = 0 only when C = 0 and Z = 0, as shown in the truth table.

Figure 4.12 BHI Branch Test

accomplished by subtracting $30 from the code via the SUBA instruction. The result of this process is the desired BCD value. Finally, the BCD value is stored in memory location $11.

Self-Test Questions 4.5

1. What class of branch instructions is required to implement the IF-THEN-ELSE structure?
2. What flowcharting symbol is required to show the IF-THEN-ELSE structure?
3. Which path will the program take (THEN path or ELSE path) if the branch test fails?

4.6 Program Loops

A program **loop** is any set of instructions that are repeated within a program. A loop always ends with a branch or a jump instruction. This final instruction causes the flow to return to some location earlier in the code to repeat a set of instructions. Some program loops are conditional; therefore they loop back to an earlier part of the program only when some condition is true. Conditional loops are called **finite loops** because they end whenever the condition is no longer true.

While and Until Loops

Often a program needs to execute a set of instructions only while a condition is true. This is called a **WHILE** loop. In other cases, the program may need to execute a set of instructions until some condition is true. This is called an **UNTIL** loop. A WHILE loop performs the branch test at the beginning of the loop; thus, the loop will never be executed if the condition is never true. An UNTIL loop performs the branch test at the end of the loop; thus, the loop is executed at least once. Figure 4.13 shows the flowcharts for the basic structure of the WHILE loop and the UNTIL loop. Either loop structure can be used to accomplish the same task. The only difference between the structures is the point at which the test is performed.

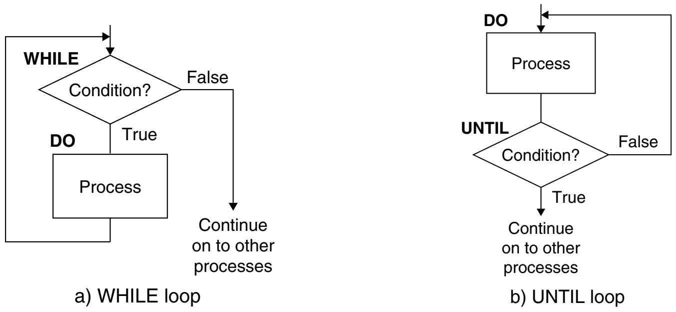


Figure 4.13 WHILE and UNTIL Loop Flowcharts

For example, while the window is open the wind blows in, yet the wind will blow in until the window is closed. The process is the wind blowing in. Notice that the window must be opened before the wind can blow in. This is a WHILE condition. The wind will never blow in unless the window is first opened. The opposite situation is the UNTIL condition. If the window is already open, the wind will continue to blow until it is closed.

The UNTIL structure is easier to code. It requires only a single branch instruction. The WHILE structure requires an additional unconditional branch to skip past the process when the condition is no longer true. This difference can be illustrated with a simple time delay function.

First consider a time delay that utilizes the UNTIL structure, as shown in Figure 4.14. The process starts by initializing a counter. The X register will be used as the counter because it can hold a large count (up to 65,535), and the HC11 provides a decrement X instruction for the process of the loop. After the count has been initialized, the count is decremented once before the condition is checked. The desired time has expired when the count is equal to zero. If the count is not zero, the program loops back up to the decrement instruction to perform the process again. When the counter equals zero, the BNE instruction fails the branch test and the process drops out of the loop.

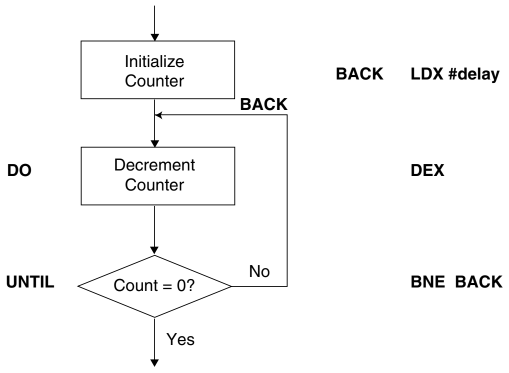


Figure 4.14 Time Delay Using UNTIL Structure

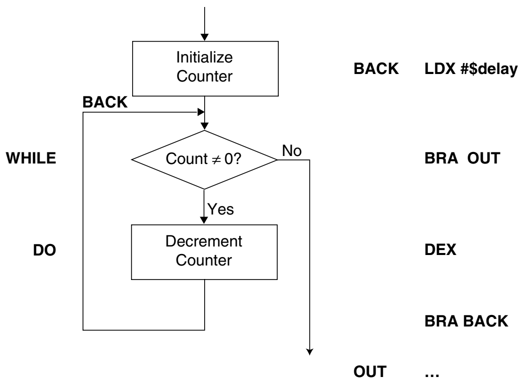


Figure 4.15 Time Delay Using WHILE Structure

The value loaded into the X register during the initialization of the count determines how long this function waits.

The time delay can be accomplished with the WHILE structure, as shown in Figure 4.15. The process starts by initializing a counter. After the count has been initialized, the condition is checked. If the count is zero, the program branches out of the while loop and the process is complete; otherwise, it drops into the decrement process. After the count is decremented, the unconditional branch causes the program to loop back to perform the test again. This repeats until the count is decremented to zero.

Timing Loops

Since each instruction requires a fixed number of machine cycles to perform the fetch and execution of the instruction, programs can be written to cause things to happen at specific points in time. There are two instructions that can be used to help cause specific timing conditions: NOP and BRN. These instructions are summarized in Figure 4.16.

The NOP instruction stands for “No operation.” It takes two machine cycles to do nothing. It is primarily for timing control within a program. It can also be a valuable tool during software debugging. An instruction within a program can be temporarily disabled by replacing it with an NOP. The BRN instruction stands for “Branch Never.” It takes three machine cycles to perform a branch test that always fails and then falls through to the next instruction, in essence doing nothing. It is primarily used during the troubleshooting of a program to temporarily disable another branch instruction.

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
NOP	No operation	Fetch opcode and do nothing	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
BRN	Branch never	Perform branch test and fall through	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-

Figure 4.16 Time Delay Instructions

Because it requires three machine cycles, it can be used in conjunction with the NOP to create and odd or even number of machine cycles within a timing loop.

The NOP and BRN instructions can be used anywhere in a program to just wait two or three machine cycles. They have no other effect on the processes. If each machine cycle is 500 ns, the NOP would create a time delay of 1 µs and the BRN would create a time delay of 1.5 µs. This may not seem like a very long time, but to a computer, it is long enough to wait on a slow memory or I/O device before proceeding with the next operation. Inside a timing loop, like the loop presented in Figure 4.14, one of these time delay instructions can greatly increase the length of the loop. Outside the loop, they could require an extra two or three machine cycles to make the time delay more precise.

Self-Test Questions 4.6

1. What is a finite loop?
2. What is the difference between a WHILE loop and an UNTIL loop?
3. Why does the UNTIL structure use fewer lines of code to implement?
4. What is one function of the NOP instruction?

Summary

This chapter focused on the instructions that have the job of controlling or altering the flow of a program. The JMP and BRA instructions are used to unconditionally change the flow of the program. There also is a large group of conditional branch instructions that perform a branch test before the branch takes place. If the test fails, the program will not branch. It falls through to the instruction following the branch instruction. The jump instruction uses an absolute 16-bit address to designate the destination address of the jump. Any address can be designated as the destination address of the jump instruction. Branch instructions use a relative address to designate the destination address. The final destination must be calculated from the PC and the relative address. Since the relative address is only an 8-bit address, the destination address is limited to a small range around the current PC (128 bytes backwards, 127 bytes forward).

Because of the ability of conditional branch instructions to perform branch tests, they can be used in various conditional-programming structures. The IF-THEN-ELSE structure allows a process to occur only IF a condition is true. The WHILE structure allows a process only while some condition is true, and the UNTIL structure allows the process to continue until some condition is true.

Chapter Questions

Section 4.1

1. What is the destination address of a jump instruction?
2. How are destination addresses usually designated in the code so that the programmer does not have to calculate the exact address?

Section 4.2

3. What effect does a branch instruction have on the program counter?
4. Are relative addresses signed or unsigned values?
5. What is the largest number of addresses that a program can branch forward?
6. What is the largest number of addresses that a program can branch backward?
7. Which address is used as the reference before the branch is calculated?
8. If a BRA instruction is located at $0134, what is the maximum forward destination address?
9. If a BRA instruction is located at $01CD, what is the maximum backward destination address?

Section 4.3

10. Which CCR flags are used by the conditional branch instructions for the branch tests?
11. What is the conditional branch test for the BLT instruction?
12. How is it determined if the branch test in question 11 passes or fails?
13. Repeat question 12 for the BHI instruction.

Section 4.4

14. What do compare instructions actually do?
15. What is the difference between a subtraction and a compare instruction?
16. On the HC11, is there any other way to compare two numbers, besides subtracting them?

Section 4.5

17. Why is a conditional branch instruction required for an IF-THEN-ELSE structure?
18. Could an IF-THEN-ELSE structure be written that tests to see if something is "not true"?

Section 4.6

19. What is the functional difference between a WHILE and an UNTIL loop?
20. Explain the idea behind a finite loop.
21. Which finite loop is easier to code, WHILE or UNTIL? Why?
22. What combination of instructions could be used to cause a delay of five machine cycles?

Chapter Problems

1. If the "N" flag bit in the CCR is a "1," what is the location of the next instruction that would be executed?

```
Address   Object Code
015A      2A 1E
```

2. Use the instruction from Problem #1. If the "N" flag bit in the CCR is a "0," what is the location of the next instruction that would be executed?

3. What is the conditional test for this branch instruction?

```
Address   Object Code
0148      2B E8
```

4. Use the instruction from Problem #3. If the "N" flag bit in the CCR is a "1," what is the location of the next instruction that would be executed?
5. Use the instruction from Problem #3. If the "N" flag bit in the CCR is a "0," what is the location of the next instruction that would be executed?
6. Determine the state of the N, Z, V and C bits in the CCR after the CMPA instruction is executed for each of the following:

```
a. LDAA  #$A0    b. LDAA  #$E0   c. LDAA  #$0A   d. LDAA  #$90
   CMPA  #$A0       CMPA  #$A0      CMPA  #$A0      CMPA  #$A0
```

7. Draw a flowchart and write a program that will load the X register with $8888 if the value in AccA = $88.
8. Draw a flowchart and write a program that will perform an IDIV if the value in AccD is greater than the value in the X register, else it will perform an FDIV.
9. Draw a flowchart and write a program that will perform a loop 100 times. Use AccB as the loop counter. Use AccA as an accumulator and inside the loop add $02 to the previous value. Initialize the accumulator to zero. Use the UNTIL method.
10. Redo problem #9, using the WHILE method.

Answers to Self-Test Questions

Section 4.1

1. A label is a set of characters that the programmer makes up to designate the destination address for a jump instruction. It is used following the JMP mnemonic as well as to the left of the destination instruction in the source code.
2. All jump instructions use a 16-bit absolute address to designate the destination address.

Section 4.2

1. All branch instructions use an 8-bit relative address. The destination address must be calculated.
2.

```
  $0023 current program counter
 +$0002 BRA is a two-byte instruction
 +$0062 sign-extended relative address
 ------
  $0087 destination address
```

3.

```
  $B61D current program counter
 +$0002 BRA is a two-byte instruction
 +$FF9A sign-extended relative address
 ------
  $B5B9 destination address
```

4. DA = 01DD → $\texttt{\$01DD}$
   PC = \$0193 + 2 → $+\underline{\texttt{\$0195}}$
   rr = DA – PC → $\texttt{\$00}\underline{\texttt{48}}$ `rr = $48`
5. DA = E25A → $+\underline{\texttt{\$E25A}}$
   PC = \$E273 + 2 → $\texttt{\$E275}$
   rr = DA – PC → $\texttt{\$FF}\underline{\texttt{E5}}$ `rr = $E5`

Section 4.3

1. CCR = \$D3, thus N = 0, Z = 0, V = 1, C = 1.
   BPL test checks if N = 0. Since N = 0, the test passes and DA = PC + rr.
   PC = \$0181 + 2 → $\texttt{\$0183}$
   rr = \$2C, sign extend → $+\underline{\texttt{\$002C}}$
   DA = PC + rr → $\texttt{\$01AF}$
2. CCR = \$DC, thus N = 1, Z = 1, V = 0, C = 0.
   BNE test checks if Z = 0. Since Z = 1, the test fails and DA = PC.
   DA = PC = \$01EE + 2 → `$01F0`
3. Since the sum of \$32 + \$CE equals zero, then the Z flag in the CCR is set. Because the BEQ test checks if Z = 1, then the test passes and DA = PC + rr.
   PC = \$017B + 2 → $\texttt{\$017D}$
   rr = \$A3, sign extend → $+\underline{\texttt{\$FFA3}}$
   DA = PC + rr → $\texttt{\$0120}$

Section 4.4

1. Compare instructions perform operations that set up the conditions codes so that the branch tests can be properly performed.
2. CMPA compares the value in AccA to any other 8-bit value. TSTA compares the value in AccA to zero.

Section 4.5

1. Conditional branch instructions are required for IF-THEN-ELSE because they perform a branch test. IF the test passes, THEN the branch will occur, ELSE the process will fall through to the next instruction.
2. The decision symbol (diamond) is used to show the IF-THEN-ELSE structure.
3. The ELSE path is followed if the branch test fails.

Section 4.6

1. A set of instructions that terminates when a conditional test fails.
2. The WHILE condition is tested at the start of the loop before the loop task is executed. The UNTIL condition is tested at the end of the loop after the loop task is executed.
3. The UNTIL needs only one conditional branch instruction. The WHILE structure requires a conditional branch and an unconditional branch.
4. The NOP instruction can be used as a two-cycle time delay.

c h a p t e r 5

Indexing Through Memory

Objectives

After completing this chapter, you should be able to:

- Calculate the effective address for an indexed mode instruction
- Perform summation on a list of numbers
- Find items in a list using indexed mode to index through the list
- Count items in a list using indexed mode to index through the list
- Accomplish memory-to-memory copy of data

Outline

Introduction

Indexed addressing was introduced in chapter 2, but only in the context of the other addressing modes. Chapter 3 provided a more in-depth look at the application of the inherent, immediate, direct and extended addressing modes. Chapter 4 focused on the application of the relative addressing mode. This chapter provides further explanation of how the indexed addressing mode works. Several examples of small programs are provided to illustrate the advantages of the indexed mode to access memory.

The indexed addressing mode provides a means of accessing data that is stored in banks, lists, look-up tables or any other sequential method of storage. Because it uses a base address as a reference point or starting point, only the offset is necessary to sequence through the memory one byte at a time. By combining the instructions that increment or decrement the index registers with some looping, small blocks of code can perform much larger data access functions.

For example, when someone is given directions to a store, these directions could come in several forms. The actual address could be given, such as "2149 W. Dunlap Avenue." The actual address is analogous to the method used in extended addressing mode. The whole or absolute address is given as part of the instruction. The directions could also be provided in reference to a known location, such as "third building west of 19th Avenue and Dunlap." This method is analogous to the way indexed addressing mode works. The index register contains an address that is the common known location, and the instruction provides an offset that tells the processor how far the effective address is from the reference. The address in the index register is like saying "19th Avenue and Dunlap." It establishes the neighborhood, and the offset tells the actual location to which to go. The principles of how this works are provided in this chapter.

This chapter directly correlates to section 6.2.4 of the HC11 Reference Manual and Section 3.4.4 of the Technical Data Manual.

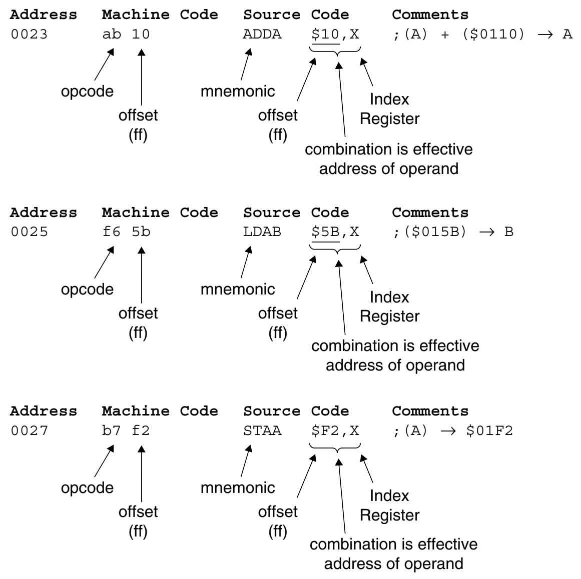
5.1 Using Indexed Addressing

The indexed addressing mode allows stepping through the memory space from a common base address. The common base address or reference base address is contained in one of the 16-bit index registers. The instructions use an 8-bit unsigned offset to index forward into memory from the base address. The effective address of the operand is always the sum of the address contained in the designated index register and the offset. Figure 5.1 provides three examples of how the indexed mode instructions look. The operand field contains the offset.

In the indexed mode, an offset follows the opcode in memory. The offset indicates the number of bytes to index forward in memory to the effective address. It is an offset from the current base address stored in one of the index registers. Since it is an unsigned 8-bit number, it is added to the contents of the 16-bit register by zero filling the 8-bit word out to 16 bits. Zero filling is the process of filling all bit positions with zeros that are not occupied by data. The result of this operation produces the effective address, as shown in Equation 5.1. The index register must be loaded with the base address only once. This is usually done at the start of the program.

```
EA = X + ff                                             Equation 5.1
EA = Effective Address
X = Address in the X index register (or Y index register)
ff = Offset
```

Assume X Register contains $0100 for each of the following examples.



```
Address   Machine Code   Source Code   Comments
0023      ab 10          ADDA  $10,X   ;(A) + ($0110) → A
```

```
Address   Machine Code   Source Code   Comments
0025      f6 5b          LDAB  $5B,X   ;($015B) → B
```

```
Address   Machine Code   Source Code   Comments
0027      b7 f2          STAA  $F2,X   ;(A) → $01F2
```

Figure 5.1 Examples of Indexed Mode Instructions

NOTE: The effective address of an indexed mode instruction is the sum of the base address from either index register (X or Y) and an unsigned 8-bit offset (ff) supplied by the instruction.

Figure 5.2 contains an example similar to the program found in Example 3.8, except that the extended-mode instructions have been replaced with indexed-mode

```
0140                ORG   $0140
0140 ce 01 80       LDX   #$0180    ;Set base address
0143 a6 00          LDAA  $00,X     ;Get 1st number (Base+0)
0145 ab 01          ADDA  $01,X     ;Add 2nd (Base+1) to 1st
0147 a7 02          STAA  $02,X     ;Store sum in Base+2
0149 3f             SWI
```

a) Summing two numbers using Index mode.

```
0180         BASE   EQU   $0180     ;Base address

0140                ORG   $0140
0140 ce 01 80       LDX   #BASE     ;Set base address
0143 a6 00          LDAA  $00,X     ;Get 1st number (Base+0)
0145 ab 01          ADDA  $01,X     ;Add 2nd (Base+1) to 1st
0147 a7 02          STAA  $02,X     ;Store sum in Base+2
0149 3f             SWI
```

b) Summing two numbers using Index mode. Base address is designated using a label.

```
BASE   $0180   $68   1st number
       $0181   $19   2nd number
       $0182   $81   sum of 1st + 2nd
```

c) Memory map of locations accessed by the sum program.

Figure 5.2 Addition Program Using Indexed Instructions

Operation	Description	Functional Result	
Fetch LDAA opcode	PC is moved to MAR.	PC → MAR	$0140 → MAR
(1 machine cycle)	PC is incremented.	PC + 1 → PC	$0141 → PC
	Opcode is read from memory.	Opcode → IR	$A6 → IR
Execute LDAA	PC is moved to MAR.	PC → MAR	$0141 → MAR
(3 machine cycles)	PC is incremented.	PC + 1 → PC	$0142 → PC
	Offset is read from memory.	rr → Temp	$00 → Temp
	Calculate effective address.	IX + ff → MAR	$0180 + $0000 = $0180 → MAR
	Read operand from effective address.	1st number → A	$62 → A
	Update status in the CCR.	N → 0, Z = 0, V = 0 = CCR	
Fetch ADDA	PC is moved to MAR.	PC → MAR	$0142 → MAR
opcode	PC is incremented.	PC + 1 → PC	$0143 → PC
(1 machine cycle)	Opcode is read from memory.	Opcode → IR	$AB → IR
Execute ADDA	PC is moved to MAR.	PC → MAR	$0143 → MAR
(3 machine cycles)	PC is incremented.	PC + 1 → PC	$0144 → PC
	Offset is read from memory.	rr → Temp	$01 → Temp
	Calculate effective address.	IX + ff → MAR	$0180 + $0001 = $0181 → MAR
	Read operand from effective address.	2nd number → Temp	
	Add two numbers.		$19 → Temp
	Update status in the CCR.	A + Temp → A	$68 + $19 = $81 → A
		H = 1, N = 1, Z = 0, V = 1, C = 0 → CCR	
Fetch STAA opcode	PC is moved to MAR.	PC → MAR	$0144 → MAR
(1 machine cycle)	PC is incremented.	PC + 1 → PC	$0145 → PC
	Opcode is read from memory.	Opcode → IR	$A7 → IR
Execute STAA	PC is moved to MAR.	PC → MAR	$0145 → MAR
(3 machine cycles)	PC is incremented.	PC + 1 → PC	$0146 → PC
	Offset is read from memory.	rr → Temp	$02 → Temp
	Calculate effective address.	IX + ff → MAR	$0180 + $0002 → $0182 → MAR
	Store operand in effective address.	A → EA	$81 → M($0182)
	Update status in the CCR.	N = 1, Z = 0, V = 0 → CCR	

Figure 5.3 Process of Executing the Index Mode Instructions

instructions. Instead of the absolute 16-bit address following each instruction, an offset is provided following each instruction and a reference to the X register. This offset must be added to the base address in the X register to calculate the effective address for each instruction. This requires that the index X register be loaded with the base address (initialized) before the other instructions can be run.

In Figure 5.2a, the base address is loaded in the index X register. The actual value of the base address ($0180) is shown in the code. This base address is then used by the next three instructions to access three memory locations. Figure 5.3 summarizes the sequence of events that occur when the LDAA, ADDA and STAA instructions are executed. Additional examples of how the indexed addressing works are shown in Example 5.1.

Example 5.1

Problem: Calculate the effective address (EA) for each of the following instructions. Assume the X register contains $0012 and the Y register contains $1000 for each instruction.

	Address	Machine Code	Source Code
a.	0109	E6 E8	LDAB $E8,X
b.	0100	AA 00	ORAA $00,X
c.	01E2	18 6F 04	CLR $04,Y
d.	E234	18 A7 04	STAA $30,Y

Solution: In each case, the effective address (M) must be determined. Then the offset must be zero filled to 16 bits and added to the base address in the index register to calculate the effective address (Equation 5.1).

```
a.   X  = $0012            →   $0012
     ff = $E8, zero filled →  +$00E8
     EA = X + ff           →   $0100
b.   X  = $0012            →   $0012
     ff = $00, zero filled →  +$0000
     EA = X + ff           →   $0012
c.   Y  = $1000            →   $1000
     ff = $04, zero filled →  +$0004
     EA = X + ff           →   $1004
d.   Y  = $1000            →   $1000
     ff = $30, zero filled →  +$0030
     EA = X + ff           →   $1030
```

As has been shown in chapters 2 and 3, the HC11 supports 16-bit data operations. Remember that the first of the two effective addresses is called M. The next location in memory is simply M + 1. Example 5.2 shows how the M and M + 1 are calculated when two bytes must be read from memory or written to two consecutive memory locations by a single instruction.

Example 5.2

Problem: Calculate the effective address (EA) for each of the following instructions that require a 2-byte memory access. Assume the X register contains $0140 and the Y register contains $B600 for each instruction.

	Address	Machine Code	Source Code
a.	002F	AE 02	LDS $02,X
b.	0106	18 E3 00	ADDD $00,Y
c.	B723	ED 20	STD $A0,X

Solution: In each case, the two effective addresses (M and M + 1) must be determined. Then the offset must be zero filled to 16 bits and added to the base address in the index register to calculate the effective address (Equation 5.1).

a. This instruction must copy two bytes of data from memory to the SP. Therefore, two effective addresses must be calculated (M and M + 1)

```
X = $0140                 →   $0140
ff = $02, zero filled     →  +$0002
EA = X + ff               →   $0142  M
Next address is M + 1     →  +$0001
                          →   $0143  M + 1
```

b. This instruction must copy two bytes of data from memory and add them to the contents of AccD. Therefore, two effective addresses must be calculated (M and M + 1).

```
Y = $B600                 →   $B600
ff = $00, zero filled     →  +$0000
EA = X + ff               →   $B600  M
Next address is M + 1     →  +$0001
                          →   $B601  M + 1
```

c. This instruction must copy two bytes of data from the D register to memory. Therefore, two effective addresses must be calculated (M and M + 1).

```
X = $0140                 →   $0140
ff = $A0, zero filled     →  +$00A0
EA = X + ff               →   $01E0  M
Next address is M + 1     →  +$0001
                          →   $01E1  M + 1
```

Self-Test Questions 5.1

1. What kind of offset is used by all indexed-mode instructions?
2. If the address X register is $0023 and the offset = $62, what is the effective address?
3. Is there ever a situation where an instruction needs more than one effective address? If so, when?

5.2 Summing a List of Numbers

Numbers are often organized in columns and rows. Sometimes the data needs to be summed to calculate a total. The indexed method of addressing is ideal for implementing this summation. The first location in the list is set as the base address. Each subsequent instruction uses indexed-mode instructions to access the proper data to complete the summation.

Summing a List: UNTIL Method

An example of a summation program is shown in Figures 5.4 and 5.5. The objective of this program is to sum 16 numbers that are stored in sequential memory locations. The sum that is calculated will then be stored at the next memory location immediately following the list. The flowchart and code use the UNTIL method to accomplish this summation.

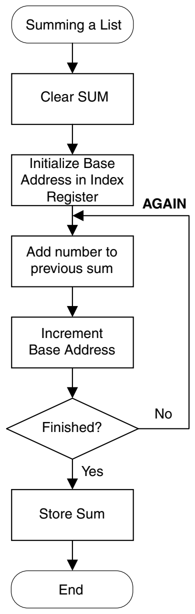


Figure 5.4 Flowchart to Sum a List of Numbers: UNTIL Method

NOTE: The source code listing for this example and all subsequent examples in the text will include sequential line numbers as the first column of the file. These line numbers are present in all listing files produced by an assembler. They are simply used as unique identifiers for lines of code in the file.

The program starts by clearing the sum. The sum must start at zero, or the first addition would cause an erroneous result. Each number that is retrieved from memory will be added to the previous total already in AccA. The second step is to initialize the base address or the reference. This program uses the X register to hold the base address. It is loaded initially with $0000, which is the address of the first number to be summed. This completes the steps necessary to initialize the program before it begins the actual summation.

The summation starts by retrieving the first number from memory and adding it to the initial sum. The next step is to increment the base address and check to see if the

```
0001                    *Program to sum a list of numbers
0002                    *using index mode - UNTIL method
0003
0004 0100 4f                    CLRA            ;Clear SUM
0005 0101 ce 00 00              LDX     #$0000  ;Initialize base address
0006 0104 ab 00         AGAIN   ADDA    $00,X   ;Add number to previous sum
0007 0106 08                    INX             ;Increment base address
0008 0107 8c 00 10              CPX     #$0010  ;Finished?
0009 010a 26 f8                 BNE     AGAIN   ;If NOT, do again!
0010 010c a7 00                 STAA    $00,X   ;Store SUM following list
0011 010e 3f                    SWI
```

Figure 5.5 Code to Sum a List of Numbers: UNTIL Method

summation is finished. The ADDA instruction will always offset $00 bytes from the base address. This offset is fixed; however, as the base address can change in the X register, this instruction can access a different memory location each time it is executed in the loop.

The program must determine if the summation is finished. Since this program is summing 16 values, it can check to see if the base address has moved 16 locations through memory. The program uses a CPX instruction to compare the value in the X register to the address following the list that is being summed. It checks against this next address because the base pointer is incremented prior to the test. Thus, it always points to the next location to be summed, rather than the last location that was summed. When the value in the X register is less than the address ($0010), the Z flag will be cleared, indicating that the two numbers are not equal. Now the program is ready to perform the conditional branch test.

The conditional BNE instruction on line 9 is used to determine if another pass through the loop is needed. If the base address in the X register is still within the desired range (i.e., less than $0010), the program loops back and continues the summation. If the base address has advanced past the end of the list of numbers to be summed, the program does not continue to loop. The last time through the loop, the base address is equal to the address of the last value to be summed. It is then incremented by the INX instruction so that the base address is one greater than the last address of the list. When the CPX instruction is executed, the base address in the X register will be equal to the address ($0010). The subtraction of the two values by the CPX instruction will result in zero, and the Z flag in the CCR will be set, indicating an equal condition. The branch test for the BNE is "?Z = 0," which will fail, and the program will fall through to the next instruction. At this point, the sum is stored in the next memory location and the program is complete.

NOTE: The program is terminated on the HC11 EVBU with an SWI instruction. The SWI instruction provides an output to the monitor indicating the execution has been completed. The SWI instruction is described in chapter 10.

Summing a List: WHILE Method

The previous summation example is repeated in this section, except that the WHILE method is used instead of the UNTIL method. The objective of the program is still to sum a list of 16 numbers that are stored in sequential memory locations. The sum that is calculated will then be stored at the next memory location immediately following the list. Figures 5.6 and 5.7 contain the flowchart and the program code for this method.

The program starts by clearing the sum and initializing the base address in the same manner as the UNTIL version. AccA will be used for the sum, and the X register will be used for the base address. Consistent with the WHILE method, the first step of the summation in the loop is to perform a test (lines 6 and 7 of file). Remember, the WHILE loop will only be executed while the condition is true. Since this program is summing 16 values, it checks to see if the base address has moved 16 locations through

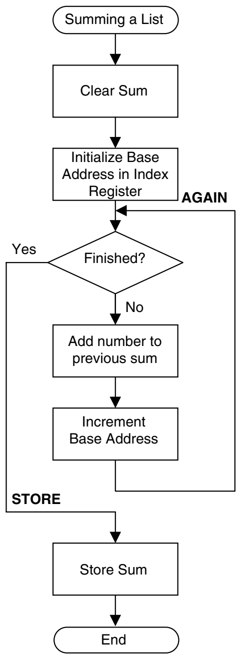


Figure 5.6 Flowchart to Sum a List of Numbers: WHILE Method

memory. The program uses a CPX instruction to compare the value in the X register to the address following the list ($0010). It checks against this next address because the base pointer is incremented each time through the loop. Thus, it always points to the next location to be summed, rather than the last location that was summed.

The summation starts by retrieving the first number from memory and adding it to the initial sum. Then it increments the base address and loops back to the top of the WHILE loop. The last time through the loop, the base address is equal to the address of the last value to be summed. It is then incremented by the INX instruction so that it is equal to the address following the list. When the WHILE loop branches back and executes the CPX instruction, the base address in the X register will be equal to the address ($0010). The subtraction of the two values will result in zero, and the Z flag in

```
                        *Program to sum a list of numbers
                        *using index mode - WHILE method

0120 4f                         CLRA            ;Clear SUM
0121 ce 00 00                   LDX     #$0000  ;Initialize base address
0124 8c 00 10           AGAIN   CPX     #$0010  ;Finished?
0127 27 04                      BEQ     STORE   ;If so, store result
0129 ab 00                      ADDA    $00,X   ;Else, Add number to sum
012b 08                         INX             ;Increment base address
012c 20 f7                      BRA     AGAIN   ;do again!
012e a7 00              STORE   STAA    $00,X   ;Store SUM following list
0130 3f                         SWI
```

Figure 5.7 Code to Sum a List of Numbers: WHILE Method

the CCR will be set. The branch test for the BEQ is "?Z = 1," which will pass, and the program will branch to the STORE section of the code. At this point, the sum is stored in the next memory location and the program is complete.

Self-Test Questions 5.2

1. Why is the base address incremented in the summation example, and not the offset value?
2. How many times is the conditional branch instruction executed in the UNTIL example? In the WHILE example?

5.3 Finding the Largest Number

In many applications, there is a need to sort the values in a list. Although there are very sophisticated sorting programs, often it is sufficient just to walk through the list and find the largest number. The process could then be repeated to find the next largest until the entire list is sorted. The objective of this program is to examine a list of 20 numbers and simply find the largest unsigned number, as shown in Figures 5.8

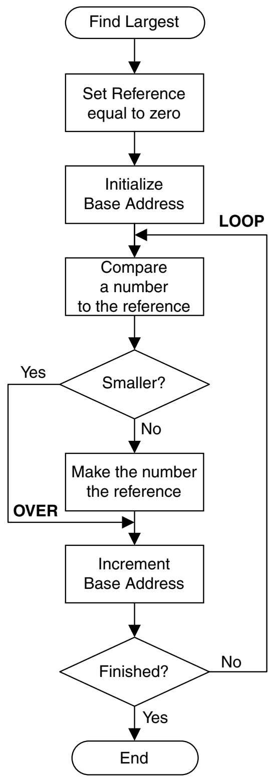


Figure 5.8 Flowchart to Find the Largest Number in a List

```
0001                    *Program to find largest number in a list
0002                    *using index mode
0003
0004 0140 4f                    CLRA            ;Clear Reference
0005 0141 ce 01 d4              LDX     #$01D4  ;Initialize Base Address
0006 0144 a1 00         LOOP    CMPA    $00,X   ;Compare number to reference
0007 0146 25 02                 BLO     OVER    ;If lower, don't replace!
0008 0148 a6 00                 LDAA    $00,X   ;Else, Replace the reference
0009 014a 08            OVER    INX             ;Point to next item in table
0010 014b 8c 01 e8              CPX     #$01E8  ;Is list finished?
0011 014e 26 f4                 BNE     LOOP    ;If NOT, do loop again
0012 0150 3f                    SWI
```

Figure 5.9 Code to Find the Largest Number in a List

and 5.9. The largest number will then be stored at the next memory location immediately following the list. Sorting an entire list will be left as a problem for the end of the chapter.

The program starts by setting a reference value to zero. In order for the comparison to work properly, the reference is set to the lowest value ($00). Each number stored in the list will be compared to the reference in order. If the value is smaller than the reference, the loop continues to the next value in the list. Otherwise, the reference is replaced by the value. This process will keep the largest value as the reference, so that when each value in the list has been examined, the final reference is the largest value from the list.

As in all programs, this one starts by initializing resources needed in the program. The reference is set to zero, and the base address is loaded into the X register. Note that the address of the first number in the list is $01D4. This address is used to illustrate that the base address can be any valid address. The search for the largest number starts by retrieving the first number from the list and comparing it to the reference. This is accomplished with a single compare instruction using indexed addressing in line 6 of the code. When the compare is complete, the flags in the CCR are updated to reflect the relationship of the number to the reference. The next step is to perform the branch test that determines if the number is smaller than the reference. The BLO instruction is used in line 7 to perform this test because it performs an unsigned test and branches if the number is lower. If the number is not lower than the reference, the branch test fails and the program falls through to the next instruction, where the reference is replaced with the number.

The program must determine if the search is finished. Since this program is comparing to 20 numbers, it can check to see if the base address has moved 20 locations through memory. The program increments the base address and uses a CPX instruction to compare the base address to the address that is 20 locations higher than the initial base address. If the current base address is not yet equal to the final address ($01E8), then the program loops and performs the next compare. The last time through the loop the base address is incremented to $01E8, the BNE test fails, and the program is finished.

Self-Test Questions 5.3

1. In addition to the WHILE loop, what programming structure is used by this program?
2. How many times will the LDX instruction in line 5 be executed?
3. Why is the CLRA instruction in line 4 necessary?

5.4 Counting the Number of Positive Numbers

Another common programming task is counting. Counters are used to keep track of the number of things that have been processed and of how much time has expired and to perform a variety of other tasks. This programming example counts the number of positive numbers in a list. The flow of the program is very similar to that of the previous program that found the largest number in a list, as shown in Figures 5.10 and 5.11.

The program starts by clearing the count and initializing the base address. Each number is loaded into AccB. The process of loading updates the condition code flags to indicate the sign of the number, so that the test can be immediately executed following the load. If the value is positive, the counter is incremented; otherwise, the

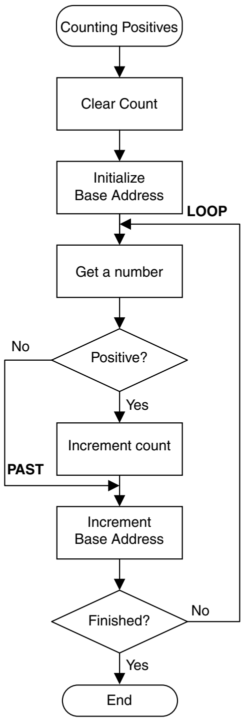


Figure 5.10 Flowchart to Count Positive Numbers in a List

```
0001                    *Program to count the positive numbers
0002                    *in a list using index mode
0003
0004 0160 4f                    CLRA            ;Clear COUNT
0005 0161 ce 01 80              LDX     #$0180  ;Initialize the base address
0006 0164 e6 00         LOOP    LDAB    $00,X   ;Get a number
0007 0166 2b 01                 BMI     PAST    ;If negative, don't count!
0008 0168 4c                    INCA            ;Else, Count the positive
0009 0169 08            PAST    INX             ;Increment base address
0010 016a 8c 01 a0              CPX     #$01A0  ;Is list finished?
0011 016d 26 f5                 BNE     LOOP    ;If not finished, LOOP
0012 016f 3f                    SWI
```

Figure 5.11 Code to Count Positive Numbers in a List

counting is skipped. When the program ends, the number of positive numbers from the list will be in the counter (AccA).

After each number is loaded into AccB, the BMI instruction is used to perform the branch test. If the number loaded in line 6 is negative, the N flag will be set; otherwise, it will be cleared. The branch test performed by BMI is "?N = 1." If this test fails, the number is positive, and the program falls through to the next instruction, where the counter is incremented. This process continues until the end of the list is reached.

Self-Test Questions 5.4

1. How many bytes does this program occupy in memory?
2. What address value is assigned to the label "LOOP"?
3. How many numbers are in the list?

5.5 Copying a Block of Memory

One of the most prevalent activities on a computer is the need to copy data from one location to another (i.e., memory to the processor, processor to I/O, etc.). Typically, the data consists of more than one byte. In chapter 3, the function of a set of data movement instructions was presented. Three types of movement are directly supported by the HC11: memory to processor register, processor register to memory and processor register to processor register. The program in this section will demonstrate a method of copying a block (one or more bytes) from a range of memory locations to another range of memory locations; thus it is a memory-to-memory movement. The data movement instructions, load and store, will be utilized with the indexed mode of addressing to methodically copy the block. A flowchart for this program is shown in Figure 5.12, and the source code listing to implement the program is shown in Figure 5.13.

The program starts by initializing the source and destination addresses. The X register holds the source address, and the Y register holds the destination address. The address of the first location of the source is $0000. The address of the first location of the destination is $0180. The numbers are read from each source location and written to

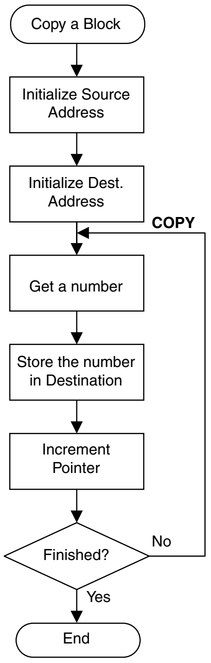


Figure 5.12 Flowchart to Copy a Block of Numbers

each destination location one-by-one, until the entire block is copied. The load instruction in line 6 reads the source data from memory and temporarily holds it in AccA. Then the store instruction in line 7 writes this data from AccA to the destination address. Since the source and destination locations are changing, both index registers must be incremented each time through the loop. The program determines if the copy operation is complete by comparing the destination address to the first address beyond the destination range. The CPY instruction in line 10 accomplishes the check in preparation for the branch test in line 11. As has been shown before, if the destination is not equal to the address beyond the end of the range, then the process loops until the copy operation is complete.

```
0001                    *Program to copy a list of numbers
0002                    *using index mode from $0000 - $0009 to $0180 - $0189
0003
0004 0100 ce 00 00              LDX     #$0000  ;Init source address
0005 0103 18 ce 01 80           LDY     #$0180  ;Init destination address
0006 0107 a6 00         COPY    LDAA    $00,X   ;Get a number from source
0007 0109 18 a7 00              STAA    $00,Y   ;Store the number in dest
0008 010c 08                    INX             ;Increment pointers
0009 010d 18 08                 INY
0010 010f 18 8c 01 8a           CPY     #$018A  ;Finished?
0011 0113 26 f2                 BNE     COPY    ;If NOT, do again!
0012 0115 3f                    SWI
```

Figure 5.13 Code to Copy a Block of Numbers

Self-Test Questions 5.5

1. Since two index registers are being used, what is the maximum distance between the source and destination ranges?
2. Why are there four bytes of machine code listed for the CPY instruction on line 10 of Figure 5.13?
3. If the program ends when the destination address is incremented to $018A, what is the last destination address that receives data?

Summary

The indexed addressing mode is a method of addressing memory that uses a 16-bit base address in conjunction with an 8-bit unsigned offset. The advantage of the instructions that use this mode is that they allow a program to loop through memory, operating on different bytes in a list each time through the loop. Five examples were presented in this chapter: two methods of summing a list of numbers, a means of finding the largest number in a list, the means of counting the number of positive numbers in a list and a method of copying a block of data from one memory area to another. Each of these examples used one or both of the index registers as a base address to reference the data in the list or block.

Chapter Questions

1. What is the effective address of the following instruction if the X register contains $0120: LDAB $20,X?
2. What is the effective address of the following instruction if the Y register contains $B629: LDAB $D7,Y?
3. What is the effective address of the following instruction if the X register contains $0005: CPX $01,Y?
4. Using the data in Figure 5.14, express the results of the following instructions, including status of the condition code flags. Which addressing mode is used for each instruction? What is the opcode (including prebytes) of each instruction? NOTE: Each instruction is independent. The results are not cumulative.

a.	`LDAA`	$F9,Y
b.	`STX`	$01,X
c.	`CLR`	$00,X
d.	`DEX`	
e.	`LDD`	$1A,X
f.	`TSX`	
g.	`STS`	$01,X
h.	`CMPB`	$07,X
i.	`TST`	$E2,Y
j.	`LSR`	$FB,Y

```
Registers  C =  $D0       Memory  00F8  00 11 22 33 44 55 66 77
           D =  $B600             0100  F3 56 E3 DB A1 A0 09 00
           X =  $0100             0108  22 44 52 88 63 77 74 33
           Y =  $0016             0110  FF 00 FF 11 FF 22 FF 83
           S =  $01FF             0118  F1 3B BB B6 D4 AD CE 00
```

Figure 5.14 Default Data Set #3

5. What is the cost in instructions of using the WHILE loop instead of the UNTIL loop from Figures 5.5 and 5.7?
6. If the block copy program from section 5.5 used a single index register as a base address, what would be the maximum distance of the copy?

Chapter Problems

1. Modify the UNTIL method summation program from Figure 5.5 to do BCD arithmetic instead of hex arithmetic.
2. Draw a flowchart of a program that will count the number of items equal to zero in a list of ten items starting at location $0000.
3. Write a program from the flowchart in problem 2. Assume the list starts at $B600 and ends at $B63F. Be sure to use the appropriate compare instruction.
4. Write a program that will count the negative and positive numbers in a list. Assume the list of numbers starts at $0100 and is 40 bytes long.
5. Given two sequential lists of ten numbers, write a program that will add the numbers in pairs and store the result of each pair in a third list. The first number will be added to the first number of the second list, and the result will be stored in the first position of the third list. The lists start at the following memory locations: $0000, $000A, and $0014.
6. Rewrite the program from Figure 5.13 to copy a block of 24 bytes from $0100–$0117 to $0168–$017F. Use only one index register to establish the base address and use different offsets to designate the source and destination locations used by the load and store instructions. Assemble the code to run starting from location $0000.
7. Rewrite the program from Figure 5.13 to perform the same function presented in problem 6, except use AccD for the temporary storage location in the processor. Modify the flow of the program to accommodate this double-byte flow of data. The end result of the program should be the same as for the program in problem 6. Which approach is a more efficient use of memory? Which approach is a more efficient use of time (fewer machine cycles)?

Answers to Self-Test Questions

Section 5.1

1. All indexed mode instructions use an unsigned 8-bit offset.
2.
```
  $0023 base address
+ $0062 zero-filled address offset
  $0085 effective address
```

3. All instructions that require two-byte memory access require two effective addresses. The first of the two addresses (M) is calculated using Equation 5.1. The second is simply M + 1.

Section 5.2

1. The offset is fixed in memory, but the HC11 provides instructions to increment the addresses in the index registers.
2. The UNTIL example executes the conditional branch 16 times. The WHILE example must execute it 17 times the way it is written.

Section 5.3

1. An IF-THEN-ELSE structure is used in addition to the WHILE structure.
2. As line 5 is outside the loop, it will be executed only one time.
3. Since the program is finding the largest number, the reference must initially be the lowest number to assure that it cannot be greater than any other number in the list.

Section 5.4

1. The program occupies 16 bytes, \$0160–\$016F.
2. The label LOOP is equal to the address \$0164, where it is defined in line 6. This destination address is used by the assembler to calculate the relative address \$F5 that is used in line 11.
3. The list consists of 32 numbers (\$01A0–\$0180).

Section 5.5

1. Since two index registers are being used, there is no limit on the range except the 64K address boundary of the HC11.
2. Instructions that utilize the Y index register require the prebyte before the opcode. The first byte (\$18) is the prebyte, followed by the opcode (\$8C), followed by the two-byte immediate data jjkk (\$018A).
3. Y = \$0189 on the last pass, so the STAA \$00,Y instruction writes to \$0189 before being incremented to \$018A and falling out of the loop.

chapter 6

Subroutines

Objectives

After completing this chapter, you should be able to:

- Use push and pull instructions to move data to and from the stack
- Organize code into subroutines
- Explain the use of subroutine control instructions: JSR, BSR and RTI
- Compare efficiency trade-offs between in-line code and code that uses subroutines
- Describe how parameters are passed to subroutines
- Use nesting to call subroutines from other subroutines
- Access subroutines in the BUFFALO monitor program

Outline

Introduction

Up to this point all programming examples have used code that is sequential in the file. Chapter 4 presented the concepts behind the use of branching instruction to change the flow of the program. Branching offers the ability to have code that does not have to be executed in the exact sequential order in which it resides in memory. This chapter presents a method of modularizing the code within a program. This method groups the instructions into functional groups, where each group completes a task. Each of the functional groups of instructions is called a subroutine. Functionally, each **subroutine** is a self-contained subprogram.

This chapter is designed to provide information surrounding the use of subroutines. A general presentation is made regarding subroutines, how they function and why they are used. Then a presentation is made on passing data to and from subroutines. A variety of subroutines contained in the BUFFALO monitor program are discussed as well.

Since all subroutines use the memory stack, a deeper understanding of the role of the stack is required. Therefore, this chapter starts with a discussion regarding the function of the memory stack and the use of push and pull instructions.

6.1 Temporary Storage Using a Stack

Often during the execution of a program there is a need to temporarily store some data in memory. Most computer systems provide an area of memory for temporary storage called the **stack**. Functionally, the stack is sequential block of memory configured as a **LIFO** (last-in, first-out). Data is written to the stack in the opposite order to that in which it is retrieved from the stack. Specifically, the last byte written to the stack is on the top of the stack; therefore, it must be the first byte taken off the stack before other data can be accessed. The concept of how this works is illustrated in Figure 6.1.

The stack of plates is a good example of how a stack works on a computer. The first plate placed on the stack ends up on the bottom of the stack. The last plate placed on the stack is on the top. When the plates are removed from the stack, they are removed from the top first; thus the last one on the stack is the first one off. A memory stack is

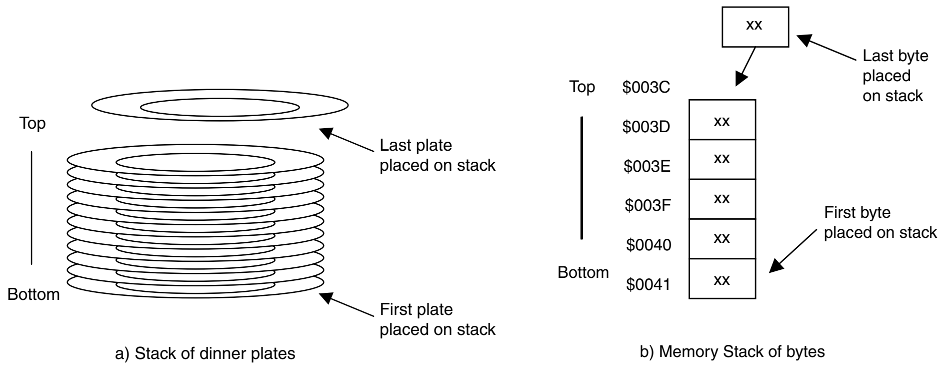

Top
Bottom
Last plate placed on stack
First plate placed on stack
a) Stack of dinner plates
xx
Last byte placed on stack
Top
$003C
$003D
$003E
$003F
$0040
Bottom
$0041
First byte placed on stack
b) Memory Stack of bytes

Figure 6.1 Examples of Stacks

a block of sequential locations in memory. The first location used is the bottom of the stack, in the same way the first plate is on the bottom of the stack. This becomes the highest address location in the stack. The last location, or the top of the stack, will be the lowest address location used by the stack, as shown in Figure 6.1b. In this example, six locations are used. The bottom of the stack is at location \$0041 in memory; and the top of the stack is at location \$003C.

As the data moves to and from the stack, the address of the next available location on the memory stack is changing. After data is placed on the stack, the address is decremented. After data is removed from the stack, the address is incremented. This address is called the **stack pointer,** and the processor keeps it in the stack pointer register. The HC11 uses a 16-bit address as the stack pointer register so that any location within the HC11 memory map can be used as the stack.

The HC11 supports a group of special instructions designed to manage the data on the stack. Push instructions are a special type of data movement instructions that are used to place data on the stack. They copy the data from a processor register and place it on the stack. They are also responsible for decrementing the stack pointer so that it continues to point to the next available location on the memory stack.

The mnemonic form for push instructions is PSHx. Data can be pushed onto the stack from the A, B, X or Y registers using the PSHA, PSHB, PSHX or PSHY instructions, as shown in Figure 6.2. Since the source of the data is included in the mnemonic and the address of the data destination is indicated by the address in the stack pointer register, all push instructions use the inherent addressing mode. Push instructions have no effect on the CCR. Push instructions are complementary to pull instructions.

Pull instructions are a special type of data movement instructions that are used to remove data from the stack. They copy the data from the stack and place it in a processor register. They first increment the stack pointer so that it points to the last byte written to the stack. When the data has been copied to the processor register, the location it occupied on the stack is now available for another byte of data. Since the stack pointer already points to this location, it does not have to be updated at the end of the instruction.

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
PSHA	Push 8 bits from AccA onto Stack, Decrement S	(A) $\Rightarrow M_S$ (S) – 1 $\Rightarrow$ S	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
PSHB	Push 8 bits from AccB onto Stack, Decrement S	(B) $\Rightarrow M_S$ (S) – 1 $\Rightarrow$ S	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
PSHX	Push 16 bits from X onto Stack, Decrement S	$(X_L) \Rightarrow M_S$ (S) – 1 $\Rightarrow$ S $(X_H) \Rightarrow M_S$ (S) – 1 $\Rightarrow$ S	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
PSHY	Push 16 bits from Y onto Stack, Decrement S	$(Y_L) \Rightarrow M_S$ (S) – 1 $\Rightarrow$ S $(Y_H) \Rightarrow M_S$ (S) – 1 $\Rightarrow$ S	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-

Figure 6.2 Push Instructions

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
PULA	Increment S, Pull 8-bits from Stack into AccA	$(S) + 1 \Rightarrow S$ $(M_S) \Rightarrow A$	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
PULB	Increment S, Pull 8-bits from Stack into AccB	$(S) + 1 \Rightarrow S$ $(M_S) \Rightarrow B$	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
PULX	Increment S, Pull 16-bits from Stack into X register	$(S) + 1 \Rightarrow S$ $(M_S) \Rightarrow X_H$ $(S) + 1 \Rightarrow S$ $(M_S) \Rightarrow X_L$	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
PULY	Increment S, Pull 16-bits from Stack into Y register	$(S) + 1 \Rightarrow S$ $(M_S) \Rightarrow Y_H$ $(S) + 1 \Rightarrow S$ $(M_S) \Rightarrow Y_L$	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-

Figure 6.3 Pull Instructions

The mnemonic form for pull instructions is PULx. Data can be pulled from the stack and loaded into the A, B, X or Y registers using the PULA, PULB, PULX or PULY instructions, as shown in Figure 6.3. Since the destination of the data is included in the mnemonic and the address of the data source is indicated by the address in the stack pointer register, all pull instructions use the inherent addressing mode. Pull instructions have no effect on the CCR. Pull instructions are complementary to push instructions.

Example 6.1

Problem: For each of the following push or pull instructions, indicate what effect the instruction has on the registers or memory.

```
0030   81 22 33 44 55 66 77 88

         A    $80        X      $B600
         B    $FF        S      $0035

     Address   Machine Code   Source Code
a.   0109      36             PSHA
b.   0100      32             PULB
c.   01E2      38             PULX
```

Solution:

a. (A) → $0035, $0034 → S
PSHA writes the contents of AccA ($80) to location $0035 in memory overwriting the previous data in this location. The stack pointer is decremented to $0034 after the data is pushed.

b. $0036 → S, ($0036) → B
PULB increments the stack pointer to point to the last location used on the stack ($0036). It then reads the contents of location $0036 from memory and loads AccB with the data ($77).

c. \$0036 $\rightarrow$ S, (\$0036) $\rightarrow$ X_L, \$0037 $\rightarrow$ S, (\$0037) $\rightarrow$ X_H.

PULX increments the stack pointer to point to the last location used on the stack (\$0036). It then reads the contents of location \$0036 from memory and loads the low byte of the X register with the data (\$77). The stack pointer is incremented again to \$0037, followed by a read of that new location. The data is loaded into the high byte of the X register (\$88).

Self-Test Questions 6.1

1. When data is pushed onto sequential locations of the stack, is the stack pointer incremented or decremented? Why?
2. Is a pull instruction a read or a write operation?
3. What data movement instruction performs a similar operation to the PULA instruction?

6.2 Function of Subroutines

A **subroutine** is a self-contained subprogram. Each subroutine contains one or more instructions associated with a specific task. For example, a subroutine could be written to perform a time delay, to add a list of numbers or to convert a number from ASCII to hex. Since subroutines are modular groups of code, they are not required to reside in sequential order in memory. The subroutines are executed in any order. This order is dictated by the main program. A **subroutine call** is a special instruction that must be used to start the execution of a subroutine. The HC11 supports two subroutine call instructions: Jump-To-Subroutine (JSR) and Branch-To-Subroutine (BSR). Each subroutine must end with a special instruction that returns the flow to the calling program. This instruction is the Return-From-Subroutine (RTS). Figure 6.4 summarizes the instructions that manage the use of subroutines.

Figure 6.4 summarizes the function of the RTS, JSR and BSR instructions. JSR operates in the DIR, EXT, INDX and INDY addressing modes. BSR operates in the relative mode. RTS operates in the inherent addressing mode. JSR, BSR and RTS have no effect on the CCR.

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
JSR	Jump to subroutine	(P) $\Rightarrow$ M_S:M_{S-1}, S – 2 $\Rightarrow$ S, Eff address $\Rightarrow$ P	-	X	X	X	X	-	-	-	-	-	-	-	-	-	-
BSR	Branch to subroutine	(P) $\Rightarrow$ M_S:M_{S-1}, S – 2 $\Rightarrow$ S, Eff address $\Rightarrow$ P	-	-	-	-	-	-	X	-	-	-	-	-	-	-	-
RTS	Return from subroutine	(M_{S+1}:M_{S+2}) $\Rightarrow$ P, S + 2 $\Rightarrow$ S	-	-	-	-	-	X	-	-	-	-	-	-	-	-	

Figure 6.4 Subroutine Instructions

The JSR instruction is similar to JMP in that it uses an absolute 16-bit address to point to the address of the first instruction of the subroutine. The BSR is similar to BRA in that it uses an 8-bit relative mode offset to point to the address of the first instruction of the subroutines. Because the JSR instruction uses a 16-bit address, it allows access to subroutines at any location within the memory map. The BSR instruction can only access subroutines within the +127 or –128 byte limits of the relative address. JSR and BSR cause an unconditional change in the program flow to the subroutine. The HC11 does not support conditional branch to subroutine instructions.

NOTE: The JSR or BSR must be executed to call a subroutine. Every subroutine must end with the RTS instruction in order to return to the calling program.

Steps to Execute a Subroutine

When a subroutine is used on the HC11, a sequence of three events takes place. Each step of the following sequence is described in subsequent sections.

1. Call the subroutine, using a JSR or BSR instruction.
2. Execute the instructions of the subroutine.
3. Return from the subroutine by executing the RTS instruction.

Calling the Subroutine

The only way to execute a subroutine is via the JSR or BSR instruction. These instructions cause the flow of a program to be routed to a subroutine. In chapter 4, the function of the JMP and BRA instructions is presented. JMP and BRA have the job of loading the program counter with the address of the next instruction to execute. Although JSR and BSR are similar to JMP and BRA, they perform an additional task. JSR and BSR cause a return address to be pushed onto the stack prior to loading the program counter with the address of the first instruction in the subroutine. The **return address** is the address of the instruction that will be executed after the program returns from the subroutine. It is always the address of the instruction immediately following the JSR or BSR in memory. This information is used by the subroutine to return to the calling program. Without the return address, the subroutine would not be able to return to the proper location in the main program.

Figure 6.5 illustrates how the return address is pushed to the stack. The low byte is written to the current address in the stack pointer. The stack pointer is then decremented and the high-order byte is written to this location. Then the stack pointer

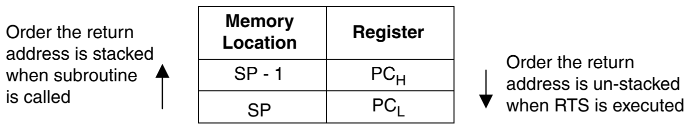

Order the return address is stacked when subroutine is called
Memory Location
Register
SP - 1
PC_H
SP
PC_L
Order the return address is un-stacked when RTS is executed

Figure 6.5 Placement of Return Address on Stack

is decremented one more time so that it continues to point to the next available location on the stack.

After the return address is pushed onto the stack, the program counter is loaded with the address of the first instruction of the subroutine. The JSR and BSR instructions acquire this starting address from the effective address included with the instruction. JSR uses the DIR, EXT, INDX or INDY addressing mode, and BSR uses the REL addressing mode to specify the effective address.

Example 6.2

Problem: For the following program, identify the return address and the starting address of the subroutine and show the effect on the stack when the subroutine is called.

```
Address  Machine Code  Label  Source Code
0100     8E 01 E5             LDS     #$01E5
0103     BD 01 09             JSR     DELAY
0106     86 29                LDAA    #$29
0108     3F                   SWI

0109     CE 0D 06      DELAY  LDX     #$0D06
010C     09            LOOP   DEX
010D     26 FD                BNE     LOOP
010F     39                   RTS
```

Solution: The Return Address is always the address of the instruction immediately following the JSR or BSR instruction. In this program it is the address of the LDAA instruction, which is $0106.

The starting address of the delay subroutine is $0109.

The starting stack pointer is $01E5, because of the LDS instruction. The low byte of the return address will be written to this location, then the stack pointer will be decremented to $01E4. The high byte of the return address is written to this location of the stack, and the stack pointer is decremented again to $01E3.

```
$06 → $01E5, $01E4 → S
$01 → $01E4, $01E3 → S
```

Returning from the Subroutine

After the subroutine is executed, the flow of the program must return to the calling program. Remember, the return address was pushed onto the stack by the calling instruction. This return address must be pulled from the stack in order to know the address of the next instruction to execute. The last instruction of each subroutine must be the RTS instruction, because the RTS instruction pulls the return address from the stack and loads it into the program counter.

Example 6.3

Problem: Given the following memory block and stack pointer, what is the address of the next instruction to be executed following an RTS instruction? What is the effect on the stack pointer?

```
01E0  00  01  FF  38  01  06  2A  4B     S = 01E3
```

Solution: The stack pointer is first incremented to $01E4. The high byte of the return address is pulled from this location ($01). The stack pointer is then incremented to $01E5, and the low byte of the return address is pulled ($06). Therefore the return address is $0106, and the stack pointer is changed to $01E5.

Efficiency Trade-offs

Subroutines offer a programmer the ability to modularize the code within a program. A function that is performed several times throughout the program can be written as a subroutine. Instead of having one continuous set of instructions, subroutines allow **code reuse**. The instructions contained in the subroutine can be executed many times, from various parts of the program. Thus, subroutines provide a means of using memory more efficiently. Without subroutines, all code would have to be in-line. **In-line** code contains all functions in the order they are used, and the program is one continuous set of instructions. In-line code executes very fast because it does not use the subroutine control instructions (JSR, BSR and RTS). Because in-line code must duplicate a function each time it is used, it does not use memory resources efficiently.

Each time a subroutine is executed, two extra instructions are required in the code. The JSR or BSR instruction is required to call the subroutine. Moreover, each subroutine must be terminated by the RTS instruction. The RTS retrieves the return address from the stack so that the program returns to the proper location in the calling program. The time it takes to execute these two instructions, as well as the memory space they require, is called **subroutine overhead**. In other words, there is a cost in execution speed (time) and memory usage (space) associated with the use of subroutines. If the overhead of using subroutines is greater than the equivalent in-line implementation, there is no advantage in using subroutines.

Some programmers feel that the main program should be nothing but calls to subroutines. In theory this is a good practice, but it can be taken a bit too far. Subroutines should be used for tasks that are executed more than once within a program. These tasks should be carefully structured to minimize the impact of the extra instructions required to call and return from subroutines.

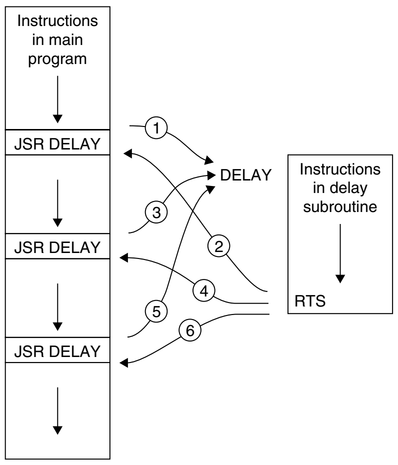

Instructions in main program
JSR DELAY
JSR DELAY
JSR DELAY
1
2
3
4
5
6
DELAY
Instructions in delay subroutine
RTS

Figure 6.6 Example of a Subroutine

NOTE: In-line code executes faster than the equivalent application using subroutines. Therefore, the in-line code uses time more efficiently. Subroutines typically use less memory than the equivalent in-line implementation. Thus, subroutines are more memory efficient.

Subroutines are typically used when a program has a need to do a group of tasks repeatedly. As a subroutine, the tasks are called by various parts of the program, as shown in Figure 6.6. The DELAY subroutine is needed in more than one place in the program. Without the ability to modularize the code into a subroutine, the DELAY task would have to be imbedded into the main program repeatedly to form one set of in-line code. It should be apparent that using the DELAY subroutine is a more efficient use of memory than would be required with an in-line implementation. Since this program has time delays built in, there is no concern for the extra time it requires to execute the DELAY subroutine.

Self-Test Questions 6.2

1. How would you describe a subroutine in your own words?
2. What value does a subroutine provide to a program?
3. How is a subroutine called?
4. How many bytes are pushed onto the stack when a subroutine is called? What are they?
5. What must happen to achieve balanced use of the stack?

6.3 Parameter Passing

Subroutines often require data from the program to complete their function. For example, a subroutine that converts a hex value to the equivalent ASCII character code must be provided the hex value that will be converted. In the same manner, subroutines often have the need to return data to the calling program. When the conversion is completed by the hex-to-ASCII conversion subroutine, the ASCII character code must be returned to the calling program. **Parameter passing** is the name given the process of passing data to and from subroutines.

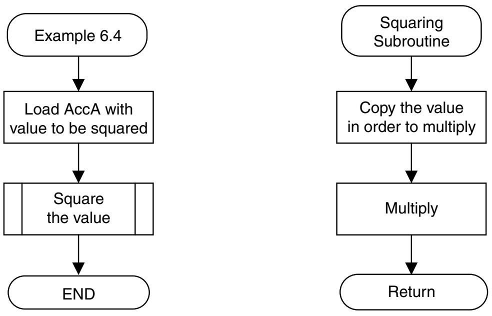
There are two primary methods of passing these parameters to and from subroutines, pass by value and pass by reference. The **pass by value** method sends a copy of data to the subroutine. The subroutine uses this copy of the data in whatever manner necessary to complete the function of the subroutine. The **pass by reference** method sends an address of the data to the subroutine. The address represents the location in memory where the data is stored. During the execution of the subroutine, the data is accessed via this address.

The easiest way to pass by value is to load a processor register with the data prior to calling the subroutine. The subroutine uses the data from the register during the execution of the subroutine. The pass-by-reference method loads either the X or Y register with an address where the data is stored. The subroutine then must use an indexed mode instruction to access the data. Data can also be returned to the calling program by using either of these methods. It is common to pass a value to a subroutine via a processor register to have the value processed and then to return the processed value in the same register.

Example 6.4

Problem: Write a subroutine to accomplish a squaring function. Use the following flowchart as an outline of what the subroutine should do. Use the pass-by-value technique to pass the data to and from the subroutine. Use AccA to pass the data to the subroutine and return the squared data in AccD. Assemble the code starting with location $0180.

Solution: The squaring function can be accomplished in two instructions. Since the value to be squared is in AccA, this value needs to be copied to AccB to prepare for the multiplication. The square is found by multiplying the contents of AccA by the contents of AccB. The result of the MUL instruction is automatically stored in AccD.


Example 6.4
Load AccA with value to be squared
Square the value
END
Squaring Subroutine
Copy the value in order to multiply
Multiply
Return

Example 6.4 Flowchart

```
0001 0180 16   SQUARE  TAB      ;duplicate value (A) → B
0002 0181 3d           MUL      ;square value, (A) x (B) → D
0003 0182 39           RTS      ;return to calling program
```

In some cases, subroutines do not require any data from the calling program. They perform operations that are independent of the data being processed. A time delay is an example of this type of subroutine. The time delay executes a set of instructions a fixed number of times and then returns to the calling program. The job of the time delay is simply to keep the processor busy for a fixed length of time.

Balanced Use of Stack

Often during execution of a subroutine, the contents of the registers that are changed by the routine are first pushed onto the stack. This way the original contents of these processor registers can be restored at the end of the routine by pulling them back off the stack. For example, if the X register is used temporarily by a subroutine, as shown in Figure 6.7, it is wise to push the contents of the X register onto the stack at the beginning of the subroutine. The routine can then use the X register for any purpose, because the original contents are saved on the stack. At the end of the subroutine, the contents of the X register must be restored by executing a pull instruction. This is called **balanced use** of the stack. Balanced use implies that whatever is pushed onto the stack is removed from the stack before another process is done.

NOTE: Values must pulled from the stack in an order opposite to that in which they were pushed onto the stack to maintain the integrity of the original data. For example, PSHA, PSHX must be complemented with PULX, PULA.

When a subroutine is called, the two-byte return address is pushed onto the stack by the JSR or BSR instruction, and the RTS pulls these two bytes back off. This is another example of balanced use of the stack.

```
*Delay Subroutine

DELAY  PSHX              ;Save X (2-bytes) to stack

       LDX    #$0D06     ;Use X as counter

AGAIN  DEX               ;Loop until X = 0
       BNE    AGAIN

       PULX              ;Restore X (2-bytes) from stack

       RTS
```

Figure 6.7 Balanced Use of Stack

Self-Test Questions 6.3

1. What is a simple method of passing a value to a subroutine?
2. If a value is passed by reference, what addressing mode must be used to access the value in memory?
3. What must be done before the RTS instruction to maintain a balanced stack if the first instruction of the subroutine is PSHX?

6.4 Converting a Hex Value to ASCII

Most data processed by the computer is stored as hex values in memory. However, if this data needs to be displayed on the monitor, it is typically converted to the ASCII character code format. ASCII is a seven-bit code that is typically stored as bytes, where as the MSB is zero filled. The remainder of this section will describe a subroutine designed to convert a single hex digit (4 bits) to the equivalent ASCII code. The hex digit will be passed to the subroutine by value in the lower nibble of AccA. The ASCII code will be passed back to the calling program by value also using AccA. If the hex value is not a single hex digit, the routine will return the hex value unchanged.

ASCII is a sequential code that follows the numeric and alphabetic order of the characters. For example, the ASCII code for "A" is $41, for "B" is $42, and so on. Higher-order digits translate to higher-order ASCII codes.

Figure 6.8 summarizes the 16 hex digits and their equivalent ASCII codes. If each of the values of the hex digits is subtracted from the corresponding ASCII codes,

Hex Value	ASCII Code	Numeric Difference
$0	$30	$30
$1	$31	$30
$2	$32	$30
$3	$33	$30
$4	$34	$30
$5	$35	$30
$6	$36	$30
$7	$37	$30
$8	$38	$30
$9	$39	$30
$A	$41	$37
$B	$42	$37
$C	$43	$37
$D	$44	$37
$E	$45	$37
$F	$46	$37

Figure 6.8 Relationship of Hex Values to ASCII Equivalent Codes

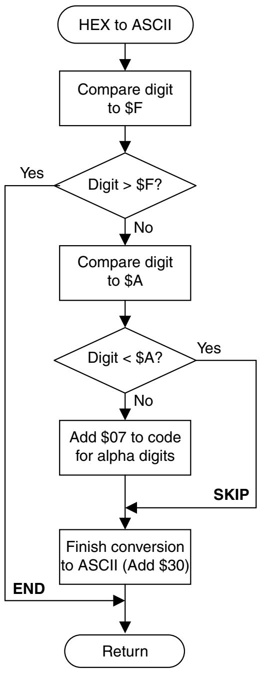


Figure 6.9 Flowchart of Hex-to-ASCII Conversion Subroutine

a difference results with a distinct pattern. The numeric digits “0” through “9” all have a numeric difference from the ASCII codes of $30. The alpha digits “A” through “F” all have a numeric difference from the ASCII codes of $37. Thus, the ASCII code for the numeric digits can be generated by adding $30 to the hex value. The ASCII codes for the alpha digits can be generated by adding $37 to the hex value.

Figure 6.9 illustrates the flowchart of this subroutine, and Figure 6.10 contains the source code listing for the subroutine.

This subroutine checks the input value to make sure that it is a valid hex digit. If the character is not is a valid hex digit, it is returned without any processing. The first step is to compare the input value to $0F. If the input value is greater than $0F, then it is not a valid hex digit. The routine returns without processing the digit. The next step is to check if it is less than $0A. If the input value is less than $0A, then the input value is a valid digit 0–9, and no special processing (addition of $07) is required. If the input value is A–F, then $07 must be added to the value. Careful examination of the ASCII table reveals why $07 must be added to some values.

If the input value is determined to be one of the numeric digits 0–9, this extra offset is skipped. The final ASCII character code is calculated by adding $30 to the hex digit if it is 0–9. If it is A–F, then $37 is added to the hex digit.

```
                   * Subroutine to convert the lower hex digit ($0-$F)
                     of a byte in AccA
                   * to ASCII equivalent.  Return ASCII of digit in
                     AccA.

0120 81 0f         HEX2ASC CMPA    #$0F      ;Is digit > $F?
0122 22 08                 BHI     END       ;Value not single hex digit
0124 81 0a                 CMPA    #$0A      ;Is digit alpha?
0126 25 02                 BLO     SKIP      ;If NOT skip extra $07
0128 8b 07                 ADDA    #$07      ;Alpha digit needs $07 more
012A 8b 30         SKIP    ADDA    #$30      ;Finish ASCII conversion
012C 39            END     RTS
```

Figure 6.10 Source Code Listing of Hex-to-ASCII Conversion Subroutine

Self-Test Questions 6.4

1. What is the purpose of the compare to $0F in HEX2ASC?
2. Why is $07 added to the code if the hex value is not lower than $0A?

6.5 Nested Subroutines

Often there is a need to call subroutines from another subroutine. This is called **nesting** and is illustrated in Figure 6.11. Nested subroutines provide increased efficiency and provide for better code reuse. Theoretically, nesting can go on for many levels, that is, a subroutine can call another, which can call another, which can call another, and so on. However, system constraints limit this practice. First, each time a subroutine is called, the two-byte return address is pushed onto the stack. If another subroutine is called before the first one is finished executing, then two additional bytes will be pushed onto the stack as the return address for this next routine. Each additional nested layer requires two more bytes for each subsequent return address. Eventually the system will run out of stack space by filling all available memory with return addresses.

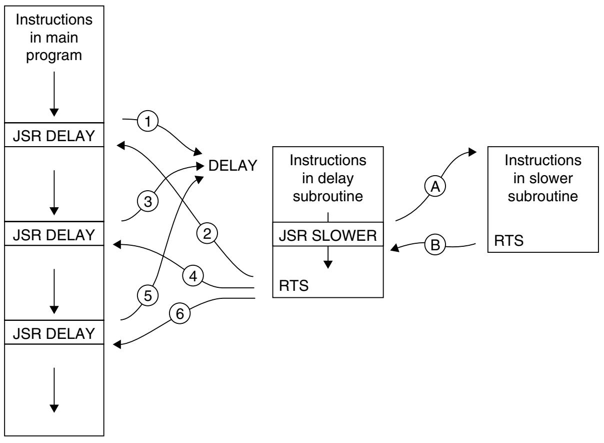

Instructions in main program
JSR DELAY
JSR DELAY
JSR DELAY
1
2
3
4
5
6
DELAY
Instructions in delay subroutine
JSR SLOWER
RTS
A
B
Instructions in slower subroutine
RTS

Figure 6.11 Example of a Nested Subroutine

Excessive nesting also leads to excessive overhead costs. The processor requires extra machine cycles to execute the JSR/BSR and RTS required each time a subroutine is called. In some cases, these extra machine cycles are more than the routine requires for execution. Therefore, moderation is the best practice. There needs to be a balance between the modular benefits of subroutines and the execution efficiencies of in-line code. The ideal balance is dependent on the particular system and its priorities on speed versus memory efficiency.

Time calculation lends itself well to this concept of subroutine nesting. For example, a one-minute delay subroutine could be implemented by calling a one-second delay subroutine 60 times. Furthermore, the one-second delay subroutine could call a 10 ms delay subroutine 100 times. This idea is illustrated in Example 6.5.

Example 6.5

Problem: Write a subroutine that will delay the processor for 1 second. Use the 10 ms delay subroutine from Example 6.2, but enhance the routine so that the registers are not changed when this delay is complete. Estimate as closely as possible with a single program loop the actual time of the 1 second delay.

Solution: The flowchart shown in Figure 6.12 illustrates the specific operations necessary to accomplish this task. The source code listing for this task is shown in Figure 6.13. This particular listing contains an additional column of information.

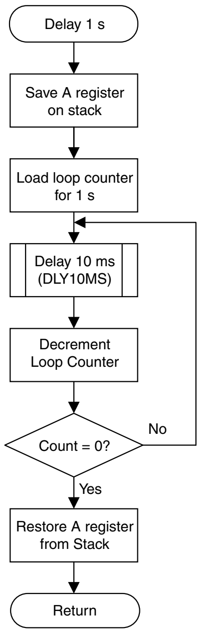


Figure 6.12 Flowchart of 1-Second Delay Subroutine

```
                              * Subroutine to delay 1 s by calling a 10 ms delay 100 times

0140 36             [ 3 ] DLY1S   PSHA             ;Save A temporarily on stack
0141 86 64          [ 2 ]         LDAA    #100     ;set loop counter
0143 bd 00 0b       [ 6 ] DLY2    JSR     DLY10MS  ;Delay 10 ms
0146 4a             [ 2 ]         DECA             ;decrement loop counter
0147 26 fa          [ 3 ]         BNE     DLY2     ;If not zero do again
0149 32             [ 4 ]         PULA             ;restore value in A
014A 39             [ 5 ]         RTS              ;return

                              * Subroutine to delay 10 ms

014B 3c             [ 4 ] DLY10MS PSHX             ;Save X temporarily on stack
014C ce 0d 01       [ 3 ]         LDX     #$0D01   ;set loop counter
014F 09             [ 3 ] DLYLP   DEX              ;decrement loop counter
0150 26 fd          [ 3 ]         BNE     DLYLP    ;If not zero do again
0152 38             [ 5 ]         PULX             ;restore value in X
0153 39             [ 5 ]         RTS              ;return
```

Figure 6.13 Source Code Listing of Nested Subroutine Time Delay

Each instruction has the number of machine cycles required to execute shown in brackets. This information facilitates the calculation of the actual delay.

The DLY10MS subroutine utilizes delays 10 ms; it needs to be executed 100 times to accomplish the 1 second delay. The overhead associated with calling this routine, returning from the routine and initializing the loop are minor compared to the overall length. The code for the DLY10MS subroutine is shown again in the same listing for continuity.

Each time the DLY2 loop is executed in the DLY1S routine, the DLY10MS routine is executed. Thus, the DLY10MS routine is nested within the DLY1S routine.

The two subroutines DLY1S and DLY10MS are written so that the original contents of the registers are not affected. These routines use AccA and the X register as counters to accomplish the delay. At the start of each subroutine, the counter register is pushed onto the stack. This saves the original contents while the register is being used for another purpose. Before the subroutine is complete, the original register value is restored to the counter register via a pull instruction. The combination of the push and pull instructions inside each subroutine maintains a balanced use of the stack.

One more example is presented to show the value of nested subroutines. In section 6.4, the process of converting a hex digit to the equivalent ASCII character code was presented. Since each byte stored in memory can contain two hex digits, the HEX2ASC subroutine could be called twice to convert each digit stored within a byte. Example 6.6 shows how a routine could be written to convert two hex digits from a single byte into two ASCII characters. It utilizes the HEX2ASC subroutine from section 6.4.

Example 6.6

Problem: Write a subroutine to convert a two-digit hex number to the two equivalent ASCII character codes. Utilize the HEX2ASC subroutine from section 6.4 to convert each digit separately and then store the results in two sequential bytes in memory. The address of the first byte is passed to this subroutine in the X register.

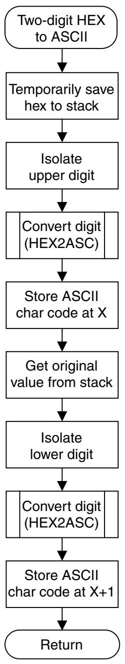

Two-digit HEX to ASCII
Temporarily save hex to stack
Isolate upper digit
Convert digit (HEX2ASC)
Store ASCII char code at X
Get original value from stack
Isolate lower digit
Convert digit (HEX2ASC)
Store ASCII char code at X+1
Return

Figure 6.14 Flowchart of Two-Digit Hex to ASCII Conversion

Solution: Each byte of data contains two hex digits. The left digit will be converted first and stored at the memory location contained in the X register. The right digit will then be converted, and the ASCII equivalent will be stored at X + 1. The conversion subroutine (HEX2ASC) requires that the hex digit to be converted is located in the right position of the byte and the upper nibble is zero. Therefore, the upper hex digit must be moved to the lower digit before it can be converted. Figures 6.14 and 6.15 illustrate how this is done.

The original data is stored on the stack so that the upper digit can be shifted and converted. The upper digit is shifted to the lower digit position, and the HEX2ASC converter is called. This subroutine returns the ASCII equivalent that can be directly

```
* Subroutine to convert two hex digits in A register
* to ASCII equivalent.  Return the ASCII in memory pointed to by X.

0160 36           HH2AA   PSHA                ;Store A on stack
0161 44                   LSRA                ;Shift upper digit to lower
0162 44                   LSRA
0163 44                   LSRA
0164 44                   LSRA
0165 bd 00 00             JSR     HEX2ASC     ;Convert digit to ASCII
0168 a7 00                STAA    $00,X       ;Store ASCII at X
016A 32                   PULA                ;Get original hex value
016B 84 0f                ANDA    #$0F        ;Isolate lower digit
016D bd 00 13             JSR     HEX2ASC     ;Convert digit to ASCII
0170 a7 01                STAA    $01,X
0172 39                   RTS
```

Figure 6.15 Source Code Listing of Two-Digit Hex to ASCII Conversion

written to memory. The indexed mode is used to address memory since the effective address is located in the X register. This completes the conversion and storage of the upper digit.

Following the conversion of the first digit, the original data is retrieved from the stack so that the lower digit can be converted. The AND instruction is used to apply a logical mask to isolate the lower digit. Again, the HEX2ASC converter is called, which returns the ASCII equivalent character code of the hex digit. This ASCII code is then written to the memory location following the location of the first digit.

Self-Test Questions 6.5

1. What is a nested subroutine?
2. What problems can be caused by nesting?
3. How many machine cycles are required to call and return from a subroutine?

6.6 BUFFALO Subroutines

The BUFFALO monitor program contains many subroutines that it uses to accomplish its job. These subroutines are also available for use by the user programs. Figure 6.16 summarizes the subroutines documented in the EVBU users manual. Each of these

Jump Table Address	Start Address	Routine Name	Description
$FF7C	$E0CE	WARMST	Go to the BUFFALO prompt ">" (skip BUFFALO message).
$FF7F	$E1B8	BPCLR	Clear the breakpoint table.
$FF82	$E1F9	RPRINT	Display the user registers on the screen.
$FF85	$E207	HEXBIN	Convert ASCII character code from the A register to a 4-bit binary number. Shift the binary number into SHFTREG from the right. SHFTREG is a 16-bit register (it will hold 4 hex digits). If the value in A is not an ASCII character code for a valid hex digit, then SHFTREG is unaffected.
$FF88	$E23A	BUFFAR	Read 4-digit hexadecimal argument from input buffer to SHFTREG.
$FF8B	$E25D	TERMAR	Read 4-digit hexadecimal argument from terminal device to SHFTREG.
$FF8E	$E285	CHGBYT	Write value from SHFTREG+1 (two right-most hex digits of 4-digit buffer) to the memory location pointed to by the X register. This operates on EEPROM locations as well.
$FF91	$E2F1	READBU	Read next character from INBUFF.
$FF94	$E2F8	INCBUF	Increment pointer into input buffer.
$FF97	$E2FE	DECBUF	Decrement pointer into input buffer.
$FF9A	$E306	WSKIP	Read input buffer until non-white space character code is found.
$FF9D	$E329	CHKABR	Monitor input for ctrl-X, Delete, or ctrl-W requests.
$FFA0	$E1AD	UPCASE	If the character code contained in the A register is a lowercase character ('a' – 'z'), then convert it to the equivalent uppercase character code ('A' – 'Z').

Figure 6.16 BUFFALO Subroutines Referenced in Jump Table

Jump Table Address	Start Address	Routine Name	Description
$FFA3	$E316	WCHEK	Test the character code in the A register and set the Z bit if the code is a white space character code (space, tab or comma).
$FFA6	$E321	DCHEK	Test the character code in the A register and set the Z bit if the code is a delimiter character code (carriage return or white space).
$FFA9	$E378	INIT	Initialize the I/O device.
$FFAC	$E39E	INPUT	Read from the I/O device.
$FFAF	$E3CA	OUTPUT	Write to the I/O device.
$FFB2	$E4DE	OUTLHL	Convert the left half of the A register to an ASCII character code and output it to the terminal port.
$FFB5	$E4E2	OUTRHL	Convert the right half of the A register to an ASCII character code and output it to the terminal port.
$FFB8	$E4EC	OUTA	Output the ASCII character code in the A register to terminal port.
$FFBB	$E4F0	OUT1BY	Convert each digit of the byte pointed to by the address in the X register to ASCII character codes and output on the terminal. Increments the X register so that it points to the next byte in memory.
$FFBE	$E4FF	OUT1BS	Same as OUT1BY, except a space is output after the two characters.
$FFC1	$E4FC	OUT2BS	Same as OUT1BS, except four characters from two consecutive bytes are output followed by a space. X is incremented twice.
$FFC4	$E508	OUTCRL	Output the ASCII carriage return followed by the line feed character codes.
$FFC7	$E518	OUTSTR	Output the ASCII character codes until end-of-transmission ($04) is found. The first location is pointed to by the address in the X register.
$FFCA	$E51B	OUTST0	Same as OUTSTR except the leading carriage return and line feed are skipped.
$FFCD	$E544	INCHAR	Input the ASCII character code from the terminal and store in the A register. This routine does not return until a valid character code is received.
$FFD0	$E357	VECINIT	Initializes the Vector Jump Table in RAM with JMP instructions and default addresses.

Figure 6.16 BUFFALO Subroutines Referenced In Jump Table *(continued)*

routines is designed to be accessed from the same location in memory regardless of the version of BUFFALO that is being used. Each version of BUFFALO has a variety of changes that cause the subroutines to appear in different locations in the memory. To assist programmers with the access of these common subroutines, BUFFALO was designed with a lookup table for these subroutines. This lookup table is called the subroutine Jump Table. Each entry in the jump table includes a JMP instruction and the actual starting address of the subroutine.

For example, there is a subroutine called UPCASE in the jump table. It converts the ASCII code that is in AccA to the uppercase equivalent. The JMP instruction for UPCASE is located in the Jump Table at $FFA0, but the actual subroutine is located at $E1AD within the BUFFALO monitor program. The Jump Table provides a fixed location to access these subroutines. On another machine the UPCASE subroutine can be located anywhere in the memory. As long as the Jump Table remains constant, the user software will still work.

Example 6.7

Problem: Write a program that will input a character from the keyboard, convert the character to uppercase and output the character on the monitor. Use BUFFALO subroutines for all functions. Assemble the program at $0180 in memory.

Solution: This program can be accomplished with three subroutine calls. BUFFALO provides a subroutine that will allow input from the keyboard (INCHAR), another to convert the character to uppercase (UPCASE) and finally one to output the character on the monitor (OUTA). Each of these passes the character via AccA so that no additional instructions are necessary. Notice the use of the assembler directive EQU to define the Jump Table addresses for each of the subroutines. The subroutines are then referenced by name in the code, which makes the code more readable.

```
   * Program to read a character from the keyboard, convert
   * it  to uppercase and then output it to the monitor.

0180 bd ff cd      JSR    $FFCD  ;get char from keyboard
0183 bd ff a0      JSR    $FFA0  ;convert to uppercase
0186 bd ff b8      JSR    $FFB8  ;output the char to monitor
0189 3f            SWI
```

In addition to the subroutines listed in Figure 6.16, there are many more subroutines within the BUFFALO monitor program. These subroutines are also accessible to the user and can be very useful. As they do not have corresponding entries in the Jump Table, the program must use the actual address of the subroutine to access them. Some of these additional subroutines are listed in Figure 6.17.

Address in BUFFALO 3.4	Routine	Description
$E290	WRITE	Write the byte from the A register to the memory location pointed to by the address in the X register. This routine support EEPROM writes. It first performs an erase of the affected location if necessary.
$E2C6	EEBYTE	Erase a single byte in the EEPROM.
$E2D4	EEBULK	Erase the entire EEPROM memory.
$E2E5	DLY10MS	Delay 10 ms. This routine does not change the contents of any processor register.
$E39E	INPUT	Reads the I/O device and returns an ASCII character code in the A register. If no key has been pressed on the keyboard, it returns 0. Unlike INCHAR, this routine does not wait for a valid character, but returns immediately.
$E502	OUTSPAC	Output a space to the monitor.
$E538	TABTO	Move cursor to the twentieth position on the monitor.
$FC38	TXBWAIT	Delay 100 ms. This overwrites the contents of the Y register.

Figure 6.17 BUFFALO Subroutines NOT Referenced in Jump Table

Self-Test Questions 6.6

1. What function does the DLY10MS subroutine provide?
2. When would it be useful to execute the INCHAR subroutine?
3. What subroutine would be executed to write a single byte to a location in the EEPROM?

Summary

A subroutine is a group of code that is modularized for a particular function. It is a set of instructions that work together to accomplish a specific task. The instructions are grouped together in a manner that allows any other program or function to access them. A subroutine is accessed by executing a JSR or BSR instruction. When the JSR or BSR instruction is executed, the flow of the program switches to the subroutine and a return address is saved on the stack. Each subroutine must conclude with the RTS instruction. The RTS instruction retrieves the return address from the stack so that the flow of the program transfers back to the main program.

When writing and using a subroutine, the programmer must remember what each subroutine does and what the starting address of the subroutine is and must know what parameters must be initialized and what registers or memory locations are changed by the subroutine.

Chapter Questions

Section 6.1

1. Define LIFO and describe how it works on a computer memory stack.
2. What is the destination of the data that gets loaded by the following instruction: PULX?
3. What is the destination of the data that gets stored by the following instruction: PSHA?

Section 6.2

4. How many bytes are pushed onto the stack when a subroutine is called? What are these bytes?
5. How many methods can be used to call a subroutine?
6. What address is always used as the return address of a subroutine call?
7. Why must each subroutine end with an RTS instruction?
8. How can the code in a subroutine be reused?
9. What is the minimum number of machine cycles required to call and return from a subroutine?
10. What is the main advantage of in-line code over subroutines?
11. What is the main advantage of subroutines over in-line code?

Section 6.3

12. Which registers could be used to pass a single byte to a subroutine?

13. Which registers could be used to pass the starting location of a list of 20 bytes to a subroutine?
14. How is passing by reference similar to the extended addressing mode?
15. Explain balanced use of the stack.

Section 6.4

16. What logical instruction could be used to assure that the value passed to the HEX2ASC subroutine is a single hex digit before conversion?

Section 6.5

17. Theoretically, how many layers of subroutines can be nested together?
18. How many nested layers are required to complete the 1-minute delay illustrated in Figure 6.12?
19. What simple method could be used to increase the 1-minute time delay to a 2-minute time delay?

Section 6.6

20. What is the purpose of the subroutine jump table?
21. What BUFFALO subroutine can be used for a 100 ms delay?
22. How is the start of the message passed to the OUTSTR subroutine?

Chapter Problems

1. Using the data in Figure 6.18, express the results of the following instructions, including status of the stack pointer. What is the opcode (including prebytes) of each instruction? NOTE: Each instruction is independent of the others. The results are not cumulative.

```
a.  PSHA
b.  PSHX
c.  PULB
d.  PULY
e.  INS
f.  PSHB
g.  RTS
```

```
Registers   A =  $D0       Memory    00F8   00 11 22 33 44 55 66 77
            B =  $64                 0100   F3 56 E3 DB A1 A0 09 00
            X =  $1000               0108   22 44 52 88 63 77 74 33
            Y =  $E216               0110   FF 00 FF 11 FF 22 FF 83
            S =  $0103               0118   F1 3B BB B6 D4 AD CE 00
```

Figure 6.18 Default Data Set #4

2. Given the following instruction, what is the return address of the subroutine call?

```
012B BD 01 A5 JSR $91A5
```

3. Which of the following short code segments are examples of balanced use of the stack?

```
a. PSHA      b. PSHY      c. PSHX      d. PSHX
   PSHX         PSHB         PULY         PSHA
   PULX         PULX                      PULB
   PULA                                   RTS
```

4. Write a subroutine to delay one minute using subroutines. Use PSH and PUL instructions to protect the original data in registers that are used by the subroutine.
5. Write a short program that will use the HH2AA subroutine to convert the four-digit hex number stored in AccD to four ASCII characters. Start the program at $0100 and store the four digits starting at $0000.
6. Write a program to send the message "HELLO WORLD!" to the monitor. Use the BUFFALO subroutines to accomplish this task.
7. Write a program to accept two decimal values from the keyboard, add the numbers and display the decimal sum on the monitor. Use BUFFALO subroutines to accomplish the task.

Answers to Self-Test Questions

Section 6.1

1. The stack pointer is decremented when data is pushed onto the stack. The last item pushed onto the stack is the first item that comes off the stack.
2. Read operation.
3. LDDA.

Section 6.2

1. A subroutine is a set of instructions grouped by function and is terminated by the RTS instruction.
2. A subroutine provides modularity by grouping instructions into functional blocks. It also provides a simple way of reusing the same code in various parts of the program, which provides a more efficient use of the memory.
3. Via a JSR or BSR instruction.
4. The two-byte PC is pushed onto the stack.
5. Each byte that is written to the stack must be read from the stack.

Section 6.3

1. Load the value into a processor register before calling the subroutine.
2. The indexed addressing must be used to access data that was passed by reference.
3. The PULX instruction must be executed.

Section 6.4

1. If the hex value passed into the subroutine is greater than $0F, then it is not a single digit.

2. The difference between the ASCII codes and the alpha digits is $37, and that between the ASCII codes and the numeric digits is $30. Thus, an extra value of $07 must be added to the codes converted from alpha digits.

Section 6.5

1. When a subroutine is called by another subroutine, this practice is called nesting.
2. Nesting causes a significant amount of overhead by pushing and pulling multiple return addresses to and from the stack. The JSR and RTS instructions require 10 or more machine cycles to execute.
3. JSR = 5, 6 or 7 cycles; RTS = 5 machine cycles.

Section 6.6

1. The DLY10MS subroutine provides a time delay of 10 ms.
2. The INCHAR subroutine is used when the user wants to wait for keyboard input.
3. The WRITE subroutine allows writes to EEPROM. It requires the A register to contain the data and the X register to contain the address to be written to.

chapter 7

Working with an Assembler

Objectives

After completing this chapter, you should be able to:

- Understand techniques of writing source code that is readable
- Utilize assembler directives, labels and directives to enhance source code
- Explain the fields in the listing file
- Define the contents of S records and be able to interpret what the fields of each record contain

Outline

Introduction

An **assembler** is a piece of software that converts mnemonic instructions into machine code. Line by line the assembler converts the mnemonic instructions to the machine code that will actually execute on the processor. This machine code is made up of the operational codes and the operands.

Many assemblers use a two-pass process to perform the conversion function. During the first pass through the source code, the assembler builds a symbol table that contains addresses and definitions of all the labels. A preliminary set of machine code is also generated. On the second pass, the labels are replaced with the definitions and addresses that the machine code needs to complete the execution of the instructions. The branch displacements are also calculated and placed in the proper place in the code. As errors are found in the source code, the assembler will inform the user via messages to the monitor or to a listing file. The error message contains a line number from the original source code file and some kind of error message that defines the error type. The final machine code file is not generated until all the errors are resolved.

Assemblers are written so that they operate on a specific set of mnemonic instructions unique to a certain processor or processor family. The assembler described in this chapter is the AS11 assembler. It generates machine code for the HC11 family of processors.

7.1 Writing Source Code

Source code is sometimes considered a form of art. Each programmer uses a personal style or a style dictated by the employer. Some styles are very organized and modular, where white space is used to separate functions and processes and each line is commented in detail. Other programmers focus solely on the function of the code and pay little attention to the look and layout of the code. Regardless of the programmer's opinion and personal style, there are conventions that should be followed to provide order, readability and maintainability of the code. Three conventions will be discussed that can make a significant impact on the readability of source code: use of labels, commenting code, and using white space.

Use of Labels

One of the key factors that make code more readable is the use of labels. Labels should be used for any reference to memory or to a constant value. When the labels are defined with terms that imply the function or operation that is being applied, it makes the code more readable. For example, use MSG as the label for a message to be displayed or PORTB as the label for the address of the PORTB register. Whenever the port B register is accessed, the label will remind the reader of the code of what is located at the effective address. Some programmers are extremists and have labels defined for every numeric constant and every address used in the code; thus, no numeric values appear in the body of the executable code. The numeric values are replaced by the label instead.

Labels are defined in the AS11 assembler in the label field of the source file. The label field starts in the left-most column of each line of the source file. If the first character is the asterisk "*", the entire line is treated as a comment; thus the line is ignored by the assembler. If the first character of a line is a space, the assembler assumes that there is no label defined for the line of code. Labels can still be used in the operand field, but these labels must be defined elsewhere in the code. If the first character of the line is a symbol character, then a label is defined in the label field. Label symbols are limited to upper- and lower-case characters a – z and the digits 0 – 9. Three special characters are also

```
HERE      LDAA #$23
          BNE  HERE

LOOP:     DECA
          BNE  LOOP

PORTB     EQU   $1004

MSG1      FCC   'Waiting'
```

Figure 7.1 Examples of Labels in Source Code

allowed: the period ".", the dollar sign "$" and the underscore "_". Labels can be one to eight characters long. Each label can start with any of the label symbols except the dollar sign. Labels are also case sensitive. To create greater visual recognition of labels when they are defined, the optional colon ":" character may be used to end the label. The colon is not part of the label but acts as a separator. Figure 7.1 contains examples of valid labels.

Commenting Code

Comments within the code provide the user with clues about the function of the program. The comments provide additional information regarding the function or operation of one or more lines of code. They should also explain why the function is being done. Comments can be most effective when two types are used. The first type occupies an entire line or several lines of the source file. It gives a general description of what is to follow. If it is describing the function of a subroutine, it may include a definition of the parameters used by the routine and the results that are passed back when complete. All full-line comments must be preceded by the asterisk character. The asterisk must appear in the left-most position of the line. This indicates to the assembler that this line contains only a comment. Blank lines within the source code file are also treated as full-line comments.

Each line of code can be commented to the right of the instruction. The AS11 assembler interprets anything that follows the last byte in the operand field as a comment. Some assemblers require the end-of-line comments to begin after a certain column in the source code or require the comments to begin with a special symbol like the semicolon character. The comments on each line should be specific to a single line or perhaps a pair of lines. They should provide the reader a reason that that line of code was included in the source.

NOTE: Often comments tell the reader what the instruction is doing. The "what" is implicit in the instruction itself and needs no further explanation. In addition to the "what," comments should emphasize the "why," so that the reader gains a sense of the reason things are done.

Using White Space

White space can provide a significant degree of readability. White space is defined as a space, tab or blank line. Subroutines can be separated by one or more blank lines to help the reader understand where one function ends and another starts. Within a

```
        ORG     $0100
        SEI
        LDD     #$0180
        STD     $00DD
        LDX     #$1000
        LDAA    #$40
        STAA    $20,X
        STAA    $22,X
        CLI
MAIN    WAI
        BRA     MAIN
        ORG     $0180
        BCLR    $23,X $BF
        LDD     $18,X
        ADDD    #$3E8
        STD     $18,X
        RTI
```

a) Without Labels, Comments and White Space

```
***********************************************************
* HC11 Output Compare Demonstration
* Squarewave Generation
*
* This program generates a 1KHz squarewave on OC2 (pin 28)
* using an Interrupt Service Routine
**********************************************************
* Declarations
**********************************************************

REGBASE EQU     $1000

OC2_JMP EQU     $00DD

TOC2    EQU     $18
TCTL1   EQU     $20
TMSK1   EQU     $22
TFLG1   EQU     $23

**********************************************************
* Main Program
**********************************************************

        ORG     $0100           ;Start program at $0100

        SEI
        LDD     #OC2_ISR        ;Load ISR address into the Jump Table
        STD     OC2_JMP

        LDX     #REGBASE        ;Load register base address into Index X

        LDAA    #$40            ;toggle OC2 on successful compare
        STAA    TCTL1,X
        STAA    TMSK1,X         ;enable OC2 interrupt

        CLI

MAIN    WAI                     ;Wait for interrupt
        BRA     MAIN

OC2_ISR BCLR    TFLG1,X $BF     ;Clear OC2 flag

        LDD     TOC2,X          ;get compare value from TOC2

        ADDD    #$3E8           ;add offset for next pulse transition
        STD     TOC2,X          ; 1000 (3E8) counts high / low for 1KHz

        RTI
```

b) With Labels, Comments and White Space

Figure 7.2 Examples of Readability

routine, white space can be used to group functional lines of code. Often, more than one line of code is needed to complete some operation. These lines should be grouped together to make the code more readable.

Figure 7.2 illustrates two versions of the same source code file. The first version in Figure 7.2a is shown without labels, comments or white space. The second version is complete with labels, comments and abundant use of white space. The improvement in readability should be evident.

Self-Test Questions 7.1

1. How can a programmer make the code more readable?
2. What are two different types of comments that can be used in a source file?
3. How can white space be used to make the code more readable?

7.2 Assembler Functions

The assembler supports the use of special symbols within the code. These symbols provide the assembler with additional information about the mnemonic instructions. In addition, these symbols tell the assembler how to process the source code.

Data Type Constants

The AS11 assembler supports five numeric data type contants: decimal, hex, binary, octal and ASCII characters. Numbers are considered decimal unless they are preceded by a special symbol. If the user wants the assembler to interpret numbers as hex values, the numbers must be preceded by the "$" symbol. If numeric values are to be interpreted as binary, they must be preceded by the "%" symbol. If numeric values are to be interpreted as octal, they must be preceded by the "@" symbol. ASCII characters will be interpreted as the ASCII character codes when they are preceded by the single quote (') character. It is common practice to provide the trailing single quote, yet the AS11 assembler does not require it. Examples of these data types are shown in Figure 7.3.

The decimal, hex and ASCII data types are the most common since they are read most easily. The binary type is used in place of hex when specific bits are being emphasized, but it is more difficult to read. The octal data type is rarely used with modern computers. Octal is a grouping of three binary digits.

Directives

The assembler supports a special set of commands called directives. A **directive** tells the assembler what to do and indirectly affects the resulting machine code. It is an instruction to the assembler. However, a directive is not an instruction that is converted to machine code. For this reason, the directives are sometimes called pseudo-ops. Figure 7.4 summarizes the directives supported by the AS11 assembler and what functionality each directive provides.

```
86 7F     LDAA     #127          ;decimal 127 will be converted

86 2B     LDAA     #$2B          ;hex 2B will be used

86 D6     LDAA     #%11010110    ;binary 11010110 will be converted

86 9C     LDAA     #@234         ;octal 234 will be converted

86 41     LDAA     #'A'          ;'A' will be converted to ASCII code
```

Figure 7.3 Assembler Data Types

Directive	Syntax	Function
BSZ	BSZ *n*	Block Store Zeros—Reserves *n* number of memory bytes and fills them with zero.
EQU	EQU *expression*	Equate—Causes the value of the *expression* to replace the label in the listing and machine code files.
FCB	FCB *n*	Form Constant Byte—Causes a byte of data *n* to be assigned to specific memory locations. Successive bytes can be separated by commas.
FCC	FCC *'String'*	Form Constant Character—Causes ASCII codes to be assigned to specific memory locations. The characters of *String* are assigned to sequential memory locations. The *String* must appear between single quotes (*'String'*).
FDB	FDB *n*	Form Double Byte—same as FCB except double bytes are formed.
FILL	FILL *val, n*	Fill Memory—Causes *n* memory locations to be filled with constant *val.*
ORG	ORG *address*	Origin—Causes the following code to be assigned to a memory block starting with *address.*
RMB	RMB *n*	Reserve Memory Byte—Causes *n* number of memory locations to be reserved.
ZMB	ZMB *n*	Zero Memory Byte—same as BSZ.

Figure 7.4 AS11 Assembler Directives

One of the directives is a control directive. Control directives do not have a label preceding them as do the data directives. ORG is a control directive. Data directives define the data used by a program. They allocate memory for that data. Data directives are preceded by a label. The address of the data assigned by the data directive is assigned to this label; thus the label points to the data in memory.

Figure 7.5 illustrates how each of these directives is used. The source code block starts with four EQU directives. They are used to assign addresses to three labels PORTB, DLY10MS and OUTSTR, as well as a byte of data to a data parameter called TEN. The first ORG directive sets the default address of the machine code to the beginning of RAM at $0000. Next, an eight-byte memory block is reserved by the use of the RMB directive. The address of the first of these eight bytes is assigned to the label BUFFER (BUFFER = $0000). Then, two bytes are reserved and zeroed. The label ZERO is the address of the first of these two bytes (ZERO = $0008).

A second ORG statement appears in the code that changes the default address memory address to $0100. At this point in the source the program instructions start. The program consists of all the instructions up to and including the SWI instruction. Following the end of the program is a message in memory. The message is assigned to memory by the FCC directive. FCC converts each character of the message to the ASCII character codes.

```
0001                          * program to print message on the monitor ten times
0002                          * designed to demonstrate the use of various directives
0003
0004 1004                     PORTB   EQU     $1004   ;assign address $1004 to label PORTB
0005 e2e5                     DLY10MS EQU     $E2E5   ;assign address $E2E5 to label DLY10MS
0006 ffc7                     OUTSTR  EQU     $FFC7   ;assign address $FFC7 to label OUTSTR
0007
0008 000a                     TEN     EQU     10      ;assign decimal value 10 to label TEN
0009
0010
0011 0000                             ORG     $0000   ;set default address to $0000
0012
0013 0000                     BUFFER  RMB     8       ;reserve 8 bytes starting at BUFFER
0014 0008 00 00               ZERO    ZMB     2       ;zero 2 bytes following 8 byte BUFFER
0015
0016
0017 0100                             ORG     $0100   ;set default address to $0100
0018
0019 0100 18 ce 00 00         MAIN    LDY     #BUFFER ;load address of buffer into Y
0020 0104 86 0a                       LDAA    #TEN    ;load value of TEN into A
0021
0022 0106 bd e2 e5            LOOP    JSR     DLY10MS ;call subroutine at address DLY10MS
0023 0109 ce 01 19                    LDX     #MSG    ;load address of MSG into X
0024 010c bd ff c7                    JSR     OUTSTR  ;output message to the monitor
0025 010f 18 a7 00                    STAA    0,Y     ;store value of A at Y+0
0026 0112 b7 10 04                    STAA    PORTB   ;store value of A at PORTB
0027 0115 4a                          DECA            ;decrement counter
0028 0116 26 ee                       BNE     LOOP    ;if not done, do it again
0029
0030 0118 3f                          SWI
0031
0032 0119 57 61 69 74 65 64   MSG     FCC     'Waited 10 ms'
          20 31 30 20 6d 73
0033 0125 04                          FCB     $04     ;EOT character - delimiter
```

Figure 7.5 Example Using Directives

Assembler Arithmetic

Expressions can be used in many places throughout the code. The assembler will resolve the expression before making the assignment. If the label BUFFER equals the address $0000, then the expression BUFFER + 1 = $0001. An expression can consist of labels, constants or the asterisk character (denotes the value of the current memory address) joined by operators.

The AS11 supports eight operators, as shown in Figure 7.6. Expressions are evaluated from left to right, and there is no provision for parenthesized expressions. How these are used is illustrated in the comments of Figure 7.7.

Operator	Description
+	Integer addition
-	Integer subtraction
*	Integer multiplication
/	Integer division
%	Integer remainder
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

Figure 7.6 Operators Defined for Expressions

```
0001 000a            TEN         EQU   10
0002 0005            FIVE        EQU   TEN/2
0003 0001            REMAIN      EQU   TEN%3
0004 0014            TWENTY      EQU   TEN*2
0005 001e            THIRTY      EQU   TEN+TWENTY
0006 0007            SEVEN       EQU   TWENTY/2-3
0007
0008 0073            VALUE1      EQU   %01110011
0009 0096            VALUE2      EQU   %10010110
0010 0012            AND         EQU   VALUE1&VALUE2
0011 00f7            OR          EQU   VALUE1|VALUE2
0012 00e5            XOR         EQU   VALUE1^VALUE2
0013
0014 0100                        ORG   $0100
0015
0016 0100            BUFFER      RMB   8
0017 0108 00 00      ZERO        ZMB   2
0018 010a 02         DATA        FCB   SEVEN%FIVE
0019 010b 65 71      UPCASE      FCB   $45|$20, $51|$20
0020 010d 53 4a      LOCASE      FCB   $73&$DF, $6A&$DF
```

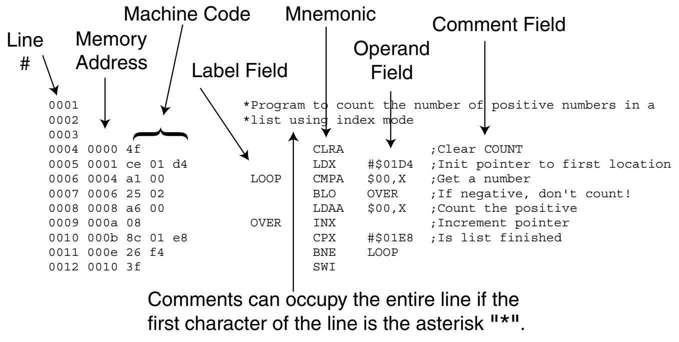
Figure 7.7 Assembler Expressions

Self-Test Questions 7.2

1. What symbol is used to designate hex numbers in the assembler?
2. What directive is used to define a byte in memory that is the ASCII character code for a single character?
3. What will be stored in memory by the following expression? SUM = $3B + 78

7.3 Listing Files

The AS11 has the ability to create a listing file. A **listing file** is a file that contains the combination of the original source code and the assembled code. Each line from the original source code file is included in the listing file, including blank lines. Each line is numbered to provide a cross-reference to the original source file. The line numbers are used when reporting error messages. Figure 7.8 illustrates all the features of a listing file.


Machine Code
Mnemonic
Comment Field
Line #
Memory Address
Operand Field
Label Field
Comments can occupy the entire line if the first character of the line is the asterisk "*".

```
0001                    *Program to count the number of positive numbers in a
0002                    *list using index mode
0003
0004 0000 4f                     CLRA           ;Clear COUNT
0005 0001 ce 01 d4               LDX    #$01D4  ;Init pointer to first location
0006 0004 a1 00         LOOP     CMPA   $00,X   ;Get a number
0007 0006 25 02                  BLO    OVER    ;If negative, don't count!
0008 0008 a6 00                  LDAA   $00,X   ;Count the positive
0009 000a 08            OVER     INX            ;Increment pointer
0010 000b 8c 01 e8               CPX    #$01E8  ;Is list finished
0011 000e 26 f4                  BNE    LOOP
0012 0010 3f                     SWI
```

Figure 7.8 Features of a Listing File.

Each line of the listing file starts with a line number. This number is a decimal number that is sequential throughout the file. The field contains the address in memory that the machine code will start for that instruction. Following the address is the actual machine code. That completes the machine code portion of the file. Everything to the right of the machine code is the original source code. The original source code includes the label, the mnemonics, the operands and any comments.

Errors

During the assembly, any errors that occur are included in the listing file. The error messages are included in the listing prior to the line that caused the error. The line number of the instruction that caused the error is referenced in the error message. Errors that are found in the first pass through the assembly process cancel the assembly. A lesser type of error called a "warning" does not cause the assembly to stop, yet it should be viewed critically. The source of warnings should be found and the problems resolved.

The AS11 assembler has the ability to identify a variety of syntax errors, but the logical and functional errors of the programs must be found and resolved by the programmer.

NOTE: During assembly, errors are reported to the monitor. The line number of the instruction within the source file that caused the error is provided in the error message.

Self-Test Questions 7.3

1. What is a listing file?
2. What process generates a listing file?
3. Where are error messages placed in the listing file?

7.4 "S" Records

The machine code that is generated by the process of assembling a source file is contained in a file called an S19 file. The S19 files are designed to simplify data transfer to the memory after assembly. The S19 format is printable; thus the transfer of data to memory can be visually monitored and the S records can be modified in a text editor. The files contain multiple S records.

The S records are essentially character strings that are made up of five fields, as shown in Figure 7.9. The type field must be "S1" or "S9" for AS11 records for the HC11 EVBU. The length field contains a two-digit hex value that represents the record length. The record length is the sum of the number of bytes in the address field and the number of bytes in the data field plus one for the checksum. The address field contains the character codes for a 16-bit address. Four characters are needed to represent the four hex digits that make up the address. The data field contains characters that represent each hex value in the machine code. Each byte of binary data is encoded as a

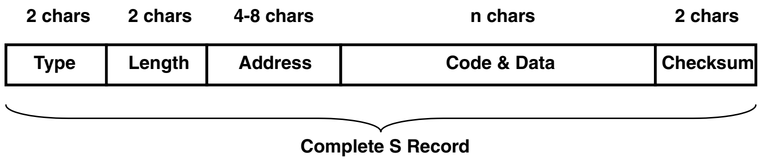


Figure 7.9 Format of S Records

two-character hex value. The first character of each two-character pair represents the upper four bits, and the second character represents the lower four bits of the machine code byte. The checksum field contains two characters. These characters are the 1's complement of the two-digit hex checksum value. The checksum is the sum of all characters, starting with the record length through the data field.

There are eight different S record types defined, but only S1 and S9 records are supported by the EVBU monitor program. S1 records contain program codes and program data in the data field. The address field of an S1 record contains the four-digit hex address of where these program codes and data are to reside. The S9 record is a termination record for a block of S1 records. The address field of an S9 record may optionally contain the address of the first instruction to be executed after the download is complete. S9 records contain an empty data field. Figure 7.10 describes all the parts of S1 and S9 records.

Self-Test Questions 7.4

1. What are the two types of S records defined for the AS11 assembler?
2. What are the fields of the S1 record type?
3. How are S9 records used?

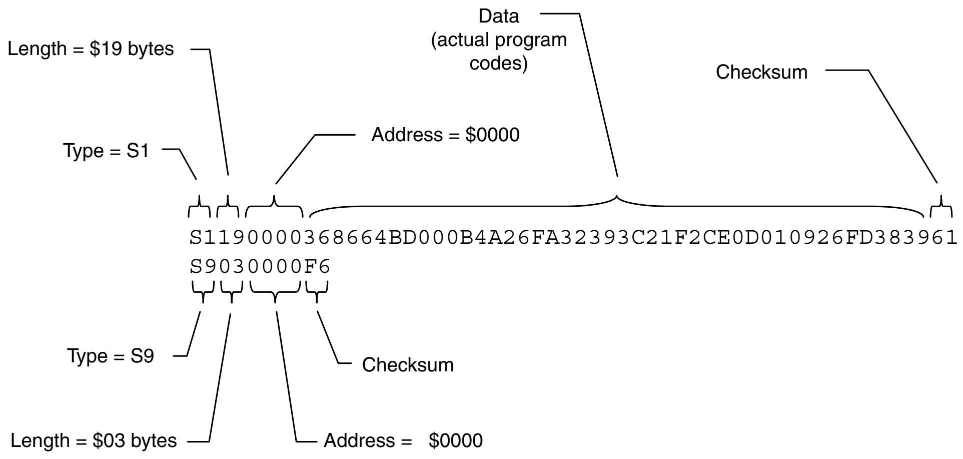


Figure 7.10 Examples of S Records

Summary

An **assembler** is a piece of software that converts mnemonic instructions into operational codes and operands. The source code is the original set of instructions written by the programmer. The use of labels, comments and white space can make the source code files more readable. The assembler contains many features that allow the programmer greater flexibility. These features include support for multiple data-type constants, directives (instructions to the assembler imbedded in the source code), the use of symbolic variables within the code called labels, and symbolic expressions.

The assembly process produces two types of output: listing files and S19 files. The listing files contain the original source code and the resulting machine code in a line-by-line format. The S19 files contain S records. The S records are character representations of the hex code and data that are to be loaded into memory for program execution. This code and data are packaged in the S record format, which includes some header fields and a trailing checksum that provide structure and organization to the download process.

Chapter Questions

Section 7.1

1. How are labels defined in the source code?
2. What symbols are allowed as labels?
3. Which of the following is an invalid label: GO_TEAM, DeVry.com, abc123, Help$, 1_time.
4. What is the maximum length of a label?
5. Should comments express the "what" or the "why" regarding the instructions?

Section 7.2

6. What symbol is used to designate ASCII character values in the assembler?
7. What directive could be used to define a string of bytes in memory that are the ASCII characters?
8. If the label REGS is defined to be equivalent to $1000, what value is loaded by the following instruction?

```
LDX   #REGS+6
```

Section 7.3

9. List the fields in the listing file.
10. What kinds of error messages terminate the assembly process?

Section 7.4

11. Essentially, what is an S19 file?

Chapter Problems

1. Modify the following example to increase readability without changing the program's function.

```
MAIN        LDAA    #$10
AGAIN       JSR     $E2E5
            DECA
            BNE     AGAIN
```

2. Use directives to define a label for the memory address $1004. Reserve a buffer of eight bytes of memory starting at address $0180 and a message called BOOT that says, "Booting up the computer," and is terminated by the EOT (end of transmission) character.

Answers to Self-Test Questions

Section 7.1

1. Use labels, comments and white space.
2. Full-line comments and comments that follow a line of code.
3. Blank lines can be inserted in the source code between subroutines to physically separate the functions. This makes it more clear where one function ends and another starts.

Section 7.2

1. The $ symbol must precede a number for the assembler to interpret the number as hex.
2. EQU could be used, as in CHARA EQU 'A'. FCC could also be used: CHARA FCC 'A'.
3. The assembler will perform the addition operation of the two constants $3B and $4E ($78_{10}$). SUM will be equal to $89.

Section 7.3

1. A listing file is an output file from the assembly process. It contains the original source code and the output machine code.
2. The assembly process generates the listing file by redirecting the terminal output to a file.
3. Error messages appear immediately prior to the line that contains the error. The error message refers to the line that caused the error.

Section 7.4

1. S1 and S9 records are used by AS11 and the EVBU.
2. The S1 record type contains five fields: type, length, address, data and checksum.
3. S9 records are used to terminate a transfer of data.

chapter 8

Memory Systems

Objectives

After completing this chapter, you should be able to:

- Describe the differences between address and data and explain how they are used
- Demonstrate the concepts of address decoding
- Show how the on-chip memory resources are allocated on the HC11
- Use the control bits that activate the EEPROM and ROM
- Use the control bits that relocate the RAM and system registers in the memory map
- Erase and write EEPROM locations
- Connect external memory devices to the HC11 using expanded mode

Outline

Introduction

One of the three critical components of a computer system is memory. Without memory, the computer could not be instructed what to do, because the instructions are stored in memory locations. Without memory, the computer system would not have any data to process because the data is also stored in the memory. This chapter focuses on aspects of the memory on an HC11. It provides a brief theoretical discussion of addresses and address decoding to broaden the understanding of memory on a computer system. The function of the three memory types on the HC11 are described: RAM, EEPROM and ROM. Special features are presented so that the user has a broad understanding of what capabilities are available, as well how these features can be implemented.

In addition to the traditional memory components, the HC11 contains a register block of 64 bytes. These registers are addressed in the same manner as the RAM, EEPROM and ROM. They also provide critical control of many HC11 system functions. Since this register block is memory in the broadest sense, a general discussion of the registers is provided here. Further information is provided in the appendix as well as in other sections of the text.

Finally, the chapter wraps up the discussion on memory with two sections related to the application of the memory. The HC11 supports the use of external memory devices. The concepts and application of the expanded memory capabilities are provided. In addition, the EEPROM can be used as the boot ROM on the EVBU. The application of this feature is discussed at the end of the chapter.

This chapter directly correlates to Sections 4 and 3.3.1 of the HC11 Reference Manual and section 4 of the Technical Data Manual.

8.1 Address Decoding

When we send a letter, it has to be addressed. Furthermore, electronic mail (e-mail) also requires an address. An **address** specifies where the data is being delivered and is not the data that is being delivered. The letter inside the envelope is the data that is being delivered. The address to which the letter is being delivered is on the outside of the envelope.

Addresses typically contain multiple parts. For example, an apartment number is an address within an apartment complex. If you were invited to visit someone in apartment number 216, you would probably ask, "In which complex?" Many complexes have an apartment 216. The complete address contains all the parts that uniquely identify a particular apartment. The complete address would include a street name and number, an apartment number, city, state, and zip code, as shown in Figure 8.1. This complete address must be unique. It identifies one apartment within a complex, one complex among the many that are on Dunlap Avenue, one street among the many streets in Phoenix, one zip code within the many zip codes within the state, and one state within the many states.

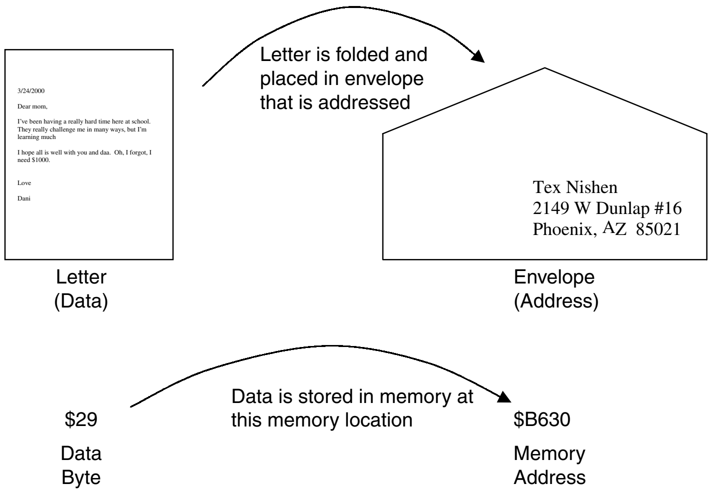
Tex Nishen
2149 W Dunlap #16
Phoenix, AZ 85021

Figure 8.1 Example of Address

In a similar fashion, addresses of computer memory locations also can contain several parts. Often the address parts need to be decoded in order for a particular memory location to be activated. Just like apartments in the apartment complex, a memory module may contain many independent storage devices that must have unique addresses. A single byte of data is stored at a location in a device. There can be multiple devices within a module. There can be multiple devices and multiple modules within the computer system; therefore, each one must be uniquely addressed so that the data gets stored in the proper memory locations, just as the letter gets delivered to the proper recipient.

Memory addresses, unlike street addresses, are just a bunch of hex digits, as shown in Figure 8.2. Unless close attention is paid, it can be difficult to discern the various parts, like module address, device address, and so on.

A good example of the concepts behind memory addressing is illustrated at most self-storage facilities. Figure 8.3 shows the layout of a typical self-storage facility. This particular facility has four buildings of storage units. Each building is a column on the drawing, numbered 1 through 4. The units within each building are numbered from 1 to 16. Each unit has a unique number (or address) that is a function of which building (column) it is in and its location within the column. The column number always is the first digit of the complete address. The unit number within the column is designated by the last two digits of the complete number.


3/24/2000

Dear mom,

I've been having a really hard time here at school. They really challenge me in many ways, but I'm learning much

I hope all is well with you and daa. Oh, I forgot, I need $1000.

Love

Dani

Figure 8.2 Relationship of Street Address to Memory Address

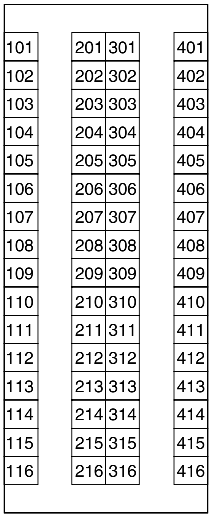


Figure 8.3 Addressing of Units in a Self-Storage Facility

The three-digit number is unique for each unit, but the various parts of each number are not (i.e., the column number is the same for all units in a column). The number for a specific unit within the facility could be referred to by the specific column and unit numbers. For example, unit 107 could be referred to as unit 07 on column 1. In this sense, the unit number is a code that can be easily decoded. The first digit of the code indicates a column and the last two digits represent the unit on that column. Although the unit numbers are unique for each storage unit, multiple boxes of "stuff" can be stored in each unit. The unit number represents where the "stuff" is stored; it is not the "stuff" itself.

Now let's apply the self-storage facility example to the operation of a memory. Figure 8.4 shows the layout of a simple memory module. This memory module looks a lot like the layout of the self-storage facility. In this example, there are four columns, and each column contains 16 storage locations. Each column in the storage facility was a building; in the memory, each column will be a memory chip. Each storage location has a unique, four-digit address. In this case, a 16-bit binary address is used. This address is shown as a four-digit hex number. The right-most digit is used to identify the location within a chip, and the left-most three digits represent a unique chip number. This addressing method is typical on many computer systems.

Each storage location can hold more than one data bit. In most computer systems, the memory is configured for bytes. This means that each memory location can hold eight binary bits of information. Each time data is written to a particular memory location, eight bits or an entire byte must be written. In the same manner, each time the data in a location is read, all eight bits are read simultaneously.

In the self-storage facility example, each unit could hold multiple boxes of stuff. The unit can even be empty. When the unit is full, a box must be taken out

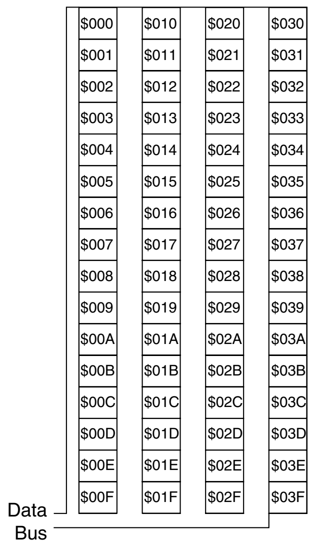


Figure 8.4 Addressing of a Memory Module

before a new box can be put in. This is never the case with a computer memory. Electronic memory always contains data, patterns of 1's and 0's; never can the memory location be empty. Often the data in a particular memory, location has no relevance to an operation, yet it is still there. When the processor writes data to memory the previous data is overwritten; thus, write operations are destructive. When the processor reads data from memory, the data is left unharmed. In reality a read operation takes a copy of the data from the memory location and brings it into the processor.

Self-Test Questions 8.1

1. What is an address?
2. Can an address contain more than one part? If so, give an example.
3. Why is the layout of a self-storage facility similar to the layout of a computer memory?

8.2 On-chip Memory

Each member of the HC11 microcontroller family has built-in memory. There are slight differences on each version of the chip, as shown in Figure 8.5. Each version contains RAM and at least one type of ROM (ROM, EPROM or EEPROM).

Each of these memory components is mapped into the address space. The **memory map** is a layout of where each memory block resides within the entire address space.

Part Number	RAM	EEPROM	ROM	EPROM	Comments
MC68HC11A8	256	512	8K	—	Entire HC11 family is built around this version of the device
MC68HC11A1	256	512	—	—	'A8 with ROM disabled
MC68HC11A0	256	—	—	—	'A8 with ROM & EEPROM disabled
MC68HC811A8	256	8K + 512	—	—	'A8 with EEPROM instead of ROM
MC68HC11E9	512	512	12K	—	Enhanced A8—Four IC functions, more RAM and ROM (used on the EVBU)
MC68HC11E1	512	512	—	—	'E9 with ROM disabled
MC68HC11E0	512	—	—	—	'E9 with ROM & EEPROM disabled
MC68HC811E2	256	2K	—	—	'E9 with more EEPROM, no ROM
MC68HC711E9	512	512	—	12K	'E9 with EPROM instead of ROM
MC68HC11D3	192	—	4K	—	Budget A8—40-pin package, reduced functionality
MC68HC711D3	192	—	—	4K	'D3 with EPROM instead of ROM
MC68HC11F1	1024	512	—	—	Enhanced A8—non-multiplexed expanded memory bus independent of PORTB and PORTC, 68-pin package
MC68HC11K4	768	640	24K	—	Enhanced A8—larger address bus, PWM, CS, more memory, 84-pin package
MC68HC711K4	768	640	—	24K	'K4 with EPROM instead of ROM
MC68HC11L6	512	512	16K	—	Enhanced E9—more I/O, more ROM, 64- and 68-pin packages
MC68HC711L6	512	512	—	16K	'L6 with EPROM instead of ROM

Figure 8.5 HC11 Memory Configurations (adapted with permission from Motorola)

This memory map is unique for each version of the HC11. On the HC11, user programs have no concern for the type of memory being accessed during read operations because all the memory types allow read operations. The RAM, EEPROM, ROM and external memory are read in the same manner. Each of these memory devices responds to a read operation within the constraints of the processor machine cycles. In other words, the memory during a read operation is at least as fast as the processor can generate read operations.

The ROM is of course "read only," so write operations make no change to the ROM memory locations. The EEPROM is also "read only," but data can be changed by a special programming operation. The RAM is general-purpose read-write memory.

The function of the EEPROM programming operation is explained in section 8.4.

Self-Test Questions 8.2

1. Do all versions of the HC11 microcontroller contain on-chip memory?
2. How much RAM is contained on the K4 version of the HC11?
3. What is the difference between the HC11E9 version and the HC711E9 version?

8.3 RAM

There are two primary types of RAM: static and dynamic. All static memory devices use a latch as the storage cell. Each bit of binary data is stored in a single latch. The term **static** refers to the fact that the state of a latch remains static or unchanged so long as power is applied or the new data is latched into the device. In contrast, **dynamic** RAM (DRAM) devices use a capacitor to store the data bits. Unlike a static device, a capacitor is constantly discharging or the voltage level is dynamic. All DRAM devices require refresh circuitry to keep the charge levels on the capacitors at the proper level to retain the 1's and 0's stored in them.

Every version of the HC11 contains SRAM. The HC11E9 contains 512 bytes that are fully functional read-write memory. The RAM is typically used for the storage of variables and data during program execution, yet it can also contain executable programs. In a microcontroller environment, executable programs are usually programmed into a nonvolatile memory (ROM or EEPROM) so that the programs are not inadvertently lost from power loss or erasure.

Location of RAM in HC11 Memory Map

The HC11E9 RAM is located by default at \$0000–\$01FF in the memory map. The user can move the location of the RAM in the memory map by changing the RAM map position control bits in the INIT register, as shown in Figure 8.6. These four bits specify the most-significant hex digit of the 16-bit RAM address in the memory map. The location can be changed to the start of any 4K memory block, as shown in Figure 8.7. 4K is the maximum number of memory locations that can be addressed with the remaining 12 bits of the 16-bit address bits (2^{12} = 4K).

NOTE: The INIT register is one of four protected control registers (BPROT, INIT, OPTION and TMSK2). In order to protect the sensitivity of the bits in the INIT register, write operations are protected in normal processor modes except under special circumstances. The INIT register can only be written once and that write must occur in the first 64 machine cycles after reset.

The ability to relocate the RAM in the address space allows the user to have greater flexibility with on-chip and off-chip resources. Address decoding circuitry is simplified if the memory device can be located in the zero page of the memory map. If the on-chip RAM were remapped to a higher memory area, the 4K zero page of the memory would be available for external connection.

INIT	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
\$103D	RAM3	RAM2	RAM1	RAM0	REG3	REG2	REG1	REG0
RESET	0	0	0	0	0	0	0	1

Figure 8.6 RAM Map Position

RAM[3:0] (Binary)	RAM[3:0] (Hex)	Address Range
0000	$0	$0000 - $01FF
0001	$1	$1000 - $11FF
0010	$2	$2000 - $21FF
0011	$3	$3000 - $31FF
0100	$4	$4000 - $41FF
0101	$5	$5000 - $51FF
0110	$6	$6000 - $61FF
0111	$7	$7000 - $71FF
1000	$8	$8000 - $81FF
1001	$9	$9000 - $91FF
1010	$A	$A000 - $A1FF
1011	$B	$B000 - $B1FF
1100	$C	$C000 - $C1FF
1101	$D	$D000 - $D1FF
1110	$E	$E000 - $E1FF
1111	$F	$F000 - $F1FF

Figure 8.7 RAM Mapping

Overlapping Memory Devices

The HC11 also allows the on-chip RAM, register block and ROM to be mapped to the same address space. They can also be mapped to the same space as an external memory device. The HC11 has internal address decode circuitry that automatically protects against memory access conflicts when the memory address space overlaps. When an internal device is read, the external bus is ignored, even if the external device drives the bus. Conflicts between on-chip resources are resolved with priority circuitry. The register block always has the highest priority, then the RAM, followed by the ROM.

On the HC11E9 the register block and RAM could be remapped to the same space, starting at $D000, as shown in Figure 8.8. In this case, the register block would respond to any memory accesses from $D000 to $D03F. The RAM and ROM would be disabled for this range. The RAM would respond to memory accesses from $D040 to $D1FF. The ROM would be disabled for this range. The ROM would then be active for the remainder of the 12K space from $D200 to $FFFF.

RAM on the EVBU

The BUFFALO monitor program on the EVBU uses a large portion of the RAM for the temporary storage of variables, data, the user stack and the interrupt vector jump table. Sometimes this area of RAM is called the **scratch pad** because of the way the BUFFALO monitor program uses it. BUFFALO reads and writes data to and from this area in a way similar to the way people jot notes on a scrap of paper. Figure 8.9 shows the layout of the scratch pad RAM area.

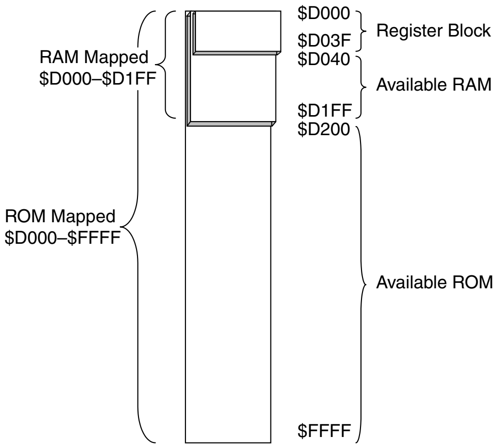


Figure 8.8 Overlapping Memory Map

NOTE: The BUFFALO Monitor program requires that the RAM be mapped in the default range of $0000–$01FF.

Self-Test Questions 8.3

1. What is the default memory location of the RAM on the EVBU? Can this location be changed?
2. How is the RAM used by the BUFFALO monitor program?

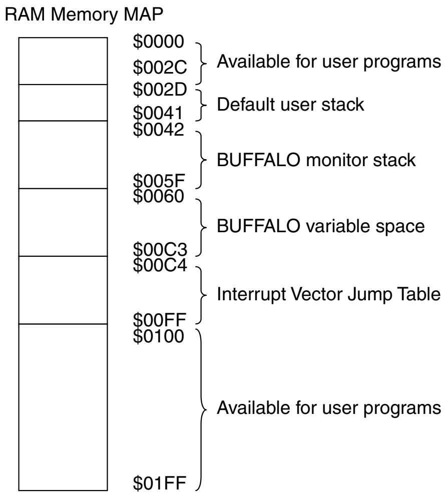


Figure 8.9 RAM Contents on EVBU

CONFIG		b7	b6	b5	b4	b3	b2	b1	b0
$103F		–	–	–	–	NOSEC	NOCOP	ROMON	EEON
	RESET	0	0	0	0	U	U	1	U

Figure 8.10 EEON Control Bit in the CONFIG Register

8.4 EEPROM

The HC11 supports on-chip EEPROM. The HC11E9 contains 512 bytes of EEPROM. The EEPROM is typically used to contain user executable programs, yet can also be used to store any data that requires more security from loss. The EEPROM on the HC11E9 is mapped to $B600–$B7FF. Unlike the RAM and the register block, the EEPROM cannot be remapped to another page of the unused memory. The on-chip EEPROM can be disabled via the EEON control bit of the CONFIG register. When the EEPROM is disabled, it is effectively removed from the memory map. Figure 8.10 illustrates where the EEON bit is located in the CONFIG register. When EEON = 1, the on-chip EEPROM is enabled. When EEON = 0, the on-chip EEPROM is disabled. The state of the EEON bit after reset is undefined. The E9 version of the chip used on the EVBU is shipped from Motorola with the CONFIG register set to $0F.

NOTE: As the CONFIG register is implemented in EEPROM, its contents remain unchanged after power loss; however, it is a separate 8-bit EEPROM. Other than the EEON bit, the CONFIG register has no connection to the EEPROM memory block.

Operation of the EEPROM

EEPROM by nature acts like a ROM, thus it is nonvolatile. During read operations, it acts like any memory location; however, the EEPROM requires special handling when data is being written to an EEPROM memory location. Write operations to an EEPROM location can only be performed by programming a location in the EEPROM.

Most EEPROM devices require special programming voltages and often require special hardware fixtures designed specifically for write operations. The EEPROM on the HC11 is designed so that none of the special hardware is needed. It can be reprogrammed via software control without the use of external circuitry or a special power supply.

The EEPROM on the HC11 is logically arranged in 32 rows of 16 bytes each. The first row occupies the locations $B600–$B60F, the second row occupies $B610–$B61F, and so on. Before a location in the EEPROM is written (programmed), it should be erased to assure that all bits in that location are high. The process of writing data into a location of the EEPROM removes fuse connections from the fuse map for all zeros in the data. Ones in the data cause the fuses to be left in place. Thus, all locations need to be connected prior to the write operation.

The EEPROM Programming register (PPROG) contains seven bits that control the operation of the EEPROM on the HC11. They layout of this register is shown in Figure

PPROG	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$103B	ODD	EVEN	–	BYTE	ROW	ERASE	EELAT	EPGM
RESET	0	0	0	0	0	0	0	0

Figure 8.11 PPROG Register

8.11. The ODD and EVEN control bits allow all locations in either the odd or the even half of the EEPROM array to be programmed with the same data simultaneously. They are used only during factory testing of the EEPROM. Bit 5 is unused and is always read as a zero.

EEPROM locations can be erased one at a time, in rows of 16 locations or all 512 bytes at once. The BYTE and ROW bits are used to determine the type of erase to perform when the ERASE bit is set, as shown in Figure 8.12. When ERASE = 1, the EEPROM can be erased. The BYTE controls single-byte erasures, and the ROW bit determines whether a ROW or the entire EEPROM will be erased. When ERASE = 0, the EEPROM is configured for normal read or programming mode.

The EEPROM Latch control bit (EELAT) controls the operations of the EEPROM. When EELAT = 0, the EEPROM is configured for normal reads. When EELAT = 1, the EEPROM is configured for programming or erasing. When EELAT = 1, the EEPROM is effectively removed from the memory map; thus it is not accessible during the relatively long time it requires to complete the erase or write operations. The relationship of the ERASE and EELAT bits is shown in Figure 8.13.

The LSB of the PPROG register is the EEPROM Programming Voltage Enable (EPGM or EEPGM in some literature). It is used to switch on the programming voltage to the EEPROM array, which is necessary for the erase and write operations. The HC11 uses an internal charge pump to build up the programming voltage so that an external

BYTE	ROW	Erase Action when ERASE = 1
0	0	Bulk Erase (Entire array)
0	1	Row Erase (16 bytes)
1	X	Byte Erase

Figure 8.12 EEPROM Erase Options

ERASE	EELAT	Function when EPGM = 1
0	0	EEPROM Read
0	1	EEPROM Write
1	0	Undefined
1	1	EEPROM Erase

Figure 8.13 ERASE and EELAT Functions

programming voltage is not required. This charge pump is activated when EELAT = 1. The charge pump must be activated before the HC11 will allow the programming voltage to be applied to the EEPROM array. The internal circuitry prevents the EPGM bit from being set unless the EELAT bit was previously set.

NOTE: The EELAT bit must be set in an operation prior to the operation that sets the EPGM bit. This is required to allow sufficient time for the internal charge pump to raise the programming voltage to the proper level. If the user attempts to set the EELAT and EPGM bits in the same operation (when they were both previously cleared), neither bit will be set.

Erasing the EEPROM

The EEPROM must be erased to assure all bits are 1's prior to a write operation. A write operation can only clear 1's; thus, bits that are already zero in an EEPROM location cannot be set to one with the write operation. Although there are three options available to erase EEPROM locations, the same process must be followed. Figure 8.14 lists the steps necessary to successfully erase EEPROM locations.

The code segments shown in Figure 8.15 illustrate how the procedure from Figure 8.14 can be implemented to erase the entire EEPROM, a row of 16 bytes in the EEPROM or a single location within the EEPROM. The three examples illustrate that there are only slight differences in the code, but significant differences in the results.

NOTE: The CONFIG register is implemented in EEPROM; however, it has no relationship to the EEPROM memory block. Therefore, none of the EEPROM code segments affects the contents of the CONFIG register.

Writing to the EEPROM

The EEPROM write operation is referred to as **EEPROM programming**. It follows a process similar to the steps of an erase operation, which is outlined in Figure 8.16.

Step	Description
1	Set the EELAT and ERASE bits of the PPROG register to prepare the EEPROM for erasure and select the desired erase operation (BYTE, ROW) as described in Figure 8.12.
2	Write any data to the EEPROM. The write operation is used only to indicate to the EEPROM control hardware which locations are to be erased; the value of the data is irrelevant. If the entire EEPROM is selected, any location within the EEPROM can be selected. If a row is being erased, then the location must be within the target row. A single location must addressed for it to be erased.
3	Set the EPGM bit in the PPROG register to start the erasure.
4	Wait 10 ms for the erasure to complete. (In some cases, the required delay may be longer than 10 ms.)
5	Clear all the bits in the PPROG register to deactivate the erase function.

Figure 8.14 Steps Necessary to Erase EEPROM Locations

```
BULKE   LDAB    #%00000110     ;ERASE=1, EELAT=1
        STAB    $103B          ;Activate BULK erase
        STAB    EEADD          ;Write any data to any EEPROM address to
                               ;  initialize erase operation
        LDAB    #%0000111      ;ERASE=1, EELAT=1, EPGM=1
        STAB    $103B          ;turn on programming voltage
        JSR     DLY10MS        ;Wait 10 ms
        CLR     $103B          ;ERASE=0, EELAT=0, EPGM=0
```

a) Bulk Erase

```
ROWE    LDAB    #%00001110     ;ROW=1, ERASE=1, EELAT=1
        STAB    $103B          ;Activate BULK erase
        STAB    EEADD          ;Write any data to EEPROM address in row
                               ;  to initialize erase operation of row
        LDAB    #%00001111     ;ROW=1, ERASE=1, EELAT=1, EPGM=1
        STAB    $103B          ;turn on programming voltage
        JSR     DLY10MS        ;Wait 10 ms
        CLR     $103B          ;ROW=0, ERASE=0, EELAT=0, EPGM=0
```

b) Row Erase

```
BYTEE   LDAB    #%00010110     ;BYTE=1, ERASE=1, EELAT=1
        STAB    $103B          ;Activate BULK erase
        STAB    EEADD          ;Write any data to any EEPROM address to
                               ;  initialize erase operation of address
        LDAB    #%00010111     ;BYTE=1, ERASE=1, EELAT=1, EPGM=1
        STAB    $103B          ;turn on programming voltage
        JSR     DLY10MS        ;Wait 10 ms
        CLR     $103B          ;BYTE=0, ERASE=0, EELAT=0, EPGM=0
```

c) Byte Erase

Figure 8.15 Examples of Code for Erasing the EEPROM

Step	Description
1	Set the EELAT bit and clear the ERASE bit of the PPROG register to prepare the EEPROM to receive data.
2	Write data to the EEPROM location. The write instruction does not accomplish the physical write but indicates to the EEPROM control hardware which location is targeted to receive the data that is provided.
3	Set the EPGM bit in the PPROG register to start the write.
4	Wait 10 ms for the write to complete. (In some cases, the required delay may be longer than 10 ms.)
5	Clear all the bits in the PPROG register to deactivate the write function.

Figure 8.16 Steps Necessary to Write to an EEPROM Location

During a write to the EEPROM, the ROW and BYTE bits of the PPROG register are not used. Data must be written to the EEPROM one byte at a time. As the erase operation, the write operation requires a 10 ms delay to allow the programming to complete. The code segment shown in Figure 8.17 illustrates one method that can be used to write a single byte of data to the EEPROM.

```
WRITEE  LDAB    #%00000010    ;EELAT=1
        STAB    $103B         ;Activate BULK erase
        STAB    EEADD         ;Write any data to any EEPROM address to
                              ;    initialize write operation for that address
        LDAB    #%00000011    ;EELAT=1, EPGM=1
        STAB    $103B         ;turn on programming voltage
        JSR     DLY10MS       ;Wait 10 ms
        CLR     $103B         ;BYTE=0, ERASE=0, EELAT=0, EPGM=0
```

Figure 8.17 Example of Code for Writing a Byte to the EEPROM

OPTION $1039	b7	b6	b5	b4	b3	b2	b1	b0
	ADPU	CSEL	IREQ	DLY	CME	–	CR1	CR0
RESET	0	0	0	1	0	0	0	0

Figure 8.18 Clock Select for EEPROM system.

EEPROM Clock

If the E clock is below 2 MHz, the 10 ms time delay may need to be increased to assure proper operation during EEPROM writes. If a longer delay is not an acceptable option, then the Clock Select (CSEL) bit of the OPTION register may have to be set. Figure 8.18 illustrates the location of the CSEL bit in the OPTION register. When CSEL = 1, the on-chip RC oscillator will drive the programming voltage (V_{PP}) instead of the E clock. When CSEL = 0, the E clock is used as the main clock for the EEPROM programming system. The CSEL bit is cleared after reset.

NOTE: CSEL is used to determine the charge pump clock source for the A/D conversion hardware as well as the EEPROM programming hardware. Setting CSEL for the EEPROM function automatically reconfigures the A/D function.

EEPROM Security

The HC11 provides a mechanism to protect the contents of the EEPROM from inadvertent writes or erasure. The Block Protect bits for the EEPROM (BPRT3–BPRT0) are located in the Block Protect register (BPROT), as shown in Figure 8.19. The BPRT3–BPRT0 bits control this EEPROM protection function. When these bits are cleared, programming and erasure of blocks of the EEPROM memory area is allowed. When these bits are set, programming and erasure are inhibited. The bits are set after reset and must be cleared by user software to allow EEPROM programming and erasure.

Each bit controls a block of the EEPROM, as shown in Figure 8.20. Each bit protects a unique block of the EEPROM. The blocks are different sizes, which allows greater

BPROT $1035	b7	b6	b5	b4	b3	b2	b1	b0
	–	–	–	PTCON	BPRT3	BPRT2	BPRT1	BPRT0
RESET	0	0	0	1	1	1	1	1

Figure 8.19 EEPROM Block Protect Control Bits

Bit Name	Block Size	Block Address Range
BPRT0	32	$B600 - $B61F
BPRT1	64	$B620 - $B65F
BPRT2	128	$B660 - $B6DF
BPRT3	288	$B6E0 - $B7FF

Figure 8.20 EEPROM Protection Blocks

versatility. When a single block is not protected, it is available for normal programming and erasure, even though the remaining blocks are protected.

NOTE: The BPROT register is one of four protected control registers (BPROT, INIT, OPTION and TMSK2). In the normal processor modes the bits in the BPROT register can be cleared only in the first 64 machine cycles after reset. The bits can be set at anytime during program operation. BUFFALO clears the BPROT register immediately following reset.

EEPROM on the EVBU

The BUFFALO monitor program on the EVBU does not use the EEPROM. All 512 bytes are available for user programs and data. Although the BUFFALO monitor program does not use the EEPROM, it provides five subroutines to program or erase the EEPROM. EEWRIT is used to write a byte of data to a particular location in the EEPROM. EEBYTE is used to erase a single location, and EEBULK is used to erase the entire EEPROM. Another routine called WRITE will write data to any memory location, including addresses in the EEPROM; if the EEPROM location is not erased, a byte-erase operation will be performed before the write is completed. The fifth subroutine is a 10 ms delay called DLY10MS.

WRITE, EEWRIT, EEBYTE and EEBULK use the X register to specify the memory location to process. WRITE and EEWRIT use the A register to contain the data that is to be written to the location specified in the X register. The 10 ms delay is automatically called by each of these subroutines, eliminating the need for the user to manage the required 10 ms delay. These subroutines do not affect any other registers.

Refer to Figure 6.16 for a brief explanation of the EEPROM subroutines as well as their location in BUFFALO.

Self-Test Questions 8.4

1. Describe the various methods available to erase the EEPROM.
2. Briefly, describe the steps required to write to the EEPROM.
3. What can be done to protect the contents of the EEPROM from accidental write or erasure?

8.5 ROM

The primary use of the on-chip ROM is storage of user's application programs. These programs cannot be changed, because they are programmed in the unit during the manufacturing process. Each series of the HC11 family of microcontrollers has a development version that has EPROM or EEPROM in lieu of ROM. This allows the user to develop and modify the programs on a single device. When the programs are debugged, an order can be placed for a high-quantity production version that will contain the user programs in ROM.

CONFIG	b_7	b_6	b_5	b_4	b_3	B_2	b_1	b_0
$103F	–	–	–	–	NOSEC	NOCOP	ROMON	EEON
RESET	0	0	0	0	U	U	1	U

Figure 8.21 ROMON Control Bit

The on-chip ROM can be disabled by a control bit in the CONFIG register, as shown in Figure 8.21. The ROMON control bit is set after reset enabling the on-chip ROM. When the ROMON bit is cleared, the on-chip ROM is disabled. When the on-chip ROM is disabled, none of the memory space is used for ROM. The user typically disables the on-chip ROM when the user programs are located in an external memory chip. Versions of the HC11 that do not contain ROM, such as the M68HC11E1, still have on-chip ROM, but it is disabled permanently. The ROM is disabled because the ROMON is permanently set to zero in the CONFIG register.

ROM on the EVBU

The BUFFALO monitor program on the EVBU is located in the ROM. It occupies the memory space from $E000 to $FFFF. The first 4K page of ROM from $D000 to $DFFF is unused and unavailable. The Interrupt Vector Table occupies the last 42 bytes of the addressable memory space, $FFD6–$FFFF, located in ROM. The BUFFALO is programmed into the ROM memory during manufacturing; thus it cannot be changed by the user.

See chapter 9 for further information on interrupts and the use of the interrupt vector table.

Self-Test Questions 8.5

1. When are the programs loaded into the ROM memory?
2. Can the on-chip ROM be deactivated? If so, how?

8.6 System Registers

The HC11 has a 64-byte block of the memory space for system registers. These registers contain control and status bits for the HC11 hardware systems. In addition, many of the registers contain data that is used by the hardware systems. The registers act as an interface between the user software and the on-chip hardware systems.

The HC11E9 system register block is located by default at $1000–$103F in the memory map. The user can move the location of the registers in the memory map by changing the REG map position control bits in the INIT register, as shown in Figure 8.22. These four bits specify the most significant hex digit of the 16-bit register block address in the memory map. The location can be changed to the start of any 4K memory block, as shown in Figure 8.23. 4K is the maximum number of memory locations that can be addressed with the remaining 12 bits of the 16-bit address bits (2^{12} = 4K).

INIT	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$103D	RAM3	RAM2	RAM1	RAM0	REG3	REG2	REG1	REG0
RESET	0	0	0	0	0	0	0	1

Figure 8.22 Register Map Position

REG[3:0] Binary	REG[3:0] Hex	Address Range
0000	$0	$0000 - $003F
0001	$1	$1000 - $103F
0010	$2	$2000 - $203F
0011	$3	$3000 - $303F
0100	$4	$4000 - $403F
0101	$5	$5000 - $503F
0110	$6	$6000 - $603F
0111	$7	$7000 - $703F
1000	$8	$8000 - $803F
1001	$9	$9000 - $903F
1010	$A	$A000 - $A03F
1011	$B	$B000 - $B03F
1100	$C	$C000 - $C03F
1101	$D	$D000 - $D03F
1110	$E	$E000 - $E03F
1111	$F	$F000 - $F03F

Figure 8.23 Register Mapping

NOTE: The INIT register is one of four protected control registers (BPROT, INIT, OPTION and TMSK2). In order to protect the sensitivity of the bits in the INIT register, write operations are protected in normal processor modes except under special circumstances. The INIT register can be written only once and that write must occur in the first 64 machine cycles after reset.

The ability to remap the register block address space allows the user to have greater flexibility between on-chip and off-chip resources. Address decoding circuitry is simplified if the memory device can be located in the zero block of the memory map. If the system register block were remapped to a higher memory area, the 4K block of the memory ($1000–$1FFF) would be available for external connection.

Further information regarding the entire register block is provided in Appendix 18.

Self-Test Questions 8.6

1. What is the default memory location of the system register block on the EVBU? Can this location be changed?
2. What is the purpose of the system register block?

8.7 Memory Expansion

The HC11 has two processor modes that are used for normal operation, single-chip mode and expanded mode. The expanded mode allows the HC11 to address memory devices that are not included on the HC11 chip. The single-chip mode does not extend the address and data busses to external devices, therefore single-chip mode designs are limited to the on-chip memory.

In expanded mode, the HC11 needs an external address bus, data bus and control lines to interface to the off-chip devices. This mode converts PORTB and Port C into a multiplexed data and address bus. PORTB becomes the upper byte (A_{15}–A_8) of the HC11 16-bit address bus, and Port C becomes the lower byte of the 16-bit address bus, which is time multiplexed with the 8-bit data bus (AD_7–AD_0). In other words, Port C is used for both address and data. In addition, the strobe A (STRA) and strobe B (STRB) control lines become the address strobe (AS) and read/write (R/W) control for the external memory devices.

During the first half of each bus cycle, the address signals A_7–A_0 are present on the lower half of the time multiplexed address/data bus. During the second half of the bus cycle, these pins are used as a bidirectional data bus. The AS signal is used as an active high latch enable. When AS is high, address information is allowed through the latch. When AS is low, the address information is latched. It remains unchanged until AS returns to the high state. The R/W signal indicates the direction of data. It is high during read cycles and low during write cycles. The E clock is used to synchronize these operations.

The schematic in Figure 8.24 illustrates the hardware connects necessary for a basic expanded-mode system. Expanded mode is selected by connecting both MODA and MODB control lines to V_{DD}. The address latch is present on the low-order byte of the address/data bus. The latch enable is connected to the AS signal. The output of the address latch forms the low-order byte of the address bus when the address/data bus is time multiplexed for data transfer. The R/W control line and E clock are connected directly to the address decode circuitry. The decode circuitry generates the unique chip select, output enable, read and write signals for the three external chips.

The address decoding used in this example is accomplished by a 74HC138 and a 74HC373 chip. The lower 13 address bits (A_{12}–A_0) are connected directly to each of the three chips. On these 13 address lines, 8K addresses can be created to uniquely address each location in the memory device. The other three address lines, A_{15}, A_{14}, and A_{13} are decoded to activate the external chip, as shown in Figure 8.25. The RAM chip responds to the fixed address range \$2000–\$3FFF.

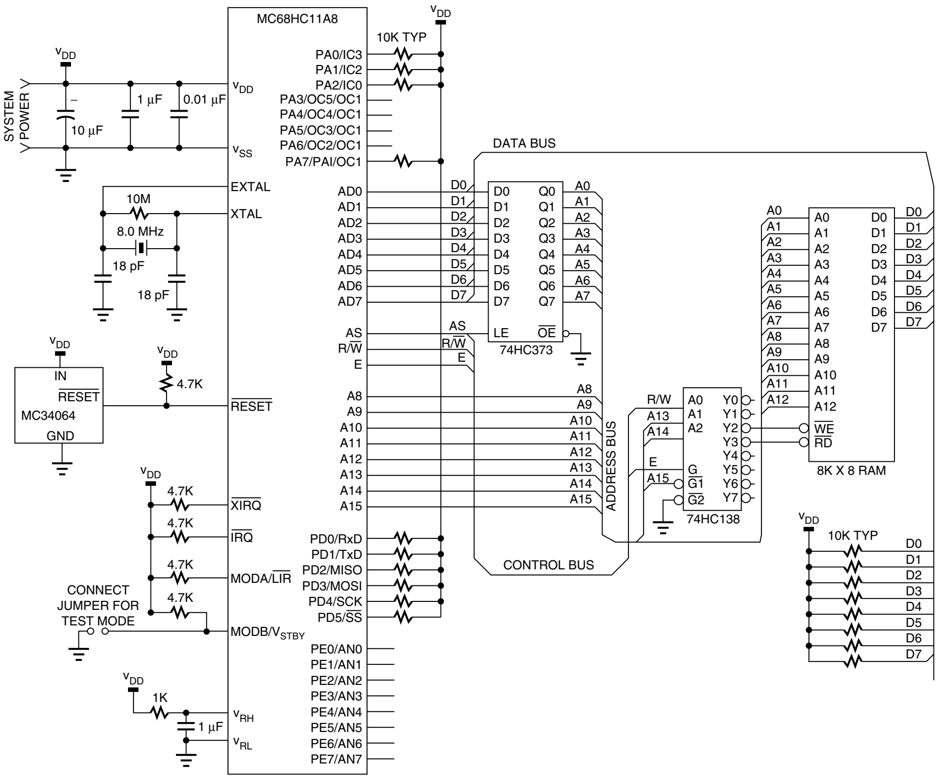


Figure 8.24 Basic Expanded-Mode Hardware Connections (adapted with permission from Motorola)

Self-Test Questions 8.7

1. What functions are performed by STRA and STRB in expanded mode?
2. How are the address and data busses created in expanded mode?
3. What role does address decoding play when the HC11 is configured for expanded mode?

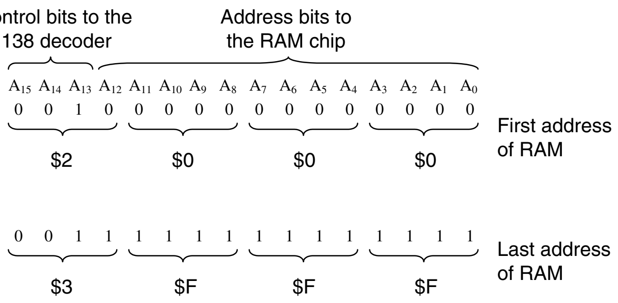


Figure 8.25 Address Decoding for the Expanded Memory

Summary

Memory is a critical part of all computer systems. All memory requires memory addresses that act as pointers to specific memory locations. The addresses must be decoded so that the individual memory devices can be addresses without the other devices conflicting. The HC11E9 contains 512 bytes of RAM, 512 bytes of EEPROM and 12K bytes of ROM.

The RAM is general-purpose volatile memory that is available for temporary storage of data and programs. By default, it is located at $0000–$01FF of the memory map. It can be moved to the beginning of any 4K block within the 64K memory map. A large portion of the RAM is used by the BUFFALO monitor program on the EVBU and is not available to the user. In other respects it acts like ROM. It is located at $B600–$B7FF in the memory map. The entire EEPROM is available to the user for programs and data. The ROM is burned during the manufacturing process. It contains the BUFFALO monitor program and interrupt vectors. It is located at $D000–$FFFF within the memory map. The EEPROM and ROM cannot be relocated within the memory map, but they can be deactivated with control bits in the CONFIG register.

In addition to the traditional memory components, the HC11 contains a register block of 64 registers. These registers are addressed in the same manner as the RAM, EEPROM and ROM. They also provide critical control of many HC11 system functions. Further information about the registers is provided in the appendix, as well as in other sections of the text.

Chapter Questions

Section 8.1

1. Why is each building on a street required to have a unique address number?
2. How many bits are typically stored at each memory location?
3. Can an electronic memory ever be empty?

Section 8.2

4. Which type of memory is present on all versions of the HC11?
5. How much RAM is contained on the A8 version of the HC11?
6. Why are there four versions within the "A" series of the HC11?

Section 8.3

7. What is the main difference between static and dynamic RAM?
8. What type of RAM is present on the HC11?
9. What value must be loaded into the INIT register to relocate the RAM to $D000–$D1FF and leave the register block in the default location?
10. What happens if the on-chip RAM is mapped to the same space that is occupied by the ROM?
11. Does the BUFFALO monitor program require RAM?

Section 8.4

12. Can the on-chip EEPROM be deactivated? If so, how?
13. What is the name of the control register used for programming the EEPROM? What is the default location of this register in the memory map?

14. Why does the EEPROM require the system clock?
15. Does BUFFALO use the EEPROM?
16. Fully explain what will happen if the control word %00000110 is written to the PPROG register.

Section 8.5

17. What type of information is typically loaded into the on-chip ROM?
18. What occupies the last 42 bytes of the ROM on the HC11?
19. Where is BUFFALO located in the ROM?

Section 8.6

20. How many bytes are occupied by the system registers on the HC11?
21. What value must be loaded into the INIT register to relocate the system registers to $B000–$B03F and leave the RAM in the default location?
22. What happens if the on-chip system registers are mapped to the same space that is occupied by the RAM?
23. Does the BUFFALO monitor program require the memory registers?

Section 8.7

24. Can additional memory be added to the HC11?
25. What hardware processor mode is required to access external memory devices?
26. How are address and data sent to the external memory devices?

Chapter Problems

1. Write a small program that will load the data $AA into EEPROM memory location $B63D.
2. Write a program to copy the contents of $0000–$000F into the EEPROM at $B730–$B73F. Use row erase to erase those two rows of the EEPROM before the copy operation. Write all code from scratch.
3. Rewrite the program from problem 2, utilizing the BUFFALO subroutines for all EEPROM programming tasks.

Answers to Self-Test Questions

Section 8.1

1. An address is a unique identifier for a memory location.
2. Yes. Addresses can contain more than one part. In the same way a street address contains a number, a street name and an apartment number, the addresses on computers contain multiple parts. The address $1002 might represent unit #$02 in building $10.
3. Self-storage facilities contain rows of buildings, where each building contains more than one storage unit. Computer memories often contain more than one device, where each device contains multiple storage locations.

Section 8.2

1. Yes. All versions contain at least some RAM.

2. The HC11K4 contains 768 bytes of RAM.
3. The HC711 versions contain EPROM; the equivalent HC11 versions contain ROM.

Section 8.3

1. The default EVBU RAM location is $0000. This location can be changed by writing to the upper four bits of the INIT register.
2. The BUFFALO monitor program uses the RAM for temporary storage of variables and data needed by the monitor program. The Vector Jump Table is also contained in RAM.

Section 8.4

1. The EEPROM can be erased one byte at a time (Byte Erase), one row of 16 bytes at a time (Row Erase) or the entire EEPROM (Bulk Erase).
2. Erase the location. Set the EELAT bit in the PPROG to activate the charge pump. Perform a write operation to tell the EEPROM hardware which location will be accessed. Set the EPGM bit to start the programming of the location. Wait 10 ms. Clear the PPROG register.
3. Write to the Block Protect Control bits in the BPROT register.

Section 8.5

1. ROM must be masked during the manufacturing process.
2. Yes, by the ROMON bit in the CONFIG register.

Section 8.6

1. The default memory location of the registers is $1000–$103F. This location can be changed by writing to the lower four bits of the INIT register.
2. The system register block contains data, control and status registers used by the on-chip hardware resources.

Section 8.7

1. They become the address strobe (AS) and read/write (R/W) control lines for external memory.
2. PORTB is converted into the upper byte of the address bus. Port C becomes the lower byte of the address byte time multiplexed as the data bus.
3. Address decoding is necessary to properly select the external memory device within the memory space of the HC11.

chapter 9

General Purpose I/O

Objectives

After completing this chapter, you should be able to:

- Write a code to output data onto the Port B pins
- Program the data direction of the Port C bits
- Use Port C for input and output of data
- Input and output data using simple strobed mode
- Input data from an external system using full-input handshaking
- Output data to an external system using full-output handshaking
- Program the direction of the Port D pins and use them for digital I/O
- Receive parallel data at the HC11 from an external device via Port E

Outline

Introduction

As a microcontroller, the HC11 will be connected to sensors and controls. It will collect data from the sensors and output signals to the controls to manage a machine process or the function of a device. These external devices are connected to the HC11 via ports.

Port	Input Pins	Output Pins	Bidirectional Pins	Shared Functions
Port A	3	3	2	Timer
Port B	-	8	-	High-Order Address in Expanded Mode
Port C	-	-	8	Low-Order Address and Data Bus in Expanded Mode
Port D	-	-	6	SCI and SPI
Port E	8	-	-	A/D Converter

Figure 9.1 I/O Port Capabilities (*courtesy of Motorola*)

The **ports** provide an interface between the processor and the outside world and support a variety of input/output functions.

There are five Input/Output (I/O) ports on the HC11. Each of these ports supports 8 bits of general-purpose digital I/O, except PORTD, which has only 6 bits defined. There are 11 dedicated input pins, 11 dedicated output pins and 16 bidirectional pins, as shown in Figure 9.1. Many features from the Motorola PIA and ACIA peripheral chips have been integrated into these five ports.

This chapter directly correlates to section 7 of the HC11 Reference Manual and Section 6 of the Technical Data Manual.

9.1 Port B—Output Only

Port B is a general-purpose digital output port. It consists of an 8-bit data register and eight Port B pins. It has the job of outputting the digital data contained in the Port B register (PORTB) to the output pins in parallel form, as shown in Figure 9.2. In single-chip mode, the function of the Port B pins is limited to digital I/O. However, in expanded mode, the Port B pins are used as the upper byte of the 16-bit external address bus.

The Port B register (PORTB) is located in the 64-byte register block. It is accessed via read and write operations to memory address $1004, as shown in Figure 9.3. Data is

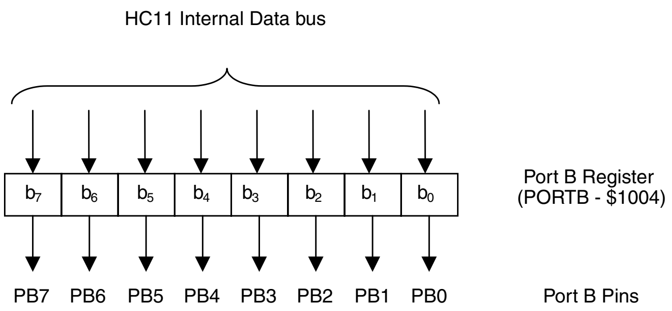


Figure 9.2 Port B Function Block Diagram

PORTB	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1004	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
DIGITAL I/O	Out	Out	Out	Out	Out	Out	Out	Out
RESET	0	0	0	0	0	0	0	0

Figure 9.3 Port B Register (PORTB) Pin Assignments

output to the Port B pins following a write to the port B register. A read of the Port B register returns the last data written to the register and has no effect on the register contents or on the data present on the Port B pins. The Port B pins are named PB7–PB0. The Port B register and the Port B pins are cleared after reset.

Example 9.1

Problem: Write a program that will add the data stored at locations $0000 and $0001 and output the result on the port B pins. Start the program at location $0100.

```
0000  81 22
```

Solution: Since the input data is located in memory locations that start with $00, the direct mode can be used to load and add them. The result can be directly stored to PORTB by writing to location $1004. When the data is written to PORTB, the data appears on the Port B output pins. Using transfer notation, this program accomplishes the following:

```
($0000) + ($0001) → $1004, $81 + $22 = $A3 → $1004

                ORG   $0100 ;Start program at $0100
0100  96 00     LDAA  $00   ;Get first number ($0000) → A
0102  9B 01     ADDA  $01   ;Add 2nd to 1st (A) + ($0000) → A
0104  B7 10 04  STAA  $1004 ;Store result (A) → ($1004)
0107  3F        SWI         ;Stop
```

Self-Test Questions 9.1

1. What is the address of the Port B register?
2. Is Port B an input or an output port?
3. What are the names of the Port B pins?

9.2 Port C—Programmable I/O

Port C is a general-purpose bidirectional port. It consists of an 8-bit data register (PORTC), an 8-bit data direction register (DDRC) and eight Port C pins. It has the job

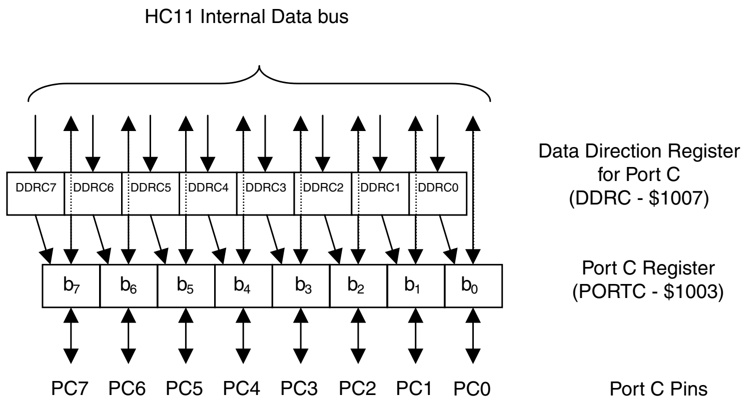

HC11 Internal Data bus
DDRC7
DDRC6
DDRC5
DDRC4
DDRC3
DDRC2
DDRC1
DDRC0
Data Direction Register for Port C (DDRC - $1007)
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
Port C Register (PORTC - $1003)
PC7 PC6 PC5 PC4 PC3 PC2 PC1 PC0
Port C Pins

Figure 9.4 Port C Functional Block Diagram

of outputting the digital data contained in the Port C register (PORTC) to the output pins or inputting the data from the Port C pins to the Port C register. Each operation is done in parallel form, as shown in Figure 9.4. The DDRC register controls the direction of the Port C register bits. Each bit in the DDRC controls the direction of the corresponding bit in the Port C register. If DDRCx = 0, the corresponding bit in the Port C register is configured as an input. If DDRCx = 1, the corresponding bit in the Port C register is configured as an output. Any combination of input and output functions can be programmed without regard to the order of the bits.

The Port C and DDRC registers are located in the 64-byte register block. They are accessed via memory addresses $1003 and $1007 respectively. Data is output to the Port C pins following a write to the Port C register. Bits that have been programmed for input operation are unaffected by a write operation. A read of the Port C register returns the data present on the Port C pins that have been programmed as inputs. The last data written to the output bits of the register will be read back. Read operations have no effect on the register contents or on the data present on the Port C pins. The Port C pins are named PC7–PC0, as shown in Figure 9.5. The Port C register is in an undefined state after reset. The DDRC register is cleared after reset, which causes the Port C register to be configured as eight input bits.

PORTC	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1003	PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0
DIGITAL I/O	In/Out	In/Out	In/Out	In/Out	In/Out	In/Out	In/Out	In/Out
RESET	U	U	U	U	U	U	U	U

DDRC	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1007	DDRC7	DDRC6	DDRC5	DDRC4	DDRC3	DDRC2	DDRC1	DDRC0
RESET	0	0	0	0	0	0	0	0

Figure 9.5 Port C Pin Assignments and Data Direction Control Register

Example 9.2

Problem: Write a program that will program the Port C pins for input, read the data from PORTC and output it to PORTB. Start the program at location $0120.

Solution: Since the Port C pins are programmable, a control word must be written to the DDRC register to program the pins. 0's in the control word cause the corresponding Port C pins to be input pins. The first thing done in the program is to clear the DDRC register, which programs the Port C pins as inputs. Then PORTC is read, and this data is directly written to PORTB.

```
                 ORG    $0120 ;Start program at $0120
0100  7F 10 07   CLR    $1007 ;Program Port C for Input
0103  D6 10 03   LDAB   $1003 ;Get Port C data ($1003) → A
0106  F7 10 04   STAB   $1004 ;Store to Port B (A) → ($1004)
0109  3F         SWI          ;Stop
```

Example 9.3

Problem: Build the control word and write the lines of code necessary to program the upper nibble of Port C for input and the lower nibble for output.

Solution: 0's in the control word cause the corresponding Port C pins to be input pins and 1's cause the corresponding pins to be outputs. Thus, the control word is %00001111. This value must be written to the DDRC register. The binary format is used to show the specific programming of each bit.

```
86 0F       LDAA    #%00001111  ;Build DDRC control word
B7 10 03    STAA    $1003       ;Program Port C
```

Self-Test Questions 9.2

1. Can each bit of Port C be programmed independently of the other bits?
2. What is the name of the register that controls the direction of data flow through the Port C register?
3. What is the address of the Port C register?

9.3 Bit-Level Operations

The HC11 contains several instructions that are concerned with *groups of bits* rather than *bytes or 16-bit words.* These instructions provide the ability to change individual bits within data bytes, perform comparison operations at a bit level and cause branching based on the condition of individual bits. The discussion of these instructions is presented at this point in the text because these instructions are often used in conjunction with the I/O ports.

Data Masking

All bit-level instructions require a mask to identify the data bits that are to be processed. A **mask** is an 8-bit word that designates which bits within the operand will be processed. Ones (1) in the mask indicate bit positions that will be processed. Zeros (0) in the mask indicate bit positions that are ignored. A data mask blocks certain data bits from being considered during the execution of a bit-level instruction, in the same manner in which a mask is used to cover features of a character's face.

Masks are used for different purposes, as shown in Figure 9.6. In some cases, a mask is worn for protection. A catcher on a baseball team and a goalie on a hockey team wear masks to protect their faces from the ball or puck. Masks can also be used to hide something. A bank robber wears a mask to prevent identification by covering facial features. Not all of the information regarding the bank robber's identity can be seen through the mask. Masks can also act as filters and block out irrelevant data. A photographer uses filters on the end of a camera lens to create special affects. Some of these filters fade the edges or create special looking frames around a subject.

Masks are used by the microprocessor to accomplish similar purposes. When data is being changed, a mask can be used to identify which data bits to change and which data bits to ignore. The data masking protects some of the bits and allows access to others. The mask can create focus so that a decision is based on only some of the bits in a data word. Figure 9.7(a) shows how three bits can be filtered out of the data word for processing. The mask contains 1's in b_6, b_5, and b_0, and 0's in all other positions. Thus, only the bits in b_6, b_5 and b_0 of the data word are processed. Figure 9.7(b) shows how to mask out the upper nibble of a data word.

The logical AND operation can be used to mask data. By ANDing a data word with a mask, unwanted data can be eliminated, as shown in Figure 9.8.

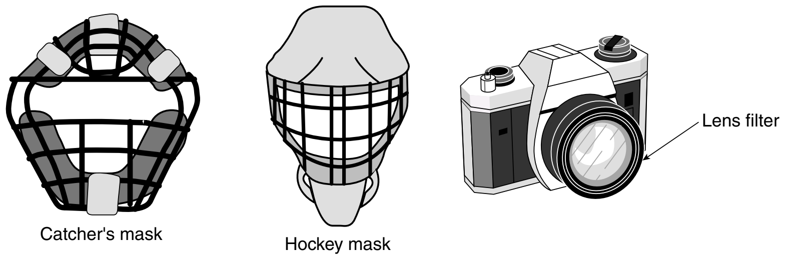


Figure 9.6 Examples of Masks

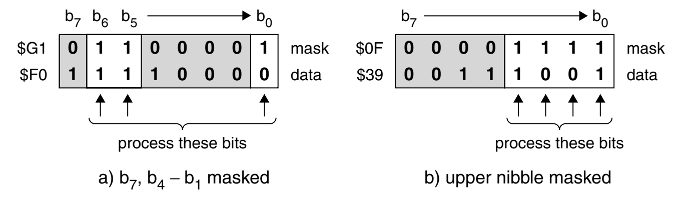


Figure 9.7 Data Masks

The HC11 has six instruction mnemonics that are specifically designed for bit-level operations. Two are called bit manipulation instructions because they are designed to change bits within a data word. Another two are designed to perform bit-level comparisons, and the last two will perform bit-level conditional branches.

Example 9.4

Problem: Build the masks necessary to isolate the following groups of bits and ignore all other bits within an 8-bit data word:

a. Upper nibble
b. Odd bits
c. b_7, b_2, b_0
d. b_7, b_6, b_4, b_3, b_1

Solution: Bits that will be isolated will have 1's in the corresponding bit positions of the mask and the bits that will be ignored will have 0's.

a. %11110000 or $F0 would isolate the upper nibble and ignore the lower nibble.
b. %10101010 or $AA would isolate the odd bits and ignore the even bits.
c. %10000101 or $85 would isolate b_7, b_2, b_0 and ignore the rest.
d. %11011010 or $DA would isolate b_7, b_6, b_4, b_3, b_1 and ignore the rest.

Bit Manipulation

The HC11 contains instructions that allow one or more data bits within a data word to be set or cleared. The operation of these instructions is summarized in Figure 9.9. The mask is used in this case to designate which bits within the data word will be affected. The BSET instruction will set the designated bits within a data word stored at a memory location. It does this by ORing the operand with the mask. Thus, all bits that are ones in the mask will be set in the memory location after the BSET operation. All bits that are zeros in the mask are unaffected.

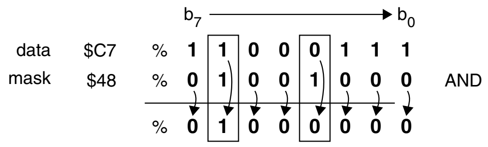


Figure 9.8 Masking Using the AND Operation.

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
BSET	Set bit(s) in contents of memory	(M)+mm ⇒ (M)	-	X	-	X	X	-	-	-	-	-	-	↕	↕	0	-
BCLR	Clear bit(s) in contents of memory	(M)•$\overline{\text{mm}}$ ⇒ (M)	-	X	-	X	X	-	-	-	-	-	-	↕	↕	0	-

Figure 9.9 Bit Manipulation Instructions

The BCLR instruction will clear the designated data bits within a data word stored at a memory location. It does this by ANDing the operand with the inverse of the mask. Thus, all bits that are 1's in the mask will be clear in the memory location after the BCLR operation. All bits that are 0's in the mask are unaffected. These two instructions operate in the DIR, INDX and INDY addressing modes. They affect only three status flags in the CCR: N and Z reflect the actual condition of the data that was loaded. The V bit is cleared, indicating that the sign (N flag) is correct.

Example 9.5

Problem: Using bit manipulation instructions, perform the following functions. Assume the Y register contains $1000 and the X register contains $0100.

a. Clear the lower nibble of PORTB.
b. Program PORTC so that the upper nibble is output and the lower nibble is input.
c. b_7, b_6, b_0 of $002E will be set.
d. b_7–b_1 of $01FF will be cleared and b_0 will be set.

Solution: Bits that will be isolated will have 1's in the corresponding bit positions of the mask and bits that will be ignored will have 0's.

a. %00001111 or $0F will isolate the lower nibble. Index Y mode is used to address PORTB because the extended mode is not supported by the BCLR instruction.

```
BCLR  $04,Y $0F
```

b. %11110000 or $F0 will isolate the upper nibble. The DDRC register is cleared first to make all bits inputs, and then the upper nibble is set, making them outputs.

```
CLR   $03,Y
BSET  $03,Y $FO
```

c. %11000001 or $C1 will isolate b_7, b_6, b_0 and ignore the rest. Since $002E uses a zero high byte, the direct mode can be used.

```
BSET  $2E $C1
```

d. %11111110 or $FE will isolate b_7–b_1 and ignore the rest.

```
BCLR  $FF,X $FE
BSET  $FF,X $01
```

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
BITA BITB	Tests bit(s) in contents of memory	A • (M) B • (M)	X	X	X	X	X	-	-	-	-	-	-	↕	↕	0	-

Figure 9.10 Bit-Level Comparison Instructions

Bit-Level Comparison

There is another set of special compare instructions. They are capable of performing bit-level comparisons. These instructions are called bit tests. The operation of the bit-level compare instructions is summarized in Figure 9.10. They are used primarily to test for zero conditions in a series of bits and therefore have results similar to the test instructions. Rather than a subtraction, BITA and BITB perform the logical AND on the data in one of the 8-bit accumulators and a byte from a memory location. This operation masks or isolates the groups of bits under test. The result of the AND operation updates the flags in the CCR. BITA and BITB operate in the IMM, DIR, EXT, INDX and INDY addressing modes. They affect only three status flags in the CCR: N and Z reflect the actual condition of the data that was loaded. The V bit is cleared, indicating that the sign (N flag) is correct.

Example 9.6

Problem: Using bit-test instructions, perform the following functions and indicate the resulting condition codes. Assume the X register contains $0180.

```
01E0  81 22 AA 55 F0 3C 7E 1F A = 14  B = 82  CCR = D0
```

a. `BITA #$37`
b. `BITB $01E2`
c. `BITB $66,X`

Solution:

a. (A) AND $37 = $14 AND $37 = $00, thus N = 0, Z = 1, V = 0, CCR = $D4
b. (B) AND ($01E2) = $82 AND $AA = $80, thus N = 1, Z = 0, V = 0, CCR = $D8
c. (B) AND ($01E6) = $82 AND $7E = $02, thus N = 0, Z = 0, V = 0, CCR = $D0

Bit-Level Conditional Branching

There are two conditional branch instructions that perform the branch test on one or more bits contained in a location of memory. The operation of these instructions is summarized in Figure 9.11. These are special branch instructions because they use the relative mode only for the branch calculation. These two instructions also use either the direct or the indexed addressing mode to access the operand and the immediate addressing mode to retrieve the mask from memory.

BRSET will cause a relative mode branch if all of the bits designated in the mask are set. The instruction accomplishes this by ANDing the mask with the inverse of the operand stored at the memory location. The branch test is then performed on the

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
BRSET	Branch if bit(s) are set	If $(\overline{M}) \bullet mm = 0$ then P + ssrr = P	-	X	-	X	X	-	-	-	-	-	-	-	-	-	-
BRCLR	Branch if bit(s) are clear	If $(M) \bullet mm = 0$ then P + ssrr = P	-	X	-	X	X	-	-	-	-	-	-	-	-	-	-

Figure 9.11 Bit-Level Conditional Branch Instructions

result. In this case, the result must be zero for the branch to take place. The result is discarded in either case. No data is permanently modified by this instruction, and it has no effect on the CCR.

BRCLR will cause a relative mode branch if all of the bits designated in the mask are cleared. The instruction accomplishes this by ANDing the mask with the operand stored at the memory location. The branch test is then performed on the result. In this case, the result must be zero for the branch to take place. The result is discarded in either case. No data is permanently modified by this instruction, and it has no effect on the CCR.

Example 9.7

Problem: Calculate the destination address for each of the following bit-level conditional branch instructions. Use the following data to evaluate each instruction.

```
0000  C6 84 F2 B1 29 35 B6 E0      X = 0000

Address      Machine Code  Source Code
a. 01D9      12 07 60 94   BRSET  $07 $60 $94
b. 0112      1F 03 43 2C   BRCLR  $03,X $43 $2C
```

Solution: In each case, the processor must determine if the branch test passes or fails. In addition, the contents of the PC must be determined. If the test fails, then DA = PC. If the test passes, then the sign-extended relative address must be added to the PC to calculate the destination address (Equation 4.1). Since each of these instructions occupies four bytes of memory, the PC will be four more than the address of the branch instruction.

a. The first step is to read the operand from memory. In this case, the DIR mode is used to address the operand. The direct mode address $07 translates to $0007. Thus the operand is $E0. Next, the mask is read from memory ($60). The binary value of the mask contains two 1's in bits b_6 and b_5. The compare is checking to see if these two bits are set in the operand (which they are). It does this by ANDing the operand inverted with the mask, as shown. Since the result of this operation is zero (i.e., bits b_6 and b_5 were set), the branch test passes. The resulting destination address is calculated, as shown.

```
 Operand $E0   %11100000 → %00011111 (inverted)
 Mask    $60   %01100000 → %01100000
                         AND %00000000
```

```
PC = $01D9 + 4        →   $01DD
rr = $94, sign extend →  +$FF94
DA = PC + rr          →   $0171
```

b. The first step is to read the operand from memory. In this case, the INDX mode is used to address the operand. The effective address is the sum of the contents of X plus the offset ($0000 + $03), which translates to $0003. Thus the operand is $B1. Next, the mask is read from memory ($43). The binary value of the mask contains three 1's in bits b_6, b_1 and b_0. The compare is checking to see if these three bits are cleared in the operand (b_0 is not). It does this by ANDing the operand with the mask as shown. Since the result of this operation is not zero (i.e., bits b_0 was set), the branch test fails. The resulting destination address is calculated, as shown.

```
Operand $B1   %10110001
Mask    $43   %01000011
              %00000001
```

```
DA = PC = $0112 + 4 → $0116
```

Self-Test Questions 9.3

1. What is a mask?
2. What bit-level operations are supported by the HC11?
3. Which instructions provide bit-level conditional branching?

9.4 I/O with Handshaking

The HC11 provides parallel I/O with handshaking. The term **handshaking** is used to describe a method of data transfer that uses control signals between the computer and the peripheral device. The handshaking subsystem consists of PORTB, Port C and two control lines, strobe A (STRA) and strobe B (STRB). There are two methods of handshaking supported on the HC11, simple and full. The various modes and functions are selected and controlled by the bits in the Parallel Input/Output Control (PIOC) register. The layout of this register is shown in Figure 9.12. The Handshake Mode Select (HNDS) and Output or Input Handshake Select (OIN) bits control which mode has been selected. Figure 9.13 summarizes how the various modes are selected.

Simple Handshaking

Simple handshaking is called **simple strobed mode**. The sending device uses a strobe to indicate to the receiving device that the data present on the parallel bus is valid.

PIOC	b_7	b_6	b_5	b_4	b_3	B_2	b_1	b_0
$1002	STAF	STAI	CWOM	HNDS	OIN	PLS	EGA	INVB
RESET	0	0	0	0	0	U	1	1

Figure 9.12 Parallel I/O Control Register (PIOC)

HNDS	OIN	Mode Description
0	X	Simple Strobed Mode
1	0	Full-input Handshaking
1	1	Full-output Handshaking

Figure 9.13 Parallel I/O Mode Selection Summary

The HC11 can simultaneously send and receive data, because the simple strobed mode send and receive functions are managed by independent hardware. Figure 9.14 illustrates how this works.

The simple strobed mode is controlled by the bits in the Parallel Input/Output Control (PIOC) register, as shown in Figure 9.12. Five of the eight PIOC bits are used in the simple strobed mode. Simple strobe mode is selected by the HNDS bit of the PIOC register. When HNDS = 0, simple strobed mode is selected. Data is output using simple strobed mode via Port B. The user writes data to the Port B register. This data is transferred to the Port B pins by the HC11 hardware, then a pulse (the strobe) is generated on the Strobe B (STRB) pin. This pulse is two E-clocks in length. The polarity of the STRB pulse is controlled by the Invert Strobe B (INVB) bit in the PIOC register. When INVB = 1, the active state of the pulse is logic one. When INVB = 0, the active state of the pulse is logic zero.

Data is input using simple strobed mode via the Port C Latched (PORTCL) register. An external device writes data to the Port C pins. This data is transferred to the PORTCL register when an edge is received on the Strobe A (STRA) pin. The PORTCL register is shown in Figure 9.15.

The polarity of the STRA edge is controlled by the Active Edge for Strobe A (EGA) bit in the PIOC register. When EGA = 1, STRA is active on the rising edge. When EGA = 0, STRA is active on the falling edge. The Strobe A Flag (STAF) bit of the PIOC register

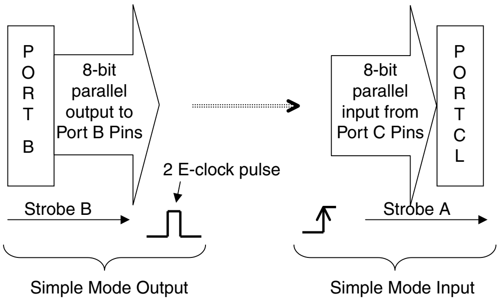


Figure 9.14 Simple Strobed Mode

PORTCL	b7	b6	b5	b4	b3	b2	b1	b0
$1005	PCL7	PCL6	PCL5	PCL4	PCL3	PCL2	PCL1	PCL0
RESET	U	U	U	U	U	U	U	U

Figure 9.15 Port C Latched (PORTCL) Pin Assignments

indicates that a valid edge has been detected on STRA and the data has been latched into the PORTCL register. In simple strobed mode, the STAF bit is cleared by reading the PIOC register followed by a read of the PORTCL register.

NOTE: Since the Port C pins are configured for input during simple strobed mode, the corresponding pins of the DDRC register must be cleared.

Simple strobed mode also supports interrupts. The STAI is a local interrupt enable bit. When STAI = 1, an interrupt will be requested each time the STAF bit is set. If STAI = 0, no interrupt will be requested when STAF bit is set.

See Chapter 10 for further explanation of the operation of interrupts.

Example 9.8

Problem: Build the control word to program the I/O system for simple handshake mode. Let the control lines remain at their default state. Then write the code to configure the hardware and cause the output of the data $48.

Solution: Since the default state of the I/O system is simple handshake mode, then the control word needs to be equal to the reset state of the PIOC register. Thus, the control word is %00000011. The OIN and PLS bits are cleared, but they have no meaning in this mode.

```
86 0F      LDAA   #%00000011   ;Build PIOC control word
B7 10 02   STAA   $1002  ~     ;Program PIOC
86 48      LDAA   #$48 ~       ;Load data to transfer
B7 10 04   STAA   $1004  ~     ;Send it via PORTB
```

Full Handshaking

Full handshaking is available for input or output operations. Full handshaking uses Port C and the STRA and STRB control lines. Figure 9.16 illustrates how this works.

The full handshake modes are controlled by bits in the PIOC register. Seven of the eight PIOC bits are used in the full handshake modes, as shown in Figure 9.12. Full handshake mode is selected by the HNDS bit of the PIOC register. When HNDS = 1, full handshake mode is selected. The Output or Input handshake Select (OIN) bit of the PIOC register selects input or output full handshake mode. When OIN = 0, full-input handshaking is selected. When OIN = 1, full-output handshaking is selected.

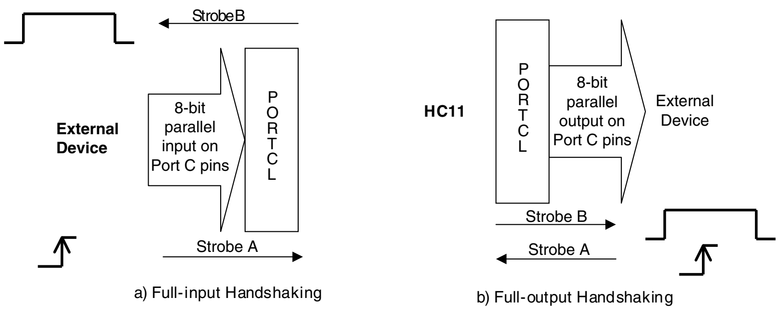

StrobeB
External Device
8-bit parallel input on Port C pins
PORTCL
Strobe A
a) Full-input Handshaking
HC11
PORTCL
8-bit parallel output on Port C pins
External Device
Strobe B
Strobe A
b) Full-output Handshaking

Figure 9.16 Full Handshake Modes

Full-input Handshaking

In **full-input handshaking** mode, the Port C pins are used to receive data from an external device. STRB acts as a ready signal to the external device. The external device waits for the STRB signal to be asserted, which indicates that the HC11 is ready to receive data. The external device then sends the data to the Port C pins and asserts the STRA signal, indicating that the data is valid at the Port C pins. The HC11 recognizes the STRA input and latches the data into the PORTCL register. Once the data is latched into PORTCL, the HC11 automatically removes the STRB ready signal, indicating to the external device that it is not ready for a data transfer.

NOTE: In full-input handshake mode the STRB indicates that the HC11 is ready to receive data. STRB is asserted by reading the PORTCL register (independent of clearing the STAF bit).

The polarity of the STRB pulse is controlled by the INVB bit in the PIOC register. When INVB = 1, the active state of the STRB signal is logic one. When INVB = 0, the active state of the STRB signal is logic zero. The polarity of the STRA edge is controlled by the EGA bit in the PIOC register. When EGA = 1, STRA is active on the rising edge. When EGA = 0, STRA is active on the falling edge. The Strobe A Flag bit (STAF) of the PIOC register indicates that a valid edge has been detected on STRA and the data has been latched into the PORTCL register. In full-input handshake mode, the STAF bit is cleared by reading the PIOC register followed by a read of the PORTCL register. A read of PORTCL also reasserts STRB, starting another full-input handshake operation.

NOTE: Since the Port C pins are configured for input during full-input handshaking, the corresponding pins of the DDRC register must be cleared.

Example 9.9

Problem: Build the control word to program the I/O system for full-input handshake mode. Let the control lines remain at their default state, except make the STRA to activate on a falling edge and cause pulsed handshaking. Then write the code to configure the hardware and cause the input of the data to be stored in location $0000.

Solution: Full handshaking is selected by HNDS = 1, input mode by OIN = 0. STRA falling edge is selected by EGA = 0. PLS will be set to cause pulsed handshaking. The default state of STAI, CWOM and INVB will be used.

```
86 0F              LDAA   #%00010101 ;Build PIOC control word
B7 10 02           STAA   $1002      ;Program PIOC
B6 10 02   CHECK   LDAA   $1002      ;wait for STAF
2A FB              BPL    CHECK      ;If not STAF, check again
B6 10 05           LDAA   $1005      ;Get input data
97 00              STAA   $00        ;Store input data
```

Full-output Handshaking

In **full-output handshaking** mode the Port C pins are used to output data to an external device. The output operation is started by writing to PORTCL. The STRB signal is automatically asserted by the write to PORTCL. This data is then transferred to the Port C pins. The external device recognizes the STRB signal as a ready signal and latches the data. Once the data is latched by the external device, it asserts the STRA signal indicating to the HC11 that the data has been received. The active edge on STRA automatically removes the STRB signal and sets the STAF bit. The polarity of the STRB pulse is controlled by the INVB bit in the PIOC register. When INVB = 1, the active state of the signal is logic one. When INVB = 0, the active state of the signal is logic zero.

The polarity of the STRA edge is controlled by the EGA bit in the PIOC. When EGA = 1, STRA is active on the rising edge. When EGA = 0, STRA is active on the falling edge. In full-output handshake mode, the STAF bit is cleared by reading the PIOC followed by a write to PORTCL.

The full handshake modes also support interrupts. The STAI is a local interrupt enable bit. When STAI = 1, an interrupt will be requested each time the STAF bit is set. If STAI = 0, no interrupt will be requested when STAF bit is set.

NOTE: Since the Port C pins are configured for output during full-output handshaking, the corresponding pins of the DDRC register must be set.

Example 9.10

Problem: Build the control word to program the I/O system for full-output handshake mode. Let the control lines remain at their default state, except make the STRA to activate on a falling edge and cause pulsed handshaking. Then write the code to configure the hardware and cause the output of the data $9F.

Solution: Full handshaking is selected by HNDS = 1, output mode by OIN = 1. STRA falling edge is selected by EGA = 0. PLS will be set to cause pulsed handshaking. The default state of STAI, CWOM and INVB will be used.

```
C6 0F              LDAB    #%00011101 ;Build PIOC control word
F7 10 02           STAB    $1002      ;Program PIOC
C6 9F              LDAB    #$9F       ;Get output data
F7 10 05           STAB    $1005      ;output to PORTCL
F6 10 02   CHECK   LDAB    $1002      ;wait for STAF
2A FB              BPL     CHECK      ;If not STAF, check again
```

Self-Test Questions 9.4

1. What are the two types of handshaking supported on the HC11?
2. Which port is used for data input using simple strobed mode?
3. What two types of full handshaking are supported?

9.5 General-Purpose I/O On Other Ports

The HC11 contains three additional I/O ports: PORTA, PORTD and PORTE. These three ports support standard general-purpose I/O as well as alternative functions. The general-purpose I/O functions are described in this chapter.

Port A

Port A is an 8-bit I/O port. The eight general-purpose I/O bits are labeled PA7–PA0. Three bits are dedicated outputs (PA6, PA5 and PA4), three bits are dedicated inputs (PA2, PA1 and PA0) and two bits are programmable input or output (PA7 and PA3), as shown in Figure 9.17.

The programmable bits are controlled by two data direction control bits. DDRA7 controls the data direction of Port A b7 (PA7), and DDRA3 controls the data direction of PORTA b3. The DDRAx control bits are in the Pulse Accumulator Control (PACTL) register and are only remotely related to the pulse accumulator function. When the

PORTA	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1000	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
Digital I/O	In/Out	Out	Out	Out	In/Out	In	In	In
Reset	0	0	0	0	0	0	0	0

PACTL	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1026	DDRA7	PAEN	PAMOD	PEDGE	DDRA3	I4/O5	RTR1	RTR0
Reset	0	0	0	0	0	0	0	0

Figure 9.17 PORTA Pin Assignments and Data Direction Control Bits

PORTD	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1008	–	–	PD5	PD4	PD3	PD2	PD1	PD0
Digital I/O	–	–	In/Out	In/Out	In/Out	In/Out	In/Out	In/Out
Reset	0	0	0	0	0	0	0	0

DDRD	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1009	–	–	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0
Reset	0	0	0	0	0	0	0	0

Figure 9.18 PORTD Pin Assignments and Data Direction Control Bits

DDRAx bit = 0, the corresponding I/O pin is configured as an input. When the DDRAx bit = 1, the corresponding I/O pin is configured as an output. After reset, the bits in PORTA are cleared and PA7 and PA3 are configured as input pins. The Port A pins are alternatively used as the on-chip timing system.

The Port A pins are alternatively used for the Port A timing system, which is described in Chapter 12.

Port D

Port D is a 8-bit I/O port. Six bits are programmable input or output (PD5 through PD0), and the two most significant bits are unused. This configuration is shown in Figure 9.18. The programmable bits are controlled by data direction control bits in the Data Direction register for Port D (DDRD). When the DDRDx bit = 0, the corresponding I/O pin is configured as an input. When the DDRDx bit = 1, the corresponding I/O pin is configured as an input. After reset, the bits in PORTD are cleared and are configured as input pins.

The PORTD pins are alternatively used as the on-chip serial communication functions, which are presented in Chapter 13.

Port E

Port E is an 8-bit input port. The eight general-purpose input pins are labeled PE7–PE0, as shown in Figure 9.19. After reset, the bits in the Port E register (PORTE) are cleared.

The Port E pins are alternatively used as analog input pins connected to the on-chip analog to digital converter. The analog-to-digital conversion function is presented in Chapter 11.

PORTE	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$100A	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0
Digital I/O	In	In	In	In	In	In	In	In
Reset	U	U	U	U	U	U	U	U

Figure 9.19 PORTE Pin Assignments

Self-Test Questions 9.5

1. How are the PORTA pins configured for general-purpose I/O?
2. When DDRD bits contain logic '1', how are the corresponding PDx bits configured?
3. How are the PORTE pins configured for general-purpose I/O?

Summary

There are five Input/Output (I/O) ports on the HC11. Thirty-eight general-purpose I/O pins are supported. PORTB and Port C support two types of handshaking for interface control, simple strobed mode and full handshaking. The PIOC control register contains control and status flag bits relevant to the handshaking operations. The function of the STRA and STRB control lines were also discussed. PORTA, PORTD and PORTE support alternative functions unrelated to the general-purpose I/O capabilities.

Chapter Questions

Section 9.1

1. What register is located at $1004 in the memory map?
2. Which port is a dedicated output port?
3. Which bit of PORTB is the pin PB4 associated with?

Section 9.2

4. What is the function of a data direction register?
5. What register is located at $1007 in the memory map?
6. What value must be written to the DDRC to program the three LSBs as outputs and the other pins as inputs?
7. What is the default direction of the Port C pins?

Section 9.3

8. What does the term "mask" mean? What do 1's and 0's represent in a data mask?
9. What 8-bit mask is needed to mask all bits except b_7, b_6 and b_2?
10. What 8-bit mask is necessary to hide the upper nibble of a data word?
11. What is the difference between the AND instruction and a bit test instruction?
12. What mask is necessary in a BITA instruction to simulate the operation of TSTA?
13. What mask is necessary in a BCLR instruction to clear the two MSBs?
14. Does the instruction BSET $34 have any effect on b_3?

Section 9.4

15. What is the difference between PORTC and PORTCL?
16. What are the functions of STRA and STRB?
17. When writing data into the PIOC register, why is the data in the MSB irrelevant?

18. What is one advantage of having the STAF bit in the MSB position of the PIOC register?
19. What is the difference between the simple strobed mode and full handshaking?
20. Fully explain the configuration of the handshaking system if the PIOC contains $17.

Section 9.5

21. What is the default configuration of the PORTA, PORTB, PORTC, PORTD and PORTE pins?
22. Which ports contain programmable I/O pins?
23. What is the total number of general-purpose I/O pins available on the HC11E9?

Chapter Problems

1. Write a code segment that will output the data $ED on the pins of PORTB.
2. Write a code segment that will output the data stored in memory location $0020 on the pins of PORTB.
3. Write a simple program that will turn on a light that is connected to PB0 when the sum of two numbers is negative.
4. Write a code segment that configures Port C so that the four MSBs are outputs and the four LSBs are inputs.
5. Write a code segment that configures the two MSBs and the LSB of Port C as outputs and the other bits as inputs.
6. Write a code segment that will recognize the logic level of a switch connected to PC7 and indicate the switch's logic level on an LED connected to PB7.
7. Write a code segment that will recognize the logic level of a switch connected to PA7 and indicate the switch's logic level on an LED connected to PA3.

Answers to Self-Test Questions

Section 9.1

1. The Port B register is located at $1004.
2. Port B is a dedicated output port.
3. The Port B pins are labeled PB7–PB0. PB7 is the MSB.

Section 9.2

1. Yes. The DDRC register allows each Port C pin to be configured independently as an input or output pin.
2. The DDRC register is the Data Direction Register for Port C.
3. The Port C register is located at $1003.

Section 9.3

1. A mask is an 8-bit word that designates which bits within the operand will be considered as part of the instruction.
2. There are six bit-level instructions supported on the HC11: BSET, BCLR, BITA, BITB, BRSET, BRCLR.

3. BRSET and BRCLR will perform branch tests at the bit level.

Section 9.4
1. The HC11 supports simple and full handshaking.
2. The PORTCL register is used to latch the input data in simple strobed mode.
3. The HC11 supports full-input handshaking and full-output handshaking.

Section 9.5
1. The three PA2–PA0 are dedicated inputs, PA6–PA4 are dedicated outputs and PA7 and PA3 are programmable I/O.
2. 1's in the DDRD register cause the PDx pins to be configured for output.
3. They are dedicated digital input pins.

chapter 10

HC11 Interrupts and Resets

Objectives

After completing this chapter, you should be able to:

- Describe the process of servicing an interrupt
- Enable and mask interrupts via global and local control bits
- Differentiate the strengths and weaknesses of maskable and nonmaskable interrupts
- Use the vector table and vector jump table
- Understand the impact of hardware reset on all hardware systems
- List the process of servicing a reset

Outline

Introduction

Resets and interrupts are often discussed together because they share a common function. Each must fetch a vector from memory to initialize the starting point of processing that follows each reset or interrupt.

The main purpose behind reset is to establish the initial conditions of the processor. Without the proper initial conditions, the processor cannot start processing instructions in an orderly fashion.

Interrupts are a powerful mechanism of computers designed to allow the more efficient use of system resources. Interrupts allow multiple tasks to happen in a completely random order, controlled by events external to the processor.

Most computer processors execute software instructions sequentially. As was described in chapter 2, a software program is nothing more than an ordered set of instructions that the processor executes sequentially. In many applications, it is necessary to execute sets of instructions in response to some external stimulus. Typically, this type of request comes from peripherals and is asynchronous to the program being executed by the processor. Thus, the external stimulus interrupts what the processor is doing and causes another set of instructions to be executed. When it is complete, it returns to what it was executing prior to the peripheral request as if there had been no interruption. Thus, an **interrupt** is an external stimulus that requests immediate attention from the processor.

The HC11 supports 21 independent interrupts. Six of these are classified as nonmaskable interrupts, and the remaining 15 are classified as maskable. In general, **maskable** interrupts can be controlled (enabled or inhibited) by the user at any time, and **nonmaskable** interrupts cannot be controlled by the user. The specifics surrounding the maskable and nonmaskable groups, as well as exceptions to the masking rule, are discussed later in this chapter.

This chapter directly correlates to section 5 of the HC11 Reference Manual and section 5 of the Technical Data Manual.

10.1 Condition Code Register Control Bits

As presented in chapter 1, the HC11 contains a processor register that is used for status and control. This register is called the Condition Code Register (CCR) and is part of the HC11 Programmer's Model. Chapter 3 provided a further study of the use of the five status flags (H, N, Z, V and C) contained in this register. This section offers explanation regarding the three control bits (S, X and I), as shown in Figure 10.1.

The **S** bit is the stop disable bit. It is a control bit and occupies b7 of the register. Setting this bit disables the STOP instruction from putting the HC11 into a low-power stop condition. It is set during processor reset to disable the STOP feature. When this bit is set, the STOP instruction is treated like an NOP instruction (an NOP is the "no operation" instruction that lets the processor idle for two machine cycles).

CCR	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
	S	X	H	I	N	Z	V	C
RESET	1	1	U	1	U	U	U	U

Figure 10.1 Condition Code Register Control Bits

Refer to section 5.5.2 of the Technical Data Manual, as well as to sections 5.2.3, 5.4.1 and 6.3.4.5 of the Reference Manual, for further information regarding the function of the S bit and STOP instruction.

The **X** bit is the XIRQ interrupt mask bit. The nonmaskable XIRQ hardware interrupt is disabled when this bit is set. It is set during processor reset to disable the XIRQ interrupt. The user can only clear this bit. Attempts to set the bit via software will be ignored.

Further explanation of the function of the X bit is provided in section 10.6.

The **I** bit is the IRQ interrupt mask bit. The maskable IRQ hardware interrupt is disabled when this bit is set. This bit can be directly manipulated by the CLI and SEI instructions. It is set during processor reset to disable IRQ interrupts.

Further explanation of the function of the I bit is provided in section 10.5.

Self-Test Questions 10.1

1. What must be written to the I bit in the CCR to enable maskable interrupts?
2. What bit position of the CCR contains the X bit?

10.2 Servicing an Interrupt

Interrupts are not unique to computer systems. Many aspects of life are filled with interrupts. For example, the best television programs are constantly interrupted with annoying commercials. A good night's sleep is interrupted by the sound of the alarm, and a quiet dinner at home is regularly interrupted by the telephone ringing. Each of these events (commercials, alarms and phone calls) is an external event that causes someone to stop what he or she is doing and do something else. They will be used throughout this section as examples to illustrate the concept of interrupts and how the processor reacts to these external stimuli.

The external stimulus is called an **interrupt request**. It signals the processor that an interrupt is pending. The ring of the telephone is the interrupt request. It is an audible signal that demands attention. The alarm on the alarm clock has the same effect. It demands attention immediately. The process of responding to an interrupt request is referred to as **servicing the interrupt**. These audio signals (ringing or buzzer) are designed to get someone's attention, and they work very effectively.

The act of responding to the interruption is almost automatic. When the phone rings during dinner, the fork is set down, the phone picked up and then a greeting is spoken, like "hello." This semiautomatic response to the telephone ring has little variation from day to day and from place to place. The telephone ring demands service, and rarely does it go without being serviced. In the same way, a computer has a fixed response to an interrupt. When an interrupt request is received by the computer, it stops what it is doing and services the interrupt. It accomplishes this by executing a set of instructions that are called an **interrupt service routine (ISR)**. The ISR is executed each time the interrupt is requested, just like the ringing telephone invokes a similar response each time.

Steps to Service an Interrupt

When an interrupt request is received at the HC11 processor, a sequence of six events takes place. The six events are listed here. Each step of this sequence is described in subsequent sections.

1. Complete execution of the current instruction.
2. Stack the current processor registers so that the current processor context can be restored when it returns from the interrupt service routine.
3. Disable the maskable interrupts by setting the I bit of the CCR (by default an interrupt service routine cannot be interrupted).
4. Load the interrupt vector into the program counter register. The vector is the address of the first instruction of the interrupt service routine.
5. Execute the instructions of the interrupt service routine.
6. Return from the interrupt by executing the RTI instruction. This causes the processor registers to be restored from the stack, including the return address, so that the processor can pick up where it left off prior to the interrupt. This also will restore the I bit of the CCR back to 0, reenabling maskable interrupts.

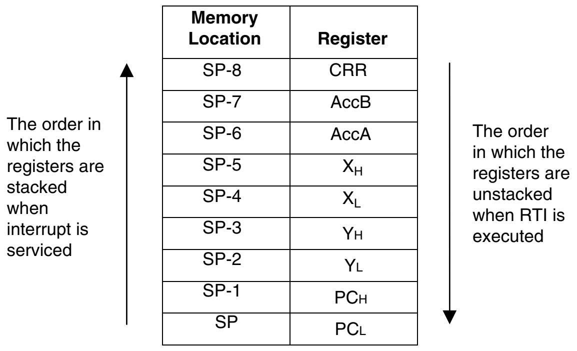
Complete the Current Instruction

All instructions require a finite length of time (a fixed number of machine cycles) to execute. Since the processor is rarely sitting idle, interrupt requests typically occur when the processor is occupied with an instruction. The processor will always complete execution of the current instruction before it services the interrupt. This requires a length of time that sometimes is significant to time-critical operations. It is important to remember that the program counter is updated during the execution of an instruction to point to the next instruction to be executed.

For example, the LDAA immediate instruction requires two machine cycles to execute. If the interrupt request occurs any time during the fetch or execution of this instruction, the hardware will wait until the instruction is complete before servicing the interrupt. The hardware will service a pending interrupt only when the execution of the current instruction is complete.

Stack Processor Registers

When the current instruction is completed, the hardware interrupt mechanism takes over. The first task is to save the program context. This way the program can pick up


The order in which the registers are stacked when interrupt is serviced
The order in which the registers are unstacked when RTI is executed

Memory Location	Register
SP-8	CRR
SP-7	AccB
SP-6	AccA
SP-5	X_H
SP-4	X_L
SP-3	Y_H
SP-2	Y_L
SP-1	PC_H
SP	PC_L

Figure 10.2 Stacking Order of Registers

right where it left off before the interrupt was serviced. The context is contained in the processor registers, and this context is saved by stacking the contents of the processor registers. This process is similar to the process of stacking the return address during a subroutine call so that the program knows where to return when the subroutine is complete. However, in addition to the return address, the entire context is saved by stacking all of the processor registers, excluding the stack pointer.

The order in which the registers are stacked is shown in Figure 10.2. The content of the stack pointer register (SP) is used to identify the first address used to stack the register contents. It is unnecessary to stack the stack pointer because it needs to continue to point to the next available stack location. After each byte is written to the stack, the stack pointer is decremented so that it always points to the next available stack location. Notice that the low byte of each of the 16-bit registers is written followed by the upper byte of the same register. By initial inspection, this may seem to violate Motorola's "hi-byte-first" convention. After closer analysis, the consistency becomes more apparent. The stack increments backwards, so the data still resides high byte/low byte in sequential memory locations.

Disable Maskable Interrupts

By default, the interrupt system inhibits interrupts after the registers have been stacked by setting the I control bit in the CCR. This blocks interrupts from occurring during an interrupt service routine. When the registers are restored at the end of the process, the interrupts are again enabled.

NOTE: In some instances, it may be desirable to allow interrupt requests to be serviced during an interrupt service routine. This is called "nesting of interrupts." It greatly complicates the system and rarely improves system performance.

Load the Interrupt Vector

When the current instruction has been completed and the processor registers have been stacked, the processor is ready to execute the interrupt service routine. The

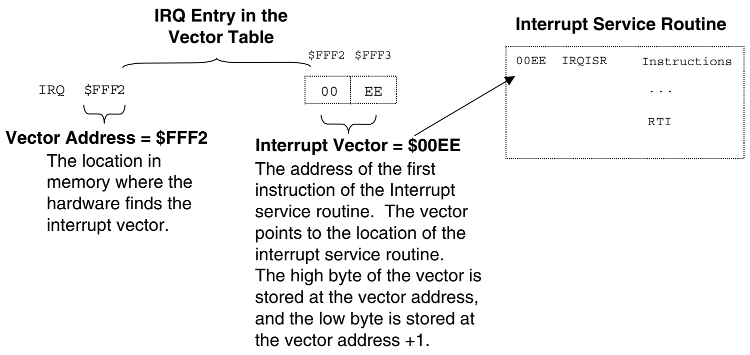

IRQ Entry in the Vector Table
$FFF2 $FFF3
IRQ $FFF2
00 EE
Vector Address = $FFF2
The location in memory where the hardware finds the interrupt vector.
Interrupt Vector = $00EE
The address of the first instruction of the Interrupt service routine. The vector points to the location of the interrupt service routine. The high byte of the vector is stored at the vector address, and the low byte is stored at the vector address +1.
Interrupt Service Routine
00EE IRQISR Instructions
...
RTI

Figure 10.3 Pointing to the Interrupt Service Routine Using the Interrupt Vector

interrupt service routine can be located anywhere in memory, but the address of the first instruction of the routine must be stored at a fixed place in memory. The address of the first instruction of the interrupt service routine is called the **interrupt vector** or simply **the vector**. The term "vector" is used to mean "pointer." The interrupt vector points to where the interrupt service routine is in memory, as shown in Figure 10.3.

This vector is stored at a specific **vector address** in memory. Since the vector is actually an address, it occupies two bytes in memory. Thus the high byte of the vector is located at the vector address, and the low byte of the vector is located at vector address + 1. The vector addresses are fixed locations in memory that cannot be changed. These addresses are designed into the hardware. For example, when an IRQ interrupt request is received, the hardware will go get the IRQ vector from the vector address $FFF2 and $FFF3. This cannot be changed by the user.

All vector addresses on the HC11 are located at the very end of the memory map. The block of memory designated as the vector addresses for the 21 interrupt sources is called the **vector table**. Figure 10.4 shows the vector table for the HC11. It contains vectors for each of the 21 interrupt sources, where each interrupt vector is stored at a unique interrupt vector address.

NOTE: The Strobe A (STRA) interrupt is internally connected to the IRQ pin; thus, it shares the IRQ vector. However, STRA has a unique local interrupt enable.

Execute the Instructions of the Interrupt Service Routine

An interrupt service routine is very similar to a subroutine. It is executed only because an interrupt request is recognized by the processor. A subroutine is executed only if it is called by a JSR or BSR instruction. An interrupt service routine is executed only if the corresponding interrupt request is received. In the same way, both routines end with a special instruction. All subroutines must end with an RTS instruction, and interrupt service routines end with the RTI instruction.

Since the vector is the address of the first instruction of the interrupt service routine, the hardware loads it into the program counter. The program counter then contains

Vector Address	Interrupt Source
FFD6, FFD7	SCI Serial System
FFD8, FFD9	SPI Serial Transfer Complete
FFDA, FFDB	Pulse Accumulator Input Edge
FFDC, FFDD	Pulse Accumulator Overflow
FFDE, FFDF	Timer Overflow
FFE0, FFE1	Timer Input Capture 4 / Timer Output Compare 5
FFE2, FFE3	Timer Output Compare 4
FFE4, FFE5	Timer Output Compare 3
FFE6, FFE7	Timer Output Compare 2
FFE8, FFE9	Timer Output Compare 1
FFEA, FFEB	Timer Input Capture 3
FFEC, FFED	Timer Input Capture 2
FFEE, FFEF	Timer Input Capture 1
FFF0, FFF1	Real Time Interrupt
FFF2, FFF3	IRQ (External Pin) / Strobe A
FFF4, FFF5	XIRQ (External Pin)
FFF6, FFF7	Software interrupt (SWI)
FFF8, FFF9	Illegal Opcode Trap
FFFA, FFFB	COP Failure
FFFC, FFFD	Clock Monitor Fail
FFFE, FFFF	RESET

Figure 10.4 HC11 Interrupt Vector Table *(adapted with permission from Motorola)*

the proper address of the instructions designed to service the interrupt. The instructions in the interrupt service routine are executed in the normal sequential manner until the service routine is completed.

Return from Interrupt and Unstack the Registers

The last instruction of every interrupt service routine must be the RTI instruction. The RTI instruction causes the nine bytes of processor register data to be pulled from the stack. They are pulled in the opposite order in which they were stacked, as shown in Figure 10.2. This process restores the context of the processor to the point it was when the interrupt occurred. The processor then continues the execution with the next instruction as if it never had been interrupted.

Example 10.1

Problem: Given the following register contents, designate the nine bytes that will be stacked if an interrupt occurs during the execution of the STAA instruction. Show the location of the bytes on the stack after the stacking operation.

```
          A    $92        X     $B680
          B    $FD        Y     $FF2D
          CCR  $E1        S     $0041
Address   Machine Code   Source Code
0002      B7 01 16       STAA  $0116
```

Solution: The first step is to complete the execution of the current instruction. The store instruction will write $92 from AccA to memory location $0116, update the CCR to $E9 because $92 is a negative number and update the PC to $0005 (STAA extended is a 3-byte instruction). Next the stacking operation starts by writing the low byte of the PC to the current location in the stack pointer, $(PC_L) \Rightarrow M_{SP}$, then decrements the stack pointer, $(SP) - 1 \Rightarrow SP$. This process continues until all nine bytes are pushed onto the stack.

```
$05 → $0041        PCL       → MSP
$00 → $0040        PCH       → MSP-1
$2D → $003F        YL        → MSP-2
$FF → $003E        YH        → MSP-3
$80 → $003D        XL        → MSP-4
$B6 → $003C        XH        → MSP-5
$92 → $003B        A         → MSP-6
$FD → $003A        B         → MSP-7
$E9 → $0039        B         → MSP-8
$0038 → SP         (SP) - 9  → SP
```

Self-Test Questions 10.2

1. What happens to the current instruction if the interrupt occurs during the fetch cycle?
2. How many bytes are pushed to the stack when an interrupt is serviced?
3. Which instruction must be executed at the end of every interrupt service routine?

10.3 Interrupt Control

Repetitive tasks in a program are usually packaged as a subroutine; however, they can also be packaged as an interrupt service routine. An interrupt service routine is functionally a subroutine that ends with the RTI instruction instead of the RTS instruction. In a sense, an interrupt service routine is a conditional subroutine. The only way an interrupt service routine can be executed is through the occurrence of an interrupt. The HC11 provides four instructions used for direct control of maskable interrupts. These instructions are summarized in Figure 10.5.

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
WAI	Wait for interrupt	*See Section 10.3*	-	-	-	-	-	X	-	-	-	-	-	-	-	-	-
RTI	Return from interrupt	*See Section 10.3*	-	-	-	-	-	X	-	↕	↓	↕	↕	↕	↕	↕	↕
CLI	Clear maskable interrupt mask	$0 \Rightarrow I$	-	-	-	-	-	X	-	-	-	-	0	-	-	-	-
SEI	Set maskable interrupt mask	$1 \Rightarrow I$	-	-	-	-	-	X	-	-	-	-	1	-	-	-	-

Figure 10.5 Maskable Interrupt Control Instructions

Each time an interrupt is executed, the current processor registers are pushed onto the stack prior to loading the program counter with the address of the first instruction of the interrupt service routine. The last instruction of every interrupt service routine must be a return from interrupt instruction (RTI). RTI causes the registers to be restored to their context prior to the interrupt by pulling the contents from the stack. The last processor register to be restored is the PC. It will contain the return address or the address of the next instruction to execute. RTI uses the inherent addressing mode. The CCR is one of the registers that was stacked; thus it is restored to the original state during the RTI instruction.

The process of stacking the registers requires many machine cycles, which leads to a delay before the interrupt service routine actually starts execution. Sometimes this delay is too long and therefore is unacceptable. The WAI instruction was designed to assist the hardware in preparing for an interrupt. When a WAI instruction is executed, the registers are stacked as if an interrupt has occurred and then the processor goes into an idle state waiting for the actual interrupt to occur. When the interrupt occurs, the WAI instruction has already stacked the registers so the execution of the first instruction of the interrupt service routine starts immediately. The normal delay that occurs during the process of stacking the registers is completely avoided. The WAI instruction uses the inherent addressing mode and has no affect on the CCR.

The CLI and SEI instructions control the maskable interrupt mask control bit (I bit) in the CCR. CLI clears the control bit enabling maskable interrupts. SEI sets the control bit disabling maskable interrupts. CLI and SEI are inherent mode instructions. Other than the I bit, they have no other affect on the CCR.

Local versus Global Control

The HC11 uses control bits in various registers to allow local and global control of interrupts. **Global control** is the process of controlling an entire class of interrupts with a single bit. For example, the I bit in the CCR is used to mask all maskable interrupts. If I = 1, all maskable interrupts are disabled regardless of the state of the local interrupt control bits. **Local control** bits provide individual control of each interrupt within the class. When the global mask is cleared (I = 0), then each individual interrupt source can be enabled or disabled using local control bits. For an interrupt source to be

Interrupt Source	Global Mask	Local Enables	Local Control Register
SCI Serial System	I bit in CCR	RIE, TIE, TCIE & ILIE	SCCR2
SPI Serial Transfer Complete	I bit in CCR	SPIE	SPCR
Pulse Accumulator Input Edge	I bit in CCR	PAII	TMSK2
Pulse Accumulator Overflow	I bit in CCR	PAOVI	TMSK2
Timer Overflow	I bit in CCR	TOI	TMSK2
Timer Input Capture 4 / Timer Output Compare 5	I bit in CCR	I4/O5I	TMSK1
Timer Output Compare 4	I bit in CCR	OC4I	TMSK1
Timer Output Compare 3	I bit in CCR	OC3I	TMSK1
Timer Output Compare 2	I bit in CCR	OC2I	TMSK1
Timer Output Compare 1	I bit in CCR	OC1I	TMSK1
Timer Input Capture 3	I bit in CCR	IC3I	TMSK1
Timer Input Capture 2	I bit in CCR	IC2I	TMSK1
Timer Input Capture 1	I bit in CCR	IC1I	TMSK1
Real Time Interrupt	I bit in CCR	RTII	TMSK2
IRQ (External Pin)	I bit in CCR	None	N/A
Strobe A	I bit in CCR	STAI	PIOC

Figure 10.6 Maskable Interrupt Control Bits (*adapted with permission from Motorola*)

enabled, both the global and local control bits must be enabled. If either is not enabled, the interrupt will be ignored by the processor.

Figure 10.6 provides a summary of the global and local interrupt control bits for all maskable interrupt sources. Note that the global control bit (I) is referred to as a mask and local control bits are enables. If I = 1, maskable interrupts are disabled (enable is active low). If a local control bit is set, the local interrupt is enabled (enable is active high).

An additional parallel can be drawn between the ring of the telephone and the interrupt requests to a computer processor. Sometimes the ring of the telephone is ignored. When it goes unserviced for four or five rings, the caller usually hangs up, which terminates the interrupt request. In some extreme cases, the ringer is turned off, so that the interrupt request is not heard. However, the telephone company still continues to send the the ring signal (interrupt request). The computer processor also has the ability to ignore interrupt requests. The interrupts can be disabled or masked, which causes the processor to never "hear" them. The external stimuli may continue to occur, but the processor will never respond to them when the interrupts are masked.

The interrupt system of a processor has to be enabled for it to function. Typically, the interrupt system is disabled after reset to avoid inadvertent interrupt requests from being serviced before the software is ready for this to happen. Control is split between the two interrupt classes: maskable and nonmaskable. The processing of enabling and inhibiting interrupts is sometimes called **masking**, which explains the use of the interrupt class names.

Self-Test Questions 10.3

1. What instruction is designed to help speed up interrupt servicing?
2. How can the user disable all maskable interrupts?
3. Is there a local interrupt control bit for Timer Overflow Interrupt? If so, where is it?

10.4 Maskable Interrupts

Maskable interrupts are a group of interrupt sources that can be globally as well as locally controlled, as shown in Figure 10.6. The I bit of the CCR acts as the global interrupt mask for the entire class. When a 1 is written to the I bit of the CCR, all maskable interrupts are inhibited. Local interrupt enable masks are overridden by the global mask. When the I bit is 0, control of maskable interrupts is transferred to the local bits.

Serial Communication System Interrupt Sources

The serial communications system contains two interrupt sources and five local interrupt enables. The SCI subsystem has four local interrupt masks, RIE, TIE, TCIE and ILIE, which are located in the SCCR2 register. Each of these bits must be set to enable the corresponding interrupt request to be made. All four of these enable bits are linked to the SCI system interrupt request. Therefore, the SCI system interrupt can be requested by four different events within the serial communications interface. The SPI has one local interrupt enable, the SPIE bit of the SPCR register. The SPIE bit must be set to enable the SPI interrupt.

Further explanation of serial communication interrupt sources is found in chapter 13.

Timer System Interrupt Sources

The timer system contains 12 local interrupt enable bits. The TOI bit in the TMSK2 register enables the timer overflow interrupt. Timer overflows occur when the TCNT register rolls over from $FFFF to $0000. The PAII and PAOVI bits in the TMSK2 register enable interrupts linked to the pulse accumulator function. PAII enables the pulse accumulator input edge interrupt. Each time a valid pulse accumulator input edge is detected, this interrupt occurs. PAOVI enables the pulse accumulator overflow interrupt. Pulse accumulator overflow occurs when the PACNT register rolls over from $FF to $00. The RTII in the TMSK2 register enables the real-time interrupt. Real-time events occur on a periodic basis at all times during normal operation of the processor.

The TMSK1 register contains eight additional timer system interrupt enable bits. OC1I, OC2I, OC3I and OC4I enable interrupts for the output-compare functions OC1–OC4

OPTION	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1039	ADPU	CSEL	IRQE	DLY	CME	–	CR1	CR0
RESET	0	0	0	1	0	0	0	0

Figure 10.7 IRQ Edge Control Bit

respectively. IC1I, IC2I and IC3I enable interrupts for the input-capture functions IC1–IC3 respectively. I4/O5I enables the interrupt for the IC4 or the OC5 function depending on which of these two functions is active. Setting an interrupt enable bit in the TMSK1 or TMSK2 registers enables the corresponding interrupt.

Further explanation of the time system interrupt sources is found in chapter 12.

External Interrupts Using IRQ

The IRQ interrupt source is an external hardware pin dedicated to the interrupt system. Unlike the maskable interrupt sources in the timer and communications subsystems, IRQ is not tied to some internal hardware system in the HC11. This pin can be used to cause an interrupt request from any outside source. It is often linked to the interrupt features of an external peripheral device. The IRQ interrupt does not have a local control capability. It is controlled only by the I bit in the CCR.

Unlike the other maskable interrupt sources, IRQ has a dedicated hardware pin that can be connected to any external interrupt source. To accommodate a greater variety of interrupt sources, IRQ can be configured for level-sensitive or edge-sensitive operation. The IRQ Edge bit (IRQE) in the OPTION register is used to select between edge and level sensitivity, as shown in Figure 10.7. If IRQE is set, IRQ is configured for edge-sensitive operation. If it is cleared, IRQ is configured for level-sensitive operation. After reset, IRQE = 0, so that the default is level-sensitive operation.

NOTE: The IRQE bit can only be written to once in the first 64 E clock cycles out of reset when in normal modes. IRQ has no local interrupt enable. It is enabled and disabled by the state of the I bit in the CCR.

Parallel I/O Interrupt Source

During parallel I/O an interrupt can be requested each time strobe A flag (STAF) is set in the PIOC register. The strobe A interrupt enable bit (STAI), also found in the PIOC register, is the local enable that controls whether the interrupt will be requested. This allows an interrupt service routine to do some special data processing only when the valid strobe A is received.

This interrupt source is unique. It shares the interrupt vector with the IRQ interrupt; therefore, only one interrupt service routine can exist for both the IRQ and the parallel I/O interrupt sources. If the strobe A interrupt is enabled, the IRQ pin should not be used. The system checks the state of the IRQ pin before it checks the state of the STAF and STAI bits in the PIOC.

Further explanation of the parallel I/O system is found in chapter 9.

Interrupt Source	Priority	PSEL[3:0]
IRQ (External Pin)	1	0 1 1 0
Real Time Interrupt (RTI)	2	0 1 1 1
Timer Input Capture 1	3	1 0 0 0
Timer Input Capture 2	4	1 0 0 1
Timer Input Capture 3	5	1 0 1 0
Timer Output Compare 1	6	1 0 1 1
Timer Output Compare 2	7	1 1 0 0
Timer Output Compare 3	8	1 1 0 1
Timer Output Compare 4	9	1 1 1 0
Timer Input Capture 4 / Timer Output Compare 5	10	1 1 1 1
Timer Overflow	11	0 0 0 0
Pulse Accumulator Overflow	12	0 0 0 1
Pulse Accumulator Input Edge	13	0 0 1 0
SPI Serial Transfer Complete	14	0 0 1 1
SCI Serial System	15	0 1 0 0

Figure 10.8 Maskable Interrupt Priority (*courtesy of Motorola)*

Maskable Interrupt Priority

Since there are 15 maskable interrupt sources, there is a reasonable probability that two or more of the sources could be enabled at the same time. In the event that two interrupts occur during the processing of the same instruction, which interrupt will be serviced first? To resolve this potential conflict, each maskable interrupt source is assigned a priority, which governs the order in which interrupts will be serviced, as shown in Figure 10.8. When two interrupt requests occur during the execution of the same instruction, the interrupt with the highest priority (1 being the highest priority, 15 being the lowest priority) will be processed first.

The 4-bit code contained in the Priority Select bits (PSELx) of the Highest Priority register (HPRIO) elevates the interrupt priority of one maskable interrupt to the highest level. Figure 10.9 illustrates the location of these bits in the HPRIO register. The highest priority interrupt will be serviced before any other maskable interrupt request. Figure 10.8 shows the default priority, as well as the value required in the PSELx bits to change this priority. When an interrupt is elevated in priority, the priority order of all other interrupts is not changed. The fact that the priority order does not change when one interrupt is elevated in priority is very evident in the interrupt priority resolution flowchart, shown in Figure 10.10.

HPRIO	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$103C	RBOOT	SMOD	MDA	IRVNE	PSEL3	PSEL2	PSEL1	PSEL0
RESET	0	0	0	0	0	1	1	0

Figure 10.9 Interrupt Priority Select Bits

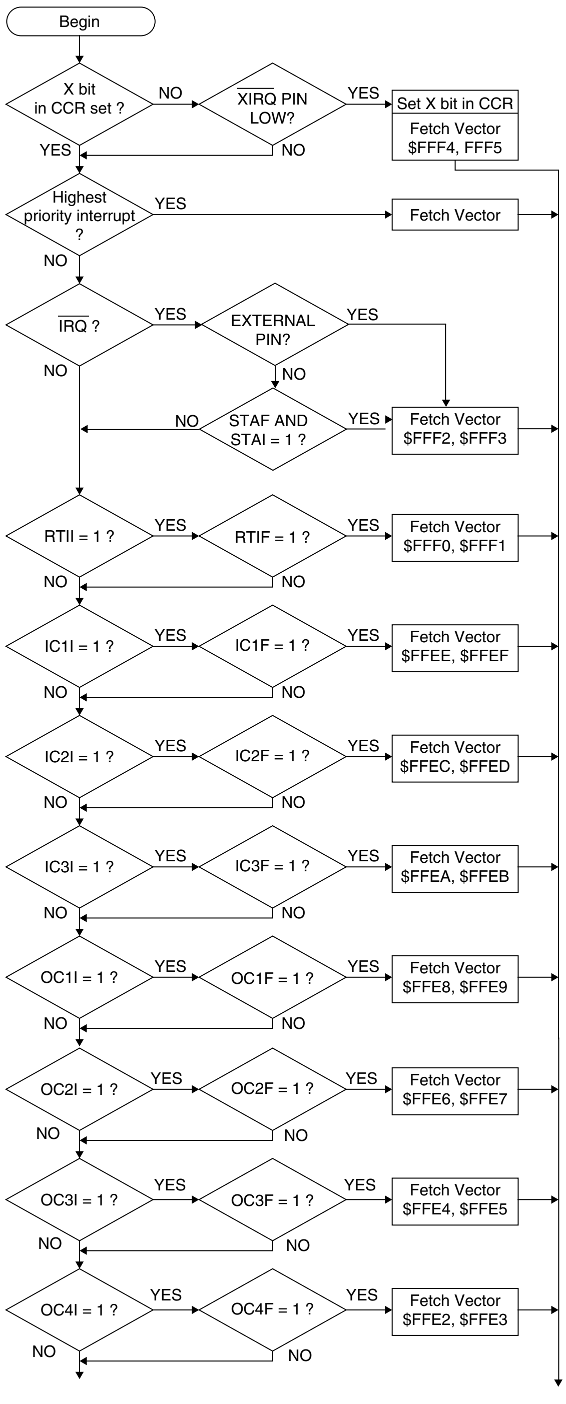


Figure 10.10 Interrupt Priority Resolution Flowchart (*adapted with permssion from Motorola*)

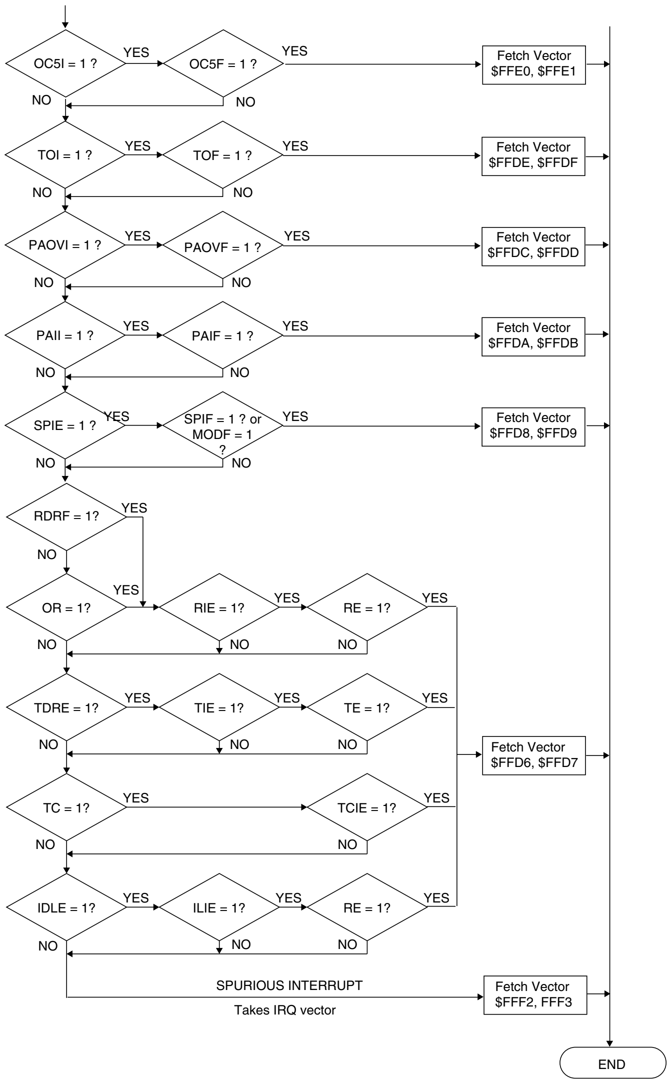


Figure 10.10 *(continued)*

Self-Test Questions 10.4

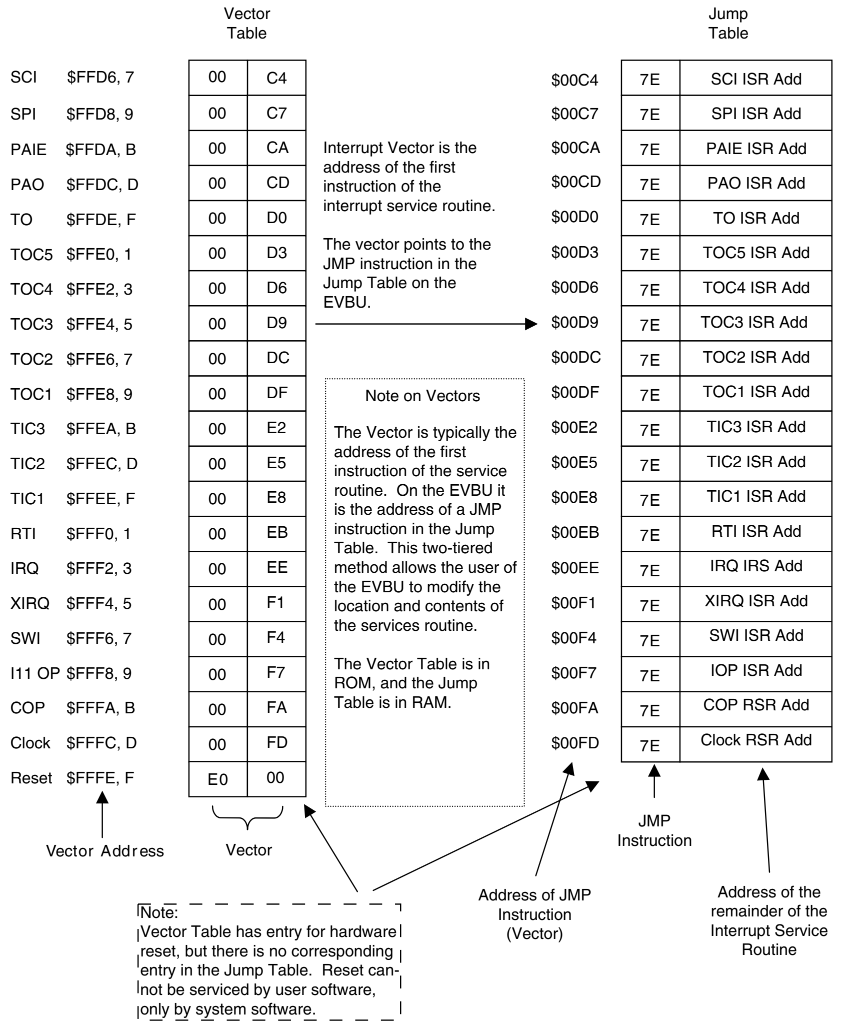
1. What binary value must be written to the IRQE bit to select level-sensitive operation?
2. What is the default priority of the RTI?
3. If the Timer Input Capture 1 and the Pulse Accumulator Overflow interrupts occur during the execution of the same instruction, which interrupt request will be serviced first?

10.5 Using Interrupts on the EVBU

As was presented in section 10.2, the interrupt vector table is located in the last locations of memory on the HC11, from $FFD6 through $FFFF. As these locations on the EVBU are in ROM, the contents of the vector table are fixed during the manufacturing process. On a normal HC11 system, the vectors are the actual addresses of the interrupt service routines used by the system. However, on the EVBU the location of interrupt service routines is not determined until the user is actually writing a program that uses interrupts. The user of the EVBU cannot change the vectors that are permanently stored in the vector table in ROM.

The fixed state of the vectors on the EVBU could severally limit what could be done with interrupts on the EVBU. The designers of the EVBU anticipated this restriction and devised a method that allows users of the EVBU to have full control over the location and length of interrupt service routines. The permanent vectors in the ROM-based vector table are address locations in the RAM. Each vector points to a JMP instruction in RAM. The JMP instruction is initialized with a default ISR address located in BUFFALO. The JMP instruction effectively becomes the first instruction of the interrupt service routine. The remainder of this interrupt service routine can be anywhere in memory. The user must overwrite the default effective address of the jump instruction with the actual start address of the ISR.

Each of the interrupt sources has a corresponding JMP instruction RAM except the RESET. The reset interrupt causes the BUFFALO monitor program to start over. It starts at location $E000 of ROM. The remaining 20 JMP instructions occupy a block of RAM that is called the **Vector Jump Table,** as shown in Figure 10.11. The theory of how interrupts work is still the same; the difference is that the BUFFALO system supplies the first instruction of each interrupt service routine via the interrupt vector jump table. The user must load the address of the rest of the interrupt service routine into the appropriate locations in the vector jump table and write the remainder of the interrupt service routine.


Vector Table
Jump Table
Vector Address
Vector
JMP Instruction
Address of JMP Instruction (Vector)
Address of the remainder of the Interrupt Service Routine

		Vector Table			Jump Table	
SCI	$FFD6, 7	00	C4	$00C4	7E	SCI ISR Add
SPI	$FFD8, 9	00	C7	$00C7	7E	SPI ISR Add
PAIE	$FFDA, B	00	CA	$00CA	7E	PAIE ISR Add
PAO	$FFDC, D	00	CD	$00CD	7E	PAO ISR Add
TO	$FFDE, F	00	D0	$00D0	7E	TO ISR Add
TOC5	$FFE0, 1	00	D3	$00D3	7E	TOC5 ISR Add
TOC4	$FFE2, 3	00	D6	$00D6	7E	TOC4 ISR Add
TOC3	$FFE4, 5	00	D9	$00D9	7E	TOC3 ISR Add
TOC2	$FFE6, 7	00	DC	$00DC	7E	TOC2 ISR Add
TOC1	$FFE8, 9	00	DF	$00DF	7E	TOC1 ISR Add
TIC3	$FFEA, B	00	E2	$00E2	7E	TIC3 ISR Add
TIC2	$FFEC, D	00	E5	$00E5	7E	TIC2 ISR Add
TIC1	$FFEE, F	00	E8	$00E8	7E	TIC1 ISR Add
RTI	$FFF0, 1	00	EB	$00EB	7E	RTI ISR Add
IRQ	$FFF2, 3	00	EE	$00EE	7E	IRQ IRS Add
XIRQ	$FFF4, 5	00	F1	$00F1	7E	XIRQ ISR Add
SWI	$FFF6, 7	00	F4	$00F4	7E	SWI ISR Add
I11 OP	$FFF8, 9	00	F7	$00F7	7E	IOP ISR Add
COP	$FFFA, B	00	FA	$00FA	7E	COP RSR Add
Clock	$FFFC, D	00	FD	$00FD	7E	Clock RSR Add
Reset	$FFFE, F	E0	00			

Interrupt Vector is the address of the first instruction of the interrupt service routine.

The vector points to the JMP instruction in the Jump Table on the EVBU.

Note on Vectors

The Vector is typically the address of the first instruction of the service routine. On the EVBU it is the address of a JMP instruction in the Jump Table. This two-tiered method allows the user of the EVBU to modify the location and contents of the services routine.

The Vector Table is in ROM, and the Jump Table is in RAM.

Note:
Vector Table has entry for hardware reset, but there is no corresponding entry in the Jump Table. Reset cannot be serviced by user software, only by system software.

Figure 10.11 Relationship of Interrupt Vector Table to Interrupt Vector Jump Table

Example 10.2

Problem: Using the flowchart in Figure 10.12, write a program that will allow the IRQ to be serviced.

Solution: The source code starts with some parameter definitions: REGBASE and so on, as shown in Figure 10.13. Note that the address assigned to IRQJMP is $00EF, not $00EE. The opcode for the JMP instruction is located in $00EE, and the absolute address is located in $00EF and $00F0. The executable code starts by loading the address of the IRQISR into the vector jump table. Note that the maskable interrupts are disabled during the initialization. Then the wait message is displayed, and the program waits for the IRQ interrupt. When the interrupt occurs, the IRQISR is executed. The IRQ message is displayed, and a two-second delay is called. The RTI instruction completes the execution of the ISR, and the control is returned to the main program with the BRA instruction following the WAI instruction.

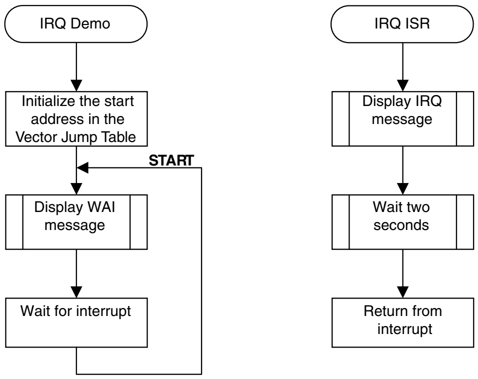


Figure 10.12 IRQ Interrupt Demo Flowchart

```
* IRQ Interrupt Demo - Message changes when IRQ interrupt occurs

REGBASE EQU     $1000
IRQJMP  EQU     $00EF

DLY10MS EQU     $E2E5         BUFFALO 3.4 10 ms delay
OUTSTRG EQU     $FFC7         BUFFALO output string of characters to monitor

        ORG     $0100
MAIN    SEI                   Inhibit maskable interrupts during init
        LDX     #IRQISR
        STX     IRQJMP        Load jump table with IRQ ISR start address
        CLI                   Re-enable maskable interrupts

START   LDX     #MSGWAI
        JSR     OUTSTRG       Display wait message
        WAI                   Wait for the IRQ interrupt
        BRA     START         Repeat forever

TDLY    LDAB    #200          Two second time delay to slow things down
UP      JSR     DLY10MS
        DECB
        BNE     UP
        RTS

IRQISR  LDX     #MSGIRQ
        JSR     OUTSTRG       Display IRQ message
        JSR     TDLY          Wait two seconds
        RTI

MSGWAI  FCC     'Waiting for IRQ'
        FCB     $04
MSGIRQ  FCC     'IRQ ISR is running'
        FCB     $04
```

Figure 10.13 IRQ Interrupt Demo Source Code

Self-Test Questions 10.5

1. Where in memory is the Vector Table located?
2. What is effectively the first instruction of every interrupt service routine on the EVBU?

10.6 Nonmaskable Interrupts

On the HC11, 6 of the 21 interrupt sources are classified as nonmaskable, as shown in Figure 10.14. In general, nonmaskable interrupts cannot be inhibited or enabled by the user. This rule holds true for three of the interrupts in this class, yet the other three can be controlled to a small degree by the user. The SWI, Illegal Opcode Fetch and Processor Reset interrupts cannot be inhibited in any way. They are active as part of normal processor function. The Clock Monitor, COP Failure and XIRQ have control bits that the user can use to inhibit these interrupts. The specific bits are shown in Figure 10.14.

External Interrupts Using XIRQ

XIRQ functions in the same manner as the IRQ interrupt; however, it has a nonmaskable characteristic. The XIRQ interrupt source is an external hardware pin ($\overline{\text{XIRQ}}$) dedicated to the interrupt system. This pin can be used to cause an interrupt request from any outside source in the same manner as IRQ. Because it has a higher priority than all maskable interrupts, it is often linked to the interrupt features of an external peripheral device.

XIRQ exhibits two characteristics of a maskable interrupt and maintains all the strengths of a nonmaskable interrupt. First, the XIRQ is inhibited shortly after reset, disabling the XIRQ function. Second, the user can enable the XIRQ by using a TAP instruction to clear this bit. After the XIRQ is enabled, it cannot be inhibited by the user, thus becoming a nonmaskable interrupt. Since the XIRQ has higher interrupt priority (as do all nonmaskable interrupts) than all the maskable interrupts, the state of the I bit is irrelevant to the XIRQ. When an XIRQ is requested, both the X and I bits in the CCR are set by the hardware, disabling further interrupts from occurring. When the RTI instruction is executed, the original state of the X and I bits is restored.

Interrupt Source	Global Mask	Local Mask	Local Control Register
XIRQ (External Pin)	X bit in CCR	None	N/A
Software Interrupt (SWI)	None	None	N/A
Illegal Opcode Trap	None	None	N/A
COP Failure	None	NOCOP	CONFIG
Clock Monitor Fail	None	CME	OPTION
Reset	None	None	N/A

Figure 10.14 Nonmaskable Interrupt Control

NOTE: The X bit is set after reset disabling the XIRQ interrupt source. The user can use a TAP instruction to clear this bit to enable this nonmaskable interrupt source. Once XIRQ has been enabled, it cannot be disabled, except via system reset.

Software Interrupt (SWI)

SWI is an instruction in the HC11 instruction set. It is executed in the same manner as all other instructions, yet it is unique. It is the only software means of directly causing an interrupt. The SWI interrupt functions in the same manner as the maskable interrupts, but is nonmaskable. When serviced, the I bit in the CCR is set, inhibiting all maskable interrupts. The CPU registers are stacked in the same manner; a vector is fetched from the vector table and an interrupt service routine is executed. SWI is truly nonmaskable because it cannot be inhibited by the X and I global interrupt masks bits in the CCR or any local interrupt enable control bits.

NOTE: Since SWI is a software instruction, it will not be fetched if any other interrupt is pending. However, once an SWI instruction begins, no other interrupt can be honored until the SWI vector has been fetched.

The BUFFALO monitor program uses SWI instructions for several useful purposes. First, if the user program that is running from a BUFFALO prompt is terminated with an SWI instruction, BUFFALO will service the interrupt by displaying on the monitor the register contents. This is a very useful means of discovering the status of the processor when program execution is complete. If the user chooses to use the SWI instruction for another purpose, the default use of SWI by BUFFALO will be disabled.

Illegal Opcode Fetch

Since not all possible opcodes or opcode sequences are defined, an illegal opcode detection mechanism has been built into the HC11. Each time an illegal opcode is detected, the Illegal Opcode interrupt is requested. The address of the first byte of the illegal opcode is stacked when the registers are stacked. The user can use this stacked value as a pointer to the illegal opcode and thus determine if the illegal condition was caused by a single- or double-byte opcode.

For example, the TEST instruction (opcode $00) is defined for the special test and bootstrap modes, but is considered illegal by the single-chip and expanded modes.

Note: Since the illegal opcode fetch is a nonmaskable interrupt, the illegal opcode vector should never be left uninitialized. If the vector address does not contain a valid interrupt vector, the processor will crash if this interrupt occurs.

Computer Operating Properly (COP) Failure

Fundamentally, the Computer Operating Properly (COP) system is a timer. It is intended to detect software processing errors. When the COP is being used, the

CONFIG	b7	b6	b5	b4	b3	b2	b1	b0
$103F	–	–	–	–	NOSEC	NOCOP	ROMON	EEON
RESET	0	0	0	0	1	1	1	1

Figure 10.15 COP Local Control Bit

software is responsible for keeping a free-running clock from timing out. If the clock times out, it is an indication that the software is no longer executing properly. This condition causes a system reset. Although this is classified as a nonmaskable interrupt source, it can be locally disabled by the NOCOP bit of the CONFIG register, as shown in Figure 10.15. When the NOCOP bit is set, the COP is disabled. When the NOCOP bit is cleared, the COP is enabled. The COP is disabled by reset.

Further information on the operation of the COP feature is provided in section 10.6.

Clock Monitor Failure

The clock monitor system uses time delay controlled by an internal resistor/capacitor (RC) to determine if the system clock is operating properly. If it detects a problem with the system clock, it will cause a system reset. Since the clock monitor circuitry is based on a delay derived from an RC delay, the clock monitor circuitry will operate properly without any MCU clocks.

The CME bit in the OPTION register is used to disable the clock monitor system, as shown in Figure 10.16. When CME is cleared, the clock monitoring system is disabled. The clock monitor feature is disabled by reset.

Further information on the operation of the Clock Monitor Failure feature is provided in section 10.7.

Power-On Reset (POR)

The POR is only intended to initialize the internal MCU circuits. Essentially, it is not an interrupt in the purest sense; however, it exhibits many interrupt-like characteristics. When the power-on reset occurs, the system stops and restarts. The main system software is executed. On the EVBU, the main system software is the BUFFALO monitor program. Since the main system software technically never ends, the software does not require the common terminating RTI instruction.

Further explanation of reset conditions on the HC11 are explained in sections 10.7 and 10.8 of this text.

OPTION	b7	b6	b5	b4	b3	b2	b1	b0
$1039	ADPU	CSEL	IRQE	DLY	CME	–	CR1	CR0
RESET	0	0	0	1	0	0	0	0

Figure 10.16 Clock Monitor Control Bit

Interrupt Source	Priority
Power-On Reset (POR)	1
Clock Monitor Fail	2
COP Failure	3
XIRQ (External Pin)	4
Illegal Opcode Fetch	5
Software Interrupt (SWI)	6

Figure 10.17 Nonmaskable Interrupt Priority

Nonmaskable Interrupt Priority

The nonmaskable interrupt sources are prioritized as shown in Figure 10.17. Unlike the maskable interrupt sources, nonmaskable sources cannot be reprioritized. Because more than one of the nonmaskable interrupt requests can occur during a single instruction, each nonmaskable interrupt has a relative priority (where 1 is the highest priority). When two or more nonmaskable interrupt requests occur at the same time, the highest-priority request will be serviced first. Each nonmaskable interrupt has a higher priority than any maskable interrupt. Therefore, if a maskable interrupt request is received at the same time as a nonmaskable interrupt request is received, the nonmaskable interrupt request will be serviced first.

Stop Instruction

The STOP instruction is a special instruction that in one sense acts like a nonmaskable interrupt. When it is executed, it causes the processor to halt. The internal system clocks stop, and the system is placed in a minimum-power standby mode. All CPU registers and I/O pins, including the CCR, remain unaffected. The STOP is controlled by the S control bit of the CCR. When S is cleared, the STOP instruction causes the system halt, as described above. If S is set when the STOP instruction is executed, it is interpreted as an NOP.

Recovery from a STOP instruction is accomplished by a hardware reset (RESET) or by the processor receiving an XIRQ or IRQ interrupt request. XIRQ can cause STOP recovery in two ways. If the X control bit in the CCR is set, which masks XIRQ interrupts, execution will continue with the instruction immediately following the STOP instruction. If X is clear, which enables XIRQ interrupts, execution will start with the processor register stacking operation common to the servicing of all interrupts.

Self-Test Questions 10.6

1. How many nonmaskable interrupt sources are available on the HC11?
2. Can any of the nonmaskable interrupts be masked? If so, which ones and how?
3. What is unique about the SWI interrupt?
4. What is the relative priority of the nonmaskable interrupts in reference to the family of maskable interrupts?

10.7 Resets

The **reset** feature of a microcomputing system is used to set initial conditions within the system and begin executing instructions from a predetermined address. The reset condition can be caused by an external or an internal stimulus. External stimuli are classified as **external resets,** and internal stimuli are classified as **internal resets**.

Typically, the reset condition is caused by some external stimulus such as power-on or a low level on the hardware RESET pin. These two interrupt sources are referred to as external resets, because they are caused by an external stimulus. The internal hardware subsystems of the HC11 can also detect illegal conditions that can also cause a reset condition. This is an internal reset condition. The internal reset condition is output on the RESET pin so that peripheral devices can be made aware of the system reset condition caused by the internal stimuli. The Computer Operating Properly (COP) and Clock Monitor subsystems are the two subsystems on the HC11 that can cause the internal reset condition.

A reset condition immediately stops the execution of the current instruction and forces the program counter to one of three vector addresses, depending on the source of the reset. Typically following the reset stimulus, a delay starts. This delay allows the system to stabilize before the servicing of the reset interrupt can begin. The process of servicing the reset stimulus is a sequence of tasks that must be performed. This delay, along with other reset related tasks require a clock. If no clock is present, the processor cannot advance past the first step of the servicing sequence. Servicing of the reset is similar to the process followed to service a general-purpose interrupt. Interrupt servicing is discussed earlier in this chapter.

There are four sources of a reset condition: the power-on reset (POR), external reset (activating the RESET pin on the chip), a COP watchdog failure, and a clock monitor failure. These reset conditions will be explained in the following sections.

Power-On Reset (POR)

The Power-On Reset (POR) is caused by an external stimulus. It is generated by a rising edge on the main power pin of the HC11 chip (V_{DD}). It is used exclusively for power-up conditions and cannot be used to detect drops in the power supply voltage. Immediately following the rising edge on V_{DD} and when the oscillator becomes active, a delay starts of 4,064 clock cycles. This delay allows the clock generator to stabilize. If the RESET pin is at logic zero at the end of this delay, the processor remains in reset until the RESET pin changes to a logic one state. In most applications the POR system is externally connected to the RESET pin via a low-voltage connect circuit. The low-voltage connect circuit is also typically connected to a system reset switch. Thus, the POR and system reset are combined to both cause a reset on the external RESET pin. This protects the processor from power fluctuations and assures that the same process is followed for both reset sources.

External Reset (RESET)

The HC11 is equipped with an external RESET pin. It functions similar to the external Reset pins on all processors, yet has additional functionality. Typically, external reset is an input-only pin on a processor. An active state on this pin causes the process to enter a reset sequence. The HC11 RESET pin functions as an external reset input as well as an internal reset output.

When any reset occurs, an internal device drives the RESET pin low for four E clock cycles. At the end of the four clock cycles the internal device releases the RESET pin. It then waits an additional two E clock cycles and samples the signal level on the RESET pin. If it is still low, the processor then handles the reset as an external reset. If the RESET pin is high after the two additional E clocks, the reset is handled as an internal reset.

NOTE: Because of the dual function of the RESET pin, an external resistor/capacitor (RC) power-up delay circuit must not be connected to the RESET pin. The RC charge time can cause the processor to misinterpret the type of reset that occurred.

COP Failure Reset

The COP system is designed to detect software processing errors. When the COP is being used, the software is responsible for keeping a free-running clock from timing out. If the clock times out, it is an indication that the software is not executing properly. When this condition is detected, the COP failure reset is initiated.

The COP failure feature of the HC11 is enabled and disabled by the NOCOP bit of the CONFIG register, as shown in Figure 10.15. Because of the sensitive nature of this control function, stringent requirements have been established for changing the NOCOP bit. The user has the ability to change this control bit by writing to the CONFIG register. However, changes to this bit do not immediately take effect. This bit is only checked after a reset. Because of this strict procedure, it is very unlikely that this bit will be accidentally changed.

The COP timer must be serviced by the user software before the COP time-out to avoid the system reset. The user software must write to the COP arm/reset register (COPRST), shown in Figure 10.18, to service the COP Timer. Servicing the timer has two steps. First, the user must write \$55 to the COPRST register to arm the COP timer clearing mechanism. Then the user must write \$AA to the COPRST register to clear the COP timer.

The time-out rate of the COP failure system is controlled by the CR1 and CR0 bits in the OPTION register, as shown in Figure 10.19. The default time-out rate is set to E/2^{15}. The COP timer reset select bits select one of the four scale factors, as shown. CR1 and CR0 can be written to only once during the first 64 machine cycles after reset.

COPRST

\$103A	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
RESET	0	0	0	0	0	0	0	0

Figure 10.18 COP Arm/Reset Register

OPTION	b7	b6	b5	b4	b3	b2	b1	b0
$1039	ADPU	CSEL	IRQE	DLY	CME	–	CR1	CR0
RESET	0	0	0	1	0	0	0	0

Control Bits				
CR1	CR0	Scale Value	Time-out Rate (E/x)	Time-out Rate (ms)
0	0	1	$E/2^{15}$	16.384
0	1	4	$E/2^{17}$	65.536
1	0	16	$E/2^{19}$	262.144
1	1	64	$E/2^{21}$	1.049 s

Figure 10.19 COP Time-Out Rate Control Bits

Since the COP timer is based on the system clock, the COP watchdog cannot detect errors that cause the system clock to stop. Placing the COP service instructions in an interrupt service routine is a bad practice. Interrupt service routines can still function when the main program is not functioning properly. In this case, the COP timer would be reset in the interrupt service routine, indicating that the software is operating properly when it actually is not.

Clock Monitor Reset

The clock monitor circuitry is designed to determine the proper operation of the internal processor clock. Specifically, E clock rates that drop below 10 kHz will be detected as a clock failure. E clock frequencies must be above 200 kHz to assure that the clock monitor circuitry does not accidentally detect clock problems. Systems that operate on E clock frequencies below 200 kHz should not use the clock monitor circuitry.

Since the COP system requires the system clock to function properly, the clock monitor system is often used as a backup to the COP failure system. The clock monitor system is also used to detect the accidental use of the STOP instruction. The STOP instruction shuts down the system clock; thus the clock monitor circuitry would detect its use.

The clock monitor system must be enabled by software. The CME control bit in the OPTION register is set to enable this function. The CME bit is cleared by system RESET, thus disabling the clock monitor system.

Self-Test Questions 10.7

1. What is the difference between an external and an internal reset?
2. Name the four reset sources.
3. What is the default COP time-out rate?

10.8 Servicing a Reset Interrupt

There are four sources of a reset condition: the power-on condition, an active low state on the RESET pin, a COP watchdog failure, and a clock monitor failure.
Each of these reset sources causes a set process to be followed. The steps are as follows:

1. Stop execution of the current instruction.
2. System hardware enters the initial conditions.
3. Load the system configuration data from the CONFIG register.
4. Establish the mode of operation.
5. Load the program counter with the reset vector.

Stop Execution of Current Instruction

Unlike the interrupt service process that waits for the execution of the current instruction to complete, the reset process forces the execution of the current instruction to stop, regardless of the state of that execution. Whatever effect this action has on the processor systems is unimportant, since the next step is to initialize the hardware systems.

Initialize Hardware Conditions

The following list provides a brief description of all HC11 systems affected by the reset function. If a particular system is not affected or if it is undefined, it is not included in this list.

- Clock Monitor system and Clock Monitor interrupt are disabled.
- COP and COP Failure interrupt are disabled.
- STOP mode is disabled.
- XIRQ and IRQ interrupts are globally disabled (masked).
- IRQ is selected as the highest-priority maskable interrupt source. It is configured for level-sensitive operation.
- All maskable interrupts (i.e., PAI, TOC2, SPIE, etc) are locally disabled.
- RAM is mapped to $0000.
- Control register block is mapped to $1000.
- ROM and EEPROM are configured for normal read mode.
- EEPROM programming controls are disabled, and EEPROM block protection is enabled.
- All I/O pins on PORTA–PORTE are defaulted to general-purpose I/O: PORTB and PA6–PA3 are configured as output pins that are cleared, and PORTC, PORTE, PD5–PD0, PA7 and PA2–PA0 are configured as inputs.
- Timer Counter (TCNT register) is cleared.
- Timer prescaler is set to a scale factor of 1.

- Timer Output Compare registers are set to $FFFF.
- OC5 is selected, and IC4 is disabled.
- All output-compare, input-capture and pulse accumulator functions are disabled.
- Real-Time interrupt rate is set to the fastest rate ($E/2^{13}$).
- All timer, OC, IC, PA and RTI interrupt flags are cleared.
- COP rate is set for the fastest rate ($E/2^{15}$).
- SCI receiver clock rate is set to E/16.
- SCI transmit clock rate is set to E/256.
- SCI transmitter and receiver are disabled. All related control and status bits are set to their respective idle conditions (some high and some low).
- SPI system is disabled.
- Analog-to-digital converter is disabled.

Load System Configuration from CONFIG Register

The HC11 includes an EEPROM-based CONFIG register. This register contains a number of master system control bits. Since this register is EEPROM based, it is nonvolatile; thus configuration is maintained during reset and power-down sequences. Writes to this register do not take effect until after the next reset sequence.

Establish the Mode of Operation

There are four modes on the HC11: Single Chip, Expanded, Bootstrap, and Special Test. The mode is selected by the status of the MODA and MODB hardware input pins.

Load the Reset Vector

After the reset operation has initialized and configured the hardware, the hardware fetches the reset vector from the vector table and loads the program counter. Control is then transferred to the software, beginning with the instruction located at the vector in memory. Similar to the process of servicing an interrupt, the vector is the address of the first instruction to be executed as a result of the reset.

Self-Test Questions 10.8

1. What are the five steps performed each time a reset is serviced?
2. What is the default state of the TCNT register after reset?
3. What is the state of timer system flags after reset?
4. What is the state of the analog-to-digital converter after reset? Why?

Summary

Interrupts and resets are often discussed together because they share a common function. Each must fetch a vector from memory to initialize the starting point of software to complete the servicing of the interrupt or reset.

Interrupts are a powerful mechanism of computers designed to allow the more efficient use of system resources. Interrupts allow multiple tasks to happen in a completely random order, controlled by events internal or external to the processor. The HC11 supports 21 independent interrupts. Six of these are classified as nonmaskable interrupts, and the remaining 15 are classified as maskable. In general, maskable interrupts can be controlled (enabled or inhibited) by the user at anytime. Nonmaskable interrupts cannot be controlled by the user.

In reality, the three highest-priority interrupts are reset functions. The reset functions establish the initial conditions of the processor prior to executing the service routine. Without the proper initial conditions, the processor cannot start processing instructions in an orderly fashion.

Chapter Questions

Section 10.1

1. What is the reset state of the S, X and I control bits in the CCR? Why is this state significant?
2. Can the user software set the X bit?

Section 10.2

3. Explain the following terms: interrupt service routine and interrupt request.
4. What purpose does an interrupt service routine serve?
5. Define in your own words: interrupt vector, vector address and vector table.

Section 10.3

6. Explain the difference between local and global interrupt control.
7. Explain the difference between an interrupt mask and an interrupt enable.

Section 10.4

8. What is unique regarding the Strobe A interrupt?
9. What value must be written to the PSELx bits of the HPRIO register to elevate the Timer Overflow Interrupt?
10. If the Real Time and the SCI Serial System interrupts occur during the execution of the same instruction, which interrupt request will be serviced first?
11. If the HPRIO register contains $0D, which of the maskable interrupts has the highest priority?

Section 10.5

12. Where is the JMP instruction located in the Vector Jump Table for the SPI Transfer Complete Interrupt?
13. Why does the EVBU require the Vector Table and the Vector Jump Table?

Section 10.6

14. For what purpose is the Illegal Opcode interrupt used?
15. Can the nonmaskable interrupts be reprioritized?
16. What is the relative priority of the SWI in reference to the other nonmaskable interrupts?

Section 10.7

17. What is the difference between an internal and an external reset?
18. What is the COP? What does it do?
19. Describe the process of arming and servicing the COP timer.
20. What happens if the E clock rate drops below 10 kHz and the Clock Monitor system is enabled?

Section 10.8

21. Describe the process of events that follows a reset condition.
22. What is the state of the Pulse Accumulator after reset?

Chapter Problems

1. If the SP is $0035 and an interrupt occurs, explain what data will be written to the stack and which address locations will be used.
2. If the SP is $0041 and an interrupt occurs, what will the SP change to after the registers are stacked?
3. If the starting address of the IRQISR is $019D, write the instructions necessary to load this starting address into the Vector Jump Table.
4. Write the code necessary to elevate the priority of the RTI interrupt to highest priority.
5. Write an interrupt service routine designed to count the number of 1's in the 8-bit word present at PORTC. Display the value on the monitor. Use the Strobe A interrupt. Originate the code at $0100.

Answers to Self-Test Questions

Section 10.1

1. The I bit must be cleared to enable maskable interrupts.
2. The X bit is located in b_6.

Section 10.2

1. The fetch and all execute cycles are completed for the current instruction before the interrupt is serviced.
2. Nine bytes are pushed to the stack.
3. The RTI instruction is required to complete the interrupt service routine.

Section 10.3

1. The WAI instruction.
2. Execute the SEI instruction. It sets the I bit in the CCR disabling all maskable interrupts.

3. Timer Overflow Interrupt can be locally controlled by the TOI in the TMSK2 register.

Section 10.4
1. IRQE = 0 to select level-sensitive operation.
2. RTI has the second-highest priority of the nonmaskable interrupts.
3. Timer Input Capture 1 has priority 3 and Pulse Accumulator Overflow has priority 12, so the IC1 interrupt would be serviced first.

Section 10.5
1. The vector table is located from $FFD6 to $FFFF in ROM.
2. The first instruction is the JMP instruction located in the Vector Jump Table.

Section 10.6
1. There are six nonmaskable interrupt sources.
2. Yes. XIRQ can be globally unmasked via the X bit in the CCR, COP Failure and Clock Monitor Fail can be locally masked via control bits in the CONFIG and OPTION registers.
3. The SWI is the only interrupt that can be directly caused by a software instruction.
4. Nonmaskable interrupts have higher priority than all maskable interrupts.

Section 10.7
1. An external reset is a stimulus received by the processor via the RESET * pin on the HC11. The internal reset sources come from internal hardware, like the COP.
2. Power-on reset, External reset, COP Watchdog Timer reset and Clock monitor reset.
3. The default COP time-out rate is 16.384 ms.

Section 10.8
1. Stop execution of the current instruction.
   System hardware enters the initial conditions.
   Load the system configuration data from the CONFIG register.
   Establish the mode of operation.
   Load the program counter with the reset vector.
2. The TCNT register is cleared after reset.
3. All timer system flags are cleared after reset.
4. The analog-to-digital converter is disabled because the ADPU bit in the OPTION register is cleared.

chapter 11

Analog Capture—Port E

Objectives

After completing this chapter, you should be able to:

- Explain the theoretical process of analog-to-digital conversion
- Calculate Range, Step Voltage and Resolution
- Predict the digital code generated from any analog input
- Describe the four major functional blocks of the HC11 analog-to-digital system
- Explain the meaning of a channel
- Set up the HC11 for single and multichannel conversions
- Explain the timing of the HC11 conversion sequence
- Interface the LM34/35 temperature sensors from National Semiconductor

Outline

Introduction

As is shown in chapter 1, the HC11 has an internal processor. This processor can process data in a digital format. It requires the data to be in a binary format. In contrast to binary-formatted data, most measurable data, like current, voltage, temperature or speed, is analog by nature. This data is not formatted in a machine-readable binary format; thus it is necessary to convert the analog data to a digital format for processing. Moreover, most measurable data is nonelectrical. Temperature, for example, must be converted from degrees to a voltage for this data to be processed by the HC11. This chapter focuses on the theory behind the conversion process of analog data to a digital format and on how this theory is implemented using the HC11 microcontroller. Specific application examples relevant to this study of the HC11 are provided.

This chapter directly correlates to section 12 of the HC11 Reference Manual and section 10 of the Technical Data Manual.

11.1 Theory of Analog Conversion

Analog data is any data that is continuous in nature. **Continuous** means "any level at any time." The signal can be at any level. For example, 1.23 V, 437.8 mV and 3.89 V are all valid analog input values. Temperatures change continuously. If the temperature is rising, it does not remain at 72 degrees exactly until it has raised a full point to 73. It moves from 72.1 to 72.2 and so on slowly and smoothly until it reaches the next level. On the other hand, **digital data** by nature is discrete. **Discrete** means that the data can only be specific values. It also means that the data is valid only at specific points in time. Digital temperatures are limited to a fixed set of possible temperatures. At any point in time, the temperature can be represented on the digital system only by a discrete value.

Analog-to-digital conversion is the process of classifying ranges of analog input values by assigning a multibit digital code to that range of input values. The number of possible input values is infinite (any signal level). However, the number of digital codes (commonly called steps) is finite. The number of codes is a function of the number of digital bits in the code. For example, the HC11 uses an 8-bit code to represent the input analog values; therefore, there are 2^8 or 256 possible codes or steps (2^n, where n = number of bits in the code).

Analog Data Capture

Many sources of data are in an analog format. Often this data is nonelectrical; it appears in a physical analog format. For example, a temperature is measured in degrees. A digital system like the HC11 cannot directly read degrees. Therefore, the physical analog quantities must be converted to electrical values before they can be captured and converted to digital. A **sensor** or a **transducer** is a device that is used to accomplish this physical-to-electrical conversion. For example, a temperature sensor is used to convert an actual physical temperature to an electrical voltage. The electrical voltage can be captured and converted to digital by the analog-to-digital system.

Figure 11.1 shows the overall conversion process of physical quantities such as temperature, pressure, light, weight or humidity to an analog voltage. The sensor translates the physical quantities to an initial electrical quantity. The output of the

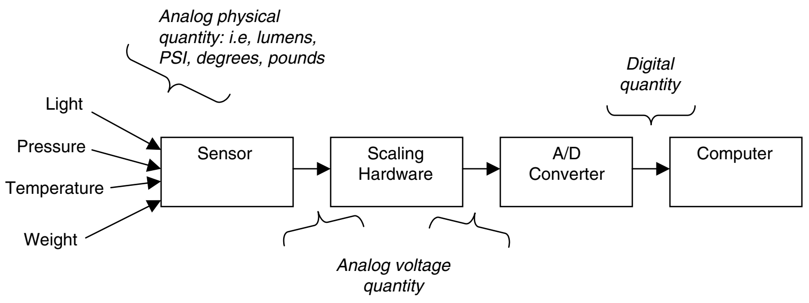


Figure 11.1 Analog Data Acquisition Process

sensor is connected to some scaling hardware. The job of the scaling hardware is to amplify and shift the electrical signal so that it matches the input requirements of the analog-to-digital converter. This analog voltage is then converted to a digital quantity so that the computer can process it.

Range

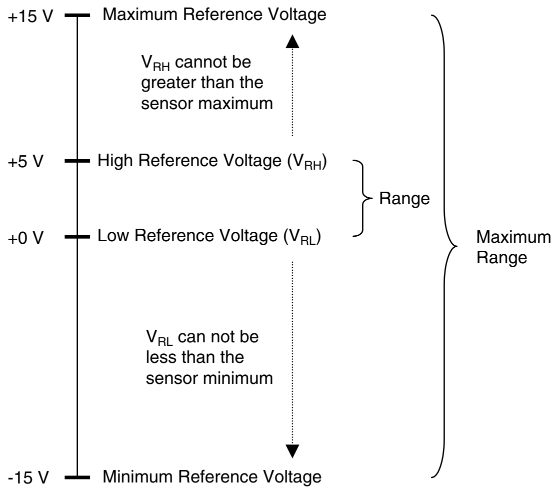
There is a limited number of digital codes that can be generated for a given analog range. Thus, limits must be placed on the analog input signals. The analog inputs will be limited to values that fall between a high and a low reference voltage. The voltage **range** is defined as the difference between the high and low reference voltages. On the HC11 the high reference is referred to as V_{RH} (voltage - reference high) and the low reference voltage is referred to as V_{RL} (voltage - reference low).

$$\text{Range} = V_{RH} - V_{RL} \qquad \text{Equation 11.1}$$

Every A/D system has a maximum and a minimum reference voltage. The **maximum range** is calculated using the maximum and minimum reference voltages. For example, a system is designed to handle analog input values in the range from –15 volts to +15 volts. These points would be the maximum and minimum voltage levels. Using Equation 11.1, the maximum range for this system would be 30 Volts, (+15 V) – (– 15 V). V_{RH} cannot exceed the maximum reference voltage, and V_{RL} cannot be less than the minimum reference voltage. However, a system could be designed so that V_{RH} is 5 volts and V_{RL} is 0.0 volts. This allows a system to focus on only a small portion of the maximum range. The maximums for the system could still be +/– 15 volts, as shown in Figure 11.2. Limiting the range to some subset of the maximum range provides a smaller step size and finer resolution. Step size and resolution are explained in the next two sections.

Steps

The conversion of analog data to digital format requires that analog input values be categorized or quantified into small ranges. A **step** is a small section of the range. Each step within the range has a name or a digital code assigned to it. These codes are always



+15 V
Maximum Reference Voltage
V_{RH} cannot be greater than the sensor maximum
+5 V
High Reference Voltage (V_{RH})
Range
+0 V
Low Reference Voltage (V_{RL})
Maximum Range
V_{RL} can not be less than the sensor minimum
-15 V
Minimum Reference Voltage

Figure 11.2 Relationship of Various Reference Voltages

a sequence of hex numbers starting with the lowest numeric value and increasing to the highest numeric value in the coded range. Figure 11.3 illustrates the relationship of steps, with their corresponding names (digital codes), to the input range.

The number of steps in a range is a function of the number of bits in the resulting digital code. It represents the number of unique codes that can be created with the given number of bits, as shown in Equation 11.2.

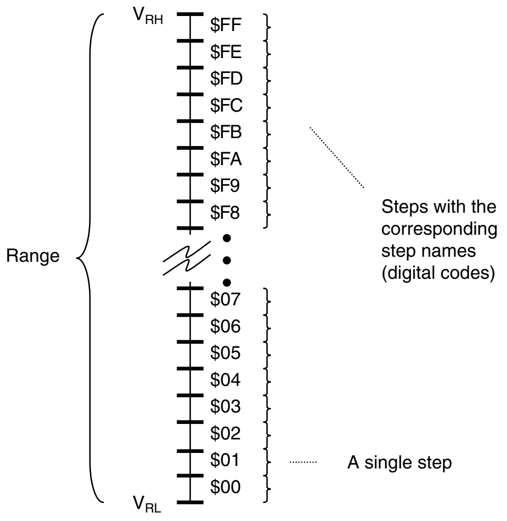


Figure 11.3 Relationship of Steps, Digital Codes and Range

$$\text{Number of Steps} = 2^n \qquad \text{Equation 11.2}$$

where n = the number of bits in the resulting digital code.

NOTE: Because of the view that zero should not be assigned to a step value, some texts and A/D circuitry calculate the number of steps as 2^{n-1}. The HC11 is designed to use the zero code for the first step and thus uses 2^n.

Calculating the Step Voltage

Each step has a unique code; the converter and the processor thus have a way of differentiating each step. The **step voltage** (V_{STEP}) is the size of each step (**step size**), measured in volts. It is calculated by dividing the range by the number of steps, as shown in Equation 11.3:

$$V_{STEP} = \frac{\text{Range}}{\text{Number of steps}} = \frac{V_{RH} - V_{RL}}{2^n} \qquad \text{Equation 11.3}$$

If V_{RH} = 5.0 V and V_{RL} = 0 V, V_{STEP} would be calculated by using Equation 11.3, as follows:

$$V_{STEP} = \frac{V_{RH} - V_{RL}}{2^n} = \frac{5V - 0V}{2^8} = \frac{5V}{256} = 19.53\text{mV}$$

This means that each step is equal to a range of 19.53 mV. Essentially the range is divided into 256 steps, where each step represents a unique 19.53 mV subset of the range. The upper and lower limits to each step are called **step boundaries**. Figure 11.4 illustrates the various steps with their corresponding digital codes and step boundaries. The digital code \$00 is assigned to the first step, which occupies the portion of the range from V_{RL} up to 19.53 mV. The digital code \$01 is assigned to the second step,

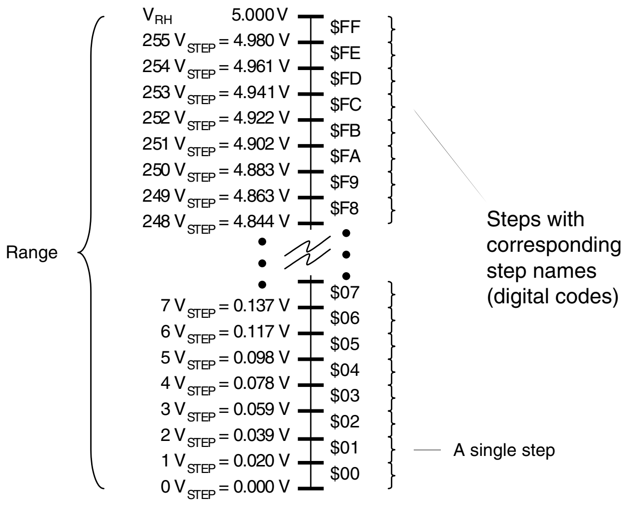


Figure 11.4 Step Boundaries for 8-bit Example

which occupies the portion of the range from 19.53 mV up to 39.06 mV. This process continues through all 256 steps until the entire range is assigned steps and digital codes. The top step, digital code $FF, will be equal to the range from 4.980 V (255 * V_{STEP}) up to V_{RH}.

The reason for having the steps is to provide categories so that the analog input voltages can be converted in an orderly manner. All input voltages that fall within the boundaries of a single step are assigned the digital code for that step. For example, if the input voltage is equal to 12.0 mV, it falls into the first step; therefore, it is assigned the digital code of $00. All analog input values at least 0.0 V and less than 19.53 mV fall into this step and are assigned the digital code of $00. In the same manner, all input voltages of at least 19.53 mV but less than 39.06 mV fall into the second step and are assigned the digital code $01.

Determining the Digital Code

It is intuitive to determine which input values are assigned to the first few steps, but what happens when the input voltage is in the middle of the range? There is a simple method of calculating which step is assigned to any input analog voltage. It is calculated by dividing the analog input voltage (V_{IN}) by the step voltage (V_{STEP}), as shown in Equation 11.4.

$$STEP = \frac{V_{IN} - V_{RL}}{V_{STEP}} \qquad \text{Equation 11.4}$$

If the input voltage is equal to 3.21 V, the step would be calculated as follows:

$$STEP = \frac{V_{IN} - V_{RL}}{V_{STEP}} = \frac{3.21V - 0.0V}{19.53mV} = 164.36\text{steps}$$

Since there is no way of quantifying partial steps, this value is truncated to 164 steps. Some systems round these values, but the HC11 calculates the steps by truncating the remainder, as has been shown. When 164 is converted to hex, the digital code of $A0 results. A quick analysis shows that this is correct. The lower step boundary of the 164th step is calculated as follows:

$$\text{Lower Step Boundary} = 164 \times V_{STEP} = 164 \times 19.53 \text{ mV} = 3.203 \text{ V}$$

Since the upper boundary is equal to the lower boundary of the next step, the upper boundary for the 164th step is calculated as follows:

$$\text{Upper Step Boundary} = 165 \times VSTEP = 165 \times 19.53 \text{ mV} = 3.222 \text{ V}$$

It is evident that the input voltage of 3.21 volts falls between these lower and upper boundaries, thus, the proper step has been assigned.

Resolution

Another term that is important to understand is resolution. **Resolution** is the relationship of each step to the range. The step voltage is the resolution of the system in volts. As the step voltage decreases in reference to the range, the resolution of the system increases. Resolution is also calculated as a percentage of the total range. Percent resolution is calculated as follows:

$$\text{Resolution (\%)} = \frac{100\%}{\text{Number of steps}} = \frac{100\%}{2^n} \qquad \text{Equation 11.5}$$

The similarities of Equation 11.5 to Equation 11.3 are apparent. Both divide the range by the number of steps to calculate the size of a step. In Equation 11.3, the step size is shown in volts, a small piece of the range. In Equation 11.5, the step size is shown as a percentage of the range, where 100% is the whole range.

The percent resolution on the HC11, with its 8-bit converter, is

$$\text{Resolution (\%)} = \frac{100\%}{2^8} = \frac{100\%}{256} = 0.3906\%$$

This means that each step of a 256-step system represents $1/256^{th}$ of the range. Systems with smaller values for percent resolution have greater resolution, i.e. more steps within the range. More steps within a given range increase the number of possible codes and minimize the errors introduced by conversion process.

Self-Test Questions 11.1

1. In your own words, define "range," as it relates to A/D conversion.
2. What is the voltage range when V_{RH} = 3.6 volts and V_{RL} = 0.0 volts?
3. In your own words define the following: step, step voltage and step boundaries, as they relate to A/D conversion.
4. In your own words, define resolution, as it relates to A/D conversion.

11.2 A/D Hardware

The analog-to-digital (A/D) system is an 8-channel multiplexed input converter that utilizes a sample and hold and an 8-bit successive approximation converter. A **channel** is a path for analog data to be converted to digital codes. A charge distribution technique performs the internal sample and hold function. The A/D conversion process can be synchronized to the internal system E clock or configured to be driven by an internal RC oscillator.

The system hardware can be broken down into four functional blocks, as shown in Figure 11.5: analog input multiplexer, analog-to-digital converter, digital control, and result storage registers. Each of these blocks will be discussed individually in the following sections.

Analog Input

As explained in chapter 10, Port E is a general-purpose digital input port. It can also be configured as an analog-to-digital (A/D) converter. When it is used for digital input,

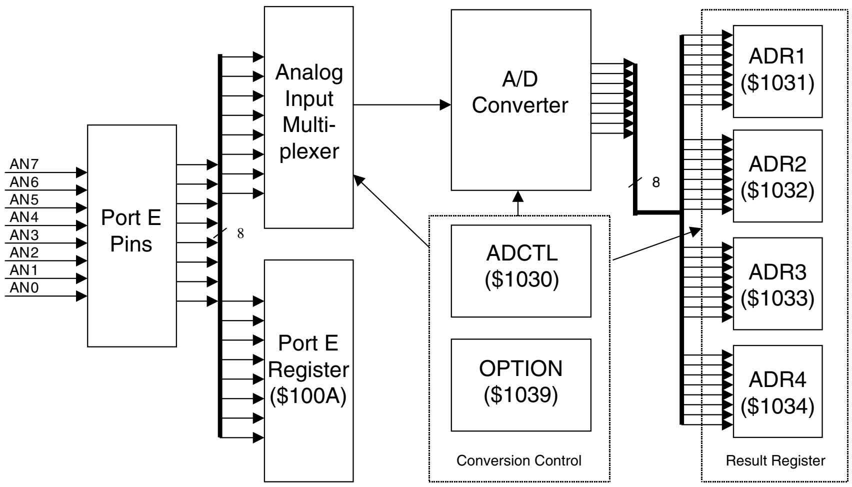


Figure 11.5 A/D Converter Hardware Block Diagram

the input pins are named PE7 through PE0. When the input pins are used for analog input they are named AN7 through AN0, as shown in Figure 11.6.

The analog input process begins at these pins. The analog voltage signals are applied to one or more of the input pins. Thus, the input multiplexer receives up to eight unique analog inputs. The job of the multiplexer is to select one of these eight inputs and forward the selected data signal to the analog converter because only one channel can be converted at a time. Each input path (pin) is called an **analog channel**. Each channel is selected by the CD, CC, CB and CA control bits in the A/D Control Register (ADCTL). This function of the ADCTL register is described later in the conversion control section.

Digital input and analog input are selectable on a bit-by-bit basis. In other words, some bits can be configured to receive digital data at the same time others are configured to receive analog signals.

NOTE: Digital input can continue on inputs not connected to analog sources. The digital signals are stored in the PORTE register, and analog values are processed by the A/D system.

Port E	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
DIGITAL IN	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0
ANALOG IN	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0

Figure 11.6 PORTE Pin Designations and Function Assignments

Analog-to-Digital Converter

The analog channel selected by the input multiplexer is converted to digital in this block. It contains a capacitive array digital-to-analog converter (DAC), a comparator and a successive approximation register (SAR). The DAC array acts as a sample and hold circuit during the conversion sequence, and it provides a comparison voltage to the comparator during the successive approximation conversion process. Each bit of the 8-bit result is calculated using the successive approximation technique, starting with the most significant bit (MSB). As each bit is calculated, it is stored in the SAR. When the conversion is complete, the contents of the SAR are transferred to one of the four result registers.

This text leaves the study of the function of capacitive arrays and the successive approximation conversion method to the student. Further information can be found in sections 12.1 and 12.2 of the Reference Manual.

Conversion Control

The A/D function on the HC11 is controlled by a set of eight control bits and one status flag. These bits reside in two control registers: the Option Register (OPTION) and the A/D Control Regsiter (ADCTL).

The OPTION register contains two control bits relevant to the A/D function: ADPU and CSEL, as shown in Figure 11.7. The ADPU bit (b7) is used to control the power-up status of the A/D hardware block. The ADPU bit acts like a master enable. A 0 in this bit disables the A/D hardware block, and a 1 in this bit enables the A/D hardware block. A delay of as much as 100 s is required after power-up (ADPU set) to allow the charge pump and comparator circuits to stabilize before the converter system can be used.

The CSEL bit (b6) determines which reference clock will be used by the A/D and the EEPROM hardware. When CSEL = 1, an on-chip RC oscillator will be used. When CSEL = 0, the system E clock will be used. Because the E clock is synchronized to the MCU clock, the comparator output can be sampled during relatively quiet times during the MCU clock cycle. Since the internal RC oscillator is asynchronous to the MCU clock, there is more error attributable to clock noise. Therefore, when the internal RC oscillator is being used, the A/D converter accuracy is reduced.

NOTE: CSEL is used to determine the reference clock for the A/D conversion hardware as well as the EEPROM programming hardware. Setting CSEL for the A/D functions automatically reconfigures the EEPROM function.

OPTION		b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1039		ADPU	CSEL	IREQ	DLY	CME	–	CR1	CR0
	RESET	0	0	0	1	0	0	0	0

Figure 11.7 OPTION Register Control Bits Used For A/D Control

ADCTL		b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
\$1030		CCF	–	SCAN	MULT	CD	CC	CB	CA
	RESET	0	0	U	U	U	U	U	U

Figure 11.8 ADCTL Register Control Bit and Status Flag

The ADCTL register contains one status flag CCF (b_7), one unused bit (b_6), and six control bits (b_5–b_0), as shown in Figure 11.8. The ADCTL register controls when to start a conversion, what type of conversion to complete and which channel or channels to scan. CCF is a read-only flag that is set only by the conversion hardware. Once a conversion sequence is invoked by writing to the ADCTL register, the conversion sequence continues without further software intervention. When the conversion is complete, the CCF is set. CCF is cleared by any write to the ADCTL register and is not affected by a read of the ADCTL. CCF is cleared after reset.

The SCAN bit controls continuous/single scan modes. When the SCAN bit is set, conversion sequences will occur continuously. The completion of a sequence will automatically start another sequence without another write to the ADCTL register. When SCAN is cleared, a single conversion sequence will occur. Subsequent conversion sequences can be started only by a write to the ADCTL register.

The MULT bit controls whether (a) one channel will be converted four times or (b) four sequential channels will be converted one time each during the conversion sequence. When MULT is set, four channels are converted, one time each. The CC control bit determines which group of four channels will be converted. When MULT is cleared, a single channel is converted four consecutive times. The CC:CB:CA control bits determine which channel is selected.

The CD:CC:CB:CA bits control which channel or channels will be converted during the sequence, as described in Figure 11.9 and Figure 11.10. There are 16 channels available. They are numbered 1 to 16. Channels 1–8 correspond to the PORTE pins.

Channel Number	Channel Select Control Bits CD	CC	CB	CA	Channel Signal	Corresponding ADRx if MULT = 1
1	0	0	X	X	AN0	ADR1
2	0	0	X	X	AN1	ADR2
3	0	0	X	X	AN2	ADR3
4	0	0	X	X	AN3	ADR4
5	0	1	X	X	AN4	ADR1
6	0	1	X	X	AN5	ADR2
7	0	1	X	X	AN6	ADR3
8	0	1	X	X	AN7	ADR4

Figure 11.9 Converter Multichannel Selection and Assignments *(adapted with permission from Motorola)*

Channel Number	Channel Select Control Bits CD	CC	CB	CA	Channel Signal
1	0	0	0	0	AN0
2	0	0	0	1	AN1
3	0	0	1	0	AN2
4	0	0	1	1	AN3
5	0	1	0	0	AN4
6	0	1	0	1	AN5
7	0	1	1	0	AN6
8	0	1	1	1	AN7

Figure 11.10 Single-Channel Selection and Assignments *(adapted with permission from Motorola)*

Channels 9–16 are used for internal testing and reference and are not used during normal operation.

CD is set only during factory testing; therefore, it is always cleared during normal operation. When MULT = 1, CC selects the upper or lower group of the eight analog input channels: set = upper group (channels 5–8), cleared = lower group (channels 1–4). CB and CA are ignored when MULT = 1. When MULT = 0, CC, CB and CA determine which channel will be converted, as shown in Figure 11.10.

Result Registers

The HC11 is configured to always perform four conversions during a conversion sequence. The 8-bit result of the four conversions are stored in a set of four result registers. Each register has a name and unique address. Each register can be read by the CPU at any time. These registers contain the results of the four conversions. Named ADR1–ADR4, they reside at locations $1031–$1034 in the register block, as shown in Figure 11.11. They are read-only registers. A write to these register addresses has no effect on the contents of the register.

Hardware Connections

The A/D utilizes 10 hardware pins, 8 for the corresponding analog inputs (AN0–AN7), the same pins used for PORTE digital input (PE0–PE7) and 2 for the reference voltage

ADRx								
$1031 ADR1	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1032 ADR2	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1033 ADR3	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1034 ADR4	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
RESET	U	U	U	U	U	U	U	U

Figure 11.11 Analog-to-Digital Result Registers

V_{RH} and V_{RL}. V_{RH} is the high reference voltage or the top end of the reference range. V_{RL} is the low reference voltage or the bottom end of the reference range.

V_{RL} must be greater than or equal to V_{SS} (processor reference voltage - ground). If the V_{RL} drops below V_{SS}, the input can be permanently damaged. V_{RH} must not exceed 6 volts. The best performance occurs when the difference between V_{RH} and V_{RL} is at least 2.5 volts ($2.5V \leq V_{RH} - V_{RL} \leq 6.0V$).

NOTE: Two of the most common A/D application errors occur when connections are made with too much or too little source impedance (series resistance). Too much impedance increases errors due to leakage currents, and too little impedance results in permanent damage to the A/D. Input impedance should be limited to 10 kΩ. Typical connections will use a 1 kΩ series resistor.

Self-Test Questions 11.2

1. What is the default use (state after reset) of the PORTE input pins?
2. How many analog input channels are provided?
3. How many bits are in each digital value are calculated during a conversion on the HC11?
4. How many channels are converted before the CCF is set?
5. Which register is programmed to select single or continuous scan?

11.3 A/D Function

After hardware reset, the PORTE input pins are designated as digital inputs that can be accessed by the PORTE register. In order to make these inputs receive and process analog data, the A/D converter must be configured as follows:

1. Build a control word for the OPTION register to power-up the A/D converter by setting the ADPU bit (ADPU = 1) and selecting the clock source (CSEL = 0/1).
2. Write this control word to the OPTION register.
3. Build a control word for the ADCTL register by answering the following questions:
   a. Should the conversion sequence be performed once or continuously (SCAN = 0/1)?
   b. Is one channel to be converted four times, or are four separate channels to be converted once each (MULT = 0/1)?
   c. Which channel or channels are to be converted (CD, CC, CB, CA)?
4. Write this control word to the ADCTL register (remember any write to this register starts a conversion sequence and clears the CCF bit).

Following are two examples showing how this is done.

Example 11.1

Problem: What has to be written to the ADCTL register to start a single scan of channel 7 and then wait for the conversion to complete?

Solution: The first step is to build a control word for the OPTION register. On the EVBU, BUFFALO activates the A/D system by setting the ADPU bit in the OPTION register; no further action is necessary. Next, a control word for the ADCTL register must be built. Since the CCF is a read-only status flag, what is written to b_7 is irrelevant. As the unused bit will always be read as a zero, what is written to b_6 is also irrelevant. To perform a single scan, SCAN = 0. To scan a single channel, MULT = 0. Using Figure 11.10, channel 7 requires CD:CC:CB:CA = 0110. These choices result in the following control word $06.

ADCTL	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1030	CCF	–	SCAN	MULT	CD	CC	CB	CA
Control word	0	0	0	0	0	1	1	0

When this control word is written to the ADCTL register, a conversion sequence is started that samples and converts the single channel 7 (AN6) four successive times and writes the results into the result registers ADR1 through ADR4 respectively. The following codes show how to write the control word to the ADCTL register and wait for the CCF to set.

```
        LDAA   #$06    ;SCAN = 0, MULT = 0, CD:CC:CB:CA = 0110
        STAA   $1030   ;Start conversion sequence
LOOP1   LDAA   $1030   ;Test for CCF = 1
        BPL    LOOP1   ;  If CCF = 0 read ADCTL again
```

Example 11.2

Problem: What has to be written to the ADCTL register to start a single scan of channels 1–4 and then wait for the conversion to complete?

Solution: As shown in the previous example, b_7 and b_6 are irrelevant. To perform a single scan, SCAN = 0. To scan multiple channels, MULT = 1. Using Figure 11.9, channels 1–4 require CD:CC = 00 and CB:CA are irrelevant. In this case they are cleared, but can be any binary combination (00, 01, 10, or 11). When this control word is written to the ADCTL register, a conversion sequence is started that samples and converts the signals on channels 1 (AN0) through channels 4 (AN3) and writes the results in ADR1 through ADR4 respectively.

ADCTL	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1030	CCF	–	SCAN	MULT	CD	CC	CB	CA
Control word	0	0	0	1	0	0	0	0

```
      LDAA   #$10   ;SCAN = 0, MULT = 1, CD:CC:CB:CA = 0000
      STAA   $1030  ;Start conversion sequence
WAIT2 LDAA   $1030  ;Test for CCF = 1
      BPL    WAIT2  ;  If CCF = 0 read ADCTL again
```

Conversion Sequence and Timing

The A/D function on the HC11 is configured always to perform a set of four conversions. This is called a conversion sequence. If the MULT bit is set in the ADCTL register, the conversion sequence will consist of four consecutive channels. The four channel groups are limited to the upper set of four channels (1–4) and the lower set of four channels (5–8). When the MULT bit is cleared, four consecutive conversions of a single channel are performed during the conversion sequence.

The timing of the conversion sequence is dependent on the status of the CSEL bit in the OPTION register. Whenever possible the CSEL bit should be cleared, selecting the E clock as the master clock for the A/D conversion sequence. When E clock is used as the master clock for the A/D converter, all A/D functions are synchronized with the system clock. Sampling takes place at relatively quiet times in the system clock cycle, thus reducing the effects of internal MCU noise. Also, the result register updates automatically occur during a portion of the system clock cycle when reads do not occur; therefore, an update to a result register cannot interfere with another system data read.

When the internal RC oscillator is used as the A/D master clock, the A/D functions are asynchronous to the system clock. There is more error attributable to internal system noise since sampling occurs anytime within the system clock cycle, rather than at a relatively quiet time. Potential conflicts that could arise between result register updates and normal read operations are avoided by the addition of a delay at the end of each channel conversion to allow for synchronization to the system E clock.

When system clock speed is less than 750 kHz, the on-chip RC oscillator should be used to minimize the potential for increased error due to charge leakage at temperature extremes. When the system clock is greater than 750 kHz, the E clock should be used because of the advantages described earlier.

Figure 11.12 shows a complete conversion sequence in relation to E clock, when CSEL = 0. This timing diagram can be used to determine the earliest availability of valid data for each converted channel. For example, ADR1 has valid conversion results 32 E clocks after a write to the ADCTL register, ADR2 will be available in 64 E clocks, and so on.

NOTE: A conversion sequence is initiated by a write to the ADCTL register. A conversion sequence in progress can be aborted by performing another write to the ADCTL register, thus initiating a new conversion sequence.

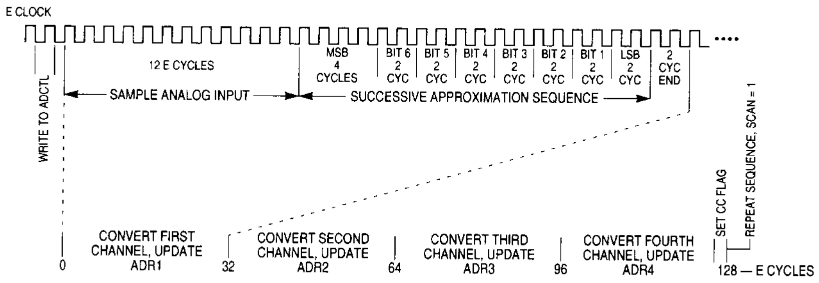

E CLOCK
WRITE TO ADCTL
12 E CYCLES
SAMPLE ANALOG INPUT
MSB 4 CYCLES
BIT 6 2 CYC
BIT 5 2 CYC
BIT 4 2 CYC
BIT 3 2 CYC
BIT 2 2 CYC
BIT 1 2 CYC
LSB 2 CYC
2 CYC END
SUCCESSIVE APPROXIMATION SEQUENCE
SET CC FLAG
REPEAT SEQUENCE, SCAN = 1
CONVERT FIRST CHANNEL, UPDATE ADR1
CONVERT SECOND CHANNEL, UPDATE ADR2
CONVERT THIRD CHANNEL, UPDATE ADR3
CONVERT FOURTH CHANNEL, UPDATE ADR4
0
32
64
96
128 — E CYCLES

Figure 11.12 HC11 Conversion Sequence Timing *(courtesy of Motorola)*

Self-Test Questions 11.3

1. How does the user start a conversion sequence?
2. When CSEL = 0, how many E clock cycles are required to convert a single channel?
3. How many clock cycles are required to complete a conversion sequence?

11.4 Example Using an LM34/35 Temperature Sensor

The A/D functions on the HC11 have the job of capturing analog data and converting it into a digital format. Since real-world sensors produce data in an analog form, the PORTE A/D functions are ideal to access and process this data. For example, the LM34 temperature sensor is a device that produces a 10 mV /˚F linear output voltage. The device is proportional and linear across the range of temperatures between –50˚F and +300˚F. 0˚F will generate the output voltage of 0 mV, and 300˚ F will generate a 3.0 V (3,000 mV) output. National Semiconductor also has a Celsius temperature sensor, the LM35. It is identical to the LM34, except it generates 10 mV/˚C.

The rest of this section will show how the LM34 is interfaced to the HC11, how the HC11 processes the data and how it displays the results on a monitor using BUFFALO. Figure 11.13 shows how the hardware would be connected. AN7 has been selected since AN0 is used by the EVBU to determine EEPROM boot. In this particular circuit, temperatures below 0˚F will produce negative voltages to the HC11. Since the LM34

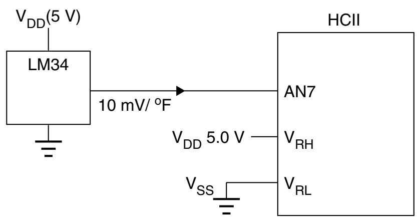


Figure 11.13 Digital Thermometer Hardware Diagram

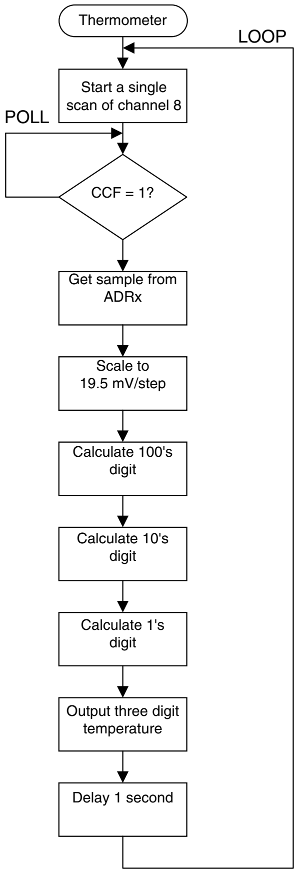


Figure 11.14 Flowchart of Thermometer Example

will not produce negative voltages greater than the –1 V maximum allowed by the HC11, no damage will be caused.

Digital thermometer software is a bit more involved than the hardware connections. A flowchart and the source code for the program are shown in Figures 11.14 and 11.15 respectively. The software must sample and convert the temperature input and then convert it to the decimal temperature value. A single subroutine in BUFFALO allows the temperature to be displayed on the monitor. The following code is an example of how this software may be written. It sets the A/D converter up to convert the data on AN7 using a single-channel, noncontinuous conversion. Since the converted code is in hex, it must be converted to decimal before display. This routine uses a common dividing and scaling scheme to accomplish the conversion. The resulting decimal digits are stored as 4-bit BCD digits in the rightmost digits of the two bytes called BUFFER. The program uses the internal BUFFALO subroutines OUT2BS and OUTCRL to

```
**********************************************************
* HC11 Digital Thermometer Example
*
* This program outputs to the monitor the decimal
* temperature in degrees Fahrenheit using an LM34
* temperature sensor and PORTE
**********************************************************

ADCTL   EQU     $1030    Define labels to make code
ADR1    EQU     $1031      more readable.

OUT2BS  EQU     $FFC1    BUFFALO subroutines - output two bytes
OUTCRL  EQU     $FFC4      - output carriage return/line feed
DLY10MS EQU     $E2E5      - internal 10 ms delay

        ORG     $0000
BUFFER  RMB     2        Temporary storage of digital temp

        ORG     $0100    Starting address of executable code

LOOP    LDAA    #$07     single scan of AN7
        STAA    ADCTL    Start the conversion sequence

POLL    LDAA    ADCTL    Wait until conversion is complete
        BPL     POLL

        LDAA    ADR1     Get the digital temp from result reg
        LDAB    #195     scale to 19.5 mV step size
        MUL
        LDX     #100     remove hundreds scale
        IDIV
        XGDX
        LDX     #100     calculate hundreds digit
        IDIV             hundreds digit in X, remainder in D
        XGDX             hundreds in B, remainder in X
        STAB    BUFFER   save hundreds until ready to display
        XGDX
        LDX     #10      calculate tens & ones digits
        IDIV             tens digit in X, ones digit in D
        STAB    BUFFER+1 save ones digit temporarily
        XGDX             tens digit to B
        LSLB             move tens digit to 4 MSBs
        LSLB
        LSLB
        LSLB
        ADDB    BUFFER+1 add ones digit to tens
        STAB    BUFFER+1 save tens and ones

        LDX     #BUFFER  load X with address of temp to display
        JSR     OUT2BS   display the temp
        JSR     OUTCRL   output carriage return/line feed

        LDAB    #100     wait 1 sec before it is done again
WAIT1   JSR     DLY10MS  BUFFALO 10 ms delay
        DECB
        BNE     WAIT1    100, 10 ms delays complete?
        BRA     LOOP     go back to top and do it again
```

Figure 11.15 Source Code of Thermometer Example

display the temperature on the monitor. Finally, the program waits one second by calling the BUFFALO 10 ms delay subroutine and then repeating the program continuously.

Self-Test Questions 11.4

1. What is the output voltage of an LM34 when the input temperature is 34 degrees?
2. What is the output voltage of an LM34 when the input temperature is 218 degrees?

Summary

Analog-to-digital conversion is the process of capturing a continuous signal and converting it to multibit digital code. The theories behind range, steps and resolution were presented. The range is the difference between the high and low reference voltages. A step is a small section of the range that has a unique name or a code assigned to it. The resolution is the ratio of the "size of the step" to the "size of the range." It is normally expressed in percentage.

Port E is a general-purpose digital input port that can also be configured as an analog input port. When configured as an analog input, up to eight independent analog channels can be accessed by the internal A/D conversion system. Each input channel can be sampled and converted to digital and stored in up to four result registers. A conversion sequence consists of four successive conversions of either a single channel or a group of four channels. Conversion sequences can be converted on a continuous or single sequence basis. Conversion sequences are initiated by writing a control word to the ADCTL register.

Chapter Questions

Section 11.1

1. What is the voltage range when V_{RH} = 4.0 volts and V_{RL} = 1.5 volts?
2. What is the step voltage on the HC11 if the voltage range is 3.456 volts?
3. What is the step voltage if V_{RH} = 4.096 volts and V_{RL} is grounded?
4. Calculate the digital code that will be assigned to the analog input of 3.65 V, if V_{STEP} = 16 mV and V_{RL} = 0 V.
5. Calculate the digital code that will be assigned to the analog input of 1.985 V, if V_{STEP} = 5 mV and V_{RL} = 1.2 V.
6. What is the percent resolution of a 12-bit A/D converter?
7. What is the percent resolution of an 8-bit A/D converter?

Section 11.2

8. When not being used for A/D, is Port E an input or an output port?
9. What label names are associated with the PORTE inputs when they are configured for analog input?
10. What hardware pin number is used for PE3/AN3?
11. What is the address of the register that receives digital data from the PEx pins?
12. Assume that channel 1 and channel 2 both have valid analog input signals applied and the A/D converter performed a multichannel conversion. At what addresses will the digital results of the channels be stored?
13. What is the address of the OPTION register?
14. What is the address of ADR4?
15. How many channels of the A/D inputs can be multiplexed at one time?
16. Which register selects the analog channel to be converted?
17. Which bit in the OPTION register must be set to power-up the A/D?
18. Which bit in the ADCTL register is set to indicate that a conversion has been completed?
19. During a multichannel conversion, which ADR register will contain the digital sample of channel 5?
20. Can the user write data to one of the ADRx registers?

Section 11.3

21. What hex value needs to be loaded into the lower nibble of the ADCTL to select channel 5 for conversion?
22. Fully describe what will be accomplished if $10 is written to the ADCTL.
23. How long in seconds does it take to complete the conversion sequence?
24. If the E clock was 3.2 MHz, how long in seconds does it take to complete the conversion sequence?

Section 11.4

25. Which LM series temperature sensor is used to convert temperature from degrees C?
26. What output voltage will be produced on an LM34 from an input temperature of 159 degrees?
27. What is the minimum resolution in degrees of a system that has an LM34 directly connected to AN7, V_{RH} = 5.0 V and V_{RL} = 0.0 V?
28. What could be done to increase the resolution to 1/2 of a degree in the system from question 27?

Chapter Problems

1. Calculate the range of voltage for the steps associated with the codes $3A, $57, $9D, and $E2, when V_{RH} = 4.0 V and V_{RL} = 0.8 V.
2. Calculate the ideal range of a system using an LM34 for temperature measurement where the desired range of temperatures is between 50˚F and 250˚F. Take measures to optimize the resolution and step size of the system for better accuracy.
3. Draw a flowchart and write a program that will convert the single channel connected to AN5. Store the results of the conversion sequence in a 4-byte memory buffer located at $0010 in RAM. Originate the program at $0100.
4. Draw a flowchart and write a program that will perform a multichannel conversion that includes channel 3. Store the results of the conversion sequence in a 4-byte memory buffer located at $01A0 in RAM. Originate the program at $0120.
5. Draw a flowchart and write a program that will perform a multichannel conversion of channels 5–8. Store each of the four results on LEDs that are connected to the lower four bits of PORTB. Write a two-second time delay and execute between each write to PORTB. Originate the program at $0160.

Answers to Self-Test Questions

Section 11.1

1. Range is the set of possible input voltages that are allowed in the system.
2. Range = $V_{RH} - V_{RL}$ = 3.6 – 0.0 = 3.6 volts.
3. Step is the smallest level of resolution of a system. Step voltage is a small voltage range associated with a single step. Step boundaries are the upper and lower voltage limits of a step.
4. Resolution is the percentage of the whole range that is associated to a single step.

Section 11.2

1. The PORTE pins are configured for digital input after reset.
2. The HC11 can support 8 independent analog input channels.
3. The HC11 is an 8-bit conversion system.
4. The HC11 is configured to always convert four channels before the CCF is set.
5. The ADCTL register controls the type of conversion to complete.

Section 11.3

1. A conversion sequence is started by writing to the ADCTL register.
2. A single channel requires 32 E clock cycles.
3. A conversion sequence requires 128 E clock cycles.

Section 11.4

1. 34 degrees × 10 mV / degree = 340 mV.
2. 218 degrees × 10 mV / degree = 2.18 V.

chapter 12

Timed Events—Port A

Objectives

After completing this chapter, you should be able to:

- Define the concept of a timed event
- Describe the pin assignments of the Port A timing system
- Show understanding of status flags and interrupt mask bits
- Capture the time of input events using input capture
- Calculate pulse width and period of input signals from times captured by input-capture hardware
- Cause output events to happen at specified times
- Generate pulses and square waves on OCx output pins
- Use input capture in conjunction with output compare to generate time delays
- Count the number of occurrences of input events using event mode in pulse accumulator
- Calculate the pulse width using gated mode in pulse accumulator
- Cause periodic or timed interrupts to occur via the real-time interrupt hardware

Outline

Introduction

Port A is a general-purpose I/O port that can also be configured for a variety of timer functions. This chapter will present some theory behind the concept of timed events and then address the five major functions performed within the HC11 timing system:

1. Timing event master control
2. Input-capture: Capturing the time at which input events occur
3. Output Compare: Causing output events to occur at specified times
4. Pulse Accumulation: Ability to count events
5. Real-Time Interrupts: Events occurring on a periodic basis

The timer system uses more registers and control bits than any other system within the HC11. Each of the input-capture and output compare features has a dedicated 16-bit data register, control bits to select the particular type of event, an event flag and an interrupt enable. The pulse accumulator uses an 8-bit accumulator, event control bits and two independent flags and interrupt enable bits. The real-time interrupt feature has additional control bits. Each data register will be described, as well as the control bits specific to each timer system function, in the following sections.

Although many of the control bits within the timer system are hardwired to specific timer functions, the timer system is essentially software driven. The system is easily adaptable via the programmable registers to a wide range of timer-related functions.

Since Port A contains only eight bits, each bit serves multiple functions, as shown in Figure 12.1. For example, if the IC4 feature is being used, then the OC5, OC1 and b_3 digital I/O feature must be disabled. When the pulse accumulator input (PAI) is being used, digital I/O on b_7 is disabled. Some of the features can be active simultaneously on the same pins. The OC1 features can be active at the same time as any of the other OCx functions. OC1 and pulse accumulation can also be active concurrently. In this case, the pulse accumulator will count events caused by the OC1 output. Further information regarding all of these features appears throughout this chapter.

This chapter directly correlates to sections 10 and 11 of the HC11 Reference Manual and section 9 of the Technical Data Manual.

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Port A	In/Out	Out	Out	Out	In/Out	In	In	In
DIGITAL I/O	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
IC FUNCTION	-	-	-	-	IC4	IC1	IC2	IC3
OC FUNCTION	OC1	OC1 OC2	OC1 OC3	OC1 OC4	OC1 OC5	-	-	-
PA FUNCTION	PAI	-	-	-	-	-	-	-

Figure 12.1 Port A Pin Designations and Function Assignments

12.1 Theory of Timed Events

All microcomputing systems have a need for functions to occur at specific points in time. For example, the digital clock on the PC is a timer system that increments the seconds, minutes and hours. A timed event in this case occurs on a one-second interval. This event causes the clock to count.

In general, an **event** is a signal transition, high to low or low to high, but can be a variety of other occurrences. An **input event** is a signal coming from some external source and is sensed by the HC11. An **output event** is created by some function within the HC11 and is output to some external circuit.

An input event is sensed by an edge transition on an input signal. For example, a control system needs to keep track of what time the lights were turned on in a particular room. The system would need to monitor the power on the line. No power would be a 0 V state. When power was applied by turning on the light, the line power would transition to a high state, which would be sensed and interpreted as an input event. The input-capture system would then store the time at which this event happened.

Input events have the tendency to be random by nature; thus they occur at many different times. Often, the time at which the event occurs is more significant than the event itself. In other cases, the frequency of events within a specified period is significant; thus the ability to count events is important. The HC11 supports each of these methods to handle and process input events. These types of events will be discussed in the input-capture and pulse accumulation sections later in this chapter.

An output event is represented by a transition on an output signal. For example, a control system needs to automatically turn off the lights 15 minutes after closing time. A control register is loaded with the time the event is to take place, for example, 15 minutes after closing. The system then compares the system clock to the event time. When the system clock matches the designated event time, the output event will take place. The output event, in this case, would create a state change on the output signal, which would signal the lights to be shut off.

Output events are often controlled by some input event. The previous example could be viewed this way. The input event of "closing time" triggered the output event of "turning out the lights." This approach allows complete flexibility in regards to the time the output events take place. Output events can also occur on a periodic basis: Every minute, go check the fluid level in the tank, the temperature in the oven, the state of a process. Periodic events allow background tasks to be controlled without the normal required overhead.

Self-Test Questions 12.1

1. In your own words, describe an event.
2. Define an input event.
3. Define an output event.

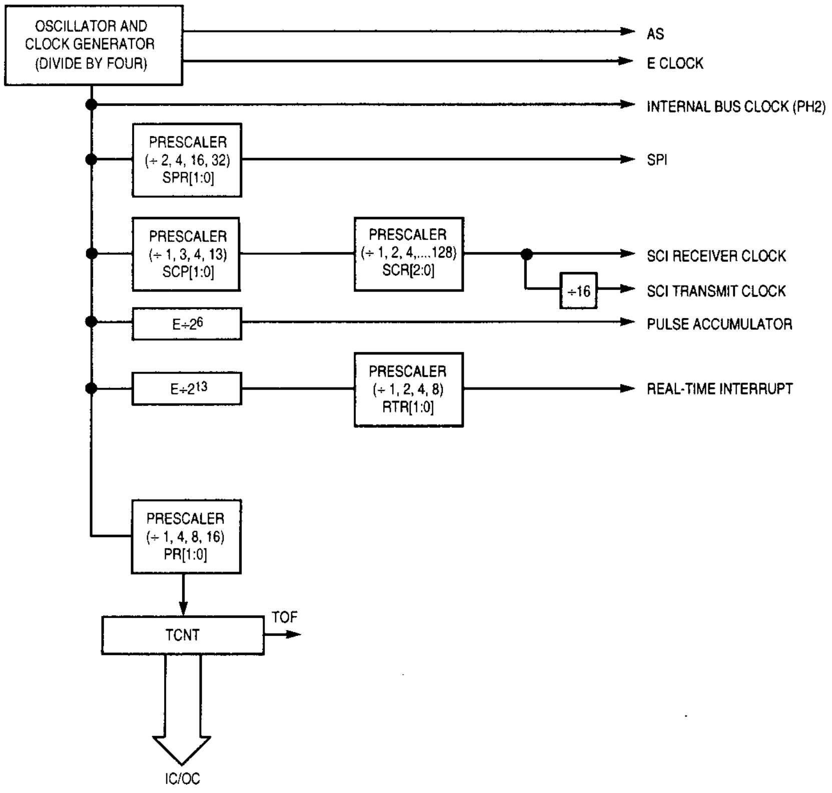

OSCILLATOR AND CLOCK GENERATOR (DIVIDE BY FOUR)
AS
E CLOCK
INTERNAL BUS CLOCK (PH2)
PRESCALER (÷ 2, 4, 16, 32) SPR[1:0]
SPI
PRESCALER (÷ 1, 3, 4, 13) SCP[1:0]
PRESCALER (÷ 1, 2, 4,....128) SCR[2:0]
SCI RECEIVER CLOCK
÷16
SCI TRANSMIT CLOCK
E÷2^6
PULSE ACCUMULATOR
E÷2^13
PRESCALER (÷ 1, 2, 4, 8) RTR[1:0]
REAL-TIME INTERRUPT
PRESCALER (÷ 1, 4, 8, 16) PR[1:0]
TCNT
TOF
IC/OC

Figure 12.2 Timer Clock Divider Chains (*adapted with permission from Motorola*)

12.2 Main Timer Functions

The HC11 timing system is comprised of five major timing chains. Three of these chains are tied to features of Port A, and the other two chains are tied to features of Port D, which are discussed in the next chapter. As shown in Figure 12.2, all timing chains are a derivative of the internal bus clock, which is at the same frequency as E Clock. Therefore, all Port A timing functions are a function of the E Clock frequency and are synchronized to the phase of the internal bus clock.

The main clock divider chain drives a 16-bit free running counter called the Timer Counter register (TCNT). The user cannot change the contents of the counter, thus the name "**free-running**" counter. The 16-bit counter occupies two bytes of register space at locations $100E and $100F, and it is cleared after reset, as shown in Figure 12.3.

Since the TCNT register is a 16-bit counter, it can count from $0000 to $FFFF, or 0 through 65,535. When it reaches the maximum count, the register rolls over and starts

TCNT								
\$100E (HI)	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8
\$100F (LO)	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
RESET	0	0	0	0	0	0	0	0

Figure 12.3 TCNT Register Layout

the sequence over again. Each time the register rolls over, the timer overflow flag (TOF) is set in the Timer Interrrupt Flag Register 2 (TFLG2). TOF indicates that the TCNT register has rolled over; thus the value in the TCNT register represents the number of counts since the last time TCNT rolled over. If the Timer Overflow Interrupt Enable (TOI) bit of the Timer Interrupt Mask Register 2 (TMSK2) is set when the timer overflow occurs, an interrupt will be requested. The location of the TOF and TOI bits within the TFLG2 and TMSK2 registers is shown in Figure 12.4.

The value in the TCNT register *always* indicates the number of counts since the last time the register overflowed (rolled over). By default, the clock for the counter is E clock; therefore, the value in TCNT is the number of E clocks since the last time the register overflowed. If the desired clock rate is slower than the E clock rate, the clock rate can be reduced by changing the programmable prescaler. Slowing the clock rate slows the TCNT register count rate and overflow rate.

The prescaler can be set to one of four scale values, E/1, E/4, E/8 or E/16, as shown in the timing chain shown in Figure 12.2. The default scale factor is 1; thus the timer count is E clock divided by 1, or the E clock rate. The scale value is selected by the value stored in the Timer Prescaler Select bits (PR1, PR0) in the TMSK2 register (location of these bits is shown in Figure 12.4). Figure 12.5 contains the timing characteristics of the TCNT register in relation to the value stored in PR0 and PR1.

NOTE: The TMSK2 register is one of four protected control registers (BPROT, INIT, OPTION and TMSK2). In order to protect the sensitivity of the PR1 and PR0 bits in the TMSK2 register, write operations are restricted. The PR1 and PR0 bits in the TMSK2 register can be written only once, and that write must occur in the first 64 machine cycles after reset.

TFLG2	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
\$1025	TOF	RTIF	PAOVF	PAIF	–	–	–	–
RESET	0	0	0	0	0	0	0	0

TMSK2	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
\$1024	TOI	RTII	PAOVI	PAII	–	–	PR1	PR0
Reset	0	0	0	0	0	0	0	0

Figure 12.4 Timer System Control and Status Bits in the TFLG2 and TMSK2 Registers

Control Bits		Scale	Timer Count	Clock	Overflow
PR1	PR0	Value	(E/x)	Period (ns)	Period (ms)
0	0	1	E/1	500	32.768
0	1	4	E/4	2,000	131.072
1	0	8	E/8	4,000	262.144
1	1	16	E/16	8,000	524.288

Figure 12.5 TCNT Register Timing for 2 MHz E clock

Clearing Timer Flags

The flag bits, within the Port A timing system, are cleared by writing a 1 to the respective bit. The most common method used to clear a timer flag is to build a control word and then write this control word to the corresponding flag register. This is accomplished by loading an accumulator with a mask that has bits set corresponding to the flags to be cleared. This mask value is then written to the Timer Interrupt Register 1 or 2 (TFLG1 or TFLG2). The write operation acts like an XOR operation. The contents of the flag register are essentially XORed with the mask. Individual flag bits will be cleared by the 1's in the mask. Zeros in the mask have no effect on the flag register bits.

Example 12.1

Problem: Assume that the TOF and the Pulse Accumulator Overflow (PAOVF) flags are set in the TFLG2 register. Clear TOF in the TFLG2 register by writing a mask to the register. This operation should have no effect on the other bits in TFLG2.

Solution: The register data is $A0 (TOF and PAOVF set). If the user wishes to clear the TOF flag only, a mask of $80 would be built and then written to the flag register, as shown with the following lines of code.

```
LDAA   #%10000000     Build mask to clear TOF
STAA   $1025          Store mask at TFLG2 register
```

Conceptually, an XOR operation takes place when these two lines of code are executed:

```
       $A0    1 0 1 0 0 0 0 0      Contents of flag register before write
XOR    $80    1 0 0 0 0 0 0 0      Mask, isolating the TOF flag only
       ----------------------
       $20    0 0 1 0 0 0 0 0      Result in flag register after store
```

The BCLR instruction can also be used to accomplish the same task; however, the mask supplied with the BCLR instruction must have 0's in the bit positions corresponding to the flags that are to be cleared. All other positions in the BCLR mask must be set to 1's. The BCLR instruction works because it performs an AND operation on the data in the flag register, using the inverted value from the mask.

Example 12.2

Problem: Assume that the TOF and the Pulse Accumulator Overflow (PAOVF) flags are set in the TFLG2 register. Clear TOF in the TFLG2 register using only a BCLR instruction. This operation should have no effect on the other bits in TFLG2.

Solution: The register data is $A0 (TOF and PAOVF set). If the user wishes to clear the TOF flag only using the BCLR instruction, a mask of $7F would be built so that the bit is identified by 0's in the mask.

```
BCLR  $25,X $7F    Clear TOF using $7F as mask
```

The BCLR instruction reads the data from the register, ANDs it with the inverted mask, then writes this result back to the register. The write operation has the same effect as the store operation in Example 12.1. Conceptually, an XOR operation takes place that clears TOF.

	$A0	1 0 1 0 0 0 0 0	Contents of flag register before write
XOR	$80	1 0 0 0 0 0 0 0	Inverse of BCLR mask, isolating the TOF flag only
	$80	0 0 1 0 0 0 0 0	Result that is then written back to flag register
XOR	$A0	1 0 1 0 0 0 0 0	Contents of the flag register during write
	$20	0 0 1 0 0 0 0 0	Result in flag register after write

It is not safe to use the BSET instruction to clear flags in the timer flag registers. This instruction performs an OR operation on the data in the flag register using the mask. Thus, flags that were previously set in the flag register may be inadvertently cleared when this data word is written back to the flag register.

NOTE: A BSET instruction used for the purpose of clearing flags will clear all flags that were set in the register regardless of the value in the mask. Thus, the BSET instruction should NOT be used to clear individual flags.

Since there are several instruction sequences that can be used to clear timer flags, Figure 12.6 summarizes some of the options. Each sequence requires a different number of bytes of object code and machine cycles to execute; the best sequence is a function of user preferences (i.e., minimum program size versus minimum execution time). Since some of the sequences provided require the indexed addressing mode,

Example Number	Instruction Sequence		Address Mode	Bytes	Machine Cycles	Total Bytes	Total Cycles
1	LDAA	#$80	IMM	2	2	5	6
	STAA	$1025	EXT	3	4		
2	LDAA	#$80	IMM	2	2	4	6
	STAA	$25,X	IND,X	2	4		
3	LDAA	#$80	IMM	2	2	5	7
	STAA	$25,Y	IND,Y	3	5		
4	BCLR	$25,X $7F	IND,X	3	7	3	7
5	BCLR	$25,Y $7F	IND,Y	4	8	4	8

Figure 12.6 Instruction Sequences to Clear TOF

the index register must be loaded with the proper address to point to the register space. Direct addressing mode sequences are not shown since the default register space on the EVBU is located from $1000 to $103F, which is outside the direct addressing modes' limited zero page address range ($0000 to $00FF).

Relationship of Flag Bits and Interrupt Enable Bits

Each timer system function has associated status flag bits and local interrupt enable bits. There are twelve flag/interrupt enable bit pairs. The flag bits are located in two Timer Interrupt Flag registers (TFLG1 and TFLG2). The local interrupt enable bits are located in two Timer Interrupt Mask registers (TMSK1 and TMSK2). The flag bits in the flag registers correspond bit for bit with the enable bits in the mask registers. For example, the TOF bit is b7 of the TFLG2 register, and the corresponding interrupt enable bit is b7 of the TMSK2 register, as shown in Figure 12.7. The register pairs are shown in Figure 12.7 to emphasize this relationship between the flag bits and the interrupt enable bits. The specific application of these bits will be discussed in subsequent sections of this chapter.

When interrupts are enabled, the corresponding flag bits indicate to the interrupt control logic that the event has taken place. This causes an interrupt to be requested. During the interrupt service routine, the user must clear the flag bit. If the user neglects to clear the flag bit, when the process returns from the interrupt service routine it will again check the flag bit and determine that an event has occurred during the prior servicing. This will cause the hardware to erroneously request another interrupt. The process will then be stuck in an endless interrupt-servicing loop.

NOTE: Flag bits must be cleared during the servicing of an interrupt or the hardware will assume that another event has occurred while the previous event was being serviced. If the flag is not cleared, the software will be stuck in an endless interrupt loop.

TFLG1		b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1023		OC1F	OC2F	OC3F	OC4F	I4/O5F	IC1F	IC2F	IC3F
	RESET	0	0	0	0	0	0	0	0

TMSK1		b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1022		OC1I	OC2I	OC3I	OC4I	I4/O5I	IC1I	IC2I	IC3I
	RESET	0	0	0	0	0	0	0	0

TFLG2		b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1025		TOF	RTIF	PAOVF	PAIF	–	–	–	–
	RESET	0	0	0	0	0	0	0	0

TMSK2		b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1024		TOI	RTII	PAOVI	PAII	–	–	PR1	PR0
	Reset	0	0	0	0	0	0	0	0

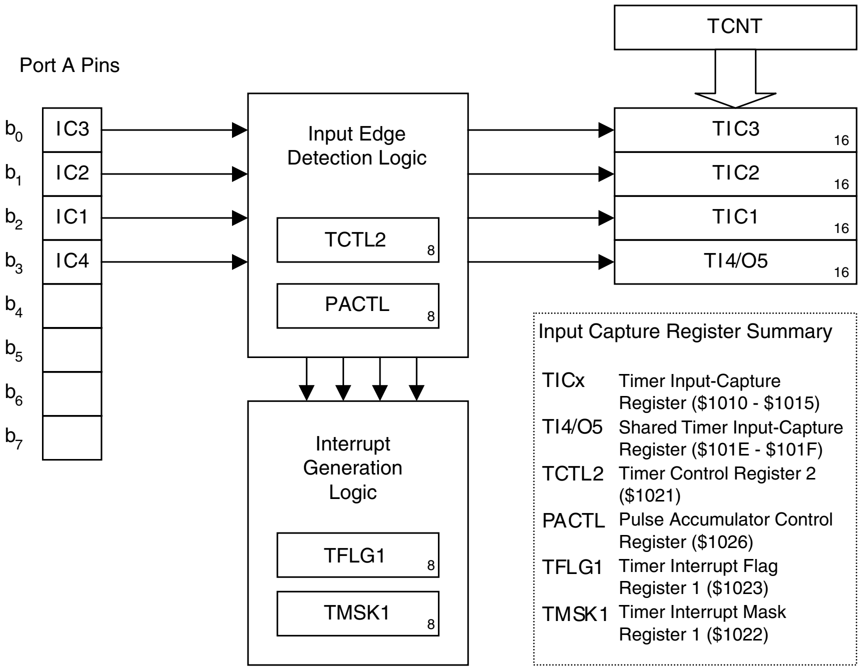
Figure 12.7 Timer System Flag and Interrupt Enable Bits

Self-Test Questions 12.2

1. How many bits are in the TCNT register?
2. Can the TCNT register be preloaded to some value by the user?
3. What is the address of the TFLG2 register?
4. How many flag bits are there in the timer system?
5. What is the purpose of the TOI bit in the TMSK2 register?

12.3 Input Capture

Input capture is designed to capture the time at which input events take place. There are four input-capture functions on the HC11E9: IC1, IC2, IC3 and IC4. Each of these functions includes an input-capture register, input edge detection logic, and interrupt generation logic, as shown in Figure 12.8. The Timer Input-Capture registers (TICx and TI4/O5) latch the current time from the free-running counter (the TCNT register). The Motorola literature refers to these registers as the "**input-capture latches**." The edge detection logic in each of the four input-capture functions is user programmable to detect rising edges only, falling edges only, or any edge (rising or falling). The interrupt generation logic includes a status flag and an interrupt enable bit for each input-capture function. The interrupt enable bit determines whether an interrupt request will be generated when an input-capture event occurs.



TICx	Timer Input-Capture Register ($1010 - $1015)
TI4/O5	Shared Timer Input-Capture Register ($101E - $101F)
TCTL2	Timer Control Register 2 ($1021)
PACTL	Pulse Accumulator Control Register ($1026)
TFLG1	Timer Interrupt Flag Register 1 ($1023)
TMSK1	Timer Interrupt Mask Register 1 ($1022)

Figure 12.8 Input-Capture Functional Block Diagram

Uniqueness of the IC4 Function

The IC4 function shares hardware with the Output-Capture 5 (OC5) function and therefore is different from the IC1, IC2 and IC3 functions. Special control bits that reside in the Pulse Accumulator Control (PACTL) register must be manipulated to assure proper operation of this function. In addition, the IC4 function does not have a dedicated TICx register (input-capture latch). It shares a 16-bit register with the OC5 function. This register is called TI4/O5. Specific differences relevant to the IC4 functions are mentioned throughout the next sections as each feature of the input-capture system is described.

Input-Capture Registers

Each input-capture function has a corresponding 16-bit Timer Input-Capture register (TIC1, TIC2, TIC3 and TI4/O5). The purpose of the registers is to capture the 16-bit value from the TCNT register when the input-capture event occurs. This value is transferred as a single 16-bit parallel word, and it indicates the relative time that the input event occurred. Each register can be read by software as a pair of 8-bit registers, as shown in Figure 12.9.

The TICx registers are not affected by reset and cannot be written by software. TI4/O5 register is reset to $FFFF and when configured for the IC4 function cannot be written by software.

Each input-capture function operates independently. Each one can capture the same 16-bit value from the TCNT register if their corresponding input events occurred simultaneously. The user should always access the data in these registers with a double byte read, that is, load D (LDD). If an input-capture event occurs during a double-byte read, the hardware write to the TICx register will be delayed until the software read is complete.

TICx

$1010 (TIC1 HI)	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8
$1011 (TIC1 LO)	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1012 (TIC2 HI)	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8
$1013 (TIC2 LO)	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1014 (TIC3 HI)	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8
$1015 (TIC3 LO)	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
RESET	U	U	U	U	U	U	U	U

TI4/O5

$101E (HI)	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8
$101F (LO)	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
RESET	1	1	1	1	1	1	1	1

Figure 12.9 TIC Register Layout

The new value will NOT be lost, but will be temporarily stored until the end of the read cycle, then be written to the corresponding register. If, however, two single-byte read instructions are used, that is, Load A (LDAA) followed by Load B (LDAB), the hardware write to the TICx register will be inhibited only for a single bus cycle. Thus, the data in the TICx register could change before the second byte is read by the user software.

Input Edge Detection Logic

The input-capture functions recognize three types of input events: rising edge only, falling edge only or any edge (rising or falling). The user can program each of the input-capture functions independently. A pair of control bits (EDGxB and EDGxA) in the Timer Control Register 2 (TCTL2) are used to select which type of edge will be detected by each of the input-capture functions, as shown in Figure 12.10. There are four two bit codes that are used to activate each of the TIC functions. The default-state of the edge bits is 00, which disables all four input-capture functions.

If the user wished to activate the IC1 function to capture on rising edges only, the binary value 01 would have to be written to b5 and b4 of the TCTL2 register. If the user wished to activate the IC3 function to capture on any edge, the binary value 11 would have to be written to b1 and b0 of the TCTL2 register. These two functions could be activated simultaneously with a single write to the TCTL2 register.

The IC4 function requires two extra control bits to assure proper operation. The extra control bits are located in the PACTL register, as shown in Figure 12.11. DDRA3 is the data direction control bit for PA3. When PA3 is being used for general purpose I/O, this bit controls whether PA3 (bit 3 of the Port A register) is an input pin or an output pin. For the IC4 function to operate properly, DDRA3 should be cleared. The I4/O5 bit controls whether the IC4 function or the OC5 function is active. I4/O5 must be set to activate the IC4 function, which in turn automatically disables the OC5 function.

NOTE: If the DDRA3 bit is set when the IC4 function is active, writes to PA3 via the Port A register will cause edges on the IC4 pin to be captured.

TCTL2	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1021	EDG4B	EDG4A	EDG1B	EDG1A	EDG2B	EDG2A	EDG3B	EDG3A
RESET	0	0	0	0	0	0	0	0

EDGxB	EDGxA	Event Configuration
0	0	Input-capture function disabled
0	1	Capture on rising edges only
1	0	Capture on falling edges only
1	1	Capture on rising or falling edge

Figure 12.10 Edge Control Bits for the TIC Functions

PACTL	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1026	DDRA7	PAEN	PAMOD	PEDGE	DDRA3	I4/O5	RTR1	RTR0
RESET	0	0	0	0	0	0	0	0

Figure 12.11 Special Control Bits for the IC4 Function

Example 12.1

Problem: What has to be written to the TCTL2 register to activate the IC2 function to capture on any edge?

Solution: A command word must be built and then written to TCTL2. EDG2B and EDG2A must be set to activate IC2 to capture on any edge.

TCTL2	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1021	EDG4B	EDG4A	EDG1B	EDG1A	EDG2B	EDG2A	EDG3B	EDG3A
CONTROL WORD	0	0	0	0	1	1	0	0

When this control word is written to the TCTL2 register, IC2 is activated to capture on any edge on the IC2 input pin.

```
LDAA    #%00001100   ;IC2 capture on any edge
STAA    $1021        ;Activate IC2
```

Interrupt Generation Logic

The input-capture interrupt generation logic contains status and control bits. The timer interrupt flag register 1 (TFLG1) contains input-capture status flags (ICxF, I4/O5F), as shown in Figure 12.12. These flags are automatically set by the hardware each time a selected edge is detected at the corresponding input-capture pin. Of course, the input-capture function must be active for the corresponding flag to be set. The status flags are cleared by writing a 1 to the corresponding bit position.

The timer interrupt mask register 1 (TMSK1) contains input-capture interrupt enable control bits (ICxI, I4/O5I). These bits provide local control of the input-capture

TFLG1	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1023	OC1F	OC2F	OC3F	OC4F	I4/O5F	IC1F	IC2F	IC3F
RESET	0	0	0	0	0	0	0	0

TMSK1	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1022	OC1I	OC2I	OC3I	OC4I	I4/O5I	IC1I	IC2I	IC3I
RESET	0	0	0	0	0	0	0	0

Figure 12.12 Input-Capture Status Flag and Local Interrupt Enable Bits

interrupts. They allow each input-capture function to be configured for polled mode or for interrupt-driven operation but do not affect the setting or clearing of the corresponding flag bits in the TFLG1 register. If the interrupt enable bit is set, the corresponding interrupt request will be made each time the status flag is set.

Because of the unique characteristics of IC4, the I4/O5F bit is set by either the IC4 function or the OC5 function, depending upon which function is enabled by the I4/O5 bit in the PACTL register. The I4/O5I bit will enable the interrupt for the IC4 or OC5 function in the same manner.

NOTE: Bits in the TMSK1 register correlate bit for bit with the flag bits in the TFLG1 register. Ones in the TMSK1 bits enable the corresponding interrupt sources.

See chapter 10 for further discussion on interrupts.

Input-Capture Application

The input-capture functions provide the ability to record the time at which various input events occur. Groups of events can be captured, allowing the measurement of the pulse width or period of a signal. The period of an input waveform is measured by capturing two successive edges of the same polarity, two rising edges or two falling edges of the same signal, as shown in Figure 12.13. The difference in time of the two events is the period measured by the counts in the TCNT register. The width of a pulse would be measured by capturing the time of two edges of opposite polarity. The width of positive-going pulses would be measured by subtracting the time of the leading (rising) edge from the time of the trailing (falling) edge. The difference in the time of these two events is the pulse width measured in counts from the TCNT register.

Example 12.2 shows a method to measure and store the period of signal present on IC1. Example 12.3 shows how to modify the program to measure the pulse width of the same waveform.

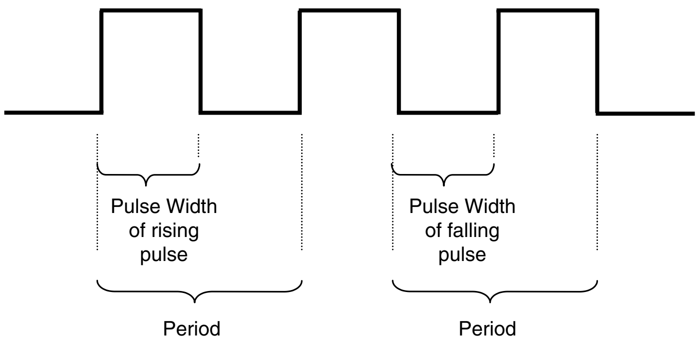


Figure 12.13 Pulse Width and Period of an Input Waveform

Example 12.2

Problem: Using IC1 to capture input edges, write a program that will measure and store the period of an input waveform.

Solution: Figure 12.14 shows a flowchart that will measure and store the period of an input waveform. The program first sets up the IC1 function to trigger off rising input edges. Then it clears the IC1 flag. Next the program waits for the IC1 flag to be set. When the flag is detected, the contents of the TIC1 register are temporarily stored, the flag is cleared and the program waits for another edge. When the second edge is detected, the period is calculated by subtracting the time of the first edge from the time of the second edge. The difference is the PERIOD of the input waveform measured in counts of the TCNT register. If the count rate is set to the default, then each count is equal to 500 ns. Source code for this program is shown in Figure 12.15.

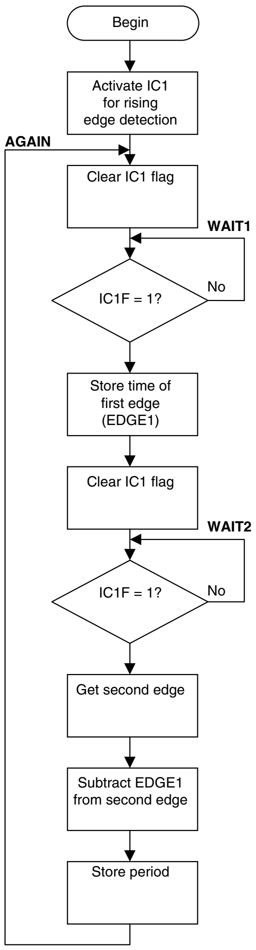

Begin
Activate IC1 for rising edge detection
AGAIN
Clear IC1 flag
WAIT1
IC1F = 1?
No
Store time of first edge (EDGE1)
Clear IC1 flag
WAIT2
IC1F = 1?
No
Get second edge
Subtract EDGE1 from second edge
Store period

Figure 12.14 Flowchart for Input Capture

```
        ORG     $0000
EDGE1   RMB     2                   ;temp storage of time of edge1
PERIOD  RMB     2                   ;storage of calculated period

        ORG     $0120               ;Start program at $0120
        LDY     #$1000              ;Load register base address in Index Y
        LDAA    #$10                ;detect rising edge of IC1 input
        STAA    TCTL2,Y
AGAIN   BCLR    TFLG1,Y $FB         ;clear IC1 flag

WAIT1   BRCLR   TFLG1,Y $04 WAIT1   ;wait for IC1 flag to be set by hardware

        LDD     TIC1,Y              ;get time of edge and store it in EDGE1
        STD     EDGE1
        BCLR    TFLG1,Y $FB         ;clear IC1 flag
WAIT2   BRCLR   TFLG1,Y $04 WAIT2   ;wait for IC1 flag to be set
        LDD     TIC1,Y              ;get time of second edge
        SUBD    EDGE1               ;PERIOD = EDGE2 - EDGE1
        STD     PERIOD              ;Store calc'ed period in PERIOD
        BRA     AGAIN               ;Do the whole thing again
```

Figure 12.15 Source Code for Input Capture Example

Example 12.3

Problem: Using IC1 to capture input edges, write a program that will measure and store the period of a negative-going or positive-going input pulse.

Solution: This program is identical to the program that measures period except for which input signal edges cause a trigger. The program first sets up the IC1 function to trigger off any input edges. The following two lines of code would be used to replace the similar lines at the beginning of the program in Example 12.2 to accomplish this change.

```
LDAA    #$30        ;detect either edge of IC1 input
STAA    TCTL2,Y
```

Self-Test Questions 12.3

1. In your own words, what is captured by the input-capture functions?
2. What are the three functional blocks of each input-capture system?
3. What is the address of each of the following input-capture registers: TMSK1, TFLG1, PACTL, TCTL2, TIC2, TI4/O5? What purpose do these registers have in regards to the input-capture functions?

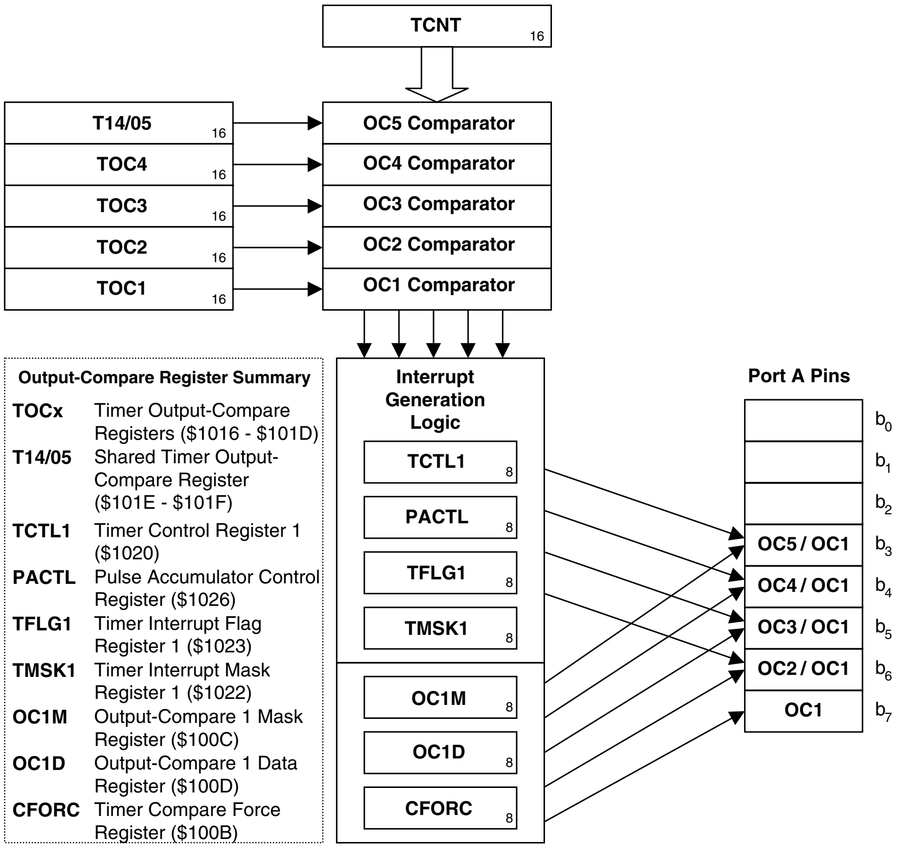
12.4 Output Compare

Output compare is designed to cause output events (pin actions) to occur on the OCx output pins at specified times. Time is defined for this function by the value in the TCNT register. There are five output-compare functions on the HC11E9: OC1, OC2, OC3, OC4 and OC5. Each of these functions is implemented with three functional hardware blocks: an output-compare register, a dedicated 16-bit comparator and interrupt generation logic, as shown in Figure 12.16. The comparators check the value in the TCNT register against the 16-bit compare registers (TOCx and TI4/O5) during each timer count. When a match is detected, the corresponding Output-Compare Flag (OCxF) is set, and the corresponding programmed output event or pin action takes place. If enabled, an interrupt will be requested.

There are two types of output compare functions on the HC11E9. The OC2 through OC5 functions constitute one of the groups, and the OC1 function has special features that will be discussed separately. All of the output-compare functions have common features; these features will be discussed first, followed by the extended features of OC1.

Uniqueness of OC5

The OC5 function shares hardware with the IC4 function, and therefore it is different from the OC2, OC3 and OC4 functions. Special control bits that reside in the PACTL



Output-Compare Register Summary

TOCx	Timer Output-Compare Registers ($1016 - $101D)
T14/05	Shared Timer Output-Compare Register ($101E - $101F)
TCTL1	Timer Control Register 1 ($1020)
PACTL	Pulse Accumulator Control Register ($1026)
TFLG1	Timer Interrupt Flag Register 1 ($1023)
TMSK1	Timer Interrupt Mask Register 1 ($1022)
OC1M	Output-Compare 1 Mask Register ($100C)
OC1D	Output-Compare 1 Data Register ($100D)
CFORC	Timer Compare Force Register ($100B)

Figure 12.16 Output Compare Functional Block Diagram

register must be manipulated to assure proper operation of this function. In addition, the OC5 function does not have a dedicated TOCx register (output-compare register). It shares a 16-bit register with the IC4 function. This register is called TI4/O5. Specific differences relevant to the OC5 function are mentioned throughout the next sections as each feature of the output compare system is described.

Output-Compare Registers

Each output-compare function has a corresponding 16-bit output-compare register: TOC1, TOC2, TOC3, TOC4 and TI4/O5. The purpose of the registers is to store the 16-bit value that will be compared to the TCNT register during each E clock cycle. This value indicates the relative time at which the output event will occur. Each register is read/write. They are accessed by software as a pair of 8-bit registers, as shown in Figure 12.17.

The TOCx registers and the TI4/O5 register are reset to $FFFF, the maximum value in TCNT.

Each output-compare function operates independently. Each one can cause output events to occur simultaneously, if the corresponding TOCx registers contain the same 16-bit value. For the active OC functions, the hardware compares the contents of the TOCx registers to the TCNT register during each timer count. Therefore, the user should always change the data in the TOCx registers with a double-byte write instruction, that is, store D (STD). During a double-byte write instruction, the compare operation is inhibited. Single-byte writes to each half of the output-compare registers are permitted; however, an 8-bit write to one byte of the output-compare registers

TOCx

$1016 (TOC1 HI)	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8
$1017 (TOC1 LO)	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1018 (TOC2 HI)	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8
$1019 (TOC2 LO)	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$101A (TOC3 HI)	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8
$101B(TOC3 LO)	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$101C (TOC4 HI)	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8
$101D(TOC4 LO)	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
RESET	1	1	1	1	1	1	1	1

TI4/O5

$101E (HI)	b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8
$101F (LO)	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
RESET	1	1	1	1	1	1	1	1

Figure 12.17 TOC Register Layouts

inhibits the output-compare function for only one bus cycle. Before the second half of the register can be written with a single-byte write, the compare operation resumes. Erroneous compares may take place during the two single-byte writes.

Read operations will work the same using single- or double-byte reads since hardware cannot change the contents of these registers.

Dedicated Comparators

The output-compare functions have dedicated comparators. If active, the comparator compares the value in the output-compare register to the value in the TCNT register. This compare occurs during every timer count. If a successful compare happens (the two registers match), the output event will take place. Four different output events can take place: no change, make the output high, make the output low or toggle the output.

The user can program each of the output-compare functions independently. A pair of control bits (OMx and OLx) in the Timer Control Register 1 (TCTL1) are used to select which type of output event or automatic pin action will take place when a successful compare takes place. Figure 12.18 shows the layout of the TCTL1 register. The default state of the bit pairs is 00, which disables the OC2, OC3, OC4 and OC5 functions. If the user wishes to activate the OC3 function to clear the OC3 pin on each successful compare, the binary value 10 would have to be written to OM3 and OL3 of the TCTL1 register. If the user wishes to toggle the OC4 pin on each successful compare, the binary value 01 would have to be written to OM4 and OL4 of the TCTL1 register. These two functions could be activated simultaneously with a single write to the TCTL1 register.

The OC5 function requires two extra control bits to assure proper operation. The extra OC5 control bits are located in the PACTL register, as shown in Figure 12.19. The I4/O5 bit controls whether the OC5 function or the IC4 function is active. I4/O5 must be cleared to activate the OC5 function, which in turn automatically disables the IC4 function. DDRA3 has no effect on the OC5 function. The state of the DDRA3 bit is overridden when the I4/O5 is cleared to activate the OC5 function.

TCTL1		b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1020		OM2	OL2	OM3	OL3	OM4	OL4	OM5	OL5
	RESET	0	0	0	0	0	0	0	0

OMx	**OLx**	**OC2 through OC5 event with successful compare (Automatic Pin Action)**
0	0	No change, OCx disabled
0	1	Toggle OCx pin
1	0	Clear OCx pin
1	1	Set OCx pin

Figure 12.18 Event Control Bits for the TOC Functions OC2–OC5

PACTL	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1026	DDRA7	PAEN	PAMOD	PEDGE	DDRA3	I4/O5	RTR1	RTR0
RESET	0	0	0	0	0	0	0	0

Figure 12.19 Special Control Bits for the OC5 Function

Example 12.4

Problem: What has to be written to the TCTL1 register to activate the OC3 function to cause the output signal to toggle on a valid compare?

Solution: A command word must be built and then written to TCTL1. OM3 and OL3 must be set to activate OC3 to toggle on each valid compare.

TCTL1	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1020	OM2	OL2	OM3	OL3	OM4	OL4	OM5	OL5
CONTROL WORD	0	0	0	1	0	0	0	0

When this control word is written to the TCTL1 register, OC3 is activated to toggle on each valid compare.

```
LDAA    #%00010000    ;OC3 toggle on valid compare
STAA    $1020         ;Activate OC3
```

Interrupt Generation Logic

The output-compare interrupt generation logic contains status and control bits. The Timer Interrupt Flag Register 1 (TFLG1) contains Output-Compare Flags (OCxF, I4/O5F), as shown in Figure 12.20. These flags are automatically set each time a successful compare occurs for one of the active output-compare functions. The status flags are cleared by writing a 1 to the corresponding bit position.

The Timer Interrupt Mask Register 1 (TMSK1) contains Output-Compare Local Interrupt enables (OCxI, I4/O5I). These bits provide local control of the output-compare interrupts. They allow each input-capture function to be configured for polled

TFLG1	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1023	OC1F	OC2F	OC3F	OC4F	I4/O5F	IC1F	IC2F	IC3F
RESET	0	0	0	0	0	0	0	0

TMSK1	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1022	OC1I	OC2I	OC3I	OC4I	I4/O5I	IC1I	IC2I	IC3I
RESET	0	0	0	0	0	0	0	0

Figure 12.20 Output-Compare Status Flag Bits

mode or for interrupt-driven operation but do not affect the setting or clearing of the corresponding flag bits in the TFLG1 register. If the interrupt enable bit is set, the corresponding interrupt request will be made each time the status flag is set.

Because of the unique characteristics of OC5, the I4/O5F is set by either OC5 or IC4, depending on which function is enabled by the I4/O5 bit in the PACTL register. The I4/O5I bit will enable the interrupt for the OC5 or IC4 function in the same manner.

NOTE: The bits in the TMSK1 register correspond bit for bit with the flag bits in the TFLG1 register. Ones in the TMSK1 bits enable the corresponding interrupt sources.

See chapter 10 for further discussion on interrupts.

OC1 Function

The OC1 function is similar to the OC2 through OC5 functions except that five output pins can be controlled concurrently. The pin actions are controlled by the values in the Output Compare 1 Mask (OC1M) and the Output Compare 1 Data (OC1D) registers, as shown in Figure 12.21. The five MSBs of each of these registers are used so that the programmed bits directly correlate to the five Port A OC1 Pins (b7–b3). The three LSBs of these registers are not used and are always read as zeros.

For a Port A pin to be affected by the OC1 function, the pin must be activated by setting the corresponding bit in the OC1M register. Only the bits that have 1's in the mask register will be activated. Any combination of 1's and 0's is allowed to activate any pattern of the five available bits. The reset state of this register is cleared, disabling the OC1 function. When a successful OC1 compare occurs, each Port A OC1 pin activated by the values in the OC1M register will assume the value of the corresponding bit in the OC1D register.

The ability to affect more than one pin with a single timing function is quite useful in applications where more than one pin is connected to a single device. In the case of a stepper motor, four of the five output pins might be connected. Since the control signals for these four pins must change simultaneously to assure proper operation of the stepper motor, the OC1 function is ideal.

OC1M	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$100C	OC1M7	OC1M6	OC1M5	OC1M4	OC1M3	–	–	–
RESET	0	0	0	0	0	0	0	0

OC1D	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$100D	OC1D7	OC1D6	OC1D5	OC1D4	OC1D3	–	–	–
RESET	0	0	0	0	0	0	0	0

Figure 12.21 OC1 Mask and Data Bits

Two separate output-compare functions can also control a single pin. OC1 can be used in conjunction with any one or all of the other output-compare functions. This allows both edges of an output signal to be programmed at the same time, greatly reducing the necessary software overhead. It also allows for an easier method of controlling which edge will cause an interrupt.

The DDRA7 and DDRA3 bits of the PACTL register have no effect on the OC1 function. The states of DDRA7 and DDRA3 are overridden when the OC1 function is active on b_7 and b_3 of Port A.

Forced State of OC1 through OC5

The output-compare subsystem contains the ability to cause forced early compares on the five output-compare pins. OCx functions are selected for early forced compare by setting the corresponding FOCx bits in the CFORC register. The FOCx bits correspond to the OCx bits of Port A, as shown in Figure 12.22. The action taken because of a forced compare is the same as if there were a successful compare between the OCx register and the TCNT register. However, forced early compares do not cause the corresponding interrupt status flag bits (OCxF) to be set. Therefore, writing to this register does not cause interrupt requests to be generated if the interrupts are enabled for the affected output-compare functions. The forced channels trigger their programmed pin actions to occur after the write to the CFORC register at the next count of the TCNT register.

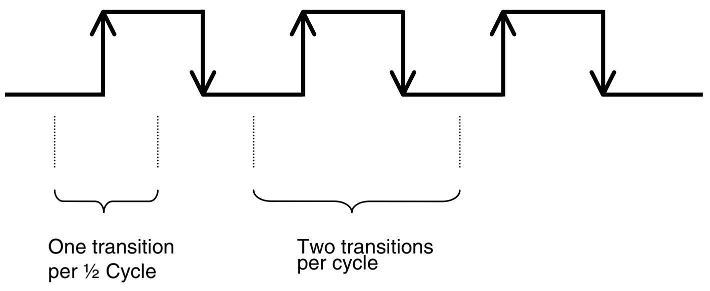
Output-Compare Application

The output-compare functions provide the ability to cause output events to occur at specified points in time, where time is measured by the TCNT register. A square wave can be generated by toggling an output pin on a regular basis. The rate at which the pin toggles determines the frequency of the waveform (see Figure 12.23). A square wave with a 50% duty cycle must toggle each half cycle. For a 1 kHz waveform, the period would be 1 ms, and the half period would be 500 µs. Since time is measured in reference to the value in the TCNT register, this half period is determined by a number of counts of the TCNT register. If the TCNT prescaler is set to E/1, the 500 µs half cycle would require 1,000 counts (500 µs/500 ns per count).

A short program is shown in Example 12.5 that generates a 1 kHz square wave on OC2.

CFORC	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$100B	FOC1	FOC2	FOC3	FOC4	FOC5	–	–	–
RESET	0	0	0	0	0	0	0	0

Figure 12.22 Early Forced Compare Control Bits



If f=1 kHz, then the period is equal to 1/f=(1/1 kHz) = 1 ms

Since there are two transitions per cycle, then the transitions must happen every 500 μs to produce a 50% duty cycle 1 kHz waveform.

Figure 12.23 Output-Compare Square Wave Timing

Example 12.5

Problem: Using OC2 to output a signal, write a program that will create a 1 kHz square wave.

Solution: Figure 12.24 shows a flowchart that causes OC2 to toggle each half cycle. The half-cycle delay is set so that the resulting waveform has a 1 kHz frequency. The program first activates the OC2 function to toggle on each event. Next the OC2 flag is cleared; then it waits for the event to take place. When the event occurs, the program calculates the next event time, stores the time in the TOC2 register and loops back to do the entire process over again. The source code for this program is shown in Figure 12.25.

Self-Test Questions 12.4

1. How many output-compare functions are available on the 68HC11E9?
2. In your own words, what is the purpose of the output-compare functions?
3. What is stored in each of the TOCx registers?
4. How many bits are contained in each of the four TOCx registers?
5. What is the reset state of the TOCx and TI4/O5 registers?

12.5 Pulse Accumulator

The **pulse accumulator** allows the user to count events. It is much simpler than the input-capture, output-compare and real-time interrupt systems. This system is based on an 8-bit counter and can be configured to operate as a simple event counter or for time accumulation. The pulse accumulator functional block diagram is shown in Figure 12.26.

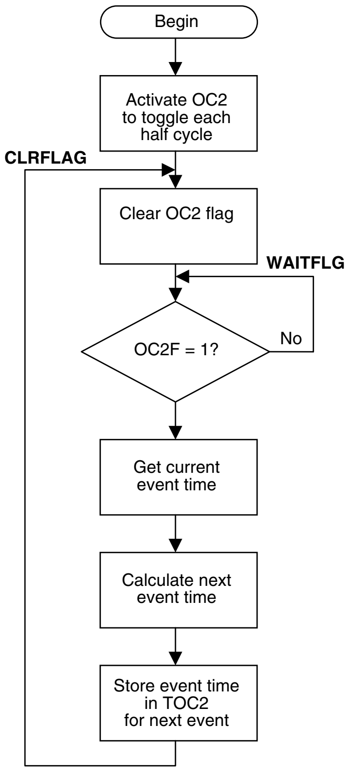

Begin
Activate OC2 to toggle each half cycle
CLRFLAG
Clear OC2 flag
WAITFLG
OC2F = 1?
No
Get current event time
Calculate next event time
Store event time in TOC2 for next event

Figure 12.24 Flowchart for Square Wave Creation

```
************************************************************
* This program generates a 1KHz squarewave on OC2 (pin 28)
************************************************************
TOC2     EQU     $18
TCTL1    EQU     $20
TMSK1    EQU     $22
TFLG1    EQU     $23

ORG      $0100                         ;Start program at $0100

         LDX     #$1000                ;Load register base address into Index X

         LDAA    #$40                  ;toggle OC2 on successful compare
         STAA    TCTL1,X

CLRFLAG BCLR     TFLG1,X $BF           ;clear OC2 flag

WAITFLG BRCLR    TFLG1,X $40 WAITFLG   ;wait for OC2 flag to be set

         LDD     TOC2,X                ;get compare value from TOC2
         ADDD    #$3E8                 ;add offset for next pulse transition
         STD     TOC2,X                ;1000 (3E8) counts high / low for 1KHz
         BRA     CLRFLAG               ;Loop continuously
```

Figure 12.25 Source Code for Square Wave Generation

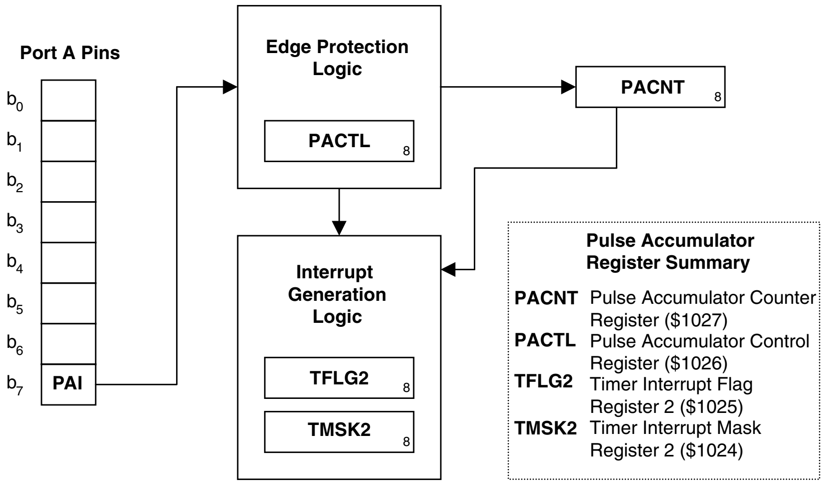

Port A Pins
b_0
b_1
b_2
b_3
b_4
b_5
b_6
b_7
PAI
Edge Protection Logic
PACTL 8
PACNT 8
Interrupt Generation Logic
TFLG2 8
TMSK2 8
Pulse Accumulator Register Summary
PACNT Pulse Accumulator Counter Register ($1027)
PACTL Pulse Accumulator Control Register ($1026)
TFLG2 Timer Interrupt Flag Register 2 ($1025)
TMSK2 Timer Interrupt Mask Register 2 ($1024)

Figure 12.26 Pulse Accumulator Functional Block Diagram

When Port A b7 is used for pulse accumulator input, it is called the Pulse Accumulator Input pin (PAI). When this bit is being used for input to the pulse accumulator function, the optional digital I/O function (PA7) is disabled. The OC1 function on this pin is also normally disabled when using the pulse accumulator; however, in some special applications both the OC1 function and the pulse accumulator function can be active. In this case, the output event caused by the OC1 function triggers the pulse accumulator input, thus allowing output events to be counted.

Pulse Accumulator Counter Register

The Pulse Accumulator Counter Register (PACNT) contains the count of external input events if the pulse accumulator is configured to count events. When the pulse accumulator is configured for gated mode, this register contains the frequency of pulses during the gated period. The PACNT has a maximum count of 255 since it is an 8-bit register, as shown in Figure 12.27. When the count reaches this maximum count, a subsequent count causes the register to roll over to 0. The register is read/write and can be accessed even when the pulse accumulator is not enabled. It is not affected by reset.

Pulse Accumulator Enable and Mode Select

Four bits control the overall function of the pulse accumulator system, as shown in Figure 12.28. The Pulse Accumulator Enable (PAEN) bit is set to enable the pulse

PACNT

$1027	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
RESET	U	U	U	U	U	U	U	U

Figure 12.27 PACNT Register Layout

PACTL		b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
\$1026		DDRA7	PAEN	PAMOD	PEDGE	DDRA3	I4/O5	RTR1	RTR0
	RESET	0	0	0	0	0	0	0	0

PAMOD	PEDGE	MODE	Action
0	0	Event	PA counter incremented by falling edge on PAI
0	1	Event	PA counter incremented by rising edge on PAI
1	0	Gated	0 on PAI inhibits counting of E/64
1	1	Gated	1 on PAI inhibits counting on E/64

Figure 12.28 Pulse Accumulator Control Bits

accumulator system. When PAEN is cleared, the pulse accumulator system is disabled. The Data Direction Port A Bit 7 (DDRA7) control bit is normally cleared when using the pulse accumulator system so that the PA7/PAI bit is programmed for input operation. However, the bit can also be used for general-purpose I/O concurrent with the pulse accumulator function being active. If PA7 is programmed for general-purpose output by setting DDRA7, then output edges will cause the input on PAI concurrently. The pulse accumulator can be operated in one of two modes, event mode and gated mode. These modes are activated by the Pulse Accumulator Mode (PAMOD) control bit in the PACTL register.

Event Counting Mode

Event mode allows the user of the pulse accumulator to count each event that is recognized at PAI. Many microcontroller applications require events to be counted. For example, a sensor could be used to sense a box moving down a conveyor system. Each time the sensor activates, it sends a signal to the pulse accumulator via the PAI bit and the event (a box in this case) is counted. To select the event counting mode, the PAMOD bit of the PACTL register must be cleared. When configured this way, the signal on the PAI pin of Port A acts like a clock input to the counter. The Pulse Accumulator Edge (PEDGE) control bit of the PACTL register allows the user to select to which edge, rising or falling, the pulse accumulator will respond. If PEDGE is cleared, falling edges on PAI will clock the accumulator. If PEDGE is set, rising edges will clock the accumulator.

Gated Time Accumulation Mode

The **gated mode** changes the pulse accumulator into a timer. This timer can be used to measure the duration of input signals by counting the number of clocks that occur during the input pulse. During an active input, the accumulator counts every 64th E clock. Thus, the time duration is accurate within 64 E clocks (32 μs for a 2 MHz clock). Gated mode is selected by setting the PAMOD bit. The active state is selected with the PEDGE bit. This bit is essentially an inhibitor. When PEDGE is cleared, the accumulator will stop counting when PAI is zero. When PEDGE is set, the accumulator will stop counting when PAI is one.

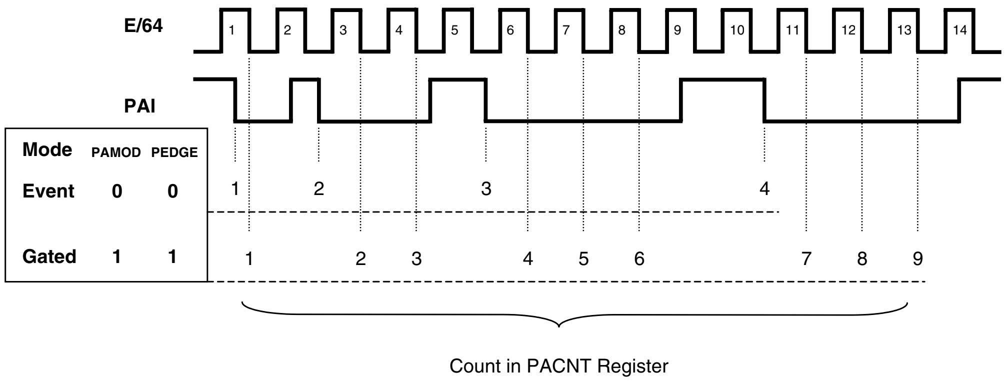


Figure 12.29 Pulse Accumulator Mode Examples

Figure 12.29 illustrates how the pulse accumulator will count input events in event mode versus how the pulse accumulator will count each E/64 during the active input on PAI while using gated mode.

Interrupt Generation Logic

The pulse accumulator interrupt generation logic contains status and control bits. TFLG2 contains a pulse accumulator input status flag (PAIF) and a pulse accumulator overflow flag (PAOVF), as shown in Figure 12.30. In event mode, PAIF is automatically set each time an input occurs on the PAI pin. For gated time accumulation mode, PAIF is set when the count is inhibited (counting is stopped). PAOVF is automatically set when the PACNT register overflows from $FF to $00. The pulse accumulator status flags are cleared by writing a 1 to the corresponding bit position.

TMSK2 contains two pulse accumulator interrupt enable control bits (PAII and PAOVI). These bits provide local control of interrupts that are related to the pulse accumulator function. They allow the PAIF and the PAOVF to be configured for polled mode or for interrupt-driven operation. The interrupt enable control bits have no

TFLG2	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1025	TOF	RTIF	PAOVF	PAIF	–	–	–	–
RESET	0	0	0	0	0	0	0	0

TMSK2	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1024	TOI	RTII	PAOVI	PAII	–	–	PR1	PR0
RESET	0	0	0	0	0	0	0	0

Figure 12.30 Pulse Accumulator Status Flag Bits

effect of the flag bits in the TFLG2 register. If the interrupt enable bit is set, the corresponding interrupt request will be made each time the status flag is set.

NOTE: The bits in the TMSK2 register correspond bit for bit with the flag bits in the TFLG2 register. Ones in the TMSK2 bits enable the corresponding interrupt sources.

See chapter 10 for further discussion on interrupts.

Self-Test Questions 12.5

1. Which bit in the PACTL register must be set to enable Pulse Accumulation?
2. What is the name of the pulse accumulator mode that counts the duration of a high or low state on PAI?
3. What is the name of the pulse accumulator mode that counts all pulses that come in on PAI?
4. How many bits are in the PACNT register?
5. What is the maximum count of the PACNT register?

12.6 Real-Time Interrupts

The real-time interrupt (RTI) function on the 68HC11 can be used to generate events at a fixed periodic rate. Unlike the other Port A features, the RTI does not have to be activated. The periodic events take place on a continuous basis by default. The user can choose to ignore these events or to do something when they happen, but cannot stop them from happening or interrupt them in any way. This free-running aspect causes the periodic rate to be constant and free of software latencies caused by flag clearing and interrupt service.

Functionally the RTI feature of the 68HC11 is a timing metronome similar to those used by musicians to set the rate at which they play a piece of music. The events take place at a fixed periodic rate and can be the trigger for any other event that the user can control via software functions. Similar to the other functions contained in the Port A timing system, the RTI function has a flag bit (RTIF) and an interrupt enable bit (RTII). These bits are located in the TFLG2 and TMSK2 registers, as shown in Figure 12.31. Each time the periodic event occurs, the RTIF bit is set. If the RTII bit is set when the periodic event takes place, an interrupt will be requested. The user cannot disable the periodic events, but can ignore that they have happened. There is no external hardware connection tied to the RTI function; it must be software controlled.

NOTE: Bits in the TMSK2 register coincide bit for bit with the flag bits in the TFLG2 register. Ones in the TMSK2 bits enable the corresponding interrupt sources.

See chapter 10 for further discussion on interrupts.

TFLG2	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1025	TOF	RTIF	PAOVF	PAIF	–	–	–	–
RESET	0	0	0	0	0	0	0	0

TMSK2	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1024	TOI	RTII	PAOVI	PAII	–	–	PR1	PR0
RESET	0	0	0	0	0	0	0	0

Figure 12.31 Real-Time Interrupt Status Flag Bit

PACTL	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1026	DDRAT	PAEN	PAMOD	PEDGE	DDRA3	I4/O5	RTR1	RTR0
RESET	0	0	0	0	0	0	0	0

RTR1	RTR0	Clock Divider E/x	RTI Rate
0	0	$E/2^{13}$	4.096 ms
0	1	$E/2^{14}$	8.192 ms
1	0	$E/2^{15}$	16.384 ms
1	1	$E/2^{16}$	32.768 ms

Figure 12.32 Real-Time Rate Control Bits

NOTE: The most common problem that users encounter with the RTI system is caused by not clearing the RTIF bit after it is recognized. If the flag is not cleared, the hardware thinks another event has taken place. Since this bit is checked when the process returns from an interrupt request, the system will immediately cause another interrupt request. The program will be lost in an endless loop.

The events happen at a fixed periodic rate, yet this rate can be changed by the user. The PACTL register contains two bits, called Real Time Interrupt Rate bits (RTR1 and RTR0). They control the real-time interrupt rate, as shown in Figure 12.32. The rates shown in the table are for a 2 MHz E clock. Typically these bits are set once at the start of a program, but can be changed at anytime.

Self-Test Questions 12.6

1. Which bit of the TMSK2 register must be set to cause an interrupt to be requested when the RTIF bit is set?
2. Can the user turn off the periodic events caused by the RTI hardware?
3. Which bit of Port A is connected to the RTI hardware?
4. How is the RTIF bit cleared?

Summary

Port A is a general purpose I/O that can also be configured for a variety of timer functions. This chapter presented the theory of timed events as well as explanation and examples of the five major functions performed within the HC11 timing system.

The input-capture and output-compare features are dependent on the master clock timer, the TCNT register. This 16-bit free-running register is dedicated to counting E clocks. The user has the ability to slow the count to every 4th, 8th or 16th E clock via two configuration bits, PR1 and PR0, in the TMSK2 register. The value in the TCNT register always represents the number of counts since the last timer overflow (TOF). This value is the time that is captured by the input-capture functions when an input event occurs. It is also the reference time used by the output-compare functions to determine when to cause output events.

This timing system also has the ability to count events using the pulse accumulator. The 8-bit pulse accumulator register can count up to 255 events before it rolls over. Pulse accumulator overflows are tracked by the pulse accumulator overflow flag (PEOVF), which has a corresponding interrupt. This register can also be preset to any value between 0 and 255, which makes it convenient to count only a few events before rolling over.

Finally, the Port A timing system has a built-in periodic event generator, the real-time interrupt feature. Any task that needs to be performed on a routine periodic basis can be linked to this timing feature.

Chapter Questions

Section 12.1

1. Can a rising edge be considered an input event?
2. Can the time that an input event will occur be typically predicted?
3. If the HC11 were connected to an external circuit, what type of event would need to take place to have the HC11 send a signal to the external circuit?

Section 12.2

4. Can the TCNT register be read at anytime by the user?
5. Which kind of register is checked to see if the timer system overflow events have occurred?
6. What value would be loaded into PR0 and PR1 of the TMSK2 register to set the TCNT clock to prescale by 16?
7. Show two methods of clearing TOF without affecting any other flags.
8. Why should BSET instructions not be used to clear timer system flag bits?
9. How many interrupt enable bits are there in the timer system?
10. What registers contain interrupt enable bits?
11. What is the relationship of the flag bits to the interrupt enable bits?

12. Why is it important for the user to clear the flag bits while servicing an interrupt?

Section 12.3

13. How many input-capture functions are available on the HC11E9?
14. What value is in the TCTL2 register after reset?
15. How many bits are contained in each of the TICx registers?
16. What value is loaded into the TICx and TI4/O5 registers at reset?
17. What binary value must be loaded into the EDGxB and EDGxA of the TCTL2 register to capture falling edges only?
18. What control word would be written to the TCTL2 register to activate the IC2 function to capture on falling edges only and to activate the IC3 function for capture on rising edges only?
19. Why are the DDRA3 and I4/O5 bits of the PACTL register significant to an input-capture function?
20. Explain what must be done to activate the IC4 function to capture on any edge.
21. Can pulse width be measured using data collected with an input-capture function?
22. Can frequency (rate of change) be calculated using data collected with an input-capture function? Why?

Section 12.4

23. How many output-compare functions are available on the HC11E9?
24. Which Port A pin(s) is/are used for the OC1 function?
25. What are the dedicated comparators comparing? Why?
26. What is the address of the TOC3 register?
27. Why should double-byte read/write instructions be used when writing data to the TOCx registers?
28. What are the three functional hardware blocks of each output-compare function?
29. What binary value must be loaded into the OMx and OLx of the TCTL1 register to cause the output-compare pin to toggle on a successful compare?
30. What hex value is written to the TCTL1 register to activate the OC3 function (disabling OC2, OC4 and OC5) so that it will cause the output to go high when there is a successful compare?
31. What value needs to be written to the I4/O5 bit of the PACTL register to select the OC5 function and disable the IC4 function?
32. What causes the OCxF bits in the TFLG1 register to be set?
33. How does the user clear the OCxF bits of the TFLG1 register?
34. If the OC3I bit is set, what happens when the OC3F bit is set by the hardware?
35. What happens on a successful OC1 compare when the OC1M register is loaded with \$F0 and the OC1D register is loaded with \$A0?
36. How can a user set the default state of the OCx output pins to be high?
37. Can the output-compare feature be used to create an event timer (i.e., sequence of events to happen at specific times)?

Section 12.5

38. Which Port A pin(s) is/are used for the pulse accumulator input?

39. What is the value loaded into the PAMOD and PEDGE bits of the PACTL register so that a rising edge on PAI increments the count?
40. What is the value loaded into the PAMOD and PEDGE bits of the PACTL register so that a high state on PAI inhibits the count?
41. Fully explain what happens when the PACNT register contains $FF, the PACTL register contains $40 and a rising edge comes in on PAI?

Section 12.6

42. What value must be written to RTR1 and RTR0 of the PACTL register to set the periodic rate to 16.384 ms?
43. At what rate do the real-time interrupts occur after reset?

Chapter Problems

1. What value should be written to the PR1 and PR0 bits of the TMSK2 register if a timing-dependent system has events that need to occur on a 200 µs interval and the entire sequence is completed on a 100 ms interval?
2. If the PR1 and PR0 bits of the TMSK2 register are set to 01 and the TCNT register contains $01D0, how long has it been (in seconds) since the TCNT register overflowed?
3. Write a program that will measure the pulse width of a signal connected to IC1. Display the pulse as a hex number of E clocks on the monitor. Originate the program at $0100.
4. Write a program that will generate a 5 kHz square wave on OC3. Originate the program at $0120.
5. Write a program that will count rising edges of pulses on PAI and display on the monitor the current count. Originate the program at $0100.
6. Write a program that will generate a 1 Hz square wave on PB0 using the RTI function. Originate the program at $0100.

Answers to Self-Test Questions

Section 12.1

1. An event is an electrical signal that is caused by the hardware or sensed by the hardware.
2. An input event is an electrical signal that comes from some external source. It is sensed by the hardware.
3. An output event is an electrical signal that is caused by some internal function. The hardware causes the event on an output pin.

Section 12.2

1. The TCNT register is a 16-bit free-running counter.
2. No, the register is cleared at reset and then counts continuously without interruption.
3. The address of the TFLG2 register is $1025.
4. There is a total of 12 timer system flag bits contained in the TFLG1 and TFLG2 registers.
5. This is a local interrupt enable control bit for the Timer Overflow Interrupt.

Section 12.3

1. The time at which the event occurred where the time is the relative time contained in the TCNT register.
2. Input-capture register, input edge detection logic, and interrupt generation logic.
3.

TMSK1	$1022	Local interrupt control bits
TFLG1	$1023	Input-capture status flags
PACTL	$1026	Control for IC4
TCTL2	$1021	Control for the type of event detection
TIC2	$1012–$1013	16-bit register where captured time is stored for IC2
TI4/O5	$101E–$101F	16-bit register where captured time is stored for IC4

Section 12.4

1. There are five independent OC functions available on the HC11E9, OC1–OC5.
2. The OC functions cause an output event at a specified time. The time of the event is relative to the value in the TCNT register.
3. The TOCx registers contain the time, relative to the time in the TCNT register, that designates the time that the output event will take place.
4. The TOCx contains 16 bits.
5. The bits of all OC registers are set high at reset.

Section 12.5

1. The PAEN bit must be set to activate the pulse accumulator system.
2. The gated mode measures the duration of a pulse.
3. The event mode counts pulses (events) on the PAI pin.
4. The PACNT register is an 8-bit register.
5. Since the PACNT register is an 8-bit register, the maximum count is $FF or 255.

Section 12.6

1. The RTII bit must be set to request interrupts when the RTIF is set.
2. No, they are constantly running, but can be ignored.
3. RTI has no external hardware connections.
4. RTIF is cleared by writing a 1 to it.

chapter 13

Serial Communication—Port D

Objectives

After completing this chapter, you should be able to:

- Explain the difference between asynchronous and synchronous serial communications
- Configure the HC11 SCI for transmit and receive
- Change the SCI BAUD rate
- Understand the function of the SPI system

Outline

Introduction

This chapter will discuss two types of serial communications interfaces: asynchronous and synchronous. Both of these interfaces are part of the PORTD subsystem in the 68HC11. The asynchronous interface is implemented with the serial communications interface (SCI) hardware, and the synchronous interface is implemented with the serial peripheral interface (SPI) hardware. The SCI subsystem is a full-duplex UART-type asynchronous system using standard nonreturn-to-zero (NRZ) format. The SPI subsystem is typically used to provide high-speed communications between the MCU and system peripheral devices that are usually located on the same circuit board.

This chapter directly correlates to sections 8 and 9 of the HC11 Reference Manual and sections 7 and 8 of the Technical Data Manual.

13.1 Theory of Serial Communications

In chapter 1, it was shown that data is stored on a computer in multibit words. These words are usually 8 bits wide and are called bytes. This data moves around inside the computer from registers to memory and the various processor components via the data bus. The data bus is a **parallel** bus in that it contains multiple wires, one wire for each data bit in the data word. This parallel structure allows all bits of the data word to be transmitted simultaneously, but requires dedicated wires for each bit in the data word.

Most communication media consist of single data conductors (wires). A **serial** interface is one in which the data is transmitted one bit at a time on a single conductor. Network systems like the Internet and local area networks use serial communication technology; the data is transmitted one bit at a time. In addition, the standard modems that are included in most computers interface to the telephone system using serial technology to allow the computer to transmit and receive data using the single pair of telephone wires.

Sending Data

Because data is stored in the computer in a parallel fashion, additional hardware is necessary to interface the parallel system to the serial interface. When data is being transmitted from the parallel environment on the computer, it must first be converted into a serial format. This is accomplished with a parallel-to-serial converter, which can be implemented with a single parallel-in, serial-out shift register, as shown in Figure 13.1.

After the data is converted to a serial stream, additional control bits are added to the data before it is transmitted. The control bits usually appear at the beginning and the end of the data stream. Often these control bits are called the data **header** and **footer**. They are used to package the actual data and help with the transmission.

Receiving Data

In the opposite fashion to the transmit hardware, the receive hardware has the job of converting the received serial bit stream back into the parallel data words. This is accomplished with a serial-to-parallel converter, which can be implemented with a single serial-in, parallel-out shift register, as shown in Figure 13.2.

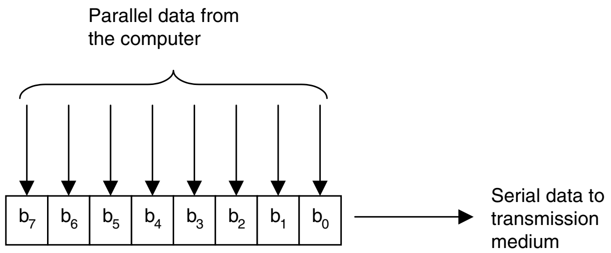

Parallel data from the computer
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
Serial data to transmission medium

Figure 13.1 Parallel to Serial Conversion

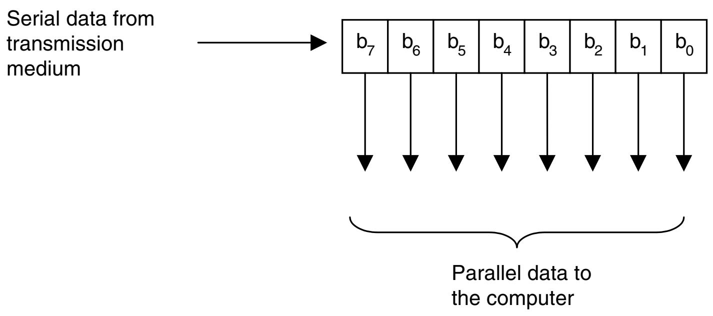

Serial data from transmission medium
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
Parallel data to the computer

Figure 13.2 Serial to Parallel Conversion

When the data is received from the serial medium, the control bits are no longer needed and are stripped from the stream. This leaves only the original data, which is converted back to the parallel words.

Asynchronous Communications

RS-232 is an interface standard for asynchronous serial communications. It outlined the connection of the computer to serial communications equipment on a telephone system. Since it was established in 1960, it has become one of the common standards followed for connecting two devices via a serial interface. The RS-232 specification designates mechanical, electrical, functional and procedural specifications. This text will focus on only a small portion of the procedural specifications in order to highlight the principles of serial data transfer.

Asynchronous communications is a method that encapsulates the data within control bits, so that a timing signal or clock is not necessary for the transmission to take place. The following rules must be observed to assure that the data is transmitted and received properly.

1. The data must be transferred one byte at a time (often it is a seven-bit code for an alpha/numeric character).
2. Data is transmitted LSB first.
3. Data is joined with control bits to form a character for transmission. The format of the character is 1 start bit, 7 or 8 data bits (LSB first), optional parity bit and 1 to 2 stop bits, as shown in Figure 13.3.
4. Start bits must be low and stop bits must be high because the idle line is high. The falling edge of the start bit indicates to the receiver that data is being sent. The stop bits return the line to the idle state.

The **start and stop bits** identify the beginning and end of a character and permit the receiver to synchronize to the transmit clock rate for each character transmitted. They are the header and footer that are added to the data. The **parity** bit is used for error detection. This bit is designated as odd or even parity and is added to the data to assure that there is an odd or even number of 1's in the data stream.

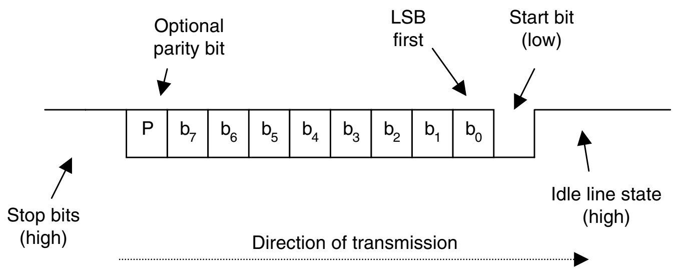


Figure 13.3 Asynchronous Serial Character Format

Synchronous Communications

Synchronous communications is a method of transmitting serial data by using a synchronizing clock. The clock is transmitted with the data so that the receiving device knows exactly when to clock in the received data. Consequently, a synchronous system can operate more efficiently than an asynchronous system.

In a synchronous system, a control signal is sent indicating that data transmission will start. When the transmission starts, the data is converted from parallel to serial and transmitted serially. The receiving device clocks the data in using the accompanying clock. No header or footer bits are required to synchronize the transmission because the transmission is controlled by separate control signals and the clock.

Self-Test Questions 13.1

1. How is data converted from the parallel format to the serial format?
2. What is the format of an asynchronous data character?
3. What is the state of the idle line on an asynchronous transmission medium?
4. What is the difference between synchronous and asynchronous communications?

13.2 Serial Communications Interface (SCI)

The Serial Communications Interface (SCI) is an asynchronous serial communications system that supports simultaneous transmit and receive. It supports one start bit, eight or nine data bits and one stop bit. The SCI transmitter and receiver are functionally independent, although they use the same data format and baud rate. The SCI uses separate pins for transmit and receive. The Port D b_1 is used for transmit (TxD) and Port D b_0 is used for receive (RxD), as shown in Figure 13.4.

NOTE: During transmit and receive operations the values stored in DDRD1 and DDRD0 bits of the DDRD register are overridden. Thus, the state of the DDRD1 and DDRD0 bits has no effect on the operation of the SCI.

Port D	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
DIGITAL I/O	–	–	In/Out	In/Out	In/Out	In/Out	In/Out	In/Out
SCI FUNCTION	–	–	-	-	-	-	TxD	RxD

Figure 13.4 Port D Pin Designations for SCI

SCI Timing Control

All timing for the SCI system is derived from the HC11 bus clock. The bus clock operates at the same frequency as the E clock, but at a different phase. Figure 13.5 illustrates the functional timing blocks of the SCI system and their relationship to main HC11 timing system.

The main bus clock feeds the SCI Baud-Rate Prescaler. The output of the prescaler is 16 times the highest baud rate available in the system. The output of the prescaler is sent to the baud-rate selector that determines the actual receive clock used by the SCI receiver. The receiver clock is 16 times the transmit baud rate. The transmit section further divides the receiver clock rate by a factor of 16 to generate the transmit baud rate. All values shown presume that the master E clock rate is 2 MHz.

The prescaler and selector are controlled by values stored in the Baud-Rate Control register (BAUD). The location of these bits in the BAUD register is shown in Figure 13.6a. The prescaler is controlled by two bits called the SCI Baud Rate Prescaler Selects (SCP1 and SCP0). Figure 13.6b illustrates the highest rate that will be selected for each of four possible SCPx bit combinations. Since the factors 3 and 13 are not evenly divisible into the 2 MHz master clock, the resulting highest baud rate is not an integer

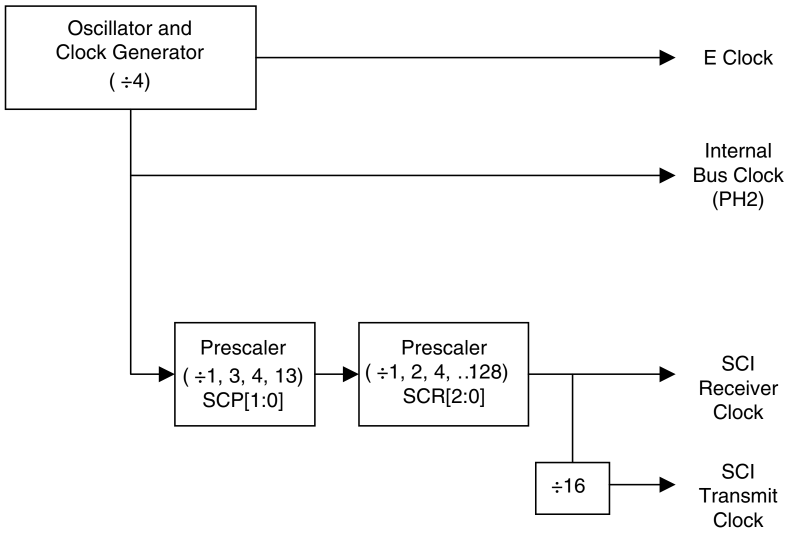

Oscillator and Clock Generator (÷4)
E Clock
Internal Bus Clock (PH2)
Prescaler (÷1, 3, 4, 13) SCP[1:0]
Prescaler (÷1, 2, 4, ..128) SCR[2:0]
SCI Receiver Clock
÷16
SCI Transmit Clock

Figure 13.5 SCI Timing Block Diagram (*adapted with permission from Motorola)*

number of cycles. The highest baud rate is approximately 41 kHz and 9600 Hz respectively.

The output of the prescaler is sent on to the baud-rate selector. Three bits are used to select the final baud rate which are called the SCI Baud Rate Selects (SCR2, SCR1 and SCR0). Figure 13.6c illustrates what baud rates will be generated from the output of the selector as a function of the eight possible values present on the SCRx bits. This table is based on the highest baud rate, equal to 9600.

The rate of the receive clock is 16 times the transmit rate to allow for over-sampling of the receive data. The receive data is checked 16 times while each bit is being received. This oversampling of the received data compensates for minor differences in the transmit and receive baud rates.

There are three additional bits in the BAUD register. One bit, b_6, is unused; and the other two, TCLR and RCKB, are used only in the processor special-test mode.

BAUD		b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$102B		TCLR	–	SCP1	SCP0	RCKB	SCR2	SCR1	SCR0
	RESET	0	0	0	0	0	U	U	U

a) Baud-Rate Register Layout

SCP1	SCP0	Factor (E/x)	Prescaler Output Rate (16X Tx Rate)	Highest Baud Rate (Tx Rate)
0	0	E/1	2 MHz	125 kHz
0	1	E/3	≈667 kHz	≈41 kHz
1	0	E/4	500 kHz	≈31 kHz
1	1	E/13	≈154 kHz	≈9600 Hz

b) Baud-Rate Prescale Selects

SCR2	SCR1	SCR0	Factor (Highest / x)	Selector Output Rate (Rx clock)	Baud Rate (Tx clock = Rx / 16)
0	0	0	1	154 kHz	9600
0	0	1	2	77 kHz	4800
0	1	0	4	38.5 kHz	2400
0	1	1	8	19.2 kHz	1200
1	0	0	16	9.6 kHz	600
1	0	1	32	4.8 kHz	300
1	1	0	64	2.4 kHz	150
1	1	1	128	1.2 kHz	75

c) Baud Rate Selects for Highest Baud Rate = 9600

Figure 13.6 SCI Baud-Rate Control

Example 13.1

Problem: Build a command word that will configure the SCI for 4800 BAUD.

Solution: The SCP1 and SCP0 bits should be set to %11 to select the maximum BAUD rate to be 9600; then the SCR2–SCR0 bits should be set to %001 to select 4800 BAUD. The whole command word would be %00110001 or $31 that would have to be written to the BAUD register.

SCI Transmitter

The SCI transmitter functions in the same manner as the transmit features of a UART (Universal Asynchronous Receiver/Transmitter). Figure 13.7 illustrates the transmit function. The transmit function is enabled via the TE bit in the SCCR2 register. The transmitter automatically sends a preamble of all 1's when it is enabled (idle state of the line). After it is enabled, a byte of data, called a character, is written to the SCI Data register (SCDR). The SCDR then moves this data to the internal shift register. The shift register adds the start and stop bits to the data and shifts the character serially to the transmit pin, TxD. Each time a character is moved from the SCDR to the shift register for transmission, the TDRE flag is set. This indicates to the user that the hardware is ready for another character. The TC flag is set when the entire transmission is complete.

The SCDR register is used for the transmit, as well as the receive operations. The layout of the SCDR register is shown in Figure 13.8a. The contents of this register are undefined after reset.

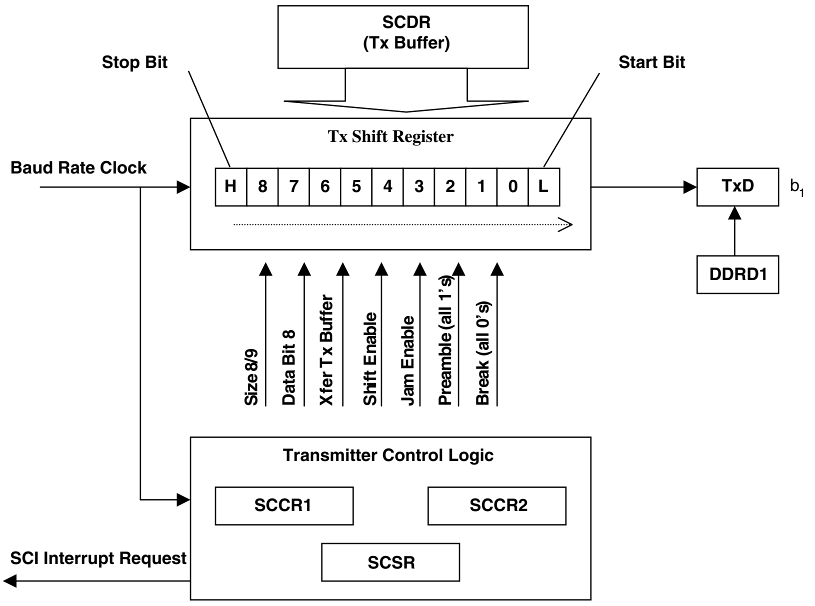


Figure 13.7 SCI Transmitter Functional Block Diagram

The process of transmitting characters is controlled by the bits in two SCI Control Registers (SCCR1 and SCCR2). The SCCR1 register contains two bits that directly control the SCI transmit function, as shown in Figure 13.8b. The SCI allows for data transfers of 8 or 9 bits. The M bit is the mode bit that selects this character format. When M = 0, the character format is set to 8-bit data. When M = 1, the character format is set to 9-bit data. When the M bit is set, the T8 bit holds the 9th bit of the transmit data character.

The SCCR2 register contains four bits that are relevant to the transmit function. The Transmit Interrupt Enable (TIE) bit is used to enable the SCI interrupt when the transmit data register is empty. When TIE = 1, an SCI interrupt is requested if the Transmit Data Register Empty Flag (TDRE) in the SCSR register is set. When TIE = 0, TDRE interrupts are disabled. The Transmit Complete Interrupt Enable (TCIE) bit is used to enable the SCI interrupt when the transmission is complete. When TCIE = 1, an SCI interrupt is requested if the TC bit in the SCI Status register (SCSR) is set. When TCIE = 0, TC interrupts are disabled.

The Transmit Enable (TE) bit of the SCCR2 register is used to enable the SCI transmit function. When TE changes from zero to one, a preamble character is transmitted. The preamble character consists of all logic ones, including the start

SCDR	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$102F	R7/T7	R6/T6	R5/T5	R4/T4	R3/T3	R2/T2	R1/T1	R0/T0
RESET	U	U	U	U	U	U	U	U

a) SCI Data Register (SCDR) Layout

SCCR1	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$102C	R8	T8	–	M	WAKE	–	–	–
RESET	U	U	0	0	0	0	0	0

b) SCI Control 1 Register (SCCR1) Layout

SCCR2	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$102D	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
RESET	0	0	0	0	0	0	0	0

c) SCI Control 2 Register (SCCR2) Layout

SCSR	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$102E	TDRE	TC	RDRF	IDLE	OR	NF	FE	–
RESET	1	1	0	0	0	0	0	0

d) SCI Register (SCSR) Layout

Figure 13.8 SCI Data, Control and Status Register Layouts (grayed bits are not relevant to the transmit operations)

and stop bits. Logic ones create the idle line condition necessary for receive hardware to recognize the first character of a transmission. Transmission continues with subsequent writes to the SCDR. The last bit in the SCCR2 register that is relevant to the transmit operation is called Send Break (SBK). It is used to send the break character, which consists of all logic zeros, including the start and stop bits. When SBK = 1, the break character is queued for transmission. Setting the SBK will not Affect the transmission of the current character, but will queue the break character to be transmitted next. As long as the SBK bit remains set, break characters will be queued and transmitted. SBK must be cleared for the transmitter to resume sending normal characters.

The SCI Status Register (SCSR) contains two bits that provide status for the transmit operations, as shown in Figure 13.8d. The Transmit Data Register Empty (TDRE) flag indicates that the last character has been moved to the shift register and the SCDR is ready to receive another character. When TDRE = 1, the SCDR is empty and ready for a new character. When TDRE = 0, the SDCR still contains the previous data. The Transmit Complete (TC) flag indicates that the last character has been transmitted and that the transmitter has entered the idle state. When TC = 1, the transmitter has completed sending the last character. When TC = 0, the transmitter is still busy transmitting. The TDRE and TC bits are cleared by a two-step process. The SCSR must be read when TDRE = 1 and/or the TC = 1 followed by a write to the SCDR.

Example 13.2

Problem: Write a program to transmit the short message "Hello!" via the SCI to a terminal running a terminal emulator like PROCOMM. Configure the SCI for 9600 Baud, 8 data bits and 1 stop bit. Place the message starting with $0000 and terminate the message with the EOT character. Assemble the program at $100.

Solution: The first step is to initialize the X and Y registers. The X register is used as the register base address so that the indexed address mode can be used to access the registers. The Y register will be the pointer into the data message. Next the baud rate must be set and the SCI configured for transmit of 8-bit data. Finally, the transmitter needs to be enabled. Once the system is enabled, a loop must be executed that includes clearing the transmit data flag (TDRE) and writing a character to the data register. Loop until complete.

```
0001                      *       This program is designed to transmit a short
                                  message to monitor via
0002                      *       the SCI without using BUFFALO subroutines.
0003
0004 1000                 REGBASE EQU   $1000
0005 002b                 BAUD    EQU   $2B
0006 002c                 SCCR1   EQU   $2C
0007 002d                 SCCR2   EQU   $2D
0008 002e                 SCSR    EQU   $2E
```

```
0009 002f                  SCDR    EQU   $2F
0010
0011 0100                          ORG   $0100
0012 0100 ce 10 00                 LDX   #REGBASE
0013 0103 18 ce 00 00              LDY   #MSG
0014
0015 0107 86 18                    LDAA  #%0011000  ;Set BAUD rate for 9600
0016 0109 a7 2b                    STAA  BAUD,X
0017
0018 010b 6f 2c                    CLR   SCCR1,X    ;Set Xmit Mode to 8-bit
0019
0020 010d 86 08                    LDAA  #%00001000 ;Enable transmitter
0021 010f a7 2d                    STAA  SCCR2,X
0022
0023 0111 a6 2e            XMIT    LDAA  SCSR,X     ;Read Status Reg to clear
                                                      flags
0024
0025 0113 18 a6 00                 LDAA  0,Y        ;Get character of message
0026 0116 81 04                    CMPA  #$04       ;Check for EOT 0027
0118 27 0a                         BEQ   DONE       ;If EOT xmit is done
0028 0029 011a a7 2f                STAA  SCDR,X     ;xmit the character
0030
0031 011c 1f 2e 40 fc      WAIT    BRCLR  SCSR,X #%01000000 WAIT   ;Wait for TC
0032 0120 18 08                    INY              ;point to next character
0033
0034 0122 20 ed                    BRA     XMIT     ;Do it again
0035
0036 0124 3f               DONE    SWI
0037
0038
0039 0000                          ORG     $0000
0040 0000 48 65 6c 6c 6f 21  MSG   FCC     Hello!'
0041 0006 04                       FCB     $04
```

SCI Receiver

The SCI receiver functions in the same manner as the receive features of a UART (Universal Asynchronous Receiver/Transmitter). Figure 13.9 illustrates the receive function.

The receive function is enabled via the RE bit in the SCCR2 register. The receiver automatically starts looking for a start bit. After the start bit is detected, the character data is shifted into the shift register one bit at a time. When the stop bit is properly detected, the received character is moved to the SCDR and the status flags are updated, RDRF is set and error flags are updated appropriately. Each time a character is moved to the SCDR from the shift register, the RDRF flag is set. This indicates to the user that the character in the SCDR register

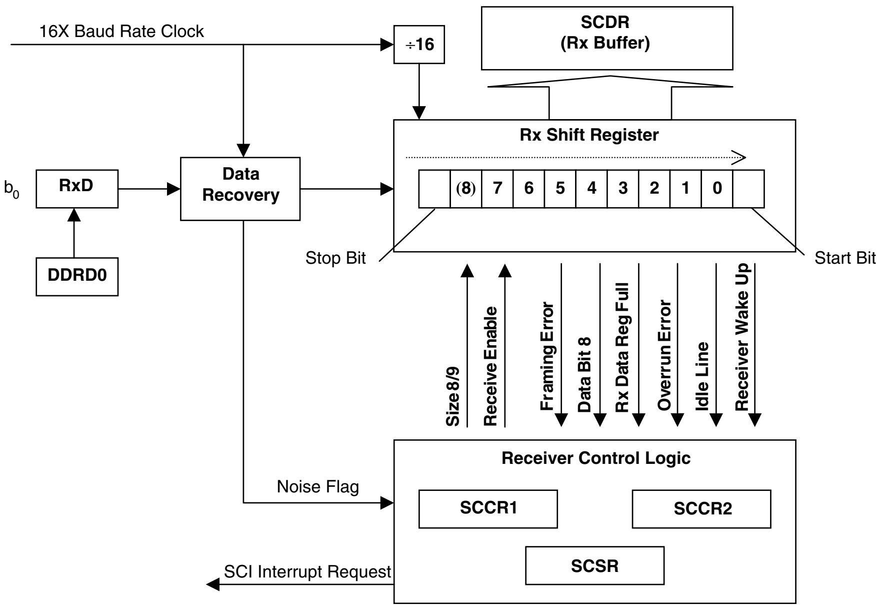


Figure 13.9 SCI Receiver Functional Block Diagram

needs to be read to avoid an overrun error. The IDLE flag is set when the entire transmission is complete and the RxD line returns to the idle state.

The SCDR register is used for the transmit operations as well as for the receive operations. The layout of the SCDR register is shown in Figure 13.8a.

The process of receiving characters is controlled by the bits in two SCI Control Registers (SCCR1 and SCCR2). The SCCR1 register contains three bits that directly control the SCI transmit function, as shown in Figure 13.10. The SCI receives data characters of 8 or 9 bits in length. The M bit is the mode bit that selects this character format. When M = 0, the character format is set to 8-bit data. When M = 1, the character format is set to 9-bit data. If the M bit is set, the R8 bit receives the 9th bit of the transmit data character when the lower eight bits are transferred to the SCDR from the shift register. The WAKE bit determines the receiver wake-up method when the RWU bit in the SCCR2 register is set. When WAKE = 1, an address mark will cause wake-up. An address mark is defined as a logic one in the MSB (8th or 9th bit position depending upon the state of the M bit) of the character. When WAKE = 0, an idle line condition will cause the receiver to wake up.

The SCCR2 register contains four bits that are relevant to the receive function. The Receive Interrupt Enable (RIE) bit is used to enable the SCI interrupt when the receive data register is full. When RIE = 1, an SCI interrupt is requested if the RDRF or the OR bits in the SCSR register are set. When RIE = 0, RDRF and OR interrupts are disabled. The Idle-Line Interrupt Enable (ILIE) bit is used to enable the SCI interrupt when the receive operation is complete. When ILIE = 1, an SCI interrupt is requested if the IDLE bit in the SCSR register is set. When TIE = 0, IDLE interrupts are disabled.

The Receive Enable (RE) bit of the SCCR2 register is used to enable the SCI receive function. When RE =1, the receiver begins waiting for character data. When RC = 0, the SCI receiver is disabled and the RDRF, IDLE, OR, NF and FE receive status flags cannot be changed. The last bit in the SCCR2 register that is relevant to the receive operation is called Receiver Wake Up (RWU). It is used to place the receiver into a standby mode until some hardware condition is met to wake up the sleeping receiver. The specific hardware condition depends on the state of the WAKE bit in the SCCR1. Wake-up mode is only used when receiver-related interrupts are inhibited. When RWU = 1, the receiver enters standby mode.

The SCI Status Register (SCSR) contains five bits that provide status for the transmit operations, as shown in Figure 13.10c. The Register Data Register Full (RDRF) flag indicates that the shift register has moved a completed character to the SCDR where the software can read it. When RDRF = 1, the SCDR contains a received character. When RDRF = 0, nothing has been received from the shift register since the last time the SDCR was read. The Idle-Line Detect (IDLE) flag indicates that the RxD data line has become idle. Idle is defined as at least one full character of all 1's. When IDLE = 1, the idle condition has been detected on the RxD line. When IDLE = 0, the RxD is either currently active or has never become active since the last time the IDLE flag was cleared.

The Overrun Error (OR) indicates that another character has been received serially and is ready to be transferred to the SCDR, but that the previous character has not yet been read by the software. When an overrun occurs, the character in the shift register is lost and the data in the SCDR is not disturbed. When OR = 1, the overrun condition has occurred. When OR = 0, no overrun error has occurred.

SCCR1	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$102C	R8	T8	–	M	WAKE	–	–	–
RESET	I	I	0	0	0	0	0	0

a) SCI Control 1 Register (SCCR1) Layout

SCCR2	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$102D	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
RESET	0	0	0	0	0	0	0	0

b) SCI Control 2 Register (SCCR2) Layout

SCSR	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$102E	TDRE	TC	RDRF	IDLE	OR	NF	FE	–
RESET	1	1	0	0	0	0	0	0

c) SCI Status Register (SCSR) Layout

Figure 13.10 SCI Control and Status Layout for Receive

The Noise Flag (NF) indicates that the data recovery logic detected noise during the reception of the character. Noise is indicated if three samples taken during the center of the data bit or stop are not the same. The noise test for the start bit involves three bits in the middle of the bit and four samples at the start of the bit. All seven samples must be zero. The NF flag is only set for a data character that was successfully transferred to the SCDR. The NF flag is updated at the same time the RDRF bit is updated. When NF = 1, noise was perceived for the last character. When NF = 0, no noise was detected during the reception of the character that is in the SCDR.

The Framing Error (FE) flag indicates that a new start bit was detected during the time that the stop bit was expected for the current character. Asynchronous serial data reception requires the receiver to be aligned with the character reception. This alignment is achieved by detecting the falling edge of the start bit. The alignment is verified by checking for the proper logic state (logic one) during the stop bit. When the alignment cannot be verified, a framing error has occurred. The FE flag is only set for a data character that was successfully transferred to the SCDR. The FE flag is updated at the same time the RDRF bit is updated. When FE = 1, a framing has occurred. When FE = 0, no framing error was detected during the reception of the character currently in the SCDR.

The RDRF, IDLE, OR, NF and FE bits are cleared by a two-step process. The SCSR must be read when one or more of the bits are set followed by a write to the SCDR.

Example 13.3

Problem: Write a program to receive data from a terminal running a terminal emulator like PROCOMM via the SCI until a <CR> is received. Store the received data in a BUFFER starting at $01E0. Configure the SCI for 9600 Baud, 8 data bits and 1 stop bit. Assemble the program at $120.

Solution: The first step is to initialize the X and Y registers. The X register is used as the register base address so that the indexed address mode can be used to access the registers. The Y register will be the pointer into the buffer where the characters will be stored. Next the baud rate must be set and the SCI configured for receive of 8-bit data. Finally, the receiver needs to be enabled. Once the system is enabled, a loop must be executed that includes clearing the receive data flag (RDRE) and reading a character from the data register and store it in the BUFFER. Loop until complete.

```
0001                *   This program is designed to receive a short message
                        from the
0002                *   keyboard via the SCI without using BUFFALO subroutines.
0003
0004 1000           REGBASE  EQU    $1000
0005 002b           BAUD     EQU    $2B
0006 002c           SCCR1    EQU    $2C
0007 002d           SCCR2    EQU    $2D
0008 002e           SCSR     EQU    $2E
0009 002f           SCDR     EQU    $2F
```

```

0120                        ORG     $0120
0120 ce 10 00               LDX     #REGBASE
0123 18 ce 01 e0            LDY     #BUFFER

0127 86 18                  LDAA    #%0011000    ;Set BAUD rate for 9600
0129 a7 2b                  STAA    BAUD,X

012b 6f 2c                  CLR     SCCR1,X      ;Set Rcv Mode to 8-bit

012d 86 04                  LDAA    #%00000100   ;Enable receiver
012f a7 2d                  STAA    SCCR2,X

0131 1f 2e 20 fc  RCV       BRCLR   SCSR,X #%00100000 RCV ;Wait for data to
                                                             arrive

0135 a6 2f                  LDAA    SCDR,X       ;Get character from Data Reg
0137 81 0d                  CMPA    #$0D         ;Check for CR
0139 27 07                  BEQ     DONE         ;If CR rcv R is done

013b 18 a7 00               STAA    0,Y          ;save the character
013e 18 08                  INY                  ;point to next character

0140 20 ef                  BRA     RCV           ;Do it again

0142 3f           DONE     SWI



01e0                       ORG    $01E0
01e0             BUFFER   RMB    32           ;reserve 32 bytes
```

Self-Test Questions 13.2

1. How are the PORTD pins configured when they are not being used for serial communications?
2. Can the SCI transmitter and receiver be used simultaneously?
3. What register is used to transfer data to the SCI as well as to read data that has been received?

13.3 Serial Peripheral Interface (SPI)

The Serial Peripheral Interface (SPI) is another alternative function available on the Port D pins. This interface is a bidirectional synchronous serial interface. It supports a master/slave configuration with multiple slaves in the configuration. In addition, it allows for simultaneous transmit and receive functions. While data is being transmitted from the master, the master can also be receiving data from the slave device.

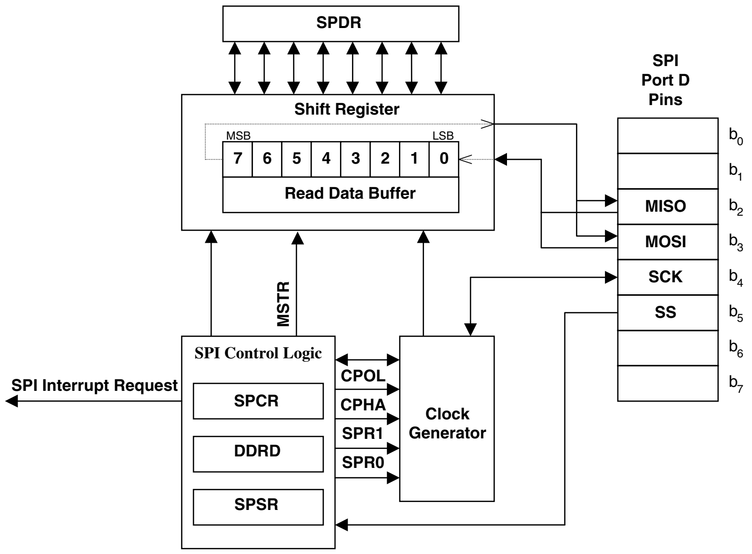


Figure 13.11 SPI Functional Block Diagram

The block diagram of the SPI system is shown in Figure 13.11. The SPI has the ability to transmit and receive data simultaneously. Data is set up to transfer by writing to the Serial Peripheral Data Register (SPDR). This data is transferred to the shift register for transmission. While a word is being shifted out, data is shifted in. When a whole data word is received, it is transferred into an internal data buffer. The same register address of the SPDR is used to both write to the shifter and read from the data buffer. Data is transmitted MSB first on this system.

There are four Port D pins, which are used for the SPI system. Slave Select (SS) is used to select a slave device. When SS = 0, the device is designated as a slave device. When a device is configured as a master, the SS pin must remain high. Serial Clock (SCK) is the input clock to the slave device. It is generated by the master device and is used to synchronize the data movement over the SPI interface. Since the SPI is a serial interface, it requires eight clock pulses to transfer a byte of data. Since the SPI interface is synchronous, it does not use start and stop bits; thus it requires only eight clock pulses to transfer a byte of data. The start and stop control is derived from the SS bit. The phase and polarity of the clock on SCK are selected by the CPOL and CPHA bits in the SPCR. Both master and slave device must be configured for the same polarity and clock phase. The clock rate is selected by the SPR1and SPR0 bits in the SPCR on the master device. The SPR1 and SPR0 bits on the slave device are ignored.

The SPI interface has two unidirectional data pins. Master In Slave Out (MISO) is used as the data input pin on a master device. It is used as the data output pin on a slave device. MISO is placed in a high impedance state if the slave device is not selected via the SS pin. Master Out Slave In (MOSI) is used as the data output pin on a master device. It is used as

Port D Pins	b7	b6	b5	b4	b3	b2	b1	b0
	–	–	SS	SCK	MOSI	MISO	TxD	RxD
RESET	0	0	0	0	0	0	0	0

DDRD	b7	b6	b5	b4	b3	b2	b1	b0
$1009	–	–	DDRD5	DDRD4	DDRD3	DDRD2	DDRD1	DDRD0
RESET	0	0	0	0	0	0	0	0

Figure 13.12 Relationship of SPI Pins to Bits in the DDRD Register

the data input pin on a slave device. Data from the master is placed on the MOSI pin ½ clock cycle prior to the clock edge used by the slave device to latch the data.

When a device is configured as a master, the SS, SCK and MOSI pins must be configured as outputs and the MISO bit must be configured for input. This is done by writing 1's to DDRD5–DDRD3 in the DDRD register. When a device is configured as a slave, the MISO bit must be configured as an output and MOSI, SCK and SS must be configured as input. This is done by writing a 1 to DDRD2 of the DDRD register. The relationship of the pins to the DDRD bits is shown in Figure 13.12.

NOTE: SPI pins that are configured for output will not operate unless the corresponding bit in the DDRD register is set. SPI pins that are configured for input ignore the state of the corresponding DDRD bits.

Once the interface is configured, data can be transferred between the master and the slave. If properly configured, both devices can simultaneously transfer data. The transfer is initiated by writing data to the SPI Data Register (SPDR) on the master. The act of writing the data causes the whole process to begin. The SPDR is an 8-bit data register that is directly linked to the internal SPI shift register. Received data is also available via the SPDR. At the completion of a transfer, the received data is placed in the receive data buffer. When the user reads the SPDR, the received data is moved from the receive buffer to the SPDR so that it can be read. The SPDR is located at $102A in the memory map; it is shown in Figure 13.13.

The SPI process is configured and controlled by seven bits contained in the SPI Control Register (SPCR), as shown in Figure 13.14. This register can be read or written at anytime. The DDRD register must also be configured for the SPI system to operate properly. The SPI Interrupt Enable (SPIE) bit is used to enable SPI interrupt requests when the SPIF or the MODF bits of the SPSR is set. When SPIE = 1, the SPI interrupt is requested. When SPIE = 0, the SPI interrupt is inhibited. The SPI System Enable

SPDR								
$102A	b7	b6	b5	b4	b3	b2	b1	b0
RESET	U	U	U	U	U	U	U	U

Figure 13.13 SPI Data Register Layout

SPCR	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
\$1028	SPIE	SPE	DWOM	MSTR	CPOL	CPHA	SPR1	SPR0
RESET	0	0	0	0	0	1	U	U

a) SPI Control Register Layout

SPR1	SPR0	E Clock Divisor E/x	Transfer Bit Rate
0	0	E/2	1.0 MHz
0	1	E/4	500 kHz
1	0	E/16	125 kHz
1	1	E/32	62.5 kHz

b) SPI Clock Rates for 2 MHz E Clock

Figure 13.14 SPI Control

(SPE) bit is the master enable for the SPI system. This bit must be set to activate the features of the SPI system. When SPE = 1, the SPI is enabled. When SPE = 0, the SPI is disabled.

The Master/Slave Mode Select (MSTR) bit configures the SPI system as a master or a slave device. Regardless of the mode, the SPI system can simultaneously send and receive data. When MSTR = 1, the SPI system is configured as a master. When MSTR = 0, the SPI system is configured as a slave. The SPI Clock Rate Selects (SPR1 and SPR0) are used to select the transfer bit rate. This transfer rate is a function of the E clock rate. The four possible transfer rates are shown in Figure 13.14b.

The Clock Polarity Select (CPOL) bit configures the active and idle states of the SPI clock present at the SCK pin of the interface. When CPOL = 1, active low clocks are selected and the SCK pin will idle in the high state. When CPOL = 0, active high clocks are selected and the SCK pin will idle in the low state. The SPI support two different transfer timing formats. The two formats are shown in Figure 13.15. The Clock Phase Select (CPHA) bit selects one of the two transfer formats. This bit is used in conjunction with the CPOL to determine the actual clock that will be present on the SCK pin.

The SPSR register layout is shown in Figure 13.16. The SPI Transfer Complete Flag (SPIF) bit is set to indicate that a transfer is complete. This bit is cleared by reading the SPSR followed by accessing the SPDR. The Write Collision flag (WCOL) is set if the MCU tries to write data into the SPDR while an SPI data transfer is in progress. Clear this bit by reading the SPSR followed by accessing the SPDR. The Mode Fault flag (MODF) is set when SS is pulled low while MSTR = 1. Clear this flag by reading the SPSR with the MODF bit set, followed by a write to the SPCR.

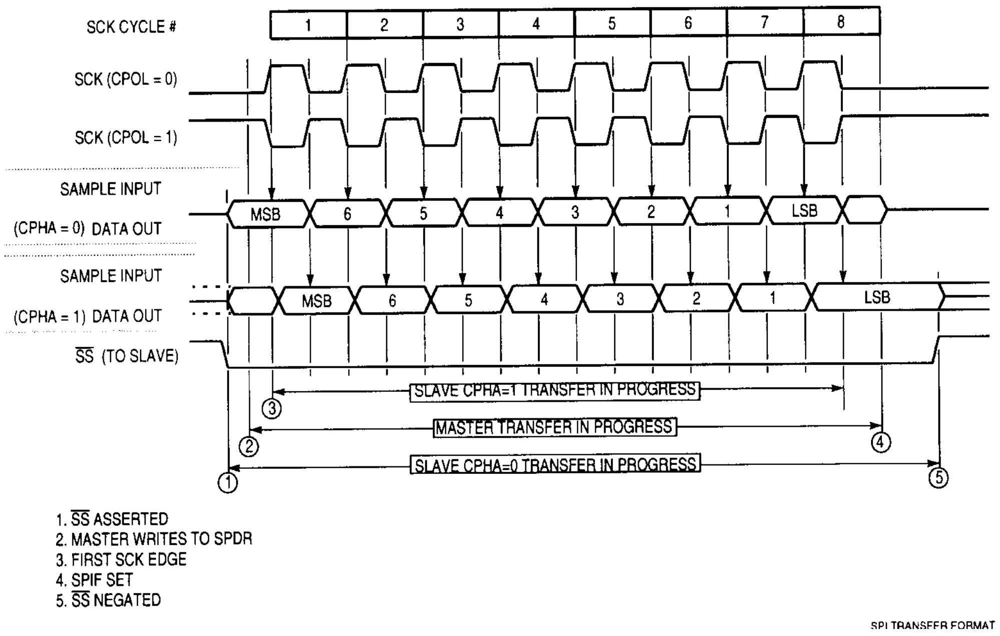


Figure 13.15 SPI Transfer Timing Formats

SPSR	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$1029	SPIF	WCOL	–	MODF	–	–	–	–
RESET	0	0	0	0	0	0	0	0

Figure 13.16 SPI Status Register Layout

Self-Test Questions 13.3

1. What are the two communications functions available in the Port D communications system?
2. Is the SPI a synchronous or an asynchronous interface?
3. What register must be written to start an SPI transfer?

Summary

The Port D serial communications system consists of two communications interfaces, the SCI and the SPI. The SCI is an asynchronous interface, and the SPI is a synchronous interface. When the Port D pins are not being used for serial communications, they can be configured for digital I/O. The DDRD register controls the direction of the I/O on these pins.

Chapter Questions

Section 13.1

1. Define serial data.
2. What was a reason that RS-232 was first developed?
3. In what order are the data bits transferred within a character in an asynchronous system?
4. What is the purpose of the start bit in an asynchronous system?
5. What is the key difference between an asynchronous system and a synchronous system?

Section 13.2

6. Which two Port D pins are used by the SCI system?
7. What is the master clock that drives the SCI timing?
8. What are the names of the bits in the BAUD register used to control the SCI timing prescale?
9. What are the names of the bits in the BAUD register used to select the specific BAUD rate?
10. What value should be written to the BAUD register to select 300 BAUD?
11. What state is automatically transmitted when the SCI transmitter is enabled?
12. What bit in the SCCR2 register is used to enable transmit data register empty interrupt?
13. What happens when the SBK bit is set?
14. What process is necessary to clear the SCI transmit flags?
15. What bit is used to enable 9-bit characters?
16. Which bit enable interrupts when the receive data register is full?
17. Why would an overrun error occur, and how does the SCI system indicate that it happened?

Section 13.3

18. Which of the Port D pins are used by the SPI system?
19. What is the fastest transfer rate of the SPI system?
20. Which signal is used to signal the receiving device that data will be sent?
21. What is the required state of the corresponding DDRD bits when SPI pins are configured for input?
22. Can there be more than one slave in an SPI system?
23. Can there be more than one master in an SPI system?
24. For continuous transmission, why would it be faster to send data each time TDRE is set rather than waiting for TC to set?
25. What process is necessary to clear SPIF?

Chapter Problems

1. Write a program to configure the Port D system for serial asynchronous communication. Use 9600 baud, 8-bit data and 1 stop bit and enable both receive and transmit functions.

2. Calculate the total amount of transmit time for 10 characters in the SCI system with the following format: 8-bit data, 1 start bit, 1 stop bit, no parity. Use 4800 BAUD. Recalculate for 300 BAUD.

Answers to Self-Test Questions

Section 13.1

1. A parallel in serial out shift register is used.
2. One start bit, 7 or 8 data bits, optional parity bit and 1–2 stop bits.
3. The state of the idle line is high.
4. Synchronous uses control signals and a clock to synchronize the transmission, and asynchronous uses header and footer bits attached to the data character.

Section 13.2

1. They are configured for digital I/O. The value in the DDRD register controls the direction of each of these I/O bits.
2. Yes. The SCI system is a full duplex communication system.
3. The SCDR is used for transmit and receive data.

Section 13.3

1. The SCI (Serial Communications Interface) and the SPI (Serial Peripheral Interface).
2. The SPI is a synchronous interface.
3. A byte of data must be written to the SPDR register to start the transfer.

chapter 14

C Programming Using the HC11

Objectives

After completing this chapter, you should be able to:

- Understand the basic concepts of programming in a higher-level language
- Write simple C statements and expressions
- Write short C programs that will accomplish specific tasks on the HC11

Outline

Introduction

In chapter 2 the idea of a computer language was presented. Higher-level languages are more English-like and therefore more readable by humans, while the lower-level languages are more similar to the actual machine code that is used on the machine.

This chapter teaches principles surrounding the C programming language and the relationship of C program statements to the resulting assembled machine code that would run on an HC11 microcontroller. In no way is this presentation a complete discussion of the numerous features, strengths and weaknesses of the C language; it is rather a brief overview of some common features and their direct correlation to blocks of assembly code.

The C language was developed by engineers at Bell Labs during the 1970s as a tool to develop the UNIX operating system. Since then, it has evolved into the primary language used for most large programming and software development projects. It has strong uses in engineering and scientific applications. It is also used as the core language when teaching programming.

C is considered a medium-level language. It has many capabilities of the highest-level languages and yet supports assembler-level code embedded in the source code files. C source code files are not converted directly to machine code by an assembler. The files are first processed by a compiler that converts them to the equivalent assembly-language source code. This assembly-level code is then assembled into the machine code. Each of the examples provided at the end of this chapter shows this relationship.

14.1 C Programming

The C programming language uses English-like verbs and key words, thus it reads in a fashion similar to a spoken language. This is in contrast to a low-level language-like assembler that consists of mnemonic instructions that translate into single machine operations. C consists of a series of program statements, each terminated by a semicolon. Each **statement** can perform the equivalent of one or many assembly-level instructions. Each statement consists of compiler keywords, user-defined variable names and operators. Figure 14.1 contains five examples of statements in C.

More than one statement can be put on a single line, or one statement can be written across many lines, as shown in Figure 14.2.

C ignores almost any combination of white space within the code. The program may contain any number of blank spaces, blank lines, tabs and new line characters. This rule regarding white space applies to all white space except for spaces contained inside of double quotes, function names and reserved words. For example, a keyword cannot contain a blank or new line character. An example of the improper use of the new line

```
z = x + y;
radius = 2.0;
sum = sum + result;
volume = (4.0/3.0) * 3.1416 * pow(radius,3);
printf("The volume of the sphere is %f, volume);
```

Figure 14.1 Examples of C Statements

```
x = 2; y = 43; z = x + y;

printf
("Hello World");
```

Figure 14.2 Clustered and Split Statements in C

```
prin
tf("This would not work because the keyword printf is split");
```

Figure 14.3 Improper Use of White Space

```
{
  x = 120;
  y = 64;
  z = x / y;
}
```

Figure 14.4 Use of the Braces to Form a Complex Statement

```
main () {
  statements;
}
```

Figure 14.5 The Main Statement in C

character is shown in Figure 14.3. In this example the keyword "printf" is split, which will cause a compiler error.

Statements are grouped together in functional groups by the braces { }. These braces are often referred to as curly brackets. This grouping of statements is sometimes required by the structure of the program and at other times is optional. Figure 14.4 shows one example of three statements being grouped together by braces into a single **complex statement**. A complex statement is sometimes referred to as a **code block**.

Each C program is structured around a series of programming routines or subroutines. These subroutines are called **functions** in C. The primary routine is designated by the keyword **main ( )**. A program must contain one and only one main ( ). The syntax for the main ( ) statement is shown in Figure 14.5.

Self-Test Questions 14.1

1. What is a C statement?
2. How can white space be used in the C source code?
3. What is the purpose of the braces { }?
4. What is the name of the primary function of each C program? How many primary functions can a C program have?

14.2 Data Types

A data type is used by the compiler to allocate memory and to establish a set of rules for a particular data variable. There are four basic data types supported in C: character, integer, floating point and double precision. When a variable is declared in C, it is assigned to a particular data type. The type controls which kinds of operations can be performed using the particular variable.

```
char a, b;  /*declares two single byte variables a and b. */
```

Figure 14.6 Char Data Type

```
int x;             /*declares a 16 bit signed variable named x.   */
unsigned int y;    /*declares a 16 bit unsigned variable named y. */
```

Figure 14.7 Int Data Type

```
float firstnum;   /*declares a floating point variable called firstnum. */
double total;     /*declares a double precision variable called total.  */
```

Figure 14.8 Float and Double Data Types

The **character** type consists of single bytes of data. This byte can contain any 8-bit data element, including ASCII character codes, signed and unsigned 8-bit binary/hexadecimal data and decimal numbers. Examples include "A", $45, –123. The programmer can use any format that makes the most sense in the source code; however, the compiler will convert this data to the binary/hex format in the actual machine code. The character data type is declared using the **char** keyword, as shown in Figure 14.6.

Integer data consists of two bytes (on some machines the basic integer type is four bytes). Any signed or unsigned integer number (a number that has no digits to the right of the decimal point) that can be represented in 16 binary bits can be stored, processed and manipulated by this data type. Examples include $13AB, +3568, –491. The range of 2-byte signed decimal values is –32,768 to +32,767. The range of 2-byte unsigned decimal values is 0 to 65,535. The integer data type is declared using the **int** keyword. The default integer type is considered signed. If an unsigned type is desired, the int keyword must be preceded by the **unsigned** keyword. Examples of the int and unsigned keywords are shown in Figure 14.7.

The **floating point** and **double precision** data types are designed to handle any signed or unsigned numbers containing a decimal point. Examples include +10.625, –6.2, 0.33, 0.0, 100.0, 1.625*e*3, –7.23*e*–4. The floating point type is sometimes called **single precision** because it typically requires half of the memory storage space of the equivalent double precision type. The "*e*" character represents exponent or power of ten and provides a simple way of showing numbers in scientific notation. The floating point type uses the **float** keyword, and the double precision type uses the **double** keyword, as shown in Figure 14.8.

Self-Test Questions 14.2

1. What is a data type?
2. How many bytes are occupied by each of the following data types: char, int and unsigned int?
3. What is the difference between a float and an int type?

14.3 Operators

Data of any type can be added, subtracted, multiplied or divided. Each arithmetic function has a defined operator, as shown in Figure 14.9. Each operator is used to indicate the respective arithmetic operation.

Each arithmetic operator requires two operands; therefore, it is called a **binary operator** (**unary operators** require only one operand). The operator is used between the two operands to designate the desired arithmetic operation. Figure 14.10 illustrates a variety of arithmetic operations using the C operators and syntax.

The division of two integers produces only the integer quotient. Sometimes it is desirable to know the integer remainder when division is performed. A special operator is defined (%) called the modulus operator that returns only the remainder of a division problem. Like the other arithmetic operators, it is a binary operator. It is used in the same manner as the division operator for integer values. Figure 14.11 illustrates the use of this operator.

There is one **unary operator** relevant to this discussion, the negation operator (–). When the minus sign is used in front of any numerical value, it negates (changes the sign of) the number.

Each operator has a relative order of precedence. When two or more operators occur within the same statement, an order must be established for the processing of the operations or the results would be ambiguous. Precedence can be changed by the use of parentheses ( ). Operations grouped within parentheses will be evaluated first,

Operation	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/

Figure 14.9 Arithmetic Operators

`3 + 7`	This operation designates the addition of two integer values 3 and 7.
`18 – 3`	This operation designates the subtraction of two integer values 18 and 3.
`12.62 + 9.8`	This operation designates the addition of two floating point values 12.62 and 9.8.
`7 * 4`	This operation designates the multiplication of two integer values 7 and 4.
`2.6 / 3.0`	This operation designates the division of two floating point values 2.6 and 3.0.

Figure 14.10 Examples of Arithmetic Operations

`5/3`	This results in the integer quotient of 1.
`5%3`	This results in the integer remainder of 2.

Figure 14.11 Example of the % Operator

Operator	Associativity
Unary -	right to left
(,)	left to right
*, /, %	left to right
+, -	left to right

Figure 14.12 Order of Precedence of Operators

followed by the operations outside the parentheses. Multiple sets of parentheses are evaluated in a left-to-right fashion. If parentheses are contained within other parentheses, the innermost parentheses are evaluated first.

All operators have a set order of precedence, as shown in Figure 14.12.

Example 14.1

Problem: Evaluate the following expression, using the proper order of precedence.

8 + 5 * 7 % 2 * 4

Solution: As the * and % operators have the same precedence, they will be evaluated one at a time, left to right, before the + operation is evaluated. Each operation is shown with the underline.

$\Rightarrow 8 + \underline{5 * 7} \% 2 * 4$	5 * 7 = 35
$\Rightarrow 8 + \underline{35 \% 2} * 4$	35 % 2 = 1
$\Rightarrow 8 + \underline{1 * 4}$	1 * 4 = 4
$\Rightarrow \underline{8 + 4}$	8 + 4 = 12
$\Rightarrow 12$	Final result

Self-Test Questions 14.3

1. What is an arithmetic operator?
2. What symbol is used for the operator that returns the remainder, rather than the quotient, from an integer division operation?
3. What is the difference between a unary and binary operator?
4. Which operator has highest order of precedence?

14.4 Variables

All data on a computer must be stored in the memory. Before higher-level languages like C were used, the source code specified the exact storage locations used in the memory. High-level languages use symbolic names in place of actual memory addresses to designate where the data is being stored. These symbolic names are called **variables**. Each variable is simply a name, given by the programmer to a storage location. In C,

```
year = 1988;
sum = 6 + 5 + 18;
rate = prime;
inches = 12 * feet;
classAvg = (grade1 + grade2 + grade3) / 3.0 * factor;
```

Figure 14.13 Examples of the Use of the Assignment Operator

```
sum = sum + newvalue;
value = value * (1 + rate);
```

Figure 14.14 Examples of Accumulation

these variable names can be almost anything. Definition of the variable names is left to the programmer and follows two simple rules.

1. The names must begin with a letter or underscore (_) and may only contain characters, digits or the underscore. They cannot contain any blanks (spaces) or special symbols, such as ( ) & $ # . , ! ?
2. A variable cannot be a keyword (keywords are reserved and have special meaning).

Naming and defining the data type that can be stored in each variable is accomplished by using declaration statements. A **declaration** statement, in its simplest form, provides a data type and a variable name. Examples of variable declarations were provided in Figures 14.6, 14.7 and 14.8.

In C, the equal sign (=) is used in assignment statements. It is a binary operator. The = symbol is called the **assignment operator** and is used to assign an operand to a variable. The variable appears on the left side of the operator, and the operand appears on the right of the operator. The operand on the right can be a constant (i.e., 2, 4.3, –16.3*e*5), another variable designation or a valid C expression. The assignment operator causes the value of the operand to be evaluated first and then be stored in the memory location(s) named by the variable; thus it has a lower relative precedence than the other operators already mentioned. Examples of valid assignment expressions are shown in Figure 14.13.

Accumulation can take place by using the variable on both sides of the assignment operator in combination with another arithmetic operator. The current value stored in the variable is then used to calculate the new value, which is then stored in the variable, overwriting the previous value. Examples of accumulation are shown in Figure 14.14.

Example 14.2

Problem: Write a short segment of C code that will calculate the average of three grades: 85, 93 and 98. Use integer variables for all values.

Solution: The variables must be declared first, then assign the grades and calculate the average.

```
Int    Grade1, Grade2,        /* variables for grades */
       Grade3;
Int    Average;               /* variable for average */
       Grade1 = 85;
       Grade2 = 93;
       Grade3 = 98;
       Average = (Grade1 + Grade2 + Grade3) / 3;
```

Self-Test Questions 14.4

1. What is a variable? How is it used?
2. What is the assignment operator used for?
3. How can accumulation be accomplished?

14.5 Special Statements

There are many special statements defined in C to perform a variety functions. These special statements use specific keywords and follow a set format (called the syntax). Consider the following special statements.

If-then-else statement

The **if-then-else** format allows statements to be written that will be executed only if the conditional expression is true. If the conditional expression is not true, an alternative statement following the else keyword can be executed. The else keyword is optional. An **expression** is similar to a program statement but uses relational operators like <,> or ==, and evaluates to a true or false state. The double equal is used for relational equality to show a visual difference from an assignment. The general syntax for the if-then-else structure is shown in Figure 14.15.

```
if(expression)
     statement1;
else
     statement2;
```

Figure 14.15 General Syntax of the If-Then-Else Statement

Example 14.3

Problem: Write a short C code segment using the if-then-else structure that will set the timeofday to afternoon if the hour is at least 12.

Solution: First the expression must be evaluated to determine if the condition is true. If it is true, then execute the statement.

```
if (hours >= 12)
        timeofday = afternoon;
```

```
for (initializing list; expression; altering list)
     statement;
```

Figure 14.16 General Syntax of the For Statement

Example 14.4

Problem: Write a short C code segment using the if-then-else structure that will check to see if x is greater than zero. If x is greater than zero, then assign x plus 1 to y, else assign zero to y.

Solution: First the expression must be evaluated to determine if the condition is true. If it is true, then execute the statement, else execute the other statement.

```
if (x > 0)
        y = x + 1
else
        y =0 ;
```

For Statement

The for statement format allows statements to be written that will be executed for a certain duration or while a conditional expression is true. The syntax for the "for" statement requires three terms: an initializing list, an expression and an altering list. The initializing list indicates the starting condition of the conditional variable. The expression designates what condition must be true for the statements to be executed. The condition is tested before the statements are executed. The altering list indicates how the conditional variable will be changed. Each term must be separated by the semicolon within the parenthesis, as shown in Figure 14.16.

Example 14.5

Problem: Write a short C code segment using the "for" structure that will set up a loop to execute 20 times.

Solution: First the loop counter (i) is initialized, the loop condition is set (< 20) and how the loop counter will be altered (i+1) is defined. The body of the "for" loop is a complex statement that assigns a value to the variables x and y.

```
for i = 1; i < 20; i = i + 1
        {
        x = x + y;
        y = z * 7 % 2;
        };
```

Self-Test Questions 14.5

1. For what purpose is the if-then structure used?
2. What does the keyword "else" provide to the if-then structure?
3. How is the "for" statement similar to the if-then structure?

14.6 Subroutines

Subroutines in C are called **functions**. The programmer can write functions to perform any task. As is explained in chapter 6, functions (i.e., subroutines) provide the ability to reuse the same code over and over without physically duplicating the code. Figure 14.17 provides an example of how subroutines work. The name of the function becomes the keyword that causes the function to be executed. Any time the name of the function is followed by the parenthesis characters (i.e., addemup()) in the source code, the function is called and the statements within the function are executed. The compiler identifies the functions by the trailing parenthesis. If the parentheses are not used, the compiler will interpret the function call as a variable and report an error.

```
main(){
 char x, y, z;      /* declare three 1-byte variables */
 x = 2;             /* Assign the value 2 to x */
 y = -4;            /* assign the value -4 to y */
 addemup();         /* perform the addition function *?
}
addemup()          /* addemup function code */
{
 z = x + y;         /* add the two numbers and assign the sum to z */
};
```

Figure 14.17 Example of C Function

The subroutine or C function addemup is called in the main. The function adds the two numbers together and assigns the sum to the variable z. Anything that can be done with in-line code can be done with functions.

Self-Test Questions 14.6

1. What is the main advantage of using functions instead of in-line code?
2. What keyword has to be used to call a function?

14.7 Examples

The following pages contain several small C programs and the listing files generated by a C compiler designed to produce HC11-compatible assembly and machine code. The purpose in these examples is to provide insight into the process the compiler goes through of translating the high-level commands (statements) into low-level mnemonic instructions. The function of these programs is somewhat trivial; however, they do serve to illustrate the relationship of higher-level "C" code to the lower-level assembly code.

Example 14.6

Problem: Write a C program to perform a 16-bit multiplication of two integers.

Solution: The program starts by defining three integer (2-byte) variables x, y and z. It then assigns the value 2 to x and the value 4 to y. It then multiplies x and y and assigns the result to the variable z. Since the HC11 is an 8-bit processor, it must use the 16-bit registers to perform this operation. Also note that this assembly-level program calls an external subroutine called c_imul (provided by the compiler) to actually complete the operation. The three variables z, y and x are assigned to the memory locations $0000, $0002 and $0004 respectively (see Figure 14.18).

```
int x, y, z;

main(){
 x = 2;
 y = 4;
 z = x * y;
}
```

```
                    ;     int x, y, z;
0000                      org $0000
                    _z:
0000                      rmb   2
                    _y:
0002                      rmb   2
                    _x:
0004                      rmb   2
                    ;
0100                      org $0100
                    ;     main(){
                    _main:
                    ;     x = 2;
0100  CC0002              ldd   #2
0103  FD0004              std   _x
                    ;     y = 4;
0106  C604                ldab  #4
0108  FD0002              std   _y
                    ;     z = x * y;
010B  FC0004              ldd   _x
010E  18FE0002            ldy   _y
0112  BD0119              jsr   c_imul
0115  FD0000              std   _z
                    ;     }
0118  39                  rts
                    LData:
0119                      .external c_imul
```

Figure 14.18 16-Bit Integer Multiply Example

Example 14.7

Problem: Write a C program to perform a 16-bit division of two integers.

Solution: This example defines three integer (2 byte) variables x, y, and z. It assigns the value 4 to x and the value 2 to y. It then divides x by y and assigns the result to the variable z. Since the HC11 supports 16-bit division, it uses the IDIV instruction to perform the division. The three variables z, y and x are assigned to the memory locations $0000, $0002 and $0004 respectively (see Figure 14.19).

```
int x,y,z;

main(){
 x = 4;
 y = 2;
 z = x / y;
}

0001                      ;     int x,y,z;
0002                            org   $0000
0003                      _z:
0004     0000                   rmb   2
0005                      _y:
0006     0002                   rmb   2
0007                      _x:
0008     0004                   rmb   2
0009                      ;
0010                      ;     org   $0100
0011                      ;     main(){
0012     0100                   _main:
0013                      ;     x = 4;
0014     0100  CC0004           ldd   #4
0015     0103  FD0004           std   _x
0016                      ;     y = 2;
0017     0106  C602             ldab  #2
0018     0108  FD0002           std   _y
0019                      ;     z = x / y;
0020     010B  FC0004           ldd   _x
0021     010E  FE0002           ldx   _y
0022     0111  02               idiv
0023     0112  FF0000           stx   _z
0024                      ;     }
0025     0115  39               rts
```

Figure 14.19 16-Bit Integer Division Example

Example 14.8

Problem: Write a C program to perform the addition of two 8-bit numbers using a subroutine.

Solution: This example defines three character (1-byte) variables x, y and z. It assigns the value 2 to x and the value –4 to y. It then adds x and y via a subroutine (function) and assigns the result to the variable z. Since the HC11 is an 8-bit processor, it uses the 8-bit AccB to perform this operation. The three variables z, y and x are assigned to the memory locations $0000, $0001 and $0002 respectively (see Figure 14.20)

```
char x, y, z;
main(){
  x = 2;
  y = -4;
  addemup();
}
addemup()
{
  z = x + y;
}
```

```
                      ;     char x, y, z;
0000                        org   $0000
                      _z:
0000                        rmb   1
                      _y:
0001                        rmb   1
                      _x:
0002                        rmb   1
                      ;
0100                        org $0100
                      ;     main(){
                      _main:
                      ;     x = 2;
0000  C602                  ldab  #2
0002  D702                  stab  _x
                      ;     y = -4;
0005  C6FC                  ldab  #252
0007  D701                  stab  _y
                      ;     addemup();
0010  8D01                  bsr   _addemup
                      ;     }
0014  39                    rts
                      ;
                      ;     addemup()
                      ;     {
                      _addemup:
                      ;     z = x + y;
0015  D601                  ldab  _y
0018  DB02                  addb  _x
001B  D700                  stab  _z
                      ;     }
001E  39                    rts
```

Figure 14.20 8-Bit Addition Using a Subroutine Example

Example 14.9

Problem: Write a C program that will determine if x equals y. If it does, it will set z equal to x, otherwise it will set z equal to y.

Solution: This example defines three character (1 byte) variables x, y and z. It assigns the value 5 to x and the value 6 to y. It then performs a conditional test to determine if x equals y. Note the use of the double-equals for relational equality. If x equals y, it assigns x to z, otherwise it assigns y to z. The three variables z, y and x are assigned to the memory locations $0000, $0001 and $0002 respectively (see Figure 14.21).

```
char  x, y, z;

main()
{
  x = 5;
  y = 6;
  if(x == y)
    z = x;
  else
    z = y;
}
```

```
0001    0000                       org   $0000
0002                         _z:
0003    0000                       rmb   1
0004                         _y:
0005    0001                       rmb   1
0006                         _x:
0007    0002                       rmb   1
0008                         ;
0009    0100                 ;     org   $0100
0010                         ;     main()
0011                         ;     {
0012                         _main:
0013                         ;     x = 5;
0014    0100  C605                 ldab  #5
0015    0102  D702                 stab  _x
0016                         ;     y = 6;
0014    0104  5C                   incb
0015    0105  D701                 stab  _y
0016                         ;     if(x == y)
0017    0107  D602                 ldab  _x
0018    0109  D101                 cmpb  _y
0019    010B  2604                 bne   L1
0020                         ;     z = x;
0021    010D  D700                 stab  _z
0022                         ;     else
0023    010F  2004                 bra   L11
0024                         L1:
0025                         ;     z = y;
0026    0111  D601                 ldab  _y
0027    0113  D700                 stab  _z
0028                         L11:
0029                         ;     }
0030    0115  39                   rts
```

Figure 14.21 If-Then-Else Example

Example 14.10

Problem: Write a C program to perform the addition of two 8-bit numbers using a subroutine.

Solution: This example defines two character (1-byte) variables x and i. It assigns the value 0 to x. It then enters a while loop that is implemented from the for statement in C. The job of the while loop is to increment x so long as i is greater than zero. Each time the loop is executed, i is decremented. The two variables i and x are assigned to the memory locations $0000 and $0001 respectively (see Figure 14.22).

```
char x, i;

main()
{
  x = 0;

  for(i = 5; i > 0; i = i - 1)
  {
    x = x + 1;
  }
}
```

```
                    ;     char x, i;
                    ;
0000                      org   $0000
                    _i:
0000                      rmb   1
                    _x:
0001                      rmb   1
                    ;
0100                      org   $0100
                    ;     main()
                    ;     {
                    _main:
                    ;     x = 0;
0000  7F0001              clr   _x
                    ;
                    ;     for(i = 5; i > 0; i = i - 1)
0003  C605                ldab  #5
0005  D700                stab  _i
                    L1:
0007  7D0000              tst   _i
000A  2708                beq   L11
                    ;     {
                    ;     x = x + 1;
000C  7C0001              inc   _x
                    ;     }
000F  7A0000              dec   _i
0012  20F3                bra   L1
                    L11:
                    ;     }
0014  39                  rts
```

Figure 14.22 For Loop Example

Summary

This chapter offered a brief overview of some common features of the C programming language and their direct correlation to blocks of HC11 assembly code. C is considered a medium-level language. It has many capabilities of the highest-level languages and yet supports assembler-level code embedded in the source code files. C source code files are not converted directly to machine code by an assembler. First, they are processed by a compiler that converts them to the equivalent assembly-language source code. This assembly-level code is then assembled into the machine code.

Chapter Questions

Section 14.1

1. What character is used to terminate statements in C?

2. How are statements grouped together into a complex statement?
3. What is the format of the main function of each C program?

Section 14.2

4. What is the maximum number that can be stored using an unsigned int?
5. Can hex, binary, decimal or ASCII data all be stored in unique char variable types?
6. What data type would be necessary to complete 89 + 34? Why?
7. What data type would be necessary to complete 634 * 72? Why?

Section 14.3

8. What arithmetic operators are used to indicate multiplication and division?
9. If a statement contains an addition operator and then a division operator, which is done first?
10. If a statement contains more than one multiplication operation, which is done first?

Section 14.4

11. Which of the following are valid uses of variables?

A. _Hello	B. George Patton
C. MiddleOfTheWeek	D. lastNAME
E. school.com	F. Window2

12. In the following statement which variable is unchanged and which is changed: x1 = y?
13. In the following statement what is accomplished: product = product * 4?

Section 14.5

14. What is the difference between a statement and an expression in C?
15. Is the "else" keyword required in the if-then-else structure?
16. Is the "for" structure more like the "while" or the "until" presented in chapter 4?
17. If count is initialized to zero, what value will be in count when this "for" structure completes execution?

```
for (x = 2; x < 18; x = x + 2)

      count = count + 1;
```

Section 14.6

18. What are subroutines called in "C"?
19. How is a function call identified in the source code?

Chapter Problems

1. Write a small program that will multiply the character variables X and Y if Y > X, otherwise it will perform an integer divide, X/Y.
2. Write a small program that will set the character variable R = 0 if the remainder of an integer division is equal to zero, otherwise it will set R = 1.
3. Evaluate the following expression using the proper order of precedence:

(18 – 5) / (–37) + 2 * 64

4. Evaluate the following expression using the proper order of precedence:

(524 – 5 * 88) % 3 + (–2) * (64 / 8 + 22)

Answers to Self-Test Questions

Section 14.1

1. The C statement is an instruction or operation terminated by a semicolon.
2. White space can be used anywhere in the program in any quantity with three exceptions. White space must be controlled within double quotes, function names and reserved words.
3. The curly brackets are used to designate a complex statement.
4. Each C program must have one and only one main ().

Section 14.2

1. A designation applied to a variable so that the compiler can properly allocate memory and establish a set of rules for processing the data.
2. Char variables need one byte. Integer and unsigned integer variables consist of two bytes.
3. An integer type must be a number without a decimal place; floating point numbers allow for decimal point (fractional) values.

Section 14.3

1. An arithmetic operator tells the compiler what operation to perform on the data.
2. The % symbol is used to designate the remainder instead of the quotient.
3. A unary operator requires a single operand (data element). The binary operator requires two operands (data elements).
4. The unary minus has the highest priority, followed by the parenthesis.

Section 14.4

1. A variable is a symbolic name that a programmer assigns to a location in memory.
2. The assignment operator is used to set a variable equal to some value.
3. Accumulation can be accomplished by using the variable name on the left and on the right of assignment operator in conjunction with another operator.

Section 14.5

1. The if-then allows conditional processing. The process will only be executed if the conditional expression is true.
2. The else keyword provides an alternative process to execute if the conditional expression is not true.
3. The “for” statement will execute a set of instructions so long as the condition is true.

Section 14.6

1. A function can be executed over and over without repeating the code.
2. Only the name of the function is required to call the function.

appendix A

Computer Mathematics

Introduction

All computer systems process data. Data represents information in various forms. However, all data processed by a computer must be in a binary form. For example, characters are stored in computer memories and processed by computer hardware. Most character-based data is stored as binary codes; for example, the ASCII code for the capital A is binary 1000001. Numeric data can also be stored as characters; yet it is more efficient to store numeric data as binary codes.

Since humans are generally accustomed only to the decimal number system, it is important for computers to allow decimal data to be input and output from the system. Internally the data will be processed in binary form; thus, the computer must convert decimal to binary and then binary back to decimal before output. Since the negative sign cannot be attached to a number in binary form, the conversion of signed numbers has additional concerns that must be addressed.

In addition, data is stored and processed on computer systems in 8-bit groups called **bytes**. Each byte is made up of an upper and a lower **nibble** of four bits each. A **word** is a group of any number of binary bits. A word can be 4 bits, 5 bits, 8 bits, 16 bits or 32 bits.

Remember that a calculator is a tool and cannot think. The user of the calculator controls what it is does and determines if the result is correct. It is important to learn the principles behind fundamental computer mathematical calculations. Without an understanding of the fundamentals of computer math, it cannot be known if the calculator is functioning properly.

A.1 Number Systems / Conversions

All number systems are based on the same principles. Each system has a set of defined symbols. The layout is the same for each number system, as shown in Figure A.1. The

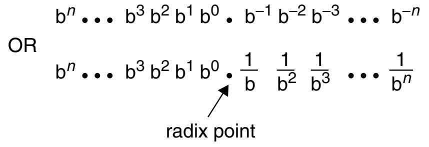

$b^n \dots b^3\, b^2\, b^1\, b^0 \,.\, b^{-1}\, b^{-2}\, b^{-3} \dots b^{-n}$
OR
$b^n \dots b^3\, b^2\, b^1\, b^0 \,.\, \frac{1}{b}\ \frac{1}{b^2}\ \frac{1}{b^3} \dots \frac{1}{b^n}$
radix point

Figure A.1 Basic Layout of a Number System

weight assigned to each position is equal to a multiple of the base. The digit to the immediate left of the radix (decimal point in decimal, binary point in binary, etc.) is always the base raised to the zero power (base^0). Positions to the left increase the power of the exponent and positions to the right decrease the power of the exponent, as shown in Figure A.1.

Decimal, Binary and Hex

A **number system** is made up of an ordered set of symbols. The ordered set of symbols that makes up a number system is called a **number line**. The symbols themselves have little relevance, for any symbols could be used. The order of the symbols is very important, because the order determines the arrangement of the ascending and the descending order. The number of symbols in the system is called the **base** or **radix** of the system. Base 2 has two symbols, base 10 has ten symbols and base 16 has 16 symbols.

Base 10 is called **decimal** because it consists of 10 characters or symbols, as shown in Figure A.2 as a number line. These symbols are the numeric digits 0–9. The symbol or digit 9 is greater than 8, which is greater than 7, and so on, because of their order in the sequence. Any ten symbols could be used to create a decimal number system. All a user has to understand is the actual symbols and their order. All rules of numbers still apply.

The distance between the symbols is always equal to a single unit, and the pattern repeats forever in a cyclical fashion. In other words, the zero (0) symbol follows the nine (9) symbol in ascending order. In the same manner, the nine (9) symbol follows the zero (0) symbol in descending order.

Base 2 is called **binary** because it consists of two symbols. Base 16 is called **hexadecimal** because it consists of 16 symbols. Binary and hex number system symbols are shown in Figure A.3. The name hexadecimal is usually shortened to just **hex**.

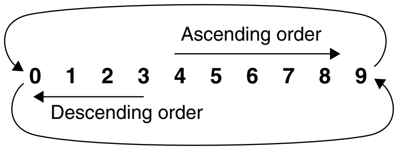


Figure A.2 Symbols of the Decimal Number System

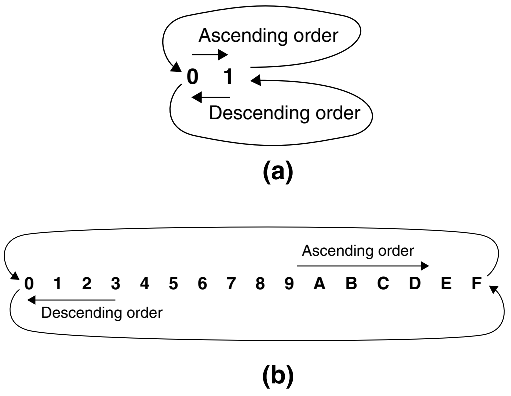


Figure A.3 Symbols of Number Systems: (a) Binary and (b) Hexadecimal.

The distance between symbols, in binary and hex, is always equal to a single unit, and the patterns repeat forever in a cyclical fashion. The zero (0) symbol follows the F symbol in ascending order in hex. In the same manner, the F symbol follows the zero (0) symbol in descending order. Binary seems more complicated because it is so simple.

Hex has a special relationship to binary. It is in essence a superset of binary. Each hex symbol can be represented as four binary bits, as shown in Figure A.4.

Humans depend on the decimal number system. In fact, the metric system is decimal based. All factors of the metric systems are functions of ten. Decimal is efficient and useful for most of our needs.

Decimal	Binary	Hex
0	%0000	$0
1	%0001	$1
2	%0010	$2
3	%0011	$3
4	%0100	$4
5	%0101	$5
6	%0110	$6
7	%0111	$7
8	%1000	$8
9	%1001	$9
10	%1010	$A
11	%1011	$B
12	%1100	$C
13	%1101	$D
14	%1110	$E
15	%1111	$F

Figure A.4 Relationship of Decimal, Binary and Hex Numbers

Conversions Between Number Systems

Since computers are built from thousands of switches, they are inherently binary. Humans have difficulty reading and writing binary because of the number of digits required for large numbers. Therefore, hex is used regularly to display data used by the computer. Three different number systems are used regularly with modern computers: binary, hex and decimal. Therefore, it is necessary to convert numeric data from one system to another.

There have been numerous conversion methods developed to provide a process of converting numbers from one system to another. It would be impractical to attempt a presentation of them all. Many of them are very similar, and most are based on the concepts behind weighting of the digits and modulo arithmetic. Two methods of conversion will be shown that are based on these two principles. Examples are provided for these methods using binary, hex and decimal only; however, they are applicable to any radix or base.

Weight and Add

The weight and add method is based on the natural weighting that occurs with a multidigit number. The symbol positions have a natural weight determined by their position in the number.

The weights increase in size to the left of the radix point and decrease in size to the right of the radix point. Simply assign the power of zero to the radix point. The powers increase to the left and decrease to the right. Thus, the first digit to the left of the radix point has a weight of Base^0, the next digit Base^1, then Base^2 and so on. To the right of the radix point the weights are Base^{-1}, Bas^{e-2}, and so on.

The rightmost digit of any number always has the weight of 1. Each subsequent digit to the left has a weight equal to a power of the base. Equation A.1 shows how the weight is calculated.

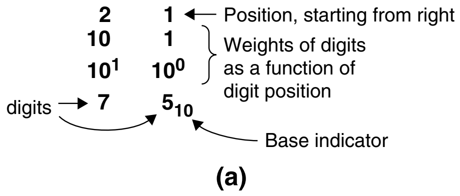
$$\text{Weight} = \text{Base}^{(n-1)} \qquad \text{Equation A.1}$$

where n = the symbol position counting from the digit immediately to the left of the radix point.

Consider the example in Figure A.5. The decimal number 75 is read *seventy-five.* This means it is equal to 7 times 10 plus 5 times 1, where 10 and 1 are the weights of the positions that the 7 and the 5 occupy respectively. Using Equation A.1, the weights for the first position and the second position from the right are calculated as follows:

$\text{Weight } (n=1) = \text{Base}^{(n-1)} = 10^{(1-1)} = 10^0 = 1$

$\text{Weight } (n=2) = \text{Base}^{(n-1)} = 10^{(2-1)} = 10^1 = 10$



2
1
Position, starting from right
10
1
Weights of digits as a function of digit position
10^1
10^0
digits
7
5_10
Base indicator
(a)

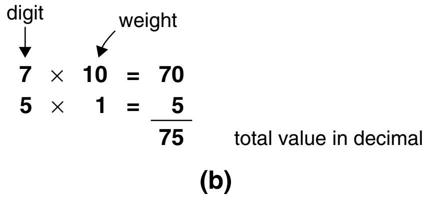
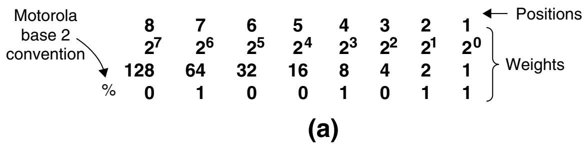


Figure A.5 Weighting of Digits in a Multidigit Decimal Number

The number 75 is represented in binary as %01001011. The weight and add method can be used to show that this binary number is equivalent to the decimal number 75. First all digits of the binary number have to be weighted using Equation A.1, as shown in Figure A.6(a). To complete the conversion, each digit must be multiplied by its weight and then totaled up, as shown in Figure A.6(b).

Seventy-five is represented in hex as $4B. The weight and add method can be used to show that this hex number is equivalent to the decimal number 75. First, weight all digits using Equation A.1, multiply each digit by its weight, and then sum up the weighted digits, as shown in Figure A.7.

The weight and add method can also be used to convert a decimal number to binary. When the conversion goes from decimal to binary, the method really should be called weight and subtract, as will be seen in the following example.



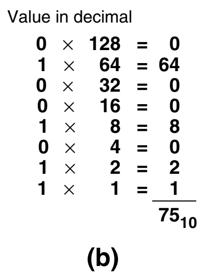


Figure A.6 Conversion from Binary to Decimal Using the Weight and Add Method

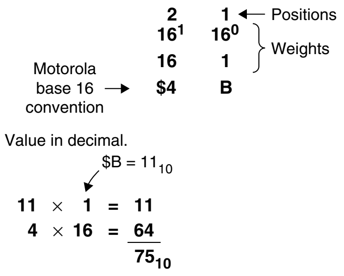


Figure A.7 Conversion from Hex to Decimal Using the Weight and Add Method

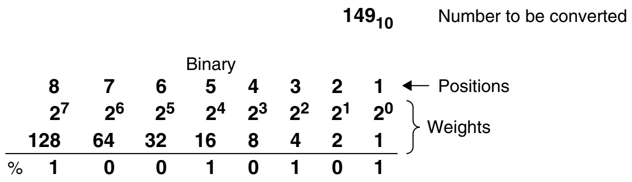

149_10 Number to be converted
Binary
8 7 6 5 4 3 2 1 Positions
2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0 Weights
128 64 32 16 8 4 2 1
% 1 0 0 1 0 1 0 1

Figure A.8 Conversion from Decimal to Binary Using the Weight and Add Method

Figure A.8 shows how this method is applied to convert 149 to binary and hex. First, calculate the weights of the positions for the target number system. Then, starting at the left, determine whether the weight is smaller than the number being converted. For a decimal to binary conversion in Figure A.8, is 128 less than 149? Yes, it is, which means that there is a 128s digit in 149. Now subtract 128 from 149 = 21. Is 64 less than 21? No; therefore, there is a zero 64s digit. Is 32 less than 21? No; therefore, there is a zero 32s digit. Is 16 less than 21? Yes; therefore, there is a 16s digit. Subtract: 21 – 16 = 5, and continue. Is 8 less than 5? No. Is 4 less than 5? Yes. Subtract: 5 – 4 = 1. No 2s digit and there is one 1s digit. Thus, 149 is equal to %10010101.

Most students find this method works best when converting a number from another number system to decimal. Other methods tend to be more straightforward when converting from decimal to binary or hex.

Divide by Base

Modulo division is concerned with the remainder of a division problem rather than the quotient. The **modulus** of a number system is equal to the number of symbols represented; therefore, this method is sometimes called **radix division**. Since decimal consists of 10 symbols, it has a modulus of 10. In the same manner, hex has a modulus of 16 and binary has a modulus of 2.

Since modulo division is concerned with the remainder, it is very useful as a conversion tool. The divide-by-base method uses the base of the target number system as a divisor when calculating the converted number. This method simply performs repeated divisions on the source number. The remainder of each division is the resulting conversion. The divisions repeat until the quotient is zero. The remainder that results

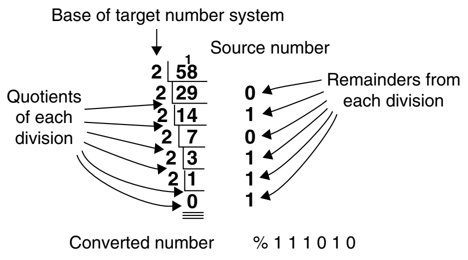

Base of target number system
Source number
Quotients of each division
Remainders from each division
2 | 58
2 | 29
2 | 14
2 | 7
2 | 3
2 | 1
0
0 1 0 1 1 1
Converted number % 1 1 1 0 1 0

Figure A.9 Conversion from Decimal to Binary Using the Divide-by-Base Method

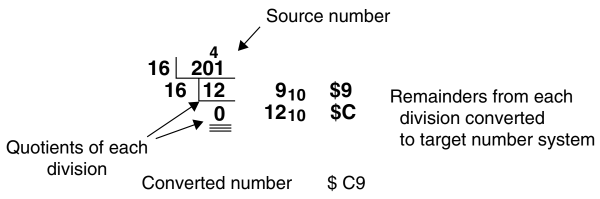

Source number
16 | 201
16 | 12
0
9₁₀ $9
12₁₀ $C
Remainders from each division converted to target number system
Quotients of each division
Converted number $ C9

Figure A.10 Conversion from Decimal to Hex Using the Divide-by-Base Method

from the first division is always the least significant digit of the result, and the remainder from the last division is always the most significant digit of the result.

Figure A.9 illustrates how this process works. The number 58 will be converted to binary using the divide-by-base method. The target number system is base 2, so 2 will be used as the divisor. 58/2 = 29 remainder 0. 29 will be used as the dividend for the next division and zero is the least significant digit of the new number. 29/2 = 14 remainder 1. 14 is the dividend for the next division, and 1 is the second digit of the number in the target number system. This process continues until the quotient is zero. For the number 58, this would require six divisions. Each division produces a remainder; the successive remainders become the digits of the target number. In this example, 58 equals the six-digit binary number %111010.

Figure A.10 illustrates the use of the divide-by-base method for decimal to hex conversion. The number 201 will be converted to hex using the divide-by-base method. The target number system is base 16, so 16 ($10) will be used as the divisor. 201/16 = 12 remainder 9. 12 will be used as the dividend for the next division, and 9 is the least significant digit of the new number. 12/16 = 0 remainder 12. The 0 result indicates that the conversion process is complete and the remainder 12 is the decimal equivalent of the most significant digit of the target number. 12 decimal is equal to $C. In this example, 201 equals the two-digit hex number $C9.

Hex ↔ Binary Conversions

Special methods are not necessary for conversions from binary to hex and from hex to binary. A hexadecimal digit is simply a grouping of four binary bits. Therefore, any binary number can quickly be compressed into the equivalent hex by grouping the

$$\begin{array}{ccccccc} \% & \underbrace{0101} & \underbrace{0011} & \underbrace{1001} & \underbrace{0110} & \underbrace{0010} & \underbrace{1101} \\ \$ & 5 & 3 & 9 & 6 & 2 & D \end{array}$$

(a)

$$\begin{array}{ccccccc} \$ & B & 7 & 0 & F & A & 8 \\ \% & \overbrace{1011} & \overbrace{0111} & \overbrace{0000} & \overbrace{1111} & \overbrace{1010} & \overbrace{1000} \end{array}$$

(b)

Figure A.11 Hex ↔ Binary Conversions

binary digits in sets of four, starting with the least significant digit. The conversion of a hex number to binary requires that each hex digit be expanded to its 4-bit binary equivalent. Figure A.11 illustrates this process for two different number combinations. This process is simple when the binary equivalents of the 16 hex digits are committed to memory.

Leading Zeros

Decimal numbers rarely are shown with leading zeros. For example, the decimal number 342 is made up of three digits: 3 in the hundreds position, 4 in the tens position, and 2 in the ones position. The number 342 is rarely written as 0342 or 00000342 because it is always assumed that all leading digits are zero filled. **Zero filling** is the process of appending zeros to the leftmost digits of a number so that the number has a fixed number of digits. These extra digits are referred to as **leading zeros**.

In some cases, leading zeros are shown. Numeric dates will often include the leading zeros. For example, 12/01/99 is used instead of the simpler 12/1/99 to represent December 1, 1999. Time is also shown with leading zeros, particularly when used to show absolute time in a military or scientific environment. For example, 3 A.M. is commonly referred to as 0300 in military time. Often these rare instances of leading zeros are required because the data is stored on a computer.

Humans have the ability to easily add leading zeros to a number if they are needed, as well as to truncate the leading zeros if they are not needed. However, computers are much more finicky about data types and formats. Inside the memory of a computer, numeric data is always stored using a fixed number of binary digits (bits). Typically, these blocks of data are grouped as bytes. If leading digits are not used, they must be filled with either 1's or 0's because there are never empty bits in a computer memory.

In Figure A.9, the decimal number 58 was converted to binary %111010. Using an unsigned 8-bit convention, this binary number would be zero filled out to eight places. %111010 would become %00111010. Zero filling always works for unsigned numbers, but does not always work for signed numbers. Signed numbers will be discussed in section A.3.

A.2 Arithmetic

Humans typically can perform simple arithmetic calculations in their heads because they have memorized the addition operations through 10 or 12. They don't have to think about how to add, they just recall from memory the result of a particular problem. There is a limit to what they can do off the top of their heads, because there is a limit to what has been committed to memory. The memory limit usually ends with two or more single-digit numbers. When the problem requires several multidigit numbers, another method must be used. In the old days, a scratch pad was used. The numbers were written out and added in columns following addition rules learned in elementary school. Nowadays, calculators or computers are used for most arithmetic. The developers of these electronic tools have designed the devices to perform any arithmetic operation within the limits of the device's memory. To understand how a computer performs arithmetic, it is necessary to understand the rules. The rules apply to any number system.

Decimal Addition

What is the answer to 3 + 4? Well, of course, the answer is 7, but why? Why isn't it 8, 5, or 9? Why isn't it $ or * or #? It is 7 because some time back in second grade, we memorized it and since then have not been required to worry about it. The number line provides us insight into the truth behind 3 + 4 = 7. Figure A.12 shows how the number line can be used to prove that 3 + 4 = 7. Start with the first number and find its position on the number line. Then move down the line to the right the distance of the second symbol. The last location on the number line represents the result of this problem.

It would be nice if no addition problems ever produced a carry, but this is simply not the case. A **carry** is the shown by a rollover on the number line. For example, when 6 + 8 is performed, the result will be 14. Actually, 4 is the result and 1 is the carry to the next column. The next column has a weight one base greater than the previous column. This is shown on the number line in Figure A.13. Multidigit numbers are added using the same rules. The carry is simply added to the next column of numbers, as shown in Figure A.13.

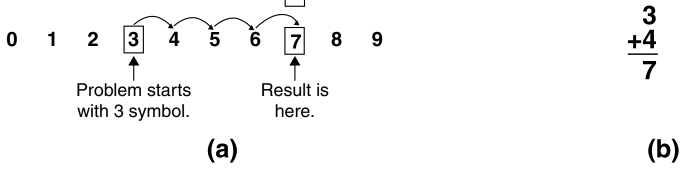


Figure A.12 Simple Addition Using a Number Line

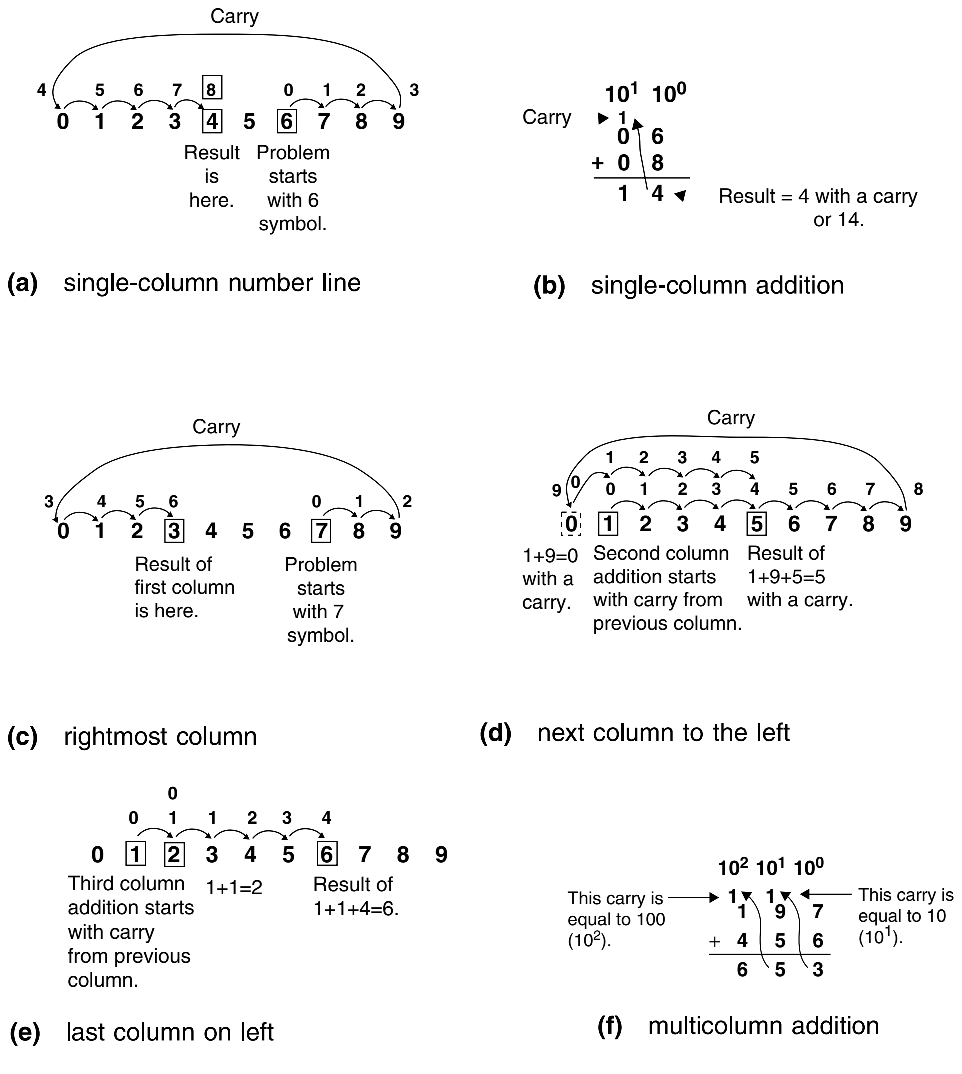

Carry
Result is here.
Problem starts with 6 symbol.
(a) single-column number line
Carry
Result = 4 with a carry or 14.
(b) single-column addition
Carry
Result of first column is here.
Problem starts with 7 symbol.
(c) rightmost column
Carry
1+9=0 with a carry.
Second column addition starts with carry from previous column.
Result of 1+9+5=5 with a carry.
(d) next column to the left
Third column addition starts with carry from previous column.
1+1=2
Result of 1+1+4=6.
(e) last column on left
This carry is equal to 100 (10²).
This carry is equal to 10 (10¹).
(f) multicolumn addition

Figure A.13 Addition with Carry Using a Number Line

Machines do arithmetic in binary and must convert the decimal input and output to and from binary/hex. The concepts surrounding the conversion of numbers from decimal to binary and then back to decimal were presented earlier in the chapter.

Binary Addition

Binary addition uses the same rules as a decimal addition problem. Consider the same problem of decimal addition, 3 + 4. For the computer to add these numbers, they first have to be converted to binary. Using the divide-by-base method, each is quickly converted to 4-bit binary, as shown in Figure A.14.

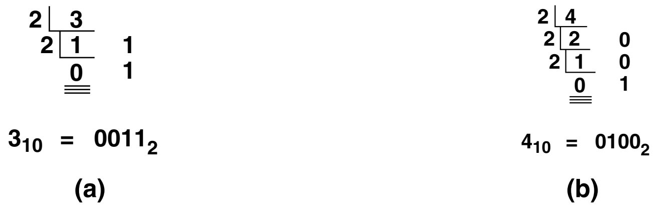


Figure A.14 Conversion of Decimal Numbers to 4-Bit Binary

First	Second	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Figure A.15 2-Bit Binary Combinations

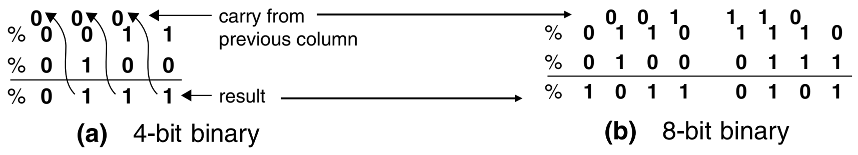


Figure A.16 Binary Additions

The number line is not very helpful with binary addition because there are only two symbols. Since there are only two symbols, there are only four combinations of bits that will ever occur; therefore, these operations can be easily summarized in a table. Figure A.15 contains all possible combinations.

Applying the information from this table to the 4-bit binary addition problem, the result is quickly found. Figure A.15 shows how 1 + 0 = 1 with 0 carry, and 0 + 1 = 1 with 0 carry, and so on. Consider another 4-bit example that shows carry conditions. Figure A.16 shows a simple 8-bit binary addition.

Hex Addition

Addition in binary can be very tedious. Although machines do addition at the binary level, the human interface is typically in hex. Earlier it was explained that a hex digit is just a grouping of four binary bits. Any hex or binary number can be quickly converted to the other form. The study of computers will require the knowledge of hex for numeric operations and the human interface; however, inside the computer the data is actually processed as binary.

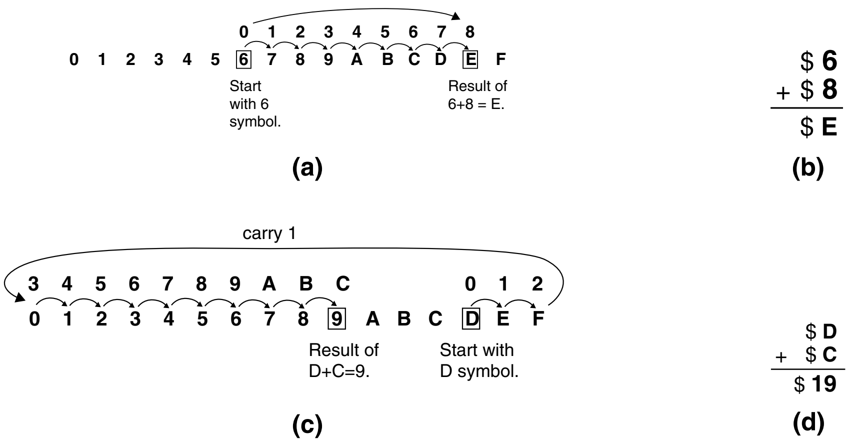


Figure A.17 Single-Digit Hex Addition

The number-line concepts and rules from the presentation on decimal addition now directly apply. Consider the single-digit example that illustrates the number-line principles applied to an addition problem consisting of hex digits. Figure A.17 shows the hex number line and addition of $6 and $8. Well, everybody knows that 6 + 8 = 14. Interesting enough, 14 is equal to $E, therefore $6 + $8 = $E. The second example in Figure A.17 shows the result of a carry. $D + $C = $19. The number line is very useful in illustrating this result.

Multidigit numbers are added using the same rules. The carry is simply added to the next column of numbers, as shown in Figure A.18. $3E9 + $78C = $B75.

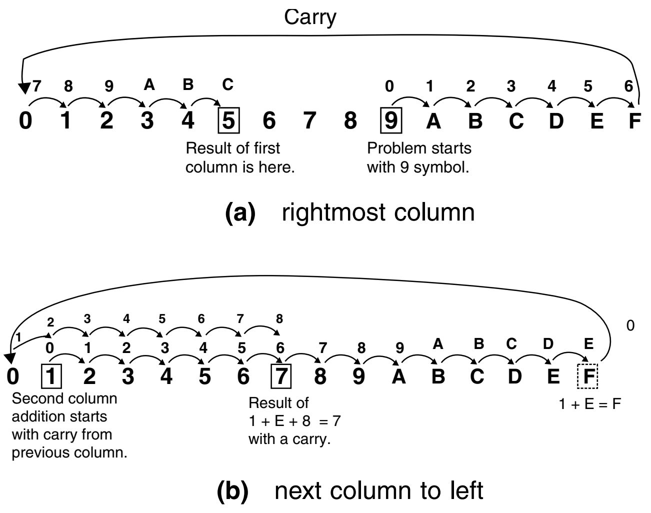
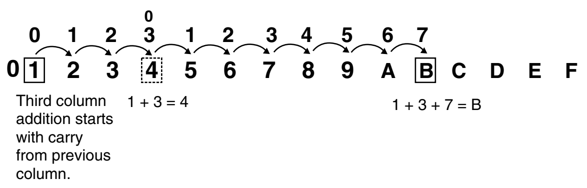
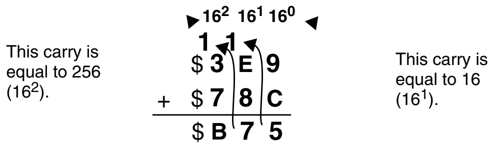

Carry
Result of first column is here.
Problem starts with 9 symbol.
(a) rightmost column
Second column addition starts with carry from previous column.
Result of 1 + E + 8 = 7 with a carry.
1 + E = F
(b) next column to left

Figure A.18 Multicolumn Hex Addition with Carry Using a Number Line (*continues*)


0
0 1 2 3 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7 8 9 A B C D E F
Third column addition starts with carry from previous column.
1 + 3 = 4
1 + 3 + 7 = B

(c) last column on left


16² 16¹ 16⁰
1 1
$ 3 E 9
+ $ 7 8 C
$ B 7 5
This carry is equal to 256 (16²).
This carry is equal to 16 (16¹).

(d) multicolumn addition

Figure A.18 Multicolumn Hex Addition with Carry Using a Number Line (*continued*)

A.3 Signed Numbers (2′s Complement Number System)

Up to this point, all numbers presented have been positive numbers. It would be nice if all numbers were positive in real life, but this is just not the case. The decimal number system uses a pair of symbols to indicate polarity. The plus sign "+" indicates a positive number, and the minus sign "–" indicates that a number is negative. Although it may not seem like it, the number zero is always considered a positive number. For this reason, **negative numbers** can be defined as any number less than zero, and **positive numbers** are any numbers that are not negative.

This extra symbol (+ or –) occupies another digit (position) in the number. If the numbers are all positive, the leading symbol is rarely shown. If no sign is shown in decimal, the number is presumed to be a positive number. Figure A.19 illustrates various positive and negative decimal numbers.

The binary number system does not have the luxury of being able to use additional symbols to indicate sign. Computers only understand 1's and 0's; therefore, signed numbers must somehow utilize these symbols to represent the sign of a number.

There are several methods used to incorporate signed numbers into the binary number system. Signed number methods use a single bit to indicate sign. The **sign bit** is the leftmost or most significant bit, similar to the sign symbol used in

Positive Numbers	Negative Numbers
465	– 465
+ 109	– 109
0	– 3

Figure A.19 Examples of Positive and Negative Decimal Numbers

decimal. In binary, the "1" symbol indicates negative and the "0" symbol indicates positive. As has been discussed, there cannot be empty bits in a binary number; therefore, all signed numbers have a 1 or a 0 in the most significant bit position to indicate negative or positive.

The **2's complement number system** is one of many ways of implementing signed numbers with binary data. This method is used by all major computer systems for signed arithmetic processes and storage of signed data. The advantage in using the 2's complement number system is observed in the arithmetic operations performed by a computer. All computers have the need to perform four basic arithmetic operations: addition, multiplication, subtraction, and division. Each arithmetic operation is performed by an arithmetic logic unit (ALU), which for arithmetic operations is in essence a smart binary adder.

Multiplication, for example, is simply repeated addition. The multiplication problem 3 × 4 can be performed by adding 4 + 4 + 4 to result in the correct answer, 12. Subtraction can be performed by an adder if the sign of the second number is changed prior to the addition operation. For example, 5 – 2 = 5 + (–2) = 3. The 2's complement number system allows for this negation operation; thus subtraction is simply a special case of addition. Since division is repeated subtraction just as multiplication is repeated addition, division can also be done in a binary adder.

To understand how a computer performs arithmetic operations, an understanding of the 2's complement number system must be gained first. All positive numbers within the 2's complement system are interpreted in the same manner as unsigned numbers (often this is called the true magnitude). %0111 is still 7, and %00110101 is still equal to 53. Note that in each of these examples the sign bit is zero, indicating positive numbers. Negative numbers are represented in a form unique to the 2's complement form.

2's Complement Negation

All negative numbers within the 2's complement number system can be derived from their positive equivalents. **Negation** is the process of calculating the negative equivalent of a number. By definition, negation is the result of subtracting a number from zero. This rule works in any number system. Since 2's complement numbers can be represented by hex or binary symbols, they can be easily subtracted from zero, as shown in Figure A.20.

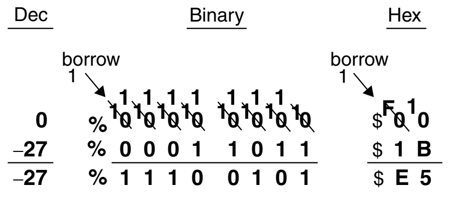


Figure A.20 Negation by Subtracting from Zero.

```
  % 0 0 0 0  0 1 0 1   +5
  % 1 1 1 1  1 0 1 0   1's complement
+   0 0 0 0  0 0 0 1   add 1
  ------------------
  % 1 1 1 1  1 0 1 1   2's complement
                       code for –5
```

Figure A.21 2's Complement Negation Process

```
  % 0 1 0 0  0 1 1 1   +71
  % 1 0 1 1  1 0 0 0   1's complement
+                  1   add 1
  ------------------
  % 1 0 1 1  1 0 0 1   –71
  % 0 1 0 0  0 1 1 0   1's complement
+                  1   add 1
  ------------------
  % 0 1 0 0  0 1 1 1   +71
```

Figure A.22 Reversibility of 2's Complement Negation

NOTE: Although the term "2's complement" technically applies only to the binary number system, the term "2's complement" will be used here to refer to negation in any number system. When a decimal number is negated, it is called the 10's complement, and when a hex number is negated, it is technically called the 16's complement.

The 2's complement negation is done by first representing the positive number in the proper number of bits, inverting all of the bits and then adding one. The process of inverting all of the bits is referred to as the **1's complement**. Figure A.21 illustrates the application of 2's complement negation to the positive number 5.

Another great advantage of the 2's complement number system is that the process is reversible. The 2's complement negation process can be applied to any positive number to find the negative equivalent and to any negative number to find its positive equivalent. Figure A.22 illustrates this reversibility.

2's Complement Negation Shortcut

Since the 2's complement negation operation requires two steps, it can be somewhat tedious to apply. A shortcut has been discovered that produces the same result, yet only

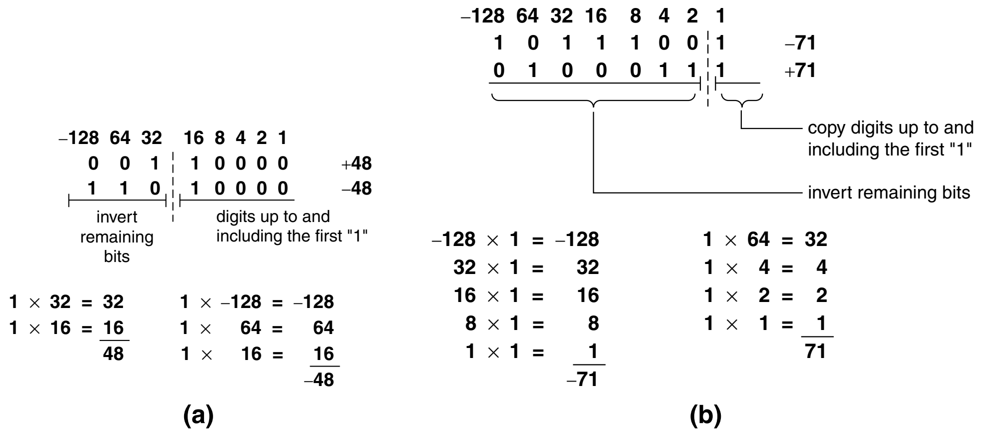


Figure A.23 2's Complement Negation Shortcut

Decimal	2's Complement Form	Decimal	2's Complement Form
0	00000000	-1	11111111
1	00000001	-2	11111110
2	00000010	-3	11111101
38	00100110	-39	11011001
75	01001011	-87	10101001
112	01101110	-100	10011100
120	01111000	-120	10001000
126	01111110	-127	10000001
127	01111111	-128	10000000

Figure A.24 2's Complement Decimal Equivalents

requires a single step. This process consists of starting with the original binary number that is to be negated. Simply begin writing down all the digits starting from the right up to and including the first "1" digit. Then invert the remaining bits. Figure A.23 illustrates how this works on a 4-bit number and on an 8-bit number.

Weighting Using 2's Complement Signed Numbers

The only thing that has changed when converting 2's complement numbers to decimal is what the numbers might represent in decimal. Figure A.24 shows several decimal numbers and their equivalent values in the 2's complement form.

The principles of weighting still apply to 2's complement sign numbers; however, the most significant weight always carries the negative sign. For example, the weights of a 4-bit binary number would be 1, 2, 4, and –8, as shown in Figure A.25. The weights of an 8-digit binary number then would be 1, 2, 4, 8, 16, 32, 64, and –128.

$$\begin{array}{rrrrl} -8 & 4 & 2 & 1 & \text{weights} \\ 0 & 1 & 0 & 1 & +5 \end{array}$$

$$\begin{array}{rcrcr} 0 & \times & -8 & = & 0 \\ +1 & \times & 4 & = & 4 \\ +0 & \times & 2 & = & 0 \\ +1 & \times & 1 & = & \underline{1} \\ & & & & 5 \end{array}$$

(a)

$$\begin{array}{rrrrrrrrl} -128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 & \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & -78 \end{array}$$

$$\begin{array}{lrcrcr} & -128 & \times & 1 & = & -128 \\ & 64 & \times & 0 & = & 0 \\ & 32 & \times & 1 & = & 32 \\ & 16 & \times & 1 & = & 16 \\ & 0 & \times & 8 & = & 0 \\ & 0 & \times & 4 & = & 0 \\ & 1 & \times & 2 & = & 2 \\ + & 0 & \times & 1 & = & \underline{0} \\ & & & & & -78 \end{array}$$

(b)

Figure A.25 Binary Weights of 2's Complement Numbers

2 digit hex	8-bit Binary 2's complement form	unsigned decimal	signed decimal
\$07	% 0 0 0 0 0 1 1 1	+ 7	+7
\$FB	% 1 1 1 1 1 0 1 1	+ 251	− 5
\$02	% 0 0 0 0 0 0 1 0	+ 258	2

Figure A.26 Relationship of Signed and Unsigned Numbers

Arithmetic Using 2's Complement Numbers

Since a number in the 2's complement form still is a binary set of digits, all the addition rules presented earlier in this chapter still apply. Figure A.26 illustrates this different relationship to the decimal equivalents. Since the largest unsigned number that can be represented in eight bits is 255 and the modulus of an 8-bit system is $2^8 = 256$, then the unsigned value of 258 is out of range. The range of unsigned 8-bit numbers is 0–255, and the range of signed 8-bit numbers is –128 to +127.

A.4 Binary-Coded Decimal

Binary-coded decimal (BCD) is a means of encoding single decimal digits into a 4-bit code. It is rarely used on computers because BCD numbers are very inefficient. Since only 10 of the 16 possible codes are used in each 4-bit nibble, storage and processing capacity is wasted. The BCD codes are shown in Figure A.27.

BCD also requires the hex adder to perform decimal adjustments after each arithmetic operation. The adder is designed for hex digits; therefore, it will produce the hex values A–F, which are invalid BCD results. Consider the example shown in Figure A.28. When 3 and 9 are added, the result should be 12. Using 8-bit BCD for each of the numbers and a hex adder, the result is \$0C, not 12. The hex number line illustrates why this occurs. There are six unused hex symbols in BCD. These symbols must be skipped each time an addition is performed.

Decimal	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Figure A.27 BCD Codes

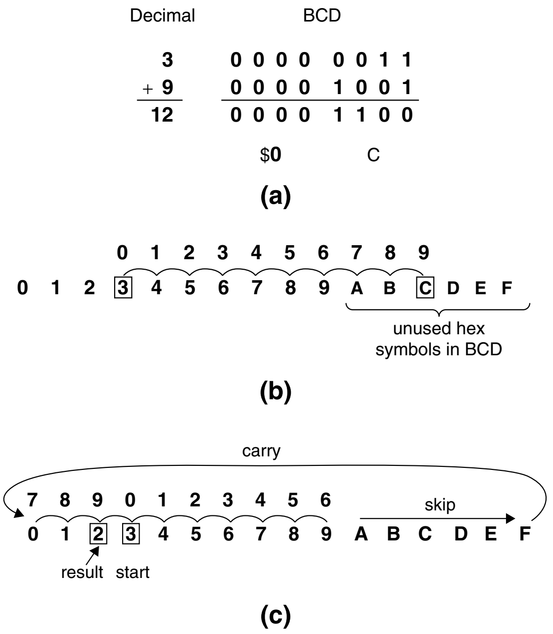


Figure A.28 Decimal Adjusting a BCD Result

The process of skipping the unused hex symbols is called **decimal adjusting** or **BCD correction**. Two specific rules can be applied to accomplish the proper decimal adjustment after BCD addition is performed in a hex adder:

Rule 1: Is the result greater than 9?
Rule 2: Did the result cause a carry?

If the answer to either of these questions is yes, then an adjustment must be made to the hex result. The adjustment is made by adding 6 to the original hex result, as shown in Figure A.29. In Figure A.29(a), the hex adder comes up with $6B. Since $B is greater than 9, Rule 1 applies, and the adjustment of 6 must be added to that column to achieve the proper result of 71. In Figure A.29(b), the hex adder comes up with $80, which

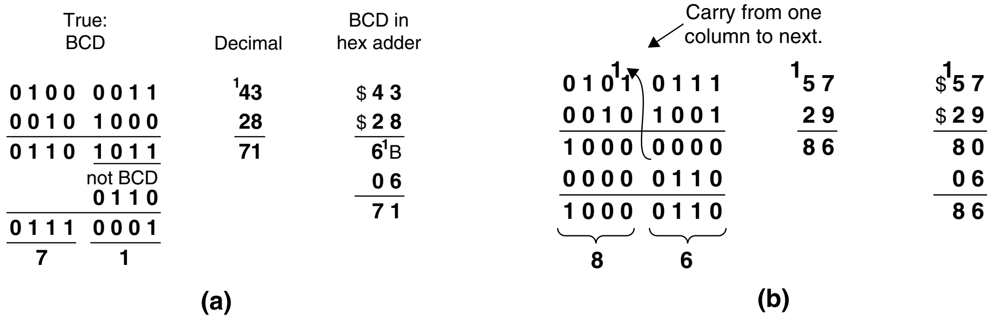


Figure A.29 BCD Correction

produces a carry from the first column to the next. This carry satisfies Rule 2, so 6 must be added to achieve the proper result of 86.

A.5 Numeric Notation

When performing mathematical calculations of any type, it is often more important to have the proper units for what is being calculated than the absolute accuracy of the calculated value. This is particularly true when scientific or engineering notation is used. Scientific and engineering notation represent values as several significant digits and a multiplier. The multiplier is usually a power of ten. The number of significant digits represents the number of digits to the left of the multiplier. Several examples are shown in Figure A.30.

Scientific notation uses any integer power of ten, and the value to the left of the multiplier must be greater than or equal to one and less than 10. Engineering notation uses powers of ten that must be an integer multiple of three (i.e., 3, 6, 9, –3, –6, –15, etc.). The value to the left of the multiplier must be greater than or equal to one and less than 1,000.

The base 10 multipliers work well in the decimal number system. The decimal point is easily shifted to the right or to the left of any decimal number with the use of the multipliers. The engineering notation can conveniently be shown using metric notation. Figure A.31 shows some of the typical metric notations in relation to the power of 10.

Computers don't use decimal as their primary number system; rather, they use the binary system. Therefore, it makes sense that if the principles of scientific and engineering notation are applied to the computer environment, the multipliers should

Number	Significant Digits	Scientific Notation	Engineering Notation	Volts Using Metric Units
27,948	3	2.79×10^4	27.9×10^3	27.9 × kV
0.194501	3	1.95×10^{-1}	195×10^{-3}	195 × mV
52,117,654	4	5.212×10^7	52.12×10^6	52.12 × MV

Figure A.30 Examples of Scientific and Engineering Notation

10^X	Decimal	Metric Notation	
10^9	1 Billion	G	Giga
10^6	1 Million	M	Mega
10^3	1 Thousand	k	kilo
10^{-3}	1 Thousandth	m	milli
10^{-6}	1 Millionth	µ	micro
10^{-9}	1 Billionth	n	nano
10^{-12}	1 Trillionth	p	pico

Figure A.31 Notation Using Powers of Ten

2^X	Decimal	Notation
2^{10}	1,024	K
2^{20}	1,048,576	M
2^{30}	1,073,741,824	G
2^{40}	1,099,511,627,776	T

Figure A.32 Notation Using Powers of Two

be powers of two. The number of possible combination of eight bits can be expressed as $1 \times 2^8 = 256$. The number of memory locations that can be addressed by a 16-bit address bus is $1 \times 2^{16} = 65{,}536$. Whenever this technique is used to refer to memory capacity, the "powers of two" notation is used. The engineering notation uses integer powers of two that are multiples of 10 (i.e., 2^{10}, 2^{20}, etc.) The 65,536 unique addresses from the 16-address bit system can then be written as $2^{16} = 2^6 \times 2^{10} = 64 \times 2^{10}$. It is common practice to replace the powers of two with a single letter, as shown in Figure A.32. Thus, the common memory convention of megabytes is really 2^{20} bytes, or Mbytes. 64 Mbytes of memory is actually 64×2^{20} bytes = 67,108,864 bytes.

A.6 Multiprecision Arithmetic

The HC11 has instructions that independently perform 8-bit and 16-bit addition and subtraction operations, as is shown in chapter 3. If the desire is to perform addition and subtraction on larger data words, special instructions must be used that are nested into loops. For example, the HC11 could be used to perform 32-bit arithmetic using these special instructions.

Consider the example shown in Figure A.33. The objective is to add two 32-bit values using the 8-bit addition operation available on the HC11. Each 8-bit pair can cause a full carry that must be considered when performing the addition on the next most significant pair. This carry must be cleared to start the operation properly. In this example, \$98 is added to \$0F, which results in \$A7 and a 0 carry. The next two are then added, \$B5 + \$70 + the zero carry. This results in \$25 with a full carry. Then \$EA is added to \$69 with the carry from the previous operation. This results in \$54 with another full carry. Finally, the last two are added, \$12 + \$5C, which results in \$6F and a zero carry.

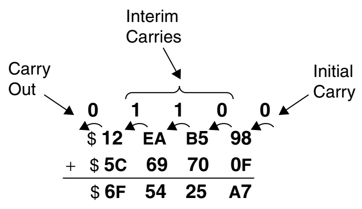
The code to accomplish this task is quite straightforward. It consists of a loop that is executed four times (four bytes = 32 bits) and the addition and storage instructions. In this particular example, the X register is the address of the most significant byte of the first number. The first number is expected to occupy (X) through (X+3), the second number (X+4) through (X+7) and the result (X+8) through (X+11).

Add and Subtract with Carry Instructions

ADCA and ADCB function similar to the ADDA and ADDB instructions, except they add with carry, as shown in Figure A.34. An 8-bit operand is added to the contents of

```

                          * This subroutine is designed to perform an addition
                          * of two 32-bit words. The X register contains the address
                          * of the data block, where the first four bytes are occupied
                          * by the first word, next four by the second word, and the
                          * last four by the result.
0100
                                 ORG     $0100
0100  0c
                                 CLC              ; Clear the initial carry flag
0101  37
0102  c6  04                     PSHB             ; Save current B
                                 LDAB    #4       ; init loop counter
0104  a6  00
0106  a9  04              AGAIN  LDAA    $0,X     ; Get first byte
0108  a7  08                     ADCA    $4,X     ; Add with carry
                                 STAA    $8,X     ; Store first result
010a  5a
010b  26  f7                     DECB             ; One add is done
                                 BNE     AGAIN    ; Do it again
010d  33
010e  39                         PULB             ; Restore B
                                 RTS
```

(a) Multiprecision Addition Code



(b) Multiprecision Addition Process

Figure A.33 Multiprecision Addition

the A or B accumulator along with the contents of the C flag of the CCR. These instructions operate in the IMM, DIR, EXT, INDX and INDY addressing modes. The ADCA and ADCB instructions affect the H, N, Z, V and C bits of the CCR register.

SBCA and SBCB function similar to the SUBA and SUBB instructions, except they subtract with carry. An 8-bit operand is subtracted from the contents of the A or B accumulator along with the contents of the C flag of the CCR. These instructions

Mnemonic	Description	Function	IMM	DIR	EXT	INDX	INDY	INH	REL	S	X	H	I	N	Z	V	C
ADCA ADCB	Add contents of memory to Acc with carry	A+(M)+C ⇒ A B+(M)+C ⇒ B	X	X	X	X	X	-	-	-	-	↕	-	↕	↕	↕	↕
SBCA SBCB	Subtract contents of memory from Acc with carry	A–(M)–C ⇒ A B–(M)–C ⇒ B	X	X	X	X	X	-	-	-	-	-	-	↕	Ã	Ã	↕

Figure A.34 Add and Subtract with Carry Instructions

operate in the IMM, DIR, EXT, INDX and INDY addressing modes, and they affect the N, Z, V and C bits of the CCR.

Placement of Radix Point

The result of the IDIV instruction assumes that the result is to the left of the radix point. Since the FDIV is calculating the result of a fraction that is less than one, it is assumed that the result is to the right of the radix point. The IDIV and FDIV instructions can be used in conjunction to determine a 16-bit value for each side of the radix point, which results in 32 bits of binary-weighted information for each calculation. Consider the example in Figure A.35.

The decimal value 9.68 can be shown as the fraction 242/25. The value 9 is actually 225/25 and the decimal value 0.68 can be calculated from 17/25. Both IDIV and FDIV would have to be used to convert the fraction 242/25 into the binary-weighted equivalent. How this is done is shown in Figure A.35. The first step is to load AccD with the numerator 242 ($00F2) and the X register with the denominator 25 ($0019). Then perform the integer divide using IDIV, which results in $0009 (stored in X) and the remainder $0011 (stored in D). The result must be saved to another location to allow the X register to be used for the next calculation.

Next load the X register again with the denominator 25 ($0019). With the remainder from the previous operation still in AccD, FDIV can be executed. Remember, FDIV

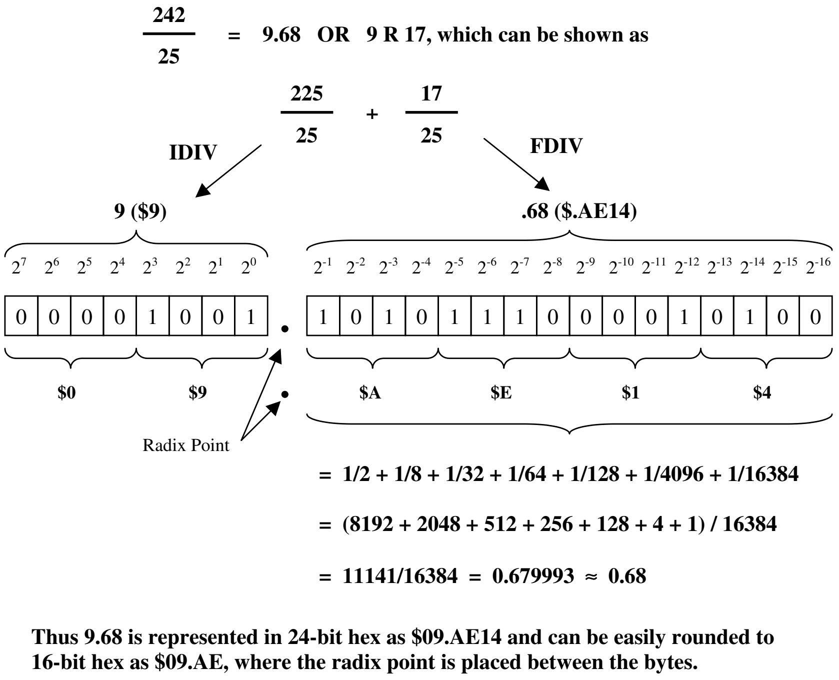


Figure A.35 Placement of the Radix Point

expects the numerator to be less than the denominator so that it can calculate the value to the right of the radix point. When FDIV is executed, the result will be $AE14 (stored in X) and the remainder $000C (stored in D). The remainder represents the difference between 0.680000 and the calculated binary-weighted equivalent 0.679993. Additional precision could be achieved by repeatedly performing the FDIV using the remainder from each preceding operation.

a p p e n d i x B

Booting from EEPROM

The BUFFALO monitor program and the EVBU were designed to allow simple access to programs loaded into the EEPROM. The programs loaded into the EEPROM can be used as the startup software after reset. Booting directly from the EEPROM requires several steps because of the many functions that the Buffalo Monitor program performs during the boot process (i.e., setting the baud rate, initializing interrupts, etc.). To properly get the EVBU to come to life running your program, two things must be accomplished:

1. The program must be modified to include some initialization code.
2. A jumper must be changed on the EVBU board.

Modifying the EEPROM Program to Include Initialization Code

In order to get the EVBU to boot directly to the program in EEPROM, several lines of code must be copied directly from the Buffalo Monitor program into the EEPROM program. These lines of code are needed to do the minimum initialization of the EVBU board to make it function.

The following entries are necessary to boot from the EEPROM program.

A. Set up the OPTION register.

```
LDAA   #$93       ;Power up A/D, Enable Startup Delay, COP rate 1.049 s
STAA   OPTION     ($1039 for BUFFALO 3.4)
```

B. Initialize the user stack space.

```
LDS    #USTACK    ($0041 for BUFFALO 3.4)
```

C. Initialize the SCI (RS-232 serial communications)

```
CLR    IODEV      ($00A5 for BUFFALO 3.4)
JSR    INIT       ($E378 for BUFFALO 3.4)
```

The following entries may be necessary if these features are being used in the EEPROM program.

D. Initialize the Interrupt Jump Table (only necessary if interrupts are being used).

```
JSR    VECINIT     ($E357 for BUFFALO 3.4)
```

E. Initialize the Timer Prescaler (only necessary if prescaler other than default is being used).

```
LDAA   #$0x        ;x= timer prescale select bits
STAA   TMSK2       ($1025 for BUFFALO 3.4)
```

F. Un-Protect the EEPROM (only necessary if data is written to EEPROM during execution).

```
LDAA   #$0x        ;x = Block protect bits
STAA   BPROT       ($1035 for BUFFALO 3.4)
```

Download this modified version of the program to the EEPROM.

See page B-4 of the Buffalo monitor listing for further information.

appendix C

Loading Programs Directly into the EEPROM

The following procedure is the process that can be used to successfully load a user program into the EEPROM. This process includes reducing the baud rate of the serial communications interface. This change is necessary to allow sufficient time (at least 10 ms) for the byte to be burned into the EEPROM location.

NOTE: The procedure shown here uses BUFFALO 3.4 and PROCOMM. The procedure can be modified for use with other versions of BUFFALO and other terminal emulators.

Procedure

1. **Change program start address**. Assemble the program using addresses in the EEPROM range ($B600–$B7FF). The address of the first instruction must be $B600. Some other minor changes may be necessary to get the program to run from the EEPROM. Beware that variable space must remain in the RAM; thus, additional ORG statements may be necessary.

   For example: `ORG $B600.`

2. **Erase contents of EEPROM**. Perform a bulk erase of the EEPROM area by executing BUFFALO command "BULK" at the EVBU > prompt.

   For example: `> BULK`

3. **Change EVBU Baud rate to 300. Change the contents of the BAUD rate register** ($102B) to talk 300 baud by using the MM command. (Default baud rate is 9600, which is selected by $30 in this register; 300 baud is selected by $35 in this register.)

   The screen will freeze when this is done because PROCOMM is still trying to talk at 9600 baud.

   For example:
   ```
   MM 02B
    >102B 30 35 <CR>
   ```

4. **Change PROCOMM to talk 300 baud**. In PROCOMM, enter the baud rate screen (ALT-P) and select 7 (300, N, 8, 1). Do not save these settings to disk. Press ESC to exit this screen.
5. **Confirm 300 baud**. Hit the enter key to confirm communications between PROCOMM and EVBU. This may require more than our <CR>.
6. **Load and verify the program**. Go through the normal process of loading a program into memory (i.e., LOAD T... NOTE: The transfer rate is now 300 bits per second, so it will appear much slower than normal), but load a program that has been assembled to execute from the EEPROM. Execute a memory dump on the EEPROM to verify that the program was loaded properly.
7. **Reset PROCOMM baud rate to 9600**. Change PROCOMM to talk 9600 baud by entering the baud rate screen (ALT-P) and selecting 11 (9600, N, 8, 1). Do not save these settings to disk. Press ESC to exit this screen.
8. **Reset EVBU baud rate to 9600**. Reset EVBU to get the baud rate set back to 9600. At this point the program can be run out of EEPROM by using the Go command and the appropriate EEPROM start address.

   For example: `>G B600`

Changing the EVBU Jumper

Once the EEPROM has been loaded with a bootable program, the EVBU must be told to boot to the program in the EEPROM. The EVBU has a dedicated hardware connection between PE0/AN0 and the boot jumper (J2). This jumper tells the BUFFALO monitor program to boot to BUFFALO or to boot to the program loaded into the EEPROM.

The Jumper (J2) on the EVBU needs to be moved from the default position (pins 2–3) to pins (1–2).

Refer to section 2.3.2 Program Execution Select Header of the EVBU User's Manual for detailed information.

appendix D

Acronym List

ABA	Add AccB to AccA
ABX	Add AccB to Index Register X
ABY	Add AccB to Index Register Y
AccA	Accumulator A
AccB	Accumulator B
AccD	Accumulator D
ADDA	Add to AccA
ADDB	Add to AccB
ADDD	Add to AccD
ALU	Arithmetic Logic Unit
ANDA	Logical AND AccA with Byte from Memory
ANDB	Logical AND AccB with Byte from Memory
ASCII	American National Standard Code for Information Interchange
ASR	Arithmetic Shift Right Byte in Memory
ASRA	Arithmetic Shift Right AccA
ASRB	Arithmetic Shift Right AccB
BCC	Branch If Carry Clear
BCD	Binary Coded Decimal
BCS	Branch If Carry Set
BEQ	Branch If Equal (to Zero)

BGE	Branch If Greater Than or Equal (Signed)
BGT	Branch If Greater Than (Signed)
BHI	Branch If Higher (Unsigned)
BHS	Branch If Higher or Same (Unsigned)
BLE	Branch If less Than or Equal (Signed)
BLO	Branch If Lower (Unsigned)
BLS	Branch If Lower or Same (Unsigned)
BLT	Branch If LessThan (Signed)
BMI	Branch If Minus
BNE	Branch If Not Equal (to Zero)
BPL	Branch If Plus
BRA	Branch Always
BRN	Branch Never
BSR	Branch to Subroutine
BUFFALO	Bit User Fast Friendly Aid to Logical Operations
BVC	Branch If Overflow Clear
BVS	Branch If Overflow Set
C	C Programming Language
CBA	Compare AccB to AccA
CCR	Condition Code Register
CD-ROM	Compact Disk – Read Only Memory
CLC	Clear Carry Flag in CCR
CLI	Clear Interrupt Mask Bit in CCR
CLR	Clear Byte in Memory
CLRA	Clear Accumulator A
CLRB	Clear Accumulator B
CLV	Clear Overflow Flag in CCR
CMPA	Compare Memory to AccA
CMPB	Compare Memory to AccB
COM	Complement Byte in Memory (Logical NOT)

COMA	Complement AccA (Logical NOT)
COMB	Complement AccB (Logical NOT)
CPD	Compare Memory to AccD
CPU	Central Processing Unit
CPX	Compare Memory to AccX
CPY	Compare Memory to AccY
DA	Destination Address
DAA	Decimal Adjust AccA
DEC	Decrement a Byte in Memory
DECA	Decrement AccA
DECB	Decrement AccB
DES	Decrement Stack Pointer
DEX	Decrement Index Register X
DEY	Decrement Index Register Y
DIR	Direct Addressing Mode on the HC11
EA	Effective Address
EEPROM	Electrically Erasable Programmable Read Only Memory
EORA	Logical XOR AccA with Byte from Memory
EORB	Logical XOR AccB with Byte from Memory
EPROM	Erasable Programmable Read Only Memory
EQU	Equate
EVBU	Motorola M68HC11 Universal Evaluation Board
EXT	Extended Addressing Mode on the HC11
FCB	Form Constant Byte
FCC	Form Constant Character
FDIV	Fractional Division
ff	Indexed Mode 8-bit Offest
IDIV	Integer Division
IMM	Immediate Addressing Mode on the HC11
INC	Increment a Byte in Memory

INCA	Increment AccA
INCB	Increment AccB
IND	Index Addressing Mode on the HC11
INDX	Index Addressing Mode that uses the X index register
INDY	Index Addressing Mode that uses the Y index register
INH	Inherent Addressing Mode on the HC11
INS	Increment Stack Pointer
INX	Increment Index Register X
INY	Increment Index Register Y
I/O	Input/Output
IR	Instruction Register
JMP	Jump to Memory Location
JSR	Jump to Subroutine
LDAA	Load Accumulator A
LDAB	Load Accumulator B
LDD	Load Accumulator D
LDS	Load Stack Pointer
LDX	Load Index Register X
LDY	Load Index Register Y
LIFO	Last In First Out
LSB	Least Significant Bit
LSL	Logical Shift Left Byte in Memory
LSLA	Logical Shift Left AccA
LSLB	Logical Shift Left AccB
LSLD	Logical Shift Left AccD
LSR	Logical Shift Right Byte in Memory
LSRA	Logical Shift Right AccA
LSRB	Logical Shift Right AccB
LSRD	Logical Shift Right AccD
MAR	Memory Address Register

MDR	Memory Data Register
MODA/B	Mode A/B (pins on HC11)
MSB	Most Significant Bit
MUL	Multiply
NEG	Negate a Byte in Memory
NEGA	Negate AccA
NEGB	Negate AccB
NOP	No Operation
NRZ	Nonreturn-to-zero
ORAA	Logical OR AccA with Byte from Memory
ORAB	Logical OR AccB with Byte from Memory
ORG	Originate
PC	Program Counter
PH1	Phase 1 of internal machine clock
PH2	Phase 2 of internal machine clock
PROM	Programmable Read Only Memory
PSHA	Push Byte to Stack from AccA
PSHB	Push Byte to Stack from AccB
PSHX	Push Two Bytes to Stack from Index Register X
PSHY	Push Two Bytes to Stack from Index Register Y
PULA	Pull Byte from Stack to AccA
PULB	Pull Byte from Stack to AccB
PULX	Pull Two Bytes from Stack to Index Register X
PULY	Pull Two Bytes from Stack to Index Register Y
RAM	Random Access Memory
REL	Relative Addressing Mode on the HC11
RMB	Reserve Memory Byte
ROL	Rotate Left Byte in Memory
ROLA	Rotate Left AccA
ROLB	Rotate Left AccB

ROM	Read Only Memory
ROR	Rotate Right Byte in Memory
RORA	Rotate Right AccA
RORB	Rotate Right AccB
rr	Sign Relative Mode Offset
RTI	Real Time Interrupt
RTS	Return from Subroutine
SBA	Subtract AccB from AccA
SCI	Serial Communications Interface
SEC	Set Carry Flag in CCR
SEI	Set Interrupt Mask Bit in CCR
SEV	Set Overflow Flag in CCR
SPI	Serial Peripheral Interface
STAA	Store Accumulator A
STAB	Store Accumulator B
STD	Store Accumulator D
STS	Store Stack Pointer
STX	Store Index Register X
STY	Store Index Register Y
SUBA	Subtract from AccA
SUBB	Subtract from AccB
SUBD	Subtract from AccD
TAB	Transfer AccA to AccB
TAP	Transfer AccA to CCR
TBA	Transfer AccB to AccA
TPA	Transfer CCR to AccA
TST	Compare Zero to Byte in Memory
TSTA	Compare Zero to AccA
TSTB	Compare Zero to AccB
TSX	Transfer Stack Pointer to Index Register X

TSY	Transfer Stack Pointer to Index Register Y
TXS	Transfer Index Register X to Stack Pointer
TYS	Transfer Index Register Y to Stack Pointer
UI	User Interface
XGDX	Exchange AccD and Index Register X
XGDY	Exchange AccD and Index Register Y

glossary

1's complement The inversion of all the bits in a binary word (logical NOT).

2's complement A method of representing signed binary data.

Absolute Addressing The effective address of the operand is stored in memory following the instruction opcode. The HC11 has two modes that use the absolute addressing technique, the extended (EXT) and direct (DIR) modes. EXT uses a 16-bit address that occupies the two bytes following the opcode in memory. DIR uses a 16-bit address in which the upper byte is always \$00 and the lower byte occupies a single byte following the opcode in memory.

Accumulator Special registers that are directly linked to the ALU to assist with the arithmetic and logical operations. The results of arithmetic operations are stored in an accumulator.

Accumulator A or B HC11 8-bit accumulators. Used as the primary data-processing registers.

Accumulator D HC11 16-bit accumulator. It is the combination of the A and B accumulators. It is not a separate hardware register. Allows for a limited set of 16-bit arithmetic operations.

Address A unique identifier of a memory location. Also called a pointer.

Address Bus Responsible for the transfer of addresses from the processor to memory or from the processor to I/O.

Address Register General-purpose registers that typically contain an address. They are used specifically by the instructions to address memory.

Addressing Mode A method of addressing or accessing memory.

Analog Continuous signal that is defined for all levels within the range, as well as at all points in time.

Analog Data Data that is continuous in nature.

Arithmetic Logic Unit (ALU) Major subsystem of a processor. It is responsible for performing simple mathematical operations like addition and subtraction, logical AND, OR and NOT operations and data shifting.

Arithmetic Shifting A method of data shifting that maintains the sign of the data. On the HC11, this is implemented only as shift right where the sign bit is retained and shifting out the LSB into the carry flag of the CCR.

ASCII A computer code used extensively for the transmission and storage of character-based data.

Assembler A piece of software that converts mnemonic instructions into machine code.

Assembly Language The lowest-level programming language, made up of special instructions called mnemonics.

Balanced Stack All bytes that are pushed to the stack by an operation are also removed when the operation is complete.

Balanced Use The process of pulling every byte from the stack that is pushed onto the stack during an operation.

Base The number of unique symbols in a number system.

Bidirectional bus Data can travel to or from the processor and other devices.

Binary Base 2. A number system that consists of two symbols.

Binary Coded Decimal (BCD) A method of representing decimal digits as 4 binary numbers.

Binary Weighted Fractions A method of representing a fractional value that is less than zero. On the HC11 this is used exclusively with FDIV instructions and is a 16-bit value where the MSB has the weight of 2^{-1} and the LSB has a weight of 2^{-16}.

Bit A single binary digit.

Bootstrap Mode A special hardware mode that is designed to allow loading of permanent programs and other production-related programming tasks.

Branching A set of instructions on the HC11 that uses the relative addressing mode to change the flow of the program.

BUFFALO Bit User Fast Friendly Aid to Logical Operations. The monitor program loaded into the ROM of chips used in the HC11 EVBU development system.

Bug An undesirable behavior of a software program.

Bus A set of two or more conductors that are ganged together to form a parallel information path to and/or from the processor. All computer systems have three main busses: address, data and control.

Byte Eight bits of binary data grouped into a data word. A byte is the fundamental data unit on most computers. The HC11 processes data in one- or two-byte units.

Central Processing Unit (CPU) A term used to refer to the main processor in a system.

Channel A path of analog data to be converted to digital. The HC11 supports eight channels or eight independent analog sources.

Compiler A program that translates the high-level language to the machine-level code in one bulk operation.

Complex Statement The grouping of two or more "C" programming statements by the use of the curly brackets {}. This is sometimes called a code block.

Condition Code Register (CCR) HC11 status and control register. It contains five status flags and three processor control bits.

Control Bus Responsible for the control signals necessary to interface the devices within a computer system.

Control Unit One of three major functional blocks within a processor. It is responsible for the control of fetching and execution of the instructions.

Data Any information used by a program. Often it refers solely to the information processed and manipulated; functionally anything stored in memory is data.

Data Bus Responsible for the transfer of data between the processor and memory or the processor and I/O.

Decimal Base 10. A number system that consists of ten symbols.

Destination Address The target address of the jump or branch instruction or the address of the next instruction that will be executed after a jump or branch instruction.

Destructive A description of a write operation in that the previous data is overwritten (destroyed) by the new data.

Digital Discrete signal that is defined for only a specific set of levels within the range as well as specific points in time.

Digital Data Data that is discrete in nature.

Directive A command that tells the assembler what to do and indirectly affects the resulting machine code.

Discrete A type of data that can only represent a specific set of values (i.e., 1 and 0).

Dynamic RAM RAM that uses a capacitor as the main storage device and must be refreshed to maintain data.

EEPROM Electrically Erasable Programmable ROM. It can be field programmed as well as electrically erased and then reprogrammed.

Effective Address The address of the operand in memory.

EPROM Erasable Programmable ROM. It can be field programmed as well as erased via ultraviolet light and then reprogrammed.

Event Signal transition, high to low or low to high.

Execute Cycle All machine cycles other than the fetch cycle necessary to complete the execution of an instruction.

Expanded Mode An alternative normal operating mode on the HC11. It allows for off-chip memory connection. In this mode, the PORTB and Port C pins are converted to a multiplexed address and data bus. In addition, the strobe A and strobe B control lines become the address strobe and read/write control lines.

External Reset A reset caused by an external stimulus.

Fetch Cycle The process of reading an instruction opcode from memory and decoding it in the instruction decoder. It is always the first instruction of a sequence of machine cycles necessary to complete the processing of the instruction.

Finite Loop A loop that continues until or while some criterion is true.

Flowchart A graphical diagram used to show the function and flow of a program. It is a representation of the processes, the input and output of data, decisions and flow of the program.

Global Interrupt Control The ability through a single bit to activate or deactivate the interrupt function for an entire class of interrupts.

Handshaking A method of data transfer that uses control signals between the computer and the peripheral device.

Hexadecimal Base 16. A number system that consists of 16 symbols.

Hi-byte First The high byte of a 16-bit data word or address will be stored in the first location, followed in memory by the low byte.

High-level languages Programming languages that are more like a spoken language. They have syntax similar to sentence structure, making them even easier to read. Single keywords can cause many machine-level instructions to be executed.

Immediate Addressing The operand immediately follows the instruction opcode in memory. The effective address of the operand is the address of the memory location immediately following the opcode in memory. The HC11 immediate mode (IMM) uses this immediate addressing technique.

Implied Addressing No address is necessary because the location of the data is implied by the instruction. The HC11 has the inherent mode, which is a direct application of the implied addressing concept.

Indeterminate Data can be any value. Data in a RAM is indeterminate after power-up. Many bits in the HC11 processor registers are indeterminate after power-up.

Index Register X or Y 16-bit address registers used by the HC11 index addressing mode instructions. They can also be used as general-purpose, 16-bit data registers.

Indexed Addressing Uses a base address plus an index (address offset) to determine the effective address of the operand. The HC11 has two index addressing modes, INDX and INDY. Each uses the address from one of the two index registers (X or Y) as the base address. The address offset is always located in the operand field of the instruction.

Indirect Addressing Uses an address to point to the t4effective address of the operand. The address that follows the instruction opcode in memory is the address of where the actual address of the operand is stored in memory. The HC11 does not support indirect addressing.

Infinite Loop A loop that continues unconditionally, that is, it never stops.

Input/Output (I/O) A term that refers to any subsystem that has the responsibility of receiving data for the processor (input) or sending data out from the processor (output).

Instruction Decoder The main section of the processor control block. It has the job of decoding the instruction opcode from the instruction register and controlling the execution of the instruction.

Instruction Pointer Another name for the Program Counter register.

Instruction Register A special register that always contains the opcode for the current instruction.

Instructions Commands that control what the processor does.

Internal Reset A reset caused by an internal stimulus.

Interpreter A program that translates the high-level language to the machine-level code in one line or command at a time.

Interrupt An internal or external stimulus that requests immediate attention from the processor.

Interrupt Request The stimulus that signals the processor of a pending interrupt.

Interrupt Service Routine A set of instructions that gets executed each time an interrupt is serviced.

Interrupt Vector The address of the first instruction of the interrupt service routine. It is stored at the vector address in the vector table.

Jump An instruction that causes an unconditional change in the program flow.

Jump Instructions Are used to change the flow of the program by using an absolute 16-bit address rather than the relative addressing mode used by branch instructions.

Jump Table A table of jump instructions that jump to the starting addresses of internal subroutines. The jump instructions are located in the same ROM memory locations for all versions of BUFFALO, regardless of the actual locations of the subroutines in memory.

Label A name of a byte or data or an address used in a program.

Least Significant Bit (LSB) The lowest-order binary bit within a data word. This bit has the lowest binary weighting (i.e., for %00000001, the 1 is the LSB). It is always best to write the LSB on the right if writing data left to right or on the bottom if writing data top to bottom.

Listing File A file that contains the combination of the original source code and the corresponding assembled machine code.

Local Interrupt Control A control bit that is specific to a single interrupt function.

Logical Shifting A method of data shifting that ignores the sign of the data. On the HC11, this is implemented by shifting in a zero and shifting out one bit into the carry flag of the CCR.

Loop Any set of instructions that is repeated within a program.

Machine Code Multibit binary code that tells the computer the specific tasks to perform. It consist of opcodes, operands, masks, offsets and addresses.

Machine Cycle A short set of steps that are performed during a single processor clock cycle.

Mask An 8-bit word that designates which bits within the operand will be considered as part of the instruction. 1's in the mask indicate bit positions under consideration, and 0's indicate bit positions that are ignored.

Maskable Interrupt An interrupt that can be controlled (enabled or inhibited) by the user at anytime.

Master/Slave A configuration that allows multiple devices to be connected together on a common medium or bus. A single device in the group acts as the master and controls the processes. All other devices in the group act as slaves taking their commands from the master.

Memory A term that refers to any component that stores data for the processor. Memory can have many forms. There are semiconductor memories, magnetic memories and optical memories.

Memory Address An *n*-bit binary number that the processor uses to select a specific memory location. The HC11 uses a 16-bit address and can address 2^{16} or 65,536 unique memory locations.

Memory Address Register (MAR) A special address register that is linked to the program counter. The job of the MAR is to contain the address of the current memory location that is being addressed.

Memory Data Register (MDR) A special data register that resides between the data bus and the various processor registers.

Memory Map The layout of each memory block within the entire address space.

Mnemonics Special abbreviations (English-like words) that are the keywords of assembly-level programming languages. Each mnemonic correlates to a single machine code instruction.

Monitor Program A piece of software that controls the functions of a system and provides a user interface so that users can interact with the system. The HC11 EVBU uses the BUFFALO monitor program.

Most Significant Bit (MSB) The highest-order binary bit within a data word. This bit has the highest binary weighting (i.e., for %10000000, the 1 is the MSB). It is always best to write the MSB on the left if writing data left to right or on the top if writing data top to bottom.

Negation The process of changing the sign of a number.

Nibble A 4-bit group of binary bits. A half-byte.

Nondestructive The data in the memory or register location is not affected by the operation. Read operations are nondestructive.

Nonmaskable An interrupt that cannot be controlled (enabled or inhibited) by the user except in special circumstances.

Nonreturn-to-zero (NRZ) A data transmission format where a 1 is a positive voltage and a 0 is negative voltage. The voltage level is maintained for the entire bit duration and does not return to the 0 state.

Nonvolatile A characteristic of a memory device. Data will be retained when power is lost.

Number Line The ordered set of symbols that makes up a number system.

Number System An ordered set of symbols where the number of symbols in the set is called the base or radix.

Object Code The actual machine-level code processed by the computer (see machine code).

Offset A value that is added to a base address to reference another memory location.

Opcode Operational Code. An opcode is a multibit code that identifies an instruction. Each opcode contains specific information about the instruction to be executed, as well as how to execute it.

Operand Data that is operated upon by the instruction.

Operand Field The group of bytes following the opcode in memory that are necessary to complete the instruction.

Parallel Multiple bits at a time.

Parallel Port A port that allows transfer data in parallel (multiple bits at a time).

Pass by Reference A method of passing data to a subroutine that allows the data to be changed. During the execution of the subroutine, the data is accessed via a reference (memory address or register), processed and returned to the reference location before returning to the calling program.

Pass by Value A method of passing data to a subroutine that does not allow the data to be changed. When the routine is completed, the data is discarded and not returned to the calling program.

Pointer Another name for a memory address.

Port An I/O connection that allows the movement of data between the computer and an I/O device.

Prebyte An additional opcode that precedes the instruction opcode in memory.

Processor The intelligent device at the center of the machine. It has the responsibility to execute instructions. It controls and manages the activities of the entire machine.

Processor Registers A set of registers needed to perform the instruction execution. The registers are used to temporarily store data and memory pointers, as well as to contain status and control information.

Program A detailed set of steps that must be followed to complete a task. A computer program is a set of instructions to complete a task.

Program Counter A processor register that keeps track of the address of the next instruction to be executed. It is also called the Instruction Pointer.

Programmers Model A name for the set of processor registers on the HC11 (i.e., A, B, D, X, Y, etc.)

Programming Language A set of English-like words that are used to create a source code file of instructions for the computer.

PROM Programmable ROM. It can be field programmed one time (write-once).

Radix *See Base.*

RAM Random Access Memory. A read/write general-purpose memory. It is volatile semiconductor memory.

Random Access The ability to address memory in any order rather than in a sequential order.

Range The difference between the high reference voltage and the low reference voltage.

Read Operation The process of accessing data in memory and moving the data to the processor. Data in memory is not affected.

Relative Address A signed 8-bit number that indicates the number of bytes to branch forward or backward in memory.

Relative Addressing Used to change the flow of the program so that the instructions do not have to be laid out in a sequential manner. The address of the next instruction is calculated relative to the current position in memory. The HC11 has one relative addressing mode that is used only by branching instructions.

Reset The feature of a microcomputing system that is used to set the initial conditions within the system and begin executing instructions from a predetermined address.

Resolution The ratio of the step voltage over the range. It is usually shown as a percentage.

Return Address The address of the instruction that will be executed immediately following the RTS or RTI instructions.

ROM Read Only Memory. ROM is used for permanent storage of programs and data. It is a nonvolatile semiconductor memory. It is programmed during the manufacturing process.

Rotating Similar to shifting; however, the data shifted in comes from the carry flag of the CCR.

Scratch Pad An area of RAM used as a temporary data storage block.

Sensor A device that accomplishes a conversion of some physical measurement to an electrical form (i.e., temperature to volts).

Serial One bit at a time.

Serial Port A port that allows transfer of data as a serial data stream (one bit at a time).

Sign Bit The MSB of a signed value represented in the 2's complement form.

Sign Extension The process of filling in each bit to the left of a value with the sign bit to maintain the sign of a number while representing the number in various bit lengths, (i.e., the 4-bit binary value 1001 would be sign extended to the 8-bit value 11111001 because the sign bit of 1001 is 1).

Single Chip Mode The normal operating mode of the HC11. As the name implies, the single chip mode requires that all software needed to control the processor is contained in the on-chip memories. External address and data busses are not available.

Source Code A program in its original form prior to compilation and assembly. It is written in a programming language.

Special Test Mode A special hardware mode intended primarily for testing during the chip production process.

Stack A sequential block of memory that is configured as a LIFO.

Stack Pointer A processor register that contains the address pointer that indicates the next available memory location on the stack.

Static RAM RAM that uses a register as the main storage device.

Status Flags A set of bits in the condition code register and other status registers throughout the HC11 system that are used to indicate the status or condition of the processor.

Status Register A group of bits that indicate the status of the last operation (processor status). The most common status bits are sign flags, carry/borrow flags, zero flags, and overflow flags.

Step A small piece of the range that is associated with a single digital code.

Step Voltage VSTEP. The voltage range of a single step. Also called Step Size.

Subprocessors Processors in a system that are subordinate to the CPU.

Subroutine A self-contained function or subprogram terminated by the RTS instruction.

Subroutine Overhead The time it takes to call and return from a subroutine (JSR/BSR plus RTS).

Transducer *See Sensor.*

Unidirectional bus One-way bus. Data moves only one direction.

Until A conditional loop that repeats until some condition is true.

User Interface (UI) A feature of the HC11 monitor program that allows a user of the EVBU to enter some simple commands to access memory, run programs and monitor the function of the HC11. The UI also displays the results of operations.

Vector *See Interrupt Vector.*

Vector Address The address of the vector in the vector table.

Vector Jump Table A set of jump instructions in RAM that are used as the first instruction of the interrupt service routines. Their location in RAM allows the users of the HC11 in conjunction with the BUFFALO monitor program to change the location of interrupt service routines in memory.

Vector Table The group of all the system interrupt vectors. On the HC11 the vector table is located in ROM.

Volatile A characteristic of a memory device. Data will be lost when power is lost.

While A conditional loop that repeats while some condition is true.

Word A group of any number of binary bits. A word can be 4 bits, 5 bits, 8 bits, 16 bits or 32 bits, and so on.

Write Operation The processing of storing data in memory.

Zero Filling The process of filling in each bit to the left of a value with zeros to maintain a fixed number of bits in each data word. (i.e., the 4-bit binary value 1001 would be zero filled to the 8-bit value 00001001).